Alma Mater Studiorum
Università di Bologna
Campus di Cesena

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

UN SISTEMA WEB PER L'ACCESSO REMOTO AI PC DEI LABORATORI DI CAMPUS

Elaborato in:
System Integration

Relatore:
Prof. Vittorio Ghini

Presentata da:
Matteo Ragazzini

Correlatore:
Dott. Ciro Barbone

Sessione II
Anno Accademico 2019-2020

"I computer sono incredibilmente veloci, accurati e stupidi.
Gli uomini sono incredibilmente lenti, inaccurati e intelligenti.
L'insieme dei due costituisce una forza incalcolabile."
Albert Einstein

Introduzione

Ripercorrendo la storia fino ai giorni nostri è possibile notare come, molto spesso, è dai momenti di difficoltà che derivano le maggiori innovazioni tecnologiche. Il periodo di lockdown, a causa del COVID19, ad esempio, ha portato alla luce la necessità per gli studenti della nostra università di accedere da remoto ai computer dei laboratori all'interno del campus. In queste macchine infatti, sono presenti applicativi spesso fondamentali per la corretta fruizione dei corsi. Un esempio concreto è il corso di System Integration, sul quale si basa la mia tesi, che quest'anno ha sopperito a questa necessità tramite lo strumento vncviewer. Quest'ultimo però richiedeva: una fase preliminare di configurazione da parte del professore, l'installazione per gli studenti di una applicazione sui propri pc personali e la conoscenza, seppur minima, del concetto di *socket*.

L'obiettivo della mia tesi sarà quello di risolvere il problema sopra citato realizzando ed installando una versione *custom* del servizio **Apache Guacamole**. Quest'ultimo permetterà agli studenti di accedere da remoto e da qualsiasi dispositivo, tramite browser web, alle macchine del campus con le proprie credenziali di ateneo. Una caratteristica importante della realizzazione di questo servizio sarà l'utilizzo della piattaforma di virtualizzazione a livello di sistema operativo *Docker*, che permetterà di rendere l'applicazione facilmente estendibile, scalabile e manutenibile.

La trattazione sarà sviluppata in 4 capitoli. Nel primo si affronterà in dettaglio il contesto di sviluppo del sistema, approfondendo i concetti teorici alla base del progetto. Nel secondo capitolo si delineeranno i requisiti per la

progettazione del sistema e le tecnologie utilizzate per la costruzione dello stesso. Nel terzo capitolo si esporranno i dettagli dell'implementazione del servizio descrivendo in particolar modo le modifiche effettuate ai prodotti già esistenti e le motivazioni che le hanno rese necessarie. Infine, nel quarto ed ultimo capitolo si discuteranno le modalità con cui l'applicazione è stata testata e verranno analizzati i risultati ottenuti.

Indice

Introduzione **i**

1 Contesto **1**

1.1 Protocolli desktop remoti 1

1.1.1 Virtual Network Computing 2

1.1.2 Remote Desktop Protocol 3

1.1.3 Evoluzione post HTML5 4

1.2 Virtualizzazione 6

1.2.1 Virtualizzazione a livello di sistema operativo 6

1.2.2 Evoluzione dei container in ambiente Linux 7

1.2.3 Docker 10

2 Analisi e Progettazione **13**

2.1 Analisi dell'architettura del campus 14

2.2 Analisi delle tecnologie utilizzate 15

2.2.1 Guacamole 15

2.2.2 Nginx 17

2.2.3 PostgreSQL 18

2.2.4 PgAdmin 18

2.3 Strumenti relativi a Docker 18

2.3.1 Docker Compose 19

2.3.2 Docker Swarm 20

2.4 Progetto Finale 21

3 Realizzazione **23**
3.1 Installazione di Guacamole su macchina fisica 23
3.2 Creazione delle immagini 25
3.2.1 Immagine di Guacamole 25
3.2.2 Immagine Guacd 28
3.2.3 Immagini di Nginx, PostgreSQL e pgAdmin 30
3.3 Integrazione dei container tramite Docker Compose 31
3.4 Deployment tramite Swarm 35

4 Testing **37**
4.1 Fasi di creazione dell'architettura del test 37
4.1.1 Singola istanza di test 37
4.1.2 Parallelizzazione del test 39
4.1.3 Test dockerizzato, parallelo e remoto 40
4.2 Esecuzione del test 44
4.3 Analisi dei risultati del test 46
4.3.1 Guacamole installato su macchina singola 46
4.3.2 Guacamole dispiegato su un cluster 48
4.4 Conclusioni del Test 51

Scenari di utilizzo di Guacamole nella didattica **53**

Conclusioni **57**

Ringraziamenti **59**

Bibliografia e sitografia **61**

Capitolo 1

Contesto

In questo primo capitolo, si andrà ad affrontare lo stato dell'arte all'atto della stesura della tesi. Avendo realizzato un gateway html5 containerizzato per l'accesso remoto, si è deciso di focalizzare l'attenzione sulle seguenti 2 macro aree:

1. protocolli desktop remoti ed evoluzione di questi ultimi con l'avvento di HTML5;

2. virtualizzazione a livello di sistema operativo e analisi della piattaforma Docker.

1.1 Protocolli desktop remoti

I *remote desktop protocols* nascono nei primi anni 2000 per supportare un'esigenza specifica: quella di **usare un dispositivo per controllarne un altro**. In informatica il desktop remoto [1] è un'implementazione del servizio di accesso remoto tramite interfaccia grafica anziché *shell* testuale. Indica quindi la capacità di aprire una **sessione** utente di tipo grafico o comunque **interattiva**, su un computer remoto, attraverso una connessione remota, utilizzando le risorse di input/output del computer locale. Una sessione remota è assolutamente indistinguibile da una sessione locale: l'utente ha il

pieno controllo dell'interfaccia grafica e dell'ambiente operativo, e gli strumenti di I/O locali funzionano come se fossero direttamente collegati al computer remoto.

Le applicazioni dei protocolli desktop remoti sono molteplici dall'ambito lavorativo a quello didattico e presentano numerosissimi vantaggi. Primo fra tutti un'estrema flessibilità in quanto studenti e/o dipendenti possono utilizzare programmi, file e risorse di computer remoti da qualsiasi postazione locale.

I protocolli remote desktop attualmente più utilizzati sono RDP *Remote Desktop Protocol* che è il protocollo proprietario di Microsoft e VNC *Virtual Networking Computing* che è la controparte *open source* e *cross-platform*

1.1.1 Virtual Network Computing

Il protocollo **VNC** (*Virtual Network Computing*) [2] è un protocollo di accesso/controllo remoto creato inizialmente nel laboratorio della Olivetti & Oracle Research Lab, acquistato successivamente nel 1999 da AT&T. Il protocollo ha la funzione di trasportare l'interfaccia grafica da un computer definito server ad un computer nel quale è stato installato un componente chiamato *viewer*. Dal prodotto originario ormai abbandonato di AT&T si sono sviluppati diversi programmi VNC, alcuni *open source*, altri commerciali, che presentano solitamente le seguenti caratteristiche:

1. sono prodotti *cross-platform*, per cui è possibile utilizzarli su sistemi operativi diversi;

2. effettuano un semplice *streaming* dello schermo remoto, senza essere a conoscenza di ciò che stanno trasmettendo. Questo comporta alcune lacune a livello di sicurezza; ad esempio collegandosi ad un computer con il quale ha effettuato l'accesso un amministratore, automaticamente acquisiremo i suoi privilegi;

3. collegano l'utente remoto al computer stesso, ciò si traduce nel fatto che se due utenti richiedono la stessa macchina, condivideranno fra loro anche le periferiche (mouse e tastiera).

Fra i software più utilizzati troviamo RealVNC, UltraVNC e PuTTY. Tutti questi protocolli sfruttano TCP come protocollo a livello di trasporto sulla porta di default 5900, ma è anche possibile utilizzare HTTP sfruttando client scritti in Java.

1.1.2 Remote Desktop Protocol

Il protocollo **RDP** (*Remote Desktop Protol*) è un protocollo di rete proprietario, sviluppato da Microsoft [3], che permette la connessione remota utilizzando di default la porta TCP e UDP 3389. Questo è uno dei componenti che abilitano i *Remote Desktop Services* di Microsoft Windows, la tecnologia *server-based* di Microsoft, che ha preso il posto dei *Terminal Services* di Windows Server 2008 e versioni precedenti. Componenti essenziali dell'architettura Remote Desktop Services (RDS) sono, lato server, il *Terminal Server* e il *Remote Desktop Gateway*, mentre lato client troviamo il Remote Desktop, o più correttamente *Remote Desktop Connection* (RDC).

I client RDP sono attualmente disponibili per Windows, MacOS, Linux, Unix, mentre il server è integrato di base in tutti i sistemi Windows.

RDP è un protocollo di tipo multicanale, basato sulla famiglia ITU-T.120, e compatibile con diverse topologie di rete, tra le quali TCP/IP, ISDN e diversi protocolli LAN. In quanto protocollo multicanale, dispone di svariati canali virtuali distinti per il trasporto dei dati di presentazione, delle comunicazioni con le periferiche seriali, delle informazioni di licenza e dei dati a elevato livello crittografica come ad esempio input da tastiera ed attività del mouse. Essendo un'estensione del protocollo T.Share mantiene molte altre capacità, ad esempio le funzionalità architetturali necessarie per supportare le comunicazioni multipunto, che consentono di trasmettere in tempo reale i dati di un'applicazione a più utenti senza dover inviare i medesimi dati a ciascuna sessione singolarmente.

RDP è inoltre **semantico**, ossia è a conoscenza di controlli, caratteri ed altre primitive grafiche simili. Ciò significa che quando si esegue il *rendering* di uno schermo su una rete, queste informazioni vengono utilizzate per comprimere il flusso di dati in modo significativo rendendo il protocollo più veloce e reattivo. Infine può supportare più utenti remoti connessi alla stessa macchina che si ignorano completamente l'un l'altro.

Fra le caratteristiche rilevanti figurano anche l'*encryption* dei dati a 128 bit tramite l'algoritmo RC4, l'implementazione degli algoritmi di compressione video H.264/AVC per l'ottimizzazione della banda, la funzionalità di audio *redirection*, che permette di ascoltare l'audio del desktop remoto sulla postazione locale, di printer e port redirection, nonché il supporto multi monitor. Tutta l'attività del protocollo RDP è gestita dal driver di periferica di *Terminal Server* e dai suoi componenti, tra cui l'*RDP driver* che gestisce i trasferimenti, la crittografia e l'interfaccia utente.

Network Level Authentication

il *Network Level Authentication* **NLA** [4] è una feature di RDP, introdotta a partire da Windows Vista, che richiede l'autenticazione degli utenti prima di creare la sessione desktop remota con il server. Questa novità fu in realtà aggiunta per migliorare la sicurezza del sistema. Infatti, avviare una sessione e mostrare la schermata di login (utilizzando quindi risorse computazionali) ad un utente non ancora autenticato esponeva il servizio ad attacchi di tipo DOS (*Denial Of Service*) o *remote code execution*

1.1.3 Evoluzione post HTML5

Nel 2016 l'avvento della versione 5 di HTML [1] oltre ad aumentare la capacità di elaborazione del linguaggio, ha migliorato la gestione dello streaming audio e video. Questi cambiamenti hanno permesso la realizzazione di applicazioni web estremamente più complesse ed ambizione, fra le quali si sono fatti

[1] Il linguaggio di markup che attualmente rappresenta lo standard de facto per la strutturazione di pagine web

largo anche molteplici gateway per l'accesso a desktop remoti. Questi gateway vengono detti *clientless*, in quanto rimuovono il vincolo dell'installazione di una applicazione lato client, trasferendo la computazione sul web.

In quell'anno Brien Posey CIO [2] di una catena americana di ospedali e network engineer presso il dipartimento di difesa degli Stati Uniti, scrive un articolo [5] nel quale elenca le motivazioni per cui l'accesso a base web a desktop remoti sarebbe diventato il nuovo standard, abbandonando l'utilizzo di applicazioni lato client. Questa affermazione era basata su un radicale cambiamento dell'IT avvenuto negli ultimi anni e ancora più visibile oggigiorno.

All'interno delle organizzazioni, che siano esse aziende o università il carico di lavoro infatti non è più semplicemente distribuito su server Windows e acceduto tramite desktop o laptop Windows, ma è sempre più comune utilizzare un mix di server Microsoft e Linux e distribuire applicazioni *Saas* [3] alle quali gli utenti accedono da una grande varietà di dispositivi: Mac, tablet Android e iOS, pc Linux, Chromebook, smartphone . . . i quali hanno una sola cosa in comune: **un browser HTML5**

Effettuare degli accessi a base web comporta inoltre, un risparmio per le aziende software le quali, non devono più né progettare applicazioni client diverse per ogni dispositivo né mantenerle aggiornate. Per fruire in maniera corretta dei servizi occorre solamente che l'utente finale mantenga aggiornato il suo browser. La fig. 1.1 nella pagina seguente mostra un esempio concreto di accesso a desktop remoti tramite gateway HTML5.

Secondo la visione di Posey, il browser sarebbe quindi diventato l'unico punto di accesso per tutti i tipi di risorse. A distanza di 4 anni dalla pubblicazione dell'articolo, analizzando le nostre abitudini e l'utilizzo che facciamo dei browser, non possiamo far altro che avvalorare la sua tesi.

[2] Chief Information Officer: dirigente dell'azienda responsabile delle tecnologie di informazione e comunicazione

[3] Software as a service: un modello di distribuzione del software applicativo dove un produttore di software sviluppa, opera e gestisce un'applicazione web mettendola a disposizione dei propri clienti via Internet

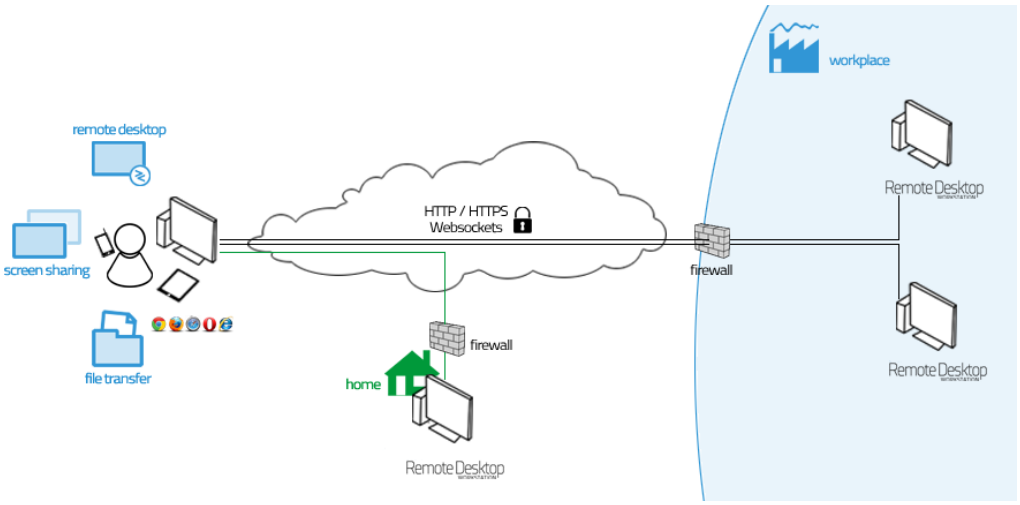

Figura 1.1: Esempio di accesso remoto tramite web browser

1.2 Virtualizzazione

Con il termine virtualizzazione ci si riferisce alla possibilità di astrarre le componenti hardware, cioè fisiche, degli elaboratori al fine di renderle disponibili al software in forma di risorsa virtuale. Tramite questo processo è quindi possibile installare sistemi operativi su un hardware virtuale che nell'insieme prende il nome di macchina virtuale. Quando oggi ci riferiamo a questa tecnica di virtualizzazione lo facciamo con l'appellativo di **virtualizzazione a livello hardware**. Negli ultimi tempi infatti, la virtualizzazione hardware ha dovuto fronteggiare la concorrenza di un'altra forma di virtualizzazione: quella a livello di sistema operativo anche detta *OS-level virtualization* o *containerization*.

1.2.1 Virtualizzazione a livello di sistema operativo

L'OS-level virtualization e l'hardware virtualization come si evince dalla fig. 1.2 nella pagina successiva, differiscono tra loro per la presenza o meno di un di un sistema operativo guest detto *hypervisor*. Quest'ultimo presente solo nella virtualizzazione hardware, ha il compito di gestire l'accesso alle risorse hardware e garantire l'isolamento tra le macchine. Nella containerizzazione

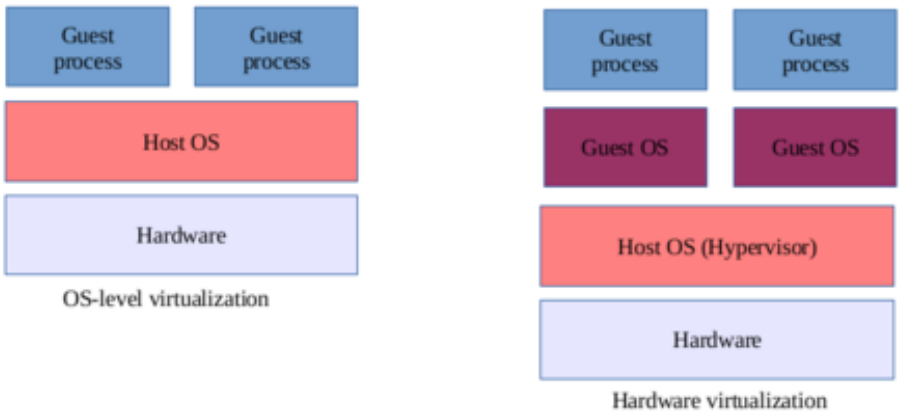

Figura 1.2: OS-virtualization vs hardware virtualization

invece, le applicazioni vengono eseguite all'interno di *container* ossia partizioni del sistema operativo che rappresentano a tutti gli effetti un ambiente isolato. Al loro interno, oltre ai binari da eseguire, sono presenti librerie, dipendenze di vario genere e file di configurazione.

Come si nota in fig. 1.2 più *container* in esecuzione sulla stessa macchina condividono tra loro il *kernel* del sistema operativo che li ospita. Oltre ai benefici caratteristici della virtualizzazione, quali limitazione delle risorse hardware, dei costi operativi e della manutenzione, la condivisione dei servizi di base offerti dal sistema operativo implica un minore *overhead* di risorse della macchina e una maggiore velocità di istanza ed esecuzione. Inoltre una volta definito, un *container* è **replicabile** e **riutilizzabile**, rendendo di conseguenza possibile sfruttare questa tecnologia per effettuare un dispiegamento più rapido delle applicazioni.

1.2.2 Evoluzione dei container in ambiente Linux

La storia dei *container* in ambiente Linux è legata allo sviluppo di funzionalità specifiche del *kernel*. Alla base della creazione di questa tecnologia c'è stata la volontà di separare lo spazio utente in unità distinte, in modo tale da coesistere all'interno dello stesso ambiente. Il primo esempio nel mondo Linux

a introdurre questa funzionalità è stato il progetto Linux-VServer, il quale ha permesso la creazione e la gestione di *virtual private server* attraverso un meccanismo di partizionamento del sistema operativo. All'interno di ogni partizione viene eseguito un server virtuale al quale vengono assegnate risorse come ad esempio la memoria, lo spazio su disco e slot per l'utilizzo della CPU in maniera dinamica. Linux-VServer operava attraverso l'applicazione di *patch* al *kernel* [6].

Le funzionalità del *kernel* Linux fondamentali per la creazione di ambienti di esecuzione isolati sono le seguenti: **cgroups** (abbreviazione per *control groups*) e ***namespace*** [7]. Per quanto concerne la prima funzionalità citata, questa permette di organizzare i processi in gruppi gerarchici e definisce delle priorità e dei limiti per l'accesso alle risorse da parte di questi gruppi. Ad esempio è possibile limitare la quantità di memoria utilizzabile da un gruppo di processi. Inoltre, cgroups fornisce delle funzionalità per il monitoraggio dell'utilizzo delle risorse. Il *kernel* espone questa funzionalità attraverso uno pseudo-filesystem chiamato **cgroupfs**. La funzionalità *namespaces* invece permette di limitare cosa un processo, o un gruppo di processi, può vedere del resto del sistema. È possibile definire il contesto di esecuzione di un processo a vari livelli, ad ognuno dei quali corrisponde appunto a uno spazio dei nomi:

mtn isolamento dei punti di mount;

pid isolamento degli ID dei processi;

net isolamento dello stack di rete, dei dispositivi e delle porte;

ipc isolamento dei meccanismi di comunicazione inter-processo ovvero oggetti System V IPC e code di messaggi POSIX;

UTS isolamento degli hostname;

user isolamento degli ID per gruppi e utenti;

time isolamento degli orologi di sistema.

I *namespace* rendono l'isolamento "componibile", in quanto è possibile decidere a quale livello isolare l'esecuzione di un processo. Per operare con i *namespace* il *kernel* espone delle *system call* specifiche, di cui le principali sono:

clone() che permette di avviare un processo creando i *namespace* desiderati se impostati attraverso le rispettive flag;

unshare() che sposta il processo chiamante in nuovi *namespace*;

setns() che sposta il processo chiamante in *namespace* già esistenti.

Nel 2008 sulla base delle funzionalità descritte finora, nasce **LXC** [8], un ambiente di virtualizzazione a livello di sistema operativo per Linux. In seguito entra in scena **Docker** [9], una tecnologia che sfruttando LXC ha aggiunto strumenti e concettualizzato entità che hanno permesso la rapida diffusione dei *container* e semplificato il loro utilizzo.

Nel corso degli anni Docker si è evoluto, abbandonando LXC e ristrutturandosi a livello architetturale. Sono nati contemporaneamente diversi meccanismi per la virtualizzazione a livello di sistema operativo, i quali consistono in software che operano a vari livelli e con strutture diverse. Nasce così l'esigenza di standardizzare le procedure e gli strumenti in ambito *container* per garantire un certo livello di interoperabilità. Nel 2015 viene fondata la *Open container Initiative* [10], un progetto della Linux Foundation che ha come scopo la definizione di standard aperti per la virtualizzazione a livello di sistema operativo, nello specifico in ambienti Linux. Questo progetto attualmente è sostenuto dalle più importanti aziende e fondazioni che si occupano di *cloud-computing* e *container*.

Si prosegue con l'analisi della piattaforma Docker, la quale nel corso degli anni ha assunto il ruolo di standard de facto per quanto riguarda la virtualizzazione a livello di sistema operativo.

1.2.3 Docker

Come descritto nel paragrafo precedente Docker è la prima tecnologia che è riuscita a diffondere e rendere relativamente semplice l'utilizzo dei *container*. Questa è una applicazione client-server che si compone principalmente delle seguenti parti:

Docker deamon (**dockerd**) che espone un'API REST sulla quale rimane costantemente in ascolto per richieste su *socket* unix, tcp e fd. Il demone gestisce tutti i *Docker objetcs* ossia *immages*, *containers*, *networks* e *volumes*;

CLI *Command Line Interface* ossia un'interfaccia a riga di comando che a sua volta sfrutta l'API REST per dialogare con il Docker Deamon.

Si procede ora con la descrizione degli oggetti fondamentali di Docker.

Docker Objects

I principali *Docker objects* gestiti dal Docker deamon sono i seguenti:

IMAGE : template di sola lettura con istruzioni per la creazione di un container. Spesso una immagine si basa su un'altra immagine con qualche modifica aggiuntiva. Per creare una propria immagine si utilizza un *Dockerfile* che, tramite una sintassi ben definita, permette di definire degli step di creazione dell'immagine. Ogni istruzione nel *Dockerfile* va a creare un layer aggiuntivo a quello dell'immagine di base. Quando andiamo ad effettuare il *build* di un'immagine, vengono ricreati solo i layer che sono cambiati rispetto alla versione precedente. Questa è parte del meccanismo che permette alle immagini di essere così leggere, piccole e veloci se comparate alle altre tecniche di virtualizzazione.

CONTAINER : istanza in esecuzione di una immagine. E' possibile creare, eseguire, stoppare, muovere o eliminare un container sfruttando il CLI. Generalmente i container risultano ben isolati gli uni dagli altri, ma

nella realizzazione di applicazioni con più container, è possibile stabilire quanto questi siano effettivamente isolati, condividendo ad esempio dati fra loro. Le modifiche effettuate su un container non sono persistenti, perciò alla rimozione del container, tutte le modifiche effettuate su quest'ultimo vengono perse, a meno che non venga creata una nuova immagine basata sullo stato attuale del container.

SERVICE : concezione astratta di un container insieme alle sue configurazioni (volumes e network). I servizi permettono di scalare i container su più *Docker deamons* facendoli lavorare insieme come uno sciame (*swarm*). È possibile definire lo stato desiderato di un servizio, specificando ad esempio il numero di repliche che devono essere attive in un determinato momento. All'utente finale il servizio risulta come se fosse una singola applicazione.

È inoltre presente un *docker registry* che immagazzina le immagini. Generalmente viene utilizzato il registry online ufficiale *docker hub* ma si può configurare Docker per utilizzare un registry diverso. Viene riportata in figura fig. 1.3 nella pagina successiva la rappesentazione dell'architettura di docker.

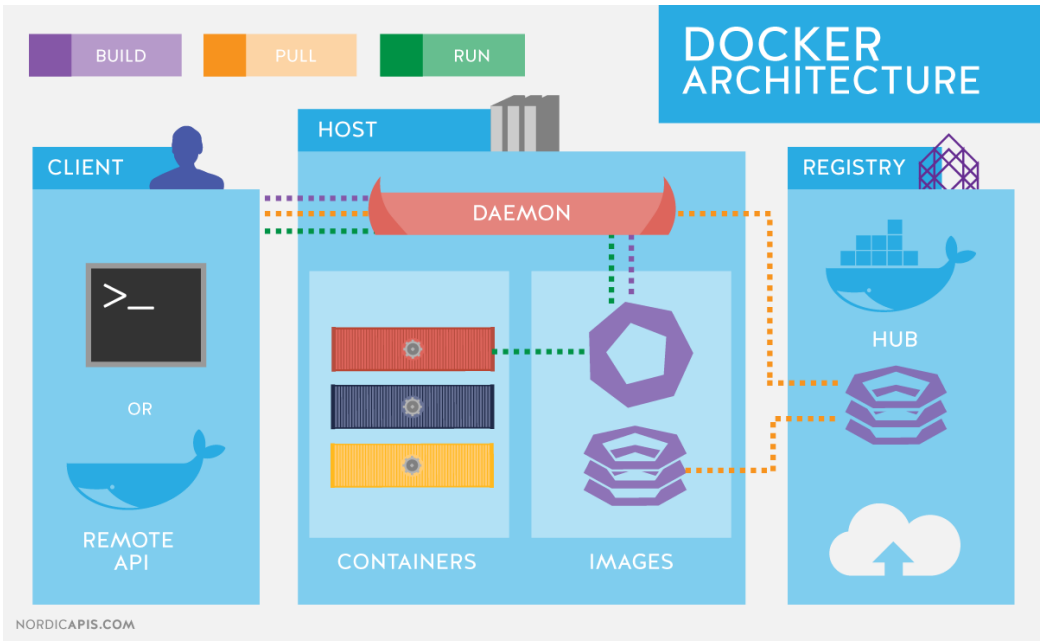

Figura 1.3: Architettura di Docker

Capitolo 2

Analisi e Progettazione

Prima di procedere con la progettazione del sistema, è necessario effettuare una chiara analisi dei requisiti dello stesso.
Dal punto di vista dei **requisiti funzionali** il sistema dovrà:

1. integrare l'autenticazione tramite server Radius;
2. garantire il funzionamento con i più famosi protocolli desktop remoti quali RDP e VNC;
3. assicurare la sicurezza dei dati degli studenti sfruttando il protocollo HTTPS.

Per quanto riguarda invece i **requisiti non funzionali** si richiede che il sistema sia:

1. composto da sole tecnologie open source;
2. trasparente agli amministratori di sistema e facilmente manutenibile;
3. di immediato utilizzo per gli studenti;
4. reattivo ed utilizzabile da qualunque dispositivo.

Per quanto riguarda il primo requisito funzionale, ossia l'aggiunta dell'autenticazione tramite server Radius, è necessario effettuare una analisi

dell'architettura del campus per comprendere come avviene l'accesso alle risorse interne all'organizzazione da postazioni esterne alla stessa.

2.1 Analisi dell'architettura del campus

L'università di Bologna utilizza *Active Directory* per la gestione delle risorse all'interno dell'organizzazione. Quest'ultimo è il servizio di directory proprietario di Microsoft ossia un insieme di programmi che provvedono ad organizzare, gestire e memorizzare informazioni e risorse centralizzate e condivise all'interno di reti di computer , rese disponibili agli utenti tramite la rete stessa, fornendo anche un controllo degli accessi sull'utilizzo delle stesse.

RADIUS [11] (*Remote Authentication Dial-In User Service*) è invece il protocollo *AAA (authentication, authorization, accounting)* utilizzato in applicazioni di accesso alle reti o di mobilità IP. Sviluppato da Livingston Enterprises Inc., nel 1991, come server di accesso di autenticazione e come protocollo di accounting, riportato successivamente nelle norme Internet Engineering Task Force (IETF), il protocollo, dotato di un ampio supporto, è spesso utilizzato dagli *ISP* e dalle aziende per gestire l'accesso a Internet o reti interne, reti wireless e servizi integrati di posta elettronica.

RADIUS viene eseguito nel livello di applicazione e può utilizzare il protocollo TCP o UDP. I server di accesso alla rete e i *gateway* che controllano l'accesso a una rete, di solito contengono un componente *client RADIUS* che comunica con il *RADIUS server*, che solitamente è un processo in background in esecuzione su un server UNIX o Microsoft Windows. Attualmente è lo standard de facto per l'autenticazione remota, prevalendo sia nei sistemi nuovi sia in quelli già esistenti ed è implementato in appositi server di autenticazione.

Nel nostro caso Guacamole non effettuerà direttamente l'autenticazione degli utenti, ma contatterà il server Radius, e lascerà a quest'ultimo il compito di verifica delle credenziali di accesso.

2.2 Analisi delle tecnologie utilizzate

Si procede ora con l'esposizione dei vari componenti e strumenti che faranno parte del progetto finale. La trattazione esaminerà il comportamento e la funzione di ognuno di essi, partendo ovviamente da Guacamole, che ricordiamo essere il cuore del sistema, passando poi ad nginx ed infine a postrgreSQL. Verranno analizzate anche le tecnologie che faranno da supporto per la realizzazione dell'applicazione come pgAdmin per la gestione del database e i due strumenti relativi a Docker: Docker Compose e Docker Swarm.

2.2.1 Guacamole

Guacamole [12] è una web application sviluppata da Apache che permette l'accesso ad ambienti desktop sfruttando protocolli desktop remoti come VNC o RDP. Questa web application è parte di uno *stack* che fornisce un *remote desktop gateway* che si pone al di sopra dei protocolli utilizzati. È scritto in JavaScript e utilizza solo HTML5 ed altri standard, perciò la parte client di Guacamole non richiede altro che un web browser per accedere a qualsiasi desktop disponibile. Storicamente Guacamole era un client VNC HTML5 e ancora prima un client Telnet chiamato RealMint (dove RealMint era l'anagramma per terminal). Da allora le cose sono cambiate: l'architettura di Guacamole si è evoluta per comprendere più protocolli desktop remoti ed è abbastanza performante per farne un uso giornaliero.

Il motivo principale per cui Guacamole è stato scelto è che permette di effettuare l'accesso ad uno o più desktop in maniera remota **da qualunque dispositivo** inclusi smartphone e tablet, **senza** la necessità di **installare un client**. Inoltre, utilizzando solo HTTP o HTTPS Guacamole permette l'accesso alle macchine da qualunque locazione geografica senza violare le policy del luogo di lavoro o dell'organizzazione.

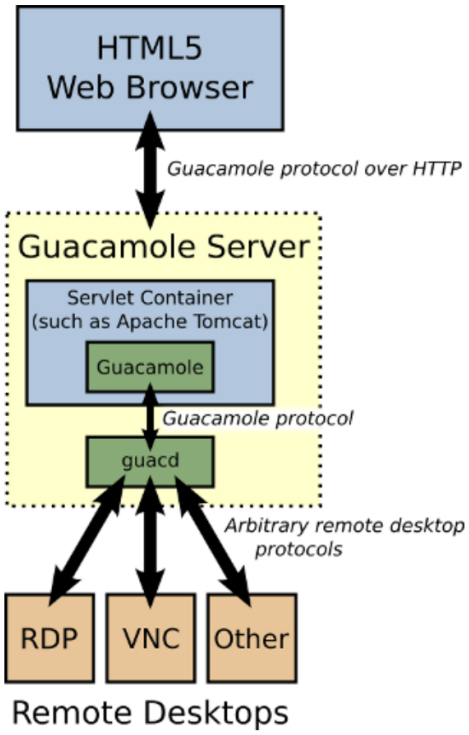

Figura 2.1: Architettura di Guacamole

Architettura di Guacamole

Guacamole non è solo una web-app ma è un servizio composto da più parti, delle quali il front-end è pensato per essere leggero, intuitivo e minimale, mentre la parte complessa si nasconde nel back-end.

Come si evince dalla fig. 2.1 gli utenti si collegano al guacamole server tramite i loro web browser. Il guacamole client, scritto in JavaScript, viene fornito agli utenti tramite un web server, nel nostro caso Tomcat. Una volta caricato, il client si connette al server via HTTP usando il protocollo proprietario Guacamole. La web application dispiegata sul server Guacamole legge il protocollo e lo inoltra a **guacd**, che è il *proxy* nativo per Guacamole. Il *proxy* interpreta il contenuto del protocollo Guacamole e si connette ad un qualsiasi numero di desktop remoti per conto dell'utente. Il protocollo Guacamole abbinato a guacd permette di ottenere un *protocol agnosticism*:

infatti, ne il Guacamole client, ne la web application devono essere informati su quale tipo di remote desktop protocol viene utilizzato.

Il protocollo Guacamole è un protocollo per il rendering remoto di display. Sebbene sulla carta questo protocollo abbia le stesse abilità dei protocolli desktop remoti, i principi con cui è realizzato sono diversi: quest'ultimo infatti non implementa nessuna feature specifica per gli ambienti desktop. Per questo è necessario un livello intermedio che traduca fra il protocollo Guacamole e i protocolli desktop remoti. Il componente che effettua questa traduzione è Guacd.

Guacd

Guacd [12] è il cuore dell'applicazione che dinamicamente carica i supporti per i protocolli desktop remoti, chiamati *plugins*, e li connette al desktop remoto in base alle istruzioni che riceve dalla web application. Guacd è un demone che gira in background e rimane in ascolto per connessioni TCP dalla web application. Anche guacd non "capisce" nessun remote desktop protocol specifico, ma si limita ad implementare quel poco che basta del protocollo guacamole in modo da determinare quale plugin vada caricato. Una volta che un plugin viene caricato, esegue indipendentemente da guacd.

Web application

La parte di Guacamole con la quale l'utente interagisce è la web application, che come menzionato in precedenza, non implementa nessun protocollo ma fa affidamento su guacd. È un interfaccia semplice e pulita con nient'altro che un layer di autenticazione e un menù dal quale è possibile selezionare il pc con cui iniziare la sessione.

2.2.2 Nginx

NGINX è un server web multipiattaforma utilizzato nel progetto per implementare il *reverse proxy* e quindi controllare l'accesso al sistema. Supporta

tutte le classiche funzionalità di un web server, incluso il supporto alla tecnologia SSL/TLS, la funzionalità di *reverse proxying*, bilanciamento di carico e controllo degli accessi. È stato scelto in quanto semplice da configurare e performante nella gestione di grandi carichi di lavoro. Infatti l'architettura di NGINX permette di gestire molte richieste con un basso *overhead* di risorse, attraverso il seguente meccanismo di funzionamento: un processo master delega la gestione delle richieste effettuate al server a dei processi *worker*, i quali riescono ad elaborare richieste multiple attraverso un meccanismo di funzionamento asincrono implementando una modalità di lavoro concorrente. La conseguenza a livello funzionale più importante è la possibilità di sfruttare nella maniera più efficiente possibile le risorse hardware.

2.2.3 PostgreSQL

PostgreSQL è un DBMS ad oggetti *open source* basato sulla versione 4.2 di Postgres e sviluppato all'Università di Berkley in California. Largamente utilizzato per la sua flessibilità, viene utilizzato all'interno del progetto per la creazione di un database nel quale salvare utenti, sessioni e dati di configurazione.

2.2.4 PgAdmin

PgAdmin è il più avanzato strumento *open source* per la gestione di un database PostgreSQL. Fornisce una interfaccia grafica, raggiungibile tramite web browser, per semplificare la creazione, il mantenimento e l'utilizzo del database. Ed è proprio per questo motivo che è stato scelto come ultimo componente della nostra applicazione.

2.3 Strumenti relativi a Docker

Si effettua ora una panoramica degli strumenti di Docker che verranno utilizzati per il dispiegamento dell'applicazione dapprima su singolo *host* e

successivamente in modalità *swarm* su 2 host.

2.3.1 Docker Compose

Docker Compose [13] è uno strumento che permette la definizione e l'esecuzione di applicazioni composte da più servizi, definite all'interno di Docker containers. I servizi che compongono un'applicazione distribuita tramite Compose sono definiti in un file YAML, così come le strutture relative ai container che implementano i vari servizi. I file di configurazione YAML utilizzati da Compose vengono chiamati *compose file*.

Focalizzando l'attenzione sui principali aspetti strutturali, si definiscono di seguito il formato e le varie sezione di un `docker-compose.yml`:

- **services** all'interno della quale si vanno a definire tutti i servizi. Ogni servizio deve specificare alcuni parametri:
  - immagine dalla quale istanziare il container;
  - rete alla quale connettere il container;
  - volumi;
  - politiche di riavvio;
  - variabili di ambiente;
  - esposizione di porte.
- **version** che definisce la versione di Docker Compose da utilizzare per il file;
- **networks** che va a definire le reti;
- **volumes** per la definizione delle strutture da utilizzare all'interno dell'applicazione che si sta istanziando.

Docker Compose permette la gestione dell'intero ciclo di vita di un'applicazione multi-container e consente di operare con i servizi che compongono il sistema in maniera semplificata offrendo una visione di insieme e funzionalità per monitorare i container in esecuzione.

2.3.2 Docker Swarm

Docker Swarm [13] permette di orchestrare più container distribuiti su un *cluster* di *Docker Engine*, e di conseguenza anche su più host. *Docker Swarm* adotta un modello dichiarativo ossia permette di definire lo stato ottimale di ogni servizio dell'applicazione, il numero di repliche che si vogliono mantenere attive, reti di *overlay* ed addirittura aggiornamenti incrementali di tutto il cluster.

I concetti chiave di *Docker Swarm* sono i seguenti:

swarm : un gruppo di *Docker* hosts che eseguono in modalità *swarm*, configurati per formare un *cluster* logico;

node : un nodo è un istanza di *Docker Engine* che partecipa allo *Swarm*. È possibile avere uno o più nodi anche all'interno della stessa macchina fisica ma solitamente il dispiegamento con *Swarm* viene effettuato su nodi che rappresentano macchine fisiche diverse.

Ogni nodo ha un ruolo (può averli entrambi) tra i seguenti:

nodo manager : all'interno di uno *Swarm* un nodo assume il ruolo di coordinatore del gruppo di macchine che formano il *cluster*. Questo nodo particolare ha il compito di gestire i nodi *worker* affidando a ciascuno di questi un unità di lavoro detta *task* e deve inoltre gestire l'operatività del cluster in maniera efficiente. Di default, un nodo *manager* è allo stesso tempo anche un nodo *worker*;

nodo worker : nodo che esegue il *task* assegnatogli dal nodo manager. Inoltre, ogni nodo worker notifica periodicamente al nodo manager lo stato corrente del task, in modo che il manager possa mantenere lo stato desiderato per ogni servizio.

stack : gruppo di *services*.

service : immagine *Docker* per un componente di un sistema *multi-container* e le relative strutture .Corrisponde alla definizione del task.

task : istanza di un *service*. Concretamente è un *container* che istanzia l'immagine che definisce il servizio e utilizza le relative strutture. Lo stesso servizio può essere implementato da più *task*, i quali possono eseguire su macchine diverse del *cluster* in contemporanea.

Questo strumento permette di scalare i servizi in maniera immediata, in quanto è possibile definire anche a run time il numero di task che si desidera eseguire relativamente a qualunque servizio. Il manager si occupa dell'istanza dinamica dei container in base allo stato definito dall'utente. Inoltre permette la creazione e la gestione di *overlay network*, ovvero reti logiche che sfruttano reti sottostanti per il funzionamento. Ulteriori funzionalità offerte da *Docker Swarm* sono il bilanciamento di carico in caso di esposizione sulla rete di qualche servizio e una funzionalità DNS interna relativa ai vari *task* in esecuzione sul *cluster*.

2.4 Progetto Finale

In seguito a quanto descritto in precedenza è ora possibile definire un progetto a livello software del sistema. Quest'ultimo si comporrà quindi delle due parti di Guacamole, che nella pratica si dividono in:

- *Guacamole-server* contenente guacd e le librerie necessarie;
- *Guacamole-client* contenente la web application e il *servlet container*.

Saranno inoltre presenti, il DBMS PostgreSQL, il *management tool* pgAdmin e il *reverse proxy* Nginx. Ogni componente girerà all'interno di un proprio container e sarà raggiungibile dall'esterno solo tramite Nginx. I container dialogheranno tra loro su una rete bridge dedicata sfruttando il DNS interno fornito da Docker. È possibile osservare l'architettura così realizzata nella fig. 2.2 nella pagina successiva.

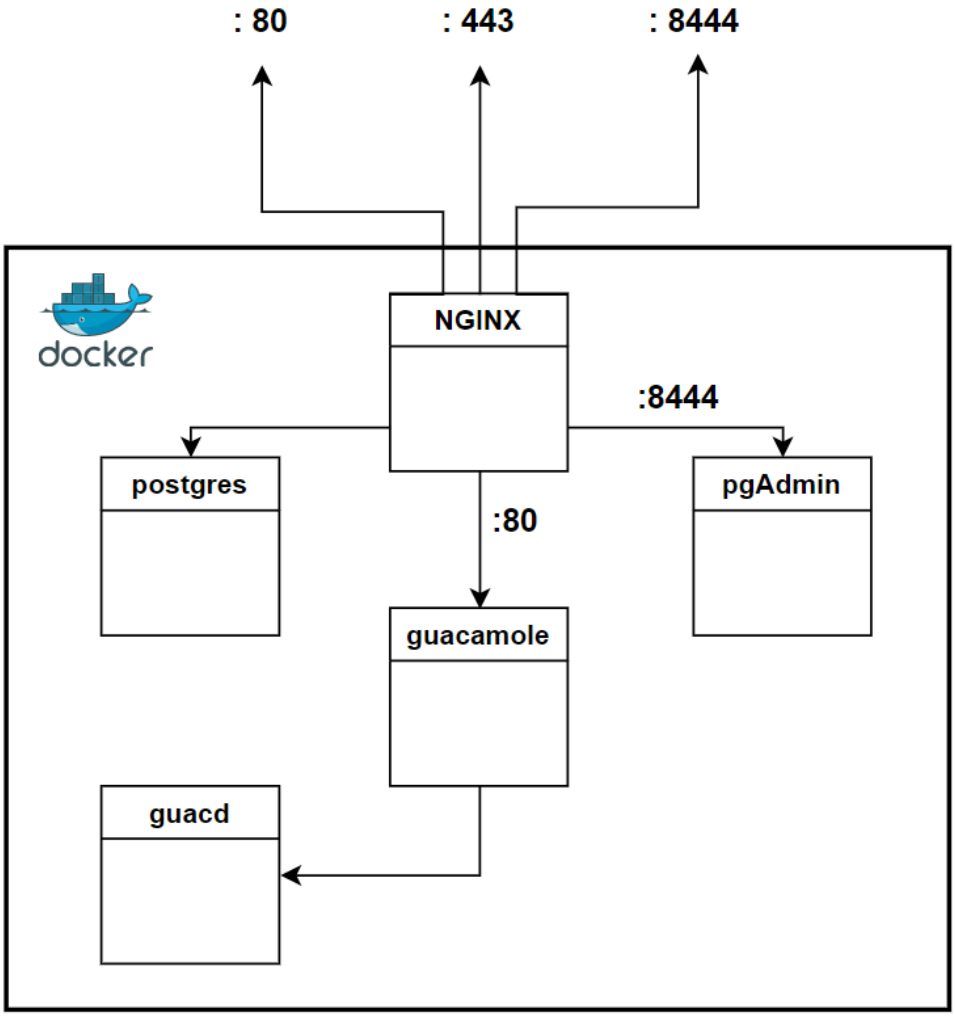

Figura 2.2: Architettura progetto finale

Capitolo 3

Realizzazione

La realizzazione dell'applicazione ha seguito un percorso incrementale che è possibile suddividere nelle seguenti 4 fasi principali :

1. installazione di Apache Guacamole su macchina fisica senza l'utilizzo della piattaforma Docker;

2. creazione delle immagini relative ai componenti descritti in fase progettuale;

3. integrazione automatica di tutti i container su una singola macchina sfruttando Docker Compose;

4. dispiegamento dell'applicazione su due macchine separate, dividendo il servizio guacd dal resto dei componenti. Il dispiegamento dei *cointaner* su più macchine è possibile tramite Docker Swarm sfruttando le funzionalità di *networking multi-host* che offre.

3.1 Installazione di Guacamole su macchina fisica

Come prima fase si è deciso di effettuare una installazione classica di Guacamole per comprendere a pieno il servizio e, soprattutto, identificare tutti

i comandi da trascrivere successivamente nei *docker-file* di ogni componente. Infatti, uno dei problemi riscontrati con le immagini ufficiali di Guacamole era che non venisse fornito il *docker-file* delle stesse. Per una questione di sicurezza, manutenibilità e anche per una valenza didattica si è deciso quindi di ricreare da zero le immagini dei due container principali (section 3.2 a fronte). Quindi, dopo aver predisposto una macchina virtuale con sistema operativo Ubuntu 20.04, si è passati all'installazione del servizio in maniera nativa, seguendo la guida ufficiale di Guacamole [14]. Questo come affermato precedentemente ci ha permesso di comprendere le varie fasi dell'installazione.

In particolare per la parte server i passi da effettuare sono i seguenti:

1. installazione delle librerie;
2. download ed estrazione del codice sorgente dal repository ufficiale;
3. esecuzione dello *script shell* `configure` per la ricerca delle librerie disponibili e dei componenti appropriati per la compilazione del pacchetto;
4. compilazione tramite il comando `make`;
5. installazione del pacchetto tramite il comando `make install`;
6. refresh della cache del sistema per vedere le nuove installazioni tramite il comando `ldconfig`.

Per la parte client invece, una volta installato il Java JDK e scaricato ed estratto l'archivio dal repository ufficiale, tramite Maven è possibile compilare il tutto semplicemente digitando il comando `mvn package` che va a creare un singolo file `.war` contenente la web application. Ovviamente, per poter funzionare, la web app ha bisogno di un *servlet container*, nel nostro caso Tomcat, che è stato installato da riga di comando. Successivamente si è spostato il file `guacamole.war`, all'interno della cartella che Tomcat utilizza per i file `.war` ossia `CATALINA_HOME/webapps`. Una volta riavviati tomcat e guacd, l'intero servizio risultava installato con successo, non disponeva di un database, né era stato minimamente configurato, ma avevamo a disposizione tutti i passi per ricreare le immagini da zero.

3.2 Creazione delle immagini

Come annunciato in precedenza dopo l'installazione nativa di Guacamole, si è passati alla realizzazione delle immagini relative ad ogni componente descritto in fase di progettazione. Vengono esaminate di seguito le immagini per ogni microservizio del sistema.

3.2.1 Immagine di Guacamole

Per la generazione dell'immagine di Guacamole, ossia il client, contenente la web application, era emersa nella fase di installazione nativa la necessità del *servlet container* Tomcat. Perciò, in fase di definizione dell'immagine si è deciso di prendere come base per il *docker-file* l'immagine ufficiale **tomcat9**. Dopo aver riportato tutti i comandi effettuati nella fase precedente, si è aggiunta l'estensione relativa a Radius, andando a copiare il file precompilato all'interno del container con l'opzione `COPY`, ed andando a settare all'interno del file `guacamole.properties` le impostazioni di Radius: hostname, password segreta condivisa, protocollo di accesso, porta di autenticazione, timeout e numero di tentativi.

Si riporta di seguito il Dockerfile per la creazione dell'immagine di guacamole.

guacamole Dockerfile

```
1 ## CLIENT
2 FROM library/tomcat:9-jre11
3
4 ENV ARCH=amd64 \
5     GUAC_VER=1.2.0 \
6     GUACAMOLE_HOME=/config/guacamole
7
8 # Apply the s6-overlay
9 RUN curl -SLO "https://github.com/just-containers/s6-overlay/
    releases/download/v1.20.0.0/s6-overlay-${ARCH}.tar.gz" \
```

```
10   && tar -xzf s6-overlay-${ARCH}.tar.gz -C / \
11   && tar -xzf s6-overlay-${ARCH}.tar.gz -C /usr ./bin \
12   && rm -rf s6-overlay-${ARCH}.tar.gz \
13   && mkdir -p ${GUACAMOLE_HOME} \
14     ${GUACAMOLE_HOME}/lib \
15     ${GUACAMOLE_HOME}/extensions
16
17 WORKDIR ${GUACAMOLE_HOME}
18
19 # Install dependencies
20 RUN apt-get update && apt-get install -y \
21  postgresql-client
22
23 # Install guacamole-client and postgres auth adapter
24 RUN set -x \
25   && rm -rf ${CATALINA_HOME}/webapps/ROOT \
26   && mkdir ${CATALINA_HOME}/extensions \
27   && curl -SLo ${CATALINA_HOME}/webapps/ROOT.war
       "https://downloads.apache.org/guacamole/${GUAC_VER}/
       binary/guacamole-${GUAC_VER}.war" \
28   && curl -SLo ${GUACAMOLE_HOME}/lib/postgresql-42.1.4.jar
       "https://jdbc.postgresql.org/download/ postgresql-42.1.4.jar" \
29   && curl -SLO "https://downloads.apache.org/guacamole/${GUAC_VER}/
       binary/guacamole-auth-jdbc-${GUAC_VER}.tar.gz" \
30   && tar -xzf guacamole-auth-jdbc-${GUAC_VER}.tar.gz \
31   && cp -R guacamole-auth-jdbc-${GUAC_VER}/postgresql/
       guacamole-auth-jdbc-postgresql-${GUAC_VER}.jar
       ${GUACAMOLE_HOME}/extensions/
       guacamole-auth-02-jdbc-postgresql-${GUAC_VER}.jar \
32   && cp -R guacamole-auth-jdbc-${GUAC_VER}/postgresql/schema
       ${GUACAMOLE_HOME}/ \
33   && rm -rf guacamole-auth-jdbc-${GUAC_VER}
       guacamole-auth-jdbc-${GUAC_VER}.tar.gz
```

```
34
35 WORKDIR /config
36
37 # Copio solamente i file necessari dalla cartella root al container
38 COPY root/etc /etc
39 COPY root/app/extensions/guacamole-auth-01-radius-1.2.0.jar
       ${CATALINA_HOME}/extensions
40 ENTRYPOINT [ "/init" ]
```

Un'importante modifica è stata fatta anche allo script di startup `/guacamole-build/root/etc/services.d/guacamole/run` che viene richiamato all'avvio del container. Questo script svolge due compiti essenziali:

1. verificare che il database PostreSQL esista. In caso contrario procede alla creazione di quest'ultimo all'interno del container postgres, utilizzando l'username e le password stabilite nel file `.env`;

2. controllare che la versione di guacamole presente nel compose file coincida con quella riportata sul database. In caso le due differissero va ad effettuare l'upgrade del database.

Si riporta di seguito il contenuto del file appena descritto

run script

```
1 #!/usr/bin/with-contenv sh
2
3 echo "$POSTGRES_HOSTNAME:5432:postgres:$POSTGRES_USER:
      $POSTGRES_PASSWORD" > ~/.pgpass
4 echo "$POSTGRES_HOSTNAME:5432:$POSTGRES_DB:$POSTGRES_USER:
      $POSTGRES_PASSWORD" >> ~/.pgpass
5 chmod 0600 ~/.pgpass
6 # Create database if it does not exist
7 psql -h $POSTGRES_HOSTNAME -U $POSTGRES_USER postgres -lqt | cut -d
      \| -f 1 | grep -qw $POSTGRES_DB
```

```
if [ $? -ne 0 ]; then
  createdb -h $POSTGRES_HOSTNAME -U $POSTGRES_USER $POSTGRES_DB
  cat /app/guacamole/schema/*.sql | psql -h $POSTGRES_HOSTNAME -U
      $POSTGRES_USER -d $POSTGRES_DB -f -
  echo "$GUAC_VER" > /config/.database-version

  /etc/cont-init.d/30-defaults.sh
  /etc/cont-init.d/50-extensions
else
  if [ "$(cat /config/.database-version)" != "$GUAC_VER" ]; then
    cat /app/guacamole/schema/upgrade/upgrade-pre-$GUAC_VER.sql |
        psql -h $POSTGRES_HOSTNAME -U $POSTGRES_USER -d $POSTGRES_DB
        -f -
    echo "$GUAC_VER" > /config/.database-version
  fi
fi

echo "Starting guacamole client..."
s6-setuidgid root catalina.sh run
```

3.2.2 Immagine Guacd

Per la realizzazione del Dockerfile del guacamole-server, ossia quello contenente guacd, il servizio che presenta il carico di lavoro maggiore, si è deciso di utilizzare come base Alpine: una distribuzione di Linux estremamente leggera, alla quale sono stati aggiunti i pacchetti necessari all'esecuzione del servizio. Seguendo il comandi testati durante l'installazione nativa il risultato ottenuto è il seguente:

guacd Dockerfile

```
## SERVER
FROM alpine

```

```
4  ENV GUACD_VERSION    1.2.0
5  ENV CXX='gcc'
6
7  RUN
8      apk add --update --no-cache cairo libjpeg-turbo libpng pango \
9      libssh2 libvncserver openssl libvorbis libwebp libsndfile \
10     pulseaudio libusb freerdp ffmpeg-dev wine-libs libwebsockets
11
12 RUN
13     apk add --update --no-cache --virtual .build-deps \
14     git make automake autoconf cmake gcc libtool \
15     build-base linux-headers bsd-compat-headers intltool \
16     musl-dev cairo-dev libjpeg-turbo-dev libpng-dev \
17     pango-dev libssh2-dev libvncserver-dev openssl-dev \
18     libvorbis-dev libwebp-dev libsndfile-dev pulseaudio-dev \
19     libusb-dev freerdp-dev libwebsockets-dev && \
20     \
21     mkdir /tmp/build && \
22     cd /tmp/build && \
23     \
24     git clone https://github.com/sean-/ossp-uuid.git && \
25     cd ossp-uuid && \
26     ./configure && \
27     make && \
28     make install && \
29     cd .. && \
30     ln -s /usr/local/lib/libuuid.so.16.0.22 /lib/libossp-uuid.so &&\
31     \
32     git clone --branch 0.23
           https://github.com/seanmiddleditch/libtelnet.git && \
33     cd libtelnet && \
34     autoreconf -i && \
35     autoconf && \
```

```
36      ./configure && \
37      make && \
38      make install && \
39      cd .. && \
40      \
41      git clone --branch $GUACD_VERSION
            https://github.com/apache/guacamole-server.git && \
42      cd guacamole-server && \
43      autoreconf -i && \
44      autoconf && \
45      ./configure && \
46      make && \
47      make install && \
48      cd .. && \
49      \
50      apk del .build-deps && \
51      rm -Rf /tmp/build && \
52      rm -f /var/cache/apk/* && \
53      mkdir -p /usr/share/fonts/TTF
54  #FONT SSH Client
55  COPY LiberationMono-Regular.ttf /usr/share/fonts/TTF/
56  EXPOSE 4822
57  #Comando eseguito allo start del container (-b interfaccia in
        ascolto) (-f esegue il processo in foregroud)
58  CMD ["/usr/local/sbin/guacd", "-b", "0.0.0.0", "-f"]
```

3.2.3 Immagini di Nginx, PostgreSQL e pgAdmin

Per quanto riguarda le immagini di Nginx, PostgreSQL e pgAdmin si è deciso di utilizzare quelle ufficiali presenti sul repository Docker-Hub. Questi sono servizi estremamente complessi di cui viene consigliato un utilizzo *"out-*

of-the-box" [1]. Nel nostro caso dovendoli integrare con altri container abbiamo dovuto effettuare alcuni passi di configurazione che verranno trattati nella sezione successiva.

3.3 Integrazione dei container tramite Docker Compose

Una volta realizzate le immagini si è proceduto all'integrazione delle stesse tramite Docker Compose. Come esposto nella section 2.3.1, Docker compose è uno strumento che permette di descrivere in un unico file `docker-compose.yml` tutti i container da creare per la realizzazione di un'applicazione multi-container. All'interno della sezione "services" del compose file, è possibile inoltre specificare una serie di configurazioni che sarebbero altrimenti state passate da riga di comando in fase di creazione del container. È stata inoltre sfruttata una peculiarità di Docker Compose : il file `.env`. Quest'ultimo permette di definire delle variabili di ambiente, utilizzabili da tutti i container. In questo modo è possibile condividere il compose file senza mettere a rischio la sicurezza del sistema poiché all'interno di quest'ultimo sono solo presenti variabili in stile bash che vengono sostituite a compile-time con i valori presenti nel file `.env`.

Si allega di seguito il *docker compose file* così realizzato.

docker-compose.yml

```
1 version: '2.0'
2
3 # networks
4 # create a network 'guacnetwork_compose' in mode 'bridged'
5 networks:
6   guacnetwork_compose:
```

[1] Espressione utilizzata per indicare qualcosa che funziona immediatamente dopo averlo acceso o avviato, così come "esce dalla scatola", e che non ha bisogno di configurazione aggiuntiva per farlo

```
7     driver: bridge
8
9  # services
10 services:
11   # guacd
12   guacd:
13     container_name: guacd_compose
14     image: guacd:Dockerfile
15     networks:
16       guacnetwork_compose:
17     restart: always
18     volumes:
19     - ./drive:/drive:rw
20     - ./record:/record:rw
21
22   # postgres
23   postgres:
24     container_name: postgres_guacamole_compose
25     image: postgres
26     networks:
27       guacnetwork_compose:
28     restart: always
29     environment:
30       PGDATA: /var/lib/postgresql/data/guacamole
31       POSTGRES_DB: ${POSTGRES_DB}
32       POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
33       POSTGRES_USER: ${POSTGRES_USER}
34     volumes:
35      - ./init:/docker-entrypoint-initdb.d:ro
36      - ./data:/var/lib/postgresql/data:rw
37     ports:
38      - 5432:5432
39
```

```
40  # pgadmin
41  pgadmin:
42   container_name: pgadmin_guacamole_compose
43   image: dpage/pgadmin4:latest
44   restart: always
45   environment:
46    PGADMIN_DEFAULT_EMAIL: ${PGADMIN_DEFAULT_EMAIL}
47    PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_DEFAULT_PASSWORD}
48   networks:
49    guacnetwork_compose
50   volumes:
51   - ./pgadmin:/var/lib/pgadmin
52
53  # guacamole
54  guacamole:
55    container_name: guacamole_compose
56    depends_on:
57    - guacd
58    - postgres
59    image: guacamole:Dockerfile
60    environment:
61     GUACD_HOSTNAME: guacd
62     #GUACAMOLE_HOME: /ghome
63     POSTGRES_HOSTNAME: postgres
64     POSTGRES_DB: ${POSTGRES_DB}
65     POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
66     POSTGRES_USER: ${POSTGRES_USER}
67    volumes:
68           - ./ghome:/root:rw
69           - ./guacamole_config:/app/guacamole
70    links:
71    - guacd
72    networks:
```

```
    guacnetwork_compose
  restart: always

# nginx
nginx:
 container_name: nginx_guacamole_compose
 restart: always
 image: nginx
 volumes:
 - ./nginx/ssl/self.cert:/etc/nginx/ssl/self.cert:ro
 - ./nginx/ssl/self-ssl.key:/etc/nginx/ssl/self-ssl.key:ro
 - ./nginx/ssl/csi-rlab.campusfc.unibo.it.cert.cer:/
     etc/nginx/ssl/csi-rlab.campusfc.unibo.it.cert.cer:ro
 - ./nginx/ssl/csi-rlab.campusfc.unibo.it.key:/etc/nginx/
     ssl/csi-rlab.campusfc.unibo.it.key:ro
 - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
 - ./nginx/mysite.template:/etc/nginx/conf.d/default.conf:ro
 - ./nginx/pgadmin.template:/etc/nginx/conf.d/
     default_pgadmin.conf:ro
 - ./nginx/redirect.template:/etc/nginx/conf.d/redirect.conf:ro
 ports:
 - 80:80
 - 443:443
 - 8444:8444
 links:
 - guacamole
 networks:
   guacnetwork_compose:
 # run nginx
 command: /bin/bash -c "nginx -g 'daemon off;'"
```

Una volta completata la realizzazione del progetto si è passati alla fase di configurazione di Guacamole. Infatti, era necessario inserire all'interno

del database tutti gli studenti e i laboratori che permettessero appunto l'accesso da remoto. Per fare ciò abbiamo effettuato il *dump* del database dell'università, e ne abbiamo effettuato il backup all'interno del container postgres sfruttando la comoda interfaccia di pgAdmin. Non è stato necessario alcun intervento di configurazione sui computer di laboratorio, in quanto sui pc Windows è attiva di default la possibilità di connettersi in maniera remota con il vincolo dell'utilizzo dell'autenticazione a livello di rete *NLA*. Configurando opportunamente Guacamole è possibile utilizzare RDP *Remote Desktop Protocol* con NLA *Network Level Authentication* che, come descritto nella section 1.1.2 a pagina 4, consente di autenticare il computer che tenta di connettersi in remoto, prima che si sia instaurata una connessione completa, cioè prima che venga visualizzata la finestra con la richiesta delle credenziali di accesso.

3.4 Deployment tramite Swarm

Dopo aver testato il progetto funzionante, completo delle opportune configurazioni, si è passati alla fase di *deploy* tramite Docker Swarm. Come primo passo è stata identificata la macchina **csi-net02** che ha preso parte allo swarm con il ruolo worker, mentre alla macchina **csi-guacamole** è stato assegnato il ruolo di manager.

Successivamente si è modificato il docker compose aggiungendo la voce `deploy` nei 5 servizi disponibili e la voce `replicas` che permette appunto di stabilire il numero di repliche per ogni servizio. Si è deciso inoltre di replicare solo il servizio guacd essendo quello con un maggior carico computazionale. La rete virtuale è stata cambiata da bridge ad overlay, in quanto Swarm richiede appunto di vedere le varie macchine come facenti parte della stessa rete.

Infine, si è deciso di forzare il deploy di tutte e due le repliche di guacd sul server csi-net02 per osservare le capacità di Swarm nel ridirigere il traffico

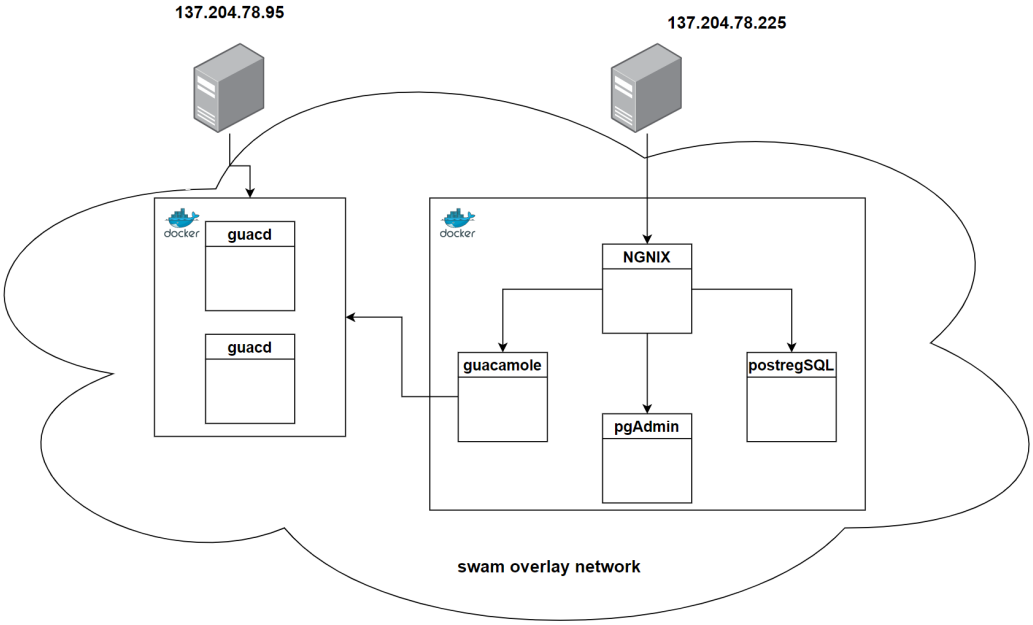

Figura 3.1: Architettura del deploy tramite Swarm

all'interno della rete di overlay. A tal proposito si fa riferimento alla fig. 3.1 per osservare l'architettura del dispiegamento tramite Swarm.

Completata la fase di dispiegamento sulle due macchine si è passati a testare anche questa soluzione, per comprendere se l'utilizzo di Swarm potesse rappresentare una buona soluzione per la scalabilità del sistema.

Capitolo 4

Testing

La fase di testing di Guacamole aveva un duplice obiettivo: verificare che il sistema funzionasse correttamente e analizzare le prestazioni del server che ospitava l'applicazione dockerizzata, traendo da queste, considerazioni sulle risorse hardware necessarie al servizio.

Il test è stato realizzato seguendo step incrementali:

1. realizzazione di una sola istanza del test tramite Selenium;
2. parallelizzazione del test tramite thread e TestNG;
3. realizzazione di un test dockerizzato, parallelo e remoto.

4.1 Fasi di creazione dell'architettura del test

4.1.1 Singola istanza di test

La prima fase di realizzazione del test è stata rappresentata innanzitutto dalla scelta di uno strumento che permettesse di automatizzare la navigazione web. La scelta è ricaduta su **Selenium** [15] : un insieme di librerie e strumenti sviluppati da Umbrella che sfruttando le API fornite dai vari browser permette l'automazione web.

Il funzionamento alla base di Selenium risiede nel DOM delle pagine web. Selenium è infatti in grado di simulare delle azioni (click del mouse, inserimento da tastiera) su elementi del DOM preventivamente selezionati tramite identificatori o regole css.

Il primo test realizzato era così organizzato:

1. collegamento al server di test 137.204.78.225 sulla porta 443;
2. accesso a Guacamole con le credenziali di un utente di test;
3. mantenimento della sessione attiva per 5 minuti;
4. chiusura del browser.

codice per una istanza di test

```
System.setProperty("webdriver.chrome.driver",
"C:\\Users\\matte\\Downloads\\chromedriver_win32\\chromedriver.exe");
WebDriver driver = new ChromeDriver();
driver.get("https://137.204.78.225:443");
driver.manage().window().maximize();
driver.findElement(By.cssSelector("input[type=text]")).
    sendKeys("test@campusfc");
Thread.sleep(1000);
driver.findElement(By.cssSelector("input[type=password]")).
    sendKeys("");
driver.findElement(By.className("login")).click();
Thread.sleep(30000);
driver.close();
```

Questo prima versione del test ci ha permesso di comprendere il funzionamento di Selenium e di mostrare il corretto funzionamento di Guacamole. Una sola istanza però non rappresentava alcun caso significativo di studio, vi era quindi la necessità di parallelizzare il seguente test per verificare il comportamento di Guacamole con un carico di lavoro più elevato. Si è passati quindi alla fase di parallelizzazione.

4.1.2 Parallelizzazione del test

La prima rudimentale idea per rendere il test parallelo è stata quella di sfruttare più **thread**, e, da un ciclo `for` all'interno del Main, istanziare più classi di Test e passare al costruttore di ciascuna di esse un intero progressivo, per poter effettuare il login con utenti diversi (test0@campusfc, test1@campusfc ...).

primo test parallelo

```
public void doTest(int n) throws InterruptedException {
    //parte precedente
    driver.findElement(By.cssSelector("input[type=text]"))
    .sendKeys("test" + n + "@campusfc");
    //parte successiva
}
public static void main(String[] args) throws InterruptedException {
    int nReplica = 10;
    LinkedList<Thread> threads = new LinkedList<Thread>();
    for(int i=0;i<nReplica;i++) {
        threads.add(new Test(i));
        threads.get(i).start();
    }
}
```

Successivamente però si è deciso di raffinare la soluzione sfruttando lo strumento **TestNG** che permette di gestire la parallelizzazione ad un livello di astrazione superiore. TestNG [16] infatti è un *framework* di test pensato per semplificare un ampio range di necessità riguardanti il test di applicazioni, dallo *unit test* (testando una classe isolandola dalle altre) al test di integrazione, ossia testando l'intero sistema. Utilizzando le annotazioni `@testNG` sul codice, viene prodotto un file `testing.xml` che permette poi di eseguire la determinata classe come TestNG.

Nel nostro caso, si è fatto un utilizzo estremamente semplificato di TestNG

andando ad effettuare un test a livello di metodo. In questo modo il suddetto metodo può essere invocato tante volte quante `invocatin count` e può essere dispiegato su **n** thread quanti sono i `threadPoolSize`.

Successivamente, per riottenere il contatore che precedentemente era realizzato dal main, si è optato per l'utilizzo di un `AtomicInteger` che ad ogni invocazione del metodo incrementasse il contatore.

La soluzione così realizzata risulta sicuramente più elastica, scalabile e riutilizzabile in quanto potendo variare il numero di invocazioni del metodo, è possibile effettuare test con un numero di sessioni via via maggiore. Il limite alla soluzione qui proposta però è risultato essere il mio pc, che non riusciva a sostenere il carico di più di 10 istanze di test. È iniziata quindi una fase di ricerca volta ad alleggerire il carico sulla mia macchina. Questa mi ha portato a notare uno strumento di Selenium, il **Remote Web Driver**, che abbinato ai Docker container ed a **VNC Viewer** mi ha permesso di cambiare completamente la struttura e la modalità di esecuzione del test.

4.1.3 Test dockerizzato, parallelo e remoto

Lo stadio finale del test è stato raggiunto grazie all'utilizzo di *Selenium Remote Web Driver*, che permette di eseguire le varie automazioni del browser, non sul pc che sta direttamente eseguendo il test, bensì su un pc remoto. Questo ha permesso di ridurre il carico di lavoro sulla mia macchina, trasferendolo su un server dell'università adatto allo scopo proprio per le sue caratteristiche hardware elevate (64 GB di RAM).

Arrivati a questo punto, si è pensato di dimostrare ancora una volta la potenzialità dei *Docker container* sfruttando questi ultimi come ambiente per il test. La creazione dei container è avvenuta utilizzando l'immagine ufficiale `selenium/standalone-chrome-debug` che oltre a contenere tutto il necessario per lavorare con Selenium, contiene anche un leggerissimo ambiente grafico e un VNC server. Questo significa che configurando opportunamente il *mapping* delle porte del container in fase di creazione dello stesso, è possibile

osservare il comportamento del test in esecuzione sul container visionando il desktop di quest'ultimo tramite lo strumento VNC viewer.

Successivamente sono stati realizzati 2 semplici script di creazione e distruzione di `n` (in questo caso 80) selenium container in modo da automatizzare ulteriormente il test.

script di creazione

```
for X in $(seq 80)
do
     docker run -d -p$(( 50000+$X)):4444 -p $((50200+$X)):5900
         --shm-size=2g --name=guacdTest$X
         selenium/standalone-chrome-debug
done
```

script di distruzione

```
for X in $(seq 80)
do
    docker stop guacdTest$X
    docker rm guacdTest$X
done
```

Inoltre, rispetto al codice del test intermedio, sono stati aggiunti dei controlli sulla presenza degli elementi da ricercare nel DOM, in quanto si era notato che spesso il fallimento del test era dovuto alla ricerca di elementi non era ancora del tutto caricati. Inoltre, si è dilatato il tempo di attesa nella sessione per permettere a **Zabbix**, lo strumento utilizzato per monitorare il server, di effettuare 4 campionamenti durante lo stress test. Infine, passato questo tempo, viene simulato l'invio da tastiera della shortcut `CTRL+ALT+SHIFT` che permette di aprire il menù a tendina di Guacamole ed effettuare il logout dalla sessione.

codice finale del test

```
AtomicInteger sequence = new AtomicInteger(-1);
```

```
private String ip = "137.204.78.99";


@Test(threadPoolSize = 80 , invocationCount = 80)
public void test(ITestContext testContext) throws
    InterruptedException {
 URL url=null;
 int count= sequence.addAndGet(1);
 String port = String.valueOf(50000+count);

ChromeOptions options = new ChromeOptions();
options.setCapability(CapabilityType.PLATFORM_NAME, Platform.LINUX);
options.setCapability(CapabilityType.BROWSER_NAME, "chrome");

System.out.println("Starting interface with ip " + ip + " and port
    " + port);
 try {
  url = new URL("http://" + ip + ":" + port + "/wd/hub");
 } catch (MalformedURLException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
 }
 System.out.println(url.toString());
 RemoteWebDriver wd = new RemoteWebDriver(url, options);
 Actions actions = new Actions(wd);
 wd.get("https://137.204.78.225:443");
 wd.manage().window().maximize();
 Thread.sleep(4000);
 wd.findElement(By.id("details-button")).click();
 wd.findElement(By.id("proceed-link")).click();
 if(wd.findElements(By.cssSelector("input[type=text]")).size() ==
     0)System.out.println("ELEMENTO username NON TROVATO");
 wd.findElement(By.cssSelector("input[type=text]")).sendKeys("test"
```

```
    + count + "@campusfc");

Thread.sleep(4000);
if(wd.findElements(By.cssSelector("input[type=password]")).size()
    == 0)System.out.println("ELEMENTO password NON TROVATO");
wd.findElement(By.cssSelector("input[type=password]"))
    .sendKeys("Lporjfm3425ss!!");
Thread.sleep(4000);
if(wd.findElements(By.className("login")).size() ==
    0)System.out.println("ELEMENTO login NON TROVATO");
wd.findElement(By.className("login")).click();
Thread.sleep(300000);
actions.keyDown(Keys.CONTROL);
actions.keyDown(Keys.ALT);
actions.keyDown(Keys.SHIFT);
actions.perform();
Thread.sleep(5000);
if(wd.findElements(By.className("user-menu")).size() ==
    0)System.out.println("ELEMENTO menu-indicator NON TROVATO");
else{
 wd.findElements(By.className("user-menu")).get(0).click();
}
Thread.sleep(4000);
if(wd.findElements(By.className("logout")).size() ==
    0)System.out.println("ELEMENTO exit NON TROVATO");
else {
 wd.findElements(By.className("logout")).get(0).click();
}
System.out.println("test " + count + "effettuato");
wd.close();
}
```

4.2 Esecuzione del test

Una volta esposti i passaggi che hanno portato alla realizzazione della versione finale del test, e l'ambiente in cui verrà eseguito, passiamo ora alla spiegazione di come effettivamente è stato eseguito il test.

Scenario di test

Il laboratorio oggetto di test è il laboratorio 2.2 del campus di Cesena che dispone di 98 computer, con un range di indirizzi ip che vanno rispettivamente dal `137.204.75.1` al `137.204.75.98`.

Questo test **simula una situazione verosimile**, in cui una classe di 80 studenti prende **virtualmente** posto all'interno del laboratorio. Vengono creati quindi 80 container sul server di appoggio `137.204.78.99` e contestualmente viene monitorato tramite VNC viewer il desktop di uno di questi, per verificare ciò che vi accade all'interno. È sufficiente osservare un solo container in quanto nei restanti 79 viene eseguito *esattamente* lo stesso test in parallelo.

Successivamente, viene lanciato il test da Eclipse sulla mia macchina personale che contatta i *remote web driver* degli 80 *container* all'indirizzo `137.204.78.99`. Questi, a loro volta, richiedono la pagina web di Guacamole all'indirizzo `137.204.78.225`. A questo punto, in ogni container, un utente di test univoco accede al servizio, mantiene la sessione RDP aperta per 5 minuti, ed infine effettua il logout.

Il seguente test, visibile in fig. 4.1 nella pagina successiva, è stato eseguito alla stessa maniera sia con l'applicazione dockerizzata su una singola macchina, sia su un cluster formato da due macchine con l'applicazione replicata e dispiegata tramite Docker Swarm. L'organizzazione dell'applicazione è infatti trasparente all'utente finale.

Si procede ora all'analisi dei risultati così ottenuti.

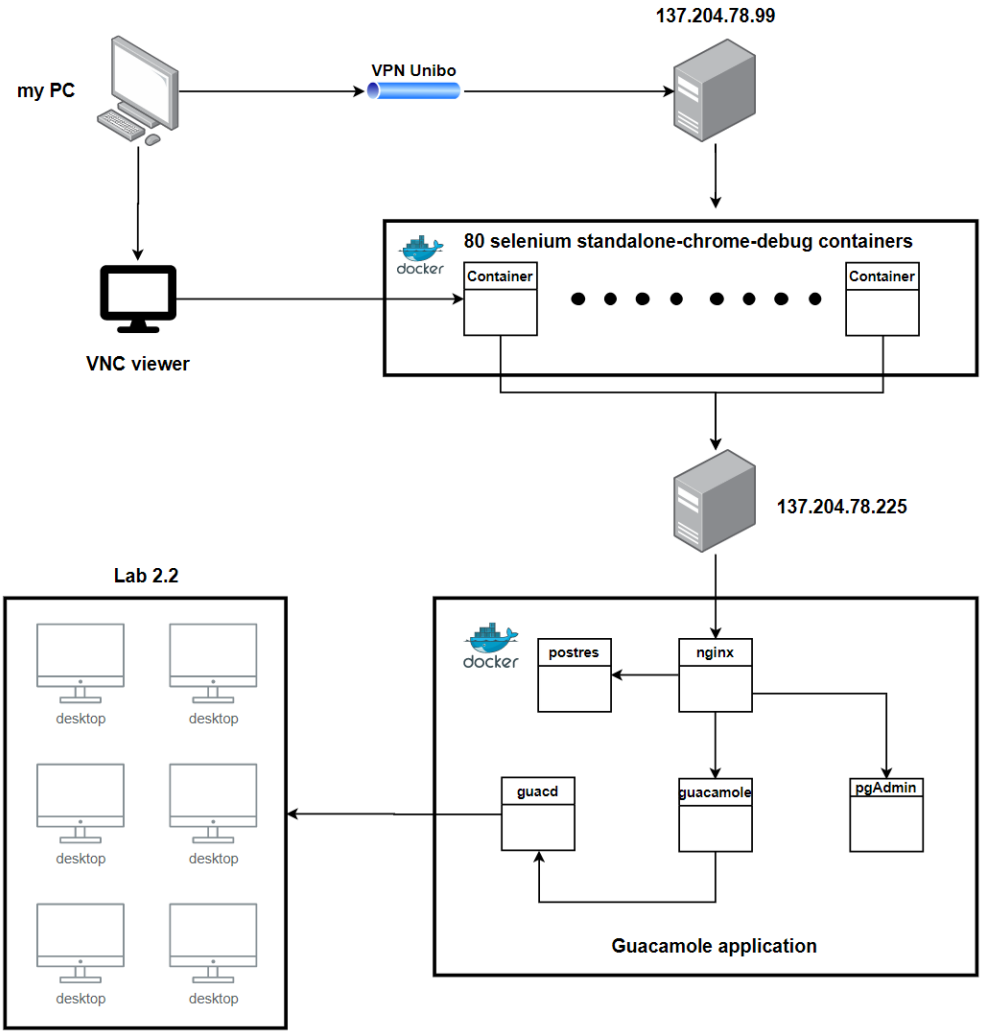

Figura 4.1: Rappresentazione dell'architettura del test

4.3 Analisi dei risultati del test

Durante il test, per monitorare i server si è fatto uso di **Zabbix**. Quest'ultimo è un software *open source* che richiede l'installazione di un demone sui server di test, il quale registra i tutti i parametri e li rende accessibili via web. In particolare, i parametri che abbiamo ritenuto più significativi per l'analisi delle prestazioni del servizio sono i seguenti:

- traffico di rete in entrata ed in uscita;
- memoria RAM disponibile;
- numero di processi;
- utilizzo della CPU.

Viene ora analizzato, per le **2 installazioni di Guacamole** (singola macchina e cluster), ogni parametro sopracitato, corredando ad esso il relativo grafico ed una spiegazione del risultato.

4.3.1 Guacamole installato su macchina singola

Traffico di rete in entrata ed in uscita

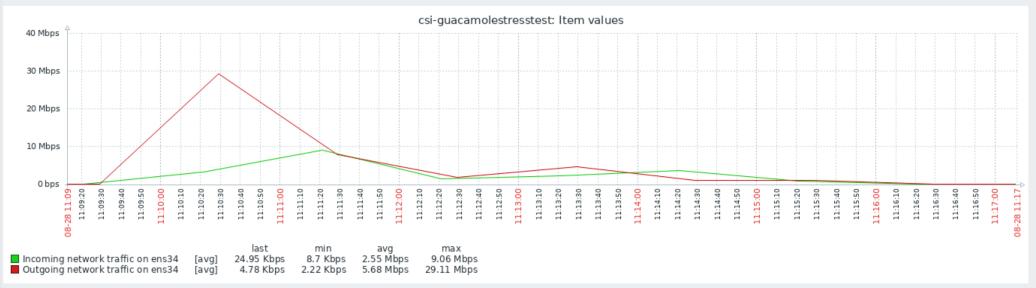

Figura 4.2: Grafico rappresentante il traffico di rete in entrata ed in uscita

Come si evince dalla fig. 4.2 il traffico in uscita è sicuramente più alto di quello in ingresso in quanto il server manda ad ogni client lo streaming video, mentre i client effettuano semplicemente una richiesta al server. Il picco

massimo in uscita, di 29.11 Mbps, si raggiunge nel momento in cui tutti gli 80 container ottengono il desktop remoto, poi va calando poiché lo streaming del desktop per i 5 minuti successivi rimane fisso. Il traffico in ingresso invece, rimane sempre molto basso con un picco di 9,06 Mbps nel momento in cui tutti i container, contemporaneamente, fanno richiesta di una sessione RDP.

Memoria RAM disponibile

Figura 4.3: Grafico rappresentante la memoria RAM disponibile

Dalla fig. 4.3 si nota che dal momento in cui tutte le connessione sono stabilite, la memoria RAM impiegata dal servizio è di circa **1,2 GB**.

Numero di processi

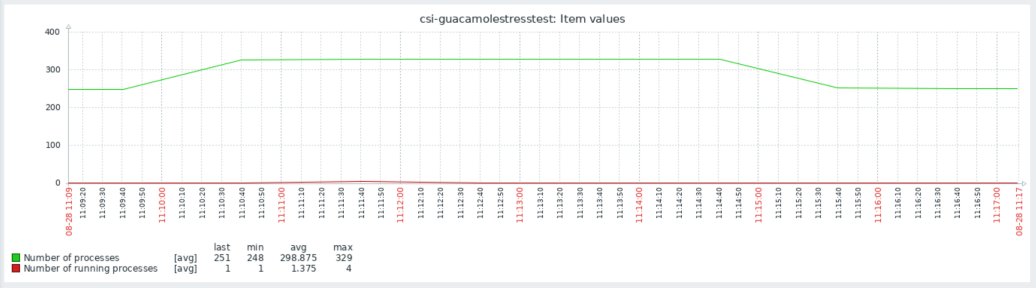

Figura 4.4: Grafico rappresentante il numero di processi

Come si evince dalla fig. 4.4 e come ci si aspettava, il numero di processi nel periodo di mantenimento delle 80 sessioni è cresciuto di esattamente 80 unità, 1 per sessione.

Utilizzo della CPU

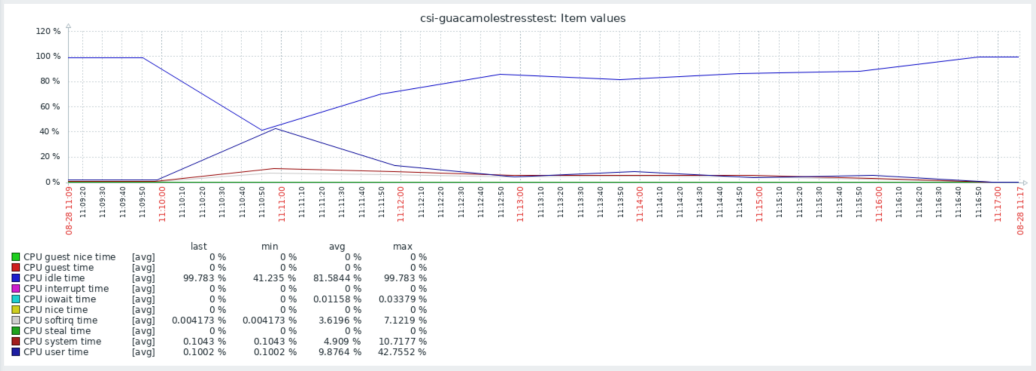

Figura 4.5: Grafico rappresentante l'utilizzo della CPU

Dalla fig. 4.5 si può notare come vari l'utilizzo della CPU. La linea blu più in alto, infatti, rappresenta la CPU nello stato `IDLE`, all'arrivo delle connessioni, la percentuale di CPU inattiva cala, mentre aumenta quella utilizzata fino ad un picco del 42% dopo il quale torna a scendere riposizionandosi su un utilizzo medio del 9%

4.3.2 Guacamole dispiegato su un cluster

Traffico di rete in entrata ed in uscita

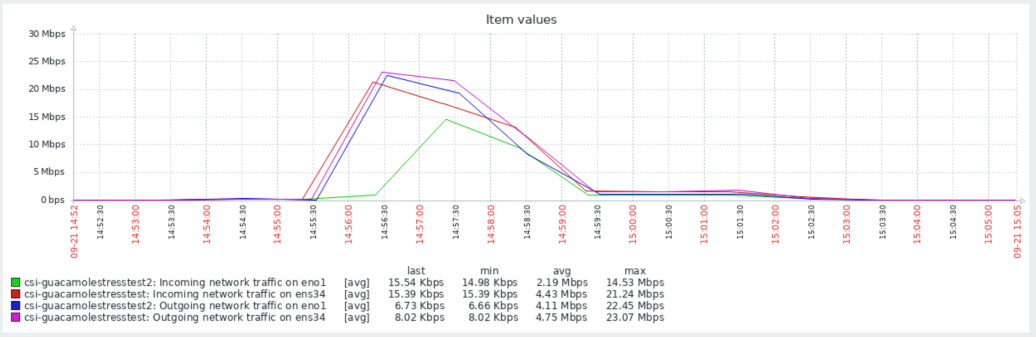

Figura 4.6: Grafico rappresentante il traffico di rete in entrata ed in uscita

Dalla fig. 4.6, si può notare come in questo scenario il traffico in uscita dal server primario sia maggiore rispetto a quello in entrata. Questo è

dovuto principalmente al fatto che guacd si trova sul server secondario, e di conseguenza tutte le richieste di sessione vengono inoltrate a quest'ultimo. Il picco massimo di 23,07 Mbps si raggiunge proprio in questo momento. Il traffico in uscita dal secondo server è giustamente più alto del traffico in ingresso, in quanto corrisponde alla somministrazione ad ogni utente del proprio desktop remoto.

Memoria RAM disponibile

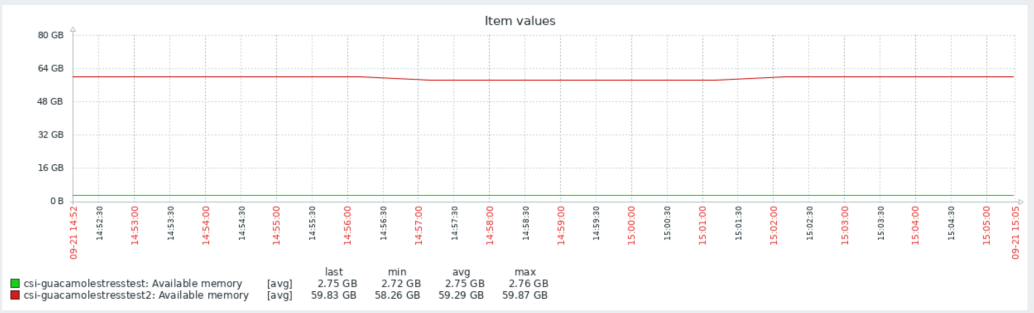

Figura 4.7: Grafico rappresentante la memoria RAM disponibile

Dalla fig. 4.7 si nota innanzitutto la differenza di disponibilità hardware dei due server utilizzati per il test. Il server secondario infatti, vanta 64 GB di RAM, contro i 4 GB del server primario. Su quest'ultimo si può notare invece, come la memoria utilizzata dall'esecuzione di Guacamole sia pressoché nulla, soprattutto se si va a confrontare tale grafico con quello in fig. 4.3 a pagina 47, nel quale il server primario manteneva da solo tutto il servizio. Con il dispiegamento sul cluster, e soprattutto il posizionamento di guacd sul server secondario, abbiamo spostato su quest'ultimo l'utilizzo maggiore di memoria, che si aggira attorno ad 1,6 GB. Il valore leggermente superiore rispetto a quello ottenuto con la singola istanza del servizio (1,2 GB) è da imputare all'utilizzo di Swarm.

Figura 4.8: Grafico rappresentante il numero di processi

Numero di processi

La fig. 4.8 conferma ancora una volta come guacd sia il componente cardine del sistema. Il numero di processi infatti, aumenta solo sul secondo server, di circa 80 unità.

Utilizzo della CPU

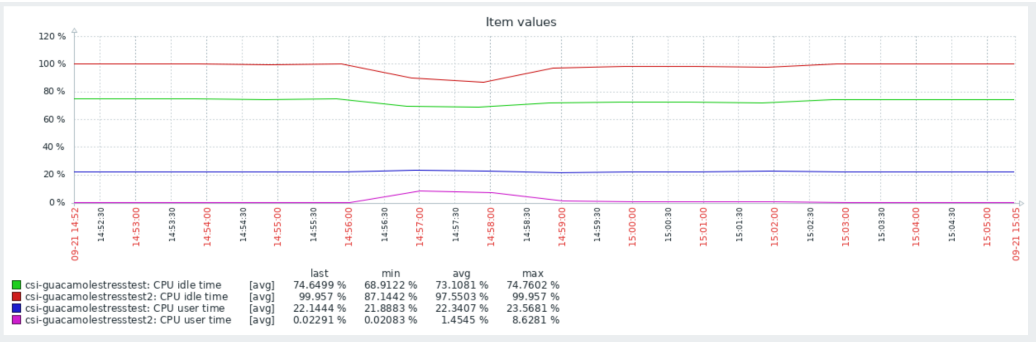

Figura 4.9: Grafico rappresentante l'utilizzo della CPU

Dalla fig. 4.9, è possibile trarre una valutazione estremamente interessante. L'utilizzo del server come nodo manager all'interno dello swarm comporta infatti, un utilizzo costante del 22% della CPU. Questo dato è ragionevole se si pensa alla potenza di Docker Swarm. L'utilizzo della CPU del server secondario ha raggiunto invece un picco di solo 8%.

4.4 Conclusioni del Test

Dai test è emerso che il dispiegamento su macchina singola richiedeva per il mantenimento di 80 sessioni circa 1,2 GB di RAM. Tale valore non rappresenta un problema per la dotazione hardware dei server presenti all'interno dell'università. Inoltre, l'utilizzo di Docker Swarm implica, come si evince dai grafici, un maggiore overhead di risorse della macchina, in particolare CPU, senza un reale miglioramento delle prestazioni che erano già ottime con una sola istanza del servizio.

Per le nostre esigenze attuali, ossia con un centinaio di sessioni contemporanee, una sola istanza si è rivelata essere sufficiente. Nel caso in cui in futuro le sessioni contemporanee aumentino considerevolmente, tramite l'utilizzo di Docker Swarm, sarà possibile dispiegare ed orchestrare più istanze dell'applicazione sopperendo a qualunque necessità.

Scenari di utilizzo di Guacamole nella didattica

Guacamole potendo sfruttare a basso livello sia il protocollo VNC che il protocollo RDP offre moltissime opportunità di utilizzo in ambito didattico. Come descritto in precedenza, entrambi i protocolli presentano punti di forza e criticità, in particolare:

- RDP fornisce all'utente un desktop virtuale, mentre VNC fornisce lo streaming dello schermo fisico;

- RDP permette l'utilizzo di una macchina da un solo utente, mentre VNC permette la condivisione della stessa macchina (periferiche incluse) da più utenti.

Sulla base di queste peculiarità dei protocolli ci sono situazioni in cui si preferisce uno rispetto all'altro. In generale però, in caso sia possibile scegliere fra i due, RDP è sempre preferibile a VNC in quanto oltre a possedere un livello di sicurezza maggiore, non richiede l'installazione di alcun server sulle macchine di laboratorio, essendo preinstallato su tutte le macchine Windows.

Vengono a tal proposito analizzati di seguito 3 diversi scenari di utilizzo di Guacamole nella didattica.

Lezione completamente in remoto

In una lezione di laboratorio svolta completamente da remoto gli studenti dopo essersi autenticati, selezionano il laboratorio sul quale desiderano connettersi. Guacd a questo punto identifica per ogni studente un computer libero ed avvia una sessione RDP. Il docente può in questo caso monitorare il lavoro degli studenti o fornire assistenza a questi ultimi, semplicemente aprendo dal menù di Guacamole le sessioni attive e cliccando sullo studente desiderato. La possibilità di entrare in una sessione altrui, solitamente, spetta solo ad un utente amministratore, ma è possibile dare tale privilegio ad un docente per un determinato laboratorio.

In questo scenario, Guacamole offre una funzionalità particolarmente utile per permettere il lavoro in coppia su uno stesso desktop remoto. Grazie allo strumento **sharing profile** è infatti possibile per un utente creare un link, tramite il quale un altro utente può osservare o anche lavorare sul medesimo desktop remoto.

Lezione in modalità mista con uno studente per macchina

Nel caso in cui si voglia realizzare una lezione in modalità mista, il docente ha due opportunità:

- scegliere di far utilizzare RDP agli studenti che si connettono da remoto, monitorando il loro operato come visto nella sezione precedente;

- far utilizzare agli studenti remoti VNC, in modo che lui stesso, girando fra i banchi possa visivamente controllare l'operato sia degli studenti in presenza che di quelli remoti, avendo la possibilità di aiutarli in caso di necessità semplicemente agendo sulle periferiche delle macchine fisiche.

Lezione in modalità mista con più studenti per macchina

Nel caso in cui si voglia realizzare una lezione in modalità mista, nella quale venga richiesto che una coppia o un gruppo di studenti lavori sulla stessa macchina, l'unica soluzione è l'utilizzo del protocollo VNC. In questo modo, se un utente remoto richiede l'accesso ad una macchina fisica sulla quale è già connesso un altro studente, quest'ultimo si vedrà apparire un pop-up per accettare o rifiutare l'accesso del secondo utente. In caso di accettazione i due condivideranno lo stesso desktop e le periferiche.

Conclusioni

Il progetto ha raggiunto con successo il suo stadio finale, rispettando tutti i requisiti posti in fase progettuale. Dopo aver testato a fondo l'applicazione, è possibile affermare che quest'ultima si comporti decisamente meglio rispetto alle soluzione che si erano adottate in precedenza per accedere remotamente ai laboratori. Guacamole risulta infatti più reattiva, intuitiva e soprattutto stabile.

L'applicazione è inoltre stata posta in produzione dal Dott. Ciro Barbone, ed è quindi attiva già da più di un mese per tutti gli studenti del campus di Cesena all'indirizzo `https://csi-rlab.campusfc.unibo.it/`. In questo arco temporale decisamente più lungo rispetto agli stress test da noi realizzati, è stato possibile monitorare il comportamento del sistema "a regime", confermando la sua stabilità.

Grazie ad un aggiornamento che è stato effettuato su guacamole, uno dei 5 componenti del sistema, è stato inoltre possibile accertare come l'utilizzo di Docker e la scomposizione in container abbia reso il sistema estremamente flessibile e facilmente manutenibile. L'operazione di aggiornamento infatti, sarebbe risultata onerosa nel caso un cui avessimo realizzato un'applicazione monolitica, mentre nel nostro caso è risultata semplice ed immediata.

L'installazione di questo strumento rappresenta per l'università un importante passo avanti per la fruizione in maniera completa degli insegnamenti a distanza. Già a partire da questo anno accademico infatti, l'applicazione verrà ampiamente utilizzata da docenti e studenti in molti dei corsi offerti dall'ateneo. Il sistema è infatti attualmente in funzione presso 5 diverse sedi,

che hanno fatto richiesta al Dott. Barbone per l'installazione e l'utilizzo della versione custom di Guacamole da noi realizzata.

Ringraziamenti

Vorrei innanzitutto ringraziare il Dott. Ciro Barbone, correlatore di questo lavoro, che mi ha seguito in ogni fase della realizzazione del progetto, mostrandosi sempre disponibile per ogni chiarimento e spiegazione. Spronandomi continuamente a migliorare ciò che avevo realizzato, mi ha condotto in un percorso di crescita personale e professionale accrescendo le mie conoscenze e solidificando quelle pregresse.

Vorrei inoltre ringraziare il prof. Vittorio Ghini che rappresenta per me il modello di professore da emulare, che, con il suo carisma, la sua umiltà e la sua competenza, mi ha fatto innamorare dei suoi corsi, e mi ha spinto a scegliere proprio System Integration come oggetto della tesi.

È doveroso ringraziare la mia famiglia, in particolare mia madre, che da sempre mi sostiene e mi sprona a dare il meglio di me in qualsiasi cosa io faccia. Indispensabili sono stati anche gli amici e soprattutto i compagni di corso, che hanno rappresentato per me una seconda famiglia con la quale ho condiviso le gioie e i momenti difficili di questo cammino.

Un ringraziamento speciale va a Marta, compagna di progetti e di studio, un'amica senza la quale probabilmente non avrei raggiunto questo primo traguardo. Con la sua determinazione e saggezza è da sempre stata il mio punto di riferimento all'interno di questo percorso, incoraggiandomi e supportandomi in ogni decisione.

Bibliografia

[1] Wikipedia. Desktop remoto, 2020. URL https://it.wikipedia.org/wiki/Desktop_remoto.

[2] Wikipedia. Virtual network computing, 2020. URL https://it.wikipedia.org/wiki/Virtual_Network_Computing.

[3] Microsoft. Remote desktop protocol, 2020. URL https://support.microsoft.com/it-it/help/186607/understanding-the-remote-desktop-protocol-rdp.

[4] Wikipedia. Network level authentication, 2020. URL https://en.wikipedia.org/wiki/Network_Level_Authentication.

[5] Brien Posey. Will html5 browsers become the new go-to vdi client? Nov 2016. URL https://searchvirtualdesktop.techtarget.com/opinion/Will-HTML5-browsers-become-the-new-go-to-VDI-client?

[6] B. des Ligneris. Virtualization of linux based computers: the linux-vserver project. In *19th International Symposium on High Performance Computing Systems and Applications (HPCS'05)*, pages 340–346, 2005.

[7] M. Kerrisk. The linux man-pages project. Nov 2016. URL https://man7.org/linux/man-pages/.

[8] Canonical Ltd. Linux container - lxc - introduction, 2020. URL https://linuxcontainers.org/lxc/introduction.

[9] Abel Avram. Docker: Automated and consistent software deployments. March 2013. URL https://www.infoq.com/news/2013/03/Docker/.

[10] Open Container Initiative. About the open container initiative. March 2013. URL https://opencontainers.org/about/overview/.

[11] Wikipedia. Radius, 2020. URL https://it.wikipedia.org/wiki/RADIUS.

[12] Guacamole. Guacamole documentation, 2020. URL https://guacamole.apache.org/doc/gug/.

[13] Docker Inc. Docker documentation, 2020. URL https://docs.docker.com/compose/compose-file/.

[14] Apache. Docker guacamole image documentation, 2020. URL https://guacamole.apache.org/doc/gug/guacamole-docker.html#guacamole-docker-image.

[15] Selenium. Selenium documentation, 2020. URL https://www.selenium.dev/documentation/en/.

[16] TestNG. Testng documentation, 2020. URL https://testng.org/doc/documentation-main.html.

[17] oznu. Docker guacamole, 2020. URL https://hub.docker.com/r/oznu/guacamole/.

[18] koromicha Co-founder of Kifarunix.com. Install apache guacamole on ubuntu 20.04, apr 2020. URL https://guacamole.apache.org/doc/gug/guacamole-docker.html#guacamole-docker-image.

[19] degenerate76. Guacd image, jul 2020. URL https://hub.docker.com/r/degenerate76/dockerfile-guacd/dockerfile.

[20] Boschkundendienst. guacamole-docker-compose, aug 2019. URL https://github.com/boschkundendienst/guacamole-docker-compose/blob/master/docker-compose.yml.

[21] Emanuele Villa. Cos'è e come funziona remote desktop protocol. Feb 2020. URL `https://www.zerounoweb.it/software/cose-e-come-funziona-remote-desktop-protocol/`.