Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Architettura cloud per lo sviluppo multi-piattaforma di Sistemi Operativi: Principi generali

Relatore:
Chiar.mo Prof.
Alessandro Amoroso

Presentata da:
Ettore Di Giacinto

I Sessione
2015-2016

Introduzione

Negli ultimi anni si è visto emergere la cultura *DevOps* nell'industria IT. Analizzeremo l'applicazione di questa metodologia nei cicli di rilascio e di compilazione dei pacchetti di una Distribuzione Linux. Vengono studiate le strategie e i metodi adottati delle principali distribuzioni. Si propone quindi un nuovo approccio sfruttando le tecnologie del campo *DevOps* introducendo un alto grado di scalabilità, flessibilità e riproducibilità anche in ambienti Cloud.

I vantaggi di questo metodo sono fasi di *building*, *testing* e *rilascio* in maniera rapida, efficiente e affidabile. Vedremo quindi come questo approccio aumenta l'automatizzazione nei cicli produttivi per la realizzazione della Distribuzione Sabayon Linux e per la definizione di un'infrastruttura automatizzata attualmente in production.

Indice

1 Introduzione **1**

1.1 Contesto 2

1.1.1 Sistemi operativi Linux 2

1.2 Cloud computing 6

1.3 Ciclo di sviluppo di un sistema operativo 7

1.3.1 Gestione del parco Software 7

1.3.2 Il sistema di rilascio 9

1.3.3 Il Kernel 9

1.4 Scopo del Lavoro 10

1.4.1 Struttura della tesi 11

2 Analisi dei sistemi esistenti **13**

2.1 Distribuzioni Linux 13

2.1.1 Fedora Linux 14

2.1.2 Debian Linux 14

2.1.3 Gentoo Linux 15

2.2 Gestione del Parco Software 16

2.2.1 Vantaggi offerti dalla pacchettizzazione del software . . 16

2.2.2 Gestione delle repository Fedora/RedHat 18

2.2.3 Gestione delle repository Debian: aptly, reprepro, dak, mini-dak 19

2.2.4 Portage 19

2.2.5 Gestione delle repository Sabayon: Eit 20

2.2.6 OBS 20
2.2.7 Sintassi 22
2.3 Gestione delle release 22
2.3.1 SUSE studio 23
2.3.2 Livecd-creator, kickstart 23
2.3.3 Debootstrap, live-build,live-wrapper 23
2.3.4 Catalyst 24
2.3.5 Metro 25
2.3.6 Molecules 27
2.4 Android 27

3 Tecnologie utilizzate e scelte implementative 29
3.1 Motivazioni sull'utilizzo delle metodologie DevOps 30
3.2 Docker 30
3.2.1 Riproducibilità 32
3.3 Vagrant 34
3.4 Packer 34
3.5 Drone 35
3.6 CircleCI 36
3.7 Travis CI 37
3.8 Git 37
3.9 GitHub 37
3.10 DockerHub 38

4 Soluzione per la generazione delle immagini 39
4.1 File di specifica 40
4.2 Compressione dei livelli 41
4.2.1 Decompressione: docker-companion 42
4.2.2 Esportazione di un container 43
4.2.3 Estrazione e ricostruzione 43
4.3 Conversione e rilascio 44
4.3.1 Immagini installabili su baremetal 44

4.3.2 Immagini LXD 44
4.3.3 Immagini Vagrant/QEMU/Virtualbox/VMWare 46
4.3.4 Immagini Docker 46
4.3.5 Snapshots 47
4.3.6 Definizione infrastruttura automatica per il rilascio . . 48
4.4 Applicazione nell'ambito dell'hosting: Scaleway 50

5 Soluzione per la manutenzione delle repository 51

5.1 Metodi per la compilazione automatica 51
5.1.1 Immagine "builder" ibrida per la compilazione automatica 52
5.1.2 Software per la CI/CD Poll: Boson 54
5.1.3 Continous Integration 57
5.1.4 Sabayon Development Toolkit 59
5.2 Creazione e rilascio dei pacchetti 61
5.2.1 Infrastruttura automatizzata 61
5.2.2 QA 71
5.2.3 Rilascio 72

Conclusioni 73

A Prima Appendice 75

Elenco delle figure

1.1 Il sistema operativo dei Chromebook e di OnHub è Gentoo Linux. 5

2.1 Struttura applicativi Flatpak 17
2.2 Struttura Open Build Service 20
2.3 Sintassi Open Build Service 21

3.1 Struttura Docker 31
3.2 Filesystem Docker, AUFS 33
3.3 Filesystem Docker, OverlayFS 34

4.1 Unificazione filesystem Docker con UnionFS 42
4.2 Architettura di conversione delle immagini 45

5.1 Building dei pacchetti con tecnologie CI 60
5.2 Architettura dell'infrastruttura definita tramite Vagrantfiles . 62
5.3 Struttura Virtual Machine con ambienti di compilazione Docker 64
5.4 Sito web per la ricerca dei pacchetti 69
5.5 Interazione ricerca del sito web 69

Capitolo 1

Introduzione

L'ecosistema Linux nel corso di questi ultimi anni si è espanso sempre di più. Il kernel sviluppato da Linus Torvalds a soli 22 anni, prende il nome dal suo autore e accompagnato dalla suite software GNU, riceve contribuzioni da qualsiasi parte del mondo. Il Kernel Linux è un progetto Open Source con licenza GPL, rendendo quindi la redistribuzione e la modifica del sorgente possibile senza alcuni vincoli legali [1].

Oggi, grazie anche ai servizi Cloud, utilizziamo inconsapevolmente il kernel Linux continuamente; è la base fondante su cui la maggior parte dei servizi informatici disponibili sul mercato fanno affidamento ed è praticamente un componente indispensabile. Ci basti pensare alla semplice azione di aprire un browser e visitare una pagina web, dal mandare una e-mail o addirittura effettuare una chiamata telefonica [2].

I sistemi operativi Linux di solito vengono solitamente accompagnati da un gestore di software, chiamato in gergo *pacchettizzatore*, o *gestore di pacchetti* che ha il compito di mantenere uno stato consistente del sistema e permettere l'installazione e la rimozione di componenti aggiuntivi [3].

L'eterogeneità di questo ambiente spinge anche grandi aziende come Google a considerare l'utilizzo delle Distribuzioni Linux per equipaggiare alcune dei suoi prodotti nel mercato. L'esigenza quindi di poter utilizzare un dispositivo con del codice comunitario è venuta a delinearsi nel tempo, fino

a maturare e arrivare ad oggi: distribuire un sistema operativo basato su Linux, per un'azienda non richiede una forte conoscenza specifica dell'architettura complessiva dell'intero sistema ma è solo una questione di oculate scelte software e modifiche di componenti hardware. Le aziende che decidono di costruire sistemi embedded, tablet o dispositivi elettronici di qualsiasi tipo, possono modificare e redistribuire i sistemi operativi basati su Linux correnti con estrema facilità, eliminando del tutto il costo di sviluppare codice dedicato e abbassando il costo di manutenzione (senza contare i costi derivanti da uno scarso riutilizzo del codice) tutto ciò grazie alla licenza GPL con cui è distribuito [4].

1.1 Contesto

1.1.1 Sistemi operativi Linux

Il kernel Linux necessita di una suite di Software per poter soddisfare diverse tipologie di utenti e casi d'uso. Nascono così le distribuzioni Linux, ovvero l'unione tra il Kernel Linux e una suite di software che rendono il Sistema Operativo fruibile per tutti. Oggi possiamo trovare centinaia di distribuzioni Linux, DistroWatch [1] enumera 274 distribuzioni ufficialmente dichiarate in questo momento [5].

La storia delle Distribuzioni Linux

Le prime distribuzioni in ordine temporale furono:

- "Boot/root": distribuita da H. J. Lu. Due dischi fornivano rispettivamente il kernel e un set minimo di strumenti provenienti dal progetto GNU. Il secondo permetteva di creare un filesystem e avere così un sistema operativo funzionante;

[1] sito storico di riferimento per la pubblicazione, review e statistiche riguardo le distribuzioni linux

- MCC Interim Linux, distribuzione sviluppata dal Manchester Computing Centre dell'Università di Manchester;
- TAMU, creata dalla Texas A&M University;
- Softlanding Linux System (SLS), creata da Peter MacDonald nell'agosto 1992;
- Slackware creata da Patrick Volkerding nel 1993 e tuttora attivamente sviluppata;
- Yggdrasil Linux, sviluppata dalla Yggdrasil, fu la prima distribuzione in grado di configurarsi analizzando l'hardware a disposizione.

Il primo sistema, che poteva essere chiamato "distribuzione" in un accezione più odierna è Slackware e il suo sviluppo è ancora attivo oggi.

Le famiglie, ovvero i "Padri" delle distribuzioni linux che differiscono sostanzialmente in specifiche ed implementazioni e vengono ereditate da altri progetti figli sono "Debian", "Slackware", "Red Hat" e la più giovane (meta-distribuzione) "Gentoo". Queste differiscono molto tra di loro, introducendo decisioni, regole e politiche del Sistema Operativo nel loro corso di sviluppo [6].

Come Debian è stata d'ispirazione per Ubuntu, Slackware è padre di Arch Linux e openSUSE che sono tra le distribuzioni più popolari, nello stesso modo RedHat cura Fedora per la versione community della loro variante Enterprise [6].

La Red Hat, la Canonical e la Novell, che rappresentano le corrispondenti e le attuali distribuzioni Fedora, Ubuntu e SUSE sono aziende che hanno deciso di investire in Linux in ambito **Business2Business**, provvedendo al supporto informatico e all'assistenza per altre aziende [6].

La Red Hat, fu la prima azienda a fornire soluzioni Open Source in borsa (1999), iniziò nel 1994 come assistenza per i *consumer*, muovendosi pian piano nel settore **B2B**, ad oggi continua a detenere il primato per contribuzioni all'ecosistema Linux [7].

Nel 2003, Andy Rubin inizia a sviluppare il Sistema Operativo Android, basato su kernel Linux, fondando l'azienda Android Inc. per poi essere acquisita da Google nel 2005 ed essere rilasciato nel 2007 con licenza Open Source (Apache 2.0)[8].

Android, a differenza delle Distribuzioni GNU/Linux non si correda del parco software GNU ma bensì da una suite applicativa basata su una filosofia diversa: focalizza l'interesse verso i dispositivi mobili e applicativi sviluppati da terzi, separando così il compito del Sistema Operativo a quello del software installabile. Ed è grazie ad Android che il Kernel Linux è arrivato anche nel mondo del mercato globale, allargando la propria porzione di utenti.

La diretta conseguenza, considerando le vendite e la predominanza del mercato mobile, porta Linux ad essere il kernel più utilizzato tra i dispositivi consumer, e dunque il più diffuso al mondo [9] [10].

Google ed altre aziende grazie ad Android hanno da allora contribuito al kernel Linux sia assumendo sviluppatori o anche per via monetaria. In questo modo si è potuto rendere il kernel tra i progetti più attivi al mondo, più maturo e ottimizzato per la maggior parte dei dispositivi.

Un caso degno di nota è Google, che utilizza la Distribuzione Gentoo Linux come base per alcuni dei suoi prodotti commerciali, ovvero i notebook basati su ChromeOS e il router OnHub.

ChromeOS è un sistema operativo Linux, e può essere definito distribuzione in quanto contiene in parte software GNU. ChromeOS nasce da un progetto OpenSource chiamato ChromiumOS dove Google tramite un apposito team, personalizza il sistema per poterlo rendere disponibile nei dispositivi chiamati Chromebook [11], osservabile nella Figura 1.1a. I Chromebook hanno anche raggiunto una certa popolarità, superando le vendite dei portatili Apple negli Stati Uniti d'America [12].

OnHub è un router WiFi di ultima generazione, corredato da un applicazione per dispositivi Android e iOS che permette di gestire e controllare la propria rete di casa, Figura 1.1b.

Il tutto ci porta ad un ecosistema tecnologico interamente nuovo; i dispo-

(a) Chromebook Pexel 2, il notebook di Google

(b) OnHub, il router wireless “smart” di Google.

Figura 1.1: Il sistema operativo dei Chromebook e di OnHub è Gentoo Linux. Fonti immagini: *https://pixel.google.com/* e *https://on.google.com/hub/*

sitivi, ad esempio l’Internet Of Things è basato completamente sull’utilizzo del kernel Linux; dove distribuzioni come Debian Fedora e Ubuntu hanno il predominio, incoraggiando anche la redistribuzione con modifiche. Il panorama *IoT* è estremamente frammentato, ma ha un lato in comune, appunto, Il kernel.

L’importanza della comunità - La cattedrale e il Bazaar

Il ruolo della comunità nei progetti Open Source è fondamentale. Lo sviluppo “distribuito” su più persone è stata la chiave per il successo dell’Open Source. Si possono infatti distinguere due modelli di sviluppo software: il modello a “Cattedrale” e il modello a “Bazaar”. Il modello detto a “Cattedrale”, rappresenta il modello predominante di sviluppo che regnava prima degli anni 90, che prevedeva un gruppo di programmatori *elitaristi* e isolati dal mondo, che rilasciava il software solo quando era pronto e perfettamente testato. Con l’avvento dell’Open Source, lo sviluppo si è allargato a tutte le persone, permettendo così cicli di rilascio più veloci [13]. Qui sono stati introdotti anche nuovi modelli di sviluppo, come *feature backporting* o i sistemi

di branching basati su *hotfixes*[2].

"Rilasciare presto e spesso". Questa è stata la chiave per poter permettere a qualsiasi sviluppatore di contribuire ad un qualsiasi progetto a qualsiasi stadio; attirando anche persone *realmente* interessate [3].

1.2 Cloud computing

Cloud computing è la tecnologia più utilizzata per fornire i servizi in larga scala su internet. Il Cloud computing è un modello che permette agli utenti di accedere sempre, a richiesta e in maniera conveniente a dei servizi computazionali condivisi con altre persone, tutto ciò può essere erogato velocemente con il minimo sforzo dal fornitore [14]. La radice del Cloud Computing la possiamo trovare con le metodologie *DevOps*, una cultura che ne sfrutta pienamente le potenzialità introdotte.

Ed è proprio in questo contesto che è interessante trovare un connubio tra Open Source e meccanismi *DevOps*.

Con *DevOps* si intende: una figura professionale, una cultura, un movimento o una pratica che enfatizza la collaborazione e la comunicazione con i software developer e i professionisti nel settore IT, automatizzando il processo di rilascio del software e di cambio di infrastruttura. [15]. È l'insieme di pratiche per il quale si permette la fase di *building*, *testing* e rilascio del software in maniera rapida, efficiente e affidabile. Perfettamente in armonia con i meccanismi delle Distribuzioni Linux Open Source.

[2] Sono modalità di sviluppo agili, dove viene agevolato lo sviluppo continuo, in cui patch a bug incontrati in uno stato successivo vengono riadattate o semplicemente riutilizzate (se coerenti) con versioni antecedenti.

[3] In questo modo, le persone veramente interessate ad un progetto o semplicemente ad una parte di esso sono più spronate a contribuire. Ad esempio, il processo di sviluppo *upstream* del Kernel Linux è impressionante, i tassi di cambiamento per il kernel 4.3 sono stati all'incirca di 8 patch per ora, da 1600 sviluppatori, per un totale di 12,131 cambiamenti in 63 giorni. - *https://lwn.net/Articles/661978/*

1.3 Ciclo di sviluppo di un sistema operativo

L'estrema eterogeneità delle distribuzioni e Sistemi Operativi Linux ha causato una eccessiva frammentizzazione delle soluzioni adibite allo sviluppo. Il ciclo di sviluppo di un sistema operativo può comunque essere suddiviso in 3 macro-aree: *Kernel*, *Gestione del Parco Software* e *il sistema di Release*. Ognuna di esse è composta da più team che collaborano, indifferentemente dalla struttura gerarchica, e assolvono a compiti diametralmente opposti che necessitano comunque di comunicare per lavorare in armonia.

1.3.1 Gestione del parco Software

Parte del Software deve essere fornito all'utente sia alla prima installazione (ci sono componenti critici che obbligatoriamente saranno presenti per garantirne un corretto funzionamento) ma deve essere anche disponibile per l'installazione e l'aggiornamento in un secondo momento.

Gestore dei pacchetti Le distribuzioni Linux vengono comunemente classificate in base al gestore di pacchetto in uso. Questo è necessario affinchè si possano differenziare le caratteristiche e il ciclo di release[4] di un sistema. I Gestori di pacchetti differiscono anche per strategia di risoluzione delle dipendenze e grado di personalizzazione del sistema [3].

Lo sviluppatore, avrà la responsabilità di fornire definizione di pacchetti aggiornati, funzionanti (a fronte di una fase di testing, dove spesso viene tipicamente demandata ad un altro team) e nella maggioranza dei casi anche

[4] I cicli di release sono differenti in base alla strategia di vita del Sistema Operativo. Esistono i sistemi versionificati, in cui ogni ramo(versione) ha un suo ciclo di vita, e raggiunto quello può essere definito deprecato, e in quel caso vengono forniti strumenti per aggiornare(cambiare)il ramo, oppure sistemi Rolling Release, dove non ci sono rami e l'aggiornamento di un sistema ne comporta l'aggiornamento allo stato pubblicato dal fornitore

pre-compilati. Ogni Gestore di Pacchetti solitamente definisce anche il flusso di compilazione del pacchetto, così da garantire **riproducibilità**[5].

Distribuzioni pre-compilate Nel caso un sistema sia pre-compilato, lo sviluppatore si occupa anche delle fasi di distribuzione dei pacchetti. Questo passaggio richiede un' infrastruttura *adhoc* per la generazione dei pacchetti e un'infrastruttura per renderli pubblicamente accessibili (*e.g. infrastrutture CDN, mirrors,...*) [3].

Le sfide Ci sono varie sfide nella gestione del software: Uno studio recente di alcuni ricercatori dell'Università dell'Arizona ha evidenziato che meno del 50% del Software potrebbe essere ricompilato [16] o installato e risultati simili sono stati ottenuti in altri studi [17]; l'ambiente di compilazione è anche un punto fondamentale nelle piattaforme dedicate alla compilazione automatica, come lo è anche risolvere le dipendenze e aggirare la *"Dependency hell"*, infatti alcuni gestori di pacchetti utilizzano tecniche di AI (Artificial Intelligence) per la risoluzione delle dipendenze, poichè il problema è esprimibile in termine di soddisfacibilità booleana (SAT)[6] . Anche lo stato della documentazione dei vari software (se presente) spesso è una vera e propria barriera. Inoltre gli eseguibili dinamici, che fanno affidamento ad altre librerie, generalmente vengono aggiornati per aggiungere funzionalità o risolvere errori dai maintainers delle distribuzioni - in questo caso, viene cambiato il codice generato finale del software, il che porta ad una situazione in cui

[5]Dipendentemente da questo varia la flessibilità della distribuzione. Gentoo ad esempio è una meta-distribuzione poichè è completamente personalizzabile. Le specifiche di compilazione e i dettagli del sistema **devono** essere specificate dall'utente. Il gestore dei pacchetti in questo caso compila ed installa le richieste dell'utente sulla base delle scelte fatte per dar luogo ad uno stato del sistema completamente differente anche con gli stessi software installati

[6]Ad esempio, è stato rilasciato il software **apt-pbo** per ottimizzare il calcolo delle dipendenze richieste, codificando il problema della risoluzione delle dipendenze in termini di pseudo-ottimizzazione booleana [18]

spesso alcuni errori non sono riproducibili[7]. Infine, introdurre un software di gestione dello stato del sistema introduce anche un nuovo vettore di attacco [19], richiedendo anche ulteriori studi di sicurezza anche durante le fasi implementative.

1.3.2 Il sistema di rilascio

Il sistema di rilascio si occupa della generazione degli artefatti che saranno i mezzi di installazione della Distribuzione Linux in oggetto (*e.g. LiveCD, Immagini per dispositivi embedded, BOOT PXE,...*). Questi procedimenti richiedono un'infrastruttura e quindi uno o più team per la manutenzione e/o sviluppo.

Del sistema di rilascio fa parte anche l'infrastruttura, che tramite le specifiche decise dalla comunità e/o dal relativo organo decisionale, permettono il rilascio di immagini installabili per diversi supporti.

L'infrastruttura solitamente è molto eterogenea e differisce sia per l'approccio alla Gestione del parco Software[8] sia per le politiche interne dell'organizzazione, senza escludere decisioni tecniche necessarie per la ridistribuzione.

1.3.3 Il Kernel

Può nascere l'esigenza di modificare e redistribuire il proprio Kernel per vari scopi (configurazioni specifiche, supporto a nuovi dispositivi, sicurezza, ecc...) comunemente le Distribuzioni Linux dispongono di un team che si occupa delle modifiche e della manutenzione del kernel.

Possiamo identificare dei problemi comuni, ovvero l'eterogeneità delle infrastrutture *adhoc* realizzate per la gestione della compilazione dei pacchetti e la gestione del rilascio, poiché il Kernel viene principalmente fornito tramite il parco software della Distribuzione.

[7] questa situazione viene chiamata *"code rot"*
[8] e dal formato del pacchetto installabile

1.4 Scopo del Lavoro

Tutto ciò porta ad una naturale richiesta: un efficace sistema di distribuzione e creazione di sistemi operativi che sia al passo con l'eterogeneità dei dispositivi in circolazione e sulla loro diversa architettura. Verranno utilizzate tecnologie Open Source moderne, con gli elementi del campo DevOps che saranno la chiave del meccanismo.

Le tecnologie di containerizzazione quali *LXC*, *LXD* e *Docker* offrono notevoli vantaggi nel campo della compilazione e testing. Il poter disporre facilmente di *sandboxing systems* ci permetterà di realizzare *whitebox testing* on-demand nel campo della Gestione del Software e realizzazione di Sistemi Operativi anche in ambienti *PaaS* e *IaaS*, campo fin'ora poco esplorato nella letteratura.

Introdurre pratiche *DevOps* nei contesti discussi fin'ora è un campo del tutto nuovo – miglioreremo così la **Quality Assurance** [20] in ogni fase operativa tramite la rilevazione di errori e report direttamente allo sviluppatore interessato [21].

L'aspetto interessante è anche che introdurremo la *Computational reproducibility* con le pratiche *DevOps*, già affrontato in letteratura scientifica parlando di esperimenti scientifici riproducibili tramite *Docker* [22], esteso ora nel contesto della Gestione dei Pacchetti e sviluppo di Sistemi Operativi; questo include la realizzazione delle immagini di rilascio, le fase di sviluppo dei pacchetti e quindi del sistema operativo stesso[9]. Definiremo quindi in termini di entità Cloud l'infrastruttura adibita per la gestione della pacchettizzazione e del rilascio di una Distribuzione Linux, permettendone quindi una replicazione delle componenti chiavi in maniera immediata in qualsiasi ambiente (anche non Cloud).

[9]La fase di realizzazione delle immagini di rilascio e quella dello sviluppo del Sistema Operativo spesso combaciano, dagli stessi *codebase* deriva lo stesso risultato convertito in formati differenti

1.4.1 Struttura della tesi

Vedremo le soluzioni esistenti fornite dalle diverse famiglie di Distribuzioni Linux, ci concentreremo soprattutto su Gentoo sia per la sua natura "dinamica" sia per l'estrema versatilità e anche perchè possiamo trovarne un applicazione in contesto Enterprise. Analizzeremo anche realtà aziendali dove parti di questo modello sono state applicate nel momento della redazione di questa tesi, con software differenti, ma con scopi affini.

Estenderemo il concetto della costruzione di sistemi operativi all'ambito Cloud tramite strumenti appositi e proporremo soluzioni per il mantenimento e la realizzazione indipendentemente dal contesto applicativo e dal contesto d'uso[10].

Le soluzioni proposte svincoleranno lo sviluppatore dall'hardware e dal sistema in uso fornendo strumenti di *testing* e *continous integration* in relazione alla gestione dei rilasci e manutenzione del Software. Verranno proposti metodi per la redazione di specifiche (*Dockerfiles*) degli artefatti destinati al rilascio tramite software già esistenti ed utilizzati in ambito Cloud, ammortizzando anche i costi di mantenimento dell'infrastruttura e manutenzione. Lo sviluppatore così non dovrà nemmeno necessitare dell'hardware che si vorrà produrre e distribuire per la realizzazione delle definizioni. Porremo una separazione netta tra fase di sviluppo e fase di testing, fornendo strumenti Cloud per entrambe isolandole ancor più nettamente, ma mettendole in comunicazione tra loro.

[10]Le soluzioni proposte possono anche essere utilizzate localmente, aprendo la possibilità alla costruzione di una propria infrastruttura nelle situazioni necessarie. *e.g. quando le architetture non sono supportate*

Capitolo 2

Analisi dei sistemi esistenti

Andremo ora ad analizzare le soluzioni che sono state sviluppate negli anni dalle principali Distribuzioni Linux nel panorama Open Source: vedremo i sistemi di pacchettizzazione e release management di Fedora, Debian, Gentoo e Sabayon. Ci focalizzeremo sull'ecosistema Gentoo, che è quello di riferimento per la Distribuzione Linux Sabayon, che è stato lo *use-case* dove possiamo osservare la tecnologia descritta applicata nel ciclo produttivo. Inoltre, ci riferiremo a Gentoo perchè costituisce un caso reale in cui una Distribuzione Linux venga utilizzata in prodotti disponibili sul mercato a vasta scala.

Vedremo ora i tratti distintivi delle maggiori Distribuzioni Linux e come vengono gestiti il Parco Software e le Releases.

2.1 Distribuzioni Linux

I software che sono stati sviluppati dalle Distribuzioni in analisi sono frutto di una collaborazione tra diversi sviluppatori nel mondo [3]. Ogni metodologia ivi esposta è frutto di una collaborazione tra diversi individui sparsi nel globo, *Bazaar*, diametralmente opposta a quella a *Cattedrale* che prevede un gruppo unico di persone addette ai lavori, e nessun ciclo di release contemplante versioni beta o instabili, di cui abbiamo già discusso nel

capitolo precedente. La massimizzazione di questo concetto l'avremmo in un contesto Cloud, di cui parleremo in seguito, dove tutti, indipendentemente dalla piattaforma software e dalla tecnologia utilizzata saranno in grado di contribuire abbattendo ulteriormente le difficoltà degli sviluppatori alle prime armi. Questo permette anche di fornire ambienti completamente isolati dove poter sperimentare [1] prima ancora di arrivare allo stage *production* [2].

2.1.1 Fedora Linux

Fedora è una distribuzione GNU/Linux, un progetto Open Source sponsorizzato dalla Red Hat e supportato dalla community [23].

Utilizzabile pressochè in qualsiasi ambito: server, uso domestico o come workstation. È distribuita su base fissa semestrale con le versioni più recenti di ogni pacchetto, compreso il kernel [23].

Il nome si deve al cappello in feltro simbolo di Red Hat, il borsalino, chiamato in alcuni paesi Fedora [24].

Il gestore dei pacchetti è DNF e permette di installare software aggiuntivo in formato binario (pre-compilato), il formato è RPM [24].

2.1.2 Debian Linux

Distribuzione con notevole importanza storica, perno della filosofia GNU/-Linux ed è la distribuzione più popolare, Ubuntu infatti è una derivata di Debian. Debian ha una community molto attiva e ben definita, suddivisa in organi decisionali anche per giudizi interni alla comunità stessa [25].

Utilizza il gestore dei pacchetti chiamato apt-get e il suo formato file è DEB. Anche in questo caso, è una distribuzione che offre binari precompilati [26].

[1] Ambienti *Playground*, completamente *sandboxed*

[2] Quando ci riferiremo a *production*, viene intesa la tecnologia o il servizio che è reso disponibile al pubblico.

2.1.3 Gentoo Linux

Gentoo Linux viene definita una meta-distribuzione [27] e infatti per la sua flessibilità viene utilizzata anche in prodotti sul mercato. Per la sua complessità nel sistema comunitario, di modi di contribuire e distribuzioni dei compiti che sono così differenti ed estremamente specializzati, Gentoo viene analizzato in letteratura anche in termini di sistemi complessi [28]. La peculiarità di Gentoo è che permette di installare qualsiasi pacchetto da sorgente, indicandone la combinazione di USE flags, che definiscono le componenti abilitabili/disabilitabili dei software. Questo ci permette di creare la combinazione di sistema che preferiamo di più, con il massimo livello di personalizzazione ed una estrema possibilità di ottimizzazione [27]. Una delle caratteristiche principali è anche l'alto livello di **QA** (*Quality Assurance*) presente, grazie ad un'efficace gestione e meccanismi di cattura degli errori integrati nelle *eclasses* che prevengono e aiutano ad identificare i problemi di compilazione o le corruzioni di sistema.

Esempio: Una macchina adibita ad uso Server può rimuovere tutte le flag di compilazioni che sono necessarie per un contesto grafico. Questo non solo snellisce la dimensione del sistema finale, ma ne permette comunque una riduzione della superficie di impatto di possibili falle di sicurezza.

Sabayon Linux

Sabayon Linux è una distribuzione basata su Gentoo, ma ne fornisce la variante in formato precompilato e quindi binario. Principalmente è un profilo di configurazione di Gentoo che permette un funzionamento generico su sistemi basati su un architettura amd64 o armv7l, rilasciata di recente [29]. Eredita la flessibilità di Gentoo ma sollevando la responsabilità dell'utente di effettuare le ottimizzazioni per il proprio sistema; in alternativa l'utente può sempre comunque utilizzare la macchina come un normale sistema Gentoo. Introduce il gestore di pacchetti Entropy, che è un framework che si interfaccia a *Portage*. La metodologia e le tecnologie di cui discuteremo sono

implementate attualmente nei cicli di release e compilazione dei pacchetti di Sabayon Linux.

2.2 Gestione del Parco Software

Garantire la possibilità all'utente di installare pacchetti tramite linea di comando, e quindi accedere ai software in maniera comoda e pratica è il core di ogni Distribuzione Linux. Personalizzazioni, regole e distribuzioni di configurazioni sono gestite dal package manager per garantire la solidità e l'integrità del sistema.

2.2.1 Vantaggi offerti dalla pacchettizzazione del software

La pacchettizzazione offre dei lati molto vantaggiosi sia per l'utente che per i *maintainers* delle Distribuzioni. La pacchettizzazione è l'operazione di archiviazione dell'artefatto prodotto dal processo di compilazione del sorgente Software. Nella fase di compilazione, vengono definite le specifiche e gli aspetti che si vogliono abilitare e disabilitare del software. In questa fase, le librerie che vengono richieste dal software vengono soddisfatte da quelle presenti dal sistema, collegandole direttamente (*shared libraries*, dinamicamente), così facendo, in *runtime*, avremo prestazioni migliori, poichè i software che utilizzeranno le stesse librerie, utilizzeranno le stesse risorse RAM indipendentemente dalle tecniche di ottimizzazione del Kernel Linux dei processi [30].

Legare i pacchetti intrinsecamente gli uni agli altri, permette anche di aumentare il livello di sicurezza globale di un sistema: in caso di falle di sicurezza in componenti critici, o librerie chiave per la comunicazione sicura (come ad esempio i recenti bug openssl e samba) gli aggiornamenti possono essere distribuiti celermente, aggiornando il singolo componente e (nella maggior parte dei casi) non tutti i software che utilizzano quella libreria.

Di recente, Ubuntu e Fedora hanno rilasciato "Snap" [31] e "FlatPak" [32] : sono delle applicazioni che comprimono i software in maniera universale, rendendoli utilizzabili anche tra diverse Distribuzioni Linux. È affine al lavoro di **AppImage** [33], il quale ha anche attirato l'attenzione e l'interesse di Linus Torvalds stesso [3].

Snap e FlatPack si differenziano piuttosto sull'implementazione e sulla gestione della sicurezza. FlatPak, a differenza di Snap, crea una sandbox dove l'applicativo esegue i comandi [32].

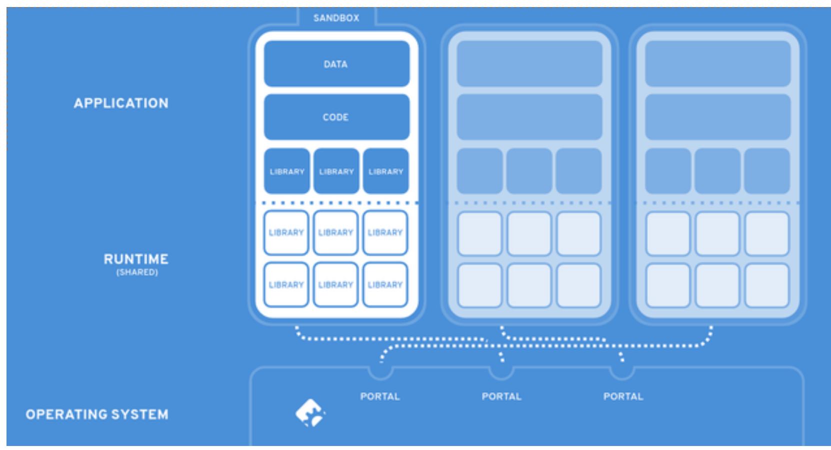

Figura 2.1: Struttura applicativi Flatpaks. Fonte immagine [34]

In entrambi i casi i pacchetti possono essere compilati staticamente o dinamicamente, nel secondo caso vengono fornite le shared libraries necessarie dentro il pacchetto. Questo sostanzialmente ci porta alla considerazione di due situazioni direttamente conseguenti: la prima, in contesto di sicurezza, il *maintainer* (ovvero lo sviluppatore che gestisce e mantiene la definizione di compilazione del software) di ogni pacchetto dovrà ora prendersi cura di aggiornare le componenti che hanno vulnerabilità, per ogni pacchetto che usa la libreria; la seconda è l'impatto sulle risorse: programmi che utilizzeranno la stessa libreria ma di diverse versioni (ma anche in generale) occuperan-

[3] https://plus.google.com/+LinusTorvalds/posts/WyrATKUnmrS

no rispettivamente diverse porzioni delle risorse che invece prima potevano condividere.

Tuttavia ci sono dei casi dove queste tecnologie avrebbero un predominante contesto d'uso, ovvero quello di software per Workstation o comunque ad uso casalingo (Giochi, CAD suites, ecc..). L'obbiettivo non è sostituire i componenti chiave della Distribuzione. In contesto Cloud, è un'ottimo strumento per isolare ancor più i componenti, permettendone comunque una grande scalabilità. La differenza tra Snap e Flatpack è la centralizzazione. Gli "Snaps", ovvero gli atomi installanti sono disponibili nelle repository di Ubuntu, chiamati in questo caso "stores" [31].

2.2.2 Gestione delle repository Fedora/RedHat

Fedora utilizza una utility da terminale per generare un database da distribuire agli utenti per poter accedere agli RPM inclusi nella repository. Vengono prima creati gli RPM, con degli appositi file di specifica in formato .ks che indicano come compilare il pacchetto. Dopodichè vengono raccolti i file in un apposita cartella, che viene poi data in input al programma "createrepo". Il risultato è un'altra directory che contiene sia i file in RPM ma anche un database che permette al client di indicizzare e risolvere le dipendenze correttamente, in accordo con il formato di repository leggibile da DNF [24].

COPR, è la centralizzazione della compilazione dei pacchetti automatizzata per la community. L'interfaccia è Web, dove si possono caricare le specifiche e operare anche solamente da browser.

L'ecosistema di Fedora è interessante, poichè incorpora anche una forte componente Business (derivata dalla RedHat) che ha richiesto sistemi sempre più avanzati per la distribuzione della compilazione dei pacchetti.

2.2.3 Gestione delle repository Debian: aptly, reprepro, dak, mini-dak

Debian, fornisce una moltitudine di tool per la gestione delle repository e dei pacchetti. Possiamo citare Aptly, che permette di gestire mirror remoti, pacchetti in una repository locale, fare snapshots, prendere le nuove versioni dei pacchetti e delle sue dipendenze ed infine anche di pubblicare la repository Debian.

Ci sono anche altri tool dediti alla stessa funzionalità, come reprepro, dak e mini-dak. Dak, acronimo per Debian Archive Kit è una collezione di tool designata per manutenere i pacchetti Debian e che permette di amministrare una repository oppure una grande collezione di pacchetti. Mini-Dak è una versione minimal, che fornisce un subset delle funzionalità, utilizzata per mantenere repository di modeste dimensioni.

2.2.4 Portage

Portage è un sistema di gestione di pacchetti, caratterizzato da una repository centrale "*ports collection*"[4] di specifiche di compilazione, che è letta, eseguita ed interpretata da *Portage*, che può essere utilizzato tramite *Emerge*, l'interfaccia al gestore di pacchetti di Gentoo. I pacchetti sono definiti tramite "*atomi*" nel seguente formato: `categoria/nomepacchetto`, vengono suddivisi quindi i software in base alla categoria di appartenenza (networking, text-editing,...). Portage può essere definito come l'insieme di file di specifica, scritte in un DSL (*Domain Specific Language*) costituito da gruppi di funzioni, definite *Eclass* che di fatto formano un dialetto del linguaggio bash, dedicato all'istruzione, alla configurazione e all'esecuzione delle varie fasi ne-

[4]La repository centrale può essere cambiata in qualsiasi momento da un sistema Gentoo, oppure non utilizzare quella centralizzata affatto: Ne è un esempio la distribuzione Funtoo, creata da Daniel Robbins (fonatore di Gentoo) che utilizza una port-collection separata. [28] [27]

cessarie per l'installazione del pacchetto (o sola compilazione) [5]. Portage viene utilizzato da ChromeOS, Sabayon e Funtoo.

2.2.5 Gestione delle repository Sabayon: Eit

Sabayon, usa Entropy anche come gestore di repository. Il Software a linea di comando è chiamato "Eit" ed è ispirato al funzionamento di Git. I pacchetti vengono compilati da una Tinderbox [6] tramite Portage, i cui artefatti vengono gestiti da *Eit*. Permette di gestire i mirror, deploy, injection, mirror sorting e news channels per le comunicazioni importanti agli utenti.

2.2.6 OBS

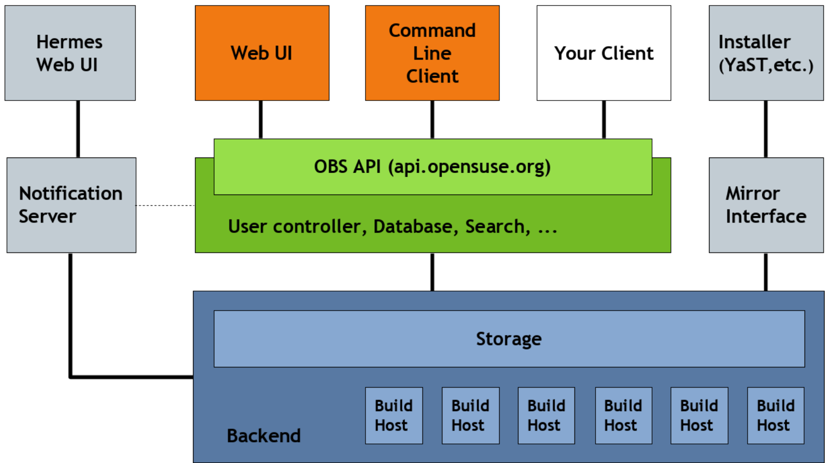

Figura 2.2: Struttura OBS: Il sistema è molto flessibile ed è formato da diversi componenti che interagiscono con il sistema di API. Fonte immagine [35].

[5] La compilazione dei pacchetti viene effetuata in una *sandbox*, per poi essere applicata al sistema. Questo garantirà che i sorgenti di diversi software rispettino la proprietà dei files di altri pacchetti, evitando qualsiasi tipo di collisione. Questo è necessario poichè, a differenza dei gestori di pacchetti binari, i specfile vengono eseguiti al lato client. [28] [27]

[6] Un ambiente di compilazione, a partire da una stage3

Viene utilizzato anche Open Build Service [36] come sistema di infrastruttura per la gestione del ciclo della Gestione Pacchetti e delle repository, si può osservare la struttura in figura 2.2.

Open Build Server è un sistema di compilazione e distribuzione dei pacchetti integrato, la sua modularità permette anche di estendere il supporto a più Distribuzioni Linux tramite specfile, tra cui: Debian, Fedora, Mandriva, Open Suse, Red Hat, Fedora e Scientific Linux. La peculiarità di OBS è l'in-

```
Name:            ctris
Summary:         Console based Tetris clone
URL:             http://www.hackl.dhs.org/ctris/
License:         GPL
Group:           Amusements/Games/Action/Arcade
Version:         0.42
Release:         1
Source:          %{name}-%{version}.tar.bz2
BuildRoot:       %{_tmppath}/%{name}-%{version}-build
BuildRequires:   ncurses-devel
Requires:        ncurses


%description
An ASCII version of the well known game Tetris

%prep
%setup -q

%build
make

%install
make install DESTDIR=$RPM_BUILD_ROOT
%debug_package

%clean
rm -rf $RPM_BUILD_ROOT;

%files
%defattr (-,root,root)
%doc AUTHORS COPYING README TODO
%doc %{_mandir}/man6/ctris.6.gz
/usr/games/ctris
```

Figura 2.3: OBS: Sintassi. Fonte immagine [35]

dipendenza dai vari package managers, che richiedono però appositi moduli di interfaccia per ognuno di essi. Per astrarre, le definizioni vengono scritte in appositi specfiles, di cui si può notare la sintassi molto leggibile, visibile in figura 2.3 [35]

2.2.7 Sintassi

Nei file di specifica, come quelli mostrati in figura 2.3 vengono riportati i comandi necessari per la compilazione, e il riempimento di alcuni metadati per l'indicizzazione, notiamo, vengono specificati i pacchetti che sono dipendenze *runtime* e *buildtime*, ovvero quelle che sono necessarie quando il programma è in esecuzione, e quelle invece necessarie per la sua compilazione. A differenza di un file di specifica di Gentoo, ad esempio, possiamo notare come non c'è un vero e proprio dialetto, ma dei campi da riempire per "indicizzare" la modalità di compilazione di un pacchetto, invocando direttamente i comandi necessari per la compilazione [35]. Questo spesso sfocia in una eterogeneità di file di specifica, che sono molto differenti tra di loro; *Portage* invece, fornisce una collezione di definizioni che formano un dialetto, dove ogni classe e funzione viene implementata a seconda del modello di building dei software in circolazione.

2.3 Gestione delle release

La fase della gestione delle release comprende il Cycle Management, che scansiona nel tempo le release e tutte le operazioni che intercorrono tra di esse, per arrivare al rilascio e alla sua generazione finale nei vari formati. Questi tool vengono accompagnati da un'infrastruttura che di solito è installata su una singola macchina, e quindi è un sistema prettamente centralizzato. Vengono utilizzati vari tool a seconda delle varie stages, ed esistono diversi software "orchestratori" delle varie fasi. Solitamente abbiamo dei specfile che vengono letti dal software e che permettono una personalizzazione della build. Questi software solitamente producono immagini masterizzabili su CD, scrivibili su pennette USB o SD card (in caso di piattaforme *ARM*), a volte vengono anche rilsaciate le *chroot*[7].

[7] le chroot sono ambienti vergini, snapshots del sistema prima di passare per la fase di conversione di formato (.ISO,.img, .tar.gz, ecc..)

2.3.1 SUSE studio

OpenSUSE è una distribuzione Linux che fornisce un servizio gratuito di costruzione della propria variante di openSUSE direttamente dal sito di SUSE studio[8]. Purtroppo il software è a pagamento[9], quindi non si è potuto valutare il *codebase*, l'implementazione architetturale e la struttura di funzionamento, ma con il *free tier* (che ha delle limitazioni) si è potuta solo valutare un'interfaccia semplice e facile da utilizzare.

2.3.2 Livecd-creator, kickstart

Il sistema utilizzato da Fedora per la creazione di LiveCD avviene tramite degli appositi file di specifica "kickstart" files che descrivono i pacchetti che vengono implementati nel sistema finale, che costruisce l'immagine installabile.

2.3.3 Debootstrap, live-build,live-wrapper

Debian ha un insieme di programmi dedicati alla modifica della chroot, partendo da una base che è pre-compilata dall'organizzazione. Con Debootstrap indichiamo anche i programmi derivati o affini come Multistrap cDebootstrap (scritto in C). In questa fase, a partire da un'immagine di partenza, attraverso dei file specifica, vengono effettuate le modifiche richieste. Questo stadio può essere personalizzato ulteriormente specificando degli *hook script*, in bash durante le varie fasi del processo. Questa operazione è applicata in differenti contesti:

- Durante l'installazione dall'immagine (es. CD-ROM) , che decomprime un ambiente base (incluso nei media installabili) sulla partizione che l'utente ha specificato. In base all'hardware in uso (questo compito è svolto tipicamente dal software dedito all'installazione - *installer*)

[8]https://susestudio.com/

[9]https://www.suse.com/products/susestudio

vengono effettuate delle operazioni di configurazione del sistema per ottimizzarlo sul dispositivo

- Durante la fase di creazione dell'ambiente che viene poi incluso nel media installabile

Non solo, queste fasi possono essere anche utilizzate per ulteriori re-mastering o *unattended install* - ovvero automazioni di installazioni su cluster di macchine.

2.3.4 Catalyst

Catalyst è un tool di release building completo, utilizzato da Gentoo Linux, creato inizialmente da Daniel Robbins [37]. Con Catalyst si possono configurare le installazioni di Gentoo completamente, permettendo di specificare anche i vari tool che sono utilizzati per l'installazione stessa. Alcune funzionalità di Catalyst sono:

- Costruire le stages di installazione[10]
- Costruire LiveCDs avviabili
- Costruire i set Gentoo Reference Platforms (GRP)
- Configurare una *chroot* come ambiente di *building test*
- Costruire immagini netboot

Catalyst (e anche Metro) lavora con dei file di specifica che permettono di esprimere le operazioni da effettuare su una *seed* image da trasformare. Analizziamo un esempio, per permettere di modificare un'immagine qualsiasi:

[10] Gentoo è diviso in 3 stages: la stage3 è quella installabile dall'utente, che contiene un sistema con solo i componenti necessari ad utilizzare il gestore di pacchetti, emerge. Gli stage 1 e 2 sono creati da *building toolchains*: nella stage 1 vengono compilati i *core components* come gcc e busybox, alla stage 2 vengono compilati emerge e le sue dipendenze.

- Si posiziona l'archivio dell'immagine seed[11] in una directory temporanea, che può essere specificata nel file di configurazione */etc/catalyst.conf*
- È necessario specificare qual'è la "Portage collection" che vogliamo utilizzare: questa può essere presa dal sistema in uso, oppure la si può scaricare dai mirror di Gentoo
- Viene definito quindi un file di specifica con le opzioni per la trasformazione dell'immagine.

Un file di specifica prende la forma visibile nel listato 2.1.

```
1 subarch: amd64
2 target: stage1
3 version_stamp: 2015.04
4 rel_type: default
5 profile: default/linux/amd64/13.0
6 snapshot: 2015.04
7 source_subpath: default/stage3-amd64-latest
8 update_seed: yes
9 update_seed_command: --update --deep @world
```

Listing 2.1: Catalyst, configurazione di esempio

Le opzioni sono visibili in tabella 2.1 e quelle che ci interessano maggiormente per questo studio sono: "target" e "rel_type". Sono le due configurazioni responsabili di specificare rispettivamente che tipo di immagine è quella in ingresso e che tipo di processo deve essere eseguito per quella in uscita.

2.3.5 Metro

Funtoo, il progetto creato e fondato da Daniel Robbins, utilizza un'infrastruttura automatizzata per il building, chiamata *Metro* [38]. Metro può assolvere i stessi compiti di Catalyst. In questi software si deve considerare che tutto ha inizio con un immagine sorgente, chiamata *seed* stage, questa

[11] che può essere presa direttamente dai mirror Gentoo

Tabella 2.1: Opzioni di Catalyst

subarch	specifica l'architettura, deve essere scelta tra quelle supportate (visibili in {/usr/lib/catalyst/arch/*.py})
target	specifica il tipo di immagine target che Catalyst costruirà.
version_stamp	è un'identificatore per la build
rel_type	definisce quale tipo di sistema di building Catalyst eseguirà: questa è l'opzione che definisce ulteriormente le modalità di composizione dell'immagine risultante
profile	specifica il profilo[12] Gentoo da utilizzare.
snapshot	definisce lo snapshot di portage da utilizzare[13].
source_subpath	definisce qual'è la path dell'immagine da utilizzare
distcc_hosts	si possono specificare host distcc per compilare utilizzando anche altri nodi
portage_confdir	permette di specificare una cartella dove sono presenti i file di configurazione relative alle specifiche di compilazione (USEflags, Configurazioni del compilatore, ecc..)
portage_overlay	specifica una cartella aggiuntiva che può essere utilizzata per leggere le specifiche di compilazione

è utilizzata come ambiente di compilazione, per creare le altre *stages*. Da questa poi, tramite appositi specfile, possiamo generare un'altra stage 3, e da questa genererare le stage precedenti. Ogni stage che viene costruito può essere utilizzato come *seed* stage per un'altra trasformazione. Nel caso di Metro, il focus è sulla *riproducibilità*, permettendo di specificare con esattezza le configurazioni di conversioni di stage. Il risultato è che così le fase di building sono rese consistenti. Questo significa che la combinazione delle varie specifiche portano sempre allo stesso risultato.

2.3.6 Molecules

Molecules, è il software utilizzato da Sabayon sviluppato da Fabio Erculiani per convertire le *seed* stages, che solitamente sono derivate dalle *stage 3* di Gentoo, e che vengono modificate manualmente. Molecules converte le *chroots* tramite appositi specfiles definiti dagli sviluppatori, permettendo di specificare *hook* scripts che intervengono durante le varie fasi. Vengono eseguiti degli step di installazione/rimozione pacchetti, permettendo di specificare anche il Backend (Emerge o Entropy) di esecuzione. Poi il risultato viene convertito in un Immagine CD/USB/SD installabile o la stage risultante in forma compressa. Per le Immagini CD è anche presente il supporto UEFI.

2.4 Android

Android si distingue da questi sistemi, poichè non possiede un vero e proprio gestore di pacchetti in grado di gestire e aggiornare i componenti di sistema in maniera atomica e quindi strettamente collegati tra loro (dipendenze, dipendenze inverse, ecc.). Per generare un'immagine, e scriverla poi sui smartphones, tablet o dispositivi supportati da Android tramite l'utility *fastboot*, vengono scritte *buildchains*[14] e toolkit appositi. L'intero procedimento è riproducibile, ma a differeza delle distribuzioni GNU/Linux che sono corredate da un parco software GNU, non ha bisogno di un'infrastruttura per la distribuzione dei software a livello di sistema, gli upgrades (dei core packages) sono frutto di ri-scrittura delle partizioni di sistema da parte dell'utente (anche in maniera del tutto trasparente). I componenti interni, si sceglie di aggiornarli tramite la sovrascrittura dell'intero sottostrato di sistema, lasciando così integri i dati dell'utente.

[14] Ovvero vere e proprie catene di compilazione, inclusi tool per la cross-compilazione per le piattaforme che si vogliono supportare

Capitolo 3

Tecnologie utilizzate e scelte implementative

Illustreremo ora le tecnologie che vengono utilizzate per le soluzioni proposte nei capitoli successivi. I tool presentati sono utilizzati nel panorama delle tecnologie *DevOps* e sono disponibili nelle piattaforme di hosting Cloud.

In questo contesto è importante l'isolazione e la riproducibilità.
Lo sviluppatore che vuole mettere in produzione un servizio, genera un file di specifica che descrive *come* il suo applicativo fornisce il servizio, che viene anche utilizzato durante le fasi di testing. Il compito delle pratiche *DevOps* è quindi rendere scalabile e orchestrare l'infrastruttura, permettendo così un alta flessibilità e agilità nel rilascio di nuove versioni e al tempo stesso anche fasi di testing più efficaci [21]. Andremo quindi a presentare i software utilizzati in questo campo che sono stati utilizzati, di fatto introducendo la gestione dell'intero ciclo di rilascio di una Distribuzione Linux con metodologie/tecnologie *DevOps*.

3.1 Motivazioni sull'utilizzo delle metodologie DevOps

La gestione delle repository e del rilascio di un sistema operativo è una *pipeline* di tecnologie e software molto eterogenei che devono coesistere tra di loro. Da anni l'intero ecosistema, proprio a causa della sua eterogeneità viene governato e rilasciato principalmente per mezzo di script in Bash o in Perl, proprio perchè non c'è un'esigenza effettiva per software *adhoc*, ma bensì sono prassi che tendono ad automatizzarsi in un processo produttivo, man mano che questo prende forma più definita. Questo porta quindi ad una sovrapposizione, ripetizione e ridondanza di logiche, e spesso anche a *spaghetti code*. Nel nostro approccio non possiamo rimuovere la componente di scripting, critica anche nel mondo *IT* e nel campo *SRE*. Bensì isoleremo i componenti e utilizzeremo metodologie/tecnologie *DevOps* per velocizzare i processi e isolare i componenti, diminuendo la necessità di scrivere istruzioni o logiche non-riutilizzabili evitando *spaghetti code*. Spesso in questo trattato, vedremo come saranno scelte soluzioni che richiedono il minor intervento umano possibile, e sarà anche la motivazione per la quale vedremo che ridurremo il procedimento alla definizione di file di specifica, e alla gestione di questi.

3.2 Docker

Docker è una piattaforma libera per costruire, avviare, e pubblicare immagini di sistemi e applicazioni. Docker permette di generare *container* dalle immagini. I Container, a differenza dei sistemi con macchine virtuali, condividono lo stesso kernel tra di loro e con quello del sistema che li ospita. Questo permette un'abbattimento del costo delle risorse nel sistema, ma ha come lato negativo una minor isolazione, con eventuali conseguenze al livello di sicurezza rispetto all'utilizzo delle Virtual Machines [39] [40].

Virtualizzazione La virtualizzazione, in termini informatici, si riferisce all'atto di creare versioni virtuali di un'entità (invece di quelle attuali), questo include e non è limitato a piattaforme hardware per computer, sistemi operativi, dispositivi di immagazzinamento dati, o di risorse di rete. Docker offre un nuovo tipo di virtualizzazione.

Le immagini possono essere paragonate a livelli di filesystems collegati tra loro, che possono essere visti come un'entità unica da un *Container* , con tutte le componenti necessarie per il suo avvio. Docker introduce un file di specifica chiamato *Dockerfile*, che descrive la composizione dell'immagine, e tramite appositi comandi è possibile operare su di essa. [39] [41]

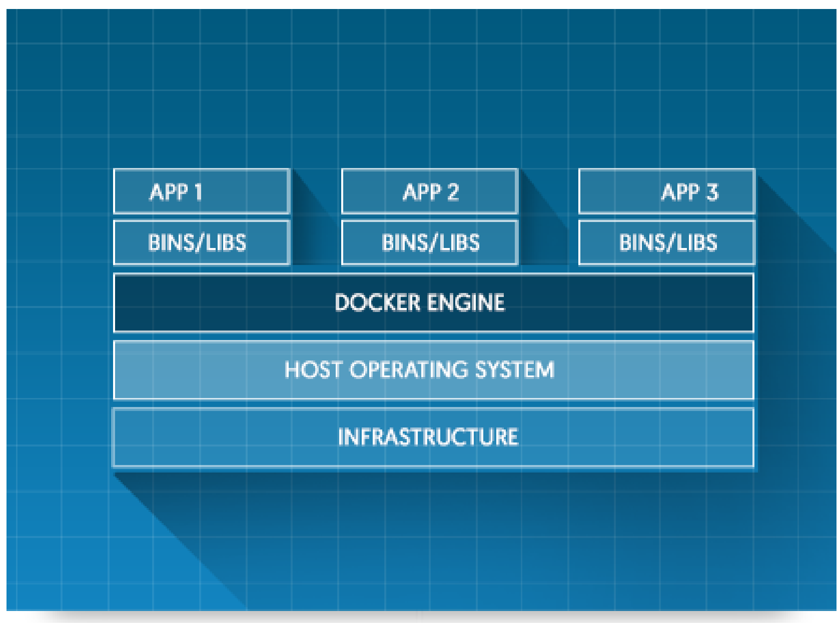

Figura 3.1: Docker System structure - Fonte immagine: *https://www.docker.com/what-docker*

Docker quindi permette di creare *Containers* basati sulle *Immagini*, che ospiteranno il nostro servizio. In questa maniera il Container eredita i layer del filesystem di base, ma può operare su un livello differente. [39] [41]

Vengono utilizzati diversi backend per comunicare con il kernel e allocare le risorse, tra cui : libcontainer(ora runc), lxc, libvirtd, systemd. Questi allocano le risorse necessarie e garantiscono l'isolamento dal sistema vero e proprio, utilizzando chiamate interne per gestirle. Per quanto riguarda la sicurezza, *policies* Seccomp e Cgroups possono essere specificate tramite Docker per un maggior controllo sui permessi dei Container.

Analogamente a come abbiamo visto nel capitolo precedente le *seed stage* di Gentoo e Funtoo, possiamo comparare le immagini Docker come *layers* sovrapposti l'uno all'altra, estendendo ancor più il concetto introdotto da Gentoo. I layer sono gestiti nativamente, e ogni immagine può avere come base una qualsiasi altra.

3.2.1 Riproducibilità

L'aspetto fondamentale introdotto da Docker è la riproducibilità. Ogni istruzione del file di specifica viene seguita in un filesystem differente, ed ognuno di questi viene dopo collegato l'uno con l'altro. Questo ci garantisce isolamento e la capacità di usare porzioni dell'immagine. Il container viene avviato al di sopra del filesystem che è l'unione dei layer formati tramite le istruzioni nel Dockerfile, oppure in alternativa è possibile collegare o aggiungere layers manualmente. In figura 3.2 possiamo osservare come sono organizzati in layer i filesystem AUFS, e come ogni strato è collegato l'uno con l'altro, per essere visto dal container come uno intero. Vedremo come questo aspetto ci permetterà di applicare specifiche di file più rigorose per dichiarare le nostre immagini che andranno installate su reale HW fisico.

Filesystem in production Docker permette di scegliere il tipo di filesystem per la propria strategia di deploy, la struttura consigliata è *lvm-direct*, ovvero partizionare un *drive* storage in una struttura lvm che replica quella che utilizza Docker internamente, affidandogli un *pool*, dove può formare autonomamente partizioni. In questo modo Docker è *mappato* nativamente

sul Drive dell'Host e ottiene la massima stabilità, condizione necessaria per essere utilizzata in ambito production.

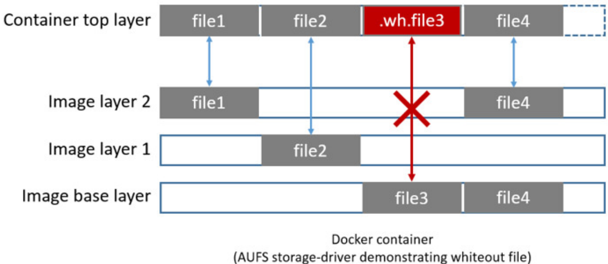

Figura 3.2: AUFS: Esempio dei filesystem a layer - *https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/*

Un Layer di un'immagine Docker è implementato direttamente come un Layer a livello filesystem. Questo viene effettuato dal driver del filesystem contenuto in Docker, che mappa le unità direttamente, permettendo un'estrema ottimizzazione degli accessi I/O. Un generico container in esecuzione avrà il secondo Layer (quello sottostante) ereditato interamente dall'immagine, mentre avrà semplicemente uno strato superiore che è uno spazio virtuale per i file che vengono generati dall'applicazione corrente. Possiamo osservare questa correlazione, in maniera più diretta con *overlayFS* in figura 3.3. Al livello implementativo differenti strategie di mapping sono poi utilizzate in base al Filesystem sottostante e al driver.

I file di specifica (*Dockerfiles*) sono tra l'altro supportati anche dallo standard ACI (App Container Specification) e quindi sono anche avviabili da software quali `rkt`, e in un ecosistema OpenStack praticamente in maniera nativa. I file di specifica Docker sono i *de-facto* standard di specifica di *containerizzazione*.

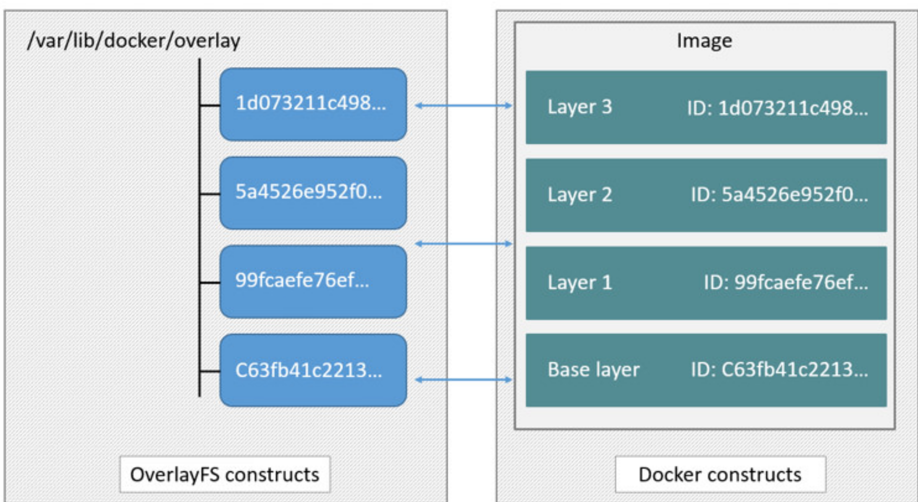

Figura 3.3: OverlayFS: Esempio dei filesystem a layer - Fonte immagine: *https://docs.docker.com/v1.9/engine/userguide/storagedriver/*

3.3 Vagrant

Vagrant, sviluppato da HashiCorp, permette di creare, configurare ambienti definiti da Virtual Machines. Questo permette di dichiarare gli ambienti in appositi file di specifica, chiamati *Vagrantfile*, che permettono di riprodurre un ambiente ivi descritto. Il software poi, tramite configurazioni sulla macchina host, avvia una Virtual Machine con uno degli Hypervisor disponibili (es. Virtualbox, KVM, ...), astraendo dall'implementazione e dall'hardware utilizzato. Rispetto a Docker, Vagrant garantisce una maggiore isolazione dal sistema in cui è lanciata l'istanza della macchina virtuale. Vedremo come Vagrant verrà utilizzato per dichiarare le nostre infrastrutture in maniera replicabile e configurabile.

3.4 Packer

Packer è un tool open source per creare immagini di macchine identiche, per più piattaforme da un singolo file di configurazione. Packer è un tool

leggero, e funziona su qualsiasi macchina recente, può anche essere avviato in parallelo. Non è un provisioner, ma bensì è un modo per automatizzare la creazione di immagini di Virtual Machines. Vedremo nel capitolo 5, come questo verrà utilizzato nella pipeline di release per il rilascio di immagini Qemu, Vagrant e Virtualbox, ma come viene anche utilizzato per creare le immagini dell'infrastruttura di compilazione dei pacchetti. Principalmente Packer permette di eseguire comandi su una Virtual Machine sfruttando le API del provider, e quindi fornendo un livello di astrazione su di esso. Questo ci permette per esempio di interagire sin dalla fase di boot della macchina (e quindi possono essere specificati anche comandi che modificano la modalità di boot) fino all'installazione vera e propria. L'operazione che esegue quindi è paragonabile ad una *unattended installation* su un disco magnetico virtuale (VirtualIO/SATA/IDE).

3.5 Drone

Drone è un software open source di Continous Integration, permette di eseguire e di definire dei task di esecuzione per una repository **Git** ad ogni commit. Esiste un file di specifica (drone.yaml) scritto in yaml che deve essere incluso nella repository Git. Un Host che utilizza Drone, deve avere poi la possibilità di accedere alla repository git e deve essere richiamato tramite *Webhook*, ovvero eventi che sono scatenati quando uno sviluppatore effettua delle modifiche in una repository.

Il Drone Server, legge il file di specifica yaml, ed esegue i passi ivi forniti in un container Docker, il quale può essere specificato nel Drone file. Gli artefatti, ovvero i risultati tangibili, esclusi i log, possono essere poi caricati tramite strategie di deploy.

```
build:
image: golang
commands:
  - go get
  - go build
  - go test
```

Listing 3.1: Drone: file di configurazione in Yaml

Il file in yaml, visibile in 3.1 è un esempio minimale: possiamo specificare una lista di comandi(da riga 4 a 6 in 3.1) da eseguire nel contesto di un container "usa e getta" che viene creato a partire dall'immagine specificata nel file (rigo 2). Nell'esempio viene mostrato come viene compilato e testato un software scritto in Golang.

3.6 CircleCI

CircleCI è un servizio proprietario che ha un funzionamento simile a Drone, ovvero tramite un opportuno file in yaml all'interno della repository, vengono definite diverse fasi tra cui: building, testing and deployment. In questa maniera viene trasferita l'intera logica e la *practice* del software direttamente nella repository che contiene il sorgente dell'applicativo. Citiamo CircleCI, poichè tra i servizi Cloud disponibili, permette di utilizzare anche Docker tra i propri strumenti, permettendo così l'automatizzazione e la costruzione di immagini Docker completamente configurabili tramite appositi script. Per ottenere la stessa funzionalità in Drone, si necessita di avviare il container in modalità privilegiata condividendo il socket di comunicazione con il demone Docker, oppure di *wrappare* l'esecuzione di Docker dentro Docker (chiamate soluzioni *"DiD"*) poichè non è ancora possibile farlo nativamente.

3.7 Travis CI

Travis CI è un servizio ospitato e distribuito di integrazione continua(*continous integration*) utilizzato per costruire e testare software. I progetti Open Source hanno diritto ad un *free tier*, e quindi possono testare il software senza nessun costo, per i progetti privati prevede un pagamento di una quota. Travis CI è *Open Source* e il codice sorgente si può trovare nella loro pagina dell'organizzazione su GitHub[1]. Supporta Docker e quindi è possibile costruire, testare e pubblicare immagini Docker attraverso il loro servizio.

3.8 Git

Git è un software per il controllo delle versioni di file. Viene utilizzato per lavorare in comune su un unico repository, permettendone operazioni di revert, branching e di gestione del contenuto in generale. Permette di modificare la storia dei file e mantiene tutte le differenze effettuate su di esso, grazie a ciò è sempre possibile vedere *quali* modifiche sono state introdotte e *da chi*.
Lo utilizzeremo non solo per definire dei punti centrali dove si possono collezionare i codici e le definizioni, ma anche per determinare ed analizzare le differenze nel codebase delle specifiche a distanza di tempo.
Vedremo che tramite Git è possibile interrogare la storia di una repository, e verrà utilizzato dal software sviluppato per determinare le modifiche avvenute in un intervallo di tempo, per poi essere processate da routine automatizzate.

3.9 GitHub

GitHub[2] è un servizio di hosting Git basato sul Web. Offre la tecnologia Git con servizi aggiuntivi sviluppati da GitHub stessa, come ad esempio Bug Tracker, Issues, Task Management, Wiki e la possibilità di ospitare pagine

[1] https://github.com/travis-ci/travis-ci
[2] https://github.com/

web statiche per ogni progetto. GitHub fornisce un'interfaccia grafica web di Git, con un integrazione anche tra mobile e desktop. Fornisce anche la possibilità di specificare i permessi a grana fine degli utenti, rispetto alla repository e alle organizzazioni. Le organizzazioni o le persone che condividono il codice sorgente possono utilizzarlo gratuitamente, invece è possibile creare repository private solo a pagamento. GitHub non è Open Source, ma esiste GitLab, la sua controparte libera. Viene utilizzato GitHub come storage delle repository di definizioni di compilazione e delle immagini Docker.

3.10 DockerHub

DockerHub[3] è la libreria che può contenere immagini Docker private o pubbliche. È una piattaforma messa a disposizione da *Docker Inc*, la quale componente del registro è Open Source. Possiamo paragonarli a databases dove vengono collezionate le diverse immagini disponibili. Utilizzeremo i servizi di hosting per il rilascio delle immagini Docker, ma è possibile internalizzare il servizio anche con il suo clone completamente Open Source sviluppato da openSUSE, Portus[4].

[3] https://hub.docker.com/
[4] http://port.us.org/

Capitolo 4

Soluzione per la generazione delle immagini

Analizzati i software utilizzati e le alternative presenti nei contesti delle Distribuzioni Linux (e non), qui introduciamo metodologie ibride che consentono di astrarre e separare le fasi del ciclo di release dalla Distribuzione o Sistema operativo utilizzato [1].
La soluzione che proporremo in questo capitolo si focalizza sulla riproducibilità delle intere fasi della costruzione del Sistema Operativo che andrà ad essere poi installato su dispositivi fisici oppure virtuali (come ad esempio Virtual Machines o Containers).
Grazie a Docker, la metodologia acquisirà una forte componente dichiarativa, che permetterà di specificare il contenuto di ogni immagine in maniera inequivocabile.
Vedremo anche come questa tecnica viene utilizzata anche da Scaleway per fornire servizi di hosting. Nell'ambiente *DevOps*, Docker viene utilizzato come strumento per fornire un servizio isolato dalla macchina che lo ospita, confinando l'applicativo in un sottoinsieme di risorse [42]. Vedremo ora l'utilizzo di Docker come sistema di *building* delle immagini destinate ad un uso su hardware reale, mentre nel capitolo successivo verrà utilizzato come

[1] nulla vieta, ad esempio di applicare queste metodologie a sistemi operativi BSD

provider di ambienti *sandboxed* per la compilazione “usa e getta”. Le architetture delle infrastrutture presentate, sono definite tramite file di specifica, che consente di replicare l’infrastruttura in ambiente cloud, oppure tramite implementazioni locali. Questo permette comunque di garantire la riservatezza e di tener controllo della locazione dei datacenter in contesti differenti. I capitoli seguenti presentano due tecnologie attualmente utilizzate nei processi di generazione e compilazione automatizzata dei cicli di release in Sabayon Linux.

4.1 File di specifica

```
FROM scratch
ADD stage3-amd64-20150702.tar /
CMD ["/bin/bash"]
```

Listing 4.1: Esempio di Dockerfile per importare la stage 3 di Gentoo in una immagine

Un Dockerfile è un’insieme di istruzioni che il demone Docker esegue in un container. Ogni istruzione viene eseguita su un *filesystem layer* separato, che vengono poi collegati a cascata. L’immagine finale poi viene salvata per poter essere riutilizzata in un secondo momento.

Nel file di specifica mostrato nel blocco 4.1, possiamo vedere come viene importata seed image di Gentoo in formato tar come radice di un’immagine vuota. Per indicare a Docker che la nostra immagine non avrà padri, utilizzeremo l’istruzione **FROM** , indicando come padre *scratch*, un entità fittizia. Tramite l’istruzione **ADD**, Docker permette la decompressione di un tar su una directory che possiamo definire. Questo procedimento è possibile automatizzarlo e renderlo riproducibile tramite l’uso di appositi script, che sono stati sviluppati e consultabili online [2]. Si può vedere lì anche un esempio su come automatizzare il procedimento per la fase di testing tramite l’ausilio del

[2] *https://github.com/Sabayon/docker-stage3-base-amd64/blob/master/update.sh*

servizio offerto da CircleCI. Ci sono altre istruzioni, che permettono azioni come:

- Lanciare un comando
- Aggiungere un file o una directory
- Creare variabili d'ambiente
- I processi da lanciare quando il container viene creato (*ENTRYPOINT* e *CMD*, che utilizzeremo in seguito)

Queste istruzioni sono quindi salvate in file chiamati *Dockerfiles*, che *Docker* legge quando viene richiesta la costruzione di un'immagine, esegue le istruzioni e quindi restituisce l'immagine finale. [43]

4.2 Compressione dei livelli

Le immagini di Docker sono solo templates di lettura dove i container sono lanciati. Ogni immagine quindi consiste di una serie di strati, chiamati *layers*. Docker utiliza l'unione dei file systems di ogni layer, combinandoli, per poi permettere al container di vedere una singolo strato. Per questo motivo viene utilizzato UnionFS che permette di vedere file e directory di filesystem separati (branches) come uno unico e di essere sovrapposti in maniera trasparente, formando quindi un filesystem coerente. [43]

Le immagini Docker quindi sono costruite utilizzando un set di istruzioni, che generano un nuovo layer per ogni istruzione. [44]

In figura 4.1 possiamo vedere la struttura di un'immagine a layer. Dalla versione `1.10` di Docker è stato introdotto un algoritmo crittografico per la generazione di hash sicuri delle immagini e dei strati, permettendo quindi ora di condividere stessi layer anche tra immagini differenti. Gli hash sono generati in base al contenuto dall'immagine, similmente a quanto accade in Git. Questo significa che si può garantire che il contenuto avviato è quello aspettato solamente tramite l'ID dell'immagine.

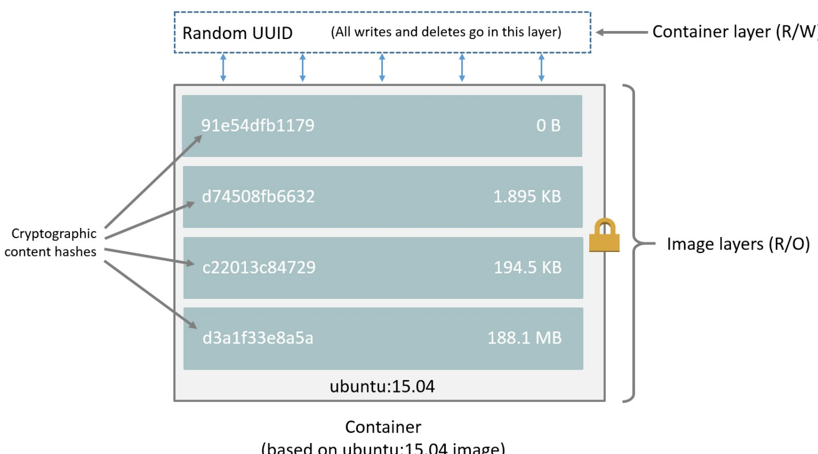

Figura 4.1: Docker: Struttura filesystem immagine a layer tramite UnionFS. Fonte immagine [44]

Creeremo quindi le immagini Docker con un modello ”a cascata” per garantire il massimo riutilizzo dei componenti, e definiremo semplicemente come andremo a trattare le immagini Docker dopo essere state costruite. Rispetto alla Figura 4.2, la fase di definizione delle immagini è quella iniziale e rappresenta la fase di dichiarazione dei componenti che devono essere aggiunti alla Distribuzione oggetto di rilascio.

Quindi a questo punto, abbiamo la necessità di riportare un'immagine che è stata precedentemente costruita da altre specifiche, ad un'insieme di file e cartelle[3], come entità unica sul disco. Avremo una cartella che conterrà quindi i file della nostra immagine, che sono ricostruiti dal sistema di layering di Docker.

4.2.1 Decompressione: docker-companion

Si è quindi sviluppato un software in Golang per automatizzare l'estrazione dei layer ed effettuarlo in maniera *corretta* e riproducibile, comunicando direttamente tramite le API di Docker e facendolo diventare un processo at-

[3] *rootfs*

tivo nella pipeline di produzione. Si poteva affrontare il problema in diversi modi, tra i quali ripercorrere sistematicamente i layer come UnionFS, ma si è scelto di procedere utilizzando le API di Docker per garantire una maggiore integrità, separando e isolando nettamente gli aspetti implementativi. Questo permetterà anche di riutilizzare il software anche in casi in cui *Docker* in un futuro possa avere differenti implementazioni a livello di storage dei dati, e quindi lavorare sempre ad un livello di astrazione maggiore.

4.2.2 Esportazione di un container

Viene quindi creato un container tramite le API di Docker a partire dall'immagine considerata. Il container, per la struttura di Docker, vedrà il filesystem sottostante (quello ereditato dall'immagine) come uno unico, grazie ad UnionFS. All'interno del container, un processo può vedere il filesystem come l'esatto stato dell'immagine, poichè è stata avviata priva di ENTRYPOINT, ma solo di uno fittizio. A questo punto il contenuto del container fornito dalle API viene rediretto ad un componente che decomprime (nativamente è fornito come tar) e scrive i file su disco. In questa maniera ottimizziamo l'intera operazione redirezionando il flusso di bit ai moduli di *compression / decompression* inclusi nel software, pilotando implicitamente i driver di interfaccia ai diversi layer dei filesystem.

4.2.3 Estrazione e ricostruzione

A questo punto, abbiamo definito la fase di estrazione della *rootfs* della nostra immagine Docker e abbiamo il sistema in forma di file e cartelle, ma non è pronta ancora per essere abbastanza "generica". Vengono qui eseguite delle routine di pulizia dai file di environment e configurazione lasciati da Docker e viene settato un DNS fittizio per evitare di avere problemi di connessione nelle immagini che utilizzeranno questa *rootfs* come *seed*. Il sorgente del software è disponibile su GitHub[4] ed è rilasciato sotto licenza GPL-2.

[4] https://github.com/mudler/docker-companion

4.3 Conversione e rilascio

Ora che abbiamo creato un software ibrido per la conversione in *rootfs* di immagini precedentemente dichiarate tramite specfile, definiamo le modalità in cui possiamo convertire la nostra immagine. Per aiutarci, in Figura 4.2 è visibile l'intero ciclo di conversione di un immagine, dalla sua definizione alla sua pubblicazione attraverso i sistemi di mirroring e distribuzione del contenuto.

4.3.1 Immagini installabili su baremetal

Dalla *rootfs* decompressa, utilizzeremo il software *platform-independent* **Molecules**, oppure un software intermedio che ne faccia le veci, che effettua le operazioni di compressione e adattamento al formato .ISO, standard per la scrittura su CD e/o drives USB. Creeremo quindi un file di specifica per Molecules, che indicherà semplicemente la cartella dove viene decompressa l'immagine Docker, e definiremo la posizione dell'artefatto prodotto (in questo caso, un'immagine .ISO). Le operazioni eseguite da Molecules sono operazioni prettamente di scrittura e lettura su file: non vengono specificate operazioni o *hook-scripts* dipendenti dalla Distribuzione Linux o dall'architettura. In questo modo svincoliamo la fase di "costruzione" dell'immagine, alla fase della conversione. Dividendo queste due aree (che fino ad adesso erano sempre viste come un tutt'uno) otteniamo una maggiore flessibilità, poichè le operazioni che sono contestuali all'architettura o al sistema, ora sono racchiuse nella fase di dichiarazione dell'immagine, e non più nella conversione del formato.

4.3.2 Immagini LXD

In modo equivalente, generiamo un'immagine per la piattaforma LXD: comprimiamo la *rootfs* precedentemente estratta dall'immagine Docker e genereremo i file di template necessari per essere letti dal demone LXD. Viene

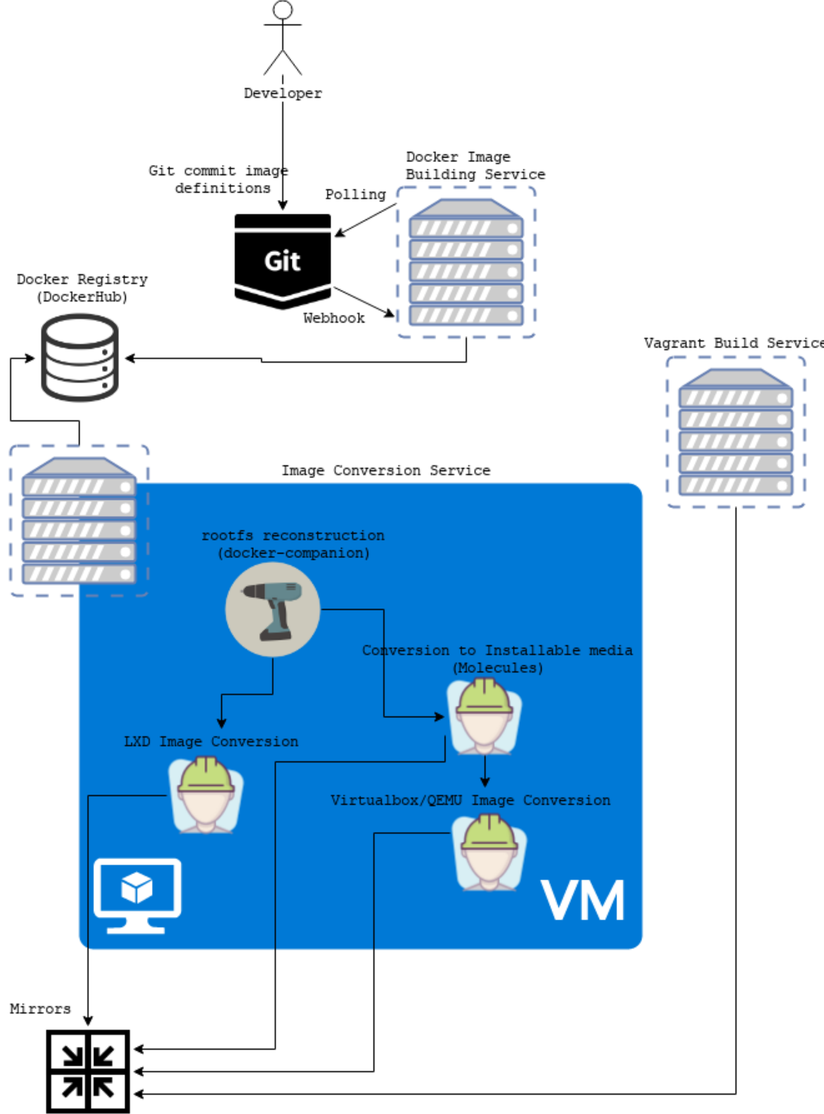

Figura 4.2: L'Architettura è altamente distribuita e i componenti sono definiti in Vagrantfiles. Qui possiamo osservare il ciclo di conversione delle immagini, dalla definizione del contenuto (Developer) al rilascio di immagini per macchine virtuali. Si noti che il servizio di conversione dell'immagine, rappresentato da una Virtual Machine, può essere eseguito ad intervalli regolari oppure al momento del commit dello sviluppatore (in questo caso, viene anche costruita l'immagine Docker)

quindi sviluppato uno script generico che converte una qualsiasi immagine Docker in una immagine LXD[5].

4.3.3 Immagini Vagrant/QEMU/Virtualbox/VMWare

Per convertire la nostra *rootfs* in immagini Vagrant, Qemu, VirtualBox e VMWare utilizziamo *Packer* fornendo in ingresso l'immagine generata tramite *Molecules*. Packer, permette di avviare e pilotare in maniera atomica i passi di esecuzione di una Virtual Machine. Una volta definita una sequenza di istruzioni da eseguire, possiamo convertire l'immagine del disco risultante nei formati per VirtualBox, QEMU e Vagrant.

Viene quindi creata una specifica per packer che, tramite un sistema di script in appoggio, crea una *unattended installation* su di un disco virtuale[6]. L'immagine risultante è poi pronta per essere affidata al sistema di distribuzione dei contenuti attraverso apposite strategie di deploy.

4.3.4 Immagini Docker

Le immagini Docker che vengono create nella nostra collezione di *Dockerfiles*, possono essere a questo punto utilizzate per creare dei container direttamente tramite la tecnologia Docker, senza necessità alcuna di essere modificati. L'unico svantaggio è che in questo caso, le immagini erediterebbero tutti i layer precedenti, ovvero quelli contenenti anche rimozioni totali di files, che comunque andrebbero ad occupare spazio alle immagini finali, nonostante siano stati cancellati. Per questo motivo, il software sviluppato *docker-companion* si occupa anche di ridurre i layer di un'immagine ad uno solo, creando una nuova immagine partendo da quella di un container creato dall'immagine sorgente[7]. In questa maniera, vengono create delle classi di immagini separate che sono utilizzabili come basi per la nuova immagine e si

[5] Disponibile online: *https://github.com/Sabayon/sabayon-lxd-imagebuilder*

[6] Il file di configurazione e il set di script ad esso correlato per l'installazione sono disponibili nella repository GitHub: *https://github.com/Sabayon/packer-templates*

[7] tramite l'opzione *squash*

possono comunque conservare tutte le operazioni effettuate su di esse poiché sempre definite da Dockerfile.

Ad ogni iterazione e composizione tramite Dockerfiles di più strati (e quindi per ogni immagine [8]) questi vengono compressi e ridotti ad un layer unico, diminuendo quindi le dimensioni finali e garantendo comunque un'alta integrità poichè l'immagine è comunque creata tramite file di specifica e i metadati vengono ri-applicati poco prima della pubblicazione.
Infine le immagini vengono pubblicate tramite servizi di *Docker Registry*, ovvero database dove vengono pubblicate le immagini e che possono essere quindi scaricate dagli utenti, nel nostro caso si è scelto di utilizzare *DockerHub*[9] che offre spazio illimitato per progetti *Open Source*.

4.3.5 Snapshots

Gli snapshots sono i formati compressi delle *rootfs* senza ulteriori modifiche. Queste, possono essere usate come *seed* in differenti fasi, ad esempio, possiamo importare un'intera *rootfs* in un immagine Docker come visto nel blocco 4.1 e poi utilizzare il risultato come base per una nuova *stage*.

Questo procedimento estremamente malleabile ci permette così di estendere il nostro metodo a qualsiasi tipo di definizione possibile. Nella nostra *pipeline*, possiamo ottenere snapshots sia da **Molecules**, sia comprimendo il risultato ottenuto dall'invocazione di *docker-companion*. Ad esempio, nel caso delle immagini in formato LXD, la *rootfs* in ingresso è fornita direttamente da docker-companion, che poi viene compressa e a cui vengono aggiunti i metadati necessari per essere indicizzato dal demone LXD.

[8] Per esempio, per creare un'immagine che contenga dei software adatti per un ambiente server, possiamo creare un'immagine Docker a partire da quella di base, installando tutti i programmi necessari, e poi convertire questa immagine, e usarla come *seed stage* per quelle successive.

[9] https://hub.docker.com/

4.3.6 Definizione infrastruttura automatica per il rilascio

Per costruire automaticamente le immagini Docker, possono essere utilizzati i servizi forniti da Cloud services come *DockerHub* o Quay.io. In alternativa è possibile utilizzare CircleCI o qualsiasi altro servizio di *Continous Integration*, in un infrastruttura esistente si possono anche definire macchine adibite alla costruzione delle immagini, utilizzando applicativi Open Source come Drone o script *adhoc* invocando direttamente Docker.

IaaS Si possono quindi definire istruzioni nel formato utilizzato dal servizio, poichè sono state isolate tutte le componenti della nostra pipeline in tool che possono essere eseguiti a cascata su ogni ambiente che fornisca un'infrastruttura. Ad esempio, possiamo definire il file 4.2 in yaml per CircleCI 4.2:

```
1  machine:
2    services:
3      - docker
4
5  dependencies:
6    override:
7     - docker build -t sabayon/spinbase-amd64 .
8     - docker-companion squash sabayon/spinbase
        -amd64
9
10 deployment:
11   hub:
12     branch: master
13     commands:
14       - docker push sabayon/spinbase-amd64
```

Listing 4.2: Esempio di file CircleCI

per avviare programmaticamente un servizio Cloud / CI che effettua le operazioni di assemblaggio dell'immagine, al momento di una modifica sulla repository di riferimento, oppure eseguirlo periodicamente. In questo modo, si è liberi di visionare i *log* e analizzare le modifiche introdotte, aggiungendo un'ulteriore stadio di testing e QA prima di passare alla production *stage*.

Gestione dello stage finale di conversione Lo stage finale di conversione, ovvero dall'immagine che contiene tutte le modifiche volute nella sua pipeline, può essere eseguita su una macchina virtuale, ed è rappresentata nel nostro caso dal software **Molecules**. Il software non è platform-dependent è può essere utilizzato per qualsiasi Sistema Operativo Linux. Si può alternativamente sviluppare un meccanismo equivalente, che deve però assolvere alle stesse funzioni.

Definizione dell'infrastruttura tramite Vagrantfile È stata comunque definita una infrastruttura base[10] tramite Vagrantfile che, data una lista di immagini Docker, preleva le definizioni su di una repository git, costruendo e poi pubblicando le immagini risultanti. È possibile consultare e replicare la macchina in qualsiasi momento, tramite l'utilizzo di Vagrant.

Ogni parte dell'infrastruttura può essere demandata a gruppi di nodi operanti su hardware differenti: ciò è possibile poichè ogni processo è rappresentabile tramite definizioni yaml (come ad esempio con Drone) e ogni entità (o raggruppamenti) è già stata definita in più Vagrantfiles.

[10] disponibile in *https://github.com/mudler/vagrant-dockerbuilder*

4.4 Applicazione nell'ambito dell'hosting: Scaleway

Possiamo trovare un esempio di applicazione di queste metodologie anche in ambito di hosting di dedicati baremetal, come Scaleway[11]. Scaleway utilizza questa metodologia per creare le immagini che poi vengono effettivamente installate sui dedicati. L'utente è libero di scegliere tra le immagini disponibili nell'ImageHub, che sono immagini Docker che contengono applicazioni installate out-of-the-box, oppure interi Sistemi Operativi. Questo permette quindi di creare ed aggiornare rapidamente immagini che poi possono essere direttamente utilizzate sull'hardware. I sorgenti con cui vengono effettuate le operazioni da Scaleway sono Open Source e disponibili al pubblico [12] ed è la prima azienda ad utilizzare Docker come sistema di costruzione e *deploy* delle immagini in hardware reale. A differenza della soluzione proposta, che è più generica e riguarda diversi ambiti (si parla infatti di supporto hardware, ma non di orchestrazione di servizi di hosting che si può implementare come ulteriore stadio produttivo) Scaleway implementa una suite di tool che modificano la definizione dell'immagine, costringendo ad una specifica più stringente e seguendo delle regole ben definite per potersi interfacciare al loro sistema di *deploy* fisico. Nel nostro caso, non dovendo allocare risorse per client (e quindi fornire accesso remoto alla macchina, ad esempio) non è contemplato nessun dialetto specifico e nessun procedimento o prassi obbligatoria all'interno delle definizioni delle immagini [13]. Il nostro metodo quindi ci permette di **riutilizzare definizioni di immagini già esistenti**, e quindi utilizzarle come *seed* per la nostra pipeline.

Potendo ora disporre dei mezzi per dichiarare l'infrastruttura in termini di immagini, passiamo a definire in termini di entità virtuali la parte dedicata alla compilazione e alla gestione del parco software.

---

[11] Scalway fornisce anche hosting su architetture ARM

[12] https://github.com/scaleway/image-builder

[13] ad eccetto dell'inclusione del binario qemu statico compilato per l'architettura dell'immagine creata, ma è del tutto opzionale

Capitolo 5

Soluzione per la manutenzione delle repository

Illustriamo ora la costruzione delle immagini adibite alla compilazione[1] del parco software della distribuzione e come automatizzare questa fase tramite la definizione di un'infrastruttura costituita da diverse entità.

5.1 Metodi per la compilazione automatica

Ci sono diversi approcci che sono stati utilizzati fino ad ora per la compilazione automatica dei sorgenti. In questa fase, vengono incluse anche quelle di *deploy* e pubblicazione dei database di indicizzazione dei pacchetti, che poi vengono propagati attraverso i sistemi di mirroring e CDN (*Content Distribution Network*).

Solitamente vengono adibite infrastrutture per la compilazione di pacchetti automatizzata tramite appositi file di specifica che indicano quali software sono destinati alla compilazione; una routine viene eseguita e procede alla verifica dei prerequisiti del sistema riguardo alle dipendenze necessarie alla fase di compilazione. In questo trattato si vuole coprire anche un'altra casistica, ovvero fornire a contributors di terze parti che non hanno accesso

[1] che a loro volta sono composte dal metodo discusso nel capitolo precedente

amministrativo ad una repository di lavoro[2] di loro interesse uno strumento per la compilazione e testing automatizzato. Daremo come assunto, che per permettere l'automatizzazione di queste procedure, esiste una repository comunque dove sono immagazzinate le definizioni di compilazione dei software. Questa repository sarà quindi gestita da un *core-team*, che valuterà e gestiranno anche le contribuzioni che vengono da sviluppatori esterni al progetto. Verranno quindi create delle immagini Docker (in base al numero di collezioni che si vogliono distribuire) che permettono la compilazione dei pacchetti specificati; queste immagini devono però soddisfare alcuni criteri per poter permettere una soluzione ibrida e che quindi possa essere eseguibile su tutte le piattaforme.

Identificheremo con il termine *build*, l'intero processo che intercorre dall'avvio di un'immagine con i vari parametri di specifica di compilazione, fino alla produzione dell'artefatto. La *build* potrà essere costituita dalla richiesta di compilazione anche simultanea di uno o più sorgenti, scritti in diversi linguaggi o possono anche solamente rappresentare un gruppo di files che vogliono essere copiati sulla macchina che installa il pacchetto.

5.1.1 Immagine "builder" ibrida per la compilazione automatica

Le immagini create con Docker, chiamate *builders* o *tinderbox*[3] dedite alla compilazione, a livello pratico vengono costruite per essere eseguite su hardware con architettura `x86_64` indipendentemente dall'architettura di supporto dell'immagine, ma in realtà potrebbero essere anche basate su altre piattaforme.
Ad esempio, si immagini di voler costruire e supportare l'architettura MIPS con la tecnologia discussa. Vengono create in questo caso, immagini *mips-builder* che producono artefatti per architttura *mipsel* ma possono comunque

[2]Come ad esempio, la repository che contiene i file di specifica per la compilazione dei pacchetti.

[3]Nel gergo Gentoo

essere eseguite da architetture host `x86_64`. Per ottenere questo risultato, vengono incluse dentro l'immagine i rispettivi binari statici di qemu in formato `x86_64` che virtualizzano per l'architettura che si vuole supportare. Per essere eseguite richiedono che sull'host dove viene creato il container vengano specificati i binari all'interno dell'immagine, utilizzando la feature del Kernel che permette di eseguire alcuni interpreti con alcuni tipi (che andremo a identificare) di binari. Viene quindi configurato **binfmt_misc** sulla macchina (in questo caso, la configurazione verrà effettuata nei Vagrantfile che descrivono l'infrastruttura) per puntare ai vari interpreti che virtualizzano e traducono le chiamate per la piattaforma host utilizzata. Ad esempio, per registrare un nuovo tipo di binario, si deve costruire una stringa di questo tipo: ":name:type:offset:magic:mask:interpreter:flags" e scriverlo in */proc/-sys/fs/binfmt_misc/register*, dipendente della locazione dell'interprete, dalle architetture e dalle personalizzazioni.

Tramite i Dockerfiles è possibile definire un Immagine Docker che all'avvio esegue un comando pre-impostato con la possibilità di definire argomenti opzionali: si possono definire *ENTRYPOINT* di esecuzione delle immagini. Nel nostro caso gli ENTRYPOINT saranno indirizzati a istruzioni che conterranno la logica per una corretta compilazione del software desiderato, con le relative parametrizzazioni (tramite variabili d'ambiente). Il programma sarà quindi un soft-wrapper attorno al package manager o alla buildchain della distribuzione e sarà *system-dependent*[4], si occuperà inoltre anche (opzionalmente) di risolvere le librerie "*rotte*" che necessitano di essere ricompilate dopo la compilazione di un pacchetto[5]. Nella versione implementata per l'applicazione della metodologia qui descritta ai cicli produttivi della distribuzione Sabayon Linux, il wrapper è stato scritto in Perl, ed è disponibile nella repository GitHub devkit[6].

[4]ovvero fortemente dipendente dalla distribuzione in uso

[5]Ad esempio: compiliamo dev-libs/foobar , ma dev-libs/foobar è utilizzata anche da app-misc/foo. app-misc/foo deve essere ricompilato e anche tutto l'albero dei pacchetti formato da quelli che dipenderanno da esso

[6]https://github.com/Sabayon/devkit/blob/master/builder

5.1.2 Software per la CI/CD Poll: Boson

Boson è il software sviluppato in Golang per permettere di effettuare l'esatta operazione inversa di Drone, vista nel capitolo precedente. Tramite dei file specifica scritti in yaml, è possibile indicare come devono essere gestite le operazioni eseguite dall'immagine e che tipo di repository è quella di riferimento. Il demone può essere lanciato in background, che ad intervalli fissi esegue le operazioni definite dall'utente, oppure può essere chiamato in modalità *one-shot* per generare i pacchetti compilati. Agisce come collegamento tra l'esecuzione di un'immagine e a quale repository è riferita.

Boson è designato in maniera modulare e possono essere inclusi plugin, che possono interagire con tutte gli stadi della *build*. Implementa un piccolo e leggero motore ad eventi, dove i plugin possono registrarsi liberamente in base alla loro tipologia.

Nella versione corrente, supporta solo Distribuzioni Linux Gentoo, ma per la sua modularità e forte separazione dei componenti, permette di essere utilizzato indipendentemente dal sistema in uso.

Motivazioni per lo sviluppo

Drone, a differenza di Boson, richiede che l'amministratore aggiunga i Webhook necessari per notificare il Drone Server per la presenza di nuovi commit, dopodichè procede ad avviare un container che seguirà i passi specificati sulla repository; inoltre, la repository di lavoro deve contenere un file che specifica le operazioni da eseguire. La differenza sostanziale è che con Boson, i contributor possono liberamente compilare e collaborare con altri gruppi di sviluppatori, indipendentemente dai permessi di scrittura sulla repository. Ciò permette quindi di separare ulteriormente le infrastrutture e i compiti, permettendo ad altre persone di rilasciare e verificare i risultati delle definizioni più rapidamente.

Contesto di utilizzo e sintassi

Boson può essere utilizzato anche attraverso vari servizi (Iaas, Paas, e SaaS) e può praticamente essere utilizzato come un agente Drone (ovvero il componente che si occupa di eseguire i comandi su di un container).

```
1  repository: https://github.com/yourusername/
       yourrepo.git
2  docker_image: docker/image
3  preprocessor: Gentoo
4  polltime: 50 # in seconds
5  artifacts_dir: /artifacts
6  log_dir: /logs
7  volumes:
8        - /host:/container:ro
9  env:
10       - LC_ALL=en_US.UTF-8
11 args:
12       - --verbose
13 separate_artifacts: true
```

Listing 5.1: Esempio: Bosonfile

La definizione è in yaml e permette di definire come devono essere gestiti gli artefatti, i log e le modalità di esecuzione dell'immagine, un'esempio(completo) è fornito nel listato 5.1. I *Preprocessors*, sono dei Plugin che si occuperanno di definire eventuali opzioni per l'avvio del container in base al contenuto della repository nel momento del commit. In caso non venga utilizzato in modalità *one-shot*[7], può essere definito un *polltime*, ovvero un intervallo nel quale viene eseguito il processo di bulding. Possiamo notare che non sono definiti qui i comandi che devono essere eseguiti nel container, ma, a differenza di Drone e dai software CI, qui spostiamo la logica dell'esecuzione

[7]E quindi, compilazioni *on-demand* in ambienti *whitebox*

dei passi dentro l'immagine stessa. Si richiede dunque ovvero di creare una immagine (che, grazie al sistema di layering di Docker, possono avere basi in comune) appositamente designata per gestire le fasi di compilazione. Questo ci permette di separare ulteriormente le logiche e definire e ritoccare i nostri *Bosonfiles* il minor numero di volte possibile. Cambiamenti a livelli di Bosonsfile, durante la stesura di un progetto sono veramente eccezionali, poichè qui andiamo a definire i risultati e i file che vengono analizzati, non l'intero processo. Boson agirà come agente che esegue passi definiti dall'immagine all'interno di un ambiente sandboxed.

Immagini Docker Boson utilizza gli ENTRYPOINT dell'immagine, supponendo che essi siano progettati per terminare in maniera corretta o non corretta a seconda dell'esito della *build*. Quindi i Bosonfiles definiscono "come" l'esecuzione può essere modificata attraverso gli argomenti e le variabili di ambiente, ma non interagisce con l'operazione stessa, separando ancor di più e astraendo le logiche d'ambiente e mantenendo l'integrità tra di esse.

Boson effettua le operazioni interagendo direttamente con il demone Docker, e quindi crea un nuovo container a partire dall'immagine specificata, esegue le logiche definite all'interno (e quindi compila un pacchetto, nel nostro caso) e poi copia gli artefatti risultanti nelle cartelle indicate. Ad ogni operazione, vengono valutate solo le differenze intercorse tra i commit che non sono stati "visti" dal software, ovvero tutti i commit che sono intercorsi tra due fasi di polling (o anche semplici invocazioni).

I file di configurazioni non devono essere necessariamente nella repository di lavoro, ma si possono creare ad esempio, in repository esterne che contengono file di specifica(anche più di uno) che si riferiscono a quelle di lavoro[8]. In questa maniera si possono anche generare *build* regolari, indipendentemente dallo stato della repository principale.

[8] Un esempio è disponibile su GitHub per generare report di QA e compilazione sulle repository di specifica di compilazione di Sabayon: https://github.com/mudler/sabayon-bosons

PaaS

In ambienti *PaaS* (Platform as a Service) è immediato l'utilizzo di Boson, esso può essere utilizzato, ad esempio in OpenShift, come demone, e quindi provvedere alla distribuzione degli artefatti e logs nelle modalità definita dalla PaaS in uso.

IaaS

In ambienti *IaaS* (Infrastructure as a Service) come ad esempio CircleCI, possiamo utilizzare Boson in modalità *one-shot* e mantenere una cache costante tra le diverse build. Questo è necessario per poter tenere traccia dei commit di cui già sono state effettuate le compilazioni, in caso questo non sia possibile, le differenze vengono conteggiate con il penultimo commit, per evitare di ricostruire e ricompilare l'intero stato del sistema.

Analizziamo ora come può essere utilizzata l'immagine e la metodologia proposta in ambito di Continous Integration con servizi esterni (o non, nulla vieta di internalizzare un servizio come Drone.io).

5.1.3 Continous Integration

I sistemi di Continous Integration (CI) come Travis, Drone o CircleCI possono essere utilizzati per compilare i pacchetti quando vengono effettuate modifiche nelle repository di definizione. Analizziamo ora un esempio per chiarirne l'utilizzo, che è utilizzato attualmente per la repository di Sabayon[9], sono stati aggiunti i file 5.2 e 5.3 alla repository Git per configurare il servizio Travis-CI.

[9] la repository in questo caso rappresenta un *overlay* di Gentoo, ovvero una collezione di definizioni di compilazione di pacchetti che viene aggiunta sulla collezione di Portage ufficiale

```
1  sudo: required
2  services:
3    - docker
4  script:
5  – ATOMS=$(./scripts/atoms–in–commit–range.sh
6  ${TRAVIS_COMMIT_RANGE} sabayon)
7    && [ –n "$ATOMS" ]
8    && docker run –e "SKIP_PORTAGE_SYNC=1"
9    –e "EQUO_MIRRORSORT=0"
10   –v $TRAVIS_BUILD_DIR:/var/lib/layman/sabayon
11   sabayon/builder–amd64 $ATOMS
12   || exit 0
```

Listing 5.2: Travis-CI: Esempio

Il servizio Travis-CI deve essere abilitato tramite *Webhook*[10] attraverso il pannello di controllo della repository Git[11].

[10]Sono delle semplici definizioni, dove ad ogni evento che accade su di una repository Git viene specificato un *endpoint* dove inviare le informazioni, in questa maniera Travis-CI può avviare una *build* non appena viene effettuata una modifica

[11]in questo caso, viene utilizzato GitHub, ma si applica anche a diversi servizi di Hosting Git, come ad esempio BitBucket o Gitlab

```
1  #!/bin/bash
2
3  if [ -z "$1" ] || [ -z "$2" ]; then
4   echo atoms-in-commit-range.sh hash..hash overlay
5   exit 1
6  fi
7
8
9  git diff-tree --name-status -r --no-commit-id ${1} \
10 | grep -v "^D" \
11 | sed -r -e 's/^[a-zA-Z0-9]+[[:space:]]*//'
12 -e 's:^([^/]+/[^/]+).*:\1:' \
13 | sort -u \
14 | grep -E '^(virtual/|[^/]+-)' \
15 | awk "{print \$1 \"::${2}\"}"
```

Listing 5.3: Travis-CI: Script per trovare le differenze tra due commit git

Lo script 5.3, prende in ingresso un *range* di commit e l'overlay di riferimento. Tramite una combinazione di comandi vengono estrapolati i file che sono cambiati nell'intervallo specificato e vengono poi tradotti nel formato Gentoo di rappresentare i pacchetti(*category/software*).

Quindi viene definito poi un file in yaml sulla radice della repository come quello mostrato in 5.2; invocheremo quindi l'immagine Docker adibita per la compilazione(e che quindi contiene le logiche di building al suo interno) con argomenti i pacchetti che devono essere compilati da Travis-CI.
Possiamo riassumere il funzionamento del meccanismo CI in Figura 5.1. Si può anche osservare qual'è l'operazione di Boson, ovvero permettere la compilazione tramite servizi di CI ad intervalli regolari per valutare la QA e la compilazione.

5.1.4 Sabayon Development Toolkit

È stato creato un toolkit per gli sviluppatori per facilitare l'utilizzo delle immagini come ambiente di testing localmente. In realtà il toolkit è un mero wrapper all'esecuzione parametrizzata degli ambienti già preparati, che

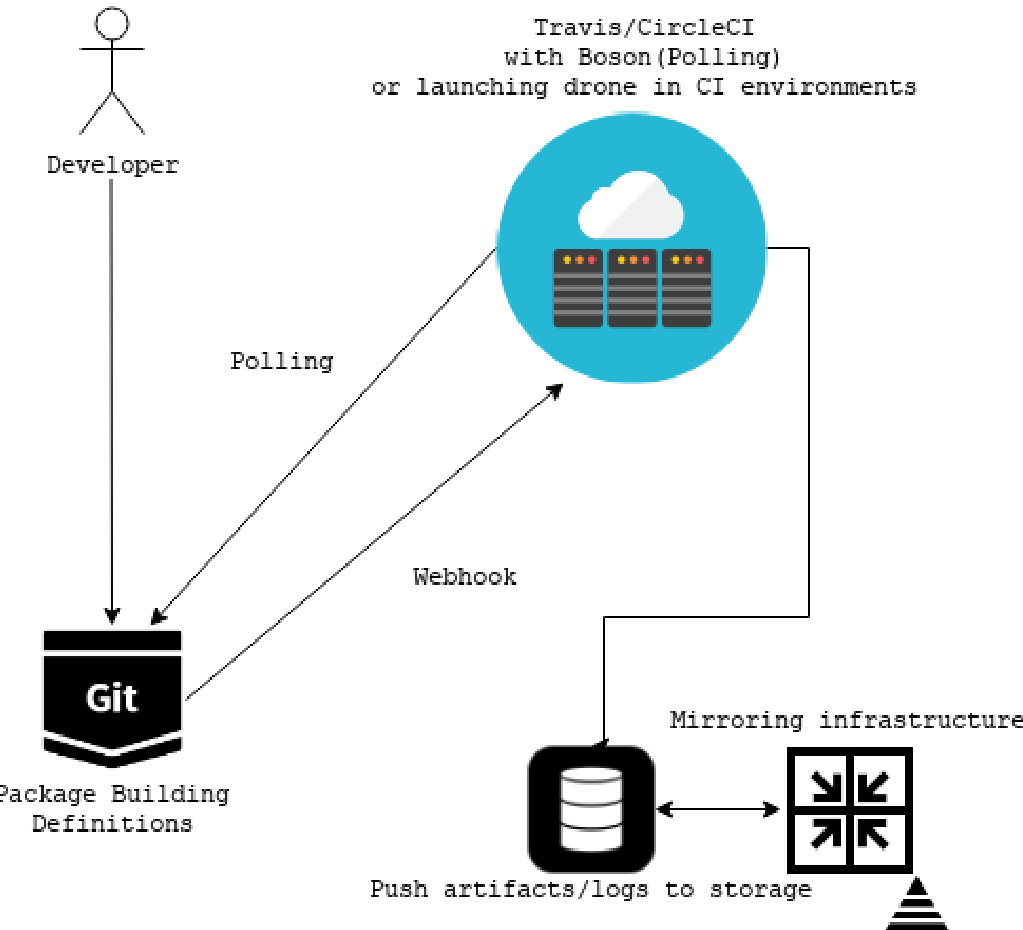

Figura 5.1: Al momento del commit di uno sviluppatore sulla repository delle definizioni di compilazione dei pacchetti, un'istanza di un servizio CI(Travis, CircleCI, Drone, ...) esegue il building del pacchetto. Per il caso opposto invece, boson, tramite i servizi CI effettua le compilazioni e fasi di *QA* ad intervalli regolari

implementano la logica di building. La suite quindi è divisa in due parti: la prima riguardante i tool che sono eseguiti da uno sviluppatore che vuole testare e compilare software utilizzando un'ambiente di lavoro *sandboxed*, evitando il rischio di rendere instabile il proprio sistema. La seconda si occupa invece della logica di compilazione utilizzata proprio all'interno delle

immagini: così possiamo aggiornare agilmente le istruzioni e al tempo stesso definire l'uso da linea di comando in una unica sezione. Così facendo abbiamo isolato in 2 parti le logiche dipendenti dal sistema su cui vogliamo eseguire le operazioni.

Compilazione in ambiente isolato

Per compilare un pacchetto che ha un file di specifica dichiarato tramite il sistema di Portage, è necessario digitare sul terminale ad esempio: 'sabayon-buildpackage app-text/vim'. Viene lanciata quindi l'immagine *builder*, con l'argomento app-text/vim. Le logiche di compilazioni vengono demandate all'immagine, che contiene il pacchetto *sabayon-devkit* già installato. *sabayon-devkit* incorpora anche le logiche di compilazione locale.

5.2 Creazione e rilascio dei pacchetti

I software, prima di essere compressi e rilasciati tramite il gestore di pacchetti vengono compilati e gestiti da un'insieme di software e toolkit che cooperano per rilasciare le varie repository per poi essere propagate negli appositi mirror dell'organizzazione.

5.2.1 Infrastruttura automatizzata

In questo contesto è facile orchestrare la situazione già presentata tramite software quali Foreman, Mesos o Marathon. Per permettere una ulteriore astrazione e quindi fornire la possibilità di replicare l'infrastruttura per piccoli gruppi di lavoro e anche di definire l'infrastruttura come entità cloud, viene sviluppato un'ambiente di infrastruttura, specificato in Vagrant. Per aiutarci, è possibile vedere l'architettura del sistema in Figura 5.2.

Vagrant L'infrastruttura è completamente definita come entità in un Vagrantfile. Questo ne permette la riproducibilità e la scalabilità in qualsiasi contesto. Definire un ambiente in termini di virtual machine e di ambienti ci

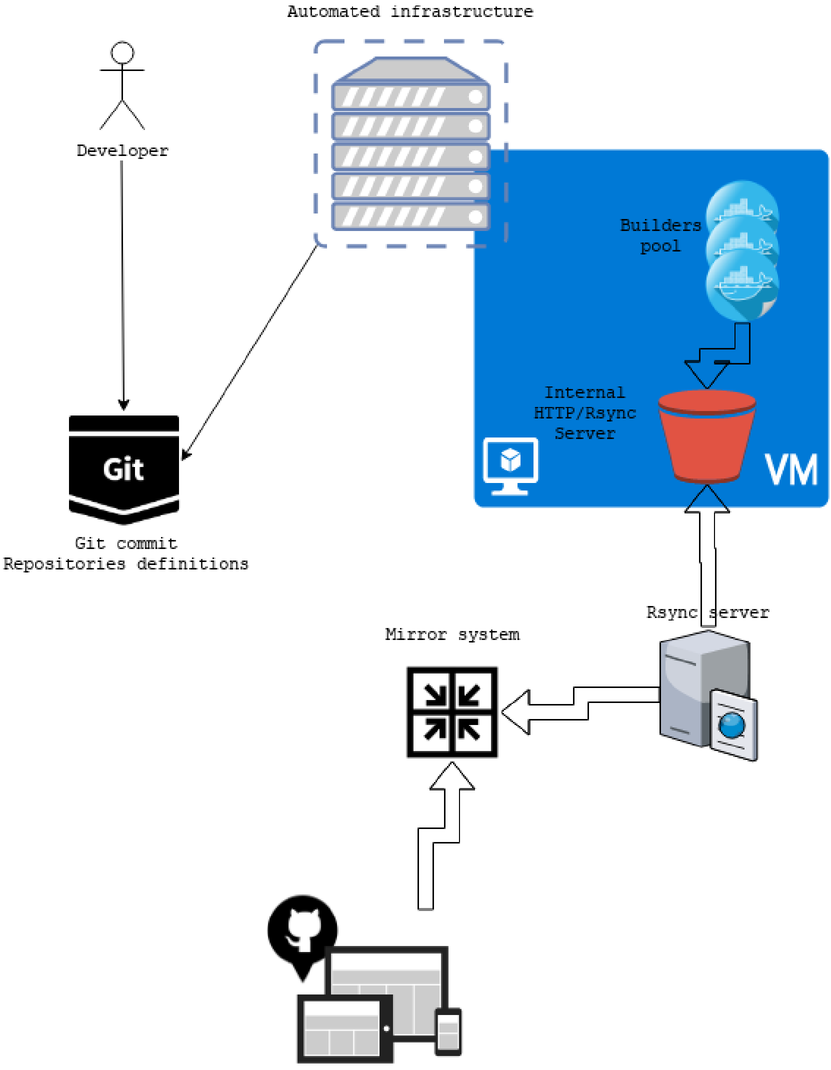

Figura 5.2: Architettura dell'infrastruttura definita tramite Vagrantfiles

permette quindi di poterlo applicare in un contesto cloud, come CoreOS/OpenStack che definiscono i servizi offerti dalla macro-infrastruttura tramite i *cloud config file*.

L'infrastruttura quindi auto-configura e amministra più repository di lavoro, dove vengono lette le specifiche di compilazione attraverso una reposito-

ry git. La repository git conterrà quindi i file di specifica per ogni repository che si vuole compilare, definendo di fatto i così chiamati *target* di compilazione. Anche qui, in accordo anche con la maggior parte dei software e sintassi già viste, si è scelto sempre yaml come modello di riferimento. Possiamo quindi definire in entità le componenti che caratterizzano le nostre repository, che verranno compilate in ambienti separati e isolati dal sistema Host (definito tramite Vagrant file).
Tramite la parametrizzazione, possiamo configurare con grana fine il nostro ambiente di compilazione, decidere ad esempio se rimuovere e distruggere ad ogni esecuzione il container dove vengono compilati i pacchetti, e quindi rigenerare per intero l'ambiente di building ad ogni ciclo (per repository), oppure conservare una cache che preserva lo stato del sistema dove vengono compilati i pacchetti.
L'infrastruttura è configurata per eseguire le specifiche quotidianamente e parallelamente utilizzando *GNU Parallel* [45]: vengono eseguiti gli ambienti di *building*[12], la cui struttura è visibile in Figura 5.3 riferite ad ogni gruppo di specifiche aventi immagini separate, gestite in maniera automatiche (che vengono rappresentate dai *Builders pool* in Figura 5.2).

A partire da una *seed* image, definita in Docker, vengono generati N ambienti (*builders pool*) a partire da essa, dove N sono il numero di gruppi di specifiche definite. Da quel momento, ad ogni iterazione, le logiche di compilazione definite nell'immagine, effettuano modifiche incrementali all'immagine, portandole ad uno stadio *build-ready*.
Tra il *seed* stage e quello *build-ready*, vengono installate automaticamente le dipendenze necessarie alla compilazione dei pacchetti specificati, permettendo alle *build* successive di compilare direttamente evitando di risolvere le dipendenze in ogni esecuzione. Gli ambienti ad ogni ciclo aggiornano i propri software ed effettuano la compilazione dei singoli atomi di pacchetto (*es. media-tv/kodi*). Questo ci permette quindi di considerare riproducibili anche gli stati definiti dai file di specifica delle repository.

[12]tramite container Docker

Figura 5.3: Struttura Virtual Machine con ambienti di compilazione Docker

In Figura 5.3 si può vedere come è strutturato l'ambiente virtualizzato di un server fisico. Gli ambienti di *building* sono formati da istanze di container Docker a partire dalle immagini (che vengono costruite per ogni repository a partire dall'immagine *builder* di base che incorpora le logiche di compilazione).

Specfiles e fase di build

Gli *specfile*, in questo contesto, sono i file di specifica che definiscono *come* e *cosa* compilare. Ogni *repository di pacchetti*, ovvero l'insieme di database e pacchetti che viene distribuita tramite i mirror e disponibile agli utenti è definita in un formato yaml, che ne dichiara il contenuto.

Viene quindi costruita un'infrastruttura e redatta una specifica sintassi per la configurazione dell'ambiente di compilazione, il risultato è pubblico su GitHub[13] ed è la definizione della Virtual Machine in Figura 5.2, di cui andremo ad analizzare il funzionamento.
Si possono anche osservare le specifiche che sono state utilizzate per la creazione di una collezione di repository disponibili a tutti gli utenti *Sabayon*, consultabile sempre su GitHub[14] e rappresentano le modifiche effettuate dagli sviluppatori.
Vengono utilizzate quindi repository Git per collezionare i specfile, relativi ad ogni repository di pacchetti.
Ogni cartella nella repository git corrisponderà ad una repository di pacchetti, e al suo interno **deve** essere presente un file denominato **build.yaml** che contiene **almeno** la lista dei pacchetti da voler compilare. Il nome della repository, se non viene specificato nel file di definizione, viene dedotto dal nome della cartella. Possono essere presenti due ulteriori cartelle all'interno: **specs** e **local_overlay**.

- **specs** contiene le ulteriori configurazioni, che variano a seconda del gestore di pacchetti.
- **local_overlay** è dove possono essere depositate le (opzionali) definizioni dei pacchetti

A livello strutturale vengono definite funzioni, raggruppate in classi anche per permettere all'utente di esprimere la richiesta di inclusione di repository esterne dove sono presenti altre collezioni di specifiche dei pacchetti da compilare, così da poter evitare di utilizzare **local_overlay** e isolando ancor più le aree di lavoro.

Il file che indica le specifiche di compilazione, **build.yaml** , in base alle configurazioni, mediamente prende la seguente forma di dettaglio:

[13] https://github.com/Sabayon/community-buildspec
[14]https://github.com/Sabayon/community-repositories

```yaml
repository:
    description: Devel channel repository
build:
  emerge:
    # Install each package separately
    split_install: 1
    features: assume-digests binpkg-logs ...
  overlays:
    - kde
  target:
    - =sys-libs/kpmcore-9999::kde
    - =sys-kernel/genkernel-next-9999
    - =app-admin/calamares-9999
    - sys-kernel/linux-sabayon
```

Listing 5.4: Esempio building specfile: Definizione della repository Developers

Notiamo, come nel caso specifico, possiamo avere configurazioni dipendenti dal sistema. Questo è inevitabile per molti motivi, principalmente:

- differenti Distribuzioni Linux, hanno diverse nomenclature per i software nella loro collezione
- differenti modi di definire le opzioni di compilazione (se presenti) [15]

Questo ha portato al design di un file di specifica il più possibile astratto, così da poter demandare l'implementazione specifica alle istruzioni incluse nell'immagine che viene usata come base di sistema per la compilazione. In questo modo, nei file di specifica inseriremo configurazioni *system-indipendent*, dove alcuni set di queste configurazioni saranno supportati da alcuni sistemi, e da altri verranno ignorati. In alternativa si possono abilitare

[15] In contesti di distribuzioni come Fedora ed ArchLinux ad esempio, le metodologie di compilare i pacchetti sono staticamente scritte nello specfile, non permettendo l'abilitazione/disabilitazione di specifiche *features*, nativamente

sezioni *Package Manager* dipendenti che rendono necessarie modifiche solo al set di istruzioni presente nelle immagini (*ENTRYPOINT*), come possiamo vedere nell'esempio in 5.4.

Ogni file di specifica non rappresenta soltanto una repository, ma è anche la configurazione di un ambiente isolato di compilazione. Per ogni repository, viene creata un'immagine apposita, a cui vengono aggiunti i layer per ogni esecuzione della nostra fase di *build*. Ulteriori modalità di utilizzo, compilazione e configurazioni specifiche possono essere richieste, si può vedere un esempio di file di configurazione completo in appendice A.1, si può anche consultare online[16].

Nell'esempio in 5.4:

- **description**: definisce la descrizione della repository
- **build** rappresenta il blocco descrittivo riferito alla build corrente
- **emerge** indica il blocco relativo al gestore di pacchetti di Gentoo
- **split_install** istruisce il *builder* di effettuare le installazioni sequenzialmente a come vengono scritte
- **features**: specifico di Gentoo, sono le feature del gestore di pacchetti
- **overlays**: ulteriori repository di definizioni di compilazione
- **target**: la lista dei software da compilare

Il risultato del processo di *building* per ogni repository è un file di log, e N pacchetti, equivalenti a quelli specificati nel file yaml. Entrambi vengono poi pubblicati nel nodo stesso, tramite protoclli rsync e http, i cui servizi sono avviati e definiti nel **Vagrantfile**. Nel Vagrantfile[17] vengono anche

[16] https://github.com/Sabayon/community-repositories/blob/master/build-example.yaml

[17] https://github.com/Sabayon/community-buildspec

definite le configurazioni specifiche relativi ai dischi e al partizionamento ottimale di Docker per un uso in production [18]. Inoltre, viene eseguita una routine settimanalmente che comprime tutte le immagini Docker delle repository utilizzando *docker-companion*, questo sia per ridurre spazio, ma per rendere anche l'esecuzione della compilazione più performanti (altrimenti, le operazioni di lettura tra multiply layer diventerebbero più *I/O intensive* che *CPU intensive*), seppur perdendo metadati, questi non sono di alcun interesse, poichè lo stato delle repository potrà sempre essere riprodotto grazie alle dichiarazioni dei contenuti in maniera dettagliata e agli ambienti di isolazione che permettono la compilazione del software in ambienti *sandboxed*. È possibile vedere tutte le opzioni che possono essere specificate nella prima appendice, nel listato A.1.

Sito web per la ricerca dei pacchetti

Viene sviluppato anche un sito web statico per la ricerca dei pacchetti tra le varie repository[19], il risultato è mostrato in Figura 5.4 e Figura 5.5. In Figura 5.4 si può vedere come l'utente, iniziando a digitare alcuni caratteri, alla fine può selezionare tra un elenco il pacchetto ricercato. A quel punto, vengono mostrate le opzioni per abilitare la repository di pacchetti dove esso è contenuto, e vengono mostrati i passi per l'installazione Figura 5.5.

La collezione delle definizione delle repository vengono identificate, nel caso specifico applicativo, in un unico gruppo dove vengono generati i metadati ed esportati per essere poi processati in un secondo momento dai client. La fase finale di compilazione e pubblicazione della repository, prevede una fase aggiuntiva adibita a generare metadati in formato testuale e **JSONP**.

[18]La configurazione di default utilizza la modalità *direct-lvm* e *thin-pool*, come consigliata dall'organizzazione di Docker stessa - *https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/*

[19]Il codice è disponibile nella repository GitHub: https://github.com/Sabayon/community-website

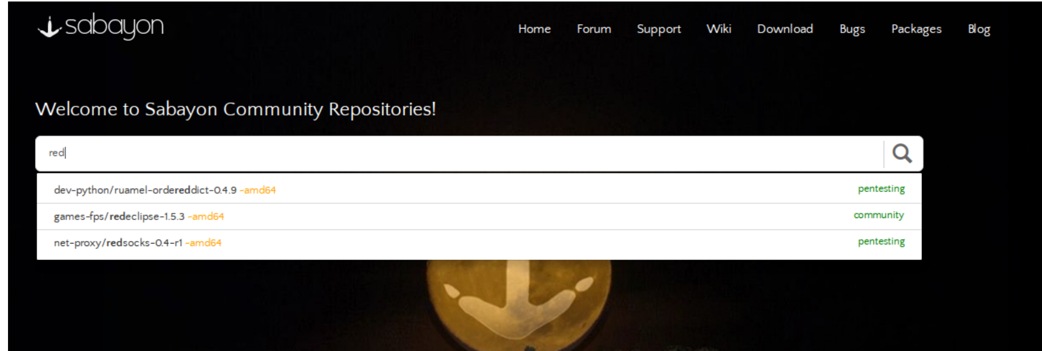

Figura 5.4: La ricerca viene effettuata iniziando a digitare nella casella di ricerca

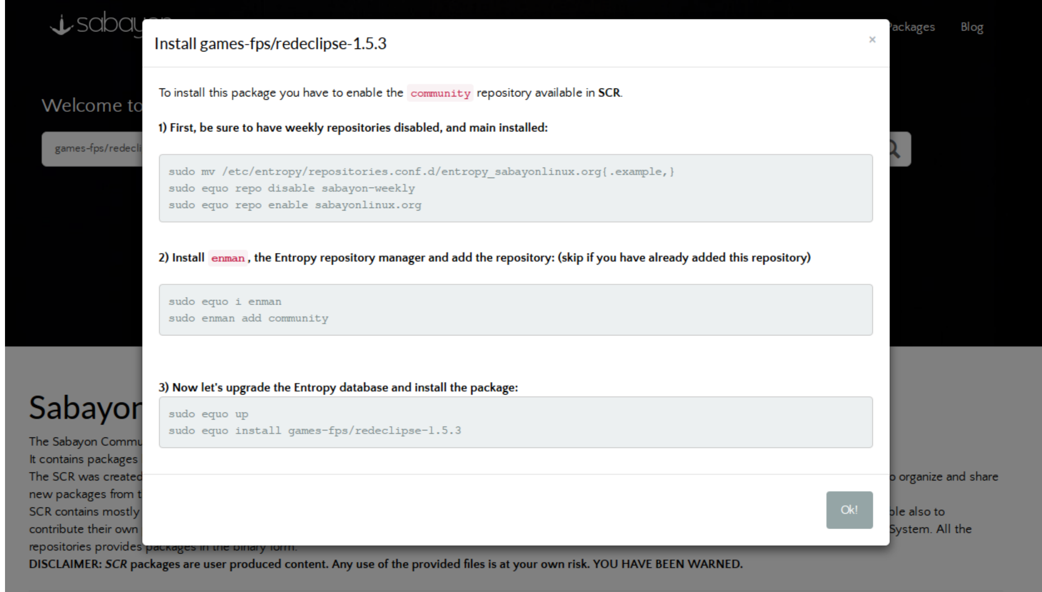

Figura 5.5: Al click, vengono mostrate i passi da eseguire per aggiungere la repository

L'applicativo web è stato sviluppato staticamente[20] tramite il software Hugo[21] e la libreria *jQuery Typeahead*[22] dove i metadati necessari per la ricerca vengono forniti tramite la tecnica **JSONP** permettendo quindi di: precomputare in formato JSON i metadati relativi alle repository, pubblicarli attraverso un sistema di mirroring (il medesimo utilizzato per i pacchetti) e quindi permettere l'operazione della ricerca direttamente dal client, senza la necessità di un server per soddisfare le richieste. Viene utilizzato Hugo poichè molto affine con le tecnologie CI. Infatti l'intera generazione delle pagine del sito è affidata a servizi esterni(CircleCI), che pubblicano direttamente il risultato attraverso i servizi di hosting statico gratuito di GitHub[23], sollevando quindi completamente il costo di risorse di questo meccanismo.

JSONP è una tecnica che è utilizzata nello sviluppo web per "bypassare" le restrizioni di cross-domain validation dei browser (ovvero, se i file richiesti da un pagina web vengono sempre dallo stesso dominio dove la pagina è servita). La metodologia si è rivelata particolarmente utile poichè i metadati vengono propagati attraverso i mirror. Quindi la tecnica è necessaria per diminuire le risorse per il mantenimento del sito, assieme all'utilizzo di tecnologie statiche per la generazione delle pagine. I dati vengono forniti in formato JSON, vediamo un piccolo esempio di file in **JSONP** 5.5.

```
1 parsePackages({ "Id": 1,
2                 "atom": "app-editors/vim",
3                 "repo": 1});
```

Listing 5.5: Esempio JSONP

Il file viene incluso normalmente come una risorsa Javascript 5.6 dal sito che vuole utilizzare i dati. Viene quindi implementata una routine per la gestione dei dati che sono salvati nel file in JSONP, al lato client, durante la fase di

[20] https://sabayon.github.io/community-website/
[21] https://gohugo.io/overview/introduction/
[22] https://github.com/running-coder/jquery-typeahead
[23] https://pages.github.com/

caricamento del sito.

```
1 <script type="application/javascript"
2       src="http://server/jsonp?callback=parsePackages">
3 </script>
```

Listing 5.6: Esempio JSONP, inclusione nella pagina web

Il client in questo caso invocherà la funzione *parsePackages*, che conterrà le logiche di trattamento dei dati.

Gestore di repository lato client Per aiutare gli utenti a fruire delle collezioni di repository, facilitando quindi l'aggiunta, la rimozione e la ricerca è stato sviluppato un software apposito, denominato *enman*[24].

5.2.2 QA

Il modello *DevOps* adottato fornisce un contributo significativo al processo di Quality Assurance. Nei sistemi informatici, la quality assurance è condivisa tra tutte le fasi di un prodotto. La quality assurance è un analisi che va fatta a qualsiasi stadio e la ritroviamo dallo sviluppo fino al cliente. La metodologia di sviluppo Agile e l'Open Source hanno modificato in parte questo ecosistema, ma il personale addetto ora ha accesso a molti più dati. La cultura *DevOps* mette in comune le operazioni, gli sviluppatori e la QA tramite strumenti e "cooperazione" delle parti.
Di solito la responsabilità di questo settore viene affidata a qualcuno che è familiare con entrambe le parti (sviluppo e operazioni), capace quindi di identificare e analizzare i dettagli che potrebbero costituire problemi per il prodotto/servizio in maniera più celere. Inoltre, potendo così monitorare su qualsiasi parte del ciclo produttivo si possono identificare con più facilità e velocità i problemi percepiti dagli utenti finali[25]. [20]

[24] https://github.com/Sabayon/enman
[25] la malleabilità della metodologia ci permette anche di aggiornare e risolvere i problemi in maniera più tempestiva (sotto il punto di vista dell'utente)

5.2.3 Rilascio

Il sistema di rilascio dei pacchetti o quello delle immagini è comune per entrambe le soluzioni qui proposte. Per rilascio si intende l'insieme di tecnologie che vengono utilizzate per permettere all'utente di scaricare effettivamente il contenuto prodotto (artefatto). Questo, principalmente comporta:

- Gestione dei mirror
- CDN (Content Distribution Network)
- Gestione dell'infrastruttura (cluster o non) per fornire i dati

In entrambi i sistemi proposti (per pacchettizzare o scrivere le immagini), si possono adottare due strategie:

- I produttori caricano l'artefatto (pacchetto/immagine) direttamente nel centro di smistaggio (uno storage tramite GCE, un cluster FS o servizi come BinTray e Nexus)
- Ci sono dei nodi che hanno il compito di prelevare gli artefatti dai Produttori

Non ci sono forti vincoli imposti dall'architettura, ma è solo una scelta implementativa che non ha nessun effetto sulle nostre strutture. In Sabayon Linux, si è scelto di svincolare ulteriormente le entità: i nostri sistemi implementano nodi intermedi che si occupano di prelevari i dati dei produttori[26] e di immagazzinarli nell'infrastruttura per la distribuzione del contenuto, che fornisce i dati attraverso più protocolli (Rsync, HTTP, torrent, ftp). Questa scelta è dovuta ad un fattore di sicurezza. Si è limitata la superficie di attacco svincolando maggiormente il buildserver dell'infrastruttura, in modo tale che tutte le entità possano essere descritte e lette anche pubblicamente.

[26] *build servers* - per quanto riguarda la compilazione dei pacchetti tramite file di specfica

Conclusioni e sviluppi futuri

Abbiamo analizzato i sistemi esistenti implementati dalle maggiori Distribuzioni Linux *Open Source* per la gestione dei pacchetti e il rilascio di immagini installabili valutando gli approcci esistenti e considerando quindi lo stato dell'arte - benchè nell'ecosistema *Open Source* è difficile delineare uno stato dell'arte - troviamo in *Gentoo* un'ottima fonte di ispirazione, poichè adotta un sistema completo e dettagliato [27]: automatizza ogni piccola parte ed implementa anche il concetto di riproducibilità, totalmente affine con la cultura *DevOps*.
Le soluzioni fornite dalle altre distribuzioni non prevedono solitamente una forte scalabilità e generalizzazione[28] e vengono maggiormente implementate come *collezione* di scripts; tipicamente vengono sfruttate le automazioni tramite *cron* o con soluzioni completamente *adhoc*. Analizzate quindi le tecniche utilizzate da Gentoo e da altre distribuzioni, in questa tesi proproniamo un'approccio diverso e nuovo, basato su metodologie *DevOps* che garantiscono quindi una futura scalabilità su piattaforme Cloud.
Uniamo la metodologia di sviluppo di Distribuzioni Linux *Open Source*, nota per essere una delle componenti più attive nelle contribuzioni, con una metodologia del campo *IT* che aiuta e velocizza i processi di sviluppo. Le tecnologie qui esposte sono attualmente implementate nei cicli produttivi della Distribuzione Sabayon Linux: permettono di automatizzare completamente i cicli di sviluppo e rilascio, sollevando gli sviluppatori dall'annoso

[27] ed è studiato anche nella letteratura scientifica in termini di sistemi complessi [28]

[28] un eccezione è la soluzione fornita da openSUSE, ma è a pagamento e non è stato possibile valutarne il funzionamento e l'architettura

compito di configurare e mantenere un proprio ambiente di lavoro.

Sostituiamo quindi ai meccanismi di rilascio e compilazione predominati dagli ambienti *chroot* i meccanismi che utilizzano metodologie e tecniche *DevOps*.

Abbiamo analizzato come è possibile definire le infrastrutture per il rilascio delle Distribuzioni Linux tramite entità ben definite in appositi file di specifica, permettendo quindi la riproducibilità di ogni parte di esso, anche in ambienti differenti da quello puramente Cloud.
Nel corso dell'implementazione, sono stati sviluppati una suite di tool per la gestione degli ambienti *sandboxed* per i sviluppatori e piccoli software che vengono ora utilizzati nella *pipeline* in produzione, di cui abbiamo parlato nei capitoli precedenti. In particolare, vengono descritte ed implementate le infrastrutture e i tool adibiti per l'automatizzazione della compilazione e della gestione del parco software e del rilascio della distribuzione.

La tecnologia illustrata può essere ampliata e migliorata ad esempo tramite l'utilizzo di nodi dedicati all'implementazione di un sistema di gestione di smistaggio del carico tra i vari *builder* e la definizione di un *endpoint* che raccoglie le richieste degli sviluppatori (o utenti), sulla falsa riga del prodotto SUSE studio, ma con un accezione FOSS ed implementata con metodologie *DevOps*.
Si potrebbe quindi realizzare un sistema di building dei pacchetti e delle Distribuzioni Linux implementando delle apposite *API* scalabili tramite l'uso di *microservices* e cluster di nodi, per la compilazione e l'assemblaggio delle immagini.

Appendice A

Prima Appendice

```
1  #Most of the configurations are optional,
2  #Required: build.target and repository.description.
3  repository:
4  # Repository description.
5    description: My testing repository # REQUIRED !!!
6    maintenance:
7    # Throws away the container and the image
8    # refeered to the repository
9      clean_cache: 1
10     # Set this value to the numbers of package
11     # versions that you want to keep.
12     # with 1 it keeps just one package version online
13     keep_previous_versions: 1
14     # Packages that might want to be manually removed
15     remove:
16       - app-foo/bar
```

```
17          - app-misc/baz-1.2
18 build:
19 # BUILD Definition.
20 # It contains description and
21 # options regarding the build.
22   overlays:
23     - mrueg
24     - games-overlay
25   # Adds the specified repository
26   # compile definitions collection
27   target: # REQUIRED!!!
28   # the software(s) that want to be compiled
29     - app-text/tree
30     - dev-lang/php
31   injected_target:  # Packages that could
       conflict
32   # will be added anyway
33     - app-text/tree
34     - dev-lang/php
35   equo: # Sabayon PM options
36     repositories:
37       - community
38     # add the repositories before the build
39     remove_repositories:
40       - pentesting
41     enman_self: 0
42     # adding your repository into the container
43     package:
44       install:
45         - app-text/tree
46         - app-misc/foo
```

```
47      # List here the packages that you might want
48      # installed with Entropy official repositories
49      # before starting the build
50      remove:
51        - whatever
52      # packages to be removed
53      # before the build
54      mask:
55        - whatever
56      # List here the packages that you might
57      # want masked before the build
58      unmask:
59        - whatever
60      # List here the packages that
61      # you might want masked before the build
62    repository: main
63    # You can build against
64    # main (sabayonlinux.org) or weekly.
65    # It is strongly encouraged to keep default
66    dependency_install:
67      enable: 1
68      # Enable/disable equo
69      # to install package dependencies
70      # (automatically detected)
71      install_atoms: 1
72      # Require to install
73      # the atoms of the dependencies,
74      # not their specific version
75      dependency_scan_depth: 2
```

```
76        # How much have to deep the dependency
             scanner
77        # to get your deps
78        prune_virtuals: 1
79        # Remove packages that satisfy a
80        # virtual dependency from the list of
             packages
81        # to install using entropy.
82        install_version : 0
83        # Install packages-specific version
84        # instead of atoms with equo.
85        # Discouraged
86        split_install: 1
87        # Install packages calling
88        # "equo install" separately
89  ## Advanced options below:
90  docker:
91    image: sabayon/builder-amd64
92    # Docker image to use as building
         environment
93    entropy_image: sabayon/eit-amd64
94    # Docker image to use as repository
         environment
95  emerge: # Gentoo PM specific options
96    default_args: -t
97    # Emerge default arguments.
98    split_install: 1
99    # If set to 1,
100   # calls emerge for each package
101   # instead in one single call.
102   features: parallel -userpriv
```

```
103     # features to enable/disable to your build
104     profile: 3
105     # Gentoo Profile to use
106     jobs: 4
107     # Package compilation/installations in
          parallel
108     preserved_rebuild: 0
109     # Run emerge of preserved libs after build
110     skip_sync: 1
111     # skip portage sync
112     webrsync: 1
113     # uses webrsync instead of emerge --sync
114     remote_overlay:
115       - git://github.com/my/repo
116       - myoverlay|git|https://github.com/foo/bar
117     remove_remote_overlay:
118       - myoverlay
119     remove_layman_overlay:
120       - plab
121       - foobar
122     # Specify a list of portage overlays
123     # that will be added before the build.
124     # You can also remove them,
125     # please always use the name of the overlay
          name.
126     remove:
127       - app-misc/foo
128     # Remove an atom with emerge
```

Listing A.1: Esempio completo di build.yaml con tutte le opzioni sviluppate

Bibliografia

[1] L. Torvalds and D. Read By-Diamond, *Just for fun: The story of an accidental revolutionary*. Harper Audio, 2001.

[2] W3Cook, "OS Market share and usage trends.." `http://www.w3cook.com/os/summary/`. Accessed: 2016-05-30.

[3] R. Di Cosmo, S. Zacchiroli, and P. Trezentos, "Package upgrades in foss distributions: Details and challenges," in *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, p. 7, ACM, 2008.

[4] G. G. P. License, "Gnu general public license," *Retrieved December*, vol. 25, p. 2014, 1989.

[5] DistroWatch, "Classifica distribuzioni linux.." `http://distrowatch.com/dwres.php?resource=popularity`. Accessed: 2016-05-30.

[6] J. M. González-Barahona, G. Robles, M. O. Pérez, L. Rodero-Merino, J. C. González, V. Matellan-Olivera, E. Castro-Barbero, and P. de-las Heras-Quirós, "Analyzing the anatomy of gnu/linux distributions: methodology and case studies (red hat and debian)," *Free/Open Source Software Development*, pp. 27–58, 2004.

[7] I. Red Hat, "Red hat history." `http://www.redhat.com/en/about/company`. Accessed: 2016-05-19.

[8] J. Markoff, “I, robot: The man behind the google phone,” *New York Times*, 2007.

[9] P. Alto, “Google’s android becomes the world’s leading smart phone platform,” 2011.

[10] P. Alto, “64 million smart phones shipped worldwide in 2006 - over 20 million converged devices ship in final quarter of year,” 2007.

[11] S. Pichai and L. Upson, “Introducing the google chrome os,” *The Official Google Blog*, vol. 7, 2009.

[12] T. V. Tom Warren, “Chromebooks outsold macs for the first time in the us.” `http://www.theverge.com/2016/5/19/11711714/chromebooks-outsold-macs-us-idc-figures`. Accessed: 2016-05-19.

[13] E. Raymond, “The cathedral and the bazaar,” *Knowledge, Technology & Policy*, vol. 12, no. 3, pp. 23–49, 1999.

[14] P. Mell and T. Grance, “The nist definition of cloud computing,” 2011.

[15] F. Erich, C. Amrit, and M. Daneva, “Cooperation between information system development and operations: a literature review,” in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, p. 69, ACM, 2014.

[16] C. Collberg, T. Proebsting, G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren, “Measuring reproducibility in computer systems research,” *Unpublished report. http://reproducibility. cs. arizona. edu (accessed 10 October 2014)*, 2014.

[17] S. Krishnamurthi, “Examining “reproducibility in computer science”.[accessed: 5. november 2014]. 2014,” *URL: http://cs. brown. edu/˜ sk/Memos/Examining-Reprod ucibility*.

[18] P. Trezentos, I. Lynce, and A. L. Oliveira, "Apt-pbo: Solving the software dependency problem using pseudo-boolean optimization," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, (New York, NY, USA), pp. 427–436, ACM, 2010.

[19] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "A look in the mirror: Attacks on package managers," in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 565–574, ACM, 2008.

[20] J. Roche, "Adopting devops practices in quality assurance," *Communications of the ACM*, vol. 56, no. 11, pp. 38–43, 2013.

[21] M. Httermann, *DevOps for developers.* Apress, 2012.

[22] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[23] R. Petersen, *Red Hat® Enterprise Linux & Fedora™ Core 4: The Complete Reference.* McGraw-Hill, Inc., 2005.

[24] J. D. Castro, "Introducing linux distros,"

[25] J. Mateos-Garcia and W. E. Steinmueller, "The institutions of open source software: Examining the debian community," *Information Economics and Policy*, vol. 20, no. 4, pp. 333–344, 2008.

[26] I. Murdock, "Overview of the debian gnu/linux system," *Linux Journal*, vol. 1994, no. 6es, p. 15, 1994.

[27] G. K. Thiruvathukal, "Gentoo linux: the next generation of linux," *Computing in Science and Engineering*, vol. 6, no. 5, pp. 66–74, 2004.

[28] X. Zheng, D. Zeng, H. Li, and F. Wang, "Analyzing open-source software systems as complex networks," *Physica A: Statistical Mechanics and its Applications*, vol. 387, no. 24, pp. 6190–6200, 2008.

[29] E. D. Giacinto, "Arm meet sabayon, sabayon meet arm.." `https://www.sabayon.org/article/arm-meet-sabayon-sabayon-meet-arm`. Accessed: 2016-04-30.

[30] H. K. W. Chan, A. J. Edwards, A. Srivastava, and H. H. Vo, "Shared library optimization for heterogeneous programs," Oct. 1 2002. US Patent 6,460,178.

[31] Canonical, "Universal "snap" packages launch on multiple linux distros." `https://insights.ubuntu.com/2016/06/14/universal-snap-packages-launch-on-multiple-linux-distros/`. Accessed: 2016-06-14.

[32] Flatpak, "Announcing flatpak – next generation linux applications." `http://flatpak.org/press/2016-06-21-flatpak-released.html`. Accessed: 2016-06-22.

[33] A. Team, "Linux apps that run anywhere." `http://appimage.org/`. Accessed: 2016-06-25.

[34] t. H. Lucian Armasu, "Flatpak is the universal linux packaging format that puts security first." `http://www.tomshardware.com/news/flatpack-universal-linux-packaging-format,32137.html`. Accessed: 2016-06-22.

[35] "Open build service documentation materials." `http://openbuildservice.org/help/materials/`. Accessed: 2016-05-25.

[36] "Open build service." `http://openbuildservice.org/`. Accessed: 2016-05-25.

[37] "Catalyst. gentoo wiki." `https://wiki.gentoo.org/wiki/Catalyst`. Accessed: 2016-05-25.

[38] "Metro. funtoo wiki." `http://www.funtoo.org/Metro`. Accessed: 2016-05-25.

[39] D. Liu and L. Zhao, "The research and implementation of cloud computing platform based on docker," in *Wavelet Active Media Technology and Information Processing (ICCWAMTIP), 2014 11th International Computer Conference on*, pp. 475–478, IEEE, 2014.

[40] T. Bui, "Analysis of docker security," *arXiv preprint arXiv:1501.02967*, 2015.

[41] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[42] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[43] Docker, "Understanding docker." `https://docs.docker.com/v1.8/introduction/understanding-docker/`. Accessed: 2016-03-02.

[44] Docker, "Storage driver: Images and containers." `https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/`. Accessed: 2016-03-02.

[45] O. Tange *et al.*, "Gnu parallel-the command-line power tool," *The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, 2011.

Ringraziamenti

Ringrazierei tutti, dal primo all'ultimo, ma sarebbe impossibile farlo in un libro "discreto", rasenterei le 1000 pagine con estrema facilità - quindi sarò breve, promesso.

Innanzitutto ringrazio i miei genitori, Teresa e Timoteo che con immensa pazienza hanno creduto in me e mi hanno sostenuto anche economicamente nel corso dei miei studi (i nomi sono in ordine alfabetico, così non bisticciate per l'ordine, erroneamente associato all'importanza). Ringrazio la mia famiglia estesa, i miei zii e le mie cuginette, la nonna e tutti i miei parenti, che da sempre mi hanno sostenuto anche nei momenti di difficoltà.

Grazie ai miei colleghi e i miei amici che mi hanno aiutato nella mia vita, anche con intermezzi goliardici; che vita sarebbe senza un sorriso (e qui, cari amici miei, vorrei davvero scrivervi tutti, ma non saprei nemmeno da dove iniziare). Ringrazio anche i docenti che mi hanno seguito e aiutato nel mio percorso di studi, grazie mille per il vostro sostegno. Grazie anche alla comunità *Open Source* di Sabayon e al team di sviluppo, senza di voi questo lavoro non sarebbe mai stato possibile.

Infine non posso non ringraziare la mia ragazza, Lucia, che di proposito cito per ultima - solo per infastidirla un pò - grazie per la tua pazienza e per il tuo sostegno, in ogni momento ho potuto contare su di te e tu . . . non hai mai chiesto nulla, a parte il caffè di mattina; grazie anche alla tua famiglia, che stoicamente sopporta il tuo ragazzo.