Alma Mater Studiorum · Università di Bologna

**facoltà di scienze matematiche, fisiche e naturali**
**Corso di Laurea Triennale in Informatica**

# Esperimenti di virtualizzazione parziale remota

**Tesi di Laurea in Sistemi Operativi**

**Relatore:**
**Chiar.mo Prof.**
**Renzo Davoli**

**Presentata da:**
**Dennis Olivetti**

**Sessione II**
**Anno Accademico 2010-2011**

*Alla mia famiglia, amici e colleghi...*

# Introduzione

Quando usiamo un programma siamo abituati ad eseguirlo sul nostro computer; quando lo stesso programma ha bisogno di accedere ad informazioni presenti su un altro computer siamo abituati ad eseguirlo sulla nuova macchina, oppure ad utilizzare le più diverse tecnologie per riuscire a rendere disponibili i dati al programma.

Questo lavoro di tesi presenta una nuova tecnologia, grazie alla quale è possibile eseguire un programma parzialmente su una macchina e parzialmente su un'altra. I confini tra una parte e l'altra di programma dipendono proprio dal tipo di operazioni che comprendono.

Al momento esistono le più diverse tecnologie per raggiungere l'obiettivo descritto, ad esempio per "spostare" su un altro computer l'esecuzione di una parte di programma riguardante la rete è possibile utilizzare un proxy, facendovi passare attraverso tutte le connessioni generate dal programma. Esistono vari tipi di proxy, differenti in base all'ambito di applicazione; tuttavia ci sono operazioni difficilmente fattibili tramite un proxy. Per fornire a un programma l'accesso al file system di un differente computer, invece, sono disponibili altre tecnologie, di natura completamente diversa tra loro.

Con questo lavoro di tesi si intende mostrare che la tecnologia proposta è in grado di unificare la soluzione dei problemi sopra esposti.

Le tecnologie attualmente esistenti per risolvere alcuni problemi, oltre ad essere specifiche per un certo ambito, in certi casi non permettono di fare tutto ciò che ci si potrebbe aspettare da loro; ad esempio coi proxy si ha difficoltà nel gestire le connessioni in ingresso, e coi file remoti si ha difficoltà

a limitarne la visibilità al solo programma interessato.

La tecnologia creata, per raggiungere l'obiettivo, sfrutta l'esecuzione remota delle system call chiamate dal programma, va quindi ad eseguire veramente in remoto la parte di un programma facente parte di un certo ambito. Questo permette sia di remotizzare "qualsiasi cosa" nello stesso modo, sia di dare la visibilità di ciò al singolo programma scelto; i restanti programmi, per i quali non vi è l'esecuzione remota delle system call, non subiranno nessun effetto collaterale: non potranno vedere ciò che vede il programma scelto.

L'esecuzione di system call remote ha dei limiti: nel caso di varie chiamate consecutive, da parte dello stesso programma, si hanno seri problemi di latenza, verranno però analizzati questi casi fornendo possibili soluzioni; solitamente varie chiamate consecutive hanno una motivazione alle spalle, provando quindi a prevedere questi casi, è possibile aggiungere del codice che limita il problema.

L'obiettivo per un futuro migliore è quello di eseguire un programma su un computer potente situato in un luogo arbitrariamente lontano, dando però al programma l'impressione di essere in esecuzione sul proprio computer, mostrandogli tutte le risorse.

# Indice

# Elenco delle figure

# Elenco delle tabelle

# Capitolo 1

# Prerequisiti

## 1.1 Metodi di virtualizzazione

### 1.1.1 Introduzione

Al giorno d'oggi la parola virtuale ha assunto innumerevoli significati e sfaccettature. Dal dizionario apprendiamo che la parola virtuale significa: "Che esiste solo in potenza ma non è ancora in atto."[Zan04] In realtà ogni ramo scientifico possiede la sua particolare accezione di questo termine: in filosofia è sinonimo di potenziale, ovvero qualcosa che può accadere, nei videogiochi si parla di realtà virtuale come sinonimo di simulazione, in fisica come l'evoluzione immaginata di un sistema. Anche in informatica, soprattutto negli ultimi anni, abbiamo assistito alla esplosione dei campi di applicazione della virtualizzazione. Ma cosa si intende principalmente per virtualizzazione in informatica?

Immaginiamo di avere uno strumento necessario alla soluzione di un problema; questo strumento avrà vari modi per essere utilizzato, ovvero un insieme di operazioni eseguibili mediante quell'oggetto. Possiamo chiamare questo insieme "interfaccia" dell'oggetto. Uno strumento "virtuale" non è altro che qualcosa che fornisce la stessa interfaccia dell'originale e che si comporta allo stesso modo. L'oggetto potrebbe avere forma e funzionamento

interno completamente diverso, l'importante è che dal punto di vista dell'utilizzatore non vi sia differenza, ovvero che il nuovo oggetto possa essere usato in sostituzione all'originale.

In informatica possiamo collegare la parola virtualizzazione a quella di emulazione, facendo invece distinzione tra virtualità e simulazione. Virtualità e simulazione sono concetti ben distinti: un oggetto virtuale può essere effettivamente sostituito ad un oggetto reale, un simulatore no, ad esempio un simulatore di volo non potrebbe essere sostituito a un aereo vero. La differenza sostanziale è che un simulatore si comporta solo a livello macroscopico come la realtà, creando qualcosa di approssimato, dando all'utilizzatore una vaga illusione di fare qualcosa che realmente non viene fatto. Virtualità ed emulazione sono invece due concetti legati tra loro: emulare un oggetto significa crearne uno nuovo, diverso dal precedente, ma che fa le stesse cose. Quindi l'emulazione serve ad implementare la virtualità.

Esistono vari livelli di virtualizzazione: si può virtualizzare un intero sistema (tramite macchine virtuali), o fare virtualizzazioni parziali (tramite syscall virtual machine o injection su librerie).

### 1.1.2 Macchine virtuali di sistema

Per macchina virtuale di sistema si intende un software in grado di emulare l'hardware di una macchina reale; su di essa viene messo in esecuzione un vero sistema operativo, kernel e programmi. Il sistema operativo ospite è convinto di essere in esecuzione su dell'hardware vero, non sente la differenza. Il software della macchina virtuale va ad implementare non solo il processore della macchina da emulare, ma anche tutti i componenti aggiuntivi necessari al funzionamento di una macchina vera. Il sistema, ad esempio, vedrà una vera scheda di rete che non è altro che l'interfaccia mostrata dalla macchina virtuale di una vera scheda; la macchina andrà poi a fare, via software, le operazioni necessarie per convertire ciò che il sistema software crede siano segnali elettrici in una qualche modalità possibile, per dare l'accesso alla rete alla macchina ospite.

Solo in alcuni casi è necessario emulare l'intero hardware della macchina reale, come ad esempio quando si vuole eseguire il sistema operativo su una diversa architettura; quando invece l'architettura della macchina reale e di quella emulata coincidono, la necessità dipende proprio dal set di istruzioni da virtualizzare: non è necessario emulare il processore quando le istruzioni per accedere alle risorse non sono eseguibili in user-mode.[PG74]

L'architettura dei processori x86 ad esempio è nata con istruzioni che non soddisfano i requisiti di virtualizzazione veloce, sono state aggiunte successivamente nuove istruzioni per ovviare a questo problema. Grazie a queste il software di una macchina virtuale va ad utilizzare il processore vero invece che implementarlo, portando la velocità della macchina virtuale a circa la stessa della macchina fisica.

Grazie a questa ottimizzazione, che ha portato la velocità a un livello accettabile, le macchine virtuali hanno avuto un notevole utilizzo, tanto da poter considerare questo come un vero e proprio "boom"; tra i più importanti utilizzi ricordiamo i principali:

- a livello individuale, eseguire contemporaneamente diversi sistemi operativi sullo stesso computer, utile, ad esempio, nel caso si vogliano eseguire contemporaneamente applicazioni non supportate da entrambi i sistemi;
- a livello aziendale, eseguire vari server, che hanno bisogno ognuno di una specifica configurazioni di sistema, su una singola macchina, risparmiando sia nel costo dell'hardware che nei consumi elettrici;
- a livello di fornitori di servizi di hosting, la possibilità di affittare ai clienti una quantità di macchine ben superiore a quelle realmente possedute, anche in questo caso abbassando decisamente i costi. Un altro vantaggio in questo caso è che se un cliente avesse bisogno di maggiori prestazioni non è necessario intervenire sull'hardware, ma sarebbe sufficiente cambiare la configurazione della macchina virtuale.

Esempi famosi di software per fare questo tipo di virtualizzazione sono:

- Qemu: sistema open source che non sfrutta le istruzioni hardware apposite per la virtualizzazione “veloce”, ma che riesce ad emulare una grande varietà di architetture;
- Kvm: sistema open source per fare virtualizzazione assistita dal kernel, basato su qemu [AKL07];
- VirtualBox: sviluppato da Oracle, parzialmente free;
- VMWare: simile al precedente, ma proprietario;
- Xen: sistema di paravirtualizzazione, che per funzionare agisce direttamente sull’hardware invece che dialogare con un sistema operativo sottostante.

A giudicare dall’ampio panorama [vms] offerto dai software di virtualizzazione notiamo quanto sia importante questo settore del software.

Figura 1.1: kvm mentre esegue un live-cd di kubuntu.

### 1.1.3 Macchine virtuali di applicazione

Per application virtual machine si intende un software necessario per l'esecuzione di un programma scritto in un linguaggio non comprensibile dalla macchina sottostante; per funzionare si pone tra il sistema operativo e il programma. Può funzionare principalmente in due modi:

- interprete: per ogni istruzione del programma viene eseguito un set di istruzioni del software della macchina virtuale, è il metodo più lento, il codice di un ciclo, ad esempio, deve essere interpretato ogni volta;

- compilatore: all'avvio, o durante l'esecuzione del programma, vengono compilate al momento alcune sue parti (compilazione just-in-time), questo è solitamente il metodo migliore, perché, oltre al fatto di dover operare solo una volta per ogni istruzione, ha anche il pregio di poter ottimizzare il codice creato in base alla macchina sottostante.

Queste macchine vengono solitamente usate per creare applicazioni eseguibili ovunque, senza la necessità di ricompilare: il codice scritto dal programmatore viene compilato generando un linguaggio generico, non eseguibile su nessuna macchina reale. Viene creata una macchina virtuale su ogni sistema reale su cui si vuole eseguire il proprio programma; ognuna di queste macchine avrà un codice, diverso dalle altre, necessario per fare da tramite tra il linguaggio generico e il linguaggio della macchina sottostante.

Esempi di linguaggi che utilizzano questo tipo di macchine virtuali sono:

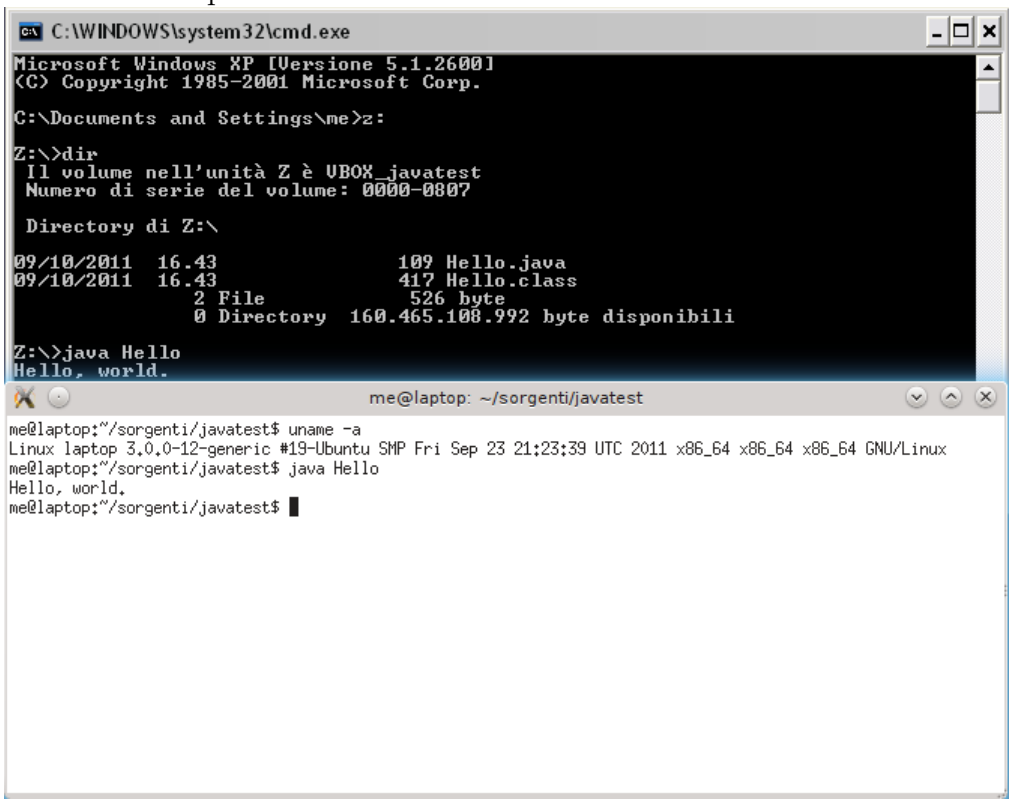
- Java: usato per eseguire la stessa applicazione su diverse architetture, sistemi operativi, cellulari. Compila immediatamente il codice, ottimizzandolo per la macchina sottostante. In alcuni casi le prestazioni risultano addirittura superiori confrontate con quelle di un programma compilato nativamente per la macchina ospitante, ma senza eccessive ottimizzazioni (è la situazione che si ha in tutti quei casi in cui venga fornito direttamente un eseguibile invece che il codice sorgente);

- Javascript: usato principalmente nelle pagine web, in passato basato su un interprete, solo recentemente basato su un compilatore;

- Python: usato per applicazioni desktop, basato anch'esso su un interprete in cui alcune parti si possono esplicitamente passare a un compilatore sfruttando librerie esterne.

Notiamo quindi che anche in questo settore le macchine virtuali sono diventate una base fondamentale nella costruzione di alcuni tra i più famosi prodotti presenti sul mercato.

Figura 1.2: Lo stesso programma java, compilato una volta e eseguito su diversi sistemi operativi



### 1.1.4 Virtualizzazione tramite libreria

Questo è il metodo più facile e veloce, ma meno sicuro, per fare virtualizzazione. Un'applicazione, solitamente, usa del codice contenuto al suo interno e del codice contenuto in librerie esterne; questo meccanismo è usato per fare in modo di non dover ricompilare tutti i programmi che usano una libreria nel caso questa venga aggiornata, e per risparmiare memoria nel caso due applicazioni utilizzino contemporaneamente la stessa libreria.

Per *virtualizzazione tramite libreria* si intende un livello software che si frapponga tra il programma e una libreria data, mendiante una nuova libreria che implementi la stessa interfaccia di quella precedente, intercettando le chiamate a funzione fatte dal programma e adattandole ai propri scopi.

Questo è il metodo di virtualizzazione che offre prestazioni maggiori, il motivo è che il "codice virtuale" è caricato nella stessa area di memoria del programma, quindi potrà accedere direttamente alla sua memoria senza nessun overhead, dato da chiamate di sistema o calcoli aggiuntivi. Per lo stesso motivo è anche il metodo meno sicuro: se il programma effettua una chiamata di funzione, al contrario di come normalmente avviene (cioè verso la libreria), direttamente verso il sistema, la virtualità non può funzionare; e la chiamata non può essere intercettata. Questo metodo quindi non può essere usato in qualsiasi situazione in cui non si ha fiducia (in termini di sicurezza) del programma virtualizzato.

Questo metodo è usato, ad esempio, nei casi in cui si vogliono modificare piccole funzionalità di qualche programma, senza modificarlo, soprattutto in ambienti in cui non si ha a disposizione il codice sorgente o non si vuole ricompilare. Ad esempio:

- patch non ufficiali a programmi proprietari: spesso su sistemi chiusi vengono eseguiti programmi chiusi, e qualsiasi modifica non autorizzata dal produttore viene spesso implementata ponendosi tra l'eseguibile e una libreria di sistema da lui usata;

- aggiunta di supporto a proxy per programmi che ne sono privi: modificare e ricompilare ogni applicazione richiederebbe troppo tempo;

- limitare funzionalità di un programma: negando l'esecuzione di alcune operazioni.

Riassumendo, questo metodo è utilizzato in tutti i casi in cui si vuole manipolare l'esecuzione di un programma di cui ci si fida, senza ridurne le prestazioni.

Figura 1.3: Virtualizzazione di una strcmp


### 1.1.5 Syscall virtual machine

#### Introduzione

Per Syscall Virtual Machine si intende un software in grado di porsi tra il sistema operativo e il programma da eseguire, intercettare le chiamate di sistema fatte da quest'ultimo e adattarle alle proprie necessità. Questo permette di estendere le funzionalità del sistema operativo senza bisogno di essere amministratori. Sulle SCVM è possibile eseguire solo programmi compilati per lo stesso sistema su cui è in esecuzione la macchina virtuale.

### User Mode Linux

Non è una vera e propria System Call Virtual Machine ma vi somiglia molto, sia per com'è implementata che per i risultati che fornisce.

UML[umlb] è in grado di eseguire un intero sistema operativo (Linux) come normale processo del sistema sottostante; in UML macchina virtuale e programma da eseguire coincidono, vengono aggiunte patch al kernel da eseguire per permettere due cose:

- rendere il tutto un comune eseguibile;
- modificare le operazioni di accesso all'hardware per fare in modo che sfruttino il sistema sottostante.

Questo sistema è utilizzato per vari motivi, tra i quali:

- si possono fare operazioni che sulla macchina ospitante si potrebbero fare solo da root;
- si possono provare programmi o kernel sperimentali senza paura di rovinare il sistema;
- si possono provare modifiche al kernel senza dover riavviare a ogni crash;
- si può debuggare il kernel come fosse un normale processo;
- si possono provare contemporaneamente differenti distribuzioni linux.

Le prestazioni con questo sistema calano del 20% circa.[umla]

### *mview

*mview[GGD08, umv, DG09] è una SCVM che crea un nuovo concetto di virtualità, quella parziale: un processo virtualizzato tramite *mview vedrà alcune parti del sistema reale e alcune parti del sistema virtuale.

E' implementata tramite un programma "monitor" che si pone tra i processi e il sistema, intercettando le syscall e passandole a dei sottomoduli che precedentemente hanno chiesto di reimplementarle.

*mview a grandi linee funziona in questo modo:

1. viene avviato un programma, la visione che ha del sistema in questo momento è quella reale;

2. viene caricato un modulo di *mview;

3. viene dato un comando di mount, che verrà intercettato dal modulo, il quale andrà a ridefinire alcune system call;

4. ora la visione del sistema del processo sarà reale per tutto tranne che per ciò che riguarda il mount precedente;

5. qualsiasi chiamata di sistema riguardante cio per cui è stato fatto il mount verrà gestita dal modulo anziché dal sistema operativo.

Le applicazioni di questo sistema sono molteplici, ad esempio:

- dare l'impressione a un processo di essere in esecuzione su un computer potentissimo che in realtà non si ha, fattibile intercettando gli accessi all'orario di sistema;

- fare mount di immagini di dischi senza bisogno di essere root, intercettando gli accessi a disco;

- rendere disponibili ramdisk a applicazioni in esecuzione in usermode e senza la necessità di privilegi di amministrazione;

- togliere accesso alla rete a un singolo processo;

- creare un device di rete virtuale, in usermode, che poi verrà implementato tramite normali funzioni di rete di libreria.

Usando questo sistema si ha una sicurezza ancora maggiore rispetto a un sistema operativo reale; prendiamo ad esempio una directory crittografata, una volta fatto il mount di quella directory l'amministratore di sistema potrà leggere i file dell'utente; con *mview invece solo il processo sul quale è stato fatto il mount vedrà questo nuovo mondo virtuale, escludendo anche root!

Parlando di performance bisogna dividere *mview in due implementazioni:

- umview, le cui prestazioni sono penalizzate dalle limitazioni di ptrace. Infatti questa system call usata da umview come da user-mode linux, è stata implementata per il debugging e non è ottimizzata per la virtualizzazione. VirtualSquare, il gruppo che ha realizzato umview, ha proposto nuove funzionalità per la system call ptrace per migliorare le prestazioni.

- kmview, implementato mediante utrace: eguaglia quasi le prestazioni di una macchina reale (perde solo il 5% senza fare mount), l'unico difetto è che utrace non è ancora stata accettata dagli sviluppatori del kernel Linux.

L'unico limite nell'utilizzo di *mview è la fantasia, perché praticamente permette di estendere a piacere le funzionalità del sistema operativo.

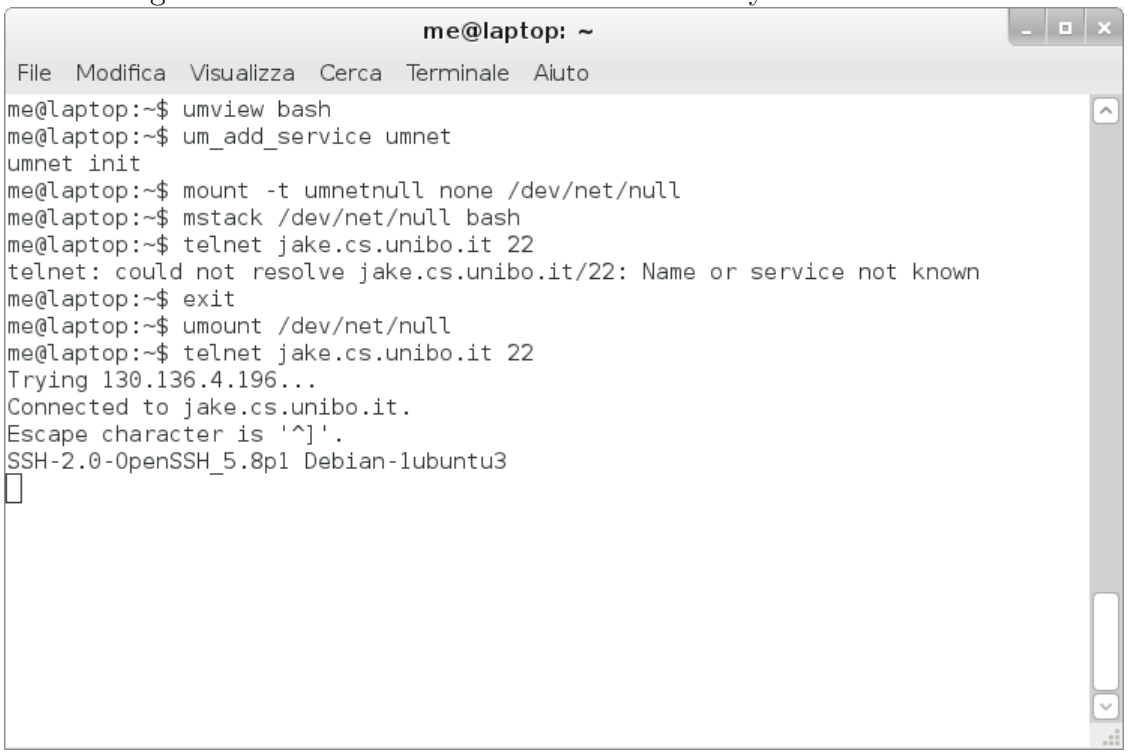
### 1.1.6 Conclusione

Sono state viste le più diverse tecnologie esistenti per fare virtualizzazione, ognuna col suo campo di applicazione; riassumendo si dividono in queste grandi categorie:

- Macchine virtuali di sistema: implementano via software una macchina reale ed eseguono al suo interno un vero sistema operativo;

- Macchine virtuali di applicazione: permettono di eseguire su diverse macchine reali un programma compilato una volta sola in un linguaggio di una macchina astratta;

Figura 1.4: umview al lavoro disattivando le syscall di rete




- Virtualizzazione tramite libreria: modificano piccole funzionalità di un programma senza necessità di modificare quest'ultimo;

- System Call Virtual Machine: estendono le funzionalità di un sistema ponendosi tra il kernel e i programmi in esecuzione, solitamente permettendo di fare cose che in un sistema reale solo l'amministratore può fare.

## 1.2 Remote procedure call

Le chiamate a procedura remota (RPC) servono a eseguire parti di programma (procedure) su un computer diverso da quello dov'è in esecuzione il programma stesso oppure per eseguire parti di programma, anche sulla stessa macchina, ma scritte in linguaggi diversi non compatibili tra loro.

Il programma che chiama una procedura remota dovrebbe accorgersi il meno possibile della differenza tra chiamate remote e locali, per fare in modo che sia così i software che permettono di fare RPC sono solitamente implementati in questo modo:

- al programma vengono forniti degli stub, ovvero un'interfaccia delle procedure disponibili, il programma farà chiamate a queste procedure come se fossero locali;
- negli stub, all'insaputa del programma, verranno fatte tutte le operazioni necessarie per dialogare con un server RPC;
- il server RPC implementa degli skeleton, ovvero del codice che si preoccupa di fare le chiamate alle procedure vere e proprie e si preoccupa poi di rimandare allo stub il risultato.

  I sistemi RPC sono usati in vari ambiti, ad esempio:

- aziende che vogliono fornire al pubblico alcune loro funzionalità;
- siti web scritti in linguaggi poco performanti che vogliono far fare i lavori più pesanti a linguaggi più performanti;
- sistemi distribuiti.

I problemi che si hanno principalmente coi sistemi RPC sono dati dai puntatori: bisogna definire ciò che è area di input o area di output, se un parametro è un'altra procedura servono sistemi in grado di fare una RPC inversa. Un altro problema comune è dato dal dover scegliere un encoding dei parametri adeguato per poterli interpretare su macchine diverse.[SS07]

I sistemi di RPC moderni vanno a dialogare tra loro basandosi su XML o altri linguaggi descrittivi: cercano di dare una descrizione dei parametri invece di supporre che sia il codice del client che quello del server sappiano già esattamente come sono fatti i parametri di una funzione.

Alcune ecnologie esistenti per fare RPC sono, ad esempio:

- SOAP: uno dei sistemi più usati, le chiamate vengono fatte mediante richieste HTTP scambiando dati in XML. Viene usato, ad esempio, da alcuni router per permettere l'autoconfigurazione delle porte da inoltrare a una macchina della rete;
- JSON-RPC: simile a SOAP ma la codifica viene fatta in un linguaggio simile a javascript;
- Java RMI: permette di fare chiamate di metodi remoti in java, la codifica dei parametri viene fatta mediante le funzioni di serializzazione già presenti in java.

Tutti questi sistemi sono perfetti per ambiti generali, ma non sono il meglio nel caso si voglia fare qualcosa di ben specifico, dove magari non è necessario scomodare un parser XML, e dove si ha bisogno di fare chiamate inverse senza poter mettere in esecuzione un server per parte.

Figura 1.5: esempio di dati scambiati tramite SOAP

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails xmlns="http://magazzino.example.com/ws">
      <productId>827635</productId>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>


<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetailsResponse xmlns="http://magazzino.example.com/ws">
      <getProductDetailsResult>
        <productName>Toptimate, set da 3 pezzi</productName>
        <productId>827635</productId>
        <description>Set di valigie; 3 pezzi; poliestere; nero.</description>
        <price>96.50</price>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```

## 1.3 Remote System Call

Per RSC si intende il fatto di eseguire alcune system call su un sistema diverso dal proprio, dando l'illusione al programma in esecuzione sul sistema locale di essere invece in esecuzione sul sistema remoto.

### 1.3.1 Sistemi esistenti di RSC

Un solo sistema esistente di remote system call è diventato famoso [Cac02], più per com'è stato applicato che per l'idea in se.

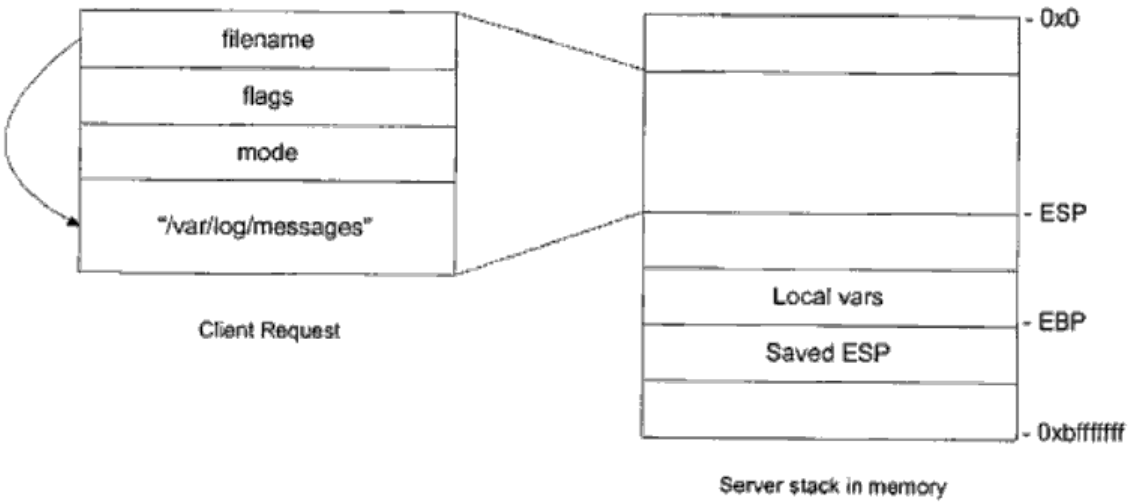
L'ambito di applicazione è la sicurezza: un nuovo modo di sfruttare vulnerabilità di tipo buffer overflow. L'idea è semplice: invece che eseguire sulla macchina vittima remota un'intera shell collegata ad un server, viene mandato in esecuzione un server RSC, il più piccolo possibile, per fare in modo che il relativo codice sia abbastanza compatto da poter essere usato sfruttando la vulnerabilità.

Il server funziona in questo modo:

- manda al client il proprio indirizzo dello stack;
- il client adatta tutti i parametri della syscall facendo in modo che, una volta messi sullo stack del server, gli indirizzi risultino corretti;
- il client manda al server l'area di memoria appena creata, il quale la copia sullo stack e fa la chiamata al sistema;
- il server manda al client il risultato della syscall e l'area di memoria modificata.

Questo sistema però, si nota subito che ha un problema: non è in grado di funzionare se le system call da eseguire sono bloccanti nel caso in cui ne venga fatta più di una contemporaneamente. Tra l'altro l'idea alla base di questo sistema RSC è, negli USA, protetta da brevetto[Cac03].

Figura 1.6: schema di funzionamento del sistema RSC esistente



# Capitolo 2

# Soluzioni attuali a problemi esistenti

## 2.1 Introduzione

Spesso, utilizzando un computer, si ha bisogno di utilizzare delle risorse remote. L'accesso a queste risorse è solitamente necessario quando non si hanno i privilegi necessari per eseguire remotamente il programma che deve accedere a tali risorse, oppure quando la macchina remota non dispone di abbastanza potenza o memoria per eseguire il programma in questione. Si vuole eseguire il programma illudendolo che quelle risorse sono disponibili localmente.

Un esempio può essere accedere a delle directory condivise in una rete, accedere a propri dati su un computer remoto, oppure utilizzare un proxy per fare delle connessioni con un indirizzo di rete diverso dal proprio. Tra le possibili applicazioni delle system call remote, verranno analizzati i casi appena descritti e alcuni problemi delle soluzioni al momento esistenti.

## 2.2 Rete

Nell'ambito delle reti le principali operazioni che si vuole riuscire a compiere possono essere, ad esempio:

- fare connessioni da un indirizzo IP che non è il proprio;
- ricevere connessioni da un indirizzo IP che non è il proprio.

Per permettere di fare questo sono stati inventati vari sistemi. Ne verranno analizzati due in particolare, Proxy e NAT.

### 2.2.1 Proxy

Un proxy è un software che fa da tramite tra un client e un server, il suo funzionamento può essere riassunto così: un server resta in attesa di connessioni, successivamente un client si collega ad esso dicendo un indirizzo verso cui effettuare una connessione, infine il server effettua una connessione verso l'indirizzo specificato. Da quel momento in poi il server proxy farà da "ponte" per tutti i dati in arrivo da entrambe le parti, cioè invierà al server i dati in arrivo dal client e viceversa.

Questa tecnologia viene attualmente applicata con successo per la soluzione di numerosi problemi. Il limite di questa tecnica consiste nella necessità che l'applicazione sia stata progettata per supportare l'utilizzo del servizio proxy.

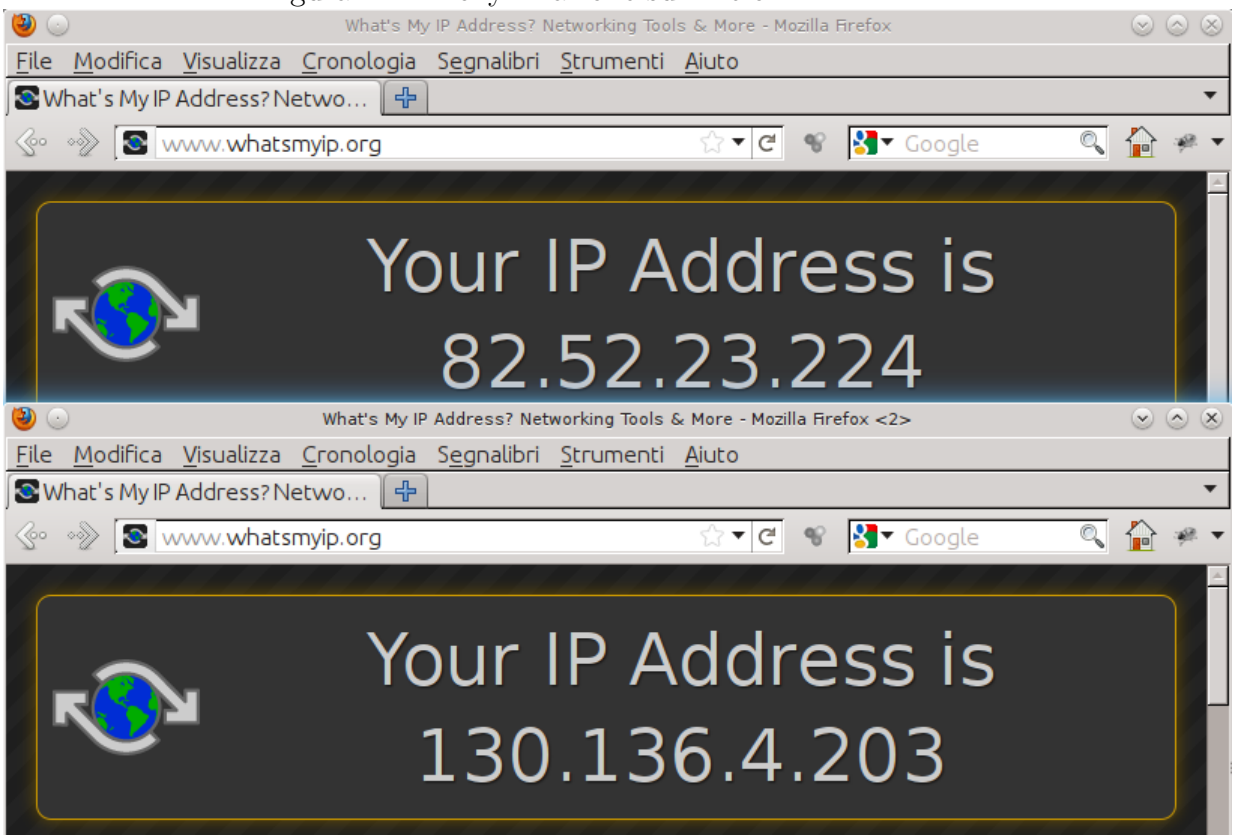
Esistono tool in grado di ovviare a questo problema, uno di questi è "tsocks". Tsocks intercetta le chiamate di rete fatte da un programma e le modifica in base alle proprie esigenze, altro non è che un sistema di virtualizzazione tramite libreria. Ha vari problemi[tso], eccone alcuni:

- per com'è implementato non è in grado di gestire chiamate asincrone;
- non è in grado di accettare connessioni in ingresso tramite proxy;

- non è in grado di intercettare chiamate a funzioni di rete quando queste vengono fatte direttamente al sistema operativo invece che a librerie di sistema.

La soluzione di questi problemi, facendo virtualizzazione tramite libreria, potrebbe risultare molto difficile.

Figura 2.1: Proxy in azione su Firefox




### 2.2.2 NAT

Il nat è un meccanismo solitamente implementato nei router per permettere di fare connessioni attraverso di esso. Viene usata una tabella contenente le connessioni attualmente in corso e alcune regole per gestire le nuove connessioni; tutti i dati in arrivo vengono mandati all'indirizzo giusto in base al contenuto della tabella.

Questo sistema potrebbe essere usato per risolvere entrambi i problemi sopra citati, ma questa soluzione è affetta da due problemi:

- solitamente è necessario essere amministratori per poter modificare le regole di configurazione di NAT;
- tutte le applicazioni in esecuzione su una macchina sarebbero affette da questa modifica.

Un altro difetto, non un problema vero e proprio, è che questo sistema viene solitamente usato nelle reti locali, volendo usare questo sistema in remoto potrebbe essere necessario creare una rete privata virtuale tra i due computer coinvolti.

## 2.3 File System

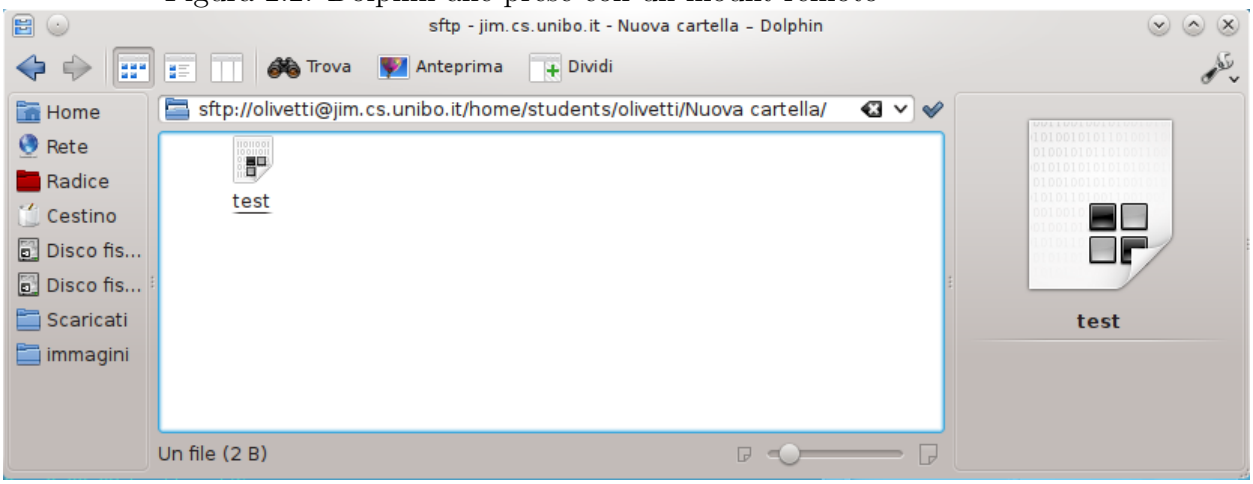
Per accedere a file su un computer diverso dal proprio esistono moltissimi sistemi, eccone alcuni:

- HTTP: può essere usato, fra le tante cose, anche per dare accesso a file o directory, solitamente non permette di effettuare un mount del disco remoto;
- FTP: probabilmente uno dei sistemi più antichi per accedere in remoto a un file system, con lo stesso problema di http;
- SMB: sistema sviluppato da IBM, e portato avanti da Microsoft per la condivisione di file, directory e altre risorse su una rete locale. Ne esiste un'implementazione libera chiamata Samba;
- SSH: è possibile, tramite fuse[ssh], utilizzare ssh per fare mount di directory remote, o accedervi come fosse un ftp crittografato tramite SFTP.

Sono tutti ottimi sistemi, le prestazioni sono ottime, ma hanno solo un difetto: una volta resa accessibile una directory remota sul proprio computer

locale, tutti i programmi in esecuzione sul computer locale saranno in grado di accedervi, lo stesso discorso vale per l'amministratore di sistema.

Figura 2.2: Dolphin alle prese con un mount remoto




## 2.4 Conclusione

Dall'analisi risulta che ci sono alcune funzionalità che non sono previste nelle tecnologie analizzate, ad esempio:

- non esistono sistemi in grado di ricevere/effettuare connessioni utilizzando un indirizzo differente dal proprio, da programmi non scritti appositamente, senza la necessità di essere amministratori del sistema remoto;
- non esistono sistemi in grado di effettuare un mount di directory remote senza che l'intero sistema ne sia coinvolto.

Nell'ambito di questo lavoro di tesi verrà proposta nei prossimi capitoli una metodologia capace di unificare le funzionalità degli strumenti analizzati in modo che possano venir superate le limitazioni sopra indicate. Non è una mera evoluzione di quanto è stato esposto in questo capitolo, la metodologia è innovativa, basata su codice e idee originali.

# Capitolo 3

# User Mode Remote System Call: umrsc

## 3.1 Introduzione

Umrsc è un software che permette di eseguire un insieme di system call su un computer diverso dal proprio, dando al programma in esecuzione l'illusione di essere sul computer remoto per le operazioni riguardanti le system call che stanno nell'insieme, ma sul computer locale per tutto il resto.

Permette ad esempio di scegliere le operazioni riguardanti la rete e avviare un programma sul proprio computer, dandogli l'illusione di usare l'indirizzo di rete appartenente a un sistema situato dall'altra parte del mondo; oppure dare l'illusione a un programma di essere in esecuzione su una directory della propria macchina, quando in realtà quella directory è situata su un altro computer della propria rete.

Permette di risolvere i problemi di alcune tecnologie esistenti spiegate nei precedenti capitoli; non solo, riesce a risolvere tutti i problemi con un'unica soluzione, basta elencare l'insieme di operazioni riguardanti il problema da risolvere e la soluzione automaticamente esisterà.

Il funzionamento è semplice: si avvia una parte di umrsc sul proprio computer, un'altra parte di umrsc su un computer remoto, poi si avvia il pro-

gramma, tutte le chiamate di sistema scelte verranno mandate da una parte di umrsc all'altra, eseguite in remoto restituendo al chiamante il risultato.

## 3.2 Uso dello stack TCP-IP remoto tramite umrsc

In precedenza si è parlato dei problemi che si ha con tecnologie esistenti riguardanti la rete nel fare certe operazioni, eccone un riassunto:

1. tramite proxy, problemi:

   - c'è bisogno di modificare un programma per riuscire a farne passare correttamente le connessioni attraverso un proxy;
   - tecnologie esistenti permettono solo parzialmente di far passare connessioni di un programma attraverso un proxy in caso di programmi non scritti appositamente.

2. tramite NAT, problemi:

   - solitamente è necessario essere amministratori di sistema;
   - tutti i programmi subiscono la nuova configurazione. Si riesce a fare una configurazione scendendo in dettaglio sul formato dei pacchetti di rete, ma ad esempio se un altro programma sulla stessa macchina vorrà fare una connessione verso uno stesso indirizzo a cui si collega il programma "bersaglio", e l'indirizzo è stato ridirezionato tramite NAT, la regola varrà anche per lui. Un altro caso è quando il programma vuole accettare connessioni, quando terminerà, se un programma eseguito successivamente a lui vorrà accettare connessioni sulla stessa porta, subirà le stesse regole del programma precedente.

Grazie alle tecnologie su cui si basa umrsc i problemi riguardanti il NAT vengono risolti automaticamente, umrsc si basa su una syscall virtual machine, questo porta ai seguenti benefici:

- il processo eseguito nella SCVM vede un mondo differente dal resto dei programmi, lui vedrà un mondo con una certa rete, gli altri programmi vedranno invece la rete originale;

- eseguire una SCVM è permesso a un normale utente.

I problemi riguardanti i proxy, invece, hanno soluzione grazie all'architettura di umrsc. Da notare che non è necessario modificare il programma, le operazioni di rete vengono automaticamente intercettate. Confrontiamo quindi umrsc con una tecnologia vista in precedenza facente cose analoghe, tsocks:

- viene intercettato il cuore della chiamata, quella fatta al sistema, non verso una libreria, quindi non c'è il caso in cui la chiamata possa sfuggire, tsocks intercetta tramite libreria, portando con se tutti i problemi legati a quel tipo di virtualizzazione;

- grazie al fatto che le operazioni vengono veramente eseguite in remoto, si riesce a gestire anche le chiamate asincrone, tsocks non ne è in grado perchè se ad esempio deve gestire una connect asincrona, dovendola fare verso un proxy invece che verso l'host vero non ha modo di inviare o ricevere i dati necessari al dialogo col proxy, perché renderebbe bloccante la chiamata. Umrsc andrebbe invece a fare una connect verso il vero host, perché la connessione al "proxy" è fatta al livello sottostante, quello che fa dialogare le due parti del programma;

- si riescono ad accettare connessioni remote in ingresso, tsocks non ne è in grado sia perché non implementa questa operazione, sia perchè i proxy server stessi solitamente non la implementano.

## 3.3 Accesso a un File System remoto tramite umrsc

Si è parlato di varie tecnologie per fare mount di directory remote, tutte con lo stesso problema: tutti i programmi in esecuzione sul proprio sistema vedono la nuova directory. Questo problema è risolto, come nel caso delle connessioni di rete, grazie al fatto che solo i programmi in esecuzione su una stessa system call virtual machine vedono il mondo che gli si vuol far vedere.

Questo è appunto un caso in cui problemi riguardanti situazioni completamente diverse vengono risolti una volta per tutte.

## 3.4 Altre applicazioni di umrsc

Questa tecnologia si può utilizzare in tutti i casi in cui si vuole dare a un processo la visione di una risorsa di un sistema remoto, aggiungendola al proprio sistema o sostituendone una già presente. Tutto questo si riesce a fare senza fatica, perché umrsc va a fornire automaticamente la soluzione alla parte comune di tutti i problemi riguardanti questa situazione. Si è visto appunto che problemi completamente diversi, quali proxy o mount remoti, vengono risolti mediante la stessa tecnologia, ponendosi tra il programma e il sistema operativo, "rubandogli" le chiamate e mandandole in esecuzione su un altro sistema.

## 3.5 Esempi pratici di utilizzo di umrsc

Verranno ora forniti esempi pratici dell'utilizzo di umrsc. Iniziamo col vedere le varie directory componenti il progetto:

- cplpc: codice facente rsc;
- local: codice destinato alla macchina locale;
- remote: codice destinato alla macchina remota;

- pipe2socket: codice necessario a cplpc.

E' presente uno script che prepara il codice al suo utilizzo, e si avvia tramite il comando:

```
$ sh prepare.sh
```

Ora è necessario compilare il codice:

```
$ cd local
$ make
...
$ cd ../remote
$ make
...
$ cd ..
```

E ora una breve spiegazione dei file di configurazione, ogni sottomodulo necessita di far parte di una directory contenente anche altri file indispensabili, ne è fornito un esempio:

```
$ cat local/conf_example/rsc_config
socket_on 1
disk_on 1
local_path /unreal
remote_path /
REMOTE read
REMOTE write
REMOTE close
REMOTE fcntl
REMOTE open
REMOTE lstat
REMOTE readlink
REMOTE getdents
REMOTE access
REMOTE lseek
REMOTE mkdir
```

```
REMOTE rmdir
REMOTE lchown
REMOTE chmod
REMOTE unlink
REMOTE fsync
REMOTE fdatasync
REMOTE link
REMOTE symlink
REMOTE pread
REMOTE pwrite
REMOTE utimes
REMOTE statfs
REMOTE rename
REMOTE socket
REMOTE bind
REMOTE connect
REMOTE listen
REMOTE accept
REMOTE getsockname
REMOTE getpeername
REMOTE socketpair
REMOTE sendto
REMOTE recvfrom
REMOTE shutdown
REMOTE getsockopt
REMOTE setsockopt
REMOTE ioctl
REMOTE sendmsg
REMOTE recvmsg
REMOTE uname
REMOTE event_subscribe
```

Ecco una descrizione dei vari token:

- socket_on 1/0: attiva/disattiva la remotizzazione delle system call riguardanti i socket;
- disk_on 1/0: attiva/disattiva la remotizzazione delle system call riguardanti il file system;
- local_path: in caso di remotizzazione delle syscall del file system dice dove verrà montata la directory;
- remote_path: consente di scegliere di quale path remota verrà fatto il mount;
- REMOTE nomesyscall: specifica che la syscall in questione dovrà essere remotizzata.

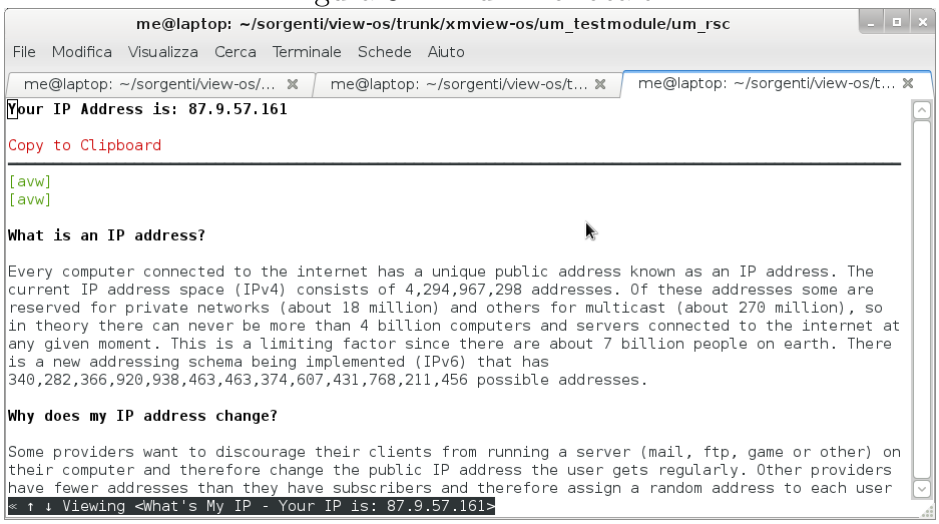
E' ora possibile avviare il tutto, iniziamo avviando umview:

```
$ umview bash
```

In questo momento tutte le connessioni e gli accessi al disco saranno diretti, controlliamo l'indirizzo IP attraverso un sito web:

```
$ w3m whatsmyip.net
```

Figura 3.1: Indirizzo locale



Carichiamo ora il modulo di umrsc:

```
$ um_add_service ./path/modulo/umrsc
```

Avviamo client e server RPC,

sul server:

```
$ cd remote
$ sh start.sh
Usage sh start.sh port_number
$ sh start.sh 1234
```

e sul client:

```
$ cd local/conf_example/
$ sh start.sh
Usage sh start.sh ip_address port_number
$ sh start.sh 130.136.4.203 1234
```

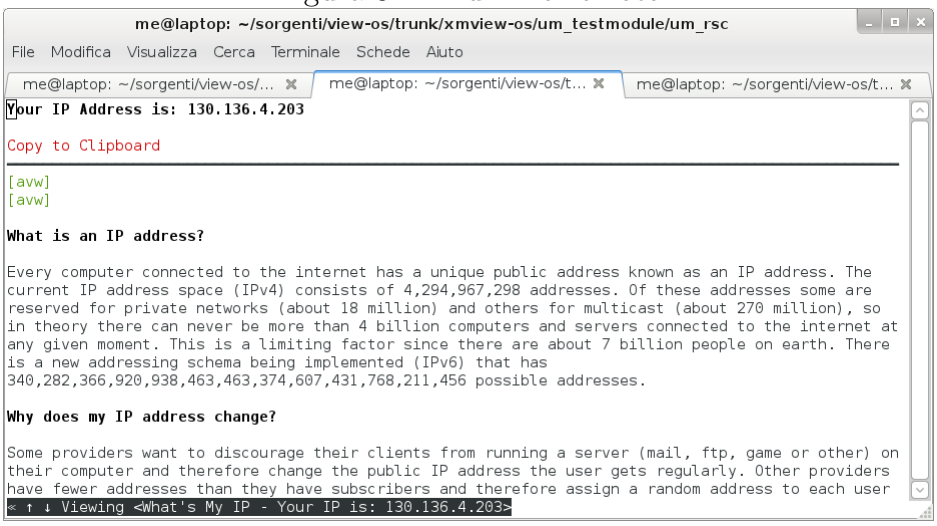
Effettuiamo il mount della configurazione:

```
$ mount -t rsc_cplpc none local/conf_example/
```

Controlliamo nuovamente l'indirizzo IP sullo stesso sito web:

```
$ w3m whatsmyip.net
```

Figura 3.2: Indirizzo remoto

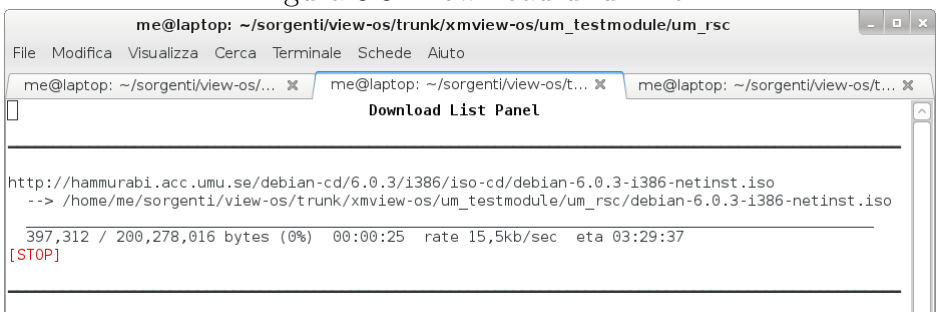



L'indirizzo è proprio quello della macchina remota.

Proviamo a scaricare un file:

```
$ w3m http://cdimage.debian.org/debian-cd/6.0.3/i386/iso-cd/debian-6.0.3-i386-net
```

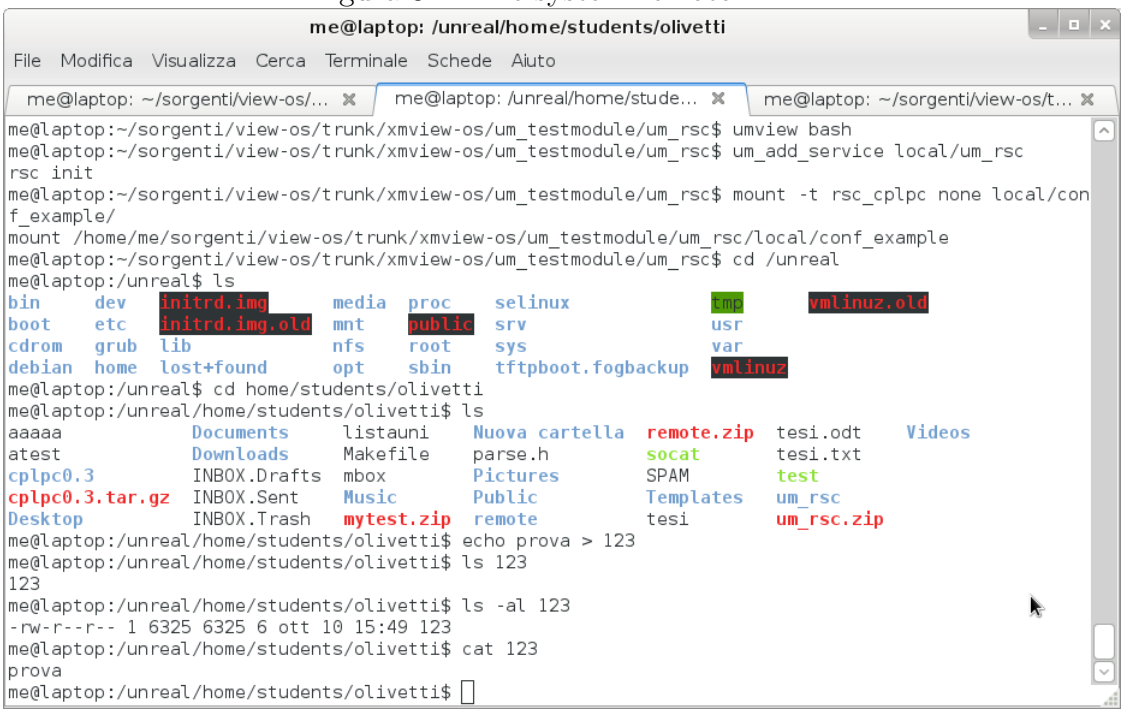
Figura 3.3: Download di un file




Si può notare che la velocità è molto bassa, verrà analizzato questo problema nei prossimi capitoli.

Proviamo ora a navigare attraverso il file system remoto:

Figura 3.4: File system remoto



Proviamo ora l'uname:

```
$ uname -a
Linux jim 2.6.38-11-generic #50-Ubuntu SMP Mon Sep 12 21:18:14 UTC
2011 i686 GNU/Linux
```

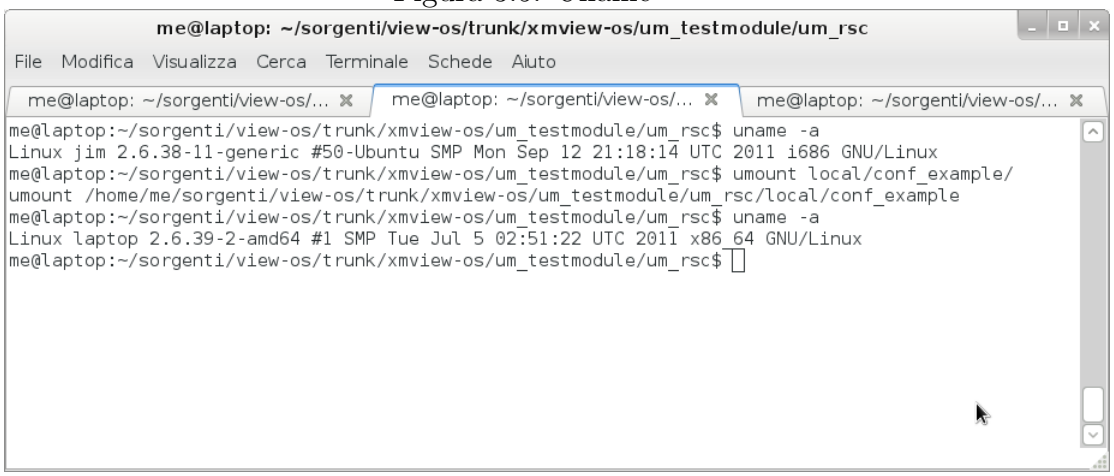
e ora effettuiamo l'umount della remotizzazione:

```
$ umount local/conf_example/
umount /home/me/sorgenti/view-os/trunk/xmview-os/um_testmodule/um_rsc/local/conf_ex
```

rieffettuiamo l'uname, questa volta sarà quello della macchina locale:

```
$ uname -a
Linux laptop 2.6.39-2-amd64 #1 SMP Tue Jul 5 02:51:22 UTC 2011 x86_64
GNU/Linux
```

Figura 3.5: Uname



# Capitolo 4

# Architettura umrsc

## 4.1 Com'è fatto

In questo capitolo verrà fornita una spiegazione di com'è implementato umrsc, quali tecnologie utilizza e in quali modi è possibile aggiungere nuove funzionalità.

Umrsc è innanzitutto un plugin di *mview, quindi non ha bisogno di preoccuparsi di intercettare le chiamate di sistema, perché questa parte è data automaticamente dalla syscall virtual machine sottostante. Umrsc si preoccupa invece di caricare altri sottomoduli, che gestiscono ognuno un sottoinsieme di system call; si preoccupa poi, quando *mview avvisa umrsc che è necessario eseguire una system call, di scegliere il sottomodulo giusto per fargli eseguire la chiamata di sistema.

Oltre a questo si preoccupa di fare da tramite tra i sottomoduli e *mview per decidere quali syscall intercettare, quali mount chiedere alla scvm, quali eventi sono scattati.

## 4.2 Plugin di *mview

Per gestire le operazioni appena descritte vengono eseguite queste azioni:

1. viene definita una funzione per ogni possibile system call da virtualizzare;

2. viene detto a *mview che si vogliono gestire le system call per cui si è appena definita una funzione;

3. si attende che l'utente effettui un mount;

4. il mount viene intercettato e come parametro è presente una directory contenente file di configurazione e altri file necessari alla remotizzazione delle system call;

5. se necessario vengono chiesti dei mount a *mview;

6. viene letto il file di configurazione, per ogni syscall specificata nel file viene caricato un indirizzo di funzione da una libreria contenuta nella directory del mount;

7. per le successive system call specificate creanti un nuovo file descriptor viene modificato il risultato per riuscire a ricordarsi quale sottomodulo ha originato quel descrittore;

8. le restanti system call vengono rimandate alle funzioni specificate precedentemente modificando il file descriptor del parametro, se presente, calcolandone il valore originale.

## 4.3 Sottomoduli

Ogni sottomodulo è una libreria che esporta le funzioni specificate nel file di configurazione e un'altra funzione alla quale è possibile chiedere di essere notificati in caso di eventi accaduti al file descriptor, questa funzione è necessaria per il funzionamento di *mview stesso. Questa libreria verrà eseguita in remoto usando le funzioni che esporta tramite RPC.

## 4.4 Cplpc

### 4.4.1 Introduzione

Resta ora da descrivere come vengono inviate le system call da un computer all'altro. Per fare questo è stato utilizzato Cross Platform Local Procedure Call (cplpc): un sistema in grado di eseguire sulla propria macchina librerie compilate per differenti architetture, adattandone alcune parti alle proprie necessità.

Perché non è stato utilizzato un normale sistema di RPC? Perché si è ritenuto che scomodare le varie tecnologie usate nei comuni sistemi RPC era eccessivo: un server, un parser xml, eccetera. Oltre a questo rimaneva il problema di dover gestire le callback, non supportate dai comuni sistemi di RPC, e scrivere comunque del codice aggiuntivo per automatizzare il tutto. Cplpc invece fornisce quasi tutto ciò che in questo contesto è necessario.

### 4.4.2 Funzionamento di cplpc

Cplpc funziona in questo modo:

1. è necessario scrivere un file (un header formato C) contenente la lista di funzioni e relativi parametri presenti nella libreria che si vuole utilizzare;

2. cplpc genera quasi tutto il codice necessario per fare RPC delle funzioni contenute nella libreria;

3. il codice va completato inserendo le parti necessarie per fare wrap/unwrap dei parametri, da un lato è necessario serializzare i parametri inserendoli in un normale array, dall'altro è necessario fare l'operazione inversa, cplpc fornisce alcune funzioni per semplificare queste operazioni, utilizzandole si ha la gestione automatica dell'endianess;

4. compilando vengono generati stub e skeleton necessari per il funzionamento, stub e skeleton dialogano tramite named pipe.

### 4.4.3 Architettura di cplpc

Cplpc può essere diviso in due parti: codice generante e codice generato.

Il codice generante è implementato mediante un parser yacc, va a leggere l'header e per ogni prototipo di funzione letto genera il codice necessario.

Il codice generato è anch'esso diviso in due parti, la parte comune e la parte veramente generata.

La parte comune è il vero cuore di cplpc, funziona in questo modo:

1. quando tutto ha inizio, sullo skeleton viene avviato un pool di thread, necessari per poter eseguire più operazioni bloccanti contemporaneamente, i thread vengono addormentati su una variabile condizione;

2. quando una funzione dello stub viene chiamata, i parametri, serializzati, vengono inviati allo skeleton, e viene poi bloccato il thread chiamante in attesa che lo skeleton rimandi il risultato

3. lo skeleton legge i dati e aggiunge la chiamata in attesa d'esecuzione ad una lista, risveglia poi un thread lavoratore;

4. il thread fa l'unwrap dei parametri, esegue l'operazione, invia il risultato e torna a dormire.

La parte generata, invece, non è altro che una minima implementazione delle funzioni in cui vengono chiamate le funzioni di cplpc comuni dopo aver effettuato il wrap/unwrap dei parametri.

Stub e skeleton in realtà hanno la maggior parte del codice in comune, perché per gestire le callback senza aggiungere complessità è stato supposto che lo stub possa comportarsi da skeleton e viceversa. Un appunto: per gestire le callback è necessario fornire a priori un numero massimo di diversi puntatori a funzione che può assumere un singolo parametro di callback, questo è necessario perchè altrimenti, implementando nello skeleton un singolo stub di una funzione di callback, non sarebbe possibile capire quale funzione la libreria vuole chiamare nel caso che precedentemente siano stati passa-

ti differenti valori di puntatori a funzione per un singolo parametro di una funzione.

### 4.4.4 Adattamento alle necessità

Cplpc è stato pensato per dialogare tramite named pipe, per essere eseguito localmente; è stato scritto un miniprogramma client/server che fa da ponte tra due pipe remote in modo che possa essere utilizzato come normale sistema RPC.

## 4.5 Tipo di parametri, automatizzazione wrap/unwrap

Il problema principale nell'eseguire RPC è quello del wrap e unwrap dei parametri, soprattutto in linguaggi in cui i tipi sono più un'indicazione che una regola, come nel caso del C.

La maggior parte delle system call ad esempio ha come parametri dei puntatori ad aree di memoria, di cui guardando il singolo parametro non è possibile sapere la dimensione, nè tanto meno è possibile sapere automaticamente se è un'area di memoria contente valori di input o se è un'area di memoria dove verrà salvato un risultato, nel caso in cui sarà un risultato non si sa nemmeno quanto sarà grande quel risultato.

Per ovviare a questi problemi è stato fornito un meccanismo che semiautomatizza tutto il lavoro, è necessario dare una "descrizione" dei parametri e in base a questa i parametri verranno automaticamente impacchettati e spacchettati.

I tipi di parametro supportati nella descrizione sono suddivisi principalmente in queste categorie:

- numeri: i più semplici da gestire, vanno solo ricopiati;
- aree di memoria di input di dimensione fissa: hanno dimensione fissa o comunque calcolabile mediante gli altri parametri;

- aree di memoria di output: aree di memoria che possono essere ignorate fornendo i dati in input, ma che è necessario fornire assieme al risultato;
- stringhe: aree di memoria di lunghezza variabile delimitate da un terminatore.

Per le aree di memoria di input e output è possibile descriverne la dimensione in uno di questi modi:

- dimensione fissa: è necessario fornirne la dimensione, in byte;
- dimensione data da un parametro: è necessario specificare qual è il parametro;
- dimensione data dereferenziando un parametro: è necessario anche quì specificare qual è il parametro;
- dimensione data in input da un parametro e in output dal risultato: è necessario specificare il parametro.

Questi casi riescono a descrivere quasi tutte le system call; ne restano fuori davvero poche, quelle che come parametro hanno una struttura contenente puntatori ad altre aree di memoria, in questi casi è possibile fornire una completa funzione di wrap/unwrap.

Cplpc viene quindi supportato in maniera importante e significativa da questo sistema di semi-automatizzazione, le sue funzioni di serializzazione non fanno altro che chiamare le funzioni dell'aiutante passando gli stessi parametri e specificando di quale system call si tratta.

## 4.6 Come aggiungere funzionalità

Per aggiungere il supporto a system call che umrsc non è ancora in grado di gestire è necessario effettuare le seguenti operazioni:

1. aggiungere all'header dato in pasto al generatore di codice di cplpc il prototipo di funzione descrivente la system call;

2. far generare il nuovo codice a cplpc, sostituire i file base e modificare quelli di serializzazione aggiungendo il nuovo codice generato, lasciando invariato il precedente, aggiungere la chiamata all'helper di cplpc;

3. aggiungere al modulo di *mview una funzione col nome della system call nello stesso formato delle altre (un banale copia-incolla);

4. aggiungere al modulo di *mview una riga dove viene registrata la funzione precedentemente creata (anche in questo caso un banale copia-incolla);

5. aggiungere al modulo remoto una funzione col nome della systemcall (un copia-incolla come negli altri casi);

6. aggiungere il supporto per la funzione a cplpc, se la system call è nel formato descritto precedentemente è necessaria una sola riga, altrimenti una funzione di wrap/unwrap.

# Capitolo 5

# Funzionamento

## 5.1 Introduzione

Verranno in questo capitolo analizzate le prestazioni di umrsc, valutandone le performance in vari casi e fornendo possibili ottimizzazioni. I casi analizzati riguarderanno le performance in caso di utilizzo di operazioni di rete remote e in caso di mount di una directory. I test sono stati eseguiti con la versione "lenta" di *mview, ovvero Umview.

## 5.2 Performance rete

Per analizzare le performance di rete verrà confrontata in vari casi la velocità di download di un file, prima rimandando le operazioni di rete sulla stessa macchina, poi su una macchina sulla stessa rete locale, poi su una macchina remota.

Il test è stato eseguito su una macchina collegata ad una comune linea adsl effettuando il download di un file piuttosto grande, in questo caso un'immagine iso di una distribuzione debian, per scaricare il file è stato utilizzato il programma w3m. La macchina remota scelta è un computer dei laboratori di informatica dell'università di Bologna.

Queste le velocità raggiunte:

- connessione diretta: 840 kB/s
- operazioni di rete "remotizzate" sulla stessa macchina: 840 kB/s
- operazioni di rete su una macchina nella stessa rete locale: 340 kB/s
- operazioni di rete su una macchina remota: 20 kB/s

Come si può notare le prestazioni calano aumentando la "distanza di esecuzione" tra il programma e le system call, le operazioni eseguite in remoto hanno prestazioni che rendono l'intero sistema non usabile in caso di applicazioni reali.

Andando ad analizzare bene i particolari si può vedere che w3m va ad eseguire tante system call "read" di 1500 byte l'una, proviamo quindi ad eseguire gli stessi test con un programma usante buffer di dimensione differente. Per fare questo test è stato scritto un programma ad hoc, vediamo una tabella riassuntiva delle performance in vari casi.

| Dim. buffer | velocità (kB/s) |
|---|---|
| 1500 | 14 |
| 3000 | 26 |
| 10000 | 99 |
| 50000 | 211 |
| 100000 | 318 |
| 200000 | 370 |
| 1000000 | 350 |

Tabella 5.1: tabella velocità relativa alla dimensione del buffer di ricezione

Si può notare che la velocità aumenta circa linearmente in base alla dimensione del buffer di ricezione. Non raggiunge comunque la velocità massima perchè mentre lo skeleton "perde tempo" ad inviare i dati allo stub, non va contemporaneamente a gestire i nuovi dati in arrivo; un altro motivo è

che dev'essere comunque chiamata ogni volta anche la funzione gestente gli eventi.

Il problema fondamentale di questo sistema è che trasforma la rete funzionante tramite TCP/IP in un sistema funzionante tramite STOP/WAIT, perché per eseguire ogni system call è necessario aspettare la conclusione di quella precedente. Verrà analizzata una possibile ottimizzazione in grado di risolvere questo problema.

Nel resto dei casi, ovvero tutte le operazioni di rete non riguardanti un puro download di file ma operazioni più "interattive", non sono stati rilevati eccessivi problemi, il ritardo delle operazioni era trascurabile.

## 5.3 Performance file system

In questo caso verranno analizzati i tempi di un "ls" eseguito in una directory contenente un discreto numero di file. I luoghi di esecuzione sono gli stessi del caso precedente. La directory su cui è stato fatto il test contiene 27 directory.

Ecco una tabella riassuntiva:

| Luogo | Tempo (secondi) |
|---|---|
| Senza virtualizzazione | 0.010 |
| Stessa macchina | 0.134 |
| Macchina locale | 0.500 |
| Macchina remota | 8.727 |

Tabella 5.2: tabella velocità relativa alla macchina remota

Si può notare che anche in questo caso le performance degradano all'aumentare della distanza di esecuzione; il problema è lo stesso del caso precedente: il dover fare stop-and-wait tra una chiamata e l'altra. Anche in questo caso verrà analizzato un possibile metodo per migliorare le performance.

Le restanti operazioni invece, navigazione/creazione/eliminazione di file o directory, non hanno dato problemi.

## 5.4 Problema comune

Si può notare che il problema comune dei vari casi in cui sono stati eseguiti i test è la latenza delle operazioni e l'impossibilità di poter far terminare la system call prima che dei dati siano inviati al computer remoto e restituiti al chiamante (stop and wait), rendendo l'intero sistema molto sensibile alla latenza tra le due macchine. Questo non influisce molto su operazioni "interattive" ma rovina le performance quando è necessario fare più operazioni consecutive, molte read in caso di download di un file e molte stat in caso di un ls.

## 5.5 Rsc puro vs ottimizzato

Il prototipo creato sfrutta un sistema di remote system call puro: tutte le operazioni vengono gestite in egual modo, permettendo di risolvere ogni problema con lo stesso codice, non sono state fatte assunzioni su come verranno eseguite le operazioni remote nei casi reali.

Un possibile modo per raggiungere prestazioni ben maggiori è quello di fare assunzioni sulle possibili applicazioni delle varie system call, o addirittura ottimizzare il codice per un particolare insieme di operazioni. Questo metodo però incrementa la quantità di codice necessaria e rende necessario del codice differente per ogni possibile applicazione delle syscall remote: ogni sottomodulo dovrà "specializzare" l'implementazione di alcune system call facendo assunzioni sull'ordine o frequenza con cui le varie operazioni verranno eseguite.

## 5.6 Possibili ottimizzazioni

### 5.6.1 Introduzione

Come premesso, sarebbe possibile modificare il sistema di remote system call puro per aumentarne le prestazioni, agendo in modo specifico per ogni utilizzo; verranno quindi analizzati gli utilizzi in caso di rete e file system fornendo possibili ottimizzazioni.

### 5.6.2 Rete

In caso di syscall di rete remote i principali problemi sono dati da:

- consecutive chiamate alla syscall read/recv;
- consecutive chiamate alla syscall write/send.

La write/send è la più semplice da risolvere; il motivo è che nel caso di socket il successo della funzione non indica che i dati siano veramente arrivati a destinazione, ma solo che il sistema sottostante è riuscito a gestire i dati. Dal manuale della send[sen] si può leggere: "No indication of failure to deliver is implicit in a send(). Locally detected errors are indicated by a return value of -1". Una possibile ottimizzazione è quindi ritardare l'effettiva chiamata alla system call send finchè non si hanno abbastanza dati da inviare, o finchè non è passato abbastanza tempo dall'ultima chiamata, quindi una mini-reimplementazione dell'algoritmo di Nagle[nag].

In questo modo l'invio dei dati e l'attesa del risultato verrà effettuato una sola volta e in modo indipendente rispetto alla vera chiamata.

Per la recv la situazione si complica leggermente: è necessario anticipare le chiamate, invece che, come nel caso precedente, ritardarle. Una possibile implementazione può essere la seguente:

- si resta in attesa della recv (come nell'rsc puro);
- al posto di richiamare la recv con buffer e lunghezza originali la si chiama con un buffer ben più grande;

- lo skeleton proverà quindi a ricevere tutti i byte disponibili in quel momento nel buffer del sistema operativo sottostante (nel caso il buffer dato come parametro sia abbastanza capiente);
- i dati letti vengono rimandati allo stub, che passerà all'applicazione i dati richiesti ma che salverà i dati aggiuntivi per le successive chiamate alla recv.

Questo sistema rallenterebbe la prima recv avvantaggiando enormemente le successive, rendendo il sistema molto meno dipendente dalla latenza. Un'ottimizzazione ancora migliore sarebbe quella di eseguire in background un thread che si occupa di ricevere i dati disponibili ancora prima che vengano richiesti dall'applicazione; in questo caso c'è da trovare un buon compromesso tra dimensioni dei buffer e RAM utilizzata.

Sia nel caso della recv che nel caso della send è necessario aggiustare la funzione che gestisce gli eventi del file descriptor aggiungendo i due nuovi casi, che, differentemente dagli altri, verrebbero gestiti localmente.

### 5.6.3 File System

Nel caso del file system remoto, come si è visto, il rallentamento maggiore è dato dalle molteplici chiamate alla system call stat nel caso dell'esecuzione di un "ls". Per ovviare a questo una possibile ottimizzazione può essere data dal supporre che, nel caso sia stata fatta una chiamata a "getdents", funzione utilizzata per ricevere l'elenco di elementi contenuti in una directory, molto probabilmente successivamente ad essa verrà fatta una chiamata a stat per ogni elemento contenuto. Un modo di incrementare le prestazioni può quindi essere quello di restituire, assieme al risultato di getdents, anche il risultato di tutte le chiamate a stat effettuate sugli elementi contenuti nel risultato di getdents; in locale verranno poi salvati in un'apposita struttura tutti gli stat ricevuti, pronti per essere riutilizzati nelle successive chiamate.

Riassumendo:

- il client effettua una getdents;

- il server risponde col risultato della getdents;
- il server invia assieme al risultato precedente anche i risultati di tutti gli stat degli elementi del risultato;
- il client salva gli stat per usi futuri;
- il client in caso di stat controlla se il risultato è già presente e in caso restituisce quello.

In questa ottimizzazione c'è da fare attenzione alla validità dei risultati: se si considera valido per troppo tempo il risultato salvato localmente, si rischia di avere delle informazioni non più valide; sarebbe quindi giusto eliminare i risultati, o dopo la prima stat eseguita dopo la getdents, o dopo un intervallo di tempo ragionevole.

# Conclusioni

Umrsc cerca di essere un metodo alternativo per la soluzione di problemi esistenti, provando a risolverne i fattori comuni, minimizzando gli sforzi. Cerca poi di offrire nuovi metodi di esecuzione dei programmi, slegandoli dalla macchina dove effettivamente vengono avviati. Di lavoro da fare in questo campo ce n'è ancora molto, sarebbe bello se si potesse eseguire i programmi su computer potenti per quanto riguarda i calcoli, ma facendoli interagire con le risorse del proprio sistema. Umrsc potrebbe essere migliorato implementando le ottimizzazioni proposte e aggiungendo il supporto ad altre system call.

# Bibliografia


[AKL07] D. Laor U. Lublin A. Kivity, Y. Kamay and A. Liguori. kvm: the linux virtual machine monitor. In *Ottawa Linux Symposium*, pages 225–230. 2007. http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf.

[Cac02] Maximiliano Caceres. Syscall proxying - simulating remote execution. Corelabs Technical Report, 2002. http://www.coresecurity.com/files/attachments/SyscallProxying.pdf.

[Cac03] Maximiliano Caceres. Distributed computing using syscall proxying, 2003. http://www.google.com/patents?id=gE-XAAAAEBAJ.

[DG09] Renzo Davoli and Michael Goldweber. View-os: Change your view on virtualization., 2009. http://www.cs.unibo.it/~renzo/view-os-lk2009.pdf.

[GGD08] Ludovico Gardenghi, Michael Goldweber, and Renzo Davoli. View-os: A new unifying approach against the global view assumption. In Marian Bubak, Geert van Albada, Jack Dongarra, and Peter Sloot, editors, *Computational Science – ICCS 2008*, volume 5101 of *Lecture Notes in Computer Science*, pages 287–296. Springer Berlin / Heidelberg, 2008.

[nag] Nagle's algorithm. http://en.wikipedia.org/wiki/Nagle%27s_algorithm.

[PG74] Popek and Goldberg. Formal requirements for virtualizable third generation architectures. In *Communica-*

*tions of the ACM*, volume 17, pages 412–421. 1974. http://www.cs.unibo.it/~renzo/vsd/popek-virt-reqmts.pdf.

[sen] send - linux man page. http://linux.die.net/man/2/send.

[SS07] Andrew S.Tanenbaum and Maarten Van Steen. *Distributed System: Principles and Paradigms*, chapter Parameter Passing. Prentice Hall, 2nd edition, 2007.

[ssh] Ssh filesystem. http://fuse.sourceforge.net/sshfs.html.

[tso] tsocks - linux man page. http://linux.die.net/man/8/tsocks.

[umla] How is user mode linux different? http://user-mode-linux.sourceforge.net/old/UserModeLinux-HOWTO-1.htmlss1.2.

[umlb] The user-mode linux kernel home page. http://user-mode-linux.sourceforge.net/.

[umv] Umview wiki. http://wiki.virtualsquare.org/wiki/index.php/UMview.

[vms] Comparison of platform virtual machines. http://en.wikipedia.org/wiki/Comparison_of_platform_virtual_machines.

[Zan04] Zanichelli. Significato della parola virtuale, 2004. http://dizionari.zanichelli.it/parola-del-giorno/2004/10/28/la_parola_del_giorno__virtule/.

# Ringraziamenti

Ringrazio tutti quelli che mi hanno supportato durante il percorso di laurea; ringrazio il relatore, professor Renzo Davoli, che mi ha seguito nello svolgimento di questa tesi.