Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

tesi di laurea

Metodi e sistema di licensing in ambienti virtuali

Relatore: Ch.mo Prof. Matteo Bertocco
Correlatore: Ing. Mauro Franchin

Laureando: Andrea Veronese

28 settembre 2010

Indice

1 Introduzione **1**

2 La virtualizzazione **3**
Introduzione 3
2.1 Virtualizzazione 4
2.1.1 Caratteristiche 4
2.1.2 Vantaggi 6
2.2 VMWare vSphere 6
2.2.1 ESX e ESXi 7
2.2.2 Virtual Simmetric Multi-Processing (vSMP) 8
2.2.3 Virtual Machine File System (VMFS) 9
2.2.4 vCenter 11
2.2.5 vMotion, Storage vMotion 13
2.2.6 Distributed Resource Scheduler (DRS) 15
2.2.7 High Availability (HA) 15
2.2.8 Fault Tolerance (FT) 16

3 Il problema del *licensing* **19**
Introduzione 19
3.1 Attuali meccanismi di *licensing* 19
3.2 *Licensing* in ambienti virtuali 20
3.3 Possibili approcci 21
3.3.1 Utilizzo di comandi da riga di comando 21
3.3.2 Accesso ad internet 22
3.3.3 NAT virtuale 22
3.3.4 *System ID* 23

4 Progettazione **25**
Introduzione 25
4.1 Requisiti strutturati 25

4.1.1 Mantenimento della compatibilitá con il sistema di *licensing* esistente 26
4.1.2 Estensioni del sistema di *licensing* 28
4.1.3 Protezione della comunicazione tra *License Server* e applicazioni 30
4.1.4 Generazione del *System ID* a partire da parametri locali della macchina del cliente 30
4.2 Requisiti funzionali 32
4.2.1 Struttura dell'ambiente 32
4.2.2 Caratteristiche della connessione tra *License Server* e applicazioni 32
4.2.3 Richiesta e fornitura della licenza 34
4.2.4 *Refresh* della licenza 35
4.2.5 Configurazione del *License Server* 35
4.2.6 Applicazioni 35
4.2.7 *Option* 36
4.3 Specifiche di progetto 37
4.3.1 Diagramma dei componenti del sistema di *licensing* 37
4.3.2 Diagramma di *deployment* del sistema di *licensing* . 39
4.3.3 Diagramma di sequenza del sistema di *licensing* . . 41
4.3.4 Diagramma di attivitá del *License Client* 42
4.3.5 Diagramma di attivitá del *License Server* 43
4.3.6 Diagramma di attivitá del modulo di gestione dei *License Files* 43

5 Realizzazione **47**
Introduzione 47
5.1 Implementazione 47
5.1.1 Aspetti principali 48
5.1.2 Architettura generale 48
5.2 Scelte implementative 50
5.2.1 *Socket* TCP 52
5.2.2 Programmazione concorrente 57

6 Test e collaudo **61**
Introduzione 61
6.1 Fondamenti del *testing* 61
6.1.1 Caratteristiche del *testing* 63
6.2 *Testing white-box* 63
6.2.1 *Testing* per cammini di base 63
6.3 *Testing black-box* 65

6.3.1 Metodo di *testing* basato sui grafi 66
6.3.2 Suddivisione in classi di equivalenza 66
6.3.3 Analisi dei valori limite 67
6.4 *Testing* del sistema di *licensing* 67

7 Considerazioni finali **71**

Bibliografia **72**

Elenco delle figure **76**

Indice analitico **78**

Capitolo 1

Introduzione

Il presente lavoro di tesi documenta l'attivitá di tirocinio svolta presso l'azienda *Mida Solutions s.r.l.*; lo scopo dell'esperienza é stato lo sviluppo di un applicativo per la gestione, in ambiente virtuale, dei licenziamenti degli applicativi forniti dall'azienda. Uno dei requisiti piú importanti che da subito é stato indicato riguarda il mantenimento della compatibilitá con l'esistente sistema di *licensing*; questo requisito ricalca l'interesse, da parte dell'azienda, di circoscrivere l'area d'influenza delle modifiche da apportare. A tal proposito si sono spese energie per la conoscenza del sistema esistente, individuando i punti d'interesse per lo sviluppo e l'estensione.

L'attivitá di tirocinio si é articolata su due aspetti: la ricerca e lo studio delle attuali tecniche di *licensing* in ambienti virtuali (per questo, si sono analizzati gli attuali sistemi di virtualizzazione); una volta trovata la metodologia per il licenziamento degli applicativi aziendali in ambiente virtuale si é passati allo sviluppo del sistema.

L'attivitá di sviluppo dell'applicativo é stata affrontata seguendo le fasi:

- ricerca e analisi dei requisiti
- progettazione
- realizzazione
- test

Pur senza alcuna esperienza lavorativa, si é cercato di affrontare ciascuna fase con le rigorose metodologie apprese in ambito accademico, costruendo un percorso che ha unito un forte coinvolgimento autonomo ad un'arricchente collaborazione con il *team* di sviluppo dell'azienda.

Nella prima fase di ricerca e analisi si sono raccolte e studiate le caratteristiche degli attuali sistemi di virtualizzazione; mai come in questi

ultimi tempi sono stati fatti passi avanti in tale campo, e molte aziende, a tal proposito, stanno muovendo verso tale direzione. Anche l'azienda in cui ho svolto il tirocinio sta puntando verso un cambiamento della logica architetturale, in vista di un passaggio in ambiente virtuale; l'applicativo da sviluppare ha richiesto, infatti, l'inserimento in ambiente virtuale.

La fase di progettazione ha preso forma con la raccolta dei requisiti: ció é stato realizzato attraverso un'intensa attivitá di interviste al tutor aziendale. Durante questi incontri é stato raccolto, in maniera non strutturata, un insieme di specifiche che si richiedeva il prodotto vantasse. Il lavoro ha richiesto, poi, la strutturazione dei requisiti, evidenziando in tal modo gli aspetti principali del prodotto da realizzare. La scrittura delle specifiche di progetto é stato l'ultimo *task* di questa fase: a tal fine, é tornata utile la conoscenza delle tecniche di costruzione dei diagrammi UML; un modo, questo, per adottare un "linguaggio" comune all'interno del *team* di sviluppo aziendale.

La fase di realizzazione é stata caratterizzata dalla necessitá di acquisire esperienza e dimestichezza con i *tool* di sviluppo forniti. La collaborazione con alcuni ingegneri dell'azienda per la risoluzione di dubbi e quesiti, ha permesso un lento ma costante miglioramento del *software*. Molto utili si sono rivelati gli incontri di allineamento cui ho partecipato assieme alle persone che in futuro saranno interessate al funzionamento dell'applicativo: mi hanno permesso di comprendere maggiormente le dinamiche e i vantaggi (non sono riuscito a scorgere svantaggi) della collaborazione.

Infine, nell'ultima fase di test si sono affrontate alcune ricerche sulle attuali tecniche di *testing* e si é stilato un *test plan* per una futura fase di test e raccolta dei risultati.

Capitolo 2

La virtualizzazione

Introduzione

La prima attivitá affrontata durante l'esperienza di tirocinio si é concentrata sullo studio dei sistemi di virtualizzazione. Questo tema, oltre ad interessare fortemente il lavoro di tirocinio, sta conoscendo un'importante evoluzione nel mondo informatico. Ad oggi, il mercato dei prodotti *software* legati alla virtualizzazione oppure quello di *hardware* con supporto alla virtualizzazione é in continuo aumento. Ogni anno grandi aziende come Microsoft, VMWare e Citrix investono ingenti quantitá di denaro per cercare di ottenere grossi profitti da questo mercato.

La fama raggiunta dalla virtualizzazione si deve soprattutto ad un'azienda: VMWare. VMWare é attualmente il leader incontrastato del mercato mondiale riguardante prodotti *software* di virtualizzazione, con piú del 90% della fetta di mercato negli USA tra le *Fortune 1000*[1]: la lista delle mille aziende statunitensi con il fatturato maggiore.

Oltre ad essere l'azienda che sta dettando legge, VMWare fornisce attualmente i *software* di virtualizzazione utilizzati dall'azienda nella quale ho svolto il tirocinio. Da alcuni mesi, infatti, *Mida Solutions s.r.l.* ha scelto di adottare un'architettura virtuale VMWare per il proprio sistema informatico.

In questo capitolo si analizzeranno dapprima le caratteristiche dei sistemi di virtualizzazione, si passerá poi ad approfondire il sistema VMWare *vSphere*, utilizzato tutt'oggi dall'azienda *Mida Solutions s.r.l.*.

[1] `http://www.vmware.com/company/customers/`

2.1 Virtualizzazione

Si puó definire la virtualizzazione come [1]:

"La virtualizzazione é la separazione di una risorsa o di una richiesta di servizio dalla metodologia con cui questa richiesta viene effettivamente servita a livello fisico."

Ad esempio, attraverso l'utilizzo di memoria virtuale un programma ha la possibilitá di accedere a piú memoria di quanta ce ne sia fisicamente installata sulla macchina attraverso l'utilizzo di *swap* su disco. Questo ragionamento chiaramente puó essere esteso anche ad altre componenti come la rete, il disco, il sistema operativo, ecc.. Comunemente la virtualizzazione é stata presentata in maniera semplificata al pubblico, come la possibilitá di far eseguire contemporaneamente sulla stessa macchina fisica, sistemi operativi diversi. Questa affermazione é parzialmente corretta dal momento che ció che é importante non é tanto il sistema operativo installato, ma l'ambiente virtuale che ne permette l'esecuzione, indipendentemente da quale sistema operativo si scelga poi di utilizzare.

2.1.1 Caratteristiche

Un ambiente virtuale é un insieme di componenti hardware e periferiche implementate via *software*, conosciuto con il nome di *macchina virtuale* (Figura 2.1). Una macchina virtuale puó permettere l'esecuzione di un sistema operativo *guest*, detto "ospite", sul quale vengono eseguite una o piú applicazioni. Ogni macchina virtuale, a sua volta, viene eseguita su uno strato di virtualizzazione chiamato tecnicamente *hypervisor* ("supervisore") che possiede un *Virtual Machine Monitor (VMM)*, facente sempre parte dello strato di virtualizzazione, che ne soddisfa i bisogni di CPU, memoria e periferiche. Ogni *hypervisor* contiene piú VMM che soddisfano le macchine virtuali e competono sulle risorse fisiche della macchina *host*.
Esistono due tipologie di *hypervisor*:

1. **Tipo 1** o **"nativo"**
2. **Tipo 2** o **"ospitato"**

Un *hypervisor hosted* é situato al di sopra di un sistema operativo *host*, detto "ospitante" (Windows, Linux, ecc.): in questo caso si parla di *Hosted Virtualization*. Un esempio di prodotto che permette la *Hosted Virtualization* é VMWare Server.

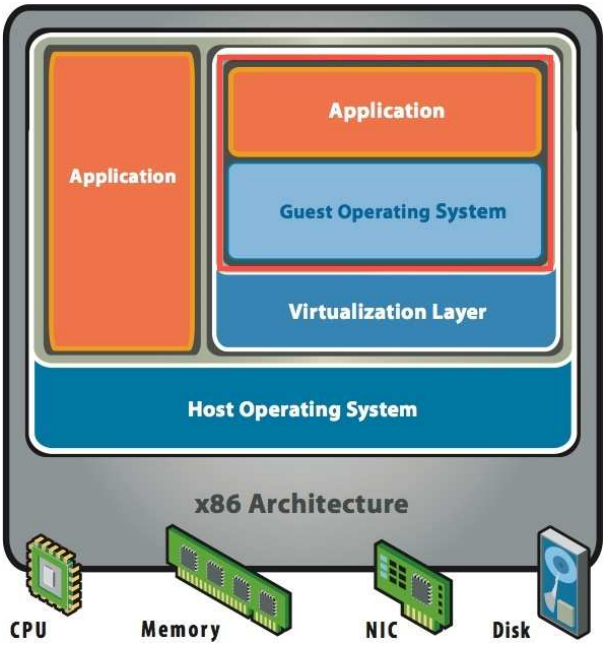

Figura 2.1: Architettura VMWare; in rosso é stata evidenziata una Macchina Virtuale.

Un *hypervisor bare-metal* é invece situato direttamente sopra l'hardware della macchina e ne possiede il controllo diretto e privilegiato. In questo caso si parla anche di virtualizzazione nativa. Un esempio di virtualizzazione nativa é data da prodotti come VMWare ESX o XenServer. Attraverso la virtualizzazione nativa é possibile ottenere prestazioni di gran lunga migliori rispetto alla virtualizzazione *hosted*, poiché ci sono meno strati per arrivare all'hardware.

Entrambe le tipologie permettono di eseguire macchine virtuali al loro interno e, dal punto di vista del *filesystem* (qualunque esso sia), vengono considerate come un insieme di file (file del disco fisso, file della RAM, file per lo *swap*, file di configurazione, ecc.).
Una macchina virtuale puó essere creata in piú modi:

1. seguendo un normale processo di creazione da menú;
2. trasformando una macchina fisica in una virtuale attraverso un programma che effettua questa conversione (chiamata P2V: Physical to Virtual), come per esempio VMWare Converter [2];
3. clonandone una giá esistente.

2.1.2 Vantaggi

Una macchina virtuale rispetto alla controparte fisica fornisce numerosi vantaggi:

1. puó eseguire concorrentemente con altre macchine virtuali, sfruttando al massimo l'utilizzo delle risorse hardware del sistema ospite;
2. puó essere facilmente clonata, archiviata o ripristinata;
3. puó eseguire i programmi in completo isolamento;
4. é portabile tra sistemi fisici che supportano la stessa tecnologia di virtualizzazione;
5. puó essere generata attraverso *template* per una rapida creazione, configurazione e attivazione.

2.2 VMWare vSphere

VMWare vSphere [3] é il nome della suite di programmi e funzionalitá fornite da VMWare per la virtualizzazione di un infrastruttura. Tra i prodotti e le funzionalitá che la compongono vi sono:

- ESX e ESXi
- Virtual Simmetric Multi-Processing (vSMP)
- Virtual Machine File System (VMFS)
- vCenter
- VMotion
- Distributed Resource Scheduler (DRS)
- High Availability (HA)
- Fault Tolerance (FT)

2.2.1 ESX e ESXi

ESX ed ESXi [4, 5] sono *hypervisor* di tipo nativo, condividono le stesso motore di virtualizzazione e forniscono lo stesso insieme di funzionalitá, ad esclusione della *Service Console* (ESXi infatti ne é privo). La *Service Console* é una console basata su Linux, che fornisce un insieme di comandi per interagire con il *vmkernel*. L'*hypervisor* é, infatti, suddiviso in due componenti: *Virtual Machine Monitor* e *vmkernel*. Il *vmkernel* é il cuore del sistema e gestisce le principali funzioni del sistema di virtualizzazione (*scheduling* della CPU, gestione della memoria, elaborazione dei dati, ...). Ogni macchina virtuale viene gestita da un *Virtual Machine Monitor* separato (come mostrato in Figura 2.2) situato all'interno dell'*hypervisor*, che ne soddisfa le richieste di CPU, memoria e I/O.

Gestione delle risorse

ESX ed ESXi forniscono la possibilitá di gestire e limitare la quantitá di CPU e RAM da assegnare alle macchine virtuali ospitate al loro interno attraverso l'utilizzo di avanzati meccanismi. Questi meccanismi sono:

Reservation Serve per riservare un quantitativo "minimo" di risorse senza le quali una macchina virtuale nemmeno si accende. Nel caso della RAM questa deve essere fisicamente presente e non puó essere fornita da *swap* su disco.

Limit Serve per limitare l'utilizzo di risorse.

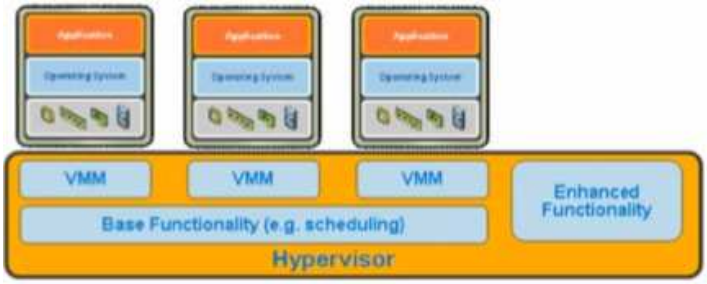

Figura 2.2: Ogni macchina virtuale viene gestita da un *Virtual Machine Monitor* separato, situato all'interno dell'*hypervisor*, che ne soddisfa le richieste di CPU, memoria e I/O. [Fonte `http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf`]

Share Indica la prioritá di una macchina rispetto alle altre di accedere alle risorse.

I meccanismi che abbiamo visto possono essere sfruttati sia per quanto riguarda la CPU che per quanto riguarda la RAM, anche se quest'ultima possiede anche altri meccanismi piú avanzati. Uno di questi é il *Memory Overcommitment*, cioé la possibilitá da parte di ESX ed ESXi di fornire piú memoria di quanta fisicamente installata sulla macchina server.

2.2.2 Virtual Simmetric Multi-Processing (vSMP)

Attraverso *Virtual SMP* [5] una macchina virtuale ha la possibilitá di sfruttare piú processori fisici (come mostrato in Figura 2.3) per portare a termine compiti particolarmente pesanti. Con l'ultima versione di ESX (ESX 4) si é raggiunta la possibilitá di assegnare fino ad 8 processori virtuali ad una macchina virtuale [6]. Partendo da ESX 2.1 é inoltre possibile sfruttare l'*Hyper-Threading*[2]. L'*Hyper-Threading* é una tecnologia Intel, che permette di sfruttare un processore per l'esecuzione contemporanea di due thread diversi, duplicando al suo interno alcune delle unitá di elaborazione maggiormente utilizzate, al fine di poter eseguire simultaneamente alcune operazioni, grazie a tecniche di *multithreading*. Un singolo *core* é, quindi, in grado di gestire due *thread* contemporaneamente; quando le istruzioni di un *thread* rimangono bloccate nella *pipeline* il processore procede ad elaborare un secondo *thread* al fine di mantenere le unitá di elaborazione sempre attive. Questo puó portare ad un incremento delle prestazioni che

[2] `http://www.intel.com/technology/platform-technology/hyper-threading/`

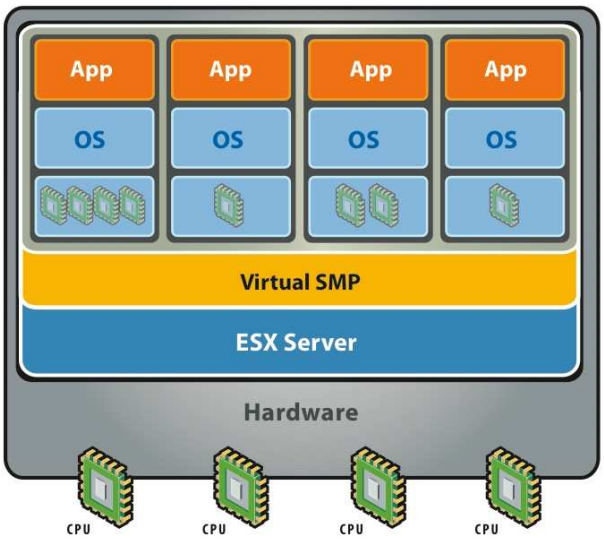

Figura 2.3: Attraverso *Virtual SMP* una macchina virtuale ha la possibilitá di sfruttare piú processori fisici.

varia a seconda della quantitá di lavoro e dalla sua tipologia. In certi casi di carichi molto pesanti le prestazioni potrebbero anche diminuire. Infatti, i due *thread* potrebbero arrivare a competere sulle risorse condivise finendo per rallentarsi a vicenda [7].

Le prestazioni di un processore con *Hyper-Treading* abilitato non saranno mai superiori a quelle di un sistema con due processori. Abilitando l'*Hyper-Threading* vSphere permetterá di vedere un quantitativo di processori logici da assegnare alle macchine virtuali pari al doppio di quelli fisici. Una delle controindicazioni, dovuta alla vSMP, é che per ogni macchina virtuale deve essere allocata una parte di RAM aggiuntiva per la gestione da parte di ESX [6]. L'*overhead* in questione é direttamente proporzionale alla quantitá di RAM assegnata alla macchina virtuale e al numero di processori virtuali assegnati; nello specifico, sale man mano che o aumenta la RAM o aumentano i processori virtuali.

2.2.3 Virtual Machine File System (VMFS)

Una macchina virtuale dal punto di vista del *filesystem* viene percepita come un insieme di file: disco, *swap*, configurazione ecc.. Questi file possono essere allocati localmente sul server ESX oppure in maniera logicamente centralizzata utilizzando *Storage Area Network* e *Network Attached Storage*. Quando lo spazio dove risiedono le macchine virtuali é centralizzato

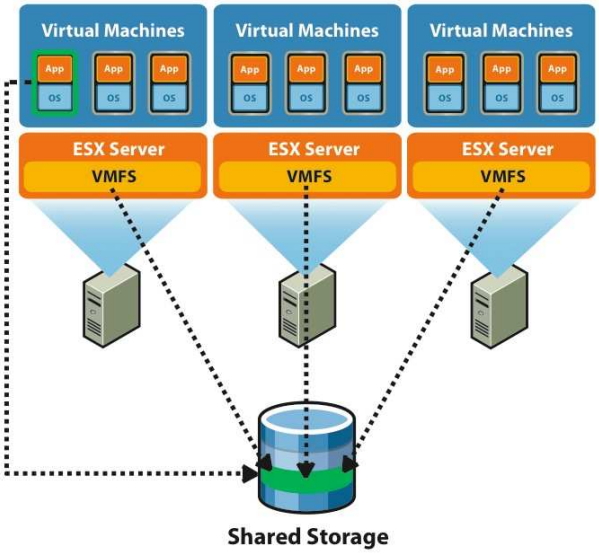

Figura 2.4: Server ESX che accedono concorrentemente a macchine virtuali situate all'interno dello stesso spazio di memorizzazione formattato con VMFS [Fonte `http://www.vmware.com/pdf/vmware_infrastructure_wp.pdf`].

sorge l'esigenza di un meccanismo per gestire la concorrenza. Questo avviene attraverso l'utilizzo di un *filesystem* concorrente come VMFS [8, 5]. Attraverso questo *filesystem*, piú server ESX possono accedere concorrentemente alle risorse presenti all'interno di uno stesso spazio di memorizzazione condiviso (vedi figura 2.4). Un sistema di *locking* sui metadati permette ad un solo server ESX di avviare una macchina virtuale; in caso di guasto di una macchina virtuale viene gestito in automatico un sistema di rilascio dei *lock* per sbloccare le risorse. VMFS permette la creazione di file fino a 2TB e puó utilizzare blocchi da 1 a 8 MB. I blocchi sono di grosse dimensioni per favorire l'I/O dal momento che un *filesystem* utilizzato per memorizzare macchine virtuali contiene pochi file di grosse dimensioni. VMFS é la base di partenza per funzionalitá piú avanzate come VMotion e DRS che si analizzano in seguito. VMFS fornisce anche le funzionalitá e i meccanismi di consistenza e *recovery* tipici di un *filesystem* di classe enterprise come, per esempio, funzionalitá di *journaling* distribuito.

2.2.4 vCenter

Attraverso vCenter [9, 5] (come mostrato in Figura 2.5) é possibile gestire in maniera centralizzata tanti aspetti relativi all'infrastruttura VMWare. Oltre al fatto che vCenter é una componente obbligatoria per poter usufruire delle piú avanzate funzionalitá di ESX come *Fault Tolerance, DRS, VMotion, Storage VMotion* (che altrimenti sarebbero disabilitate), permette anche di:

1. gestire in maniera centralizzata tutti i componenti facenti parte del nostro sistema di virtualizzazione ed avere un sistema centralizzato di autenticazione;

2. monitorare attraverso grafici l'utilizzo delle risorse (come CPU, disco, memoria) delle varie macchine virtuali e dei server ESX;

3. pianificare operazioni in determinati momenti o allertare gli amministratori in caso si verifichino situazioni anomale che si vogliono gestire;

4. aggiungere velocemente nuove macchine virtuali utilizzando template;

5. creare *cluster* di server ESX (Figura 2.6) dove le risorse globali del *cluster* sono la somma delle risorse di ogni componente. Ogni macchina virtuale appartenente ad un *cluster* avrá poi a disposizione tutte le sue risorse;

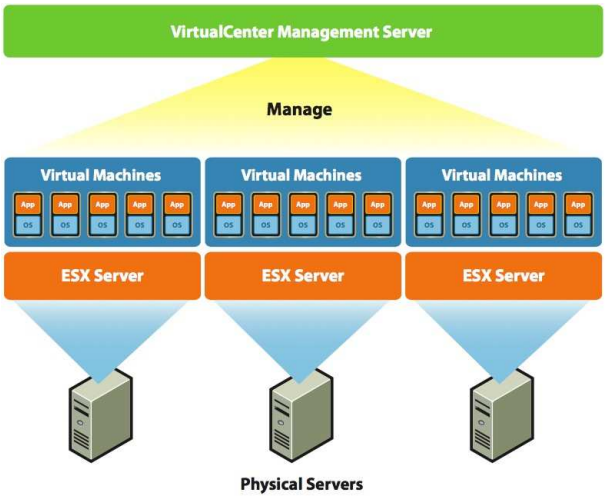

Figura 2.5: vCenter permette di gestire in maniera centralizzata diversi aspetti dell'infrastruttura VMWare [Fonte `http://www.vmware.com/pdf/vmware_infrastructure_wp.pdf`].

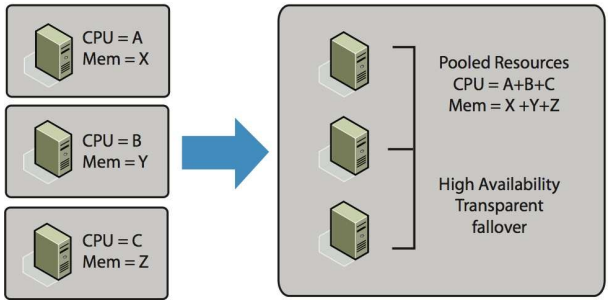

Figura 2.6: Cluster di Server ESX; vCenter permette di creare *cluster* di server ESX dove le risorse globali del *cluster* sono la somma delle risorse di ogni componente. Ogni macchina virtuale appartenente ad un *cluster* avrá poi a disposizione tutte le proprie risorse; [Fonte `http://www.vmware.com/pdf/vmware_infrastructure_wp.pdf`].

6. creare insiemi di risorse. E' possibile specificare un limite massimo di utilizzo di risorse per ogni insieme. Le macchine virtuali all'interno di un insieme (*pool*) saranno costrette a suddividersi le risorse che gli sono state assegnate.

2.2.5 vMotion, Storage vMotion

Con VMotion [10, 5] é possibile far migrare una macchina virtuale da un server ESX ad un altro "a caldo" (cioé senza dover spegnere il sistema), senza dover interromperne l'esecuzione. Questa funzionalitá é molto utile perché consente, in caso di guasto o spegnimento di un Server ESX, di spostare le macchine virtuali su altri server ESX. Inoltre VMotion sta alla base anche di alcune tecniche di bilanciamento di carico; infatti se i server sono pesantemente sotto sforzo é possibile intervenire spostando alcune macchine virtuali su server meno carichi di lavoro o addirittura su nuovi nodi (Figura 2.7).

Oltre a VMotion esiste una variante chiamata Storage VMotion [11], che funziona in maniera simile e permette di trasferire "a caldo" una macchina virtuale da un *datastore* ad un altro. Questa funzionalitá é presente a partire da ESX 3.5 ed é piú pesante di un operazione di VMotion, dal momento che deve essere spostata l'intera macchina virtuale (non solo lo stato) da uno spazio di memorizzazione all'altro.

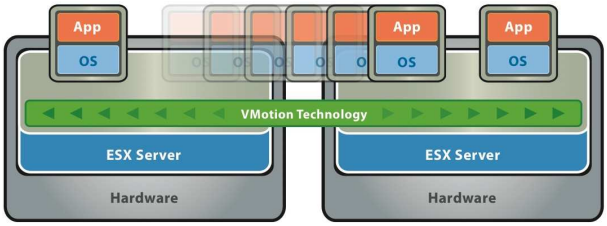

Figura 2.7: Un'operazione di vMotion tra due server ESX. Con VMotion é possibile far migrare una macchina virtuale da un server ESX ad un altro "a caldo" senza dover interromperne l'esecuzione. [Fonte `http://www.vmware.com/pdf/vmware_infrastructure_wp.pdf`].

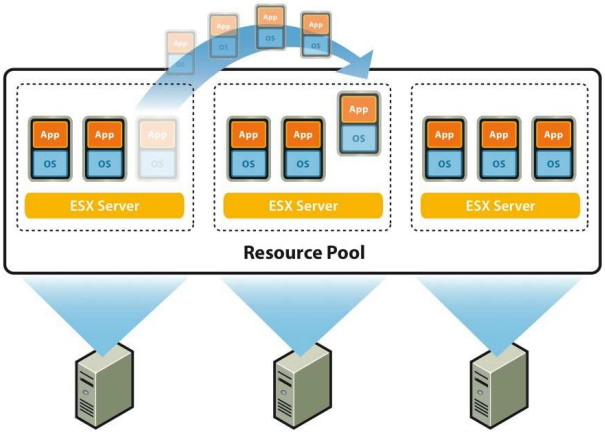

Figura 2.8: Un'operazione di ridistribuzione del carico compiuta dal DRS. [Fonte `http://www.vmware.com/pdf/vmware_infrastructure_wp.pdf`].

2.2.6 Distributed Resource Scheduler (DRS)

Il DRS [12, 5] permette di ottimizzare l'utilizzo delle risorse tra vari server fisici che utilizzano ESX, ma necessita, come ogni altra funzionalitá avanzata di VMWare, della presenza di *vCenter*. L'ottimizzazione dipende innanzitutto da un insieme di regole definite dall'amministratore che puó decidere quali macchine virtuali devono avere una maggiore prioritá rispetto ad altre e quante risorse minime e massime una macchina puó avere. Il DRS cerca di bilanciare il carico di lavoro sui server ESX esistenti in maniera adattiva. Nel caso un server sia scarico ed un altro no, il DRS attraverso *VMotion* sposta automaticamente alcune macchine virtuali sul server piú scarico cosí da bilanciarle (Figura 2.8). Nel caso in cui tutti i server sono carichi é possibile accendere ed aggiungere automaticamente a caldo un nuovo server ESX, che nel frattempo era rimasto in attesa in modalitá passiva. Questa operazione avviene anche in maniera inversa e cioé, se i server cominciano ad essere troppo scarichi, quelli in eccesso vengono spenti e le risorse ridistribuite sugli altri contribuendo a non utilizzare risorse in eccesso. DRS permette inoltre di espandere la potenza di calcolo di un centro aggiungendo nuovi server le cui risorse vengono subito messe a disposizione.

2.2.7 High Availability (HA)

L'*High Availability* [13, 5] in VMWare é fornita dall'unione di DRS, *VMotion* e VMFS. In caso di guasto di un server o di un sistema operativo, il

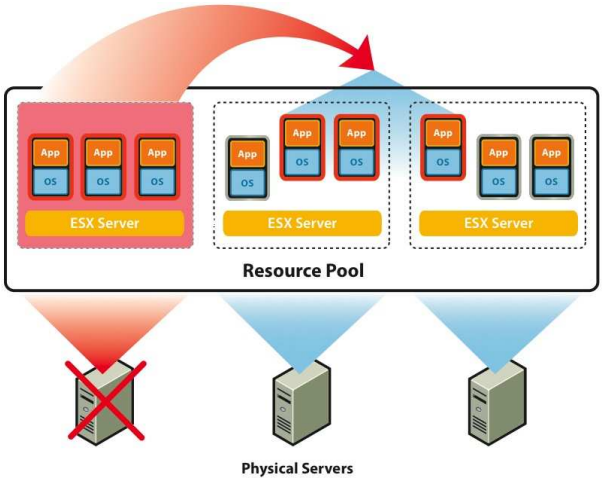

Figura 2.9: High Availability ottenuta attraverso l'utilizzo di DRS, vMotion e VMFS. [Fonte `http://www.vmware.com/pdf/vmware_infrastructure_wp.pdf`].

DRS ha il compito di ridistribuire automaticamente tutte le macchine virtuali che stava servendo sugli altri server attraverso operazioni di *VMotion* (Figura 2.9). Per rilevare e gestire automaticamente eventuali guasti viene utilizzato un servizio di monitoraggio dei server e delle macchine virtuali di tipo *Heartbeat*. É importante notare peró che prima che gli altri server ESX possano prendersi carico delle macchine virtuali rimaste orfane, queste generano un disservizio per il periodo necessario al loro riavvio.

2.2.8 Fault Tolerance (FT)

Come visto sopra DRS e VMotion vengono utilizzate da VMWare per fornire *High Availability*. Nonostante questo l'HA di VMWare non riesce ad evitare periodi di disservizio, pur se brevi, in caso di guasti di un server ESX. Per certe aziende questi brevi periodi (che si aggirano nell'ordine di 5-10 minuti, anche se é dipendente dal numero di macchine virtuali da far ripartire) peró non sono tollerabili e questo ha portato VMWare ad aggiungere un ulteriore funzionalitá chiamata *Fault Tolerance* [14]. VMWare *Fault Tolerance* si ripropone di evitare completamente periodi di disservizio, facendo si che ogni macchina virtuale crei una copia clone di se stessa su un server ESX, differente da quello su cui sta eseguendo in quel momento, cosí da evitare periodi di disservizio in caso di guasto di un nodo ESX (Figura 2.10). La macchina clone é in uno stato di completo isolamento

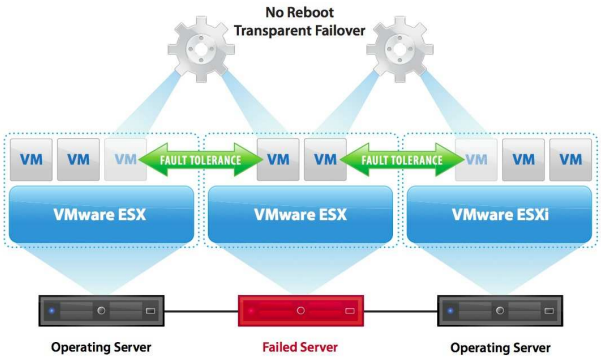

Figura 2.10: *Fault Tolerance* si ripropone di evitare completamente periodi di disservizio, facendo si che ogni macchina virtuale crei una copia clone di se stessa su un server ESX, differente da quello su cui sta eseguendo in quel momento, cosí da evitare periodi di disservizio in caso di guasto di un nodo ESX. [Fonte `http://www.vmware.com/pdf/fault_tolerance_datasheet.pdf`].

e viene continuamente sincronizzata ad ogni aggiornamento di quella originale. A questo punto si possono verificare tre situazioni: cade il nodo ESX dove é situata la macchina originale, cade il nodo ESX dove é situata la macchina clone, oppure cadono entrambi. Ovviamente, il caso in cui cadono entrambi i nodi, é il caso sfortunato che porterá ad un disservizio. Se invece dovesse cadere il nodo dove é situata la macchina originale, la macchina clone viene subito attivata per sostituire la macchina originale e a sua volta crea una nuova copia di se stessa su un altro nodo ESX per proteggersi da ulteriori guasti. Questo stesso ragionamento si applica nel caso in cui cade la macchina clone: sará la macchina originale a creare un altra copia su un nuovo nodo ESX.

Capitolo 3

Il problema del *licensing*

Introduzione

Dopo le attivitá di studio e conoscenza dei sistemi di virtualizzazione esistenti, l'attenzione si é concentrata sulla ricerca e sull'analisi delle problematiche legate all'uso dell'attuale sistema di licenziamento degli applicativi (*Licensing System*) in ambiente virtuale. É questo, infatti, un problema che dev'essere considerato da chi produce e vende applicazioni *software*; a tal proposito é interessante osservare quanto riferisce Amy Konary, *research director* per IDC[1] [15]:

> "Software publishers that have not already begun addressing how to manage the licensing of their applications in virtual environments are behind the curve. The strategy should not only include support for licensing in virtual environments but also the ability to enforce license terms once the application is deployed on a virtual machine. Without enforcement, the software publisher has no control over the license and therefore their revenue."

3.1 Attuali meccanismi di *licensing*

Oggi, la maggior parte, se non tutte, le tecnologie di *software licensing* si basano su un concetto conosciuto come *host-based license enforcement* [16]. Brevemente, si tratta di un concetto in cui le politiche di licenziamento sono legate ad una specifica macchina fisica, conosciuta ed autorizzata.
Tipicamente, una licenza *software* viene strettamente agganciata ad un

[1] International Data Corporation, `http://www.idc.com`

computer attraverso un meccanismo autorizzato conosciuto come *hardware fingerprinting* o *node locking*. Tale meccanismo consiste nel proteggere la licenza da duplicazione e condivisione non autorizzate, vincolandola alla macchina. Se la licenza viene copiata su una nuova macchina con una nuova impronta (*fingerprint*), questa diviene automaticamente non valida. L'esempio piú comune é quello di associare la licenza a parametri fisici unici come l'identificativo dell'*hard disk* o il MAC *address*. Il *fingerprinting* é una tecnica ampiamente utilizzata per proteggere le licenze dei *software*, e le applicazioni licenziate, dalle copie illegali.

3.2 *Licensing* in ambienti virtuali

La virtualizzazione [17] ha introdotto un'importante sfida a questo metodo di protezione dalla clonazione di licenze. La possibilitá di creare *hardware* virtuali fornisce, conseguentemente, la possibilitá di creare anche impronte virtuali. Un'impronta clonata normalmente deriva da una macchina virtuale clonata, e il meccanismo di *licensing* tratterá, inconsapevolmente, le impronte virtuali come se fossero impronte derivate da macchine fisiche reali.
Come tutte le applicazioni e i processi in ambiente virtuale, anche i sistemi di *licensing* in genere si comportano come se l'ambiente virtuale fosse una macchina fisica, separata ed indipendente. In un ambiente virtuale, i parametri *hardware* utilizzati per agganciare una licenza sono virtualizzati e, quindi, facilmente duplicabili. I sistemi di *licensing*, come per una macchina fisica, calcoleranno un'impronta per quel determinato ambiente virtuale. Il blocco delle licenze per un *hardware ID* viene aggirato in ambiente virtuale attraverso la creazione di una macchina virtuale e successivamente, dopo l'installazione del *software* necessario, con la clonazione di quella macchina virtuale. Ogni istanza in esecuzione di quella macchina virtuale avrá, infatti, lo stesso *hardware ID*. Dal momento che l'*hardware* fisico é disaccoppiato ed emulato dal *software* virtuale, qualsiasi *hardware ID* puó essere manipolato e cambiato in modo che corrisponda ad un altro ambiente virtuale. Questo invalida la sicurezza del sistema di *fingerprinting* della macchina. La clonazione dell'ambiente virtuale per un *license server* consente ad un utente di raddoppiare la quantitá di licenze disponibili con ogni istanza di virtualizzazione.

Quello che sembrava un collaudato e sicuro meccanismo contro la pirateria, ora non fornisce piú alcun livello di garanzia e sicurezza al venditore di *software*.
A tutto questo, si aggiunge un'ulteriore preoccupazione per i venditori di

software: l'utente "onesto" comune é ora in grado, inavvertitamente, di clonare licenze con ordinarie operazioni. In altre parole, quella che sta diventando una metodologia comune per lo sviluppo di applicativi, risulta un accidentale tentativo di duplicazione di licenze *software*. Questo presenta un ulteriore problema, per cui il venditore potrebbe avere meno potere da un punto di vista legale, non essendo in grado di prendere una posizione certa nei confronti di coloro che duplicassero inavvertitamente le proprie licenze.

3.3 Possibili approcci

Presa conoscenza del problema, si sono analizzate diverse modalitá di reperimento dei parametri fisici di una macchina fisica (*host*), a partire da una macchina virtuale (*guest*) su di essa installata. Di seguito vengono esaminate in dettaglio le diverse strategie affrontate; per ciascuna, vengono riportati i risultati cui si é arrivati.

3.3.1 Utilizzo di comandi da riga di comando

La prima strategia affrontata si é concentrata sull'utilizzo di comandi che fornissero quante piú informazioni possibili relative ai dettagli fisici della macchina *host* sulla quale veniva eseguita la macchina virtuale. Di seguito vengono analizzati i risultati ottenuti[18]:

`hwinfo` fornisce parametri fisici di tutti gli *hardware items*;

`lspci` fornisce informazioni su tutti i PCI[2] *buses* del sistema e di tutti i dispositivi ad esso connessi;

`dmesg` comando utilizzato per esaminare o controllare il *Kernel Ring Buffer*; mostra i dettagli *hardware* di tutto ció che é visto dal sistema operativo;

`cat /proc` il *file system* `proc` fornisce informazioni sulla memoria fisica e sulla CPU di un sistema in tempo reale;

`fdisk` fornisce informazioni riguardanti partizioni, spazio disponibile, spazio allocato, *swap* e altro.

`dmidecode` permette di leggere il contenuto della DMI[3] *table* la quale contiene informazioni riguardanti componenti dell'*hardware* di sistema.

[2] *Peripheral Component Interconnect*
[3] *Desktop Management Interface*

Risultati

Dopo un'attenta analisi dei risultati ottenuti dai comandi descritti, si é chiarito che i parametri restituiti si riferiscono alla macchina virtuale *guest*; modificando, infatti, alcuni parametri fisici dell'*hardware* assegnato alla macchina virtuale, si é visto che anche le informazioni restituite dai comandi sopra descritti mutano. Questa strategia diviene, dunque, non accettabile, in accordo con quanto analizzato in 3.2.

3.3.2 Accesso ad internet

Un'altra possibilitá analizzata prevede l'accesso ad internet da parte dell'utente che vuole licenziare un determinato applicativo. In particolare, tale soluzione consiste nell'accesso ad un sito web indicato dal venditore per l'attivazione della licenza. Solo dopo la connessione a tale sito e l'inoltro della richiesta di attivazione, il venditore puó fornire l'autorizzazione all'utilizzo del *software*.

Risultati

Per quanto semplice e pratica, tale soluzione ha trovato da subito un grosso ostacolo: per motivi di sicurezza, i prodotti applicativi di interesse per l'azienda sono inseriti in ambienti privi di accesso a internet. Questa scelta é legata all'esigenza di proteggere e isolare le applicazioni da terzi non autorizzati. Anche questa strategia risulta non accettabile.

3.3.3 NAT virtuale

L'indirizzo Ethernet di una macchina fisica é un parametro che spesso viene usato come identificativo al quale agganciare la licenza, come accennato in 3.1.

Un'ulteriore strategia prevede la creazione di una rete NAT[4] virtuale che interconnette due macchine virtuali. La macchina virtuale sulla quale viene eseguito l'applicativo che si vuole licenziare viene configurata con due interfacce di rete:

1. BRIDGED, per l'accesso alla rete LAN;

2. NAT.

[4] *Network Address Translation*

La prima volta che la suddetta macchina virtuale si connette al *License Server* per la richiesta del licenziamento del prodotto che ospita, dal *License Server* sará possibile vedere l'indirizzo IP dell'interfaccia di rete fisica della macchina *host*. Attraverso un processo ARP[5] a partire dall'indirizzo IP prima ricavato sará possibile ottenere il MAC *address*.

Risultati

Per quanto la soluzione possa sembrare illuminante, l'ambiente vSphere non consente la definizione di interfacce di rete NAT per le macchine virtuali (*"...the ESX hypervisor does not provide a NATed network option"* [19]). Questa restrizione rende non accettabile la soluzione.

3.3.4 *System ID*

Secondo quanto analizzato in 3.1, risulta indispensabile trovare un identificativo della macchina fisica che ospita la macchina virtuale al quale agganciare la licenza. Un'ultima soluzione cui si é arrivati consiste nel generare un *System ID* identificativo di una macchina virtuale; tale *System ID* viene costruito in funzione di due parametri:

$$System\ ID = f(MAC_{host}, MAC_{LS})$$

dove MAC_{host} é il MAC *address* della macchina *host*, mentre MAC_{LS} é il MAC *address* del *License Server*. A livello di configurazione si richiede l'indirizzo IP della macchina *host* e, come in 3.3.3, si ricava il MAC *address* con protocollo ARP.

Risultati

É interessante osservare alcuni casi particolari che si possono presentare; ciascuno é stato opportunamente analizzato.

- Qualora l'indirizzo IP della macchina *host* ricevuto fosse di un'altra macchina, la licenza generata risulterebbe legata a tale macchina; qualsiasi tentativo di scambio della macchina *host* verrebbe subito rilevato, e la licenza bloccherebbe l'utente;

- qualsiasi tentativo di clonazione della macchina virtuale sulla stessa *subnet* fallirebbe dal momento che tale operazione non é consentita;

[5] *Address Resolution Protocol*

- qualora l'indirizzo IP richiesto in fase di configurazione fosse di una macchina virtuale, tale tentativo verrebbe immediatamente scovato. Avviene, infatti, un controllo sul MAC *address* ricavato dall'indirizzo IP ricevuto: tutti gli indirizzi Ethernet delle macchine virtuali VMWare rientrano in un *range* prestabilito [20], e di conseguenza se il MAC *address* ottenuto dovesse rientrare in tale *range*, il *Licensing System* bloccherebbe la richiesta di licenza in questione;

- esiste la possibilitá di forzare il MAC *address* di una macchina virtuale; per quanto questo tentativo possa risultare dannoso e incontrollabile, fino ad oggi non si é mai verificato tra i clienti dell'azienda tale caso.

Considerate e analizzate le varie possibilitá appena descritte, si é giunti alla conclusione che quest'ultima soluzione risultata accettabile per i prodotti ed il mercato dell'azienda.

Limiti della soluzione scelta

Attualmente esiste un caso in cui la soluzione scelta risulta non praticabile. Infatti, se tra la macchina *host* e la macchina sulla quale viene eseguito il *License Server* vi é un *router*, risulterá impossibile ricavare il *MAC address* dell'*host* che richiede il licenziamento di un prodotto. Il calcolo del *MAC address* della macchina *host* da parte del *License Server* avviene, infatti, attraverso due operazioni:

1. una prima operazione di *ping* con la quale il *License Server* raggiunge l'*host*;

2. una seconda operazione che permette di visualizzare l'*ARP table* del *License Server*: la lettura del *MAC address* é quindi immediata.

Nel caso in cui ci sia un *router* frapposto tra *License Server* e *host*, sebbene la prima operazione riesca con successo, l'indirizzo *MAC* che verrá visualizzato con la seconda operazione apparterrá al *router* e non alla macchina *host*. Poiché il *System ID* generato dipende strettamente dal *MAC address* della macchina *host*, questo problema meriterá in ambito aziendale ulteriori considerazioni per nuove possibili soluzioni.

Capitolo 4

Progettazione

Introduzione

Dopo aver analizzato i sistemi di virtualizzazione e, in particolare, la piattaforma VMWare *vSphere*, si sono affrontate le attivitá di raccolta e analisi di tutte le specifiche che si voleva il sistema di *licensing* vantasse.

Le interviste al *tutor* aziendale hanno permesso di raccogliere i requisiti dell'applicativo, in modo da determinare i punti da implementare interamente piuttosto che le funzionalitá da estendere. La strutturazione dei requisiti raccolti ha permesso di individuare le funzionalitá principali del *software* da implementare, specificandone le precise caratteristiche. L'ultimo *step* progettuale si é sviluppato nella rappresentazione delle specifiche di progetto, illustrando le caratteristiche architetturali mediante l'utilizzo dei diagrammi UML.

4.1 Requisiti strutturati

La prima attivitá nel lavoro di progettazione si é sviluppata nella raccolta delle specifiche del nuovo sistema di *licensing* da implementare. Tali requisiti sono stati successivamente analizzati e strutturati secondo le regole apprese nei corsi di Basi di Dati e di Ingegneria del Software. In particolare si é scelto un approccio di tipo incrementale: definiti e chiariti gli aspetti piú importanti, di volta in volta si sono aggiunte specifiche sempre piú dettagliate e particolareggiate.

Verranno ora presi in esame i seguenti requisiti strutturati cui si é giunti attraverso un'intensa attivitá di interviste e di raccolta di documentazione aziendale:

1. mantenimento della compatibilitá con il sistema di *licensing* esistente;

2. estensioni del sistema di *licensing*;

3. protezione della comunicazione tra *License Server* e applicazioni;

4. generazione del *System ID* a partire da parametri locali della macchina del cliente.

4.1.1 Mantenimento della compatibilitá con il sistema di *licensing* esistente

Vengono ora descritti il sistema di *licensing* esistente dal quale si é partiti per lo sviluppo del nuovo sistema e l'ipotesi di realizzazione di quest'ultimo. É indispensabile notare che la struttura descritta vale per ogni macchina fisica; nella situazione di partenza si assume che il sistema non sia inserito in ambiente virtuale.

Sistema di *licensing* esistente

Il parametro fisico della macchina sulla quale viene installato l'applicativo da licenziare é il MAC *address*; a partire da questo parametro, il *System ID Generator*, risiedente nella macchina del cliente, genera un *System ID* identificativo di tale macchina. Il *System ID* cosí generato viene fatto pervenire all'azienda; qui, il *License Generator* genera il *License File* a partire dal *System ID* ricevuto. Il *License File* viene consegnato al cliente il quale puó, quindi, attivare ed eseguire l'applicativo ora licenziato (si veda la Figura 4.1 (a)). Tutti gli applicativi forniti dall'azienda sono contraddistinti da un ID che li identifica univocamente; la richiesta, da parte del cliente, di un determinato applicativo avviene secondo la seguente modalitá: il cliente attraverso un'interfaccia grafica seleziona le applicazioni per le quali é interessato; sempre attraverso la stessa interfaccia genera il *License File* che contiene le informazioni relative alla propria scelta ed invia tale *file* all'azienda. In azienda, per ogni applicazione indicata dal cliente, viene selezionato il corrispondente ID, precisando sia il numero di *token* che le eventuali opzioni richieste. I *token* corrispondono a dei "gettoni" che dipendono strettamente dall'applicativo, mentre le opzioni individuano degli ulteriori servizi o estensioni che l'applicativo selezionato puó fornire. Il numero di *token* é un numero intero, mentre le *option* sono numeri in logica *bitwise*.

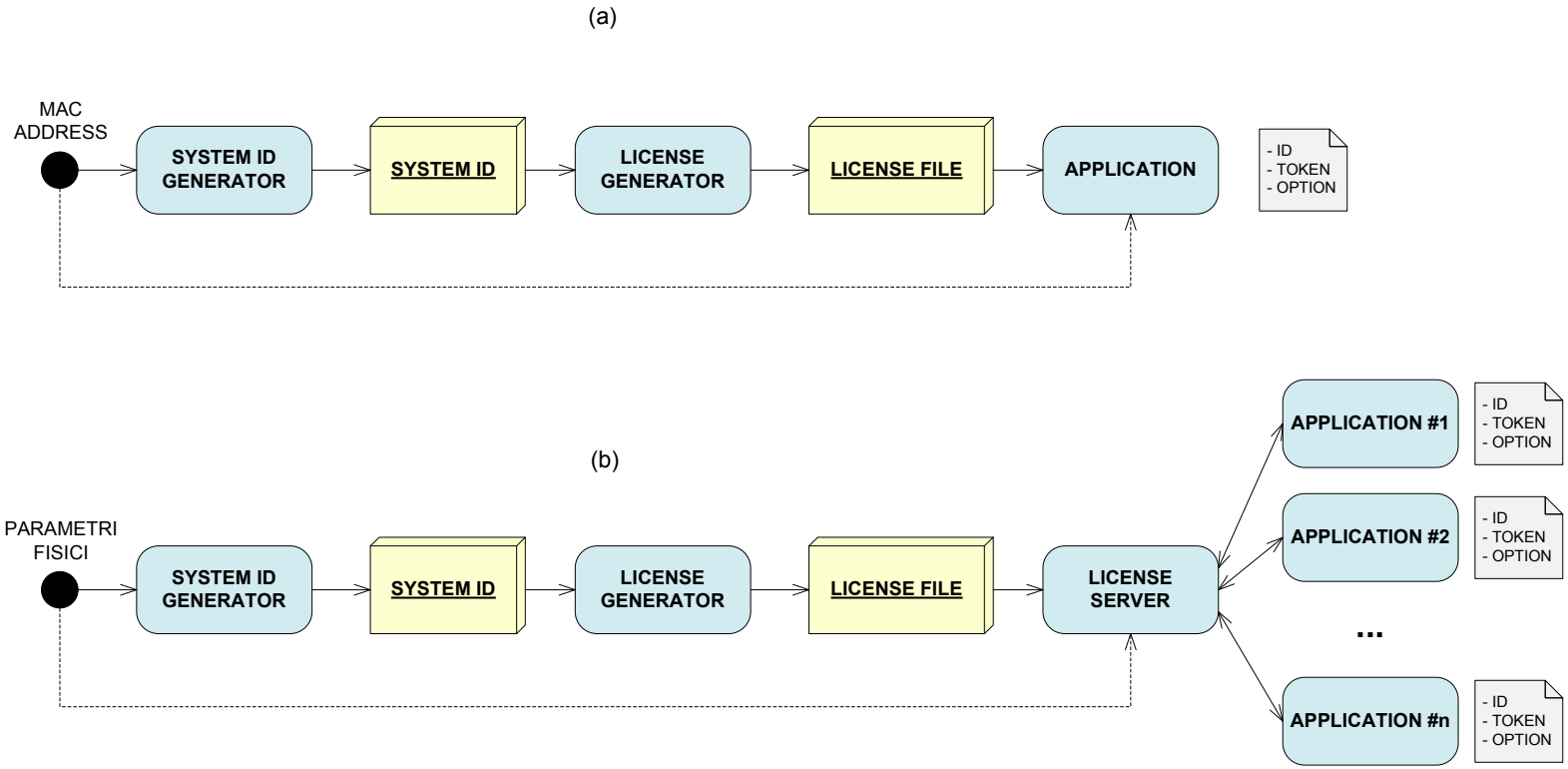

Figura 4.1: (a) Situazione attuale del sistema di *licensing*; (b) ipotesi di situazione finale del sistema di *licensing*

Limiti e problematiche del sistema di *licensing* esistente

Uno dei principali limiti del sistema di *licensing* appena descritto sorge nel momento in cui si vuole licenziare un'applicazione per due o piú macchine di uno stesso cliente; in questo caso, infatti, le due macchine devono essere licenziate separatamente: questo rende macchinosa un'operazione relativamente semplice.Inoltre, e questo é il limite principale, uno stesso *License File* non puó essere condiviso tra piú macchine.

Inoltre, va considerato un importante cambiamento architetturale che sta coinvolgendo gran parte delle aziende negli ultimi tempi (tra queste anche Mida solutions s.r.l.): il passaggio ad ambienti virtuali. Tale cambiamento é legato essenzialmente ai vantaggi che la virtualizzazione comporta, come descritto in 2.1.2.

Ipotesi di situazione finale del sistema di *licensing*

Sulla base di quanto riportato precedentemente, il sistema di *licensing* viene ristrutturato nel modo seguente.

A partire da un parametro fisico (in questa fase di analisi dei requisiti non vengono approfonditi i dettagli su questo parametro) della macchina sulla quale viene installato l'applicativo da licenziare (in presenza o meno di un ambiente virtuale), viene generato il *System ID*. Da questo, il *License Generator* genera il *License File*. Fin qui, nulla di nuovo; la novitá architetturale risiede nella presenza di un *License Server* che possa gestire le richieste di licenziamento degli applicativi da parte dei clienti. Si vuole che questo *License Server* risieda nella macchina del cliente e sia in grado di comunicare alle applicazioni i parametri di funzionamento (ID, *token* e *option*) sulla base dello stato dei *License Files* (si veda la Figura 4.1 (b)). Una prima importante considerazione risolve uno dei problemi prima illustrati: in questa nuova struttura del sistema di *licensing* non ha piú senso considerare dove risiede ogni applicazione; addirittura piú istanze della stessa applicazione possono coesistere sulla stessa macchina. Infine, si vuole che con il nuovo sistema di *licensing* sia mantenuta la compatibilitá con il sistema esistente.

4.1.2 Estensioni del sistema di *licensing*

Nel sistema di *licensing*, ad ogni *License File* é associata una durata; tale durata é unica per ogni *License File*. É richiesto che sia associata una durata di licenza diversa **per ogni** ID dell'applicazione. Attualmente la durata della licenza puó essere di due tipi:

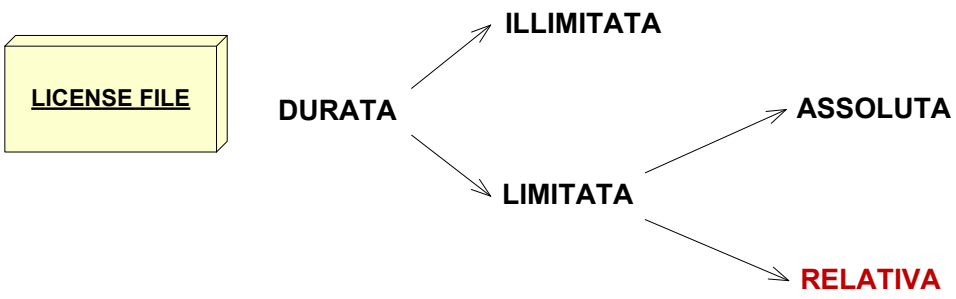

Figura 4.2: Estensione del sistema di *licensing*: aggiunta della data relativa

- illimitata;
- limitata.

Nel caso di durata illimitata, il prodotto fornito al cliente non conosce data di fine attivitá. Nel caso di data limitata, la licenza scade il giorno stabilito e in quella data il prodotto cessa la propria attivitá. É richiesta l'aggiunta della possibilitá di impostare una data relativa, ovvero un numero finito di giorni entro i quali il prodotto puó essere utilizzato (si osservi lo schema i Figura 4.2). L'inizio di tale intervallo di tempo coincide con l'istante di attivazione del prodotto.

4.1.3 Protezione della comunicazione tra *License Server* e applicazioni

Si richiede che la comunicazione tra *License Server* e applicazioni avvenga mediante un canale di comunicazione protetto; in particolare, si vuole che tali entitá comunichino attraverso una connessione socket.
É richiesta una comunicazione **affidabile, sicura** e **protetta**:

affidabile: i dati inviati dal mittente non devono arrivare danneggiati al destinatario;

sicura: tutti i dati inviati del mittente devono arrivare al destinatario;

protetta: la comunicazione non deve essere ascoltata da terzi: rischio *poisoning*[1].

Si richiede, quindi, un sistema di *encription* e di certificazione; il contesto é di ambiente *Enterprise*, non *Consumer* (si osservi la Figura 4.3).

4.1.4 Generazione del *System ID* a partire da parametri locali della macchina del cliente

Si richiede che il *System ID* sia ricavabile a partire dai parametri fisici della macchina nella quale é in esecuzione l'applicazione, anche se si tratta di una macchina virtuale. Si consideri che la generazione del *System ID* avviene in assenza di connessione a internet, come mostrato in Figure 4.4.

[1]Con il termine *poisoning* s'intende l'inquinamento volontario dei dati di un determinato sistema

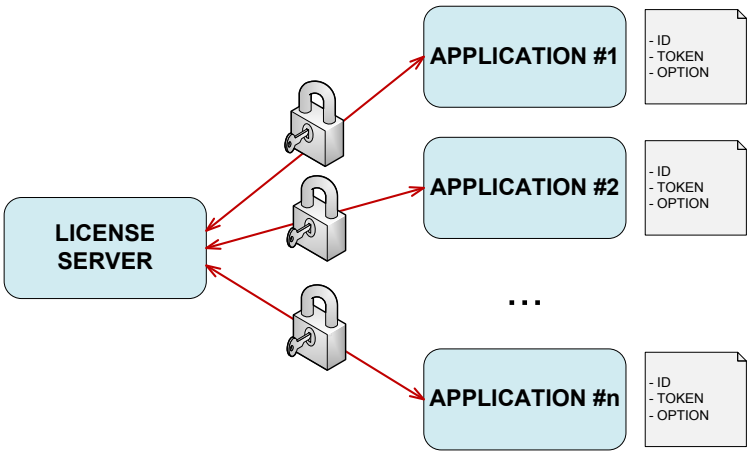

Figura 4.3: Sicurezza nella comunicazione tra *License Server* e applicazioni

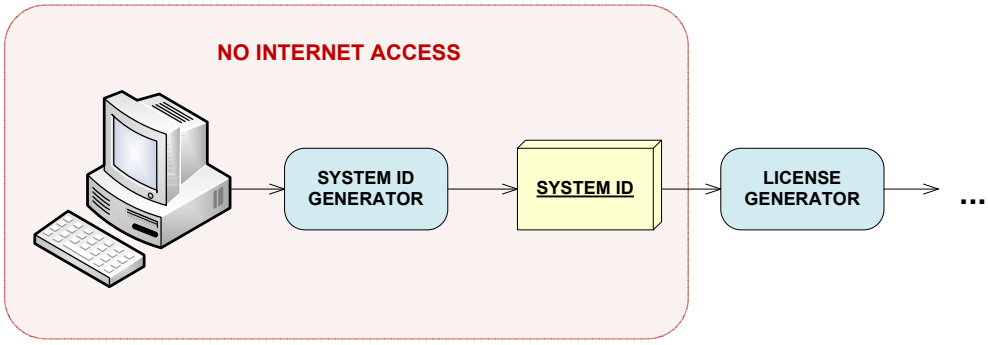

Figura 4.4: Generazione del *System ID* a partire da parametri locali della macchina del cliente in assenza di connessione ad internet

4.2 Requisiti funzionali

Raccolti e analizzati i requisiti del sistema, si é passati alla definizione dei requisiti funzionali; per far ció si é cercato di raggruppare i requisiti trovati in aree funzionali in modo da poter passare, successivamente, con facilitá e chiarezza alla fase di realizzazione. Le funzionalitá selezionate rappresentano gli aspetti cardine che compongono l'intero sistema di *licensing*:

- struttura dell'ambiente;
- caratteristiche della connessione tra *License Server* e applicazioni;
- richiesta e fornitura della licenza;
- *refresh* della licenza;
- configurazione del *License Server*;
- applicazioni;
- *option*;

4.2.1 Struttura dell'ambiente

Il License Server potrá essere eseguito come servizio in ambiente Linux e Windows o come macchina virtuale negli altri ambienti (Windows, vSphere, Linux).

Il License Server deve supportare i meccanismi di High Availability e Fault Tolerance (vMotion) di vSphere, descritti in 2.2.7 e 2.2.8.

4.2.2 Caratteristiche della connessione tra *License Server* e applicazioni

L'applicazione che richiede la licenza contatta il *License Server* del quale conosce l'indirizzo IP. Il *License Server* viene, quindi, a conoscenza dell'indirizzo IP dal quale arriva la richiesta e diviene in grado di comunicare con l'applicazione. Il *License Server* é un servizio (*daemon*) con il quale é possibile comunicare attraverso una connessione *socket*; attraverso questo *socket* tutte le applicazioni, quando vanno in esecuzione, chiedono delle informazioni relative alle licenze disponibili. La creazione (e il rilascio) della connessione avviene come mostrato nel diagramma di flusso di Figura 4.5

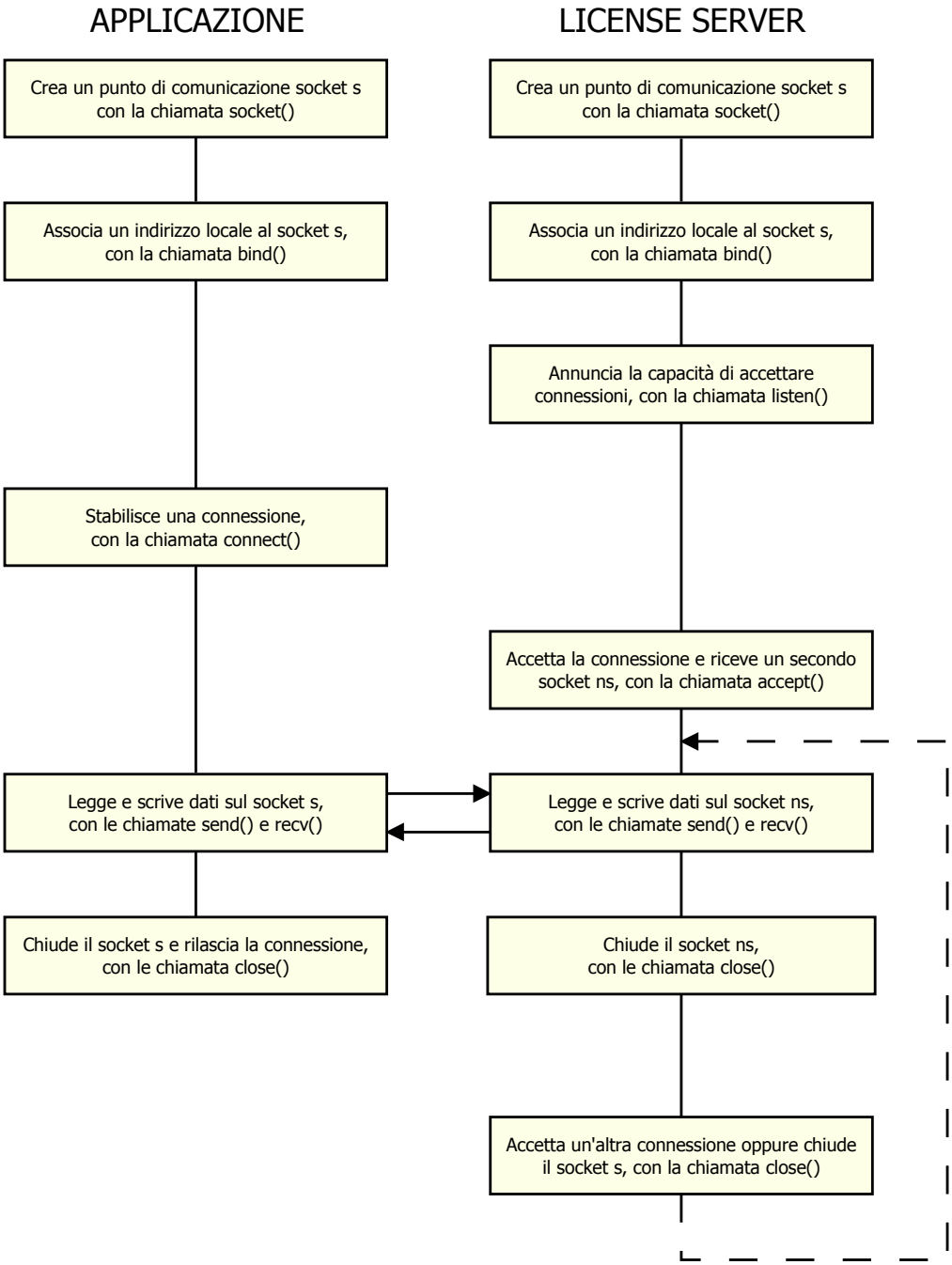

Figura 4.5: Diagramma di flusso della comunicazione tra *License Server* e una generica applicazione

La connessione tra *License Server* e applicazioni prevede un sistema crittografico che garantisca sicurezza e protezione alle informazioni scambiate. Ciascuna applicazione utilizza un modulo di libreria per comunicare con il *License Server*; questo permettete ad ogni applicazione di evitare l'onere di tutti i meccanismi di comunicazione.

4.2.3 Richiesta e fornitura della licenza

Il *License Server* gestisce le informazioni di licenza solo a livello sintattico; l'interpretazione semantica é demandata all'applicazione. In questo modo, ogni volta che viene creata una nuova applicazione, non si é costretti a cambiare *License Server*; il *License Server* gestisce le richieste in base ai valori dei soli parametri ID, *token* e *options*: sará poi compito dell'applicazione, relativa a quel determinato ID, interpretare correttamente il significato dei valori di tali parametri. Dopo che é stata instaurata la connessione tra *License Server* e applicazione, la comunicazione per la richiesta della licenza avviene secondo la seguente mimica:

- l'applicazione richiede il soddisfacimento delle proprie esigenze, espresse con la stessa sintassi del *License File*: tramite un *token* e tramite la mappa delle *options*;

- il *License Server* risponde con una configurazione di *token* e *options* che é in grado di soddisfare, tenendo conto di quanto giá autorizzato ad altre istanze aventi lo stesso ID. Ad esempio, se per un certo ID arriva una richiesta di 11 token e il *License File* ne prevede 20 ma ne ha giá assegnati 10, il *License Server* risponde con una configurazione di 10 token.

Il *License Server* prende le informazioni di configurazione e la/le chiave/i di licenza in locale; i *file* di configurazione sono strutturati come una sequenza di coppie chiave-valore. Il *License Server* quando viene contattato da un'applicazione per una richiesta di licenza, deve verificare la scadenza temporale di quella licenza. Se si tratta di una scadenza relativa deve verificare:

- se quella licenza non é ancora stata attivata, la attiva da quel momento e da lí decorre la durata;

- se é giá stata attivata, rientra nella normale logica delle scadenze (avrá una propria data di scadenza).

Sia in caso di ambiente Linux che di macchina virtuale, le informazione relative alla scadenza e all'attivazione saranno nascoste all'interno o dell'*Appliance* o della macchina virtuale.

4.2.4 *Refresh* della licenza

Si vuole supportare una funzionalitá che é la verifica a *runtime* della licenza: poiché la generica applicazione non deve tener traccia della scadenza temporale (non ha senso caricare l'applicazione dell'onere del controllo della scadenza), si richiede di realizzare una funzionalitá nella quale l'applicazione esegue un *refresh* periodico della licenza. Quindi, il *License Server* deve distinguere il caso in cui un'applicazione chiede per la prima volta una licenza, dal caso in cui chiede il *refresh* della propria licenza; il *License Server* memorizza le informazioni relative agli stati di assegnazione delle licenze in un'apposita struttura ad esso collegata. Tale struttura deve prevedere un'operazione di *reset* allo *startup*: ovvero, quando il *License Server* viene riavviato, la struttura dev'essere ripulita. A tal proposito, poiché le applicazioni gestite dal *License Server* sono qualche decina, non ha senso memorizzare la struttura degli stati d'assegnazione su disco, ma é ragionevole tenerne traccia in memoria.

4.2.5 Configurazione del *License Server*

License Server viene letta da *file* di testo residenti in locale: questi *file* saranno modificati tramite il portale web della piattaforma aziendale. Tutte le applicazioni sono gestite da portale web; attualmente questo portale é in fase di reingegnerizzazione, l'obiettivo é quello di avere un portale unico e omogeneo per tutte le applicazioni. Si vuole disaccoppiare il *License Server* (e quindi le applicazioni) dal portale web; ogni applicazione si basa solamente sui *file* di configurazione locali: qualsiasi modifica a questi *file* non interessa l'applicazione.

4.2.6 Applicazioni

Come giá detto, un'applicazione é caratterizzata da un ID che la identifica univocamente. Il valore del campo *token* relativo ad un'applicazione indica una particolare disponibilitá propria di quella stessa applicazione. Il popolamento della struttura che tiene traccia della disponibilitá avviene leggendo le informazioni contenute nei *License file*. In particolare, per una determinata applicazione, se giá presente nella struttura, il valore del campo *token* viene sommato al numero di *token* giá disponibili. Si vuole,

inoltre, che venga valutata la data di scadenza: questo valore viene aggiornato solo se la data di scadenza relativa all'applicazione in questione risulta minore (cioé scade prima) rispetto alla data di scadenza giá presente nella struttura.

Attualmente il valore dell'ID di un'applicazione é un numero a due cifre.

Il valore del campo *token* in una richiesta per un determinato applicativo puó assumere diversi valori, ciascuno con un significato specifico:

- se il valore é 0, significa che vengono richiesti tutti i *token* disponibili per l'applicazione selezionata;
- se il valore é -1, significa che viene chiesto il rilascio di tutti i *token* precedentemente assegnati per l'applicazione selezionata;
- se il valore é uguale al numero di *token* assegnati precedentemente, viene interpretata come una richiesta di *refresh*;
- infine, se il valore é minore o maggiore (non 0, -1 o uguale) al numero di *token* giá assegnati, vengono rispettivamente rilasciati o assegnati *token* secondo la disponibilitá.

A seconda della disponibilitá, il campo *error_code* del pacchetto di risposta conterrá un valore che indicherá:

- richiesta accettata;
- richiesta parzialmente accettata;
- richiesta rifiutata.

4.2.7 *Option*

Tra gli applicativi forniti dall'azienda esiste la possibilitá che alcuni di questi offrano delle opzioni aggiuntive. Tali opzioni possono essere richieste dal cliente per aggiungere maggiori funzionalitá ad un'applicazione scelta. La gestione di queste *Option* avviene nel modo seguente: ogni *Option* é caratterizzata da un ID (numero intero) che la identifica univocamente. Attualmente, un *Option ID* é un numero composto da tre cifre, a differenza di un *App ID* che é composto da due cifre. Il numero di *token* che viene indicato per una *Option* non viene interpretato come un'informazione sulla disponibilitá, come nel caso delle applicazioni. Viene, invece, interpretato come un numero con logica *bitwise*; il valore, infatti, convertito in un

numero binario, permette di selezionare la configurazione scelta. Il campo *Options* rappresenta un ulteriore controllo per verificare se si tratta di un'opzione o di un'applicazione.

Risulta indispensabile, quindi, porre attenzione sull'aggiornamento della struttura che tiene traccia della disponibilitá: il valore del campo *token* di una *Option* non viene sommato al valore presente per quella stessa *Option* nella struttura suddetta. Viene, invece, valutata la data di generazione del *License File* contenente l'*Option* in questione; solo se tale data risulta posteriore alla data di generazione del *License File* contenente l'*Option* presente nella struttura, viene sostituita l'*Option* con il proprio valore di *token*.

4.3 Specifiche di progetto

L'ultima fase dell'attivitá di progettazione si é sviluppata nella definizione formale, mediante diagrammi UML[2], sia dell'intero sistema di *licensing* attraverso i diagrammi dei componenti e di *deployment*, sia dei moduli principali che compongono il sistema attraverso i diagrammi di attivitá.

4.3.1 Diagramma dei componenti del sistema di *licensing*

I diagrammi dei componenti sono utili quando si vuole suddividere il sistema che si sta analizzando in moduli distinti e si vogliono documentare le relazioni che legano questi moduli. Nel sistema preso in esame, si sono individuati tre componenti principali: il *License Server*, il *License Client* e l'applicazione. Questi componenti costituiscono la base del sistema di *licensing*: il *License Server* riceve una richiesta di licenziamento dall'applicazione attraverso un *License Client* che funge da intermediario, interpreta la richiesta da parte del cliente, e la inoltra al *License Server* attraverso un'apposita interfaccia. Da quest'ultimo riceve una risposta la quale viene poi inoltrata all'applicazione. Da notare che *License Server* e applicazione sono **processi**, mentre il *License Client* é una **libreria**, in accordo con 4.2.2. La comunicazione tra *License Server* e *License Client* é permesso grazie ad una connessione TCP *socket* criptata, che protegge i dati in transito. La richiesta da parte dell'applicazione viene costruito sulla base di alcuni parametri che questa dev'essere in grado di leggere da un *file* o da un *database* apposito. Dopo che il *License Server* riceve

[2]Unified Modeling Language™, `http://www.uml.org/`

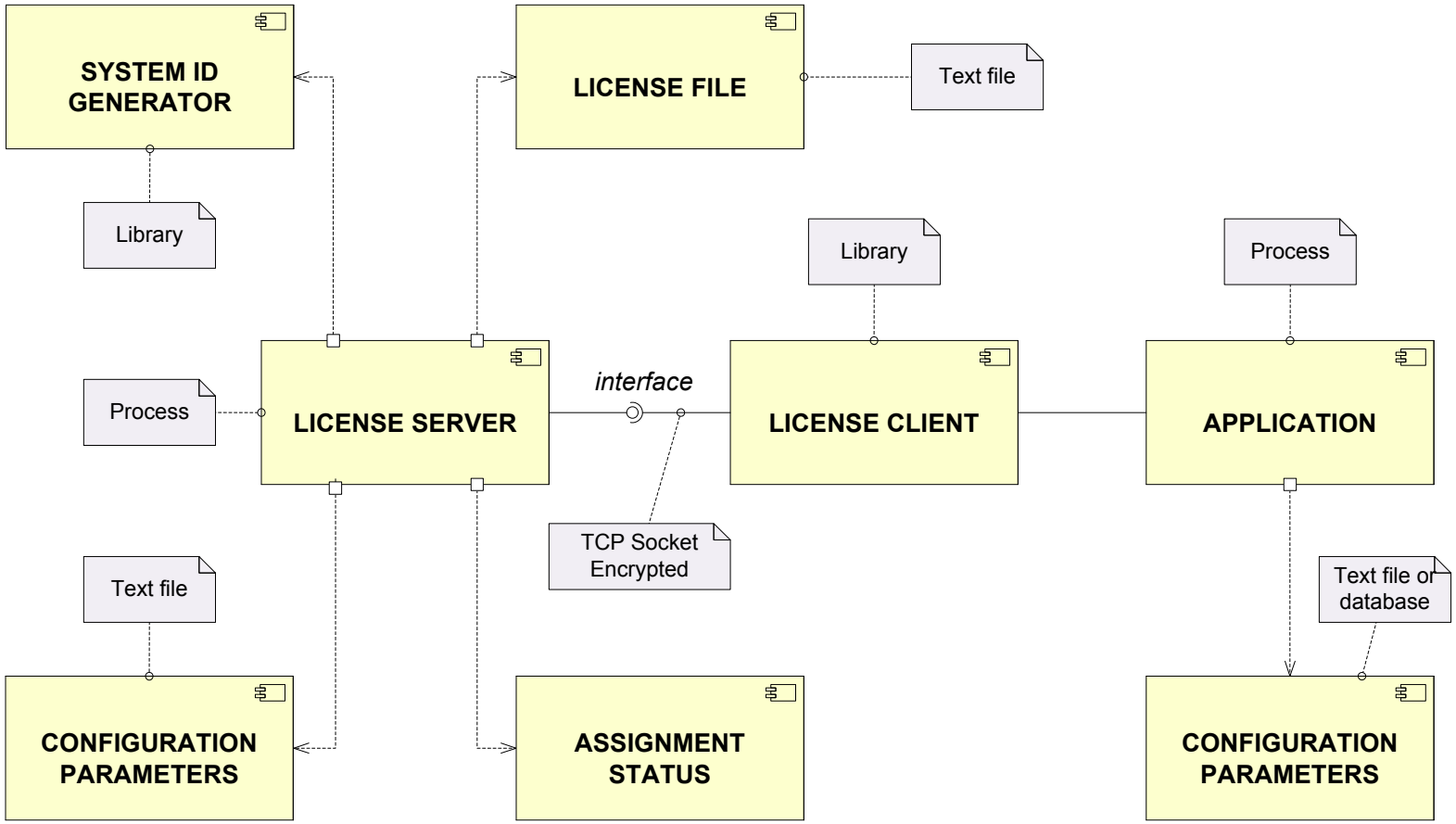

Figura 4.6: Diagramma dei componenti dell'intero sistema di *licensing*

una richiesta, deve compiere alcune operazioni prima di poter fornire una risposta; queste operazioni sono:

- generazione del *System ID* della macchina del cliente che si vuole licenziare; operazione compiuta dall'apposita libreria *System ID Generator*;
- lettura dei *License Files* siti in una cartella locale, il cui *path* viene passato esplicitamente al *License Server*;
- lettura di alcuni parametri di configurazione da un *text file* residente in locale;
- popolamento di apposite strutture che costituiscono lo stato di assegnazione delle licenze.

Tutti i componenti descritti sono schematizzati in Figura 4.6.

4.3.2 Diagramma di *deployment* del sistema di *licensing*

Si é scelto di inserire anche il diagramma di *deployment* del sistema di *licensing* poiché permette di documentare la distribuzione fisica del sistema, mostrando i vari pezzi di *sostware* in esecuzione sulle macchine fisiche. Nella Figura 4.7, i box rosa corrispondono ai **nodi** del sistema; un nodo rappresenta una qualsiasi entitá fisica che sia in grado di mandare in esecuzione del *software*. I nodi possono essere di due categorie:

dispositivo é sempre *hardware*, e puó essere un computer completo oppure un componente hardware piú semplice collegato al sistema;

ambiente di esecuzione é un pezzo di *software* che puó contenere ed eseguire altro *software*.

Nel sistema preso in esame, i nodi dispositivi sono il *Server*, che ospita il *License Server*, e la macchina fisica o virtuale che ospita l'applicazione da licenziare. I nodi ambiente, invece, sono: il modulo dello stato di assegnazione, il *License Server* e il modulo dei parametri di configurazione del *License Server*, appartenenti al nodo *Server*, mentre il *License Client*, l'applicazione e il modulo dei parametri di configurazione dell'applicazione per il nodo macchina. Da notare che il nodo applicazione é un'istanza (o multi-istanza) d'applicativo. In azzurro, invece, sono rappresentati gli **elaborati** (*artifact*) che rappresentano manifestazioni fisiche del software

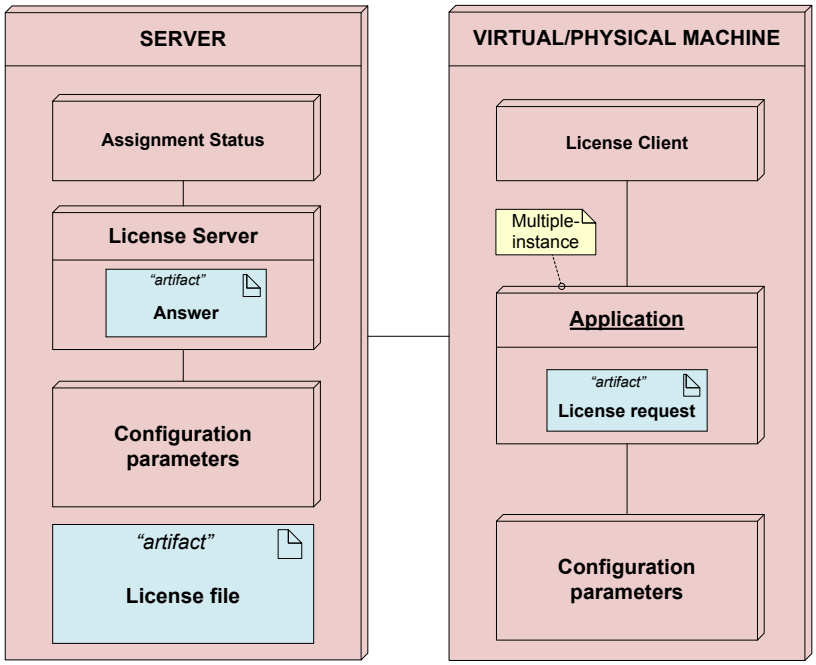

Figura 4.7: Diagramma di *deployment* dell'intero sistema di *licensing*

(tipicamente sono *file*); quando un elaborato viene disegnato all'interno di un nodo indica la propria posizione nel sistema durante l'esecuzione. Infatti *License request* e *Answer* rappresentano i prodotti rispettivamente dei nodi *applicazione* e *License Server*. L'elaborato *License File* costituisce un *file* proprio dell'intero nodo dispositivo *Server*.

4.3.3 Diagramma di sequenza del sistema di *licensing*

Fino ad ora non si é precisato come avviene la mimica tra *License Client* e *License Server*, e quali siano le informazioni che compongono una generica richiesta del primo e la risposta a tale richiesta del secondo.

Il pacchetto di richiesta di licenziamento é composto da sei campi:

1. AppID,
2. Option,
3. Token,
4. PackID,
5. Port,
6. IPaddress,

I primi tre campi seguono la logica ampiamente illustrata in 4.2.3. Il quarto campo, *PackID*, é un numero che permette di distinguere una nuova richiesta da una richiesta di *refresh*, specifica illustrata in 4.2.4; permette inoltre di distinguere richieste distinte per lo stesso applicativo provenienti dalla stessa macchina (quindi con stesso campo *Port* e indirizzo IP). Quando il numero di *PackID* é 0 rappresenta una nuova richiesta, quando é diverso da 0 rappresenta una richiesta di *refresh*. Il quinto campo, *Port*, é un numero che indica la porta sulla quale il *License Client* rimane in attesa per la risposta del *License Server*. Infine, il sesto campo rappresenta l'indirizzo IP dal quale proviene la richiesta di licenziamento.

Il pacchetto di risposta é composto da sei campi:

1. AppID,
2. Option,
3. Token,
4. AssPackID,

5. ExpDate,
6. ErrCode,

Il pacchetto che il *License Server* ritorna, contiene le informazioni relative alla risposta alla richiesta fatta. In base alla disponibilitá, per un determinato *AppID* saranno fornite le *Options* e i *Token* indicati; viene assegnato un nuovo *PackID* in modo da poter distinguere in futuro una richiesta di *refresh* per l'applicativo in questione. Nel quinto campo viene indicata la data di scadenza, e nel sesto campo viene inserito un codice d'errore che permette al *License Client* di interpretare correttamente i valori dei campi del pacchetto di risposta.

Il diagramma di sequenza di Figura 4.8 illustra efficacemente la successione delle azioni di richiesta e risposta che compongono l'interazione tra il *License Server*, *License Client* e applicazione.

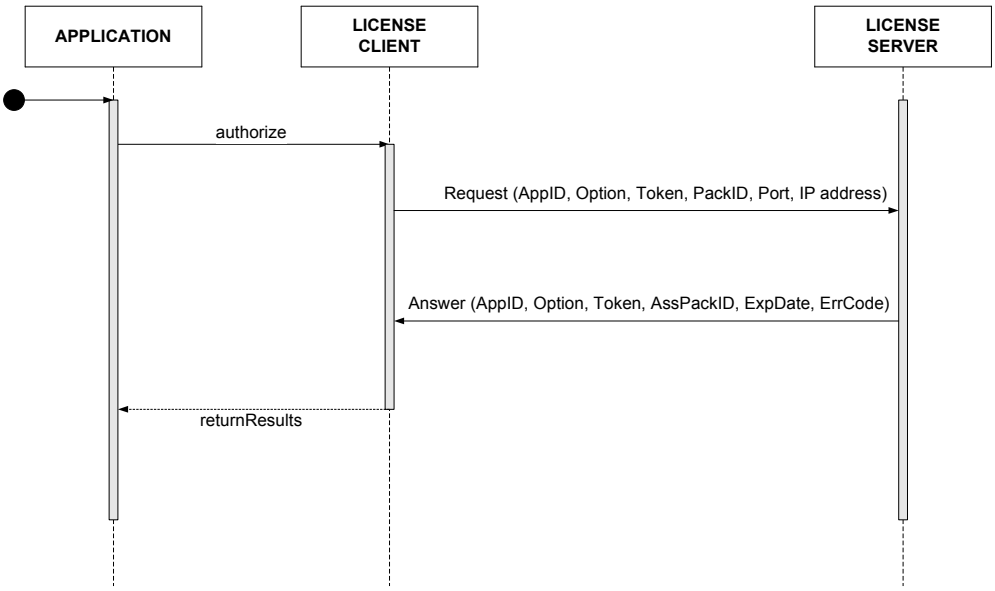

Figura 4.8: Diagramma di sequenza dell'intero sistema di *licensing*

4.3.4 Diagramma di attivitá del *License Client*

Il diagramma di attivitá di Figura 4.9 illustra fedelmente il ciclo di vita del *License Client*: ricevuta la richiesta di autorizzazione dall'applicazione, il *License Client* crea una nuova *socket* e invia una richiesta di connessione al *License Server*. Se quest'ultimo non é in ascolto, termina l'esecuzione;

altrimenti invia un pacchetto di richiesta al *License Server* nella forma illustrata precedentemente in 4.3.3. Se non riceve risposta entro un intervallo di tempo prefissato termina l'esecuzione. Se, invece, riceve una risposta, la inoltra all'applicazione non prima di aver chiuso la socket precedentemente aperta.

4.3.5 Diagramma di attivitá del *License Server*

Il *License Server* dopo esser stato avviato crea una socket sulla quale si mette in ascolto; da questo momento rileva tutte le richieste di connessione provenienti dal *License Client*. Ricevuta una richiesta di connessione, legge il pacchetto di richiesta: a questo punto inizia ad elaborare la risposta sulla base dello stato d'assegnazione delle licenze. Esegue il *check* della licenza, aggiorna le tabelle dello stato d'assegnazione delle licenze, costruisce il pacchetto di risposta, secondo la logica illustrata in 4.2.3. Composto il pacchetto lo invia sulla socket e si rimette in attesa di una nuova richiesta di connessione. L'intero processo descritto é illustrato nel diagramma di attivitá di Figura 4.10.

4.3.6 Diagramma di attivitá del modulo di gestione dei *License Files*

L'ultimo diagramma di attivitá illustra come avviene la gestione dei *License Files*. Viene aggiornata la lista dei *License Files* disponibili contenuti in una cartella locale: se sono presenti, uno a uno vengono selezionati; per ciascuno viene eseguita l'operazione di *unscrumble* della chiave di licenza, viene aggiornata la tabella delle licenze e la tabella dello stato d'assegnazione. Queste operazioni vengono descritte nel diagramma di attivitá di Figura 4.11.

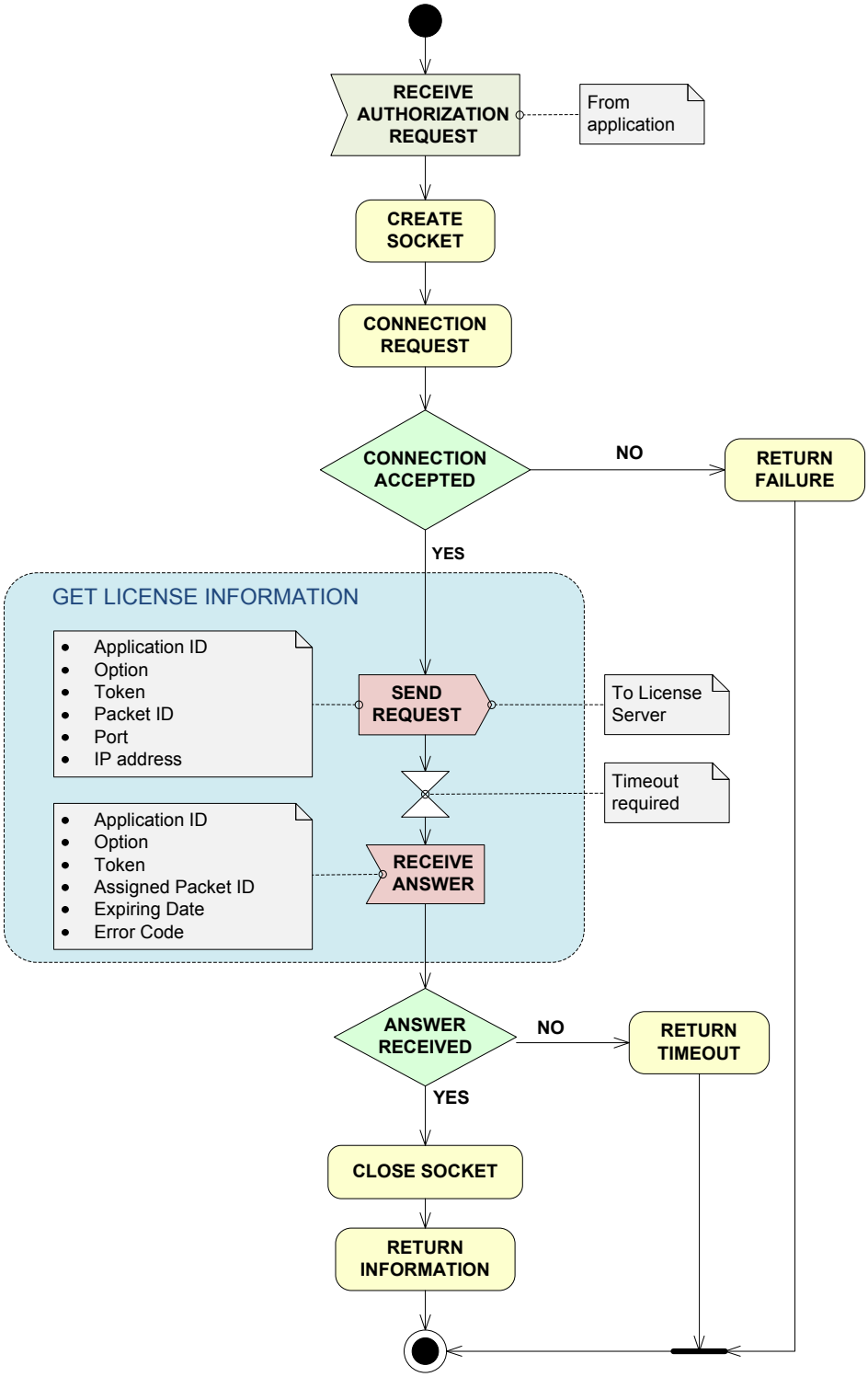

Figura 4.9: Diagramma di attivitá del License Client

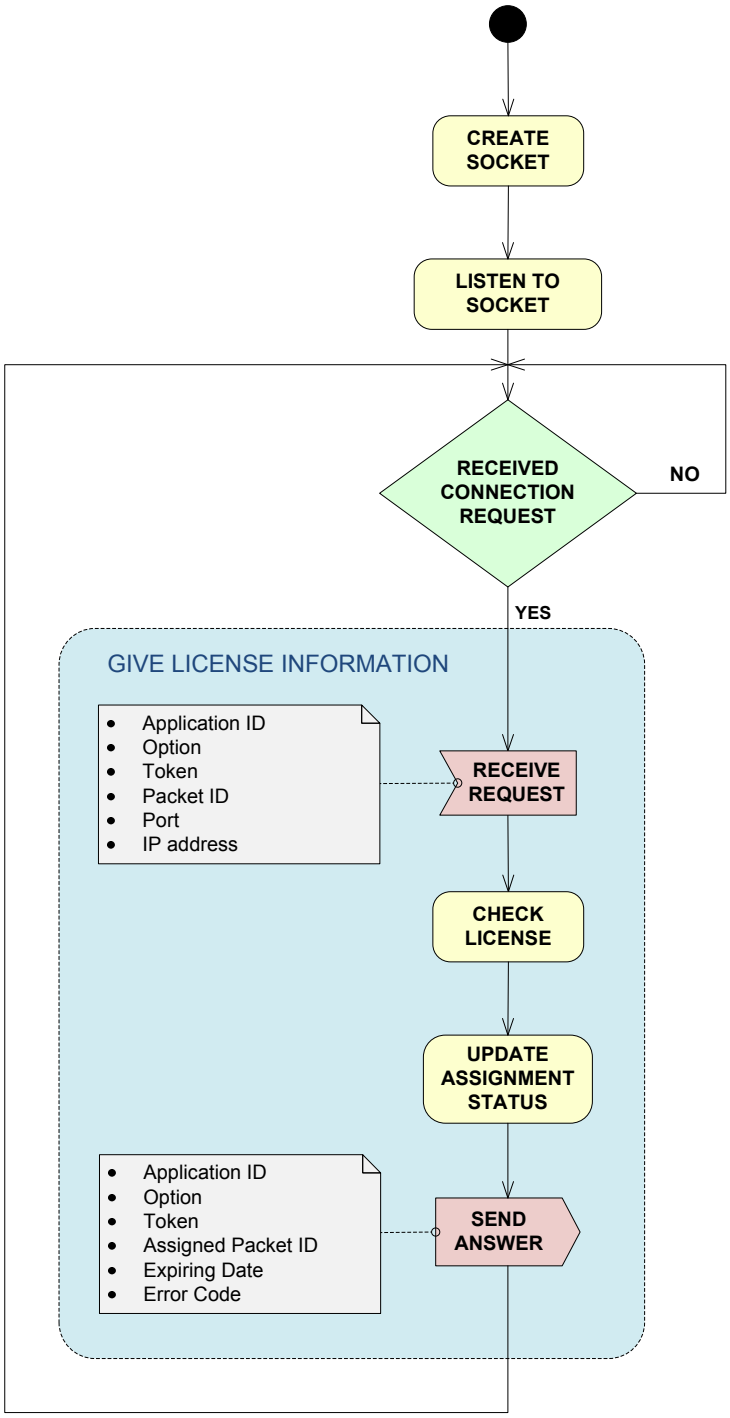

Figura 4.10: Diagramma di attivitá del License Server

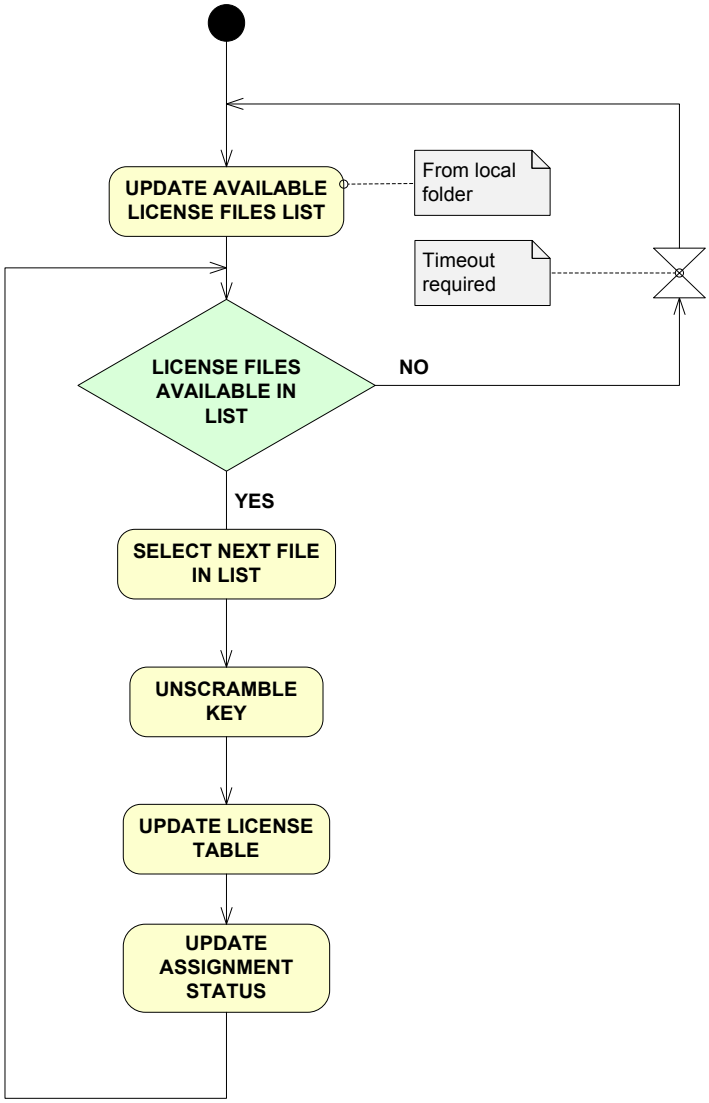

Figura 4.11: Diagramma di attivitá del modulo di gestione dei License Files

Capitolo 5

Realizzazione

Introduzione

L'implementazione del *software* fino ad ora progettato é stata l'ultima fase dell'esperienza di tirocinio. La possibilitá di lavorare all'interno di un *team* di sviluppo specializzato ha garantito un costante miglioramento dell'applicativo. Le riunioni aziendali atte a tener traccia dell'evoluzione e dei cambiamenti del *software* hanno permesso di abbracciare determinate scelte implementative, evitando, nella maggior parte dei casi, il rischio di incorrere in errori sia funzionali che sintattici.

In questo capitolo vengono presentate le scelte implementative seguite per le principali funzionalitá dell'intero sistema, riportando per queste i pezzi di codice che sono ora effettivamente presenti nel sistema di *licensing* aziendale.

5.1 Implementazione

Per quanto complesse, le scelte implementative per la realizzazione del *License Server* si basano su alcuni aspetti principali che ne caratterizzano la strutturazione. Questi aspetti sono frutto di un'attenta analisi compiuta in fase di definizione delle specifiche.

Verranno ora analizzati questi aspetti, rendendo chiara la successiva illustrazione dell'intera struttura; verranno poi presi in esame gli aspetti critici che hanno richiesto particolare attenzione durante l'attivitá di sviluppo.

5.1.1 Aspetti principali

Secondo quanto illustrato in 4.3.1, si é dovuta considerare l'esigenza di realizzare il *License Server* come un solo processo; questo requisito ha, da subito, orientato la scelta implementativa dell'intero sistema verso una struttura *multithreaded*. É risultato, quindi, indispensabile affrontare considerazioni in merito alla programmazione concorrente, per una corretta gestione delle risorse condivise. Un altro aspetto importante riguarda la gestione dei pacchetti di richiesta, in entrata al *License Server*. Si é cercata una soluzione che evitasse la perdita di pacchetti in arrivo dal *License Client*: ció é stato reso possibile dall'implementazione di due *thread* e di una struttura di dati apposita, da questi condivisa. Un'altra specifica richiedeva di differenziare due o piú richieste di licenziamento per la stessa applicazione, provenienti dalla stessa macchina: la gestione di questo caso particolare é stata resa possibile dall'interazione congiunta di due tabelle che registrassero e mantenessero aggiornati sia lo stato delle licenze che le richieste giá effettuate.

5.1.2 Architettura generale

Il diagramma di Figura 5.1 illustra l'intera struttura del *License Server*.

La classe *ApplicationMain* é la classe "nucleo" dell'intero programma: all'interno del metodo *main* viene istanziato un oggetto della classe *LicenseManager*. Questa classe rappresenta il punto di partenza di tutto l'applicativo: da qui infatti vengono avviati i *thread Listener, RequestManager* e *LicFileReader*, e viene istanziato un oggetto della classe *ReqStatusManager*. All'avvio del *License Server*, la classe *LicenseManager* legge e aggiorna alcuni parametri di configurazione (livello dei *log*, indirizzo IP della macchina, *path* della cartella in cui sono contenuti i *License Files*).

Il *thread Listener* ha il compito di costruire una connessione *socket* e di porsi in ascolto su una porta preconfigurata; all'arrivo di un pacchetto di richiesta, dopo un modesto e veloce controllo dimensionale, questo viene immediatamente copiato in un *buffer* apposito. Dopo la copia il *thread Listener* si rimette in ascolto sulla medesima porta. Il *buffer* su cui viene copiato il pacchetto di richiesta é condiviso con il *thread RequestManager*; tale condivisione é controllata da due semafori. Un primo semaforo (*semPckReceived*) serve al *Listener* per avvertire il *RequestManager* dell'avvenuta copia di un nuovo pacchetto nel *buffer* condiviso; un secondo semaforo (*semMutex*) gestisce l'accesso per le operazioni di scrittura e lettura al *buffer*.

Il *thread RequestManager*, una volta letto il pacchetto dal *buffer* suddetto, ne elabora il contenuto; prima di descrivere come avviene tale elaborazione, é indispensabile illustrare le due strutture che costituiscono lo stato d'assegnazione (menzionato in 4.3.5): la *License Status Table* e la *License Request Table*. La *License Status Table* é composta dai campi:

- AppID
- Option
- TotToken
- AssignedToken
- Scadenza

Tale struttura viene popolata dal *thread LicFileReader*; questo *thread* esegue nell'ordine le seguenti operazioni:

1. calcola il MAC *address* della macchina *host* a partire dall'indirizzo IP;
2. verifica che non appartenga al *range* VMWare (come descritto in 3.3.4);
3. solo se non appartiene a tale *range*, ne calcola il *System ID*;
4. con il *path* della cartella dei *License Files* (parametro acquisito dal *LicenseManager*), estrae da ogni *License File* la *License Key*;
5. esegue l'operazione di *unscrumble* su tale chiave, e ne ricava un *System ID*;
6. confronta il *System ID* ricavato con quello prima calcolato;
7. soltanto se il confronto ha esito positivo, popola la *License Status Table* secondo le informazioni contenute nella *License Key*.

La lettura e l'aggiornamento della *License Status Table* avvengono ciclicamente ad ogni scadenza di un intervallo di tempo opportunamente impostato.

L'elaborazione del pacchetto da parte del *thread RequestManager* avviene confrontando i parametri richiesti con la disponibilitá delle licenze della *License Status Table*; in particolare, viene eseguito un controllo sulla disponibilitá dei *token* per l'*AppID* selezionata. L'accesso alla *License*

Status Table é controllato da un apposito semaforo. La risposta potrá essere di tre tipi: richiesta soddisfatta, richiesta parzialmente soddisfatta e richiesta non soddisfatta.

Essendoci la possibilitá che possano giungere al *License Server* due o piú richieste per una stessa applicazione dalla stessa macchina, nel pacchetto di richiesta é stato inserito un campo, *Packet ID*, che permette di distinguere tali richieste. La distinzione é resa possibile grazie alla presenza di una seconda tabella, la *License Request Table*. Questa tabella é composta dai campi:

- AppID
- ReqID
- IP
- AssignedToken
- LastKeepAlive

Ogni richiesta soddisfatta (anche parzialmente) viene registrata su questa tabella, in modo da poter riconoscere future richieste di *refresh*: nel pacchetto di risposta, infatti, viene assegnato un nuovo *Packet ID* (solo in caso di richiesta parzialmente o completamente soddisfatta), che sará l'*ID* con cui l'applicativo dovrá chiedere il *refresh* della licenza.

É da precisare che la manutenzione della *License Request Table* é un compito assegnato al *thread RequestManager*: questo, infatti, esegue ciclicamente un controllo sulla scadenza di ogni *entry* della tabella (campo *LastKeepAlive*), ne calcola la distanza temporale dall'istante in cui viene eseguita tale verifica, e soltanto se questo valore risulta maggiore di un intervallo di tempo impostato elimina l'*entry* dalla tabella.

Costruito il pacchetto di risposta, il *thread RequestManager* crea una nuova connessione *socket* e invia il pacchetto al *License Client* sulla porta che era stata opportunamente indicata per la risposta.

5.2 Scelte implementative

Si passa ora ad un'analisi piú approfondita del *software* sviluppato, illustrandone i punti piú importanti e riportando le scelte implementative adottate.

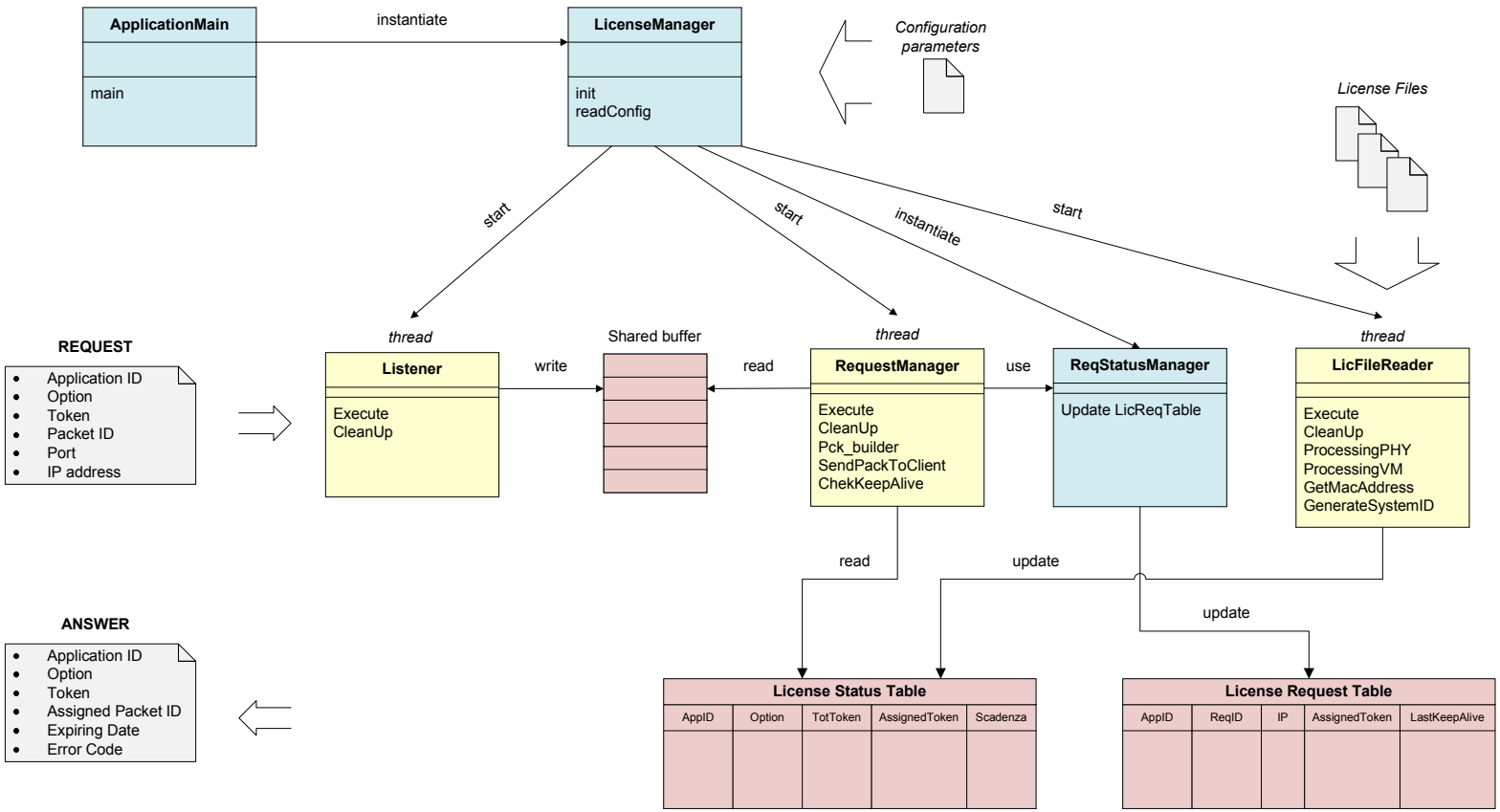

Figura 5.1: Struttura generale del *License Server*

5.2.1 *Socket* TCP

Sia analizzano ora i dettagli teorici e implementativi della connessione *socket* tra *License Server* e *License Client*.

Protocolli TCP e UDP

Il protocollo di riferimento adottato nel *software* é il protocollo TCP/IP, lo standard su cui poggia l'intera infrastruttura di internet. L'altro protocollo (UDP), basato su datagrammi, consente l'invio di pacchetti di dimensioni variabili ed é, per alcune applicazioni, relativamente piú veloce, ma non garantisce l'invio di pacchetti ordinati, né l'effettivo recapito a destinazione dei pacchetti stessi (si tratta infatti di un modello *connectionless oriented*, a differenza del TCP che é *connection oriented*). Poiché il *License Server* da sviluppare é un'applicazione *internet-oriented*, si adotterá la famiglia protocollare che in C++ identifica il dominio di comunicazione internet (definito in `<sys/socket.h>` come `AF_INET`, per il formato di indirizzi IPv4, e `AF_INET6`, per il formato IPv6).

Indirizzi IP

L'indirizzo IP identifica univocamente una macchina all'interno di una rete, e consiste (almeno nella versione 4 del protocollo, versione universalmente usata da anni in tutte le reti) in 4 gruppi di numeri che possono andare da 0 a 255 (in esadecimale 0,...,FF). Un indirizzo IP occupa quindi complessivamente 32 bit (4 byte) in memoria. Per poter utilizzare indirizzi IP é necessario passare la stringa che corrisponde all'IP alla funzione `inet_addr`, definita in `<arpa/inet.h>`.

```
#include <arpa/inet.h>

.....

in_addr_t addr;
struct in_addr a;

addr = inet_addr("127.0.0.1")

// Viene associata al membro s_addr della struttura
   in_addr la variabile appena associata all'indirizzo
a.s_addr=addr;
```

In `<netinet/in.h>` é definita la costante `INADDR_ANY`, che identifica un qualsiasi indirizzo IP (usato nel codice dei server per specificare che l'applicazione puó accettare connessioni da qualsiasi indirizzo).

Un aspetto importante nell'instaurazione della conessione *socket* é l'utilizzo di un membro della struttura `in_addr`. Tale struttura (relativamente scomoda e di per sé poco utile, ma preservata nella gestione degli indirizzi in C++ per una compatibilitá con il passato) contiene gli indirizzi di rete, ed é cosí definita:

```
struct in_addr
{
    u_long s_addr;
}
```

`s_addr` conterrá l'indirizzo ottenuto con `inet_addr`.

Porte

Per poter effettuare una connessione non basta un indirizzo IP e il protocollo da usare, é necessario anche specificare la porta dell'*host* alla quale si desidera collegare il *socket*, ovvero il servizio da richiedere. In definitiva, quindi, per costruire un socket per la comunicazione tra un *client* e un *server* servono:

- il protocollo per la comunicazione (TCP, UDP);
- l'indirizzo IP di destinazione;
- la porta su cui effettuare il collegamento.

Per poter utilizzare un *socket* in un programma si fa ricorso alle strutture `sockaddr`, definite in `<sys/socket.h>`. La struttura `sockaddr` di riferimento é strutturata nel modo seguente:

```
struct sockaddr
{
  // Famiglia del socket
  short sa_family;

  // Informazioni sul socket
  char sa_data[];
}
```

In questa sede si utilizza la struttura `sockaddr_in`, convertendola in `sockaddr`, quando richiesto, tramite operatori di *cast*:

```
struct sockaddr_in
{
  // Flag che identifica la famiglia del socket, in
     questo caso AF_INET
  short sa_family;

  // Porta
  short sin_port;

  // Indirizzo IP, memorizzato in una struttura di tipo
     in_addr
  struct in_addr sin_addr;

  // Riempimento di zeri
  char sin_zero[8];
}
```

Vi sono caratteristiche in questa struttura quantomeno curiose e apparentemente obsolete e ridondanti, ma conservate per tradizione e per compatibilitá con il passato. In *primis* il riferimento alla struttura `in_addr` (illustrata precedentemente) per memorizzare l'indirizzo IP, quando si sarebbe potuto ricorrere ad una variabile *long*. Il riferimento a questa struttura all'interno di `sockaddr` é uno dei piú profondi misteri della tradizione Unix. In *secundis*, il riempimento della struttura con una stringa (`sin_zero`) che non fa altro che contenere degli zeri, o comunque caratteri spazzatura. Ció é necessario per rendere la dimensione della struttura pari esattamente a 16 *byte*, in modo da poter effettuare senza problemi il cast da `sockaddr` a `sockaddr_in` e viceversa (sono della stessa dimensione).

Inizializzazione dell'indirizzo

Per inizializzare l'indirizzo all'interno della nostra applicazione si dovrá, quindi, ricorrere ad un membro della struttura `sockaddr_in`, specificando, al suo interno, famiglia protocollare (`AF_INET`), porta e indirizzo IP. Per fare ció, conviene creare una procedura esterna al main che esegua:

```
void addr_init (struct sockaddr_in *addr, int port, long
    int ip)
{
// Inizializzazione del tipo di indirizzo (internet)
addr -> sin_family = AF_INET;

// Inizializzazione della porta
```

```
7 addr -> sin_port = htons ((u_short) port);
8
9 //Inizializzazione dell'indirizzo (passando per la
    struttura in_addr)
10 addr -> sin_addr.s_addr = ip;
11 }
```

Creazione del *socket* e connessione

Per l'inizializzazione di un *socket* si ricorre, invece, alla primitiva `socket`, che prende come parametri il dominio a cui fare riferimento (nel caso di un'applicazione internet `AF_INET`), il tipo di *socket* (`SOCK_STREAM` nel caso di TCP) e il protocollo (impostando questo parametro a zero, il protocollo viene scelto in modo automatico). La funzione ritorna un valore intero che identifica il *socket* (*socket descriptor*) e che verrá usato in seguito per le connessioni, le letture e le scritture, allo stesso modo di un descrittore per file, per processi o per pipe. La funzione ritorna -1 nel caso in cui non sia stato possibile creare il *socket*.

Per la chiusura del *socket* si ricorre alla primiva `close`, passando come parametro il descrittore del *socket*.

A questo punto é possibile connettersi all'*host* sfruttando il *socket* appena creato, con la primitiva `connect`. Tale primitiva richiede come parametri:

- il descrittore del socket da utilizzare;
- un puntatore a `sockaddr`, contenente le informazioni sul dominio del *socket*, indirizzo IP di destinazione e porta;
- la dimensione del puntatore a `sockaddr`.

La funzione, in modo analogo a *socket*, ritorna -1 nel caso in cui la connessione non sia andata a buon fine.

Lettura e scrittura di informazioni sul socket

Creato il canale di comunicazione, per sfruttarlo all'interno dell'applicazione si deve scrivere o leggere dati su di esso, in modo da mettere in comunicazione le due macchine. Ció é possibile grazie alle funzioni `recv` e `send`, rispettivamente per leggere e scrivere sul *socket*. La sintassi di queste due funzioni é praticamente uguale.

```
ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t send(int s, const void *buf, size_t len, int
    flags);
```

Entrambe prendono 4 parametri:

- il *socket* da sfruttare (da cui leggere o su cui scrivere);
- un puntatore ai dati interessati (una variabile su cui salvare i dati letti o la variabile da scrivere su *socket*);
- la dimensione dei dati (da leggere o da scrivere);
- un eventuale flag (si lascia a 0 nella maggior parte dei casi).

Entrambe le funzioni ritornano il numero di byte letti o scritti (0 se non ci sono piú dati).

Lato *server*

La procedura per inizializzare una comunicazione di rete su un *server* in genere é:

- **addr_init** inizializzazione della variabile di tipo `sockaddr_in`;
- **socket** creazione del *socket*;
- **bind** creazione del legame tra il *socket* appena creato e la variabile `sockaddr_in` che identifica l'indirizzo del *server*;
- **listen** mette il *server* in ascolto per eventuali richieste da parte dei *client*;
- **accept** accettazione della connessione da parte di un *client*.

La sintassi di `bind` é la seguente:

```
int bind (int sockfd, const struct sockaddr *my_addr,
    socklen_t addrlen);
```

dove `sockfd` é l'identificatore del *socket*, `*my_addr` il puntatore alla variabile di tipo `sockaddr` che identifica l'indirizzo e `addrlen` la lunghezza di tale variabile. La funzione ritorna 0 in caso di successo, -1 in caso di errore.

La sintassi di `listen` invece é la seguente:

```
int listen (int sockfd, int backlog);
```

dove `sockfd` é il descrittore del *socket* e `backlog` il numero massimo di connessioni che il *server* puó accettare contemporaneamente. Anche questa funzione ritorna 0 in caso di successo e -1 in caso di errore.

La sintassi di `accept` infine é la seguente:

```
int accept (int sockfd, struct sockaddr *addr, socklen_t
    *addrlen);
```

dove `sockfd` é il descrittore del *socket*, `*addr` il puntatore alla variabile di tipo `sockaddr` e `*addrlen` il puntatore alla sua lunghezza. In caso di successo `accept` ritorna un valore maggiore di 0 che é il descrittore del *socket* accettato, mentre ritorna -1 in caso di errore. La funzione `accept` ritorna un nuovo identificatore di *socket*, che é il *socket* da utilizzare per le comunicazioni con il *client*. Inoltre, alla `accept` va passato il puntatore alla variabile `sockaddr` che identifica il *client*, non quello del *server*.

5.2.2 Programmazione concorrente

Come illustrato in 5.1.1, si ha proceduto nello sviluppo del *License Server* con struttura *multithreaded*, in moda da gestire correttamente e senza perdita le richieste provenienti da *License Client*. Per far ció, nella classe *LicenseManager* sono stati creati e avviati i *thread Listener, RequestManager* e *LicFileReader*; in questa stessa classe viene, inoltre, creata un'istanza della classe *ReqStatusManager*, secondo l'organizzazione illustrata in 5.1.2:

```
...
class RequestManager* myRequestManager;
class Listener* myListener;
class LicFileReader* myLicFileReader;
class ReqStatusManager* myReqStatusManager;

...

void LicenseManager::Init()
{
    ...
    //Instance ReqStatusManager
    myReqStatusManager=new ReqStatusManager;

    //Thread Listener
    myListener=new Listener;
```

```
17     myListener->start();
18
19     //Thread RequestManager
20     myRequestManager=new RequestManager;
21     myRequestManager->start();
22
23     //Thread LicFileReader
24     myLicFileReader=new LicFileReader;
25     myLicFileReader->start();
26 }
27 ...
```

Mutua esclusione

Nella descrizione dell'architettura generale del *License Server* (5.1.2) si é illustrata la struttura condivisa tra i *thread Listener* e *RequestManager*. L'accesso a tale struttura é controllato da due semafori, in modo che le operazioni di scrittura (da parte del *thread Listener*) e di lettura (da parte del *thread RequestManager*) siano eseguite correttamente in mutua esclusione. La rete di Petri di Figura 5.2 descrive la gestione della condivisione del *buffer* per la scrittura e il recupero dei pacchetti, governata dai semafori *sem_mutex* e *sem_pck_received*. Il primo regola l'accesso in mutua esclusione alla risorsa condivisa, il secondo segnala l'avvenuta scrittura/lettura sul/dal *buffer*.

Questa mimica tra i *thread Listener* e *RequestManager* é stata realizzata anche nel codice; per quanto riguarda il *thread Listener*, dopo che questo ha ricevuto un pacchetto dal *socket* sul quale si era posto in ascolto, lo inserisce nel *buffer* con le seguenti istruzioni:

```
1  ...
2  semaphore sem_mutex(1);
3  semaphore sem_pck_received(0);
4
5  ...
6
7  sem_mutex.wait();//attende sul sem_mutex
8      if (elem_counter < MAX_BUFFER_SIZE)
9      {
10         buffer_pointer=&buffer[write_pointer];
11         memcpy(buffer_pointer, &pck_tmp[0], sizeof(
               PckReceived));
```

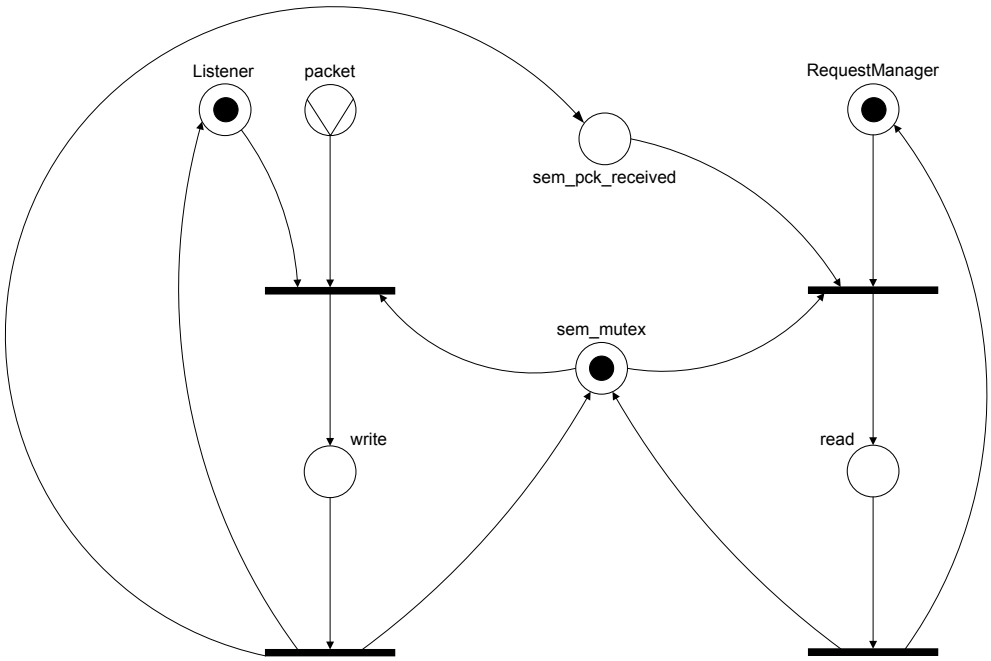

Figura 5.2: Rete di Petri che illustra la gestione del *buffer* per la scrittura e il recupero dei pacchetti, governata dai semafori *sem_mutex* e *sem_pck_received*

```
12          write_pointer=(write_pointer+(sizeof(PckReceived)
                /4))%MAX_BUFFER_SIZE;
13          elem_counter++;
14          sem_pck_received.signal(); //libera
                sem_pck_received
15      }
16  sem_mutex.signal();//libera sem_mutex
17  ...
```

dove `elem_counter` é un contatore del numero di pacchetti inseriti nel *buffer*, `buffer_pointer` é un puntatore al *buffer* condiviso, `write_pointer` é un puntatore alla prima posizione libera su cui scrivere nel *buffer*, `pck_tmp` é una struttura temporanea sulla quale viene inserito il pacchetto con la funzione `recv` e `PckReceived` é la struttura che contiene il pacchetto.

É importante osservare che il semaforo *sem_mutex* é inizializzato con un *token*, in accordo con la rete di Petri di Figura 5.2.

Per il *thread RequestManeger*, invece, il codice per la lettura dal *buffer* é:

```
1  ...
2  while(true)
3  {
4      sem_pck_received.wait();//attende sul semaforo
           sem_pck_received
5      sem_mutex.wait();//attende sul semaforo sem_mutex
6      ...
7      buffer_pointer=&buffer[read_pointer];
8      memcpy(&pck_rec, buffer_pointer, sizeof(PckReceived));
9      read_pointer=(read_pointer+(sizeof(PckReceived)/4))%
           MAX_BUFFER_SIZE;
10     elem_counter--;
11
12     sem_mutex.signal();//libera il semaforo sem_mutex
13 }
14 ...
```

dove `read_pointer` é un puntatore alla prima posizione da leggere nel *buffer* e `pck_rec` é una struttura temporanea sulla quale viene copiato il pacchetto preso dal *buffer*.

Capitolo 6

Test e collaudo

Introduzione

Dopo aver generato il codice sorgente, si deve testare il software per scoprire e correggere quanti piú errori possibili prima di fornire il prodotto al cliente. Lo scopo é quello di progettare una serie di *test case* che abbiano un'elevata probabilitá di individuare gli errori (prima che lo faccia il cliente). Questa fase si sviluppa, per le applicazioni convenzionali, attraverso tecniche *white-box* o *black-box*. Per le applicazioni *object-oriented* il *testing* inizia prima dell'esistenza del codice sorgente ma, dopo che é stato generato il codice, vien progettata una serie di test per verificare il funzionamento di una classe ed esaminare se esistono errori di collaborazione fra una classe e le altre.

Si deve progettare e documentare un insieme di *test case* progettati con lo scopo di controllare sia la logica interna, le interfacce, la collaborazione fra componenti, sia i requisiti esterni del programma, si devono definire i risultati attesi e registrare i risultati ottenuti.

6.1 Fondamenti del *testing*

L'obiettivo principale della progettazione dei test case consiste nel determinare una serie di test che abbia un'alta probabilitá di scoprire gli errori nel *software* [21]. A tale scopo si utilizzano tecniche di due tipi: test *white-box* e test *black-box*.

I test ***white-box*** sono rivolti alla struttura di controllo del programma. I *test case* devono garantire l'esecuzione di tutte le istruzioni del programma almeno una volta ed il passaggio per ogni condizione logica. Il *testing* per cammini base si serve dei *flow graph* per derivare un insieme

di cammini linearmente indipendenti. I test per condizioni e per *data flow* approfondiscono l'esame della logica del programma; il *testing* per cicli arricchisce le altre tecniche fornendo una procedura per l'esecuzione di cicli di vari livelli di profonditá.

I test ***black-box*** hanno lo scopo di scoprire errori nei requisiti funzionali, indipendentemente dalla struttura interna del programma. Le tecniche *black-box* si concentrano sul dominio informativo del *software*; i *test case* sono ricavati ripartendo il dominio di *input* e quello di *output* del programma in modo da assicurare la massima copertura. I metodi di *testing* basati sui grafi esplorano il *behaviour* degli oggetti in un programma e le relazioni tra questi. La suddivisione in classi di equivalenza ripartisce il dominio di *input* in classi destinate a trattare i dati ai limiti del campo di accettabilitá.

Un programma sviluppato viene valutato secondo le seguenti proprietá:

Testability La facilitá con cui un programma puó essere testato.

Operativitá Meglio funziona, meglio puó essere testato.

Osservabilitá Ogni dato di *input* genera un distinto dato di *output*. Gli stati e le variabili del sistema sono visibili od esaminabili durante l'esecuzione.

Controllabilitá Chi prepara i test puó controllare direttamente gli stati e le variabili del *software* e dell'*hardware*.

Scomponibilitá Il sistema *software* é composto da moduli indipendenti che possono essere testati separatamente.

Semplicitá Il programma deve esibire le seguenti doti: semplicitá funzionale (le funzionalitá si limitano a soddisfare i requisiti), semplicitá strutturale (l'architettura é modulare cosí da limitare la propagazione degli errori) e semplicitá del codice (adottare uno standard di codifica).

Stabilitá Meno modifiche si apportano, meno situazioni devono essere testate.

Comprensibilitá Piú informazioni si hanno, piú adeguati sono i test che si possono svolgere.

6.1.1 Caratteristiche del *testing*

Si possono, a questo punto, definire quali sono le caratteristiche del *testing*:

- Un test é buono se ha un'alta probabilitá di scoprire un errore.
- Un buon test non é ridondante.
- Un buon test deve essere il migliore nella sua categoria.
- Un buon test non deve essere né troppo semplice né troppo complesso.

6.2 *Testing white-box*

Il *testing white-box* é un metodo che sfrutta la struttura del controllo definita nel *design* a livello di componente per ricavare i test case. Adottando metodi di questo tipo, si possono ricavare test case con le seguenti caratteristiche:

1. garanzia che tutti i cammini indipendenti entro un modulo siano eseguiti almeno una volta;
2. esecuzione dei rami *true* e *false* per ogni decisione logica;
3. esecuzione di tutti i cicli nei casi limite ed entro i confini operativi;
4. uso di tutte le strutture dati interne per garantirne la validitá.

6.2.1 *Testing* per cammini di base

Il *testing per cammini di base* é una tecnica mediante la quale si puó calcolare una misura della complessitá logica di un *design* procedurale e ricavare da tale misurazione indicazioni per la scelta di un insieme di cammini di base di esecuzione. I *test case* derivati da tale insieme garantiscono l'esecuzione di ogni istruzione del programma almeno una volta.

Flow graph

Il *flow graph* rappresenta il flusso logico di controllo; la Figura 6.1 ne illustra la notazione. La costruzione risulta lievemente piú complicata quando il *design* procedurale contiene condizioni composte.

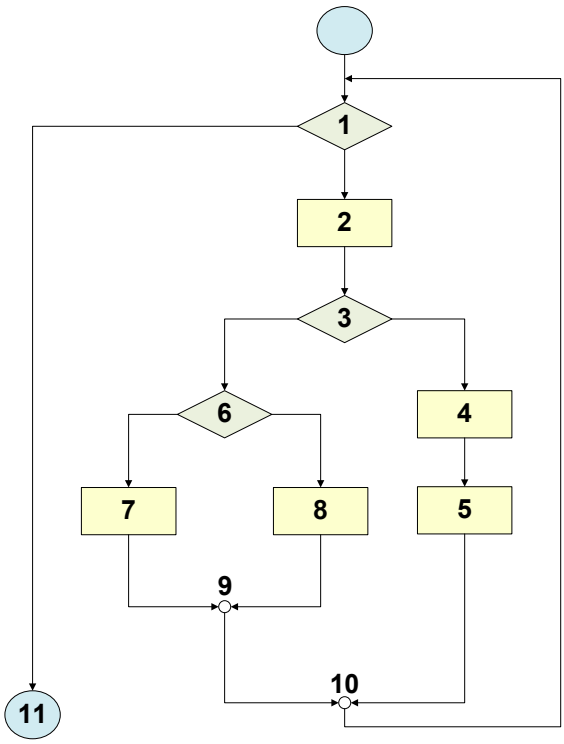

Figura 6.1: Flow graph

Cammini indipendenti del programma

Un cammino del programma si dice indipendente se introduce almeno un nuovo insieme di istruzioni od una nuova condizione. In un *flow graph*, un cammino é indipendente se attraversa almeno un arco non percorso prima di definire il cammino in questione.

Nella Figura 6.1 i cammini indipendenti sono:

1. 1-11
2. 1-2-3-4-5-10-1-11
3. 1-2-3-6-8-9-10-1-11
4. 1-2-3-6-7-9-10-1-11

Complessitá ciclomatica

La *complessitá ciclomatica* é una misurazione del *software* che fornisce una valutazione quantitativa della complessitá logica di un programma.

Fornisce il limite superiore per il numero di *test case* che devono essere condotti per garantire che tutte le istruzioni vengano eseguite almeno una volta.

Dato un grafo G, la complessitá ciclomatica *V(G)* é uguala al:

1. numero di aree chiuse + 1, *oppure*
2. numero archi - numero nodi + 2, *oppure*
3. numero condizioni semplici + 1

Nell'esempio in figura 6.1 $V(G) = 4$.

Derivazione del test case

Per determinare l'insieme di base possono essere applicati i seguenti passi:

1. sulla base del *design* o del codice, tracciare il *flow graph* corrispondente;
2. calcolare la complessitá ciclomatica del *flow graph*;
3. determinare una base di cammini linearmente indipendenti;
4. preparazione dei *test case* che portano all'esecuzione di tutti i cammini trovati.

6.3 *Testing black-box*

I *testing black-box*, chiamati anche testing del *behaviour*, si concentrano sui requisiti funzionali del software. Ció significa che ci si pone l'obiettivo di derivare un insieme di condizioni di *input* che esercitino completamente tutti i requisiti funzionali di un programma.

Un *testing black-box* mira a rilevare errori compresi nelle seguenti categorie:

1. funzioni errate o mancanti;
2. errori d'interfaccia;
3. errori nelle strutture dati o negli accessi a *database* esterni;
4. errori di *behaviour* o prestazionali;
5. errori di inizializzazione e terminazione.

Applicando le tecniche *black-box*, si ricava un insieme di *test case* che soddisfano i seguenti criteri:

- *test case* che riducono almeno di uno il numero di ulteriori *test case* necessari a raggiungere un livello di *testing* ragionevole;
- *test case* in grado di fornire indicazioni sulla presenza o assenza di classi di errori.

6.3.1 Metodo di *testing* basato sui grafi

Il grafo é una collezione di nodi che rappresentano gli oggetti, di archi che rappresentano le relazioni fra gli oggetti, di pesi associati ai nodi, che ne descrivono le proprietá, e di pesi associati agli archi, che ne descrivono alcune caratteristiche. I metodi di *testing* basati sui grafi sono:

Modellazione del *transaction flow* I nodi rappresentano passi di una transazione; gli archi rappresentano i nessi logici fra i passi.

Modellazione a stati finiti I nodi rappresentato gli stati del *software* osservabili dall'utente; gli archi rappresentano le transizioni di stato.

Modellazione del data flow I nodi sono oggetti-dato; gli archi sono trasformazioni che traducono un oggetto in un altro.

Modellazione degli aspetti temporali I nodi sono oggetti di programma; gli archi sono nessi sequenziali fra gli oggetti. I pesi degli archi specificano i tempi di esecuzione richiesti.

6.3.2 Suddivisione in classi di equivalenza

La *suddivisione in classi di equivalenza* é un metodo che suddivide il dominio dei dati di *input* di un programma in classi di dati, dalle quali derivare *test case*. Un *test case* ideale rilava, da solo, una classe di errori che potrebbero altrimenti richiedere l'esecuzione di molti *test case*, prima di poter dedurre l'errore generico.

L'ideazione dei *test case* in questo caso si basa su una valutazione delle classi di equivalenza per una condizione di *input*. Le classi di equivalenza possono essere definite secondo le seguenti linee guida.

1. Se una condizione di *input* specifica un intervallo, vengono definite una classe di equivalenza valida e due non valide.

2. Se una condizione di *input* richiede un valore specifico, vengono definite una classe di equivalenza valida e due non valide.

3. Se una condizione di *input* specifica un elemento di un insieme, vengono definite una classe di equivalenza valida e una non valida.

4. Se una condizione di *input* é un valore booleano, vengono definite una classe di equivalenza valida ed una non valida.

6.3.3 Analisi dei valori limite

Il maggior numero degli errori tende ad accumularsi ai limiti del dominio delle informazioni invece che in prossimitá del "centro". Per questo motivo é stata sviluppata la tecnica di *testing* detta *boundary value analysis* (BVA o analisi dei valori limite). Essa porta ad una selezione di *test case* che provino proprio i valori limite.

LA BVA é una tecnica di progettazione *test case* complementare alla suddivisione in classi di equivalenza. Le linee guida per la BVA sono simili a quelle descritte per la suddivisione in classi di equivalenza.

1. Se una condizione di *input* specifica un intervallo delimitato dai valori a e b, i test case devono prevedere i valori a e b e i valori poco sopra e sotto a e b.

2. Se una condizione di *input* specifica un insieme di valori, i *test case* devono esercitare i valori minimo e massimo, oltre ai valori appena sopra e sotto questi.

3. Le linee guida 1 e 2 si applicano anche alle condizioni di *output*.

4. Se le strutture dati interne di un programma hanno limiti predefiniti é necessario predisporre un *test case* che provi tale struttura di dati ai suoi valori limite.

6.4 *Testing* del sistema di *licensing*

La strategia adottata per testare il *software* prodotto combina caratteristiche dell'una e dell'altra metodologia. Si é stilata una lista di *test* che verificasse correttamente i requisiti funzionali individuati in 4.2 (tecnica *black-box*). Ogni voce di questo *plan* (si veda la Tabella 6.1) testa il corretto funzionamento di un modulo specifico del programma.

TITOLO	SCOPO
1. Connessione *Client-Server*	Verificare se il *License Client* é in grado di creare un *socket* e di connettersi al *License Server* che é in ascolto su una porta indicata attraverso i parametri di configurazione
2. Invio/ricezione pacchetto	Verificare se il *License Client*, dopo aver inviato un pacchetto, riceve risposta dal *License Server*
3. *Encrypted Communication*	Verificare se la comunicazione tra *License Client* e *License Server* é criptata
4. Porta del *License Server*	Verificare se é possibile selezionare, nei parametri di configurazione, la porta sulla quale il *License Server* deve porsi in ascolto: verificare che all'avvio il *License Server* utilizzi la porta impostata
5. Impostazioni di *log* del *License Server*	Verificare se é possibile impostare, nei parametri di configurazione, le impostazioni di *log* del *License Server*: verificare che all'avvio il *License Server* utilizzi tali impostazioni
6. Generazione *System ID*	Verificare la correttezza del *System ID* generato dal *License Server*
7. *Unscrumble* della *License Key*	Verificare la correttezza dei valori ottenuti dall'operazione di *unscrumble* della *License Key*
8. Stato di assegnazione del *License Server* per le applicazioni	Verificare se il *License Server* gestisce correttamente l'assegnazione dei *token* per le applicazioni
9. Stato di assegnazione del *License Server* per le opzioni	Verificare se il *License Server* gestisce correttamente l'assegnazione delle opzioni per le applicazioni
10. Lettura *License Files List*	Verificare se la lettura di *License File* successivi da parte del *License Server* viene eseguita correttamente

Tabella 6.1: *Test plan* del *Licensing System*

Ogni modulo, poi, é stato sottoposto ad un *test case* che verificasse

l'esecuzione di tutte le istruzioni e il passaggio attraverso tutte le condizioni logiche (tecnica *white-box*).

Capitolo 7

Considerazioni finali

Il mercato della virtualizzazione é uno tra quelli che ha avuto il maggior sviluppo nell'ultimo anno e sicuramente nel futuro piú immediato riserverá grosse novitá, anche a livello aziendale. L'interesse per le dinamiche dei nuovi sistemi di virtualizzazione, interesse nato tra i banchi universitari, si é concretizzato con lo studio e la conseguente attivitá di sviluppo aziendale.

Il lavoro di tirocinio si é rivelato proficuo sotto diversi aspetti:

- studio degli attuali sistemi di virtualizzazione;
- approfondimento della conoscenza del linguaggio di programmazione C++;
- applicazione pratica delle tecniche e delle metodologie di progettazione di un applicativo *software*;
- applicazione pratica delle tecniche di costruzione dei diagrammi UML per la documentazione della progettazione;
- approfondimento delle metodologie di programmazione concorrente, attraverso l'implementazione di un applicativo *multithreaded* e la gestione di risorse condivise;
- conoscenza delle dinamiche di un ambiente aziendale e collaborazione con il *team* di sviluppo.

Bibliografia

[1] "Virtualization overview." http://www.vmware.com/pdf/virtualization.pdf, 2006.

[2] "VMware vCenter Converter, Enterprise-class migration tool for converting physical machines to VMware virtual machines." http://www.vmware.com/files/pdf/09Q1_VM_CONVERTER_DS_EN.pdf, 2009.

[3] "VMWare vSphere 4, la migliore piattaforma per creare infrastrutture cloud." http://www.vmware.com/files/it/pdf/vsphere_datasheet_it.pdf, 2009.

[4] "VMWare ESX and ESXi, The Market Leading Production-Proven Hypervisors." http://www.vmware.com/files/pdf/esx_datasheet.pdf, 2009.

[5] "Building the Virtualized Enterprise with VMware Infrastructure." http://www.vmware.com/files/pdf/vmware_infrastructure_wp.pdf, 2008.

[6] S. Lowe, *Mastering VMWare vSphere 4*. Wiley Publishing Inc., 2009. Indianapolis, Indiana.

[7] "Best Practices Using VMWare Virtual SMP." http://www.vmware.com//pdf/vsmp_best_practices.pdf, 2005.

[8] "VMware vStorage VMFS, High-Performance Cluster File System for Storage Virtualization." http://www.vmware.com/files/pdf/VMware-vStorage-VMFS-DS-EN.pdf, 2009.

[9] "VMware vCenter Server 4, Unify and Simplify Virtualization Management." http://www.vmware.com/files/pdf/vcenter_server_datasheet.pdf, 2009.

[10] "VMware vMotion, Live Migration for Virtual Machines Without Service Interruption." http://www.vmware.com/pdf/vmotion_datasheet.pdf, 2009.

[11] "VMware Storage vMotion, High-Performance Cluster File System for Storage Virtualization." http://www.vmware.com/files/pdf/storage_vmotion_datasheet.pdf, 2009.

[12] "VMware Distributed Resource Scheduler (DRS), Dynamic Load Balancing and Resource Allocation for Virtual Machines." http://www.vmware.com/files/pdf/drs_datasheet.pdf, 2009.

[13] "VMware High Availability, Easily Deliver High Availability for All of Your Virtual Machines." http://www.vmware.com/files/pdf/ha_datasheet.pdf, 2009.

[14] "VMware Fault Tolerance, Deliver 24x7 Availability for Critical Applications." http://www.vmware.com/files/pdf/fault_tolerance_datasheet.pdf, 2009.

[15] "Industry's First and Only Licensing Solution Enabling Software Vendors to Authorize and Control Applications in Any Virtual Environment." http://www.safenet-inc.com/About_SafeNet/News_and_Media/News_and_Media_Items/2010/SafeNet_Enables_Software_Monetization_in_Virtual_Environments.aspx, 2010.

[16] "Software Licensing in Virtual Environments." http://service-architecture.blogspot.com/2008/04/software-licensing-in-virtual-world.html, 2010.

[17] "Application Virtualization and Software Licensing: Best Practices for Software Vendors." http://www.softwaremag.com/trk.cfm?uid=101, 2010.

[18] "Find hardware details of VMWare host from command line." http://vikashkumarroy.blogspot.com/2008/06/i-being-non-unix-guy-was-struggling-to_26.html, 2008.

[19] "NAT in VMWare vSphere/ESX Ð In a nut shell." http://www.javatuning.com/nat-in-vmware-vsphereesx-in-a-nut-shell/, 2008.

[20] "Knowledge Base: Changing the MAC address of a virtual machine." http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=507, 2010.

[21] R. S. Pressman, *Principi di Ingegneria del software*. The McGraw-Hill Companies, S.r.l., 2008.

[22] "VMWare ESX and ESXi, The Market Leading Production-Proven Hypervisors." `http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf`, 2009.

Elenco delle figure

2.1 architettura-vmware 5
2.2 Hypervisor-VMM 8
2.3 vSMP 9
2.4 VMFS 10
2.5 vcenter 12
2.6 cluster server ESX 13
2.7 vmotion 14
2.8 drs 15
2.9 high availability 16
2.10 fault tolerance 17

4.1 sistema di *licensing* 27
4.2 estensione scadenza 29
4.3 sicurezza socket 31
4.4 generazione *System ID* 31
4.5 socket 33
4.6 diagramma componenti sistema di *licensing* 38
4.7 Diagramma di *deployment* dell'intero sistema di *licensing* . 40
4.8 Diagramma di sequenza dell'intero sistema di *licensing* . . 42
4.9 Diagramma di attivitá del License Client 44
4.10 Diagramma di attivitá del License Server 45
4.11 Diagramma di attivitá del modulo di gestione dei License Files 46

5.1 struttura *License Server* 51
5.2 rete di petri 59

6.1 Flow graph 64

Indice analitico

analisi dei valori limite, 67

complessitá ciclomatica, 64

flow graph, 63

hypervisor, 4

licensing
- approcci, 21
- problematiche, 19
- soluzione, 23

macchina virtuale, 4
- vantaggi, 6

progettazione
- requisiti funzionali, 32
- requisiti strutturati, 25
- specifiche di progetto, 37

realizzazione
- *socket* TCP, 52
- architettura, 48
- programmazione concorrente, 57

suddivisione in classi di equivalenza, 66

testing, 61
- black-box, 65
- sistema di *licensing*, 67
- white-box, 63

Virtualizzazione, 4
VMM, Virtual Machine Monitor, 4
VMWare vSphere, 6
DRS, Distributed Resource Scheduler, 15
ESX e ESXi, 7
Fault Tolerance, 16
High Availability, 15
storage vMotion, 13
vCenter, 11
VMFS, Virtual Machine File System, 9
vMotion, 13
vSMP, Virtual Simmetric Multi-Processing, 8