Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Triennale in Informatica

Studio ed Implementazione di Metodi per la Migrazione di Processi tramite DMTCP

Tesi di Laurea in Progetto di Sistemi Virtuali

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Federico Barocci

Sessione II
Anno Accademico 2012-2013

Dedicato alla mia famiglia,
ai miei amici piú cari
e tutti coloro che rendono la vita
un'entusiasmante avventura.

Introduzione

Quando si utilizza un programma si è soliti assumere che la sua esecuzione riguardi la macchina in cui questo viene eseguito, dal momento dell'avvio fino alla terminazione. Questo è quanto avviene usualmente, in particolar modo prendendo in esame i Sistemi Operativi convenzionali.

Il problema di questa gestione sorge nel momento in cui si ha la necessità di spostare l'esecuzione di un'applicazione durante la sua esecuzione. Le ragioni che nutrono questo bisogno per un utente sono molteplici, come la possibilità di riavviare o spegnere una macchina senza l'obbligo di arrestare l'esecuzione delle applicazioni in uso, la possibilità di cambiare macchina senza chiudere e rieseguire programmi aperti, la possibilità di poter utilizzare in maniera continuativa un'applicazione in due differenti località senza coinvolgere dispositivi portatili.

I vantaggi derivanti dalla migrazione di processi sono numerosi ed oltre a fornire situazioni di comodità per gli utenti possono fornire degli strumenti utilizzabili dagli amministratori di sistema, quali meccanismi di tolleranza ai guasti, bilanciamento del carico di lavoro ed una modalità di realizzazione di applicazioni che possono fare uso di risorse di calcolo fornite attraverso una rete distribuita di calcolatori.

Lo scopo iniziale di questo elaborato è analizzare quali sono le caratteristiche della migrazione, quali elementi vincolano un processo all'ambiente di esecuzione, valutando modalità e difficoltà legate al suo spostamento e come possono essere gestite. Il secondo obiettivo è quello di presentare alcune possibili soluzioni e valutarne gli aspetti implementativi. Infine viene presentato

un strumento sviluppato come prova a sostegno della teoria ed in grado di svolgere la migrazione di processi senza necessità di modificare il Sistema Operativo, ULPM (*User Level Process Migration*). Attraverso la tecnica di *Checkpoint/Restart* si permette di congelare l'esecuzione di un programma in esecuzione e contestualmente di riattivarlo in un'altra macchina connessa in rete. L'implementazione di ULPM fa utilizzo di uno strumento dal consolidato sviluppo, DMTCP (*Distribuited MultiThread CheckPointing*), il quale permette di estrarre e ripristinare lo stato di esecuzione di un processo.

La seguente sinossi presenta una panoramica di questo documento:

Il primo capitolo affronta l'argomento della migrazione di processi nei suoi aspetti generali attraverso motivazioni, caratteristiche e problematiche;

Il secondo capitolo introduce delle considerazioni più approfondite in merito gli aspetti teorici, prendendo in esame le modalità di migrazione e le tecniche di implementazione;

Il terzo capitolo focalizza nelle tecniche di *Checkpoint/Restart* e distingue quali aspetti facilitano e quali complicano il loro impiego per l'operazione di migrazione;

Il quarto capitolo è dedicato all'analisi del funzionamento di DMTCP, lo strumento di *Checkpoint/Restart* utilizzato per la creazione di ULPM;

Il quinto capitolo è dedicato alla presentazione dello strumento di migrazione ULPM, mostrando gli elementi implementati ed i risultati conseguiti.

Indice

Elenco delle figure

Capitolo 1

Migrazioni di processi

1.1 Introduzione

Nei Sistemi Operativi, un *processo* rappresenta una istanza di un programma in esecuzione. Per *migrazione di processo* si intende l'atto di trasferire un processo durante la sua esecuzione verso un altro sistema connesso in rete. Questa operazione deve anche garantire continuità di elaborazione e trasparenza rispetto l'ambiente di esecuzione. Garantire *continuità* significa che il processo deve poter continuare la propria esecuzione dall'identico stato in cui si trova nel momento in cui sopraggiunge la richiesta di migrazione, mentre garantire *trasparenza* significa che il processo non deve percepire alcuna differenza nell'ambiente tra l'istante prima e quello dopo la migrazione, esattamente come se questa non fosse mai avvenuta. Questa procedura permette una serie di vantaggi tra cui la distribuzione del carico di lavoro attraverso una rete di elaboratori, la semplificazione dell'amministrazione del sistema, la possibilità per il processo di accedere a dati locali ed una modalità di tolleranza ai guasti che potrebbero interrompere l'erogazione di un servizio. Tuttavia un processo è fortemente legato al sistema in cui è in esecuzione e poiché non è un compito dei Sistemi Operativi tradizionali garantire la proprietà di trasparenza tra più calcolatori, questa tecnica viene principalmente utilizzata nei Sistemi Distribuiti. Anche le differenze *hardware* e *software* e le

differenti disponibilità di risorse presenti tra sistemi posti in comunicazione inducono a delle complicazioni. Viste queste difficoltà si potrebbe ipotizzare di garantire il servizio solo per reti di macchine con *hardware* e *software* omogenei. Sebbene questo approccio fornisca una semplificazione di alcuni aspetti si può però notare che altre problematiche restano invariate, ad esempio tutti gli elementi che vengono gestiti dal Sistema Operativo e che caratterizzano il processo nel momento dell'esecuzione: *process id*, *file descriptor*, connessioni di rete, terminali, relazioni padre-figlio e comunicazione con altri processi, ... Ricerche e studi sulla migrazione di processi svolti nel corso degli ultimi decenni [4] hanno evidenziato possibili strategie e tecniche implementative per far fronte alle difficoltà pratiche riscontrate nelle macchine reali, oppure proponendo soluzioni alternative attraverso l'uso di macchine virtuali o linguaggi di programmazione (interpretati o semi-compilati) dotati di strumenti ideati appositamente per questi scopi.

Nei capitoli successivi verrà invece analizzata nel dettaglio una tecnica che permette di migrare un processo senza richiedere considerevoli modifiche a livello di sistema, cioè la tecnica del *Checkpoint/Restore*. Questa tecnica può avere delle limitazioni nella tipologia di processi migrabili, quindi verranno studiate delle soluzioni possibili per superare o contenere queste difficoltà.

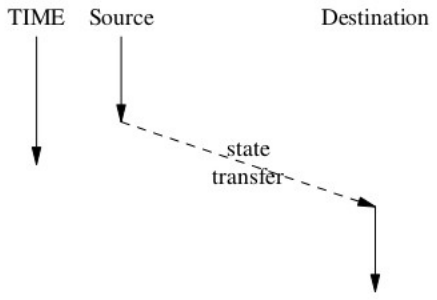

Figura 1.1: La migrazione di un processo [4]

1.2 Contesto di analisi

Analizzando il tema della migrazione di processi si fa riferimento prevalentemente al contesto delle reti di calcolatori. Ogni macchina viene chiamata *nodo* della rete e, considerando alcuni degli aspetti relativi alla migrazione dei processi, vengono evidenziati alcuni nodi caratteristici:

Nodo sorgente è la macchina dalla quale si intende migrare un processo;

Nodo destinatario è la macchina verso la quale si intende migrare un processo;

Nodo sovraccarico è la macchina avente una quantità di processi attivi tale da non garantire prestazioni ottimali, è una possibile candidata a diventare sorgente per la migrazione di processi;

Nodo scarico è la macchina avente pochi processi attivi e può fornire maggiori potenze di calcolo, è una possibile candidata a diventare destinataria per la migrazione di processi.

Al fine di affrontare la tematica con riferimento a casi reali quanto seguirà nei successivi paragrafi farà riferimento principalmente alle distribuzioni dei Sistemi Operativi GNU/Linux ed ai sistemi UNIX e derivati.

1.3 Tipologie di migrazione

Rispetto lo stato di esecuzione del processo, al variare del momento in cui avviene la procedura di migrazione si possono distinguere due diverse classificazioni:

Senza pre-rilascio La migrazione del processo avviene prima che questo sia avviato nel sistema. Questa modalità è relativamente semplice in quanto non vi sono dati allocati da recuperare e non è necessario ricostruire tutti gli elementi che lo legano al sistema originario (come

socket, *file descriptor*, *process id*, relazioni tra processi, meccanismi di comunicazione ed altro ancora).

Con pre-rilascio La migrazione avviene dopo che il processo è stato avviato nel sistema. Questa modalità è relativamente complessa in quanto richiede che tutti i dati allocati siano recuperati, salvati, spostati verso il sistema destinatario e lo stato del processo deve essere ricostruito esattamente come nel momento in cui di questo è stato richiesto il rilascio, ricostruendo tutti i canali di comunicazione con altri processi e tutte le connessioni di rete.

1.4 Motivazioni

La migrazione dei processi acquista una particolare utilità in cluster di computer e, più in generale, nei Sistemi Distribuiti ed in tutte quelle situazioni in cui si richiede che più elaboratori uniscano le proprie potenze di calcolo per operare in parallelo. Affinché questo parallelismo di calcolo sia effettivo ci si aspetta che i sistemi siano capaci di coordinarsi ed eventualmente bilanciare il carico di lavoro tra le macchine disponibili [6].

Analizzando i vari aspetti che caratterizzano i possibili scenari di utilizzo, si possono evidenziare alcune situazioni in cui la migrazione dei processi produce dei vantaggi misurabili.

Aumento delle prestazioni

Se un sistema è sovraccarico si può considerare la possibilità di spostare uno o più processi che richiedono risorse verso un'altra macchina libera. La politica di gestione implementata deve scegliere quale processo migrare, in quale momento e in quale macchina migrarlo.

Tolleranza ai guasti

Nel caso in cui si verifichi un malfunzionamento o un danno parziale (non invalidante) ad un nodo della rete tale da richiedere manutenzione, come l'arresto o il riavvio del sistema, può essere preferibile la migrazione dei processi per i quali si richiede continuità di esecuzione e/o processi a lungo termine. Questo permetterebbe loro di proseguire nelle operazioni di calcolo o nell'erogazione di un servizio.

Spostamento di una sessione di lavoro

Se un utente utilizza un'applicazione per lungo tempo potrebbe avere la necessità di continuare l'esecuzione della sessione corrente presso un'altra macchina mantenendo tutte le informazioni temporanee presenti nella singola istanza. A titolo esemplificativo supponiamo che un utente desideri spostare l'esecuzione di un programma dal proprio desktop nel suo computer portatile mantenendo la continuità di esecuzione, come le schede aperte in un *browser web*, i file di testo aperti in un *word processor*, il punto di esecuzione di un video o di una traccia audio in un *player* multimediale.

Accesso a dati locali

Un processo potrebbe richiedere alcune risorse non disponibili nel sistema in cui è in esecuzione ma necessarie per proseguire in una particolare elaborazione. A meno di disporre di un qualche strumento atto all'accesso di queste risorse attraverso la rete, l'esecuzione del compito potrebbe essere compromessa. In tal caso occorre valutare ogni situazione e considerare la possibilità di migrare il processo direttamente nel nodo in cui si richiedono le risorse ed operare in locale invece di instaurare comunicazioni di rete che introdurrebbero latenze e complicazioni a livello applicazione.

Efficienza di accesso alle risorse

Qualora un processo necessiti di comunicare con un altro attraverso la rete, questi vengono connessi tramite un canale di comunicazione in cui possono scambiare informazioni (dati, messaggi ed istruzioni) incapsulate in pacchetti. Latenze nella rete e continuo scambio di messaggi possono rendere la comunicazione remota meno efficiente che se eseguita localmente. Una possibile soluzione è migrare interamente il processo al fine di trasformare la comunicazione remota nella più efficiente comunicazione locale tra processi.

1.4.1 Bilanciamento del carico di lavoro

Con "*bilanciamento del carico di lavoro*" si indica una particolare configurazione in un sistema distribuito o in un cluster atta a garantire equilibrio nell'allocazione delle risorse disponibili e conseguente aumento delle prestazioni complessive del sistema. Nel caso in cui un nodo della rete si sovraccarichi di processi, questo può avere la necessità di spostare l'esecuzione di una parte dei propri lavori in calcolatori non occupati.

Questa operazione richiede generalmente tre informazioni che è compito della politica di bilanciamento [5] individuare e gestire:

1. Selezionare *quale* processo migrare;
2. Selezionare *quando* migrare il processo;
3. Selezionare *dove* migrare il processo.

Questa funzionalità viene spesso fornita nei Sistemi Distribuiti con varianti implementazioni di algoritmi per la selezione del nodo destinatario e del processo da migrare.

1.4.2 Tolleranza ai guasti

Con "*tolleranza ai guasti*" si intende una tipica funzionalità offerta in un cluster o in Sistemi Distribuiti con lo scopo di garantire disponibilità

continuata del servizio anche in caso di guasto ad uno dei nodi della rete. Solitamente si cerca di fornire affidabilità del servizio tramite la presenza di ridondanza di apparati.

Nel caso in cui si verifichi la necessità di scollegare o riavviare una macchina avente in esecuzione un particolare processo, la cui interruzione potrebbe creare disservizi, si potrebbe rischiare di perdere informazioni o dati importanti. Invece, con la possibilità di migrare il processo verso una macchina secondaria non si incorrerebbe in questi rischi e l'erogazione del servizio potrebbe continuare in maniera del tutto fluida.

1.4.3 Sistema ad alte prestazioni

Con "*sistema ad alte prestazioni*" ci si riferisce ad una tipologia di configurazione di un Sistema Distribuito o di un cluster tale da garantire elevate prestazioni di calcolo parallelo. Lo scopo è di permettere ad un processo di utilizzare tutte le risorse disponibili distribuite tra tutte le macchine, così da massimizzare la potenza di calcolo. I processi eseguiti solitamente devono essere realizzati appositamente per questo scopo, utilizzando meccanismi di comunicazione tra processi [8]. In questo contesto dovrebbe essere compito del Sistema Distribuito occuparsi della migrazione dei processi verso macchine non occupate e garantire la comunicazione dei dati tra processi come se questi fossero eseguiti in una singola macchina.

1.4.4 Accesso a dati locali

In un contesto distribuito non sempre tutte le risorse sono disponibili contemporaneamente in tutte le macchine. Alcuni processi potrebbero avere la necessità di accedere a strutture dati, come *file* o *database*, non disponibili sulla macchina locale. Se non si dispone di una qualche forma di interfaccia *software* predisposta per potervi accedere attraverso la rete, occorre allora individuare una qualche soluzione per risolvere il problema.

In questo caso alcune possibilità potrebbero essere:

1. accesso remoto al terminale (ad esempio ssh) per eseguire localmente il processo che richiede l'accesso ai dati;

2. realizzazione di una applicazione con apposita interfaccia di rete per permettere l'accesso e la comunicazione remota (tipica soluzione adottata nel web attraverso la predisposizione di API, *Application Program Interface*);

3. permettere al processo di migrare nella macchina remota, operare sui dati o raccogliere le informazioni, quindi migrare nuovamente nella macchina sorgente per completare l'elaborazione.

Analizziamo queste possibili situazioni:

Nel primo caso, l'accesso remoto alla macchina che dispone dei dati, si otterrebbe l'elaborazione all'interno della macchina in cui viene eseguito il processo, pertanto verrebbero utilizzate le risorse di calcolo della macchina remota per tutto il tempo di esecuzione. Se l'elaborazione è particolarmente lunga e più utenti devono compiere contemporaneamente questa tipologia di lavoro, la soluzione potrebbe non essere ottimale, anzi, si rischierebbe di sovraccaricare la macchina con conseguente perdita di prestazioni. Si potrebbe allora decidere di procedere con il trasferimento locale dei dati. Purtroppo questa soluzione non è generalizzabile per tutti i casi e quindi non sempre possibile, come nel caso di un *database*. Prendiamo ad esempio l'inserimento di un record in un file: una soluzione potrebbe essere copiare localmente il file, inserire il record e sostituire il file remoto con quello appena aggiornato. Però, se due processi distinti eseguono contemporaneamente la stessa operazione, si verificherebbe una situazione di *race condition* da gestire al fine di evitare la perdita (causa sovrascrittura) di uno dei due aggiornamenti al file. Si potrebbe allora prevedere un meccanismo di blocco che permetta di accedere al file ad un processo alla volta. In questo caso, ricordando il contesto di rete, potrebbero verificarsi

problemi di comunicazione durante l'operazione di aggiornamento che, se non gestiti correttamente, potrebbero impedire l'accesso alla risorsa anche ad altri processi. Si può dedurre che tra la copia locale dei dati e l'esecuzione remota del processo non vi sono elementi che giustificano una scelta come nettamente migliore rispetto all'altra, entrambe presentano degli aspetti positivi ma anche problematiche, da prevedere e gestire nel modo corretto.

Nel secondo caso, la predisposizione di interfaccia API di comunicazione, occorre realizzare un apposito programma con lo scopo di interfacciare i dati con i processi remoti. Questa soluzione, pur gravando sulla complessità architetturale del *software*, potrebbe essere conveniente in termini di protezione ed astrazione dei dati. Il processo remoto, non potendo accedere direttamente ai dati, non dovrebbe essere in grado di arrecare danni e pertanto è una delle soluzioni più adottate, soprattutto nei casi in cui si vuole permettere di accedere ai dati anche ai processi provenienti da sorgenti non sicure. Di contro vi è una limitazione nelle operazioni possibili, ovvero un processo remoto può operare sui dati solo con gli strumenti e con i canali di comunicazione previsti in fase di realizzazione delle API. Inoltre, sebbene un formato di rappresentazione dati condiviso permetta ad applicazioni distinte di comunicare ed interagire, si potrebbe rivelare necessario adottare meccanismi di conversione tra le differenti rappresentazioni.

Il terzo caso, la migrazione del processo, può invece semplificare la complessità del codice dell'applicazione e l'esecuzione del processo potrebbe procedere nella macchina originaria dopo raccolti i dati di cui ha bisogno. Questa soluzione è da prendere in considerazione solo in caso di processi provenienti da sorgenti sicure perché, di contro, la massima libertà di esecuzione del codice produce anche una equivalente perdita di sicurezza. Al fine di contenere i rischi legati all'esecuzione di codice potenzialmente atto ad arrecare danni, potrebbe rivelarsi necessario

limitare le possibili operazioni del processo migrato, ad esempio eseguire codice in un ambiente isolato rispetto il resto del sistema, come una Macchina Virtuale o una *sandbox*.

1.5 Caratteristiche implementative

L'implementazione del servizio di migrazione può avvenire sia a Livello Kernel che a Livello Utente. Entrambe le possibilità offrono dei vantaggi e delle difficoltà, a seconda degli obiettivi prefissi si possono valutare scelte verso una o l'altra modalità di realizzazione.

La scelta di implementare a **Livello Kernel** è solitamente più complessa in quanto richiede numerose modifiche ed integrazioni di funzionalità all'interno del Sistema Operativo. Ad ogni modo, a fronte della complessità, il risultato risulta nettamente migliore in termini di prestazioni ed affidabilità [4].

L'implementazione a **Livello Utente** non è necessariamente più semplice: la natura stessa del processo, fortemente legato all'ambiente di esecuzione, può rendere questa operazione ancora più complessa e meno affidabile. Il vantaggio considerevole di questa soluzione è l'assenza del requisito di attuare modifiche nel Kernel delle macchine a disposizione, operazione non sempre possibile (come nel caso di Sistemi Operativi proprietari).

Una possibilità intermedia riguarda l'impiego di **moduli ausiliari** che non richiedano l'intera modifica del Kernel ma solo l'estensione di alcune funzionalità che applicazioni a Livello Utente potrebbero utilizzare per eseguire la migrazione del processo. Questo compromesso può essere tuttavia limitativo in quanto un modulo per il Kernel potrebbe non essere sufficiente.

Un'altra possibile soluzione intermedia richiede l'uso di **Macchine Virtuali**. Questa scelta permette di supportare macchine eterogenee astraendo l'architettura sottostante, semplificando alcuni aspetti implementativi. D'altra parte le Macchine Virtuali possono incorrere in problemi di prestazioni che possono essere visti come limitativi, dovuti ad un sovraccarico di interfac-

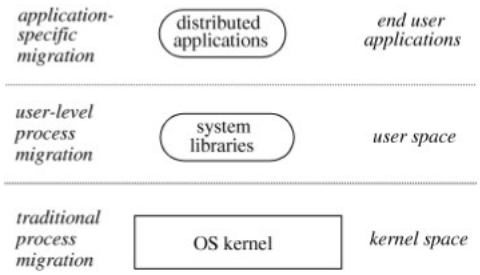

Figura 1.2: Livelli di implementazione del servizio di migrazione [4]

ce intermedie rispetto la macchina reale. Occorre però ricordare che queste limitazioni possono essere contenute o aggirate attraverso degli accorgimenti implementativi [16] e delle semplificazioni.

Altre possibilità riguardano l'impiego di sistemi intermedi ***Middleware*** o di *framework* per lo sviluppo di Sistemi Distribuiti. Queste due soluzioni sono simili per le modalità di migrazione. Entrambe costituiscono un'interfaccia tra l'applicazione ed il Sistema Operativo con lo scopo di garantire, dal punto di vista del processo, una rappresentazione unica ed omogenea di più sistemi connessi in rete. Entrambe hanno lo svantaggio di richiedere alle applicazioni che ne vogliono fare uso di essere sviluppate o compilate appositamente. La differenza principale tra queste due soluzioni è che il *Middleware* si preoccupa di fornire astrazione di risorse eterogenee per semplificare lo sviluppo di applicazioni distribuite, invece un *framework* per Sistemi Distribuiti costituisce un unico ambiente condiviso in rete da più applicazioni che interagiscono attraverso operazioni di calcolo parallelo.

1.6 Problematiche

Le difficoltà principali nel fornire il servizio di migrazione di processi sono riassumibili in alcuni punti principali.

Allocazione e Scheduling dei processi

Uno degli elementi più importanti da considerare è la scelta del nodo verso cui migrare il processo. La politica adottata per il bilanciamento del carico di lavoro deve effettuare delle scelte basandosi su diversi fattori quali la quantità delle operazioni in esecuzione nei nodi, la segnalazione di guasti, la presenza di hardware o dispositivi particolari, lo scambio di informazioni con altri processi, la presenza dei dati e la quantità degli accessi effettuati a questi. Occorre ricordare che una delle motivazioni che giustificano la migrazione del processo è infatti la possibilità di ottenere migliori prestazioni, pertanto se la scelta del nodo di destinazione avviene in maniera errata si possono al contrario ottenere peggioramenti dovuti anche ad un sovraccarico delle comunicazioni di rete.

Salvataggio e trasferimento dello stato del processo

Al momento della migrazione uno degli aspetti importanti da considerare è anche il tempo in cui il processo resta congelato prima di essere riavviato nel nodo remoto. Gli elementi che influiscono maggiormente nel tempo di migrazione [4] sono il recupero delle informazioni appartenenti allo stato del processo e lo spostamento di queste attraverso la rete. Esistono diverse tipologie di scelte, analizzate dettagliatamente nel capitolo seguente, che possono incidere o meno nelle prestazioni e nella complessità del servizio.

Trasparenza del File System

Nei contesti che non prevedono una visione unica e distribuita del *File System* questa problematica può essere molto delicata. Un file presente in un nodo non ha alcuna garanzia di essere ugualmente disponibile in un altro nodo, anzi potrebbe non essere presente o potrebbe essere memorizzato in un altro percorso. Una soluzione relativamente semplice prevede l'uso di *Network File System* (NFS).

Elementi assegnati dal sistema

Ogni processo dispone di informazioni che sono assegnate dal sistema per ogni sua singola istanza di esecuzione, come il *process id*. Dopo la migrazione si potrebbero verificare conflitti, ad esempio *process id* già assegnato ad un altro processo. Una possibile soluzione [4] prevede di eliminare le informazioni assegnate dal sistema originario e sostituirle con delle nuove. Tuttavia questa situazione può non essere gradita e pertanto occorre anche valutare soluzioni alternative.

Scalabilità

Un altro aspetto importante da considerare è la possibilità di aggiungere o rimuovere nodi dalla rete. Il servizio di migrazione, specialmente se implementato attraverso un Sistema Distribuito, deve essere sufficientemente flessibile da poter gestire queste eventualità.

Livello di trasparenza

Come si è potuto evidenziare in questi punti, a volte una totale trasparenza risulta molto complicata da fornire. Qualora non fosse possibile garantire tutta la trasparenza teoricamente ideale occorre domandarsi quali aspetti sono essenziali e quali invece sono trascurabili per il caso d'uso.

Capitolo 2

Tecniche di migrazione ed aspetti implementativi

2.1 Introduzione

Nel capitolo precedente si sono analizzati alcuni dei vantaggi perseguibili tramite la migrazione di processi, come la distribuzione del carico di lavoro e il contenimento dei danni derivati da malfunzionamenti. Da esperimenti fatti negli ultimi decenni [18] è stata dimostrata l'efficacia di questa tecnica, in quali tipologie di problemi è conveniente utilizzarla e quali percentuali di incremento di prestazioni dovremmo attenderci. Ulteriori esperimenti svolti [5] hanno anche dimostrato che l'impiego di questa tecnica anche nelle situazioni in cui apparentemente non si presenta di alcun vantaggio, non produce nemmeno un degrado delle prestazioni generali del sistema. A discapito dei vantaggi esistono delle difficoltà molto complesse da considerare, tali da rendere la migrazione dei processi una tecnica che in alcune situazioni può essere limitata con compromessi tra tipologie di processi supportati, prestazioni e portabilità.

I prossimi paragrafi verranno destinati ad analizzare alcune delle possibili varianti per implementare il servizio di migrazione, prendendo in esame le soluzioni più note e considerando le caratteristiche prestazionali.

2.2 Tecnica generale

Per garantire che la migrazione di processo sia una tecnica in pratica concretizzabile, alcune funzionalità devono essere garantite dal sistema o deve essere possibile implementarle:

- Possibilità di esportare/importare lo stato del processo
- Trasparenza di esecuzione del processo
- Pulizia degli elementi non migrabili

Queste caratteristiche principali sottolineano che si deve disporre di un qualche strumento che consenta di estrarre lo stato del processo dal nodo sorgente per poi poterlo importare nel nodo destinatario. A seconda dell'implementazione si potrebbe decidere di posticipare la richiesta per permettere di eseguire l'operazione nei momenti più idonei dell'elaborazione. La trasparenza deve inoltre garantire che l'esecuzione possa procedere come se non fosse avvenuta alcuna migrazione e permettendo al processo di accedere alle stesse risorse. Al fine di consentire flessibilità dello strumento, nessun meccanismo dovrebbe essere implementato preventivamente all'interno del codice del processo. Infine, considerando le situazioni in cui non tutti gli elementi del processo sono migrabili o alcune tracce del processo migrato appaiono presenti nel nodo sorgente, occorre disporre di un meccanismo di pulizia affinché questi elementi vengano rilasciati una volta conclusa la migrazione.

Queste funzionalità possono essere fornite dal Sistema Operativo, dal linguaggio di programmazione o da altri elementi dell'ambiente di esecuzione a cui il processo ha accesso.

Sebbene ogni implementazione possa differire nelle modalità di erogazione del servizio, si possono individuare [7] alcuni passaggi generali:

1. La richiesta di migrazione viene inviata al nodo destinatario, il quale si prepara a ricevere le informazioni dal nodo sorgente;

2. Il processo nel nodo sorgente viene sospeso, l'esecuzione viene temporaneamente interrotta per prevenire cambiamenti di stato interni;

3. Le comunicazioni verso il processo vengono dirottate e gli eventuali messaggi in arrivo vengono inseriti in delle code. Questa operazione viene proseguita fino al completamento della migrazione nel nodo destinatario;

4. Viene estratto lo stato del processo completo di memoria allocata, contenuto dei registri, stati di comunicazione, file aperti e tutte le informazioni rilevanti;

5. Viene creata una istanza di processo nel nodo destinatario con lo scopo di raccogliere tutte le informazioni di stato del processo remoto;

6. Lo stato del processo viene trasferito attraverso la rete dal nodo sorgente alla nuova istanza di processo nel nodo destinatario;

7. Vengono reindirizzati tutti i messaggi e ripristinati tutti i canali di comunicazione nel processo migrato. I messaggi posti in attesa vengono consegnati al processo;

8. L'esecuzione del processo viene ripristinata, permettendo ad esso di riprendere la normale esecuzione del codice. La migrazione del processo è a questo punto completata.

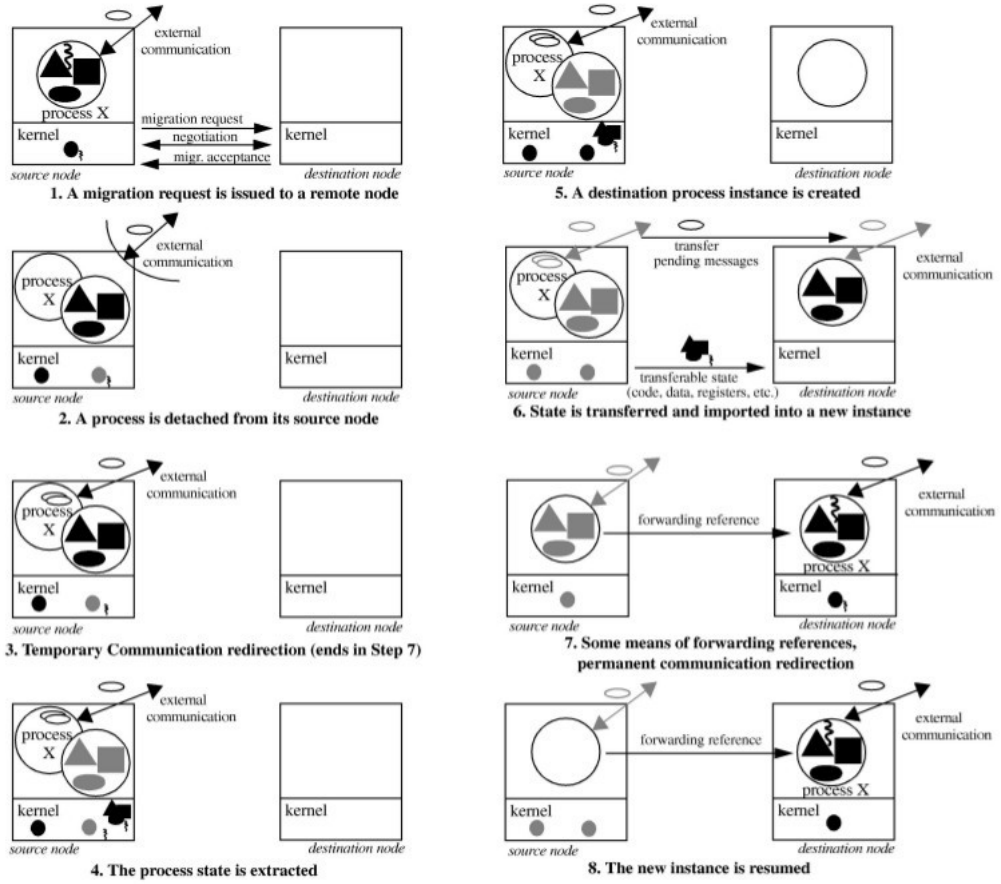

Figura 2.1: I passaggi coinvolti nella tecnica generale di migrazione [4]

2.3 Trasferimento della memoria allocata

Uno tra gli aspetti implementativi più importanti da considerare è quello del tempo necessario per completare l'operazione di migrazione, il quale viene fortemente influenzato dalla dimensione dei dati da trasferire. Lo spazio di memoria utilizzato dal processo è solitamente l'elemento di dimensione maggiore da considerare, pertanto sono state evidenziate diverse strategie [10] per cercare di minimizzare il tempo di trasferimento.

Eager (all)

Questa strategia consiste nella copia tutto lo spazio di memoria utilizzato al momento della migrazione. Viene utilizzata solitamente nella tecnica di *Checkpoint/Restart*.

Eager (dirty)

Questa strategia viene utilizzata quando il Sistema Operativo supporta il remote paging. Si tratta di una variante della strategia *Eager (all)* che trasferisce solo le pagine di memoria modificate, memorizzando le restanti in un dispositivo di memorizzazione secondaria condiviso. La richiesta di una differente pagina di memoria avviene quindi attraverso la rete e sarà compito del Sistema Operativo gestirla propriamente.

Copy-On-Reference (COR)

Questa strategia è molto simile alla *Eager (dirty)* in cui il processo può richiedere pagine di memoria sia al dispositivo di memorizzazione comune, sia al nodo sorgente. Questa tecnica riduce ulteriormente il tempo di trasferimento in quanto l'accesso alle pagine non è necessariamente vincolato all'accesso al disco, quindi al tempo necessario per accedervi. Altro vantaggio di questa tecnica è che non è necessario trasferire alcuna pagina al momento della migrazione, in quanto queste vengono recuperate dal processo nel momento in cui questo ne richiede l'utilizzo. Purtroppo questa tecnica produce una dipendenza residua verso il nodo sorgente che deve continuare a mantenere pagine di memoria allocate nella memoria principale.

Flushing

Questa tecnica si pone come soluzione alla dipendenza residua verso il nodo sorgente pur mantenendo una elevata velocità nella migrazione. Il processo può essere migrato senza alcuna pagina di memoria (come in COR) e queste vengono memorizzate nel dispositivo di memoria secondaria condivi-

so. Ogni richiesta di una differente pagina di memoria viene reindirizzata verso la memoria secondaria che di contro ha un tempo di accesso maggiore rispetto quella principale. Quindi, a vantaggio di una migrazione più veloce e dell'annullamento delle dipendenze residue verso il nodo sorgente, vi è una perdita di prestazioni del processo dopo la migrazione dovuti agli accessi al disco per recuperare le pagine di memoria.

Precopia

Questa strategia consiste nel traferire verso il nodo destinatario le pagine di memoria non in uso prima che il processo venga congelato per la migrazione. In questo modo il tempo in cui il processo resta bloccato è viene ridotto ulteriormente e una volta trasferito trova subito a disposizione la maggior parte delle pagine di memoria. L'operazione di migrazione viene pertanto rimandata fino al momento in cui non viene raggiunta una certa soglia di pagine di memoria copiate nel nodo destinatario. Le pagine restanti possono essere trasferite insieme al processo oppure successivamente alla migrazione. Anche con questa tecnica le dipendenze residue verso il nodo originario sono annullate. Per un breve periodo di tempo vi è una duplicazione dei dati allocati tra i due nodi ma questo viene considerato un inconveniente minore.

2.4 Politiche di bilanciamento del carico

Altro aspetto strettamente connesso alla migrazione dei processi è quello del bilanciamento del carico di lavoro. In un Sistema Distribuito si rivela utile disporre di uno strumento capace di identificare quali nodi sono sovraccarichi e quali sono scarichi, oltre ad identificare quali processi sono più idonei ad essere migrati rispetto agli altri.

Questo compito viene delegato ad uno strumento denominato "*scheduler distribuito*", il cui compito è quello di analizzare le informazioni di carico provenienti dai singoli nodi. Tramite queste informazioni lo scheduler dis-

tribuito deve individuare quale processi devono essere migrati, in quali nodi ed in quale momento. Il modo in cui questa scelta viene determinata prende il nome di "*politica di scheduling distribuito*" ed esistono differenti algoritmi che implementano tale la politica, quindi saranno presentati i principali e più noti.

Sender-initiated

Se un nodo della rete è sovraccarico ma deve eseguire nuovi processi oppure desidera distribuire parte del proprio carico di lavoro verso altri nodi, parte dei processi vengono migrati in rete verso un nodo non sovraccarico. In questo caso si gestisce la migrazione senza pre-rilascio nel caso dell'esecuzione di nuovi processi oppure la migrazione con pre-rilascio nel caso il nodo desideri distribuire il proprio carico di lavoro già avviato. Questa strategia è preferibile in caso di sistemi non eccessivamente carichi, con pochi nodi sovraccarichi. Solitamente si rivela conveniente per situazioni di invocazione remota (*Remote Procedure Call*, RPC) e quindi per migrazione senza pre-rilascio.

Receiver-initiated

Se un nodo della rete non ha carico di lavoro da eseguire si può offrire disponibile, segnalandolo agli altri nodi, ad eseguire processi presenti nei nodi sovraccarichi. In questo caso la strategia è ottimale per sistemi solitamente molto carichi e con pochi nodi scarichi, particolarmente adatta per la migrazione di processi con pre-rilascio, in quanto la migrazione può avvenire in qualsiasi momento dell'esecuzione del processo.

Simmetrico

Si tratta di una combinazione della politica *sender-initiated* e *receiver-initiated*, cerca di raccogliere le buone caratteristiche di entrambe le strategie. Si rivela adatta per la maggior parte delle situazioni.

Casuale

Si tratta di una variante della politica *sender-initiated* ma il nodo di destinazione viene scelto casualmente. Questa politica è particolarmente vantaggiosa quando la raccolta delle informazioni di carico dei sistemi si rivela una procedura troppo gravosa da implementare, pertanto la semplice scelta casuale produce un significativo incremento di prestazioni nello *scheduler*.

2.5 Trasparenza, Single System Image (SSI) e Sistemi Operativi Distribuiti

Garantire trasparenza nella migrazione significa che sia il processo migrato che gli altri non migrati non percepiscano alcun cambiamento nell'ambiente durante la loro esecuzione. Le comunicazioni con il processo migrato possono essere ritardate durante l'atto di migrazione ma nessun messaggio deve essere perduto. Dopo la migrazione il processo deve poter continuare a comunicare attraverso i canali precedentemente aperti. Inoltre si assume che il processo migrato sia in grado di eseguire qualsiasi System Call disponibile nel nodo di origine.

Una possibile soluzione che permette una completa forma di trasparenza è definita come *Single System Image* (SSI) la quale prevede di fornire una vista unica del sistema, anche se composto da più macchine eterogenee. Dal punto di vista delle reti possiamo definire questa soluzione come se più nodi connessi tra loro vengano considerati come un unico nodo. L'obiettivo di una SSI è principalmente focalizzato nella fornire una gestione delle risorse completamente trasparente, scalabile, performante ed in grado di supportare applicazioni utente.

Tale tipologia di soluzione viene classificata come una particolare tipologia di Sistema Distribuito, il quale è definito *"collezione di computer indipendenti che appare ai propri utenti come un singolo sistema coerente"* [11].

Una SSI può essere caratterizzata da varie proprietà [13] che concorrono alla completa trasparenza del Sistema Distribuito:

Trasparenza di posizione All'interno dell'ambiente distribuito è logico supporre che vi siano risorse, come file e periferiche, dislocate tra tutti i sistemi. Queste risorse sono usualmente identificate da dei nomi che generalmente non contengono informazioni necessarie per associare il nome della risorsa al singolo sistema in cui è posta. Inoltre, proprio in virtù della trasparenza, l'utente non dovrebbe essere in grado di determinare se una risorsa è presente nel proprio sistema locale oppure in un altro sistema connesso alla rete. Garantire trasparenza di posizione significa pertanto fornire la possibilità di accesso a tutte le risorse disponibili, indipendentemente da dove queste sono collocate.

Trasparenza di replicazione Una tecnica solitamente adottata per aumentare le tolleranze ai guasti e le prestazioni di un sistema consiste nel disporre di una ridondanza di risorse. Qualora nella SSI si decida di adottare questa proprietà, ad esempio con copia locale di file per migliorarne i tempi di accesso, occorre prevedere dei particolari accorgimenti affinché non si verifichino delle situazioni di incoerenza tra più versioni replicate della stessa risorsa.

Trasparenza di migrazione Come già discusso, trasparenza di migrazione per i processi significa che questi devono procedere nelle proprie esecuzioni indifferentemente se l'atto di migrazione avviene oppure meno. Riferendosi alle risorse questo significa invece che queste possono essere fisicamente ricollocate tra i vari sistemi ma in modo del tutto impercettibile alle applicazioni.

Trasparenza di accesso Questa proprietà si preoccupa di nasconde le differenze nella rappresentazione dei dati e nella modalità di accesso a questi. Garantire trasparenza di accesso significa fornire la possibilità di accedere a tutte le risorse disponibili, indipendentemente da come queste sono collocate.

Trasparenza di concorrenza Garantire trasparenza di concorrenza significa disporre di strumenti (semafori, *monitor*, ...) per consentire la mutua esclusione tra processi che contendono le stesse risorse ma disposti su macchine fisiche differenti.

Trasparenza di prestazioni Questo requisito richiede che il sistema distribuito sia in grado di configurarsi dinamicamente per bilanciare il carico di lavoro tra i vari singoli sistemi che lo compongono. Una panoramica sugli aspetti relativi il bilanciamento del carico è stata presentata nei paragrafi precedenti.

Trasparenza di errore Questo requisito richiede che il sistema distribuito sia capace di gestire eventuali situazioni dovute al guasto, all'arresto, al riavvio o alla rimozione di un singolo sistema appartenente al sistema distribuito. In questo sistema potrebbero essere in esecuzione processi remoti oppure potrebbero esservi delle risorse accedute da altre macchine.

Trasparenza di scalabilità Questa proprietà consiste nel permettere di inserire o rimuovere dinamicamente macchine al sistema distribuito mentre in questo vi sono operazioni attive.

Le SSI possono essere realizzate su più possibili livelli ma molto difficilmente riescono a garantire tutte le caratteristiche sopra citate. A Livello Utente una SSI di solito consiste nel fornire accesso trasparente a tutte le risorse che appartengono all'ambiente distribuito, mentre a livello di Sistema Operativo questa è solitamente composta da un *File System* distribuito e da una gestione di processi distribuita. Sebbene implementare la migrazione di processi all'interno di una SSI possa apparire più semplice in quanto l'ambiente di esecuzione è unico e condiviso, in realtà le semplificazioni sono minime. Infatti, le difficoltà riscontrate nella realizzazione di un servizio di migrazione trasparente [4] vengono spostate nella realizzazione della SSI ma non vengono eliminate.

Un Sistema Operativo che implementa a Livello Kernel le caratteristiche di una *Single System Image* viene definito "*Sistema Operativo Distribuito*" [9].

2.6 Macchine Virtuali e Live Migration

Una tecnica interessante in grado di facilitare la migrazione di processi fa utilizzo delle Macchine Virtuali (VM). Mentre nella migrazione di processi tra macchine reali si riscontrano difficoltà legate all'ambiente di esecuzione, con le Macchine Virtuali questo ambiente viene isolato dal resto del sistema. In questo caso si prevede di migrare l'intera VM la quale al suo interno contiene i processi in esecuzione. La semplificazione di questa tecnica è correlata alla problematica della trasparenza: dal punto di vista del processo questo viene migrato insieme al suo ambiente di esecuzione, contenuto nella VM. In questo modo è logico dedurre che il processo può continuare la sua esecuzione nello stesso ambiente di esecuzione presente nel sistema originario, eliminando inoltre il problema della dipendenza residua in quanto è possibile garantire isolamento tra la macchina reale e quella virtuale. Questa tecnica viene denominata "*Live Migration*".

La tecnica *Live Migration* offre delle caratteristiche che sono la combinazione degli aspetti inerenti la migrazione dei processi e le Macchine Virtuali, perciò è una tecnica la cui implementazione dipende strettamente dalla tipologia di Macchina Virtuale e da come questa si colloca rispetto il Sistema Operativo.

Per valutare questa tecnica occorre iniziare dal distinguere quali sono [17] le principali tipologie di Macchine Virtuali e di tecniche di virtualizzazione:

Macchine Virtuali Complete Implementano l'architettura completa di un computer, vengono utilizzate per eseguire un Sistema Operativo completo all'interno di un altro Sistema Operativo. A sua volta, all'interno di questo si possono eseguire molte delle applicazioni sviluppate per esso come se fosse installato in una macchina reale.

Paravirtualizzazione Si tratta di una VM completa che però non traduce le istruzioni *Instruction Set Architecture* (ISA) tra la macchina virtuale e quella reale. Invece della traduzione viene implementato uno strato minimale di virtualizzazione, detto monitor, con lo scopo di astrarre l'hardware della macchina reale e virtualizzarlo per tutti i livelli sovrastanti. Questa tecnica offre prestazioni molto elevate ma richiede la ricompilazione dei Sistemi Operativi per poter essere eseguiti sul nuovo *hardware* virtuale.

Virtualizzazione di Sistema Operativo Si tratta di una VM che esegue un Sistema Operativo all'interno di un processo, a sua volta in esecuzione in un altro Sistema Operativo. A differenza della VM completa, questa non astrae né traduce l'ISA della macchina ma fa riferimento a quello della macchina reale. Questa alternativa fornisce migliori prestazioni rispetto la VM completa ma possiede meno portabilità in quanto l'implementazione è influenzata sia dal Sistema Operativo ospite che da quello ospitante.

Macchine Virtuali di Processo Tipologia di VM che non fornisce isolamento verso la macchina reale ma si limita ad offrire una interfaccia di sistema ai singoli processi che ne fanno uso. Vengono utilizzate generalmente per offrire nuovi elementi non disponibili nel sistema reale oppure per eseguire applicazioni realizzate per Sistemi Operativi differenti.

Macchine Virtuali Parziali Si tratta di una classe di VM che utilizza il concetto di "*vista di sistema*" di cui i processi dispongono durante la loro esecuzione. Queste VM permettono di modificare (in tutto o in parte) la percezione che i processi hanno dell'ambiente in cui sono eseguiti. Con questa tecnica è possibile fornire ai processi funzionalità eventualmente non disponibili nel sistema reale e i processi all'interno della VM possono accedere allo stesso tempo sia alle funzionalità offerte dallo strato di virtualizzazione parziale che a quelle offerte dal Sistema Operativo sottostante. Nel caso di una sovrapposizione tra le

funzionalità offerte da VM e Sistema Operativo vengono offerte quelle fornite dalla VM, permettendo così di modificare il funzionamento di servizi già presenti o eventualmente di nasconderli.

L'impiego della tecnica di *Live Migration* ha prodotto ottimi risultati in termini di affidabilità e prestazioni [14]. Uno degli elementi più critici nelle prestazioni di ogni forma di migrazione è il tempo in cui i processi restano congelati per consentire lo spostamento verso il nodo destinatario. Sebbene utilizzando VM la quantità di dati da migrare è maggiore rispetto la singola struttura del processo, impiegando nel modo migliore di questa tecnica si possono ridurre i tempi necessari per l'operazione rendendoli paragonabili a quelli del caso ottimo nella tradizionale migrazione del processo, ovvero attraverso la precopia. La meccanica è infatti identica a quella descritta precedentemente ed avviene distinguendo tre fasi principali per l'operazione:

1. **Fase *push*:** mentre la VM continua la sua esecuzione inizia la copia della memoria;
2. **Fase *stop-and-copy*:** la VM viene congelata e si completa la copia;
3. **Fase *pull*:** si sostituiscono le pagine di memoria modificate durante l'esecuzione della VM tra la prima e la seconda fase di copia.

I passaggi descritti focalizzano l'aspetto inerente la migrazione della memoria. Insieme a questi si individuano altri passaggi che complessivamente compongono il generico algoritmo di *Live Migration*:

1. **Fase *pre-migration*:** viene selezionato il nodo destinatario;
2. **Fase *reservation*:** il nodo destinatario si prepara ad accogliere la VM allocando le risorse necessarie;
3. **Fase *interative pre-copy*:** analoga della fase push di migrazione della memoria;

4. **Fase *stop-and-copy*:** analoga alla fase omonima di migrazione della memoria;

5. **Fase *commitment*:** corrisponde alla fase *pull* di migrazione della memoria e il nodo sorgente rilascia le risorse precedentemente allocate dalla VM migrata;

6. **Fase *activation*:** la VM viene riattivata nel nodo destinatario.

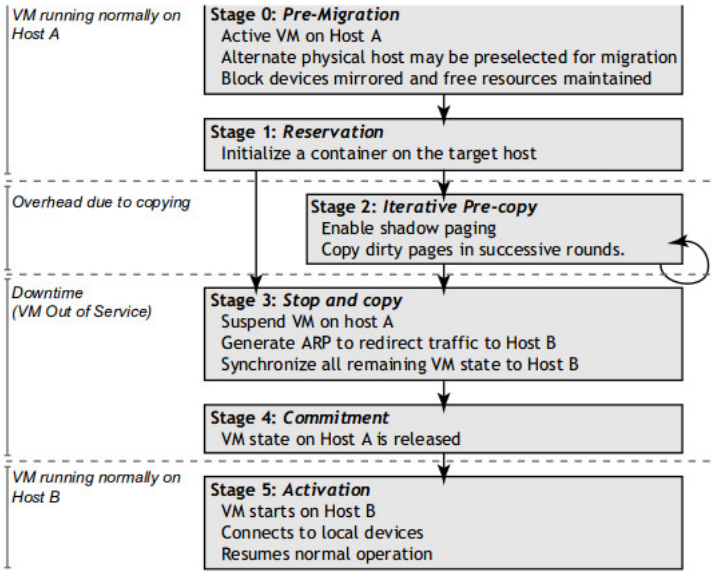

Figura 2.2: Schema di Live Migration [14]

Questa tecnica offre numerosi vantaggi e semplificazioni rispetto la tradizionale migrazione di processo [15]. Purtroppo possiede intrinsecamente una difficoltà che è bene non trascurare. Migrando tutto l'ambiente fornito dalla VM vengono anche migrati tutti i processi in esecuzione all'interno. Se è la VM stessa a soffrire del sovraccarico di processi contenuti, questa forma di migrazione non separa l'ambiente di esecuzione tra i processi. Questa situazione può essere affrontata in teoria duplicando l'istanza della VM e arrestando simmetricamente parte dei processi attivi, così da ottenere una

scissione della VM che può ricordare la semantica della System Call *fork()*. Questa ultima soluzione però non è generalizzabile in quanto dipende strettamente dalla tipologia di Macchina Virtuale e dal modo in cui il Sistema Operativo gestisce i processi. Le singole implementazioni di Macchine Virtuali possono tuttavia predisporre strategie atte a risolvere questa problematica a seconda delle modalità in cui queste sono implementate. Un'alternativa potrebbe essere infine limitare l'esecuzione di una VM ad un singolo processo.

2.7 Alternative alla migrazione di processo

Viste alcune delle difficoltà nel garantire una migrazione affidabile e trasparente, fatta eccezione in alcuni sistemi sviluppati originariamente per garantire questa tecnica, la questione deve essere posta anche per Sistemi Operativi tradizionali non facenti uso delle tecniche particolari precedentemente evidenziate. In queste situazioni esistono alcune soluzioni [4] semplici e spesso utilizzate per rispondere alle necessità, purtroppo raramente riescono a colmare tutte le possibili esigenze. Alcune tra le possibilità possono essere:

Esecuzione remota: Si tratta della soluzione più adottata in alternativa alla migrazione del processo. L'esecuzione remota è solitamente più veloce e semplice della migrazione in quanto il processo che necessita di eseguire un particolare codice in un altro sistema in rete si limita ad eseguire una richiesta ed attendere la risposta. Il sistema destinatario si preoccupa di ricevere le richieste, eseguirle e rispondere con il risultato della funzione eseguita. Questa tecnica è nota anche come RPC, *Remote Procedure Call*;

Clonazione: Questa tecnica è una variante della migrazione. Nei sistemi che implementano lo standard POSIX sono disponibili System Call come *fork()* e *clone()*, le quali si occupano di creare un nuovo processo che è una copia di quello che lo ha generato, permettendo ad entrambi di eseguire lo stesso codice dallo stesso punto in cui è avvenuta la richiesta. Come estensione, alcuni sistemi [19] implementano la possibilità

di eseguire queste System Call attraverso la rete, all'interno di un'altra macchina. Questa tecnica viene usualmente denominata "*Remote Forking*". Occorre prestare particolare attenzione in quanto questa meccanica non semplifica l'implementazione a basso livello del servizio di migrazione e non è prevista dallo standard POSIX;

Supporto del linguaggio di programmazione: Occorre ricordare che alcuni linguaggi non sono eseguiti direttamente nella macchina reale, ma sono interpretati o emulati all'interno di altre strutture. L'ambiente di esecuzione del processo è a sua volta un processo (solitamente implementato a Livello Utente), quindi è possibile, eventualmente attraverso alcune modifiche o estensioni dell'ambiente di esecuzione, effettuare un *checkpoint* del processo attivo per poterlo poi continuare ad eseguire in altre macchine oppure permettergli di procedere in un momento futuro.

2.8 Implementazioni esistenti

MOSIX è un Sistema Operativo per cluster capace di fornire un SSI e la migrazione trasparente dei processi. Il suo sviluppo procede dal 1977 e risulta ancora mantenuto ed utilizzato, in particolare per applicazioni di calcolo distribuito. Supporta la distribuzione del carico di lavoro e meccanismi di tolleranza ai guasti. Distribuito con licenza proprietaria.

OpenMosix Si tratta della versione libera ed *open source* di MOSIX, il quale è diventato proprietario a partire dal 2001. Il suo sviluppo, iniziato nel 2002, è stato abbandonato poi nel 2008 in quanto gli sviluppatori ritenevano che tale Sistema Operativo fosse diventato troppo difficile da mantenere e soluzioni alternative potevano fornire soluzioni analoghe alla SSI ma più economiche rispetto i cluster per il quale OpenMosix era sviluppato. Licenza GNU GPLv2.

LinuxPMI (*Linux Process Migration Infrastructure*) è semplicemente il proseguimento del Sistema Operativo OpenMosix, il suo sviluppo è stato

ripreso da altri in quanto ritenevano lo strumento utile da mantenere. Lo sviluppo consiste in una collezione di patch al Kernel che implementano il servizio di migrazione di processi ma senza fornire SSI. Licenza libera.

OpenSSI è stato sviluppato per fornire una SSI all'interno di cluster. Basato sul Kernel Linux, supporta varie caratteristiche delle SSI incluso il servizio di migrazione di processi. La sua implementazione cerca di recuperare gli approcci di LOCUS, un Sistema Operativo Distribuito oggi non più sviluppato e considerato storico. Rilasciato con doppia licenza libera LGPL e GPLv2.

Kerrighed è un Sistema Operativo SSI, simile ad OpenSSI, pensato per operare su cluster di macchine. La sua implementazione consiste in un insieme di patch e moduli ausiliari per il Kernel Linux. La migrazione dei processi è tuttavia fornita con un supporto limitato ad alcune caratteristiche e non tutti possono essere migrati. Distribuito con licenza libera GNU GPLv2.

Capitolo 3

Tecniche di Checkpoint/Restart

3.1 Introduzione

Nei precedenti capitoli si è presentata una panoramica relativa la migrazione di processi, le caratteristiche, i vantaggi e le problematiche. Si sono anche approfonditi gli aspetti implementativi e alcune delle possibili alternative per garantire che il servizio sia affidabile e trasparente. Sebbene la teoria dimostra che la tecnica è generalmente possibile, l'implementazione della completa trasparenza a livello di Sistema Operativo nei sistemi UNIX e derivati si è rivelato essere un compito estremamente difficile poiché gli elementi che legano un processo all'ambiente di esecuzione sono molteplici. Tanenbaum ritiene [11] che l'obiettivo di realizzare una SSI sia impossibile da raggiungere nella sua totalità. Trascurando parte della trasparenza, la tematica potrebbe essere affrontata in termini di "*pseudo-migrazione*" e consentendo che alcuni degli elementi non siano migrati (come il *process id*) oppure potrebbero essere utilizzate tecniche di virtualizzazione in grado di astrarre parte dell'ambiente di esecuzione del processo [7] in modo tale che questo possa essere visto omogeneo tra le macchine. Milojičić ritiene [4] che l'obiettivo potrebbe essere perseguito tramite una architettura di Sistema Operativo pensata fin dalle origini per essere distribuito, tuttavia allo stesso tempo afferma che questo compito non è perseguibile in pratica poiché occor-

rerebbe investire troppo tempo ed energia, perdendo infine la compatibilità delle applicazioni con i sistemi UNIX e derivati. Implementazioni esistenti (MOSIX ed altri) tentano invece di perseguire lo scopo attuando modifiche a livello Kernel, raggiungendo ottimi risultati ma talvolta riscontrando difficoltà nell'implementazione della completa trasparenza. La tendenza generale è stata quindi quella di risolvere la problematica utilizzando Macchine Virtuali che possono garantire elevate prestazioni, isolamento, affidabilità ed altri vantaggi implementativi.

La tematica che verrà affrontata con maggiore dettaglio in questo capitolo è quella del *Checkpoint/Restart*, una tipologia di tecnica che permette di estrarre lo stato di un processo in esecuzione e salvarlo su file con un idoneo formato di rappresentazione. Questo *checkpoint* può quindi essere utilizzato per avviare successivamente il processo, permettendogli di ripartire dallo stato in cui il *checkpoint* è stato creato. Le possibilità di utilizzo di questa tecnica sono molteplici, tra i quali permettere la migrazione del processo qualora il riavvio (*restart*) avvenga in una macchina differente da quella in cui il *checkpoint* è stato creato. Vantaggi, limitazioni, problematiche ed utilizzi nel contesto della migrazione dei processi verranno affrontati nel corso dei successivi paragrafi.

3.2 Aspetti generali

La tecnica di *Checkpoint/Restart*, anche nota come *Checkpoint/Restore* e riferita comunemente con la sigla C/R, viene utilizzata da molto tempo nei Sistemi Operativi e le prime testimonianze [18] di utilizzo risalgono agli anni '60. Le singole implementazioni variano a seconda del Sistema Operativo e dalle scelte di realizzazione.

Questa tecnica consiste nel creare una "fotografia" (*snapshot*) di un processo in esecuzione. Questo snapshot, se accuratamente dettagliato, può essere utilizzato per l'operazione di *restart*, ovvero rieseguire il processo in un

momento futuro consentendo di ripartire dallo stesso istante di elaborazione in cui lo *snapshot* è stato creato.

La creazione dello *snapshot* è solitamente seguita dalla creazione di uno o più file che memorizzano sul disco le strutture dati che descrivono il processo. L'unione di queste due operazioni è quanto consiste l'operazione di *checkpoint*. L'insieme di tutti i file creati durante l'operazione di *checkpoint* prende, a sua volta, il nome di *checkpoint*.

Durante la fase di *checkpoint* il processo viene congelato per assicurare che nessuna operazione produca un cambiamento di stato prima di aver completato la lettura di tutte le informazioni, operazione che quindi deve avvenire atomicamente. L'esecuzione del *checkpoint* può essere o meno seguita (contestualmente) dalla terminazione del processo su cui si sta operando.

Questa tecnica viene utilizzata per differenti ragioni, alcune delle quali sono:

- Fornire un meccanismo di tolleranza ai guasti, in caso di terminazione non desiderata del processo è possibile ripristinare l'esecuzione dall'ultimo *checkpoint* effettuato;

- Fornire uno strumento di *debugging* delle applicazioni attraverso il quale lo sviluppatore può determinare lo stato del processo nel momento in cui si è verificato un errore;

- Miglioramento delle prestazioni di avvio di un processo che può essere ripristinato dallo stato seguente una fase di inizializzazione particolarmente lunga;

- Salvataggio della sessione di lavoro;

- Migrazione di processo ad alto livello che non richieda un esplicito supporto di migrazione da parte del Sistema Operativo.

3.3 Lo snapshot di un processo

La tecnica di C/R si basa essenzialmente sulla possibilità di congelare l'esecuzione del processo ed estrarne lo stato. L'operazione comunemente effettuata è quindi quella di scrivere gli elementi che compongono lo stato in uno o più file di rappresentazione dello stato. In questo caso non vi è una regola stabilita che impone di memorizzare i dati in un formato particolare, a titolo di esempio nei sistemi GNU/Linux si potrebbe optare per la decisione di utilizzare un formato standard di rappresentazione come "*a.out*" per vecchie versioni del Kernel Linux (precedenti alla 2.0) oppure ELF (*Executable and Linkable Format*) che è il formato adottato nelle versioni più recenti.

L'aspetto più importante nella creazione dello snapshot è la possibilità di garantire che il restart possa avvenire correttamente, mettendo a disposizione tutte le informazioni necessarie che caratterizzano lo stato del processo.

Le informazioni che descrivono lo stato del processo [12] comprendono:

Text section contiene il codice del programma;

Data section contiene le variabili globali;

Stack section contiene i dati temporanei;

Heap section contiene la memoria allocata dinamicamente;

Process Control Block (PCB) struttura dati attraverso la quale il Sistema Operativo gestisce il processo.

Il PCB a sua volta contiene informazioni molto importanti come il *Process ID*, il *Program Counter*, i valori dei registri della CPU, lo stato di esecuzione, la priorità di esecuzione e molte altre ancora che non devono essere tralasciate, pena un mancato corretto ripristino.

3.4 Migrazione di processo tramite C/R

La tecnica di C/R può essere utilizzata in combinazione con un servizio di trasferimento file per fornire una tipologia di migrazione di processo che può essere implementata anche ad alto livello [18]. Lo schema di funzionamento è molto semplice e può essere riassunto nei seguenti passaggi:

1. Viene creato un *checkpoint* di un processo in esecuzione;
2. I dati vengono trasferiti verso la destinazione remota;
3. Completato il trasferimento dei dati, si esegue il *restart* del processo.

Volendo classificare questa modalità di migrazione sulla base di come viene copiata la memoria, questa tecnica appartiene alla tipologia *eager (all)*. Come è stato presentato precedentemente, questa modalità ha lo svantaggio di essere particolarmente lenta rispetto le possibili alternative. Tuttavia, supponendo di disporre di uno strumento C/R, per implementare i passaggi descritti non è richiesto di intervenire direttamente nel Sistema Operativo, se il *checkpoint* è fornito sotto forma di file allora la migrazione può avvenire ad alto livello con qualsiasi meccanismo di trasferimento e condivisione file.

A discapito della relativa semplicità vi sono altre complicazioni riguardanti l'atto del *restart* che complicano ulteriormente la trasparenza di migrazione rispetto le tecniche viste precedentemente.

3.5 Problematiche nel restart di un processo

In uno strumento di C/R l'implementazione della parte di *checkpoint* può apparire relativamente semplice. Se il Sistema Operativo fornisce gli strumenti idonei ad estrarre lo stato di esecuzione di un processo completo delle variabili di ambiente che lo caratterizzano, l'operazione può concludersi con il salvataggio di strutture dati su di un file binario. In effetti, con riferimento al Sistema Operativo GNU/Linux, le tecniche per ottenere informazioni

dettagliate su di un dato processo sono molteplici, alcune delle quali sono anche ben consolidate ed utilizzate anche da strumenti per il debugging delle applicazioni. A titolo esemplificativo basti pensare alle System Call *ptrace()* o *strace()*.

Questione ben diversa si pone nel momento in cui si desidera ripristinare l'esecuzione di un processo. Nel caso che il *restart* del processo avvenga in una macchina differente, si presentano tutte le problematiche di trasparenza viste in precedenza. Oltre a queste si presentano anche altre difficoltà [19] intrinseche, proprie di questo modo di procedere.

Supponiamo quindi di eseguire un processo, crearne il *checkpoint* in un momento qualsiasi della sua esecuzione, arrestare il processo ed infine eseguire il *restart*. Anche disponendo di tutte le informazioni più dettagliate possibili si può notare che alcune di queste sono apparentemente inutili. Il processo mantiene internamente molte informazioni e alcune di queste sono dei riferimenti a delle risorse e strutture gestite nel Kernel, altre sono assegnate alla singola istanza del processo e che al momento della chiusura vengono rilasciate.

Gli elementi più rilevanti che riscontrano queste problematiche sono:

- *Process ID* (PID)
- *File Descriptor*
- Stato delle connessioni di rete
- Strumenti di *Inter-Process Communication* (IPC)
- Gestione dei segnali
- Terminale associato al processo

Cercando di ripristinare il processo con i valori preesistenti nel *checkpoint* alcune operazioni potrebbero essere negate (come richiedere che un processo venga creato con un dato PID), oppure il processo potrebbe richiedere di continuare operazioni in delle strutture rilasciate (come *pipe*, socket, memoria

condivisa o altro ancora) oppure le stesse risorse potrebbero essere occupate da altri processi (come nel caso del terminale).

Oltre a queste vi sono altre problematiche caratteristiche del *checkpoint* e che, sebbene di carattere più generico rispetto la tematica, possono influire sul corretto funzionamento e ripristino del processo.

Privilegi di esecuzione

Il processo del quale è stato richiesto il *checkpoint* potrebbe essere in esecuzione con particolari privilegi, ad esempio di amministratore. Al riavvio, secondo la linea di principio di continuità trasparente, il processo dovrebbe possedere gli stessi privilegi della precedente esecuzione. Questo può portare a numerose considerazioni in tema di sicurezza da non trascurare.

Tempo di esecuzione

La questione per questo aspetto si presenta in maniera sottile. Sorge spontanea una domanda: come stabilire quale sia il tempo di esecuzione di un processo, se deve essere considerato come il tempo trascorso dal momento del primo avvio o come il tempo effettivo in cui il processo appare in esecuzione nel processore? Inoltre, se il *checkpoint* avviene mentre il processo stava eseguendo la System Call *sleep()* o analoghe, se il *restart* avviene dopo un tempo sufficientemente ampio, come valutare il tempo trascorso in questa situazione? Queste domande hanno risposte che possono essere soggettive e la scelta finale ricade nell'implementazione del sistema di C/R.

Genealogia di processi

Un processo può eseguire System Call come *fork()*, *system()*, *popen()* ed altre ancora che permettono di eseguire nuove istanze di processo che vengono indicate come "*processi figli*". Le considerazioni fatte riguardo la tecnica di C/R non dovrebbero pertanto essere limitate ad individuare come obiettivo del *checkpoint* un singolo processo. L'approccio generalmente seguito infatti

è di estendere il *checkpoint* a tutta la genealogia discendente del processo designato, come anche il *restart*. Questa scelta viene fatta nell'implementazione del servizio di C/R, tuttavia questo aspetto può essere fortemente consigliabile qualora gli strumenti basilari di IPC vengono supportati.

Possibili soluzioni

Risolvere tutti questi problemi non è una questione semplice. Prendendo in esame le implementazioni esistenti sono tendenzialmente tre le soluzioni fornite:

1. Limitato o nessun supporto per le caratteristiche riportate;
2. Estensione delle funzionalità di sistema tramite modifica al Kernel o moduli ausiliari;
3. Porre una interfaccia di virtualizzazione tra processo e Sistema Operativo.

3.6 Utilizzo della Virtualizzazione Parziale

L'impiego delle tecniche di C/R riscontra delle difficoltà nel momento in cui il processo viene ripristinato. L'utilizzo delle funzionalità offerte dal Sistema Operativo attraverso le System Call forniscono una semplificazione nella realizzazione delle applicazioni che utilizzano elementi di sistema (ad esempio il *File System*) in quanto non richiedono di implementare internamente tutta la gestione a basso livello. Tuttavia questi aggiungono ulteriori legami tra il processo e l'ambiente di esecuzione.

Una possibilità per risolvere questa problematica è inserire uno strato di astrazione posto tra processo e Sistema Operativo in modo da isolare e contenere l'ambiente di esecuzione. Intercettando le System Call è quindi possibile rilevare quali sono gli elementi di sistema che vengono utilizzati ed in quale modo. Allo stesso tempo è possibile cambiarne il comportamento

in modo tale da far si che sia lo strato virtuale ad eseguire le System Call per conto del processo. Questa tipologia di virtualizzazione appartiene alla classe delle Macchine Virtuali Parziali.

L'unione delle tecniche di virtualizzazione parziale e delle tecniche di C/R, attraverso l'implementazione di caratteristiche atte a fornire al processo una rappresentazione trasparente dell'ambiente, semplifica l'operazione di *restart* e può garantire una continuità nella fornitura dei servizi di sistema. Utilizzando questo modo di procedere per la migrazione è necessario che insieme al processo venga migrata anche la Macchina Virtuale Parziale o almeno le informazioni raccolte durante la sua esecuzione. Al momento del *restart* gli elementi come il *process id* o i *file descriptor* possono cambiare dal punto di vista della Macchina Virtuale ma questa può gestirli e farli apparire al processo come gli elementi della precedente esecuzione. Anche per gli elementi che hanno bisogno di essere ripristinati, come le connessioni di rete, l'operazione di ripristino può essere delegata alla Macchina Virtuale ed il processo può quindi continuare l'esecuzione senza che sia reso consapevole della modifica avvenuta. Questa tipologia di soluzione nelle sue varianti di sviluppo viene utilizzata generalmente [15] con i sistemi per il calcolo distribuito.

3.7 Implementazioni esistenti

CRIU (*Checkpoint/Restore In Userspace*) è uno strumento di *checkpoint* in fase di sviluppo implementato quasi interamente a Livello Utente, per fornire funzionalità aggiuntive necessita del supporto di moduli Kernel ausiliari. Sebbene ancora considerato non terminato riesce ad effettuare il *checkpoint* di numerose tipologie di applicazioni. L'implementazione di questo ha richiesto di integrare numerose patch al Kernel che sono poi state inserite nella *mainline* di sviluppo del Kernel Linux. Lo stesso Linus Torvalds[1] ha preso in esame il lavoro effettuato rimanendo tut-

[1] `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=099469502f62fbe0d7e4f0b83a2f22538367f734`

tavia scettico in merito un completo funzionamento finale. Utilizzando caratteristiche del Kernel introdotte recentemente, per funzionare richiede una versione di Linux almeno 3.9 o superiore. Rilasciato con licenza GNU GPLv2.

BLCR (*Berkeley Lab Checkpoint/Restart*) offre numerose funzionalità ma è incompleto, non riesce a coprire tutte le possibili situazioni e non supporta i socket. Sviluppato in Livello Kernel sotto forma di modulo aggiuntivo, questo impedisce alle applicazioni di poterlo includere per usufruire delle sue funzionalità. Tuttavia in questo modo riesce a gestire ed effettuare il ripristino di elementi come *process id*, *thread group id*, *pipe* ed altro ancora. Rilasciato con licenza BSD.

OpenVZ Si tratta di uno strumento di virtualizzazione a livello di Sistema Operativo, è in grado di eseguire più istanze isolate di Macchine Virtuali (denominate "*containers*") che forniscono ciascuna un Livello Utente separato ma condividendo lo stesso Kernel. Questo strumento è reso possibile attraverso delle modifiche al Kernel Linux. Tramite una estensione denominata CPT è in grado di supportare il *checkpoint* ed il *Live Migration* degli interi container. Distribuito liberamente con licenza GNU GPLv2.

CryoPID Strumento sviluppato principalmente per vecchie versioni del Kernel Linux (comprese tra 2.4 e 2.6). Attualmente sembrerebbe non essere supportato nelle versioni recenti. Il funzionamento consiste nell'estrazione dello stato del processo che viene scritto su di un file. Molte caratteristiche non sono supportate o lo sono solo parzialmente, come la gerarchia di processi, i terminali, i socket ed i *file descriptor*. Rilasciato con doppia licenza BSD e GNU GPLv2.

Zap Sviluppato dalla *Columbia University*, non richiede modifiche al Kernel e riesce ad effettuare il *checkpoint* di applicazioni distribuite e *multi-threaded*. Il servizio viene fornito attraverso un minimo strato di virtualizzazione posto tra processo e Sistema Operativo che produce un

completo isolamento dell'ambiente di esecuzione. Licenza proprietaria, non disponibile pubblicamente.

Deja-Vu Sviluppato da *Virginia Tech*, opera a Livello Utente e riesce ad effettuare il *checkpoint* di applicazioni distribuite e *multithread*. Da esperimenti effettuati con programmi di *benchmark* sembra che questo strumento generi un *overhead* significativo nei processi tracciati producendo un rallentamento nell'esecuzione di circa 97 volte in più rispetto la normale esecuzione. Licenza proprietaria, non disponibile pubblicamente.

DMTCP Strumento in grado di effettuare *checkpoint* simultaneo di più applicazioni, incluse applicazioni distribuite e *multithread*, sviluppato interamente a Livello Utente senza avere necessità di modificare il Kernel o installare moduli ausiliari. Per essere utilizzato non richiede la modifica del codice dell'applicazione. Il prossimo capitolo verrà dedicato ad analizzare nel dettaglio questo strumento. Rilasciato con licenza LGPL.

Capitolo 4

DMTCP: Distributed MultiThreaded CheckPointing

4.1 Introduzione

Dalle analisi effettuate nei precedenti capitoli si sono evidenziati gli aspetti problematici della migrazione. Nonostante le difficoltà di fornire una trasparenza distribuita tra più Sistemi Operativi, anche gli elementi che caratterizzano lo stato in esecuzione di un processo sono ardui da migrare in quanto alcuni di questi, come le connessioni di rete, sono in parte gestiti all'interno del Kernel. Tra le varie possibilità si può prevedere di estendere le funzionalità del Kernel attraverso moduli ausiliari oppure interponendo tra processo e Sistema Operativo una idonea interfaccia che, dal punto di vista del processo, appare come un Sistema Distribuito o uno strato di virtualizzazione dell'ambiente sottostante.

In questo capitolo verrà analizzato nel dettaglio DMTCP, uno strumento ideato per la creazione di *checkpoint* di processi e con lo scopo di supportare computazioni parallele. Gli aspetti implementativi di DMTCP verranno presi in esame nel dettaglio con riferimento alla versione 1.2.8 e successivamente verranno mostrate delle possibilità di utilizzo per migrazione di processi in GNU/Linux con implementazione a livello Utente.

DMTCP è rilasciato con licenza *Lesser GNU Public License* (LGPL) ed è liberamente scaricabile dal sito `http://dmtcp.sourceforge.net/`

4.2 Descrizione

DMTPC [1] è uno strumento implementato interamente a livello utente che permette la creazione di *checkpoint* di applicazioni distribuite. Questo viene definito "*trasparente*" in quanto per essere utilizzato non richiede la modifica o la ricompilazione di applicazioni precedentemente realizzate. Inoltre, non sono richiesti privilegi di amministratore per essere eseguito e viene fornita compatibilità dalla versione 2.6.9 del Kernel Linux.

Questo strumento è in grado di supportare il *checkpoint* di applicazioni che utilizzano System Call ed elementi di sistema quali: *fork()*, *exec()*, ssh, *mutex*/semafori, pseudo-terminali, modalità e controllo del terminale, gestori di segnali, file aperti, *file descriptor* condivisi, strumenti di I/O, memoria condivisa, relazioni padre-figlio tra processi, virtualizzazione del pid ed altro ancora.

Il funzionamento di questo strumento si basa su di un altro precedentemente realizzato, MTCP (*MultiThreaded CheckPointing*). Quest'ultimo consente di eseguire il checkpoint di singoli processi [3]. DMTCP ed MTCP sono due strati separati l'uno dall'altro e possono comunicare attraverso delle API. Questa scelta è stata adottata per incrementare la portabilità dello strumento verso sistemi non-Linux, rendendo sufficiente modificare MTCP in modo che fornisca le stesse funzionalità ma con astrazione del sistema. Come analizzato in precedenza, l'implementazione delle funzionalità di *checkpoint* dipende dalle caratteristiche del Sistema Operativo e DMTPC estende il funzionamento di MTCP replicando lo stesso servizio in tutti i processi discendenti nella genealogia, inoltre fornisce funzionalità aggiuntive per la gestione di processi distribuiti in rete che comunicano con meccanismi di IPC.

Alcuni dettagli implementativi possono essere recuperati in:
`dmtcp-1.2.8/dmtcp/src/mtcpinterface.cpp`

Alla base del *checkpoint* in questo strumento [2] vi è l'inserimento di un thread aggiuntivo in ogni processo e che possiede funzionalità di gestione. Inoltre vengono inseriti dei "*wrapper*" (involucri) alle funzioni della *glibc* al fine di controllare l'esecuzione delle operazioni eseguite dal processo. La combinazione del *thread* di gestione e dei *wrapper* alle funzioni della *glibc* permette di tracciare l'esecuzione delle System Call eseguite dal processo, modificarne il funzionamento (come nel caso della *fork()*, per le ragioni descritte sopra), bloccare l'esecuzione delle operazioni durante la richiesta di *checkpoint* e posticipare questa richiesta qualora avvenga durante l'esecuzione di una System Call. Allo stesso tempo il *thread* di gestione memorizza le operazioni eseguite e permette un corretto ripristino al momento del *restart*, modificando eventualmente l'esecuzione delle System Call e cambiando i valori di ritorno, sostituendoli a quelli appartenenti alla precedente esecuzione.

A fine esemplificativo prendiamo in esame la System Call *getpid()*. Al momento della prima esecuzione il pid del processo è un determinato valore, al momento del restart questo valore è in generale differente in quanto dal punto di vista del Sistema Operativo si riferisce alla nuova istanza. Invece di eseguire la System Call del Sistema Operativo, la funzione *wrapper* di *getpid()* si interpone alla chiamata e ritorna il valore del pid salvato in precedenza. In questo modo, sebbene per il Sistema Operativo il processo possiede un pid differente, per il processo questo non vale in quanto gli appare sempre lo stesso valore.

In `dmtcp-1.2.8/dmtcp/src/pidwrappers.cpp` notiamo infatti:

```
extern "C" pid_t getpid()
{
   return dmtcp::VirtualPidTable::instance().pid();
}
```

Una delle limitazioni per poter utilizzare le funzionalità fornite da questo strumento è il requisito che il programma del quale si intende richiedere suc-

cessivamente il *checkpoint* sia avviato contestualmente a DMTCP (fornendo il programma come parametro).

4.3 Utilizzo dello strumento

DMTCP viene distribuito con una collezione di strumenti che permettono il servizio di C/R. Gli strumenti principali sono:

dmtcp_coordinator Viene utilizzato per coordinare applicazioni distribuite che richiedono il *checkpoint*. L'utilizzo di questo strumento è necessario per poter comunicare con i processi che utilizzano DMTCP. Nel caso si debba gestire una sola applicazione locale questo può essere utilizzato come server *localhost*.

dmtcp_checkpoint Si tratta dello strumento più importante, viene eseguito contestualmente all'applicazione della quale si richiede il *checkpoint*. I dettagli di funzionamento di questo strumento verranno descritti in seguito.

dmtcp_command Questo strumento viene utilizzato per comunicare con dmtcp_coordinator e richiedere di eseguire dei specifici comandi. In particolare può essere utilizzato per richiedere il checkpoint delle applicazioni gestite.

dmtcp_restart Viene utilizzato per riavviare applicazioni dai rispettivi *checkpoint*.

Alla richiesta di un *checkpoint* vengono creati i rispettivi file, uno per ogni processo che comprende l'applicazione. Questi file, con estensione .dmtcp, contengono tutte le informazioni di stato neccessarie per un corretto ripristino. Insieme a questi file viene creato anche lo script "`dmtcp_restart_script.sh`" che può essere utilizzato per il *restart* del processo in alternativa a dmtcp_restart. Questo script in realtà è un link simbolico ad un altro script creato con l'ultimo *checkpoint* richiesto: può risultare particolarmente utile nel caso vi

siano numerosi *checkpoint* di una applicazione e si desidera riavviare soltanto l'ultimo a disposizione.

Un esempio di utilizzo, tramite terminale differente per ciascun comando, può essere il seguente:

1. Avviare localmente lo strumento di coordinazione di DMTCP:

   ```
   $ dmtcp_coordinator
   ```

2. Avviare il programma desiderato insieme al servizio di *checkpointing*:

   ```
   $ dmtcp_checkpoint <programma> [args..]
   ```

3. Richidere il *checkpoint* del programma in esecuzione:

   ```
   $ dmtcp_command --checkpoint
   ```

4. *Restart* del programma utilizzando lo script "`dmtcp_restart_script.sh`" oppure eseguire:

   ```
   $ dmtcp_restart <file_1.dmtcp> .. <file_n.dmtcp>
   ```

4.4 Inizializzazione di DMTCP

Al momento dell'avvio di un nuovo processo, DMTCP si preoccupa di eseguire un "*injection*" della libreria "`dmtcphijack.so`" tramite l'utilizzo della variabile di ambiente LD_PRELOAD. Questa libreria altro non è che il *thread* di gestione nel processo e, come descritto, il suo compito è permettere l'esecuzione dei successivi *checkpoint* e *restart* desiderati. Viene utilizzato LD_PRELOAD in quanto consente di eseguire porzione di codice prima dell'esecuzione della routine *main()* del programma. Essendo una operazione che avviene dinamicamente all'atto dell'esecuzione, non richiede pertanto la modifica o la ricompilazione del sorgente del programma che intende essere eseguito con DMTCP.

La prima operazione eseguita mediante "`dmtcphijack.so`" è quella di verificare la presenza di dmtcp_coordinator in esecuzione (localmente o remotamente) e nel caso in cui questo non fosse attivo, eseguirlo sotto forma di demone locale.

Le operazioni eseguite successivamente nel *thread* principale ("*user thread*") prima della routine *main()* sono:

1. Creazione del socket per la comunicazione con dmtcp_coordinator;
2. Creazione del *signal handler* "*stopthisthread()*" utilizzato per gestire il segnale `STOPSIGNAL` (definito di default `SIGUSR2`) utilizzato per comunicare l'esecuzione della routine di *checkpoint*;
3. Creazione del *thread* di gestione ed attesa del completamento dell'operazione;
4. Esecuzione della routine *main()*.

Il *thread* di gestione, chiamato "`checkpointhread`", esegue le seguenti operazioni:

1. Connessione del socket precedentemente creato con con dmtcp_coordinator;
2. Comunicazione del completamento dell'operazione all'*user thread*;
3. Esecuzione della *select()* con il socket connesso a dmtcp_coordinator.

Alcuni dettagli implementativi possono essere recuperati in:
`dmtcp-1.2.8/dmtcp/src/dmtcp_checkpoint.cpp`
`dmtcp-1.2.8/dmtcp/src/dmtcpworker.cpp`

La logica di questa struttura è che in qualsiasi momento dell'esecuzione dell'applicazione si possono verificare solamente due possibili situazioni:

1. Il thread principale (*user thread*) è in esecuzione ed è disponibile il *signal handler*, mentre il `checkpointhread` è bloccato in attesa sulla *select()*;

2. Il thread principali (*user thread*) è bloccato sul signal handler, mentra il `checkpointthread` è stato risvegliato dalla *select()* con un messaggio diretto al socket connesso a dmtcp_coordinator.

Questa mutua esclusione è garantita dall'esecuzione di `STOPSIGNAL` (`SIGUSR2`) in `checkpointthread` subito dopo il risveglio dalla *select()*.

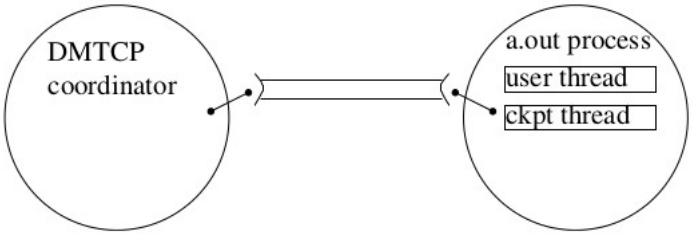

Figura 4.1: Controllo dei thread in DMTPC [2]

Occorre notare che un processo può creare nuovi *thread* che inizialmente non avrebbero il *signal handler* come quello creato per l'user *thread* principale. In tal caso interviene la funzione *wrapped* per la creazione del *thread* ed aggiunge il *signal handler* richiesto. In questo modo tutti i *thread* creati (eccetto `checkpointthread`) dispongono di un *signal handler* con lo scopo di bloccare l'esecuzione al momento del *checkpoint*. Questo principio viene applicato con tutte le System Call che consentono la creazione di nuovi *thread*, nuovi processi e nuovi processi remoti, come *pthread_create()*, *clone()*, *fork()*, *system()* e altre. L'utilizzo della libreria "`dmtcphijack.so`" viene utilizzata nella creazione di ogni nuovo processo, quindi anche le funzioni di *wrap* della *glibc* vengono interposte a quelle usuali.

Alcuni dettagli implementativi possono essere recuperati in:
`dmtcp-1.2.8/dmtcp/src/execwrappers.cpp`
`dmtcp-1.2.8/dmtcp/src/threadwrappers.cpp`

4.5 Checkpoint in DMTCP

Il *checkpoint* del programma in esecuzione avviene nei seguenti passaggi:

1. dmtcp_coordinator, alla richiesta dell'utente o dopo un intervallo di tempo stabilito, manda un messaggio "`CKPT`" a tutti i socket connessi;
2. Tuttli i `checkpointhread` creati da DMTCP vengono risvegliati dalla *select()*;
3. Ogni `checkpointhread` a sua volta manda il segnale `STOPSIGNAL` (`SIGUSR2`) ai *signal handler* dei vari *user thread* del processo;
4. Ogni *thread* esegue la funzione *stopthisthread()* che ne blocca l'esecuzione su di un semaforo;
5. Quando tutti gli *user thread* sono bloccati, viene eseguito il *checkpoint*;
6. Vengono sbloccati tutti i *thread* rimasti in attesa sul semaforo;
7. Il `checkpointhread` riesegue la *select()* e ritorna in attesa di future richieste di *checkpoint*.

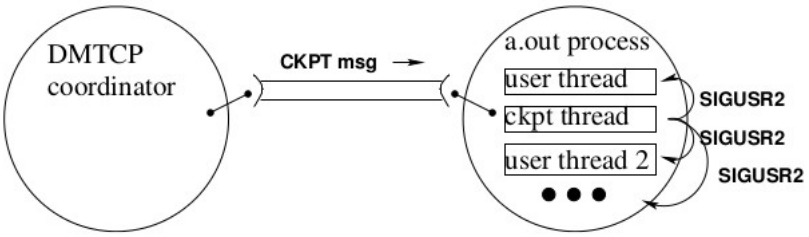

Figura 4.2: Esecuzione del checkpoint in DMTCP [2]

Nel momento del salvataggio dello stato del processo avvengono alcune operazioni:

1. Si elegge il processo "*leader*" per *file descriptor* condivisi;

2. Vengono raccolti i dati ancora presenti nei *buffer* del Kernel;

3. Vengono scritti i file di *checkpoint* sul disco;

4. Vengono ripristinati i dati nei rispettivi *buffer* del Kernel.

Le operazioni di raccolta e ripristino dati dei buffer del Kernel avvengono con lo scopo di garantire un corretto ripristino, sono implementate in quanto il Kernel, per migliorare le prestazioni generali del sistema, gestisce dei buffer di dati destinati all'applicazione ma che potrebbero non essere ancora stati letti. Esempi tipici in cui i buffer possono contenere dati non ancora ricevuti dalle applicazioni si presentano con le comunicazioni di rete, con la lettura di dati dal disco e con la gestione dell'output nel terminale.

Alcuni dettagli implementativi possono essere recuperati in:
`dmtcp-1.2.8/mtcp/mtcp.c`

4.6 Restart in DMTCP

L'operazione di restart è la più delicata in quanto è richiesto di ripristinare tutti gli elementi di sistema utilizzati dall'applicazione. Le difficoltà sono state prese in esame nel precedente capitolo.

DMTPC utilizza diverse strategie di risoluzione, alcune delle quali sono di natura di virtualizzazione (come nel caso della gestione del pid) altre invece sono legate al corretto utilizzo delle informazioni raccolte. Questo punto avviene nei seguenti passaggi:

1. Riapertura dei file e dei socket e ricostruzione dei pseudo-terminali;

2. Ricostruzione delle connessioni dei socket;

3. Esecuzione di *fork()* per il ripristino della genealogia;

4. Ricollocazione dei *file descriptor* tra i processi utilizzando *dup2()* e *close()*;

5. Ripristino della memoria e dei *thread*;

6. Riempimento dei *buffer* del Kernel;

7. Ripristino degli *user thread*.

Alcuni dettagli implementativi possono essere recuperati in:
`dmtcp-1.2.8/dmtcp/src/dmtcp_restart.cpp`

Il ripristino delle connessioni è una delle parti più delicate. Al fine di poter ricostruire correttamente la comunicazione si richiede alle funzioni di *wrap* di salvare informazioni di connessione nel momento di esecuzione delle System Call *connect()* ed *accept()*. Tra queste informazioni vi è inclusa la "*globally unique socket ID*" che funge da riferimento univoco per il socket in tutta la rete. Al momento del *restart* questo valore viene utilizzato per ritrovare il socket remoto con cui era precedentemente connesso. Appena questo viene ritrovato vengono eseguite *dup2()* e *close()* da entrambe le parti, permettendo all'applicazione di continuare nella comunicazione ripristinata. La limitazione di questa tecnica è che entrambi i socket devono essere connessi attraverso l'interfaccia di DMTCP, oppure il socket remoto dovrebbe essere fornito con il protocollo di ripristino implementato.

Alcuni dettagli implementativi possono essere recuperati in:
`dmtcp-1.2.8/dmtcp/src/connection.cpp`
`dmtcp-1.2.8/dmtcp/src/connectionmanager.cpp`

4.7 Integrazione di nuove funzionalità

Tra le funzionalità offerte da DMTCP vi sono anche due possibilità che permettono di estendere il funzionamento di questo servizio:

Moduli per DMTCP L'impiego dei moduli è stato ideato per permettere di modificare il funzionamento della libreria "`dmtcphijack.so`" in modo da consentire agli sviluppatori di poter utilizzare le funzioni di *wrapping* alla *glibc* ma con un comportamento differente rispetto quello cui

cui DMTCP è fornito. Inoltre è possibile aggiungere nuovi *wrapping* ad altre funzioni oppure *wrapping* alle funzioni di *wrapping* già presenti. Infine vengono messi a disposizione degli *hooks* denominati "*DMTCP event*", i quali permettono di aggiungere porzioni di codice determinati punti dell'esecuzione: prima del checkpoint, dopo il *checkpoint* e prima del *restart*.

DMTCPAware Questo strumento viene fornito per permettere di costruire applicazioni che utilizzino internamente DMTCP, ovvero consente di aggiungere funzionalità di *checkpoint* a nuovi programmi. Questo strumento dispone sia gli *hooks* "*DMTCP events*" resi disponibili per i moduli DMTCP, sia di altre funzionalità di controllo (ad esempio la verifica che lo strumento sia disponibile per l'esecuzione corrente).

Capitolo 5

ULPM: User Level Process Migration

5.1 Introduzione

Lo strumento di Checkpoint/Restart DMTCP si è rivelato un eccellente strumento atto a fornire un meccanismo di tolleranza ai guasti ed un'ottimo punto di partenza per lo studio e lo sviluppo di ULPM, *User Level Process Migration*, uno strumento implementato interamente a Livello Utente che permette la migrazione di processi. Dalla documentazione fornita insieme a DMTCP si può infatti notare che lo strumento è stato pensato anche per permettere la migrazione di processi. Tuttavia il suo funzionamento non consente di eseguire questa operazione in maniera atomica. L'utente che volesse utilizzare DMTCP per migrare una applicazione tra due macchine connesse in rete dovrebbe eseguire i seguenti passaggi:

1. Avviare l'applicazione attraverso dmtcp_checkpoint;
2. Richiedere la creazione dei file di checkpoint tramite dmtcp_coordinator;
3. Terminare il processo;
4. Copiare i file di checkpoint nella macchina destinata ad eseguire il processo;

5. Ripristinare l'esecuzione del processo attraverso dmtcp_restart.

Questi passaggi non avvengono in maniera automatica in DMTCP, inoltre potrebbero presentarsi alcuni inconvenienti. Notiamo infatti che la creazione dei file di *checkpoint* non è subordinata alla terminazione del processo. Al momento del ripristino è quindi possibile che l'applicazione riprenda ad eseguire operazioni che sono già state eseguite precedentemente nell'intervallo di tempo compreso tra il *checkpoint* e l'arresto del processo. Inoltre potrebbero verificarsi situazioni in cui due istanze dello stesso processo sono contemporaneamente in esecuzione su due macchine differenti. Queste situazioni potrebbero non essere desiderabili in una ottica di migrazione.

ULPM intende aggiungere alcune nuove funzionalità a DMTCP: rendere l'arresto del processo contestuale al momento del *checkpoint*, implementare un servizio di comunicazione di rete per la migrazione dei dati e riattivare il processo nella macchina remota. Gli strumenti di C/R e di trasferimento dati vengono quindi riuniti in un'unica funzionalità al fine di rendere la migrazione una procedura automatica e disponibile a richiesta dell'utente.

5.2 Panoramica dello strumento

ULPM è una applicazione che utilizza le funzionalità offerte da DMTCP per implementare interamente a Livello Utente un servizio di migrazione di processi. La tipologia di migrazione della memoria è di tipo *eager (all)* in quanto subordinata alla creazione del file di *checkpoint*. Questa tipologia di migrazione, come descritto nel relativo capitolo, è tipica delle implementazioni che fanno utilizzo di strumenti C/R. ULPM è stato realizzato in linguaggio C e le applicazioni che lo compongono sono rilasciate con licenza *GNU Public License v3*.

ULPM è composta da due applicazioni:

ulpm Si tratta dell'elemento principale. Si occupa di creare un ambiente idoneo alla migrazione sia per l'applicazione in uscita che per l'applicazione in entrata.

ulpm-command Si tratta dell'applicazione destinata ad essere di interfaccia tra l'utente ed ulpm. Attraverso questo strumento l'utente può richiedere l'esecuzione della funzionalità di migrazione.

Utilizzando il comando `./ulpm -h` si ottiene un elenco minimale delle opzioni possibili:

```
Usage #1: ./ulpm [OPTION] COMMAND [args]
Run user COMMAND in ulpm environment.

OPTION:
 -d dir, --dir-ckpt-files dir  set process checkpoint directory
                               (default: ./ulpm_send/)

Usage #2: ./ulpm [OPTION] ...
Let ulpm ready for accepting and run a remote process.

OPTION:
 -s, --save-ckpt-files         keep checkpoint files after restore
 -d dir, --dir-ckpt-files dir  set temp directory for incoming process
                               (default: ./ulpm_recv/)
 -p num, --port num            set the network TCP port (default: 3490)
 -h, --help                    show this help message
 -v, --verbose                 show additional output
 -V, --version                 show version information and exit
```

Si nota immediatamente la doppia modalità di utilizzo: se viene dato come parametro un programma ulpm si preoccupa di preparare l'ambiente di esecuzione per esso ed eseguirlo, altrimenti si prepara ad accogliere il *checkpoint* da un altra macchina della rete. In entrambe le situazioni è possibile scegliere (utilizzando l'opzione `-d <path>`) in quale cartella memorizzare i file temporanei prodotti da DMTCP. Nella seconda modalità, quella in cui resta in attesa del *checkpoint* del processo, è possibile impostare alcune opzioni

come il salvataggio dei file di *checkpoint* (opzione `-s`) oppure impostare alcuni parametri della comunicazione di rete.

Analogamente con `ulpm-command -h` otteniamo le seguenti informazioni:

```
Usage: ./ulpm-command [OPTION]
Migrate a process in ulpm environment to a remote host.

OPTION:
 -q, --quit                    close existing coordinator
 -s, --save-ckpt-files         keep checkpoint files after migration
 -d dir, --dir-ckpt-files dir  set directory with local process checkpoint
                               (default: ./ulpm_send/)
 -n addr, --net-address addr   set the remote host for migrate process
                               (default: 127.0.0.1)
 -p num, --port num            set the network TCP port (default: 3490)
 -h, --help                    show this help message
 -v, --verbose                 show additional output
 -V, --version                 show version information and exit
```

Questa applicazione consente di eseguire la migrazione attraverso la creazione dei file di *checkpoint* ed il loro trasferimento in rete. Anche in questa situazione è possibile configurare alcuni parametri della connessione e richiedere il salvataggio dei file di *checkpoint*. Inoltre viene fornita la possibilità di eseguire `ulpm-command -q` per arrestare l'esecuzione di tutti i processi in esecuzione in ulpm nella macchina in cui viene eseguito.

5.3 Dettagli implementativi di ULPM

Per poter utilizzare le funzionalità di *checkpoint* fornite da DMTCP è necessario che l'applicazione sia eseguita precedentemente tramite dmtcp_checkpoint. Allo stesso tempo è necessario che ad avvenuto *checkpoint* il programma fornito come parametro venga terminato. Per fare questo si è scelto di utilizzare gli *hooks* "*DMTCP events*" per il momento che segue l'esecuzione del

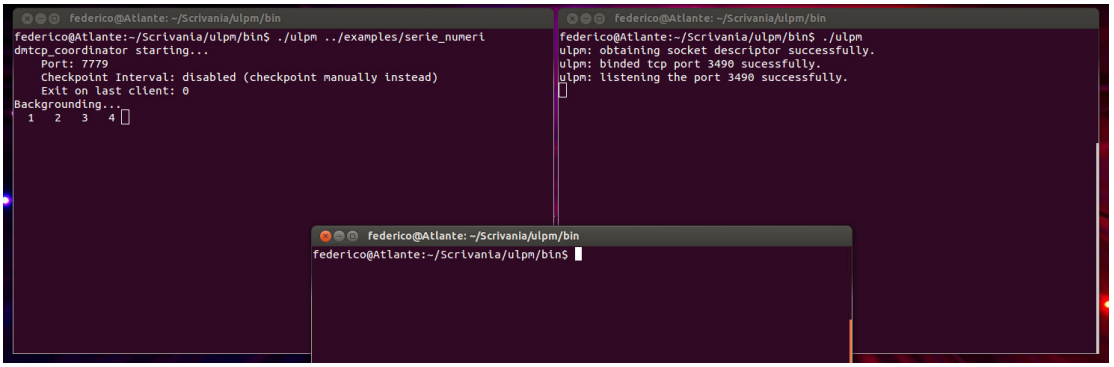

Figura 5.1: Stampa di una serie numerica prima della migrazione

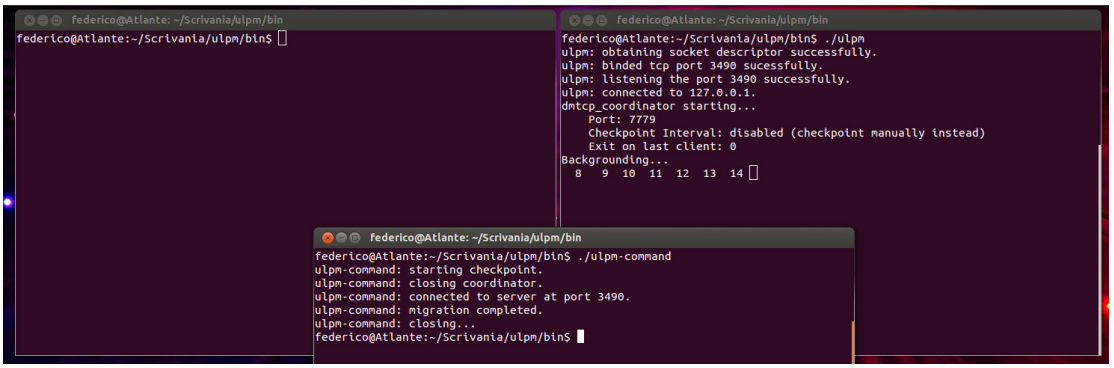

Figura 5.2: Stampa di una serie numerica dopo la migrazione

checkpoint. Tuttavia per poter utilizzare gli *hooks* è necessario che ulpm sia eseguito all'interno di dmtcp_checkpoint.

Questa situazione viene risolta con i seguenti passaggi riassunti in pseudo-codice:

```
if (dmtcpIsEnabled()) {
   dmtcpInstallHooks(pre, post, restart);
   pid = fork();

   if (pid == 0)
      eseguo_il_processo_utente();
```

```
   else
      wait(NULL);
}
else {
   eseguo_dmtcp_coordinator();
   pid = fork();

   if (pid == 0)
      rieseguo_ulpm_in_dmtcp();
   else {
      wait(NULL);
      ricostruisco_il_terminale();
   }
}
```

Si possono notare le funzioni *dmtcpIsEnabled()* e *dmtcpInstallHooks(pre, post, restart)*. Queste funzioni sono fornite da "*DMTCPAware*" e consentono rispettivamente di verificare se un processo è eseguito tramite di DMTCP e di installare gli hooks per gli eventi gestiti (prima del *checkpoint*, dopo il *checkpoint* e dopo il *restart*). In questo caso *pre()* e *restart()* sono funzioni segnaposto ma che di fatto non eseguono codice. La funzione post() invece si occupa di eseguire *_exit(0)* in modo da terminare il processo successivamente la scrittura del file di *checkpoint*. Purtroppo in questa forma di uscita da DMTCP possono verificarsi delle situazioni in cui il terminale risulti "sporco" e non correttamente utilizzabile. La soluzione scelta è di mantenere una ultima istanza di ULPM disponibile prima della chiusura dell'applicazione con lo scopo di ricostruire il terminale dell'utente.

Nel caso in cui invece si utilizzi ULPM per accogliere una applicazione, il funzionamento riassunto nello pseudo-codice seguente:

```
attendo_checkpoint();
ricevo_checkpoint();
```

```
pid = fork();

if (pid == 0)
   restart_checkpoint();
else {
   wait(NULL);
   ricostruisco_il_terminale();
}
```

Anche in questo caso è necessario mantenere attiva una istanza di ULPM, qui con un duplice scopo: oltre alla pulizia del terminale può essere richiesto di eseguire nuovamente il *checkpoint* dell'applicazione ed una nuova migrazione. Con questa istanza di ULPM è possibile procedere con la migrazione del processo tra nodi della rete senza limite di volte.

5.4 Comunicazione di rete

Affinché le istanze di ULPM possano comunicare attraverso la rete è stato un semplice protocollo di comunicazione. Avendo a disposizione i *checkpoint* sotto forma di file la comunicazione di rete è limitata al trasferimento di questi da un nodo all'altro. I dati vengono pacchettizzati in blocchi di dimensione 512 byte ai quali viene inserito in testa un *header* di 6 byte per informazioni di stato.

Il protocollo utilizza la seguente struttura di pacchetto, definito `ulpm_packethead` per l'*header* ed `ulpm_packet` per il pacchetto completo:

```
typedef struct _ulpm_packethead {
   char cmd;
   int len;
} ulpm_packethead;

typedef char* ulpm_packet;
```

Il valore `len` indica la dimensione del messaggio (riferito in byte, valore compreso tra 0 e 512) mentre `cmd` si riferisce alla tipologia di messaggio inviato, il quale può assumere determinati valori di riferimento per indicare la tipologia di messaggio.

L'insieme di tutto il protocollo di comunicazione è stato costruito utilizzando le implementazioni dei seguenti prototipi di funzione:

```
int ulpm_send(int sockfd, ulpm_packet p);
ulpm_packet ulpm_recv(int sockfd);
ulpm_packet ulpm_message(char cmd, int len, char *buf);
int ulpm_sendfile(int sockfd, char *fs_name);
int ulpm_recvfile(int sockfd, char *fs_name);
int ulpm_sendfilename(int sockfd, char *filename);
char* ulpm_recvfilename(int sockfd);
int ulpm_send_ack(int sockfd);
int ulpm_recv_ack(int sockfd);
int ulpm_send_filecount(int sockfd, int count);
int ulpm_recv_filecount(int sockfd);
```

Queste funzioni vengono quindi utilizzate dalle due istanze remote di ULPM per comunicare i file corrispondenti ai *checkpoint* dei programmi. Le funzioni implementate sono definite *ulpm_process_out()* per la comunicazione in uscita ed *ulpm_process_in()* per la comunicazione in entrata. La loro implementazione è riportata qui sinteticamente per mostrare il loro funzionamento:

```
int ulpm_process_out(int sockfd, char *path, char *filename)
{
   char *fs_name;

   fs_name = make_string("%s%s", path, filename);
   ulpm_sendfilename(sockfd, filename);
   ulpm_recv_ack(sockfd);
```

```
   ulpm_sendfile(sockfd, fs_name);
   ulpm_recv_ack(sockfd);
   free(fs_name);

   return 0;
}

char* ulpm_process_in(int sockfd, char *path)
{
   char *filename;
   char *fs_name;

   filename = ulpm_recvfilename(sockfd);
   fs_name = make_string("%s%s", path, filename);
   free(filename);
   ulpm_send_ack(sockfd);
   ulpm_recvfile(sockfd, fs_name);
   ulpm_send_ack(sockfd);

   return fs_name;
}
```

5.5 Esperimenti e risultati conseguiti

Dallo studio e dagli esperimenti condotti con ULPM si è verificato il corretto funzionamento nel Sistema Operativo GNU/Linux. Per la parte di test si è fatto utilizzo della distribuzione Ubuntu 13.04 (Kernel Linux 3.8.0-3) e di computer con architettura 32 e 64 bit. Le applicazioni prese in esame sono sia quelle rilasciate per eseguire i test e disponibili nel pacchetto di DMTCP che altre realizzate per verificarne il funzionamento con socket, file, pid e *fork()*. Ulteriori test sono stati condotti con le applicazioni `top` e

`more`. I risultati sono stati positivi ed hanno confermato il comportamento descritto. La migrazione è stata provata sia verso la stessa macchina che una macchina differente posta in rete locale. Il cambio di architettura (32/64 bit) ha mostrato alcune difficoltà, rendendo impossibile ripristinare l'esecuzione dei processi. Questo errore è imputabile alla differente dimensione dei registri del processore e quindi alla non compatibilità dei processi tra le due macchine. Ad eccezione di questi casi il *restart* avviene in modo corretto.

I primi esperimenti di funzionamento sono stati effettuati con piccole applicazioni di test (numero limitato di processi figli e bassa quantità di memoria allocata) per verificare la corretta migrazione tra due macchine connesse in rete locale, operazione che solitamente richiede tra 0,5 e 0,6 secondi. Tra i principali fattori che influenzano questo valore vi sono la potenza di calcolo delle macchine utilizzate, la quantità e la tipologia di altri processi in esecuzione e la velocità di trasmissione dati della rete. Le ragioni di possibili rallentamenti sono dovuti anche al tempo necessario a DMTCP ad estrarre gli stati dei processi, copiare la memoria e scrivere i dati su file (quindi anche ai tempi di accesso al disco), sia alla conseguente quantità di dati da comunicare ed eventuale congestione della rete. Questo tempo può anche variare sulla base della dimensione della memoria allocata e dalla dimensione della genealogia generata dal processo.

Successivi test eseguiti hanno fatto uso di *benchmark* per la valutazione delle prestazioni del sistema. Lo strumento impiegato per questo scopo è `lmbench` v. `3.0-a9` (`http://sourceforge.net/projects/lmbench/`) rilasciato con licenza GNU GPL2. Qui riportati vi sono i risultati ottenuti:

```
Processor, Processes - times in microseconds
------------------------------------------------------------------------
Host             Mhz null null      open slct sig  sig  fork exec sh
                     call  I/O stat clos TCP  inst hndl proc proc proc
-------------  ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
Macchina Reale 2840 0.06 0.15 0.59 1.47 3.85 0.20 1.15 308. 1156 2785
ULPM + DMTCP   2827 0.20 0.20 1.01 1.97 4.98 0.23 1.19 6834 134K 107K
```

```
Basic integer operations - times in nanoseconds
-----------------------------------------------
Host             intgr intgr  intgr  intgr  intgr
                 bit   add    mul    div    mod
--------------  ------ ------ ------ ------ ------
Macchina Reale  0.3600 0.1800 1.0600 8.0300 8.9100
ULPM            0.3600 0.1800 1.0900 8.0300 8.9700

Basic uint64 operations - times in nanoseconds
----------------------------------------------
Host            int64  int64  int64  int64  int64
                bit    add    mul    div    mod
--------------  ------ ------ ------ ------ ------
Macchina Reale   0.360        2.2900   16.9   21.7
ULPM             0.360        2.3100   17.9   21.8

Basic float operations - times in nanoseconds
---------------------------------------------
Host             float  float  float  float
                 add    mul    div    bogo
--------------  ------ ------ ------ ------
Macchina Reale  1.0600 1.7800 8.5300 8.5200
ULPM            1.0700 1.8200 8.6600 8.5200

Basic double operations - times in nanoseconds
----------------------------------------------
Host             double double double double
                 add    mul    div    bogo
--------------  ------  ------ ------ ------
Macchina Reale  1.0600 1.7700 8.5300 8.4700
ULPM            1.0700 1.8000 8.6300 8.5400
```

Context switching - times in microseconds

Host	2p/0K ctxsw	2p/16K ctxsw	2p/64K ctxsw	8p/16K ctxsw	8p/64K ctxsw	16p/16K ctxsw	16p/64K ctxsw
Macchina Reale	10.2	14.9	24.4	16.4	27.6	16.5	28.2
ULPM	12.8	17.5	28.5	19.1	30.9	19.3	31.2

Local Communication latencies in microseconds

Host	2p/0K ctxsw	Pipe	AF UNIX	UDP	RPC/ UDP	TCP	RPC/ TCP conn
Macchina Reale	10.2	22.5	27.2	39.8	42.1		26.
ULPM	12.8	19.7	12.0	32.0	47.1		100.

File & VM system latencies in microseconds

Host	0K File Create	0K File Delete	10K File Create	10K File Delete	Mmap Latency	Prot Fault	Page Fault	100fd selct
Macchina Reale	12.6	21.7	445.2	30.9	9359.0	0.233	1.14190	1.339
ULPM	12.9	18.6	431.6	14.0	7619.0	0.212	0.97080	2.929

Local Communication bandwidths in MB/s

Host	Pipe	AF UNIX	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write
Macchina Reale	1763	7871	3830	3734.2	7138.9	3738.5	3035.4	7519	4480.
ULPM	7857	7703	3020	3681.9	7504.8	3724.9	3002.4	7184	4531.

I test sono stati eseguiti più volte al fine di minimizzare eventuali errori nella raccolta dei dati, i risultati ottenuti si sono discostati di poche unità da ogni esecuzione presa in esame.

In questi dati possiamo evidenziare un rallentamento considerevole in *fork*, *exec* ed *sh* tra i valori raccolti con esecuzione pura in macchina reale ed esecuzione in ULPM. Questo peggioramento può essere dovuto alla routine eseguita da DMTCP al momento di creazione di un nuovo processo, con seguente inserimento in esso del *thread* di gestione. Questa operazione viene eseguita ogni volta per ogni processo discendente creato, rendendo questi valori necessariamente maggiori in confronto con quelli ottenuti nella macchina reale.

Possiamo anche notare che le prestazioni di operazioni aritmetiche e di calcolo sono tendenzialmente uguali in quanto non vengono influenzate da alcuna traduzione del codice macchina. Il codice eseguito è esattamente quello ottenuto dalla compilazione, con sola interposizione delle System Call. Otteniamo infatti valori di prestazione leggermente inferiori in alcuni test che fanno riferimento ad alcune di queste.

Un aspetto interessante è che in alcune System Call l'esecuzione è persino più rapida rispetto l'esecuzione su macchina reale. Un comportamento analogo è stato evidenziato anche in altri contesti [1]. Sebbene questo aspetto richieda di essere approfondito ulteriormente, è possibile notare dal codice delle funzioni di *wrap* alle System Call in DMTCP che alcune di queste vengono modificate. Ad esempio si può notare che l'esecuzione di *pipe()* viene convertita in "`socketpair(AF_UNIX, SOCK_STREAM, 0, fds);`" e queste tipologie di conversioni sembrano sufficienti a fornire il miglioramento di prestazioni misurato. Altri parametri che mostrano un miglioramento si possono individuare dai rispettivi valori misurati, ad esempio nella latenza inferiore con *Unix Domain Socket* (*AF UNIX*), socket UDP, cancellazione di file e nella maggiore velocità di comunicazione delle *pipe*.

[1] `http://wiki.virtualsquare.org/wiki/index.php/System_Call_Interposition:_how_to_implement_virtualization#Basic_Performance_Evaluation`

In conclusione è possibile affermare che un parziale rallentamento di esecuzione dei processi può essere osservato in ULPM, in confronto con l'esecuzione pura su macchina reale, causato prevalentemente dalle funzioni di *wrap* alle System Call. In alcuni casi queste funzioni possono essere utilizzate per convertire le System Call nelle analoghe più performanti. Altri rallentamenti riguardano invece il comportamento complessivo e sono causati dal tempo di migrazione, dalla scrittura dei *checkpoint* e dalle fasi di inizializzazione. In assenza di migrazione, un processo eseguito in ULPM offre prestazioni circa equivalenti a quelle offerte nell'esecuzione pura, con un minimo *overhead* dovuto al *wrapping* delle System Call.

Grafici ottenuti dall'analisi delle prestazioni

Questi grafici sono stati ottenuti tramite `lmbench` e forniscono una panoramica sulle prestazioni di alcuni degli aspetti testati.

I grafici posti a sinistra sono stati ottenuti da esecuzione su macchina reale, quelli a destra invece da esecuzione in ULPM.

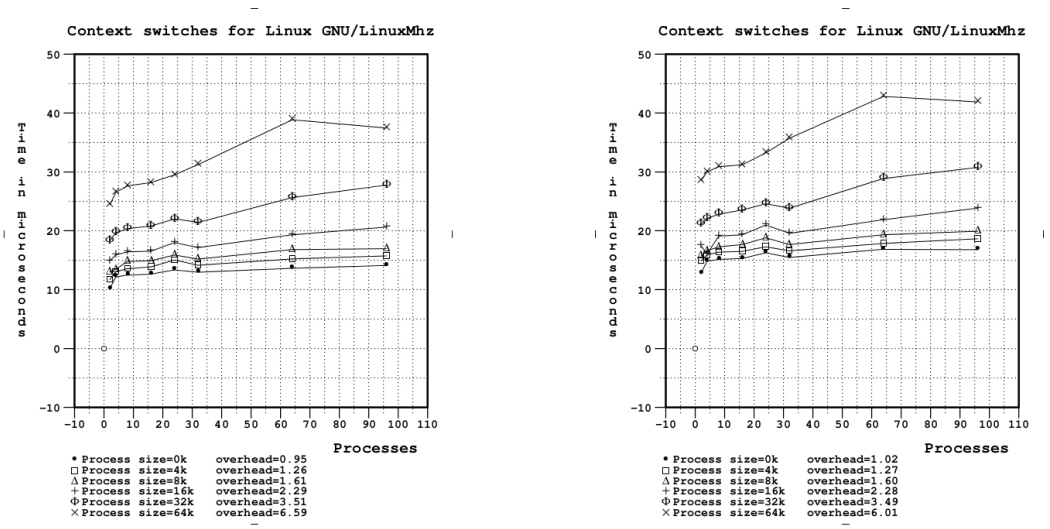

Figura 5.3: Confronto sul tempo impiegato nel *Context Switching* di processo

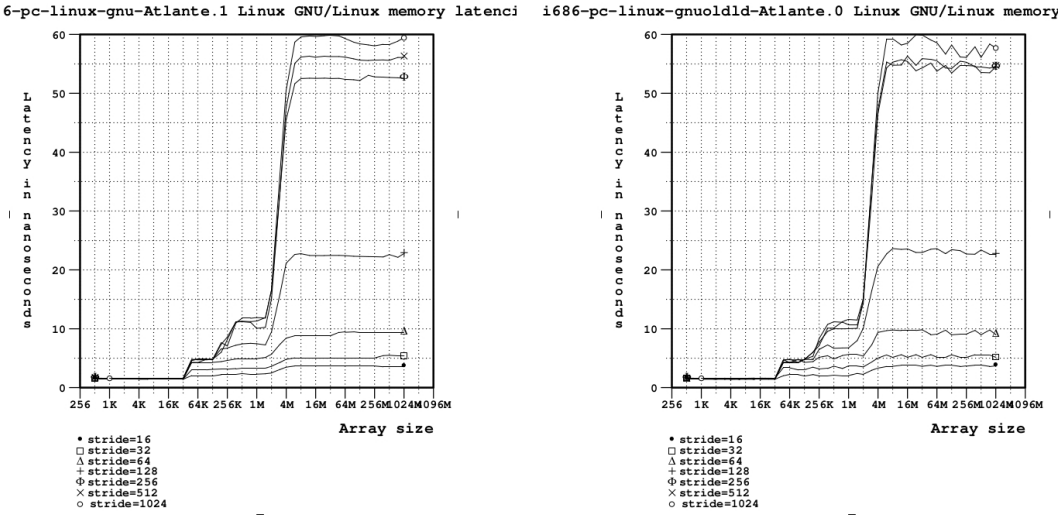

Figura 5.4: Confronto sulla latenza di accesso alla memoria

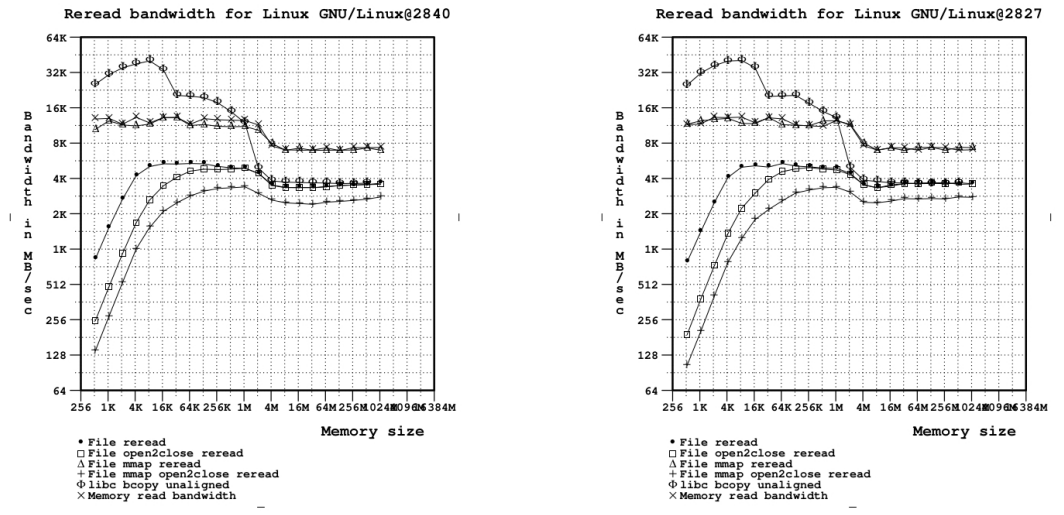

Figura 5.5: Confronto sulla velocità di rilettura di dati

Conclusioni e futuri sviluppi

Lo studio e la ricerca nella migrazione dei processi ha interessato la comunità scientifica per diversi decenni. I vantaggi derivati dal corretto impiego di questa tecnica sono molteplici e possono offrire meccanismi di tolleranza ai guasti, al bilanciamento del carico, alla mobilità delle applicazioni ed alla creazione di un ambiente unico distribuito per gli utenti e per le applicazioni. L'implementazione interna al Sistema Operativo si è dimostrata un'operazione possibile ma anche molto articolata e difficile da mantenere nel tempo. Nel corso degli ultimi anni i progressi conseguiti in tema di virtualizzazione e di Macchine Virtuali ha dato una nuova prospettiva per l'impiego di questa tecnica che si dimostra particolarmente efficace qualora la migrazione interessi una Macchina Virtuale completa. Infatti, le ragioni che sono alla base delle difficoltà di migrazione sono strettamente correlate ai legami che si instaurano tra applicazione ed ambiente di esecuzione, vincolando il processo all'ambiente locale originario con proprietà, come il pid o i privilegi di esecuzione, e servizi, come per il file system o per le connessioni di rete.

Lo studio si è quindi fatto carico di valutare la possibilità di impiegare una minima interfaccia di virtualizzazione in grado di porsi da intermediaria nel fornire e gestire i servizi offerti dal Sistema Operativo, in particolare le System Call. Si è cercato quindi di provare se la migrazione di un processo insieme ad un'interfaccia intermedia al Sistema Operativo possa semplificare l'operazione ed anche non richiedere espliciti interventi (integrazioni o modifiche) a Livello Kernel. L'indagine è proseguita attraverso lo studio e la valutazione di strumenti di *Checkpoint/Restore*, individuando tra le possibilità

esistenti DMTCP, uno strumento sviluppato interamente a Livello Utente e facente utilizzo di *wrapping* alle System Call per intercettarne l'esecuzione e modificarne parzialmente il funzionamento. Attraverso DMTCP si è infine giunti allo sviluppo di ULPM nel tentativo di realizzare uno strumento per la migrazione di processi, sviluppato anch'esso interamente a Livello Utente.

Gli esperimenti condotti sull'attuale implementazione di ULPM hanno verificato tale possibilità, ciononostante è bene precisare che lo sviluppo può essere esteso e perfezionato. A tale proposito si ricorda che un aspetto cruciale riguarda la velocità di migrazione. Il salvataggio dei dati di *checkpoint* su disco produce uno dei rallentamenti principali che potrebbe essere risolto attraverso l'impiego di RAM disk o soluzioni analoghe. Altra possibilità potrebbe essere velocizzare la migrazione del processo eseguendo il trasferimento dei dati man mano questi vengono estratti, operazione possibile attraverso la modifica di MTCP, lo strumento utilizzato internamente a DMTCP per svolgere l'operazione di *checkpoint*.

Un altro importante miglioramento in ULPM può derivare dal solo utilizzo di MTCP e dalla riscrittura o fusione della parte che riguarda DMTCP. Sebbene lo strumento `dmtcp_coordinator` sia stato impiegato per la realizzazione di ULPM, allo stesso tempo ne sono stati constatati limiti e problematiche derivanti dal suo utilizzo, come la non possibilità di scegliere il nome dei file di *checkpoint*, il vincolo di *checkpoint* simultaneo a tutti i processi gestiti ed altre difficoltà legate alla migrazione di file aperti (all'interno del file di *checkpoint* il percorso viene salvato con riferimento assoluto) e connessioni (il protocollo di ripristino richiede che entrambi gli estremi della comunicazione usino DMTCP, tuttavia dalla versione 3.5 del Kernel Linux si potrebbe impiegare `TCP_REPAIR` [2] [3]). Inoltre, il *thread* di gestione interno al processo sembra sufficiente ad eseguire il *checkpoint*, `ulpm-command` potrebbe allora comunicare direttamente con questo senza un ulteriore passaggio intermedio verso `dmtcp_coordinator`.

[2] `http://kernelnewbies.org/Linux_3.5`

[3] `http://lwn.net/Articles/495304/`

Le possibilità di estendere ULPM possono riguardare anche i servizi forniti dallo strumento, come il bilanciamento automatico del carico di lavoro tra più nodi connessi, la realizzazione di una comunità virtuale di utenti che possono condividere tra loro risorse di calcolo e il potenziamento degli strumenti di virtualizzazione a livello di System Call per fornire una migrazione più trasparente possibile, anche valutando l'impiego di altre tecniche come l'utilizzo di *ptrace()*.

I risultati conseguiti si sono rivelati positivi ed incoraggianti verso una sperimentazione dettagliata e completa in materia di migrazione di processi, approfondendo l'impiego di questa tecnica con delle minime interfacce di virtualizzazione.

Bibliografia

[1] Jason Ansel, Kapil Arya and Gene Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, 2009.

[2] Gene Cooperman. Architecture of DMTCP. Documentation on the internals of DMTCP. Gennaio 2012.

[3] Michael Rieker, Jason Ansel and Gene Cooperman. Transparent User-Level Checkpointing for the Native POSIX Thread Library for Linux. In *The 2006 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2006.

[4] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler and Songnian Zhou. Process migration. *ACM Computing Surveys*, Settembre, 2000.

[5] Vatsal Shah, Viral Kapadia. Load Balancing by Process Migration in Distributed Operating System. *International Journal of Soft Computing and Engineering*, Marzo 2012.

[6] M. Rasit Eskicioglu. Design Issues of Process Migration Facilities in Distributed Systems. In *Scheduling and Load Balancing in Parallel and Distributed Systems*, 1995.

[7] Nalini Vasudevan and Prasanna Venkatesh. Design and Implementation of a Process Migration System for the Linux Environment. In *3rd In-*

ternational Conference on Neural, Parallel and Scientific Computations, 2006.

[8] Rajkumar Buyya, Toni Cortes and Hai Jin. Single System Image. In *International Journal of High Performance Computing Applications*, Maggio 2001.

[9] George Coulouris, Jean Dollimore and Tim Kindberg. Distributed Systems: Concepts and Design. Edition 4, 2005.

[10] Harvey M. Deitel, David R. Choffnes and Paul J. Deitel. Sistemi operativi. 2005.

[11] Andrew S. Tanenbaum and Maarten Van Steen. Sistemi distribuiti. Principi e paradigmi. 2007.

[12] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne. Operating System Concepts. Eight Edition, 2008

[13] Deo Prakash Vidyarthi, Biplab Kumer Sarker, Anil Kumar Tripathi and Laurence Tianruo Yang. Scheduling in Distributed Computing Systems. Analysis, Design and Models. 2008.

[14] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design and Implementation*. Volume 2, 2005.

[15] Yunfa Li, Wanqing Li and Congfeng Jiang. A Survey of Virtual Machine System: Current Technology and Future Trends. In *Proceedings of the 2010 Third International Symposium on Electronic Commerce and Security*. 2010.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt and Andrew Warfield. Xen and the

Art of Virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003.

[17] Renzo Davoli and Michael Goldweber. Virtual Square: Users, Programmers and Developer Guide. 2011.

[18] Jonathan M. Smith. A survey of process migration mechanisms. In *ACM SIGOPS Operating Systems Review*, 1988.

[19] Jonathan M. Smith and John Ioannidis. Implementing remote fork() with checkpoint/restart. In *IEEE Technical Committee on Operating Systems Newsletter*, 1989.

Ringraziamenti

Ringrazio tutti coloro che mi hanno supportato durante il percorso di Laurea; ringrazio il relatore, professor Renzo Davoli, che mi ha seguito nello svolgimento di questa tesi. Desidero in particolare ringraziare i miei genitori, Mauro e Lorella, sui quali ho potuto sempre fare affidamento; la mia famiglia per essere sempre stata al mio fianco; i miei amici più cari per i bei ricordi ed i momenti felici trascorsi insieme.