Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Rootless V^2CI
Continuous Integration di progetti V^2 tramite cross-compilazione non privilegiata e integrazione con stack ELK

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Francesco Ciofini

III Sessione
Anno Accademico 2024/2025

"Perciò chiunque ascolta queste mie parole e le mette in pratica, è simile a un uomo saggio che ha costruito la sua casa sulla roccia. Cadde la pioggia, strariparono i fiumi, soffiarono i venti e si abbatterono su quella casa, ed essa non cadde, perché era fondata sopra la roccia."

Introduzione

L'ubiquità dei sistemi di calcolo moderni e l'eterogeneità delle architetture hardware su cui essi operano impongono, nel panorama dello sviluppo software contemporaneo, sfide sempre più complesse legate alla portabilità e alla distribuzione del codice. Il Progetto Debian, da decenni pilastro della comunità open source, fonda la propria filosofia sulla garanzia di un sistema operativo universale, capace di adattarsi a molteplici piattaforme hardware mantenendo inalterati standard di stabilità e sicurezza. In parallelo, il laboratorio e la comunità di Virtualsquare (V^2) perseguono l'ambizioso obiettivo di democratizzare la virtualizzazione, offrendo strumenti che permettano la creazione di reti e macchine virtuali in contesti privi di privilegi amministrativi.

Il presente lavoro di tesi si colloca esattamente nell'intersezione tra queste due filosofie, nascendo dall'esigenza concreta di estendere l'accessibilità del progetto sshlirp - un tool innovativo per la connettività TCP/IP in user-space - a un vasto spettro di architetture, dai comuni amd64 e arm64, a port più specifici come riscv64.
La necessità di distribuire binari statici pronti all'uso per tali architetture, destinati a operare su host remoti eterogenei, ha evidenziato le limitazioni degli approcci tradizionali alla cross-compilazione, spesso vincolati da pesanti emulazioni di sistema, dipendenze da privilegi root o infrastrutture cloud.

L'elaborato descrive quindi il percorso evolutivo che ha condotto alla progettazione e all'implementazione di Rootless V^2CI (*Rootless Virtual Square Continuous Integration*), un motore di integrazione continua distribuito, altamente configurabile e nativamente indipendente da privilegi elevati. Tale sistema non si limita alla mera automazione della cross-compilazione, ma rappresenta anche un ecosistema completo in grado di gestire il ciclo di vita dei binari, dalla risoluzione delle dipendenze in ambienti isolati alla loro rotazione e archiviazione, garantendo persistenza, idempotenza e sicurezza.

La trattazione dello sviluppo che ha portato a questo risultato è strutturata in un percorso incrementale che rispecchia le fasi di ricerca e implementazione affrontate.

Nel primo capitolo verrà delineato il contesto tecnologico di origine, introducendo l'ecosistema Virtualsquare e il progetto sshlirp. Verranno inoltre analizzate le motivazioni che rendono necessaria una distribuzione capillare e multi-architettura di tale strumento, ponendo le basi per i requisiti del progetto.

Il secondo capitolo invece documenterà le fasi embrionali del lavoro, focalizzate sulla pacchettizzazione Debian manuale. Saranno analizzate le criticità emerse dall'uso di Qemu System Emulation per architetture obsolete e la transizione verso un approccio più snello basato su Qemu User-Mode Emulation combinato a debootstrap. Si argomenterà inoltre la scelta di prediligere una soluzione di build locale rispetto alle pipeline CI cloud-based (come Salsa CI), in virtù di una maggiore necessità di controllo sulle risorse e persistenza degli ambienti di compilazione.

Il terzo capitolo illustrerà il fondamentale passaggio dallo sviluppo manuale all'automazione, attraverso l'analisi dei prototipi sshlirpCI e Rootless sshlirpCI. In questa sede verranno esposte le soluzioni tecniche adottate per abbattere il vincolo dei privilegi, sfruttando primitive del kernel Linux quali gli User Namespaces e l'astrazione dei permessi fornita da fakeroot. Si analizzeranno inoltre le vulnerabilità di sicurezza intrinseche all'uso di `sudo` e `system()` e come queste siano state superate.

Infine il quarto e ultimo capitolo è dedicato alla maturazione finale del progetto in Rootless V^2CI. Di questo se ne descriverà l'architettura finale, caratterizzata da un approccio concorrente che vede demoni indipendenti per ogni progetto coordinare thread di compilazione specifici per architettura. Verranno quindi approfonditi aspetti implementativi cruciali quali la gestione della concorrenza su risorse condivise, il meccanismo di disaster recovery per la resilienza dei rootfs, e le politiche di rotazione dei binari. Infine, si presenterà l'integrazione del motore di CI con uno stack ELK (Elasticsearch, Logstash, Kibana) containerizzato, progettato per offrire una monitorabilità centralizzata e granulare dei processi di build distribuiti, trasformando log grezzi in dashboard visuali intuitive.

Rootless V^2CI si propone dunque non solo come soluzione tecnica a un problema di ingegneria del software, ma come dimostrazione accademica di come sia possibile orchestrare processi complessi, e solitamente privilegiati, in user-space, coniugando efficienza computazionale, sicurezza operativa e usabilità.

Indice

Capitolo 1

Contesto di origine: Virtualsquare e sshlirp

In questo primo capitolo si introdurrà il contesto di nascita di Rootless V^2CI, illustrandone motivazioni e necessità madri e collocando queste all'interno del più ampio scenario dello sviluppo software per progetti di virtualizzazione proposto da Virtualsquare.
In particolare si illustrerà la principale origine dello sviluppo del progetto di tesi: sshlirp. Di questo si esporrà il processo di nascita, le caratteristiche e le funzionalità che ne permettono l'integrazione all'interno dell'architettura Virtualsquare. Inoltre di quest'ultima verranno descritti gli scopi, i principi e la struttura generale, con particolare attenzione al suo sottoinsieme di componenti che interagiscono con sshlirp e quindi per i quali il progetto di tesi è stato originariamente pensato.

L'intento iniziale di Rootless V^2CI infatti era quello di dare un contributo al progetto Virtualsquare attraverso il rilascio di pacchetti multi-architettura di sshlirp.

1.1 Virtualsquare

Virtualquare è definibile e pensabile come un container di progetti, tool e librerie il cui scopo è quello di permettere la creazione di un ambiente virtuale unificato in cui sia garantita la comunicazione e l'interazione tra vari componenti quali macchine virtuali, sistemi operativi in user space e stack di rete [1].
Al coltempo Virtualsquare rappresenta una comunità open source e un concetto accademico originato all'interno dell'Università di Bologna, grazie alla ricerca e allo sviluppo del fondatore Renzo Davoli.

1.1.1 Origini di Virtualsquare

L'ambiziosa idea alla base di Virtualsquare fonda le sue origini, nel 2004, sulla necessità accademica di permettere agli studenti universitari del corso di *Laboratorio di Sistemi Operativi* dell'Università di Bologna, di sperimentare, amministrare e configurare i propri sistemi operativi in un ambiente virtuale in cui non fossero richiesti privilegi root e da cui, allo stesso tempo, fosse possibile l'accesso a risorse di rete reali [2].

Originariamente infatti un sistema Virtual Square (V^2) era pensato come un insieme di macchine virtuali connesse tramite reti virtuali e primariamente come un'architettura governata da tre caratteristiche fondanti [2]:

- **Coerenza di emulazione**: il sistema virtuale nel suo insieme doveva comportarsi e apparire a tutti gli effetti come un insieme di host e connessioni di rete reali, in cui lo strato di virtualizzazione aggiunto causava al più un overhead prestazionale che si traduceva in un apparente rallentamento dei device;

- **Possibilità di integrazione o isolamento**: i vari componenti dell'ecosistema V^2 erano pensati sia per comunicare con le reti e i sistemi reali sottostanti in modo da poter essere integrati in essi consentendo l'accesso a risorse esterne, che per essere isolati completamente da essi;

- **Sicurezza**: macchine e reti Virtual Square erano progettate per eseguire come normali processi utente non privilegiati e per cui la garanzia di corretta comunicazione con il sistema sottostante era data al più da configurazioni root-required sull'host stesso.

Tali regole erano in principio implementate tramite un'ampia infrastruttura che vedeva l'uso di host V^2 dedicati e di tool per la creazione di reti virtuali.

Host V^2

I nodi del network Virtual Square potevano essere implementati tramite vari tipi di VMs, tra cui [2]:

- **User-Mode Linux**: questo emulatore non è altro che un kernel Linux ricompilato per eseguire come processo utente e per interfacciarsi con le risorse del sistema host tramite sole system calls; questo tipo di mappatura virtuale

accesso hardware $\longleftrightarrow$ *system calls* garantisce il funzionamento del kernel virtuale di UML completamente in user-space, senza la necessità di privilegi di root [2, 3].

- **Qemu**: un emulatore di macchine open source impiegato sia per l'emulazione completa di architetture hardware (*System Emulation*) che per l'esecuzione di singoli programmi compilati per architetture diverse da quella dell'host (*User-mode Emulation*), con conseguente emulazione della CPU. Il cuore di esecuzione di Qemu è il *Tiny Code Generator* (TCG), un traduttore dinamico di codice che converte le istruzioni della CPU guest in istruzioni della CPU host, impiegato sia in modalità System Emulation che in User-mode Emulation. Di conseguenza, preso singolarmente, Qemu è classificabile come un emulatore puro e perciò come un hypervisor di tipo 2 (*hosted*). In caso di integrazione con KVM invece, Qemu permette di accedere a funzionalità di virtualizzazione hardware rientrando nella categoria degli hypervisor di tipo 1 (*bare-metal*) e permettendo full virtualization. Ciò, d'altro canto, esclude la possibilità emulazione di CPU diverse da quella dell'host [4, 5]. L'impiego di Qemu nello sviluppo del progetto Rootless V^2CI ha avuto un ruolo decisivo.

- **Bochs**: questo storico emulatore puro fornisce un sistema virtuale completo i386, disponibile su più piattaforme ed eseguito in user-space. Essendo fondato su tecniche di interpretazione delle istruzioni CPU standard, Bochs pecca di prestazioni rispetto alle soluzioni più moderne.

- **PearPC**: un emulatore analogo a Bochs impiegato però per l'emulazione di architettura PPC.

- **MPS/μMPS**: sviluppato per scopi accademici, anche questo host V^2 è un sistema virtuale completo, minimale e leggero.

Reti V^2

Tra i tool di networking impiegati per la creazione di reti virtuali V^2 vi erano [2]:

- **VDE Virtual Distributed Ethernet**: un'intera rete virtuale a user-level in grado di instradare pacchetti ethernet tra macchine virtuali sullo stesso host, macchine virtuali distribuite, sistemi operativi e, con semplici configurazioni di rete root-required per l'utilizzo di interfacce di rete tuntap, anche su host

reali tramite l'uso di switch, hubs e cavi virtuali. I nodi della rete VDE erano collegati a questi componenti tramite i `vde_plugs`. Originariamente gli host virtuali supportati da VDE erano UML, Qemu, Bochs e MPS/μMPS. Successivamente è stato aggiunto supporto anche per Virtualbox [1].

- **Supporto kernel TUN/TAP**: questi device virtuali di rete, implementati come moduli kernel, permettono la creazione di interfacce di rete virtuali che operano a livello di link layer (TAP) o di network layer (TUN) e che, anzichè inviare i pacchetti alla scheda di rete fisica dell'host, li reindirizzano a processi utente.

- **Slirp**: un tool di rete originariamente pensato per permettere a utenti non privilegiati di accedere da host reali a reti esterne tramite connessione PPP simulata sopra una semplice connessione da terminale. Slirp, isolando quindi il traffico di rete del client da quello del server, anticipava già nel 1995 il concetto di NAT [6] e acquisiva proprietà che gli avrebbero concesso in un secondo momento di essere impiegato come stack di rete user-space per macchine virtuali. È importante sottolineare infine che da slirp è nato, nel Febbraio 2024 grazie alle idee e allo sviluppo di Renzo Davoli e del team Virtualsquare, il successivo progetto `sshlirp` [7], da cui Rootless V^2CI prende origine.

1.1.2 Virtualsquare oggi

Dopo più di 20 anni di ricerca e sviluppo, Virtualsquare non è più solamente un insieme di host e connessioni virtuali pensate per scopi didattici, ma rappresenta un ecosistema completo di tool e librerie open source pensate per la creazione di interi ambienti virtuali complessi e personalizzabili.
Sebbene infatti i principi esposti nella precedente sezione siano rimasti le fondamenta delle ambizioni Virtualsquare, i progetti esistenti si sono estesi e nuovi tool sono stati sviluppati per permettere la creazione di ambienti virtuali sempre più complessi e realistici. Questi obbiettivi appunto sono stati raggiunti attraverso un lungo processo evolutivo diretto da linee guida di scalabilità, quali [8]:

- Re-use di tool esistenti;
- Modularità e compatibilità;
- Nessun vincolo di architettura richiesta.

In particolare, VDE ha visto l'integrazione di funzionalità innovative e il livello di virtualizzione conseguibile dagli strumenti Virtual Square è stato dilatato a livello

di sistema operativo, grazie allo sviluppo di `VUOS`, e sul piano di networking a livello applicativo, per mezzo di `IoTh`.

Gli sviluppi di VDE

I principali progressi di VDE hanno coinvolto aspetti come:

- **Compatibilità di vde_plug**: come anticipato nella sezione 1.1.1, la possibilità di impiegare il concetto di VDE del *plug* - che nel tempo è diventato fondante - si è estesa a più host virtuali (tra cui VirtualBox) e a nuovi programmi che implementano stack di rete; inoltre lo stesso `vde_plug` è stato integrato nella più estesa prospettiva di `vdeplug4`, una libreria `C` che fornisce nativamente plug-ins sotto forma di librerie dinamiche per la connessione a un ampio set di reti virtuali: `vde`, `ptp`, `tap`, `vxlan`, `vxvde` e `udp`; quest'ultime vengono identificate dal plug tramite il cosidetto *Virtual Network Locator* (VNL), una stringa che identifica la risorsa virtuale come un URL è in grado di fare per una risorsa web [9].

- **Isolamento e virtualizzazione di rete tramite namespace**: grazie allo sviluppo del nuovo tool `vdens`, VDE ha acquisito un'ulteriore caratteristica di isolamento e virtualizzazione di rete, permettendo, tramite l'uso di network namespaces, la creazione di reti virtuali completamente isolate da quella dell'host e collegabili a una rete VDE esistente tramite un VNL condiviso [1, 10].

- **Scalabilità e distribuibilità con vxvde**: attraverso il plug-in `libvdeplug_vxvde`, è stata aggiunta la possibilità di connettere nodi VDE posizionati su host disgiunti - sebbene connessi alla medesima LAN - a reti distribuite `vxvde`. Tale tool permette quindi di scalare orizzontalmente le reti virtuali VDE, estendendone la portata oltre il singolo host fisico, senza la necessità di host dedicati che espongano switch VDE e implementando quindi il concetto di *Local Area Cloud* [1, 11].

- **Connettività tramite libslirp**: come anticipato nella sezione 1.1.1, l'impiego di slirp si è integrato nell'ecosistema VDE grazie al modulo `libvdeplug_slirp`, che ha permesso l'uso di slirp come stack di rete user-space e router fornitore di NAT, DHCP, DNS e port forwarding che desse connettività TCP/IP esterna a nodi VDE, grazie alla libreria interna `libslirp` (e alla sua versione più recente `libvdeslirp`) [1, 12, 13]. Sarà poi da questa

libreria che nascerà il progetto `sshlirp`, su cui fonderà le proprie origini lo stesso Rootless V^2CI.

- **Incapsulamento e sicurezza con plug-ins innestati**: un'ulteriore avanzamento per VDE è stato lo sviluppo di plug-ins come `vdeplug_agno` e `vdeplug_vlan`, che permettessero di sommare rispettivamente incapsulamento crittografico e tagging VLAN al normale traffico VDE, garantendo così sicurezza e segregazione del traffico all'interno delle reti virtuali [1].

VUOS

Una delle evoluzioni più importanti per il progetto Virtualsquare - sebbene secondaria per lo sviluppo di Rootless V^2CI - è stata la creazione di `VUOS`. Questo kernel modulare e configurabile esegue in user-space, ricordando i concetti introdotti da UML e introducendo al coltempo innovazione attraverso l'idea di "vista (view)" per processo o thread. `VUOS` infatti è interpretabile non solo come filtro e forwarder di system calls - grazie al suo hypervisor `umvu` - ma anche come un sistema di gestione delle risorse che permette di definire viste personalizzate per ogni processo o thread in esecuzione che si intende virtualizzare [1]. Infatti, per mezzo di moduli quali `vufs`, `vufuse`, `vudev` e `vunet`, `VUOS` aggiunge uno strato di virtualizzazione tra il processo utente e il kernel host.

IoTh

Un'altra espansione dell'ecosistema Virtualsquare ha visto l'implementazione dell'innovativa idea di conferire il ruolo di nodo Internet a singoli processi, o addirittura thread, utente. Questo cambio di prospettiva distribuisce i poteri di virtualizzazione di rete concessi dai tool VDE a livello applicativo: un processo utente ha ora accesso a uno o più stack di rete configurabili [1].
Anche questo sviluppo non è direttamente collegato a Rootless V^2CI ma rappresenta comunque un passo avanti verso lo scopo comune di rendere accessibili a utenti e processi non privilegiati funzionalità di virtualizzazione e networking avanzate.

1.2 sshlirp

Come accennato in precedenza, `sshlirp` costituisce il principio di avvio del processo di sviluppo ed evoluzione che ha portato alla realizzazione di Rootless V^2CI.

Questa tecnologia pone le sue origini nel progetto `slirp`, di cui si sono già introdotti gli scopi.

Slirp: origini e funzionamento

Slirp nasce il 30 Marzo 1995 grazie allo sviluppo di Danny Gasparovski [6], e il suo scopo era intrinsicamente legato al momento storico delle sue origini.
L'accesso a Internet era principalmente garantito tramite l'uso di terminali ed era quindi comune connettersi ai propri account shell su server remoti per avere accesso alla rete esterna [14]. Ciò chiaramente non permetteva ai client di avere un link diretto alla rete né, conseguentemente, di utilizzare processi TCP/IP direttamente sul proprio host.
Una semplice connessione PPP (Point to Point Protocol) invece integrava nativamente tali servizi e, se il corrispondente ISP lo prevedeva, poteva essere usata per accedere a Internet con un proprio indirizzo IP pubblico, permettendo traffico in entrata senza overhead di NAT o port forwarding.
D'altro canto, l'uso di PPP prevedeva costi maggiori rispetto a una semplice connessione terminale [14, 15].

In tale contesto storico slirp si proponeva come un tool in grado di simulare una connessione PPP sopra una semplice connessione terminale, permettendo così a utenti non privilegiati di avere accesso a servizi TCP/IP senza la necessità di sostenere i costi di un vero e proprio link PPP.

In generale un emulatore SLIP/PPP come slirp esegue soltanto sull'host remoto su cui è quindi richiesta l'installazione dei protocolli TCP/IP. L'host locale invece, affinché possa comunicare correttamente con l'emulatore, deve integrare nel suo kernel il protocollo PPP [16]. Dati questi requisiti, il client, una volta configurato il ppp daemon (`pppd`) affinché utilizzi il canale shell come link di rete, può effettuare richieste PPP con la garanzia che l'emulatore sarà in grado di catturare e inviare correttamente i pacchetti fuori sulla Rete. I dati in ingresso invece, con un processo del tutto simmetrico, vengono catturati dall'emulatore sull'host remoto che si occuperà di inviarli sul collegamento SLIP/PPP simulato al client. In questo modo la Rete continuerà a percepire il traffico rete del client come proveniente dal modem ospitante l'account shell remoto, mentre il client potrà utilizzare servizi TCP/IP come se fosse connesso tramite un vero link PPP [16].

Chiaramente questo approccio, per quanto innovativo ed economico per l'epoca, presentava delle grosse limitazioni rispetto a un vero SLIP/PPP link. Ad esempio il traffico di rete del client, essendo incapsulato all'interno di una connessione terminale, non poteva beneficiare delle ottimizzazioni e delle funzionalità di sicurezza offerte da un vero link PPP.
Una connessione telnet-like implicava infatti che ogni dato venisse trasmesso in chiaro, esponendo così il traffico di rete del client a potenziali intercettazioni [16].
L'altro profondo svantaggio di un emulatore rispetto un vero SLIP/PPP è che al client non veniva assegnato nessun indirizzo IP univoco, rendendo impossibile l'instradamento di traffico in entrata direttamente verso di esso e aggiungendo quindi un overhead non indifferente [16].

VDE slirp e libslirp

Nonostante le limitazioni appena esposte, l'idea di slirp di fornire connettività TCP/IP a utenti non privilegiati ha ispirato, come anticipato nella sezione 1.1.1, lo sviluppo di `libslirp` e della sua versione più recente `libvdeslirp`, mirati a essere impiegati come stack di rete user-space per nodi VDE.[1].

sshlirp

Dalla fusione delle idee che hanno portato prima alla creazione di slirp, poi allo sviluppo di VDE e infine di libvdeslirp, nasce sshlirp. Questo semplice tool, sviluppato a partire dal Febbraio 2024 da Renzo Davoli, unisce [7, 17]:

- l'esigenza di connettività TCP/IP user-space grazie a connessioni tramite account shell a host remoti, alla base dell'ambizione di slirp;
- la possibilità di connettere nodi VDE alla rete reale, attraverso socket user-space, introdotta da VDE attraverso libvdeslirp;
- diffusione, sicurezza e tunnelling SSH per l'accesso remoto a host shell;

sshlirp infatti si basa sull'idea di, dato un tunnel SSH tra un nodo VDE e un host remoto, e traffico TCP/IP del nodo incapsulato e mascherato sul canale di `STDIN/STDOUT` fornito da SSH, decapsulare i pacchetti ricevuti attraverso libvdeslirp, reindirizzarli allo stack di libslirp e, grazie a quest'ultimo, aprire socket user-space per comunicare con la rete esterna.
Questo processo permette quindi a un nodo VDE non solo di avere connettività TCP/IP esterna senza la necessità di privilegi di root, ma anche di usufruire della

sicurezza e del tunnelling SSH per un servizio sia di NAT che di VPN [17].

L'uso tipico di sshlirp, che solitamente prevede la creazione di un network namespace tramite `vdens` sul client VDE, è intrinsecamente vincolato alla disponibilità del binario di `sshlirp`, pronto all'uso, sull'host remoto.

1.3 Le origini di Rootless V^2CI: il contributo a sshlirp

Oggi Rootless V^2CI si propone come strumento altamente configurabile, accessibile e facilmente monitorabile, per l'automazione del processo di build in user space non privilegiato di binari statici cross-compilati per architetture multiple e progetti multipli.
Le sue origini però incarnavano una semplice soluzione alle più specifiche esigenze dell'utilizzo di sshlirp, esposte nella precedente sezione.
Nato infatti come progetto di pacchettizzazione per il corso di *Sistemi Virtuali* dell'Università di Bologna, ha visto un eterogeneo processo di sviluppo che lo ha portato dal semplice scopo di creare pacchetti multi-architettura di sshlirp, ufficiali e rilasciabili attraverso un package manager affinché fossero facilmente installabili su host remoti, all'idea di un sistema di CI per sshlirp, mirato alla costruzione rootless di binari pronti alla copia e all'uso da remoto, fino all'ambizione odierna di un tool generalizzato e user-friendly per la consegna di binari cross-compilati statici aggiornati per progetti multipli.

Rootless V^2CI dà quindi corpo all'ambizione di contribuire, con uno sviluppo open source regolato da licenza GPL-2.0, alla trasmissione e diffusione dei progetti Virtualsquare, permettendone l'utilizzo distribuito su host remoti e agevolandone di conseguenza l'espansione da sistema virtuale locale a infrastruttura virtuale decentralizzata, dandone un primo esempio con la sua comprovata applicabilità al progetto di sshlirp.

Capitolo 2

Pacchettizzazione Debian di sshlirp

La prima fase di sviluppo di Rootless V^2CI prende origine, come anticipato, dalla semplice idea di costruire pacchetti `.deb` del progetto di `sshlirp`, per tutti i principali port Debian, in modo che fosse conseguentemente possibile distribuirli attraverso un package manager ufficiale e poi installarli facilmente su qualsiasi host remoto, su cui si volesse utilizzare sshlirp.
Per il raggiungimento di questo scopo è stato quindi avviato un processo di pacchettizzazione Debian manuale e in locale, che si è basato principalmente sull'utilizzo di `Debian Salsa GitLab`, `qemu` e `gbp`.
Tale processo è stato preferito all'utilizzo di pipeline CI standard e distribuite su cloud per le motivazioni che verranno esposte nella sezione 2.3.
In questo primo approccio, del tutto "manuale", la priorità è stata centrata sulla possibilità di una cross-compilazione e di una build dei pacchetti facilmente ripetibile. Si sono quindi acerbamente impiegati tool standard e di facile utilizzo, trascurando approcci più complessi che avrebbero invece garantito il funzionamento anche per utenti non privilegiati.

Al fine di riassumere e illustrare questo percorso di pacchettizzazione Debian di sshlirp, è stato realizzato il seguente schema di flusso che include i principali passaggi di ricerca teorica e sviluppo che hanno condotto dall'obbiettivo iniziale al raggiungimento di una soluzione ottimizzata.

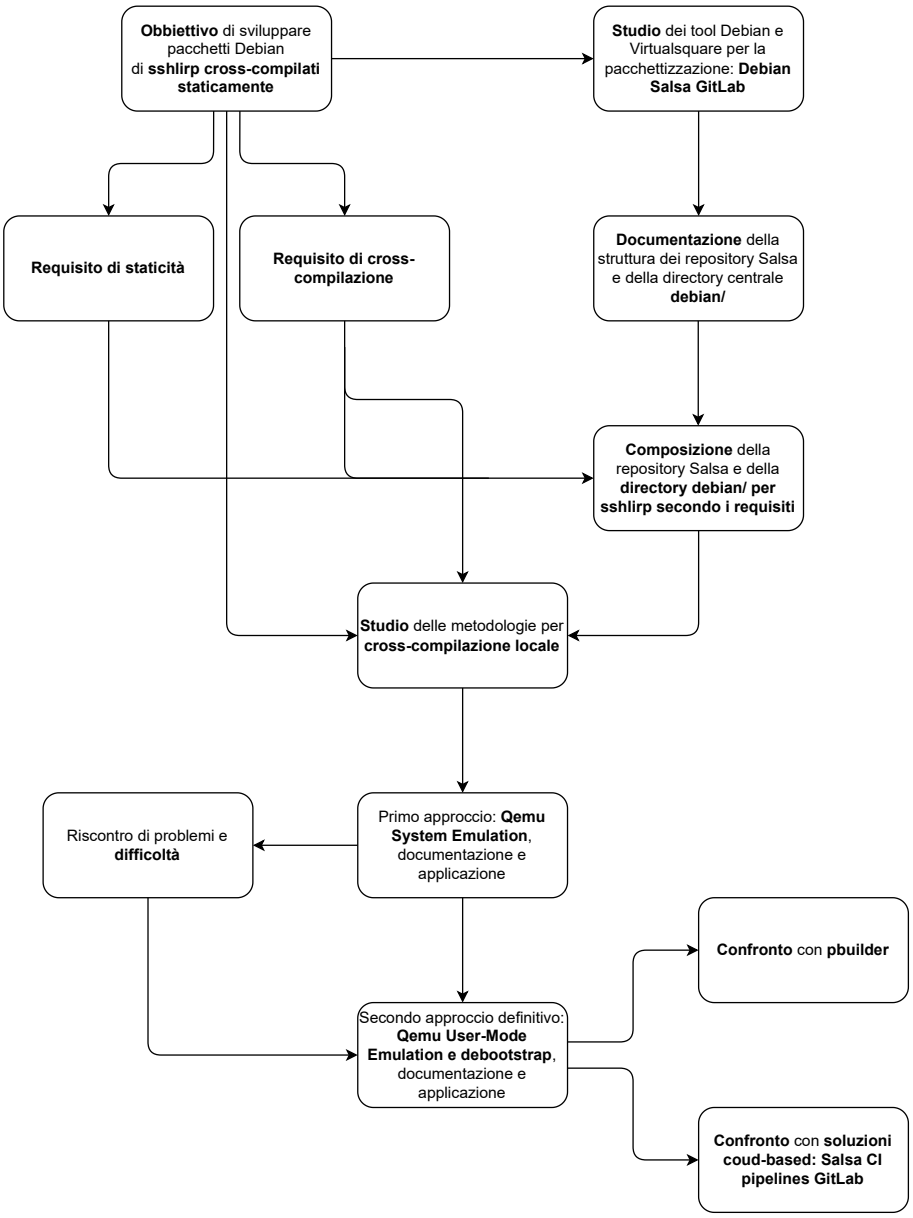

Figura 2.1: Schema di flusso del processo di pacchettizzazione Debian di sshlirp

2.1 Impiego di Debian Salsa GitLab

Come già fatto per alcuni degli altri progetti firmati Virtualsquare [1], anche per la distribuzione dei pacchetti di sshlirp si è scelto di impiegare Debian Salsa GitLab. Questo servizio di hosting GitLab, fornito dalla Debian Foundation, permette la creazione di repository Git pubblici e privati per progetti open source legati a Debian, e offre inoltre un'infrastruttura CI/CD integrata per l'automazione di build, test e deployment di pacchetti Debian [18].

Nel caso specifico di sshlirp, Debian Salsa GitLab è stato impiegato principalmente come repository remoto per il codice sorgente e per la directory `debian/` di pacchettizzazione, seguendo le direttive Virtualsquare [1]. Ciò ha permesso che il processo di build incrociata dei pacchetti potesse essere eseguito in locale tramite l'uso, nativamente complementare a Debian Salsa [18], di `gbp`.

2.1.1 Introduzione a Debian Salsa

Salsa, successore di `git.debian.org` e `Alioth` [19], come anticipato è l'istanza GitLab ufficiale di Debian, ossia il server Git per lo sviluppo collaborativo dei pacchetti Debian. Si tratta di una piattaforma su cui i manutentori Debian collaborano, versionano i sorgenti dei pacchetti, gestiscono merge request, CI, test e altre attività legate al processo di pacchettizzazione [18, 19].
Essendo basata su GitLab Community Edition, fornisce inoltre molte delle caratteristiche standard di GitLab (repository Git, merge request, issue), integrandole con componenti specifici per Debian, come Salsa CI, il sistema di continuous integration custom per i pacchetti Debian, attraverso la quale i manutentori possono automatizzare la compilazione su più architetture [19].

Nel contesto della pacchettizzazione Virtualsquare, Debian Salsa è impiegata principalmente come servizio di hosting per i repository Git dei progetti, a partire dai quali si esegue la build locale dei pacchetti Debian con `gbp`.
Sebbene questa scelta escluda l'uso delle pipeline CI integrate di Salsa, essa permette di mantenere il controllo completo sul processo di build e di eseguirlo in ambienti personalizzati, come verrà illustrato nella sezione 2.3.

In generale lo sviluppo Virtualsquare basa il suo processo di distribuzione e diffusione sull'uso di Debian Salsa GitLab secondo le linee guida esposte dalla Debian Policy, dalla Debian wiki e nei documenti DEP [1, 21, 22, 23].

Come viene usata Debian Salsa nel progetto Virtualsquare

Un repository Virtualsquare su Debian Salsa GitLab è strutturato in tre branch principali [1]:

- **upstream**: questo branch contiene il codice sorgente originale del progetto, ereditabile dal repository ufficiale di sviluppo (che solitamente è locato su GitHub per i progetti Virtualsquare) grazie a un suo puntatore URL

(`Repository`) che può essere specificato nel file `upstream/metadata`; infatti questo branch è pensato per essere aggiornato automaticamente dai sorgenti originali, senza commit manuali [19];

- **debian/sid**: questo branch definisce, in aggiunta al codice ereditato da `upstream`, la directory `debian`, contenente una serie di file che regolano il processo di pacchettizzazione Debian del progetto [1, 21, 23];

- **pristine-tar**: quest'ultimo branch (opzionale per Virtualsquare ma raccomandato sia dalla developers reference Debian che dai documenti DEP [23, 24]), contiene un archivio di file binari *delta* che descrivono come ricostruire il tarball dei sorgenti originale a partire dai file presenti nell'upstream branch; in sostanza, specificando l'opzione `--git-pristine-tar` durante l'uso di `git-buildpackage`, è possibile rigenerare il tarball "pulito" originale senza dover conservare una sua versione compressa nel repository Git [24].

I branch dei repository Virtualsquare hostati su Debian Salsa GitLab provengono da un flusso di lavoro standardizzato per lo sviluppo Virtualsquare, e conforme ai processi di pacchettizzazione Debian, che prevede un'esecuzione ordinata di operazioni preparatorie, compositorie, di debug, di test e infine di upload [1] quali:

1. Creazione di un repository Git pubblico su Debian Salsa GitLab;
2. Clone del repository ufficiale del progetto (solitamente su GitHub) in locale;
3. Creazione del branch `upstream` e push su Debian Salsa GitLab;
4. Aggiunta e push di tag di versione al branch `upstream`;
5. Creazione del branch `debian/sid` a partire da `upstream` e popolamento della directory `debian/` con i file di pacchettizzazione;
6. Debug e test della build dei pacchetti in locale con `gbp buildpackage -us -uc`;
7. Creazione del branch `pristine-tar` e popolamento con i file delta generati;
8. Push di tutti i branch su Debian Salsa GitLab;

Questo processo ordinato è stato meticolosamente seguito anche per la pacchettizzazione di sshlirp, la quale però, prevedendo il requisito aggiuntivo di cross-compilazione statica per tutti i port ufficiali Debian, ha richiesto ulteriori accorgimenti, per quanto riguarda la composizione della directory `debian`, come verrà esposto nella prossima sottosezione 2.1.2, e scelte progettuali addizionali per l'emulazione delle architetture target, descritte nella sezione 2.2.

2.1.2 Sviluppo della directory debian

I file interni alla directory `debian` che permettono a `gbp` di eseguire la build di pacchetti Debian sono molteplici. In particolare, quelli ritenuti fondamentali dalle linee guida Virtualsquare [1], e quindi impiegati anche per la pacchettizzazione di sshlirp, sono [22]:

- **changelog**: scritto in un formato specifico, questo file contiene la cronologia delle modifiche apportate al pacchetto, con dettagli su ogni versione rilasciata, come numero di versione, data di rilascio, autore e descrizione delle modifiche;
- **control**: questo file di testo definisce le informazioni essenziali del pacchetto, come nome, versione, mantenitore, dipendenze, descrizione e altri metadata; è cruciale per il corretto funzionamento del package manager durante l'installazione e la gestione del pacchetto;
- **copyright**: in questo file vengono specificate le informazioni sul copyright e la licenza del pacchetto, indicando i diritti d'uso, distribuzione e modifica sia del software originale del progetto che di quello aggiunto per il processo di pacchettizzazione;
- **rules**: questo file di testo, scritto in formato Makefile, contiene le istruzioni per la compilazione e l'installazione del pacchetto; definisce i comandi necessari per costruire il software dai sorgenti, installarlo temporaneamente in una directory di staging e prepararlo per la creazione del pacchetto finale;
- **source/format**: questo file specifica il formato del pacchetto sorgente Debian utilizzato, come `3.0 (quilt)`, che supporta l'uso di patch e altre funzionalità avanzate per la gestione dei sorgenti;
- **compat**: in questo file è reperibile la versione utilizzata di `debhelper`, un tool di supporto per la creazione dei pacchetti Debian, che implementa in modo standardizzato molte delle operazioni necessarie alla direzione del processo di build;
- **gbp.conf**: questo file di configurazione specifica le impostazioni per `git-buildpackage` attraverso diverse sezioni, di cui le rincipali sono [25]:
  - `[DEFAULT]`: questa sezione definisce opzioni che si applicano a tutti i comandi di gbp, a meno che non vengano sovrascritte dalle sezioni successive, specifiche per comando;

  - `[buildpackage]`: questa sezione sovrascrive le opzioni definite in `[DEFAULT]` per il comando `gbp buildpackage`;
  - `[import-orig]`: analogamente alla precedente, questa sezione sovrascrive le opzioni per il comando `gbp import-orig`.

- **watch**: questo file serve a monitorare gli aggiornamenti del software upstream in modo da poter verificare lo stato di sincronizzazione con le nuove releases su GitHub;

- **<bin>.install**: questi file - solitamente in numero corrispondente ai binari presenti nel pacchetto - specificano quali file devono essere installati e in quali directory target del pacchetto finale durante il processo di build; mentre il path di origine relativo alla directory di staging è sempre obbligatorio, il path di destinazione può essere omesso, nel qual caso viene "guessato" da `dh_install` [26].

Trascurando gli aspetti implementativi dei file `changelog`, `copyright`, `source/format`, `compat`, `gbp.conf` e `watch`, la cui composizione ha seguito le direttive standard della Debian Policy [22], con la sola aggiunta costumizzata di configurazioni specifiche per l'abilitazione del ramo `pristine-tar` (in particolare in `gbp.conf`), i file più rilevanti per la pacchettizzazione di sshlirp sono stati `control`, `rules` e `sshlirp.install`, che sono stati sviluppati tenendo conto dei requisiti specifici del progetto, quali la cross-compilazione per architetture multiple e la creazione di binari statici.

Requisito di cross-compilazione e sviluppo della directory debian

La necessità di ottenere pacchetti cross-compilati di sshlirp per tutti i port ufficiali Debian - ossia `amd64`, `arm64`, `armhf`, `armel`, `i386`, `ppc64el` e `s390x` - affinché avessero massimo grado di usabilità per lo scopo prefissato del progetto stesso, ha richiesto, a livello di sorgenti Debian, la sola accortezza di specificare correttamente l'architettura target nel file `control`.

In un contesto di cross-compilazione manuale infatti l'opzione `Architecture`, interna al file `control` e definita nella sezione `Package`, deve essere impostata al valore `any`, non solo per coerenza logica, ma anche per permettere al builder specificato in `gbp.conf` di riconoscere correttamente l'architettura target del pacchetto durante l'installazione senza necessità di cambiamenti ai sorgenti Debian [22].

Per questo motivo, la sezione `Package` del file `control` di sshlirp è stata sviluppata come segue [22]:

```
Package: sshlirp
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Description: sshlirp creates an "instant VPN"
    sshlirp converts a text based shell connection (e.g. ssh[1])
    into a VDE virtual private network.
```

Requisito di staticità dei binari e sviluppo directory debian

Come anticipato nella sezione 1.3, una delle features attese più importanti per i pacchetti di sshlirp consisteva nella staticità dei loro binari, anch'essa intrinsicamente legata allo scopo di diffusione e usabilità del progetto.
Questo requisito ha richiesto accortezze più sostanziali ai file della directory `debian`, derivate da uno studio preliminare delle dipendenze e del processo di compilazione di sshlirp previsto dal suo `CMakeLists.txt` originale.

Da questa analisi sono emersi tre punti fondamentali che hanno inciso successivamente sulla scrittura dei file Debian:

1. sshlirp dipende da `libglib2.0-dev`, `libpcre2-dev` e `libvdeslirp-dev` per la compilazione;
2. `libvdeslirp-dev` dipende dal pacchetto ufficiale `libslirp-dev` il quale, sebbene distribuito da Debian, anche nelle releases più recenti non contiene nativamente i file `.a`, ossia gli archivi necessari per il linking statico [27];
3. la compilazione di sshlirp con `make` genera due binari distinti: `sshlirp`, corrispondente al binario linkato dinamicamente, e `sshlirp-<arch>`, il binario linkato staticamente per l'architettura host.

Il primo punto ha semplicemente richiesto l'aggiunta delle dipendenze di build al file `control`, nella sezione `Build-Depends`, che ha quindi assunto la seguente forma:

```
Build-Depends: debhelper (>= 10), cmake, libslirp-dev,
  libglib2.0-dev, libpcre2-dev
```

Ciò ha garantito che gran parte delle dipendenze di build fossero installate automaticamente dal package manager prima dell'inizio del processo di compilazione [22], e

che fossero quindi resi disponibili i corrispondenti archivi statici necessari per la fase di linking di sshlirp.

Il secondo punto invece non ha influito direttamente sulla scrittura dei file Debian, quanto sulla fase di build vera e propria, rendendo necessaria la compilazione manuale e l'installazione preliminare di `libslirp` per ogni architettura target, in modo da ottenere gli archivi statici mancanti.
L'immediato effetto di questa necessità non è stato solo l'incremento di complessità nel processo di cross-build, come verrà esposto nella sezione 2.2, ma anche il conseguente bisogno di override delle direttive `dh_shlibdeps` di `deb_helper`.
Infatti, grazie a quanto emerso dal precedente studio del `CMakeLists.txt` di sshlirp e riportato nel terzo punto, è stato possibile compiere la seguente deduzione: dal momento che gli archivi di `libslirp` utilizzati sia per il linking dinamico che statico sarebbero stati quelli generati dalla compilazione manuale del progetto stesso, si sarebbe reso necessario evitare che `deb_helper`, durante la build dinamica, eseguisse il check automatico della provenienza delle *shared libraries* (ossia degli archivi dinamici `.so`) appunto attraverso `dh_shlibdeps` [22], che, in questo contesto specifico, sarebbe fallito a causa della "non tracciabilità" di `libslirp.so`.
Questo blocco funzionale a `deb_helper` è stato perciò implementato nel file `debian/rules` attraverso il seguente override:

```
override_dh_shlibdeps:
  dh_shlibdeps --dpkg-shlibdeps-params=--ignore-missing-info
```

Infine, il terzo punto, derivato dal `CMakeLists.txt`, ha influito sia sulla scrittura di `debian/rules` che di `debian/sshlirp.install`.
Per quanto riguarda le direttive di build, la conoscenza preliminare della generazione di due binari distinti ha permesso di escludere l'inserimento di flag per `cmake` che avrebbero forzato la compilazione statica globalmente.
Questi infatti, sebbene intuitivamente necessari e aderenti al requisito essenziale di staticità, avrebbero in realtà compromesso la generazione del target dinamico `sshlirp`, per il quale non era previsto, nel `add_executable` corrispondente del `CMakeLists.txt`, il linking di simboli `Sysprof`, da cui dipendeva l'esito della compilazione statica di `glib-2.0`, inserito comunque come target secondario.
Per questo motivo, il file `debian/rules` ha assunto la seguente struttura finale, conforme agli standard Virtualsquare [28]:

```
#!/usr/bin/make -f
```

```
# output every command that modifies files on the build system.
export DH_VERBOSE = 1

# Enable all hardening options
export DEB_BUILD_MAINT_OPTIONS = hardening=+all

# Compile with multiple jobs in parallel
export DEB_BUILD_OPTIONS = parallel=$(shell nproc)
# Instruct the linker not to include unnecessary shared libraries
export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed

# Variables for CMake
CMAKE_FLAGS = -DCMAKE_BUILD_TYPE=RelWithDebInfo

# Use the CMake build system for dh
%:
  dh $@ --buildsystem=cmake

# Advanced configuration with CMake
override_dh_auto_configure:
  dh_auto_configure -- $(CMAKE_FLAGS)

# Specify the installation path
override_dh_auto_install:
  dh_auto_install --destdir debian/tmp

# Bypass the dependency provenance check (if I had to manually
# compile a project for a certain architecture, dh will complain)
override_dh_shlibdeps:
  dh_shlibdeps --dpkg-shlibdeps-params=--ignore-missing-info
```

Per quanto riguarda invece il file `sshlirp.install`, la distinzione tra i due binari prodotti ha semplicemente permesso di selezionare, in fase di loro installazione nel pacchetto, il solo binario statico, come segue:

```
usr/bin/sshlirp-*
```

2.2 Costruzione dei pacchetti sshlirp

Il passo successivo allo sviluppo della directory `debian/` è stata l'esecuzione del processo di build vero e proprio tramite `gbp`.
Dopo un primo test di build in locale per l'architettura host (`amd64`), che ha permesso di verificare la correttezza dei file Debian e di debuggare eventuali errori, si è proceduto alla costruzione cross-compilata dei pacchetti per tutte le altre architetture target che, come anticipato, corrispondevano ai port Debian `arm64`, `armel`, `armhf`, `i386`, `mips64el`, `ppc64el`, `riscv64` e `s390x`.
Per l'esecuzione di questo processo "architetturalmente incrociato", è stato necessario impiegare strumenti di emulazione che, solo in un secondo momento, sono stati scelti con maggior consapevolezza.
Un primo sviluppo dei pacchetti di sshlirp si è infatti appoggiato sull'uso grezzo di *Qemu System Emulation*, che, sebbene molto potente e versatile, si è presto rivelato essere una soluzione "over-dimensionata" per lo scopo di una semplice cross-compilazione.

2.2.1 Approcci iniziali: Qemu System Emulation

Come anticipato nella sezione 1.1.1, Qemu System Emulation permette di emulare interi sistemi hardware, inclusi CPU, memoria, dispositivi di I/O e periferiche, consentendo l'esecuzione di sistemi operativi completi all'interno di macchine virtuali e fornendo un ambiente isolato per testare e sviluppare software su architetture diverse da quella host, grazie all'impiego diretto del suo TCG [4].

Inizialmente si è pensato che l'impiego di Qemu System Emulation potesse costituire una soluzione pratica e ripetibile al problema della cross-compilazione di sshlirp, in quanto avrebbe permesso non solo di eseguire la build incrociata ma anche di seguire le pratiche di virtualizzazione documentate da Virtualsquare, che prediligono odiernamente l'uso di Qemu come host virtuale [1].
Nonostante ciò, durante la costruzione del pacchetto di sshlirp per la particolare architettura `armel`, si è concluso che l'uso di Qemu System Emulation introduceva non solo un sostanziale overhead in termini di risorse di sistema e tempi di esecuzione, ma anche una complessità di gestione delle configurazioni e delle dipendenze che rendeva il processo di build meno agevole e ripetibile, specialmente per port meno documentati e/o obsoleti.

Questa prima strategia è stata quindi presto abbandonata dopo la sperimentazione di solo due delle architetture target: `arm64` e appunto `armel`. Infatti, sebbene l'emulazione e la cross-compilazione per la prima fossero andate a buon fine senza particolari difficoltà, l'architettura `armel` ha introdotto numerosi ostacoli, per quanto riguarda sia la configurazione e l'avvio dell'host Qemu che il processo di build di sshlirp, dovuti principalmente alla sua obsolescenza e alla conseguente scarsità di supporto, in particolare per lo scopo di una sua emulazione di sistema.
Nonostante ciò, l'esperienza acquisita durante l'impiego di Qemu System Emulation per queste due architetture ha permesso di affrontare la soluzione successiva con maggior consapevolezza, in particolare per quanto riguarda le difficoltà che si sarebbero comunque dovute affrontare per la risoluzione delle dipendenze di build statica.

Per comprendere al meglio i motivi che hanno portato al celere abbandono di questa prima strategia è bene illustrare:

1. il funzionamento generale dell'avvio di un'emulazione di sistema tramite Qemu, che ha fatto emergere i primi limiti sostanziali di questo approccio applicato a port obsoleti;
2. le differenze di emulazione che hanno introdotto al successo del primo tentativo di emulazione per `arm64` e al fallimento del secondo per `armel`;
3. le difficoltà di risoluzione delle dipendenze di build statica riscontrate per il port più datato, correlate con l'emulazione di sistema e determinanti nella scelta di passare a *Qemu User-Mode Emulation*.

Avvio di un host Qemu

L'avvio di un intero host virtuale Debian per un'architettura specifica attraverso Qemu, richiede la disponibilità e l'impiego di componenti fondamentali senza i quali l'hypervisor stesso non si può avviare [29]:

- **Un kernel** minimale di una versione Debian specifica, compilato per l'architettura target e necessario per l'avvio dell'installer. Questo kernel può essere fornito sia da un'immagine CD ISO di installazione che direttamente da un file binario `vmlinuz`;
- **Un initrd** (*initial ramdisk*), ossia un archivio `.gz` caricato in RAM al momento dell'avvio del kernel, che contenga gli strumenti e i driver necessari per l'installazione e il montaggio del filesystem root. Anche questo componente è derivabile da un'ISO o da un file dedicato;

- **Un metodo di avvio**, che può essere basato su BIOS, UEFI o *Direct Linux Boot*, a seconda dell'architettura target e delle sue specifiche di boot;
- **Un'immagine di disco virtuale**, che funga da storage primario per l'host Qemu e sulla quale l'installer scriverà il sistema operativo Debian, dopo le prime fasi di setup.

Dipendentemente dall'architettura target, la guida ufficiale di installazione Debian su Qemu [29] suggerisce specifiche per l'avvio profondamente diverse.
Per l'emulazione di architetture moderne e odiernamente supportate il metodo suggerito risiede in una combinazione di ISO `netinst` + UEFI (`EDK II`) + `qcow2`. L'uso di un'immagine ISO di tipo `netinst` (*Network Install*) permette infatti di avviare un'installazione minimale e personalizzabile del sistema operativo, che scarichi in fase di setup solo i pacchetti necessari, riducendo così i tempi di setup iniziali [30]. Inoltre l'adozione di un sistema di avvio tramite UEFI pittosto che BIOS, consente un boot più prestazionale. Per questo motivo, in ambito di emulazione di sistema Qemu, si sceglie solitamente di installare `edk2` di *TianoCore*, l'implementazione di riferimento dell'UEFI sviluppata da Intel®, e di utilizzarne i file `AAVMF_CODE.fd` e `AAVMF_VARS.fd` - rispettivamente il codice e le variabili del firmware - per l'avvio della macchina virtuale [31, 32]. Infine, l'uso di un'immagine di disco virtuale in formato `qcow2` offre funzionalità avanzate di allocazione di spazio sparsa, compressione e crittografia, ottimizzando così l'uso delle risorse di archiviazione [33].

Un approccio più tradizionale, suggerito invece per l'emulazione di architetture più datate o meno diffuse, prevede una somma di `netboot` + *Direct Linux Boot* + `qcow2`. In questo caso, sebbene l'impiego di un disco virtuale moderno sia comunque supportato, l'avvio dell'host virtuale Qemu avviene tramite boot diretto del kernel, ossia caricando direttamente in memoria sia il file `vmlinuz` che l'initrd [29, 34], seguendo quindi un approccio di tipo `netboot`.

Ostacoli preliminari per l'avvio dell'host `armel` e differenze con `arm64`

L'errata supposizione che un'emulazione di sistema fosse il giusto approccio per il problema della cross-compilazione di sshlirp è stata sfortunatamente validata dall'iniziale successo della sua applicazione per l'architettura `arm64`, derivato da un'esecuzione di un processo standardizzato, documentatamente corretto e funzionante per questo port moderno [29]:

1. **Download dell'immagine ISO**: seguendo le linee guida Debian, è stata scelta un'immagine di tipo `netinst`;
2. **Creazione dell'immagine di disco virtuale**: anche per questo punto è stato seguito il suggerimento di un'immagine `qcow2`;
3. **Installazione di un UEFI arm64 sull'host**: `qemu-efi-aarch64` di TianoCore EDK II;
4. **Installazione del sistema operativo Debian arm64**: attraverso il comando `qemu-system-aarch64` si è avviato un processo in cui sono stati selezionati gli attributi di partizionamento, configurata la rete e installato il software di base;
5. **Avvio della macchina virtuale, clone dei repository e build del pacchetto**: una volta completata l'installazione dell'OS sull'immagine `qcow2`, si è avviata la macchina virtuale. Al suo interno si è poi dovuto clonare il repository Git di `libslirp` per il soddisfacimento del requisito di staticità, come esposto nella sezione 2.1.2. Dopo la sua conseguente compilazione - con allegata risoluzione di dipendenze innestate - e installazione a livello di sistema dei suoi archivi statici, si è clonato il repository Debian Salsa GitLab di `sshlirp` - costruito come descritto nella sezione 2.1.2 - e si è infine eseguita la build del pacchetto `.deb`;
6. **Trasferimento del pacchetto tramite port forwarding**: una volta generato il pacchetto `.deb` di sshlirp, si è proceduto al suo trasferimento sull'host reale tramite l'uso di `scp`, abilitato da un port forwarding della porta SSH della macchina virtuale Qemu verso l'host reale.

L'immediato buon esito di questo piano, ha lasciato spazio al tentativo disinformato di applicarlo anche all'architettura `armel`, per cui però è fallito al punto 4. Tale insuccesso è stato quindi seguito da una successiva analisi che ha portato alla luce quanto detto nella sezione precedente.
Infatti, sebbene per tale port datato fossero forniti da Debian sia un'immagine ISO `netinst` (per la release *bookworm* 12.12.0 [35]) che un UEFI compatibile (`qemu-efi-arm` [32]), è noto che i kernel Debian per armel non includono il supporto EFI stub. Un bug Debian nota osserva infatti: "Per avviare un kernel in modalità UEFI, deve essere compilato con `CONFIG_EFI`. Ma nessun kernel Debian armel (es. linux-image-marvell, linux-image-rpi) è costruito con `CONFIG_EFI`", rendendo così il GRUB UEFI per armel praticamente inutilizzabile [36].
Questa conclusione ha logicamente reso necessaria la sostituzione del metodo di avvio

ISO `netinst` + UEFI `EDK II` con la combinazione `netboot` + *Direct Linux Boot*, che però ha obbligato al dowonload manuale di un kernel `vmlinuz` e di un initrd più datati. Ciò a sua volta ha introdotto difficoltà nella risoluzione delle dipendenze di build di `libslirp`.

Build statica su armel e interruzione d'uso di Qemu System Emulation

Come osservato nella sezione 2.1.2, le difficoltà relative all'adempienza del requisito di staticità si sono concentrate sulla compilazione manuale di `libslirp`. A sua volta, una grossa fetta dello svolgimento di questo compito appartiene al caso particolare del tentativo di compilazione di tale libreria per l'architettura `armel` emulata con Qemu System Emulation.
Infatti, l'impiego di un kernel e un initrd datati ha portato con sé il limite di un mirror Debian obsoleto. Ciò ha significato il sorgere di innumerevoli conflitti di versioning tra le dipendenze di `libslirp` stesso.
Quando è stato evidente che la catena di compilazione manuale delle dipendenze, necessaria per risolvere tali problemi di versioning, avrebbe assunto dimensioni ingestibili per lo scopo di una pacchettizzazione da ripetere per tutti i port Debian, si è deciso di abbandonare l'approccio basato su Qemu System Emulation in favore di una strategia più leggera e specifica per la cross-compilazione: *Qemu User-mode Emulation*.

2.2.2 Impostazione dell'approccio definitivo: uso di Qemu User-mode Emulation combinato a debootstrap

La scoperta di *Qemu User-Mode Emulation* ha permesso di superare tutti i limiti legati alla riproduzione dell'hardware di sistema, concentrandosi invece sull'emulazione di una compilazione per architetture diverse da quella host, aderendo perfettamente all'ambizione di distribuzione globale di pacchetti statici di sshlirp.
In modalità *User-Mode Emulation* Qemu infatti non fa altro che tradurre codice user-space del programma guest in codice equivalente compatibile con il sistema host, non intervenendo quindi con l'emulazione di sistemi operativi né tantomeno di hardware [37].

Intuizione teorica sull'impiego di Qemu User-Mode Emulation

Sostanzialmente, Qemu User-Mode Emulation è stato impiegato per intercettare, grazie al suo JIT TCG, le chiamate di sistema del programma guest, pensabile per semplicità come il compilatore binario per sshlirp compilato per un'architettura diversa da quella host, mapparle in systemcalls equivalenti sul sistema host (adeguando endian e formati a 32/64 bit) ed eseguirle direttamente su sshlirp.
Il risultato è stato un binario di sshlirp compilato da un compilatore che traduce codice sorgente in istruzioni macchina dello stesso tipo delle istruzioni macchina che lo compongono, ossia un binario di sshlirp cross-compilato per l'architettura dello stesso compilatore. E ciò che ha consentito l'esecuzione di tale compilatore su un sistema host di architettura differente è stata proprio l'emulazione a run-tim svolta da Qemu.

Quindi, assumendo che:

- la notazione $\mathrm{C}^{L0}_{L1,L0}$ rappresenti un compilatore scritto in un linguaggio $L0$ che traduce codice $L1$ in codice $L0$;
- la notazione I^{L0}_{L1} rappresenti un interprete scritto in un linguaggio $L0$ che esegue codice $L1$;
- la notazione P^{L0} rappresenti un programma scritto in un linguaggio $L0$;

Possiamo ridurre:

- il programma di pacchettizzazione di sshlirp per un'architettura target (*e.g.* `gbp` per `armel`) a un compilatore $\mathrm{C}^{t}_{s,t}$ scritto in linguaggio macchina *target* che traduce codice *sorgente* in codice *target*;
- il TCG di Qemu a un interprete I^{h}_{t}, scritto in linguaggio macchina *host* che esegue codice *target*;
- il codice di sshlirp a un programma P^{s} scritto in linguaggio *sorgente*;

E infine concludere che il processo di cross-compilazione, esposto poc'anzi, è rappresentabile attraverso la seguente espressione formale:

$$\mathrm{I}^{h}_{t}\left(\mathrm{C}^{t}_{s,t}(\mathrm{P}^{s})\right) = \mathrm{P}^{t}$$

La validità di questa similitudine si appoggia però su un tassello fondamentale non ancora introdotto: un ambiente isolato in cui il compilatore $\mathrm{C}^{t}_{s,t}$ possa essere eseguito dall'interprete I^{h}_{t} senza conflitti di dipendenze; ossia, parafrasando, un filesystem minimale Debian in cui il processo di pacchettizzazione incrociata, svolto da `gbp`, possa essere mappato dal TCG di Qemu e eseguito correttamente.

L'impiego di debootstrap

In precedenza, l'ottenimento di un ambiente Debian isolato e di architettura straniera per la pacchettizzazione incrociata era già incorporato nella soluzione di Qemu System Emulation.
Con Qemu User-Mode Emulation invece, la creazione di tale ambiente è stata resa possibile dall'uso di `debootstrap`.
Questo strumento - che necessita nativamente di privilegi root - permette infatti di installare un filesystem Debian minimale in una directory specificata, scaricando e configurando i pacchetti di base necessari per il funzionamento di un ipotetico sistema operativo per un'architettura target [38].
Tale processo, nel contesto di "cross-debootstrapping" [38], viene svolto in due fasi principali:

1. **Download ed estrazione dei pacchetti di base con** `debootstrap --foreign`: `debootstrap` scarica i pacchetti essenziali dall'archivio ufficiale specificato, estraendoli nella directory di destinazione data dall'utente, e installa nel sistema base una copia del suo stesso programma che sarà utilizzata nella seconda fase; questo primo processo viene eseguito "esternamente" al filesystem in costruzione ed è avviabile attraverso il comando [39]:

   ```
   debootstrap --arch=<architettura_target> --foreign
   <release> <destinazione> <mirror>
   ```

2. **Configurazione dei pacchetti e del filesystem con** `--second-stage`: una volta estratti i pacchetti, `debootstrap` configura il filesystem minimale, impostando le directory di sistema, i file di configurazione e le dipendenze necessarie per l'esecuzione di base. Questo passo, a differenza del primo, viene eseguito "internamente" al filesystem in costruzione in quanto richiede l'esecuzione di alcuni dei binari estratti per il completamento dell'installazione.
   Per questo motivo è necessario copiare preventivamente l'eseguibile di Qemu User-Mode Emulation, corrispondente all'architettura target, all'interno del filesystem minimale o installare `binfmt-support` sul sistema host. Questo semplice tool permette infatti di registrare i binari di Qemu come interpreti per i file eseguibili di architettura straniera, consentendo così l'esecuzione trasparente di tali binari all'interno del filesystem debootstrap-ato [40].

Una volta predisposto l'ambiente si può quindi chroot-are in esso per eseguire infine il comando [39]:

```
/debootstrap/debootstrap --second-stage
```

Il risultato di questo iter è un *rootfs* Debian chroot-able per l'architettura target, che ha concesso al processo di pacchettizzazione incrociata di sshlirp un ambiente dove la cross-compilazione potesse eseguire correttamente e più agevolmente rispetto a quanto accadeva nei sistemi emulati integralmente con Qemu System Emulation.

Risoluzione delle dipendenze su port datati

Nonostante la compilazione manuale di `libslirp` e molte delle sue dipendenze sia rimasta un compito necessario per il soddisfacimento del requisito di staticità, l'uso di Qemu User-Mode Emulation combinato a `debootstrap` ha ridotto notevolmente la "lunghezza" e la "larghezza" della catena di archivi statici `.a` dipendenti tra loro, su port datati come `armel`, `armhf` e `i386`.
Infatti la facilità di setup e utilizzo di questi strumenti, che si è palesata ad esempio nella possibilità di scegliere intuitivamente l'architettura target, la release e il mirror oltre che nelle prestazioni di emulazione ottimizzate, ha rivelato anche un alto grado di supporto e aggiornamento.

2.2.3 Vantaggi rispetto a pbuilder

Un'altra soluzione al problema della cross-compilazione altamente documentata è rappresentata dal wrapper di `pbuilder`, `qemubuilder` [41, 42].

`pbuilder` è lo strumento alla base dello stack di build Debian e permette, appoggiandosi anch'esso a `debootstrap`, un processo di pacchettizzazione "pulito", altamente automatizzato e ripetibile con semplici comandi di `create`, `build` e `update` [41].
Sebbene le avanzate funzionalità built-in di risoluzione delle dipendenze rendano questo strumento altamente affidabile, il suo wrapper `qemubuilder` introduce complessità aggiuntive - quali creazione di un'intera immagine qemu e avvio di un host virtuale con tanto di kernel e initrd [42] - che lo etichettano come "ridondante", anche nella stessa wiki Debian [41], rispetto alla più moderna soluzione basata su Qemu User-Mode Emulation e `debootstrap`.

2.2.4 Considerazioni sull'approccio finale

L'adozione di Qemu User-Mode Emulation combinato a `debootstrap` e `chroot` ha sancito il confine tra la fase di sviluppo e testing "manuale" dei pacchetti di sshlirp e quella di automazione del processo di build incrociata dei binari.
Infatti questo approccio finale estremamente leggero, prestazionale e ripetibile ha permesso di intravedere la possibilità di automatizzare l'intero processo di cross-compilazione e delivery dei binari di sshlirp.
La prima soluzione che implementerà quest'idea, documentata nel capitolo 3, sarà `sshlirpCI`. Questa sarà poi rimpiazzata dalle successive evoluzioni a causa del suo limite principale, nato proprio dall'adozione di questo approccio combinato Qemu-debootstrap-chroot: la necessità di privilegi elevati per l'esecuzione di `debootstrap` e `chroot`.

L'unica alternativa nativamente "rootless" all'uso di questi strumenti sarebbe stato lo switch da uno sviluppo in locale a uno cloud-based, sfruttando le pipeline di Continuous Integration di Salsa CI.
Essendo però questo requisito di "rootlessness" sorto solo in un secondo momento ben successivo alla fase di pacchettizzazione manuale, lo si è escluso in principio.
Inoltre, anche dopo un confronto a posteriori, sarebbe stata confermata la scelta di un approccio locale e validata la supremazia - per il caso d'uso specifico - della sua versione finale Rootless V^2CI.

2.3 Confronto tra build in locale e uso delle pipeline di CI di GitLab

Come anticipato nella sezione 2.1, l'istanza GitLab di Debian, Salsa, fornisce un sistema di Continuous Integration (CI) che permette di automatizzare il processo di build, test e deployment dei pacchetti Debian [44].
Questa modernissima tecnologia quindi non mira solo alla costruzione di pacchetti `.deb` una tantum, ma consente anche di creare intere pipeline di build che, dalla fase di commit del codice sorgente fino al rilascio del pacchetto, eseguono in container Docker [43] in cui l'intero processo, comprensivo dell'installazione dei pacchetti necessari e della risoluzione delle dipendenze, viene svolto automaticamente sotto le direttive specificate in un file `.gitlab-ci.yml` [44].

Sebbene questo strumento di CI sia il più avanzato e largamente impiegato per la costruzione di pacchetti Debian, come sottolineato nella precedente sezione, si è concluso a posteriori che, per il requisito specifico di costruzione e delivery di pacchetti cross-buildati staticamente una tantum, un approccio locale rimanesse comunque preferibile.
Inoltre, le motivazioni che hanno portato a validare la soluzione locale per il soddisfacimento dei requisiti di pacchettizzazione, sono rimaste vive anche in presenza dei successivi requisiti di cross-compilazione statica automatizzata, premiando allo stesso modo la soluzione finale di Rootless V^2CI rispetto a un'ipotetica pipeline di CI in Salsa.
Tali motivazioni, definibili quindi "globali", sono riassunte di seguito.

2.3.1 Prestazioni

Dal momento che le pipeline di CI in Salsa eseguono il processo di build all'interno di container Docker ospitati su macchine virtuali condivise [43], le risorse di sistema assegnate a tali container sono limitate e soggette a variazioni in base al carico del server.
Inoltre, Salsa CI, per ogni fallimento di build, reimposta completamente l'ambiente di esecuzione del container, costringendo a ripetere l'installazione delle dipendenze e la configurazione dell'ambiente da zero.
Questo vincolo, in una prima fase di studio e testing del processo di pacchettizazzione, avrebbe rallentato notevolmente il ciclo di sviluppo. Per esempio, al verificarsi di un semplice fallimento di build causato da frequenti e probabili errori di dipendenza e versioning, mentre lo sviluppo locale sarebbe potuto proseguire con compilazioni iterative, con Salsa CI si sarebbero ottenuti innumerevoli jobs falliti e ampi tempi di attesa dovuti alla ricostruzione degli ambienti containerizzati.

2.3.2 Monitorabilità e debugging

La necessità di testare e correggere costantemente il processo di pacchettizzazione incrociata, al fine di studiarne le pediculiarità e i requisiti specifici per ogni architettura in questa prima fase di build manuale, si traduceva nell'esigenza di reperibilità e completezza dei log di sistema.
Sebbene Salsa CI fornisca un sistema di logging integrato che consente di visualizzare i log di build direttamente nell'interfaccia web [44], l'analisi e il debugging di errori complessi o ricorrenti possono risultare difficili a causa della natura temporanea dei

container Docker e della limitata esposizione dei log di sistema degli stessi.
La build in locale invece ha portato massimo grado di controllo e monitorabilità, accellerando e permettendo di affinare il processo di sviluppo.

2.3.3 Re-use delle risorse di build

Come accennato nella sottosezione 2.3.1, la natura effimera dei container Docker in Salsa CI implica che ogni build inizi con un ambiente "pulito", privo di qualsiasi pacchetto o configurazione precedentemente installata [44].
Questo non solo rallenta il processo di build, come già scritto, ma impedisce anche il riutilizzo di risorse precedentemente configurate, come cache di pacchetti o librerie compilate.
Nel contesto di sviluppo locale invece, la persistenza dei rootfs debootstrap-ati ha permesso di ottimizzare l'impiego delle risorse di sistema e di mantenere uno stato di build coerente tra le diverse iterazioni, senza doverlo rigenerare per ogni insuccesso.

Capitolo 3

sshlirpCI e Rootless sshlirpCI

La seconda fase di sviluppo di Rootless V^2CI è sostanzialmente figlia di un cambio dei requisiti e delle necessità che avevano avviato la prima di pacchettizzazione e che sono stati soddisfatti dalla corrispondente soluzione finale che ha previsto, come già ampiamente discusso, la combinazione tra Qemu User-mode Emulation, debootstrap e chroot.
Questa espansione di requisiti ha visto quindi una proporzionata e simmetrica espansione della soluzione, la cui forma finale, che ha preso il nome di *Rootless sshlirpCI*, ha poi permesso di intravedere la possibilità di un'ulteriore generalizzazione.
L'essenzialità di questa fase intermedia risiede proprio in questo: con *sshlirpCI* e successivamente *Rootless sshlirpCI*, non solo si sono costruite le principali fondamenta del risultato finale introducendo automazione e "rootlessness", ma si sono anche modellati i mattoni che compongono l'architettura di Rootless V^2CI. Quest'ultimo infatti si baserà sul concetto di esecuzione sincrona di molteplici processi "Rootless sshlirpCI - like".

Dal momento che questi due prototipi intermedi sono stati comunque generati da un lungo processo di progettazione e sviluppo, risulta necessario dare una visione dall'alto introduttiva del percorso seguito, attraverso il seguente schema. sshlirpCI-RootlessSshlirpCIRoadMap.drawio.pdf

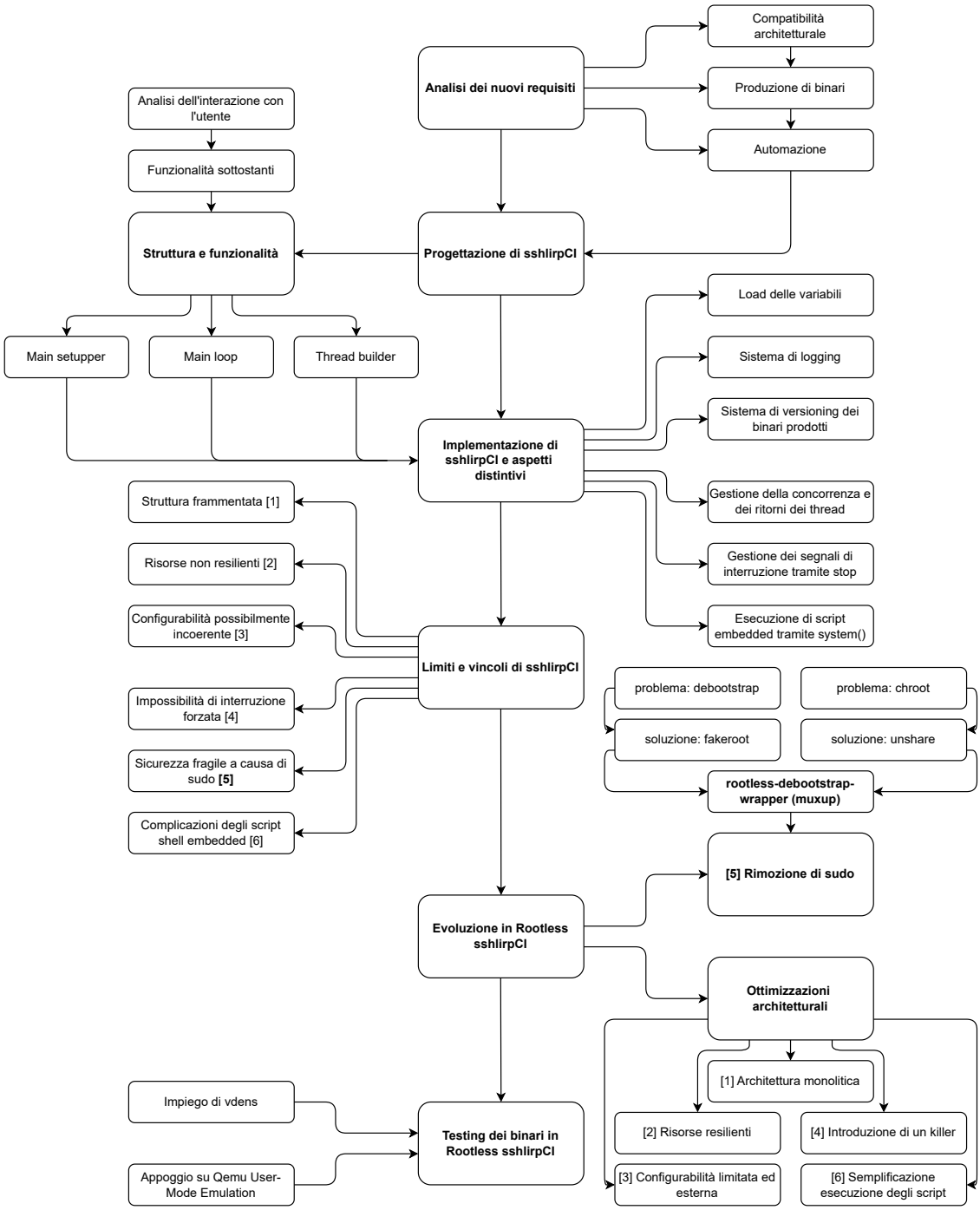

Figura 3.1: Roadmap di sviluppo di sshlirpCI e Rootless sshlirpCI

Per fornire una descrizione completa del processo che ha portato allo sviluppo di questi due motori per la cross-compilazione di sshlirp, è innanzitutto necessario esporre quali requisiti sono cambiati e come e quali si sono aggiunti rispetto alla fase precedente.

3.1 I nuovi requisiti e l'evoluzione del contesto

Partendo da sshlirpCI, il contesto non è più stato focalizzato su uno sviluppo manuale e su operazioni di testing, debugging e risoluzione delle dipendenze con lunghe catene di compilazioni in locale; bensì si è evoluto prima nella ricerca di una soluzione concettualmente aderente ai nuovi requisiti, e dopo nella sua implementazione.
Partendo quindi dai requisiti sorgente, è possibile descrivere come tale contesto è passato dal compito di pacchettizzazione di sshlirp alla modellazione di una prima soluzione, elencando di questi le tre principali differenze rispetto alle necessità precedenti:

- **Compatibilità**: dopo un'analisi più precisa del problema legato alla distribuzione di sshlirp su host remoti, si è concluso che la produzione di pacchetti statici anche per architetture obsolete e meno diffuse nel contesto Virtualsquare fosse ridondante e che quindi si rendesse necessaria una riduzione del numero di port per cui costruire i binari. Per questo motivo si è deciso di limitare la produzione alle architetture più supportate di `amd64`, `arm64`, `armhf` e `riscv64`. La scelta di escludere in particolare `armel` ha sollevato il contesto di sviluppo da molte delle difficoltà legate alla risoluzione delle dipendenze.

- **Produzione di binari**: sempre grazie a un orientamento più mirato alla risoluzione del problema specifico, si è concluso che la costruzione di interi pacchetti `.deb` fosse superflua e troppo problematica rispetto a un semplice processo di cross-compilazione e distribuzione di binari statici.
  Questa considerazione è anche figlia del limite imposto da `libslirp`, di cui si è discusso nella sezione 2.1.2: essendo il suo archivio statico `libslirp.a` assente dal pacchetto Debian ufficiale `libslirp-dev`, la compilazione "manuale" di tale libreria e il suo linking statico con sshlirp avrebbero firmato sshlirp stesso come un pacchetto non affidabile e quindi non distribuibile tramite repository ufficiali Debian, vanificando così lo scopo di una pacchettizzazione standard.
  Tale problema invece non sarebbe sussistito per la semplice costruzione e diffusione di binari statici su archivi esterni.

- **Automazione**: combinando la semplicità e ripetibilità della soluzione basata su Qemu User-Mode Emulation, debootstrap e chroot con la riduzione del numero di port e la semplificazione del processo di build, si è potuto intravedere la possibilità di automatizzare l'intero processo di cross-compilazione e deliv-

ery dei binari statici di sshlirp.
Quest'idea, di cui la prima finalizzazione sshlirpCI si sarebbe basata principalmente sulla semplice trascrizione in script Bash e file `C` dei comandi eseguiti per portare a termine il processo di build descritto nella sezione 2.2.2 del capitolo precedente, ha dato avvio a una nuova fase di sviluppo focalizzata sull'implementazione di un demone che eseguisse in sequenza e in modo autonomo le operazioni di cross-compilazione per i port target, operando in una directory di build, trasferendo i binari finali in una destinazione specifica e verificando la presenza di eventuali aggiornamenti al codice sorgente di sshlirp ogni intevallo di durata fornita dall'utente.

È importante sottolineare che, sebbene l'introduzione di questi nuovi requisiti abbia portato a un'evoluzione sensibile del progetto, la prima risposta a tali necessità non era stata ancora pensata per un'esecuzione "rootless".

3.2 sshlirpCI: prima automazione root-required

Creazione root-required dei rootfs per port target, chroot privilegiato in essi, clone dei repository necessari, risoluzione delle dipendenze e compilazione di sshlirp sono le operazioni presenti nell'insieme intersezione tra lo sviluppo manuale dei pacchetti svolto in precedenza e l'esecuzione del sistema di continuous integration implementato da sshlirpCI.
L'idea che risiede alla base di questo motore per la cross-compilazione consiste infatti nell'ereditare le soluzioni adottate in precedenza, riadattarle in script e programmi `C` per soddisfare i requisiti di compatibilità e produzione di binari, e infine orchestrare l'esecuzione di tali componenti in un demone che operi in un contesto configurabile dall'utente.

3.2.1 Struttura, funzionalità e gestione delle risorse

Come accennato in precedenza, sshlirpCI è stato progettato per eseguire le operazioni di cross-compilazione e delivery in background, fornendo diverse funzionalità, in modo che un utente privilegiato potesse:

- configurare preliminarmente l'esecuzione del demone stesso, specificando diversi parametri quali:
  - gli URL dei repository Git da cui effettuare il clone iniziale e i pull di aggiornamento per la produzione dei binari statici di sshlirp;

  - la directory di build in cui sshlirpCI avrebbe debootstrap-ato i rootfs per ogni architettura richiesta, posizionato i sorgenti e generato i file di compilazione;
  - la directory di destinazione dei binari finali;

- avviare sshlirpCI e delegargli la produzione e l'aggiornamento dei binari statici di sshlirp;

- interrompere l'esecuzione del demone in modo sicuro e coerente;

- riavviare sshlirpCI con garanzia di idempotenza, ossia con la certezza che sia in presenza che in assenza di un ambiente già precedentemente setup-ato, le funzionalità offerte dal motore di cross-compilazione potessero riprendere correttamente e restituire i risultati desiderati.

Sebbene gli unici punti di contatto tra l'utente finale e sshlirpCI siano rappresentati da quanto descritto poc'anzi, le funzionalità che permettono la sussistenza e la garanzia di tali punti sono molteplici e articolate.
Per fornire una visione d'insieme di queste, prima di scendere in una loro descrizione più dettagliata e focalizzata sull'implementazione delle stesse, è possibile affermare che l'architettura di sshlirpCI è stata costruita principalmente sui seguenti componenti e sotto-componenti:

- **Il loader della configurazione e il setupper**: questo elemento del processo principale - ossia il processo che esegue per primo dopo l'avvio di sshlirpCI da parte dell'utente - si occupa di caricare le impostazioni - di cui si è accennato nel precedente elenco puntato - fornite in un file di configurazione `ci.conf` ed utilizzarle, dopo essersi demonizzato, per settare una tantum il "guscio" dell'ambiente di build che, in sshlirpCI, si assume rimanga persistente ad ogni ciclo di esecuzione.

- **Il loop principale**: il cuore di sshlirpCI risiede in questo ciclo infinito che si occupa periodicamente di svolgere le seguenti operazioni:

  - **clone o pull dei sorgenti sull'host**: in caso di primo avvio di sshlirpCI, il demone clona i repository specificati nella configurazione all'interno della directory di build. In caso invece di esecuzione avviata o riavviata con un ambiente di build già esistente, sshlirpCI effettua un pull dei sorgenti per verificare la presenza di nuovi commit al solo codice sorgente

di sshlirp, trascurando la verifica dello stato di aggiornamento della sua dipendenza `libslirp`;

- **setupper dei thread builder per ogni port target**: internamente al main loop, se dal componente precedente risulta che siano stati effettuati dei cambiamenti al codice sorgente di sshlirp, sshlirpCI procede ad avviare per ogni architettura target un thread builder che si occuperà di svolgere tutte le operazioni cardine necessarie alla cross-compilazione di sshlirp dentro ambienti chroot. Affinché tale processo sincrono avvenga correttamente per ogni thread, il main dovrà costruire preliminarmente anche una struttura dati da passare ai thread builder, contenente le informazioni necessarie per la loro corretta esecuzione nei rootfs corrispondenti;
- **join dei thread builder e merge concatenato dei log prodotti**: una volta avviati tutti i thread builder, il main attende la loro terminazione in seguito alla quale procede a raccogliere i log di build prodotti da ciascuno di essi, concatenandoli in un unico file di log principale salvato nella directory di build e contenente i log precedentemente prodotti dal resto dei componenti, in modo da fornire una visione d'insieme completa di tutte le operazioni svolte in ogni ciclo di build;
- **move dei binari finali**: infine, una volta completate le operazioni di build per ogni port target, sshlirpCI si occupa di trasferire i binari statici prodotti dai thread builder in una sotto-directory "versionata" della directory di destinazione specificata nella configurazione, sovrascrivendo i file esistenti e garantendo così che per ogni tag del codice sorgente esista un path che contenga sempre le ultime versioni dei binari;
- **sleep e amministrazione dei segnali di interruzione**: essendo, come anticipato, prerequisito essenziale per sshlirpCI la possibilità di essere interrotto in modo sicuro e coerente, durante il conclusivo ciclo di sleep tra un'iterazione e l'altra del main loop, il demone si predispone a ricevere segnali di interruzione da parte dell'utente.

- **I thread builder**: questi componenti innestati nel cuore del flusso di esecuzione del main loop, si occupano di svolgere idempotentemente per ogni port target le operazioni ordinate di:

  1. creazione del rootfs debootstrap-ato per l'architettura specifica, qualora non fosse già presente nella directory di build, in caso di primo avvio;

2. creazione, se non già esistenti, delle directory e dei file di build all'interno del rootfs corrispondente, contestualmente alla prima iterazione del main loop;
3. copia dei sorgenti di sshlirp e libslirp all'interno dell'ambiente chroot;
4. installazione delle dipendenze necessarie e cross-compilazione di libslirp e sshlirp internamente al rootfs chroot-ato;
5. rimozione dei sorgenti di copia dal rootfs una volta completata la build.

Al fine di riassumere ed esporre graficamente l'architettura di sshlirpCI, è stato realizzato il seguente diagramma rappresentante le componenti principali e le loro interazioni:

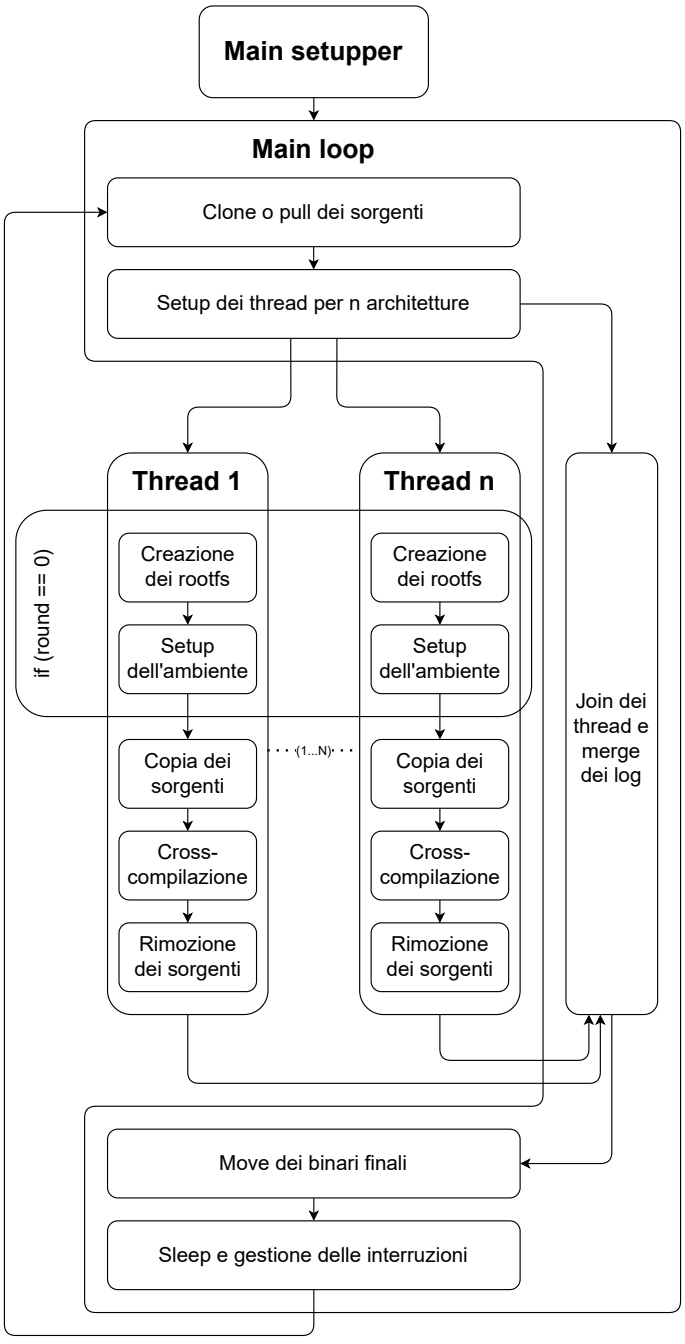

Figura 3.2: Diagramma architetturale di sshlirpCI

A partire da questa descrizione generale è possibile scendere nella delineazione di alcuni aspetti implementativi fondamentali che permettono il corretto funzionamento dell'intero sistema e l'interazione tra i componenti rappresentati.
In particolare si analizzeranno:

- i meccanismi di loading delle variabili di configurazione;
- il funzionamento del sistema di logging che ha ispirato l'importante espansione lato monitoring in Rootless V^2CI;
- il sistema di versioning adottato per la gestione e il delivery dei binari finali;
- i meccanismi di comunicazione con cui i thread builder amministrano l'uso delle risorse di sistema e quello con cui interagiscono in fase di setup e ritorno con il main loop;
- il metodo di gestione dei segnali di interruzione per l'arresto sicuro e coerente del demone;
- l'approccio adottato per l'esecuzione di operazioni direttamente sul filesystem host e su quelli chroot-ati.

Loading delle variabili di configurazione e setup

SshlirpCI, come anticipato, basa la sua esecuzione su alcuni parametri di configurazione che ne modellano il comportamento e le risorse impiegate.
Infatti, per motivi di utilizzo delle risorse di sistema, monitorabilità, correttezza di funzionamento e grado di compabilità e aggiornamento, l'utente ha il compito preliminare - all'avvio del demone - di specificare nel file `ci.conf` i seguenti parametri:

```
# Gli URL dei repository Git da cui effettuare il clone
# iniziale e i pull di aggiornamento:
SSHLIRP_REPO_URL=https://github.com/virtualsquare/sshlirp.git
LIBSLIRP_REPO_URL=https://gitlab.freedesktop.org/slirp/libslirp.git

# La directory di build in cui si desidera che sshlirpCI crei
# i rootfs, posizioni i sorgenti e salvi i log:
MAIN_DIR=/home/sshlirpCI

# Il file di versioning che si desidera venga utilizzato per
# tenere traccia delle versioni dei binari prodotti:
VERSIONING_FILE=/home/sshlirpCI/versions.txt

# Rispettivamente, la directory target in cui i thread builder
# posizioneranno provvisoriamente i binari all'interno dei rootfs
# chroot-ati (indicata con path relativo al rootfs), e la directory
# dell'host in cui si desidera che sshlirpCI esponga i binari finali:
```

```
THREAD_CHROOT_TARGET_DIR=/home/sshlirpCI/thread_binaries
TARGET_DIR=/home/sshlirpCI/binaries

# Le directory dell'host in cui si voglia che sshlirpCI
# salvi i sorgenti:
SSHLIRP_SOURCE_DIR=/home/sshlirpCI/sshlirp
LIBSLIRP_SOURCE_DIR=/home/sshlirpCI/libslirp

# Il file di log principale, su cui sshlirpCI concatenera' i
# log di tutti i componenti:
LOG_FILE=/home/sshlirpCI/log/main_sshlirp.log

# A scopo di monitoring a run-time, rispettivamente, la
# directory in cui i thread builder salveranno i loro log di
# esecuzione sull'host e il file dei log prodotti da ciascun
# thread builder durante la loro esecuzione interna al rootfs
# chroot-ato (indicato con path relativo al rootfs):
THREAD_LOG_DIR=/home/sshlirpCI/log/threads
THREAD_CHROOT_LOG_FILE=/home/sshlirpCI/log/thread_sshlirp.log

# L'intervallo di polling per la verifica di aggiornamenti ai sorgenti
# e le architetture target per la build:
POLL_INTERVAL=3600 # secondi -> 1 ora
ARCHITECTURES=amd64,arm64,armhf,riscv64
```

Tali variabili verranno lette dal loader attraverso una funzione di parsing `conf_vars_loader()` che, presi in input puntatori a variabili vuote corrispondenti ai parametri di configurazione e allocate dal `main` con dimensioni prefissate globalmente, aprirà il file `ci.conf`, ne leggerà riga per riga il contenuto e assegnerà i valori letti alle variabili puntate.
Successivamente, il setupper controllerà se è attiva una sua altra istanza - verificando la presenza di un pid file atteso in `/tmp` - e in caso negativo procederà a demonizzarsi tracciando la sua esistenza con la creazione del pid file.
Infine, a partire dalle variabili di configurazione caricate, procederà a creare - se non già esistenti - la directory di build, il file di versioning, la directory di log sull'host e il log file principale all'interno di essa, predisponendo così l'ambiente per l'esecuzione del main loop.

Il sistema di logging

SshlirpCI è stato progettato per fornire sia una singola interfaccia di logging di riferimento per l'utente finale, utile alla visualizzazione completa di tutte le operazioni svolte dal demone e alla verifica, a termine di ogni iterazione, dell'esito della cross-compilazione, che processi di logging distribuiti per ogni thread builder, essenziali per scopo di debugging e monitorabilità a run-time.

In generale, sia i log centrali appartenenti al file definito dalla variabile di configurazione `LOG_FILE`, che quelli distribuiti sui file "personali" di ogni thread builder, sono generati attraverso il semplice meccanismo di apertura di un `FILE` stream per il path specificato, di stampa su di esso tramite `fprintf`, e di chiusura dello stesso al termine del processo che se ne serve:

```
FILE* log_fp = fopen(log_file, "a");
if (!log_fp) {
    return 1;
}
fprintf(log_fp, "Log message here\n");
...
fclose(log_fp);
```

Inoltre, l'obbiettivo di fornire un unico log file centrale all'utente che desiderasse verificare l'esito complessivo di ogni iterazione del main loop, è stato finalizzato attraverso l'uso di log "speciali" che permettessero di comprendere l'intero processo di esecuzione di sshlirpCI, tramite separazioni chiare tra le sezioni di log prodotte dai diversi componenti - concatenate al termine del join dei thread builder - e timestamps per ogni operazione fondamentale quale l'inizio di un'iterazione del main loop, il completamento del processo di build per ogni port target, la conclusione dell'iterazione e l'eventuale interruzione del ciclo di sleep finale.
È importante sottolineare che l'inserimento di timestamps formattati ha anticipato l'idea di un sistema di monitoring più avanzato, che sarebbe stato implementato in Rootless V^2CI, e che avrebbe permesso di trasmettere i log prodotti dal demone a un sistema esterno di analisi e visualizzazione.

Versioning e gestione dei binari finali

A scopo di organizzazione, tracciabilità e aggiornamento dei binari statici prodotti, sshlirpCI si occupa di mantenere una struttura di sotto-directory "versionate" all'interno della directory di destinazione specificata nella configurazione.
Per fare ciò, oltre a servirsi di un file di versioning su cui vengono progressivamente scritti i nuovi tag dei sorgenti di sshlirp a cui corrisponderanno i binari prodotti nella stessa iteazione di build, sshlirpCI istanzia, preliminarmente al main loop, due strutture dati `commit_status_t`:

```
typedef struct {
  int status;
  char *new_release;
} commit_status_t;
```

In quest'ultime verranno salvati l'esito e il tag dell'ultimo commit rilevato - `"unstable"` se assente - rispettivamente per l'operazione di `check_host_dirs()` iniziale - che si occupa per il primo avvio di sshlirpCI di clonare i sorgenti - e per ogni successivo pull effettuato nel main loop attraverso la funzione `check_new_commit()`.
In questo modo, al termine di ogni iterazione del main loop, sshlirpCI, indipendentemente dal contesto di primo avvio o di esecuzione continuativa, potrà creare una sotto-directory nella directory di destinazione con nome corrispondente al tag dell'ultimo commit rilevato, e spostare in essa i binari prodotti dai thread builder, sovrascrivendo quelli eventualmente già presenti.

```
// main loop
while (1) {
  ...
  // 6. Controllo se ci sono le directory dell'host e i repository git
  if (round == 0) {
      log_time(log_fp);
      fprintf(log_fp, "Avvio del demone per la prima volta...\n");

      // return.status:
      // 0 -> tutto ok, le repo esistono gia'
      // 1 -> errore
      // 2 -> le repo sono state clonate ora
      initial_check = check_host_dirs(target_dir, sshlirp_source_dir,
```

```
        libslirp_source_dir, log_file, sshlirp_repo_url,
         libslirp_repo_url, thread_log_dir, log_fp, versioning_file
    );
    // Nota: questa funzione non fa nulla se
    // le dirs sono gia' esistenti e se esiste
    // gia' la repo git (possibile causa crash o interruzione)
    ...
}

// 6.1. Se non e' il primo avvio (e quindi avevo gia' clonato e
// ho atteso poll_interval secondi) o se la repo era gia' clonata,
// tento di pullare eventuali nuovi commit
if (round > 0 || initial_check.status == 0) {
    new_commit = check_new_commit(sshlirp_source_dir,
      sshlirp_repo_url, libslirp_source_dir,
      libslirp_repo_url, log_file, log_fp, versioning_file
    );
}

// 7. Se e' il primo avvio e ho effettivamente clonato o se ho
// trovato nuovi commit, preparo i thread per la build
if ((round == 0 && initial_check.status == 2) ||
  new_commit.status == 2
) {
  ...
  // 7.5. Sposto i binari in target_dir/initial_check.new_release
  // (oppure in target_dir/new_commit.new_release)
  for (int i = 0; i < num_archs; i++) {
    ...
    if (new_commit.status == 2) {
      snprintf(final_target_dir, sizeof(final_target_dir), "%s/%s",
        target_dir, new_commit.new_release
      );
    } else {
      snprintf(final_target_dir, sizeof(final_target_dir), "%s/%s",
        target_dir, initial_check.new_release
      );
    }
```

```
      ...
    }
  }
  ...
}
```

Thread builder: sincronizzazione e gestione delle risorse

Nel caso in cui il demone di sshlirpCI constati la necessità di eseguire una nuova iterazione di build - a seguito del primo avvio o della rilevazione di nuovi commit - il main loop procederà ad avviare per ogni architettura target un thread builder a cui delegherà l'esecuzione dell'intero processo di cross-compilazione per il port corrispondente.
A tal fine, il main costruirà preliminarmente un numero di strutture dati `thread_args_t` pari al numero di architetture target, ognuna delle quali conterrà le informazioni necessarie per l'esecuzione del thread builder associato:

```
#define MAX_ARCHITECTURES 9
#define MAX_CONFIG_LINE_LEN 512
#define MAX_CONFIG_ATTR_LEN 256
#define MAX_COMMAND_LEN 2048
#define MAX_VERSIONING_LINE_LEN 128

typedef struct {
    int pull_round;
    char arch[16];
    char sshlirp_host_source_dir[MAX_CONFIG_ATTR_LEN];
    char libslirp_host_source_dir[MAX_CONFIG_ATTR_LEN];
    char chroot_path[MAX_CONFIG_ATTR_LEN];
    char thread_chroot_main_dir[MAX_CONFIG_ATTR_LEN];
    char thread_chroot_sshlirp_dir[MAX_CONFIG_ATTR_LEN];
    char thread_chroot_libslirp_dir[MAX_CONFIG_ATTR_LEN];
    char thread_chroot_target_dir[MAX_CONFIG_ATTR_LEN];
    char thread_chroot_log_file[MAX_CONFIG_ATTR_LEN];
    char thread_log_file[MAX_CONFIG_ATTR_LEN];
    pthread_mutex_t *chroot_setup_mutex;
} thread_args_t;
```

Come è possibile dedurre dalla struttura dati soprastante, l'unica operazione per cui è risultata necessaria una serializzazione tra i thread builder tramite mutex è stata la creazione dei rootfs tramite `debootstrap`.
Infatti, essendo questa operazione particolarmente dispendiosa in termini di risorse di sistema e tempo di esecuzione, si è deciso di permettere che solo un thread per volta potesse eseguirla, in modo da evitare un sovraccarico di I/O e CPU che avrebbe potuto compromettere la stabilità dell'intero sistema host.

```
void *build_worker(void *arg_ptr) {
  thread_args_t* args = (thread_args_t*)arg_ptr;
  ...
  fprintf(thread_log_fp, "Worker started for arch %s. Pull round: %d.\n",
    args->arch, args->pull_round
  );
  if (args->pull_round == 0) {
    fprintf(thread_log_fp, "First run (pull_round 0). Checking and
      eventually setting up chroot for %s.\n", args->arch
    );
    pthread_mutex_lock(args->chroot_setup_mutex);
    int setup_status = setup_chroot(args, thread_log_fp);
    pthread_mutex_unlock(args->chroot_setup_mutex);
    ...
  }
  ...
}
```

In assenza di quest'accortezza si sono infatti sperimentati numerosi fallimenti di avvio di build per l'ultimo thread lanciato, a causa di una saturazione delle risorse di sistema nel contesto di esecuzione su host Ubuntu 24.04.3 LTS con processore 11th Gen Intel® Core™ i5-1155G7 × 8 e 16 GB di RAM.

Infine, per quanto riguarda il ritorno dei thread builder al main loop, si è deciso di impiegare una semplice struttura dati che permettesse di trasmettere l'esito del thread stesso e un eventuale messaggio di errore, in modo che il main potesse loggare tali informazioni nel file di log centrale al termine del join di tutti i thread e fornire così un resoconto completo e granulare dell'esito di ogni iterazione di build.

```
typedef struct {
```

```
  int status;
  char *error_message;
} thread_result_t;
```

Interruzione sicura e coerente del demone

Attraverso la compilazione di sshlirpCI tramite `cmake`, l'utente ottiene due eseguibili distinti:

- `sshlirp_ci_start`, che si occupa di avviare il demone;
- `sshlirp_ci_stop`, che invia un segnale di interruzione al demone in esecuzione.

Per permettere a quest'ultimo di arrestare in modo sicuro e coerente l'esecuzione di sshlirpCI, si è deciso di predisporre il main loop alla ricezione del segnale `SIGTERM` solo durante la fase di sleep tra un'iterazione e l'altra.
Questa funzionalità è figlia di un'interazione particolare tra il processo di start e quello di stop che non si basa solo su operazioni di read e write sul pid file e sull'invio di segnali, ma che prevede anche il salvataggio dello stato di esecuzione del demone in un file `STATE_FILE`, al path `/tmp/sshlirp_ci.state`, in cui vengano registrati i valori `WORKING` o `SLEEPING`, a seconda che il demone stia eseguendo un'iterazione del main loop o stia attendendo il completamento del ciclo di sleep.
In questo modo, il processo di stop potrà verificare lo stato corrente del demone leggendo tale file e, nel caso in cui risulti `SLEEPING`, inviare il segnale di interruzione `SIGTERM`; in caso contrario, attenderà un intervallo di tempo prefissato e ripeterà la lettura del file di stato, per un numero di tentativi totali definito.

```
while (attempts < MAX_WAIT_SECONDS) {
  ...
  char current_state[50] = {0};
  state_file_ptr = fopen(STATE_FILE, "r");
  ...
  if (state_file_ptr) {
    if (fgets(current_state, sizeof(current_state),
      state_file_ptr) == NULL
    ) {...}
    ...
  }
  if (strcmp(current_state, DAEMON_STATE_SLEEPING) == 0) {
```

```
    printf("Il demone e' in stato SLEEPING. Invio SIGTERM...\n");
    if (kill(daemon_pid, SIGTERM) == 0) {
      printf("Segnale SIGTERM inviato con successo.\n");
      // Attendo un po' che il demone termini e pulisca i suoi file
      sleep(3);
      ...
      printf("Demone sshlirp_ci terminato.\n");
      fclose(state_file_ptr);
      return 0;
    } else {
      ...
      fclose(state_file_ptr);
      return 1;
    }
  } else {
      if (attempts == 0) {
        printf("Il demone e' attualmente in stato '%s'. Quindi
          attendo...\n", strlen(current_state) > 0 ? current_state
          : "UNKNOWN"
        );
      }
      sleep(1);
      attempts++;
  }
  fclose(state_file_ptr);
}
```

Lato processo start invece, il main, durante l'esecuzione della funzione di demonizzazione, setta un handler per `SIGTERM` che imposta una variabile globale `terminate_daemon_flag` a `1` in caso di ricezione del segnale, in modo che il ciclo di sleep, se interrotto, possa verificare il valore di tale flag ed eventualmente terminare l'esecuzione di sshlirpCI, pulendo il pid file e il file di stato.

```
volatile sig_atomic_t terminate_daemon_flag = 0;
static void sigterm_handler(int signum) {
  if (signum == SIGTERM) {
    terminate_daemon_flag = 1;
  }
```

```
}
static void cleanup_daemon_files() {
  remove(PID_FILE);
  remove(STATE_FILE);
}
static void update_daemon_state(const char *state) {
  FILE *fp = fopen(STATE_FILE, "w");
  if (fp) {
    fprintf(fp, "%s", state);
    fclose(fp);
  } else {
    perror("Failed to update daemon state file");
  }
}
static void daemonize() {
  ...
  signal(SIGTERM, sigterm_handler);
  ...
}
int main() {
  ...
  daemonize();
  atexit(cleanup_daemon_files);
  ...
  while(1) {
    update_daemon_state(DAEMON_STATE_WORKING);
    ...
    update_daemon_state(DAEMON_STATE_SLEEPING);
    log_time(log_fp);
    fprintf(log_fp, "Demone in attesa per %d secondi...\n",
      poll_interval
    );
    // Ciclo di sonno e gestione dei segnali di interruzione
    unsigned int time_left = poll_interval;
    while(time_left > 0) {
      time_left = sleep(time_left);
      if (terminate_daemon_flag) {
        fprintf(log_fp, "Sleep interrotto da segnale di
```

```
            terminazione.\n"
          );
          break;
        }
        if (time_left > 0) {
          fprintf(log_fp, "Sleep interrotto, %u secondi rimanenti,
            continuo ad attendere...\n", time_left
          );
        }
      }
      round++;
    }
    ...
}
```

Operazioni su filesystem ed esecuzione degli script

Per garantire la corretta esecuzione delle operazioni di setup e build all'interno degli ambienti chroot-ati, sshlirpCI si affida all'esecuzione di comandi shell, invocati tramite la funzione `system()`.
In particolare, le operazioni di clone e pull dei sorgenti, di setup degli ambienti chroot da parte dei thread, di copia dei sorgenti dall'host ai rootfs, di cross-compilazione e di rimozione dei sorgenti dagli ambienti debootstrap-ati, si appoggiano tutte all'esecuzione di eseguibili `.sh`.
Quest'ultimi vengono generati a run-time, posizionati nella directory `/tmp` con un path univoco - settato tramite la funzione `mkstemp()` -, popolati attraverso la copia in essi di script embedded in stringhe - costanti e definite in componenti header dedicati - e invocati tramite `system()`.
Ad esempio, nel file sorgente di sshlirpCI `src/include/scripts/remove_source_copy_script.h`, è definito lo script di rimozione dei sorgenti di sshlirp e libslirp dai rootfs chroot-ati, sotto forma di stringa:

```
#ifndef REMOVE_SOURCE_COPY_SCRIPT_H
#define REMOVE_SOURCE_COPY_SCRIPT_H
static const char remove_source_copy_script_content[] =
  "#!/bin/bash\n"
  "\n"
  "chroot_path=$1\n"
```

```
    "chroot_sshlirp_dir=$2\n"
    "chroot_libslirp_dir=$3\n"
    "logfile=$4\n"
    "\n"
    "..."
    "exit 0\n";
#endif
```

Questo verrà poi invocato da ogni thread builder - con lo scopo di "pulire" il proprio rootfs al termine della build - attraverso la funzione `remove_sources_copy_from_chroot()`, la quale, a sua volta, farà riferimento a un metodo di utilità `execute_embedded_script_for_thread()`, che si occuperà appunto della generazione, della scrittura e dell'esecuzione dello script corrispondente.

Questo approccio, basato su script shell embedded, è stato inizialmente prediletto rispetto all'invocazione diretta di script esterni per motivi di portabilità e indipendenza dal filesystem. Infatti, l'incapsulamento degli stessi script all'interno del binario finale di sshlirpCI e la loro riproduzione su file `.sh` temporanei hanno sollevato lo sviluppo da problematiche legate sia alla distribuzione a compile-time di un'eventuale directory `script/` in una destinazione sicura, che alla gestione dei permessi di scrittura ed esecuzione degli script esterni, che, in caso di salvataggio permanente e invocazione diretta, sarebbero stati invece soggetti a modifiche accidentali o malevole da parte di utenti o processi non autorizzati.

Nonostante sshlirpCI si possa considerare un sistema di automazione di build incrociata per sshlirp abbastanza completo, diverse scelte implementative e architetturali hanno fatto emergere limiti e vincoli che hanno successivamente spinto allo sviluppo di una versione potenziata e allo stesso tempo più leggera e sicura: Rootless sshlirpCI.

3.2.2 Limitazioni, vincoli e privilegi

Le principali restrizioni che rendono sshlirpCI un'architettura valida ma non ottimale per lo scopo di cross-compilazione automatizzata di sshlirp si basano principalmente su sei fattori:

1. Composizione frammentata e over-ingegnerizzata;
2. Amministrazione non persistente delle risorse di build;

3. Fragilità e vulnerabilità legate a configurabilità eccessivamente granulare;
4. Impossibilità di interruzione forzata e pulita durante le operazioni di build;
5. Aspetti di sicurezza e limiti legati all'uso di `sudo`;
6. Complicazioni correlate all'impiego di script shell embedded e alla loro esecuzione tramite `system()`;

Frammentazione dell'architettura

Come è possibile osservare dal diagramma architetturale di sshlirpCI 3.2, la struttura di questo primo sistema di continuous integration per sshlirp presenta una composizione piuttosto frammentata.
In particolare la scomposizione dei task per ogni thread in diversi componenti - e conseguentemente in molteplici script shell - sebbene abbia permesso di isolare e incapsulare le singole operazioni di build, ha al contempo introdotto un eccessivo grado di complessità e over-ingegnerizzazione, che non solo ha reso difficile la manutenzione del codice e l'individuazione di bug, ma che ha anche avuto un impatto negativo sulla performance complessiva del sistema, a causa del numero elevato di operazioni di I/O e di creazione e distruzione di processi figlio per l'esecuzione degli script.
Dal momento che ogni thread builder lavora in modo indipendente dagli altri, un'architettura "monolitica" in cui ogni worker esegue un unico script di build che racchiuda tutte le operazioni necessarie - dalla creazione del rootfs allo spostamento dei binari finali - avrebbe permesso di semplificare notevolmente il flusso di esecuzione.

Persistenza e coerenza delle risorse

Un altro aspetto critico di sshlirpCI riguarda la gestione debole delle risorse di sistema durante le fasi di setup sia del main loop che dei thread builder.
Infatti, per quanto i metodi setupper `check_host_dirs()` - interno al main - e `setup_chroot()` e `check_worker_dirs()` - interni ai thread builder e addetti rispettivamente alla creazione dei rootfs e alla verifica delle directory di lavoro - permettano di evitare operazioni ridondanti in caso di esecuzione avviata, garantendo comunque idempotenza al riavvio, trascurano l'eventualità di una perdita o rimozione a run-time delle risorse di build.
Se ad esempio durante il primo ciclo di sleep del main loop venissero rimossi i rootfs, al termine di questo e all'avvio della seconda iterazione di build, i thread, ricevendo

un `thread_args_t* args` tale per cui `args->pull_round > 0`, darebbero per assunta l'esistenza dei rootfs e tenterebbero di copiare i sorgenti dall'host a directory non esistenti, causando un fallimento della build.

Allo stesso tempo anche il sistema di reperimento dei sorgenti basato sulla combinazione di clone iniziale sull'host, copia e rimozione post-build nei rootfs, sebbene permetta di minimizzare il carico di storage richiesto, il tempo di setup e il traffico di rete, introduce molteplici rischi di incoerenza. Per esempio, una modifica accidentale ai sorgenti sull'host durante l'esecuzione di una build potrebbe compromettere l'integrità del processo di cross-compilazione per tutti i threads. Ancora più critico sarebbe il caso in cui sshlirpCI venisse arrestato forzatamente con `sudo kill -9 <pid>` prima che i thread potessero rimuovere i sorgenti dai rispettivi rootfs. In questo scenario, al successivo avvio del demone, il main potrebbe pullare una nuova versione dei sorgenti, mentre i therad builder, attestando l'esistenza della directory `.git` dei sorgenti all'interno degli ambienti chroot-ati, non sovrascriverebbero i file presenti con quelli aggiornati, portando a un'incoerenza tra i sorgenti usati per la build e quelli presenti sull'host.

Configurabilità granulare e possibilmente incoerente

Durante la prima fase di progettazione di sshlirpCI, si è pensato che fornire all'utente la possibilità di configurare in modo granulare ogni aspetto dell'esecuzione del demone potesse essere un vantaggio in termini di flessibilità e adattabilità a diversi contesti di utilizzo.
Tuttavia, in seconda analisi, si è compreso che questa scelta avrebbe potuto portare a situazioni di incoerenza e vulnerabilità.
In particolare, la possibilità di settare path innestati arbitrariamente avrebbe potuto causare errori di esecuzione in caso di directory mancanti o non scrivibili.

Assenza di un killer

Un'altra limitazione di sshlirpCI riguarda l'impossibilità di interrompere forzatamente e comunque in modo pulito il demone durante le operazioni di build.
Infatti, come descritto nella sezione 3.2.1, dedicata all'interruzione sicura del demone, sshlirpCI può essere arrestato solo durante la fase di sleep tra un'iterazione e l'altra del main loop.
Ciò implica che l'utente, nel caso in cui desideri terminare l'esecuzione del demone

durante una fase di build, debba attendere il completamento di tutte le operazioni di cross-compilazione per ogni architettura target o, in alternativa, forzare l'interruzione del processo con `sudo kill -9 <pid>`, rischiando però di lasciare risorse di build in uno stato incoerente, come descritto nella sezione precedente.

Sicurezza e privilegi

L'impiego di `sudo` in sshlirpCI, necessario sia per la creazione dei rootfs tramite `debootstrap` che per l'esecuzione di `chroot`, espone il sistema host a potenziali rischi di escalation dei privilegi.
SshlirpCI è stato sviluppato nel corso di Giugno 2025, in un contesto in cui `sudo` non era ancora stato aggiornato alla versione `1.9.17p1` [45]. Prima di tale release, `sudo` - anche nella sua versione appena precedente `1.9.17` - era affetto da una vulnerabilità di sicurezza che riguardava proprio il suo uso combinato con `chroot` e le sue equivalenti flag `-R` e `--chroot` [46].
Come descritto dalla voce Common Vulnerabilities and Exposures CVE-2025-32463 [46] e come confermato dal relativo commit di patch a `sudo fdafc2c` [47], l'elevazione dei privilegi seguita da un'operazione di "spostamento radicale" della root directory del processo comportava che `sudo` eseguisse *prima* il cambio di root e *dopo* la risoluzione di utenti e gruppi servendosi del file `/etc/nsswitch.conf` interno al chroot, caricando così sull'host le librerie condivise specificate in esso, al momento dell'esecuzione di lookup NSS (*Name Service Switch*).
Questa falla permetteva a un utente o a un processo non privilegiato malevolo di definre un file `nsswitch.conf` costum all'interno dell'ambiente chroot - qualora quest'ultimo gli fosse accessibile - che facesse riferimento a moduli dinamici arbitrari - ad esempio una shared library che contenesse codice per lanciare una shell con UID 0 - ottenendo così, grazie all'uso di `sudo` combinato a `chroot` da parte di un programma innocuo, un'escalation di privilegi.
Sebbene questa vulnerabilità sia stata successivamente risolta con la release di `sudo 1.9.17p1`, l'uso di `sudo` in sshlirpCI, come illustrato dal precedente scenario, può comunque rappresentare un punto debole dell'architettura, oltre a imporre all'utente la necessità di disporre di privilegi root e a esporre il sistema host a rischi legati a errori di configurazione e bugs.

Complicazioni da script embedded e system()

Come anticipato nella sottosezione 3.2.1 dedicata all'illustrazione del meccanismo di esecuzione degli script shell embedded in sshlirpCI, l'uso di tale approccio ha sì privilegiato portabilità e indipendenza dal filesystem ma compromettendo sicurezza e prestazioni.

Infatti, sebbene l'invocazione di script temporanei avvenga direttamente dal binario `sshlirp_ci_start` con passaggio di parametri fissati, escludendo quindi in gran parte il rischio di *Command Shell injection*, il posizionamento di tali file `.sh` nella directory `/tmp` espone sshlirpCI a potenziali attacchi di tipo TOC/TOU (*Time-of-check/Time-of-use*): in un sistema multi-utente, un processo malevolo potrebbe intercettare la creazione di questi script temporanei e sostituirli con eseguibili o symlink arbitrari, portando, anche in questo caso, a un'escalation di privilegi o a danni al sistema host.
Ad aumentare la vulnerabilità di questo approccio vi è inoltre l'uso di `system()`, il quale lancia a run-time una shell che eredita tutte le variabili di ambiente dal processo chiamante. In un ambiente "sporco" o compromesso, questo potrebbe portare all'esecuzione di comandi il cui effetto differisce da quello atteso, causando malfunzionamenti o, ancora peggio, permettendo l'esecuzione di codice malevolo. Per questi motivi, l'impiego di `system()`, specialmente in programmi dotati di privilegi elevati, è generalmente sconsigliato in favore di chiamate di sistema più sicure e dirette, come `execve()` [48].
Infine, la generazione di script shell "usa e getta" in combinazione con la loro esecuzione tramite `system()` e la frammentazione architetturale di sshlirpCI, di cui si è parlato poc'anzi, introduce un overhead significativo in termini di performance, a causa del numero elevato di operazioni di I/O e di apertura e chiusura di processi figlio, che si ripercuote negativamente sul consumo di risorse di sistema e sul tempo totale di build.

Tutte queste vulnerabilità e limitazioni sono state sorpassate grazie a una re-ingegnerizzazione completa dell'architettura e a una considerevole semplificazione dell'implementazione di sshlirpCI, le quali, con anche una maggiore attenzione alle problematiche di sicurezza e coerenza, hanno portato allo sviluppo di Rootless sshlirpCI.

3.3 Evoluzione in Rootless sshlirpCI

Ciò che ha primariamente spinto sshlirpCI a evolversi in una soluzione più sicura, che si sarebbe poi dimostrata anche più leggera, scalabile e performante, è stata l'introduzione del nuovo requisito di "rootlessness", rispetto alle già soddisfatte prerogative di compatibilità, produzione di binari e automazione, discusse nella sezione 3.1.
A scopo di finalizzare tale ambizione, con lo sviluppo di Rootless sshlirpCI, si sono esplorate molteplici soluzioni escludenti l'uso di `debootstrap` e `chroot` tramite `sudo`, di cui quella adottata ha assunto una struttura più robusta, snella, sicura ed efficente, divenendo così la candidata perfetta per il ruolo di componente di base per la successiva e finale evoluzione in Rootless V^2CI.

3.3.1 Rimozione di sudo: impiego di fakeroot, proot e unshare

Come anticipato nelle precedenti sezioni, l'impiego sia di `debootstrap` che di `chroot` richiedeva, per l'utilizzo di sshlirpCI, il possesso di privilegi root sull'host.
Con lo scopo di eliminare questa dipendenza sono stati esplorati diversi tool alternativi, di cui quelli mantenuti nella soluzione finale sono stati `fakeroot` e `unshare`.

Uso di fakeroot per debootstrap

Per quanto riguarda la creazione di ambienti rootfs, l'uso di `sudo` risulta necessario dal momento che - per configurazione standard - `debootstrap` non crea solo directory e file necessari ad emulare un filesystem Debian minimale - scaricando pacchetti essenziali dal mirror specificato ed estraendoli nel base system - ma tenta anche di impostare permessi e proprietà degli stessi in modo coerente con quelli previsti da un'installazione Debian nativa, ossia con ownership impostata a `root:root`.
Inoltre `debootstrap`, usato singolarmente senza l'aggiunta di flag opzionali né in combinazione con altri tool di emulazione di privilegi, esegue anche operazioni di `mknod()` per device essenziali. La funzione di cui si serve per fare ciò è `setup_devices()` il cui contenuto prevede infatti di default la creazione di nodi device attraverso `setup_devices_simple()`:[49]

```
# create the static device nodes
setup_devices () {
```

```
  if doing_variant fakechroot; then
    setup_devices_fakechroot
    return 0
  fi
  case "$HOST_OS" in
      kfreebsd*)
    ;;
      freebsd)
    ;;
      hurd*)
    ;;
      *)
    setup_devices_simple
    ;;
  esac
}
...
setup_devices_simple () {
  # The list of devices that can be created in a container
  # comes from src/core/cgroup.c in the systemd source tree.
  mknod -m 666 $TARGET/dev/null c 1 3
  mknod -m 666 $TARGET/dev/zero c 1 5
  mknod -m 666 $TARGET/dev/full c 1 7
  mknod -m 666 $TARGET/dev/random c 1 8
  mknod -m 666 $TARGET/dev/urandom  c 1 9
  mknod -m 666 $TARGET/dev/tty  c 5 0
  mkdir $TARGET/dev/pts/ $TARGET/dev/shm/
  ln -s pts/ptmx $TARGET/dev/ptmx
  ln -s /proc/self/fd   $TARGET/dev/fd
  ln -s /proc/self/fd/0 $TARGET/dev/stdin
  ln -s /proc/self/fd/1 $TARGET/dev/stdout
  ln -s /proc/self/fd/2 $TARGET/dev/stderr
}
```

Debootstrap quindi non nasce con il solo scopo di generare in user-space un filesystem tree "fittizio" per l'esecuzione di processi isolati tramite l'impiego di risorse dell'host, ma è stato pensato per essere impiegato come strumento preliminare all'installazione e alla configurazione di sistemi Debian completi, comprensivi di device,

dischi partizionati e bootloader.
Operazioni di questo tipo richiedono necessariamente privilegi di amministrazione e operano su risorse del rootfs debootstrap-ato che, per coerenza e correttezza, devono quindi essere di proprietà dell'utente root.
Perciò, essendo `debootstrap` di per sé un tool che non svolge operazioni root-required - se non il setup di device minimali - ma che richiede tali privilegi principalmente per conformità, si è pensato di poter aggirare il vincolo sui permessi dei file creati impiegando `fakeroot` e trascurare completamente la creazione dei nodi device, i quali non sarebbero stati strettamente necessari per l'esecuzione di processi chroot-ati eseguibili in user-space, quali la cross-compilazione di sshlirp.

`fakeroot` infatti, attraverso una libreria condivisa `/usr/lib/*/libfakeroot-*.so`, caricata tramite il meccanismo di `LD_PRELOAD`, intercetta chiamate di sistema relative alla manipolazione di file, quali `getuid()`, `chown()` e `stat()`, e le sostituisce con implementazioni "simulate" che permettono a un processo di operare come se avesse privilegi di root, non modificandone l'UID né alterando i permessi dei file sul filesystem host, bensì mantenendo una tabella di mapping interna che associa file e directory a UID, GID e permessi "fittizi" [50].

L'astrazione che `fakeroot` permette di introdurre sulle operazioni svolte da `debootstrap` è inoltre nativamente supportata da quest'ultimo, il quale, in ambienti con "privilegi simulati", trascura automaticamente la creazione dei device con `mknod`, sostituendola con `setup_devices_fakechroot()` che si limita a creare dei link simbolici tra la directory `/dev` dell'host e quella del rootfs debootstrap-ato [49]:

```
setup_devices_fakechroot () {
  rm -rf "$TARGET/dev"
  ln -s /dev "$TARGET"
}
```

L'impiego di `fakeroot` ha quindi permesso di eseguire `debootstrap` senza privilegi di amministrazione, generando sì rootfs con file e directory di proprietà dell'utente corrente e privi di device "reali", ma superando comunque il primo ostacolo verso il raggiungimento di un modello completamente rootless per sshlirpCI.

Esplorazione di soluzioni rootless per chroot

L'altro grande ostacolo che impediva ancora a sshlirpCI di soddisfare il requisito di rootlessness riguardava la necessità di eseguire processi isolati all'interno dei rootfs debootstrap-ati.
Un primo tentativo disinformato di eseguire tale operazione senza privilegi di amministrazione - e quindi senza l'uso diretto di `chroot` - si è basato sulla sperimentazione di `fakeroot` anche per questa fase del processo di build.
Tuttavia, l'assunzione che tale tool potesse permettere di astrarre anche le chiamate di sistema relative al cambio di root directory si è presto scontrata con ciò che è anche ufficialmente documentato nel manuale dello stesso `fakeroot`. Questo strumento, come anticipato, non concede alcun tipo di capability aggiuntiva ai processi che esegue. Di conseguenza, un operazione di `chroot`, anche se wrappata da `fakeroot`, fallirebbe nel caso in cui l'utente non disponga della capability `CAP_SYS_CHROOT` [50, 51].

A seguito di questa constatazione, si è deciso di esplorare un'altra soluzione rootless per l'emulazione di `chroot`: `proot`.
Questo strumento permette di eseguire processi in ambienti isolati senza richiedere privilegi di amministrazione, sfruttando il meccanismo di *ptrace* per intercettare e manipolare le chiamate di sistema effettuate dal processo figlio.
Di fatto `proot` non esegue un vero e proprio cambio di root directory, ma piuttosto capta le chiamate di sistema che fanno riferimento a path assoluti e le riscrive in modo che puntino alla directory specificata come "nuova root" [52].
Questo tool in user-space però, oltre ad avere grosse limitazioni legate alla sua natura intrinseca di traduttore e manipolatore di systemcalls piuttosto che di meccanismo di isolamento nativo del kernel, ha mostrato, nel corso della sua sperimentazione alternativa a chroot, diversi problemi di compatibilità con Qemu User-Mode Emulation e rootfs di architettura guest differente da quella host.
Infatti `proot` non solo ignora la presenza di `qemu-<arch>-static` tra gli interpreti registrati tramite `binfmt_misc` - rendendo quindi necessaria la sua esplicita invocazione tramite flag `-q` - ma presenta anche problemi di risoluzione dei path, mount, e gestione delle variabili di ambiente e delle librerie condivise [53, 54], rendendo complicato, macchinoso, poco performante e per niente ripetibile il suo impiego in un contesto di continuous integration per la cross-compilazione di sshlirp.

Per questi motivi, dopo un ulteriore approfondimento delle alternative rootless a `chroot`, si è deciso di adottare un approccio differente.

Upgrade verso gli user namespaces e l'impiego di unshare

L'isolamento di UID, GID, capabilities e privilegi offerto dall'impiego degli user namespaces del kernel Linux [55] ha rappresentato la soluzione definitiva al nuovo requisito di rootlessness per sshlirpCI e funzionalmente alternativa all'impiego sia di `chroot` che di `proot`.
Questo potente meccanismo di sandbxing permette infatti ad utenti non privilegiati di guadagnare permessi di amministrazione all'interno di ambienti confinati, senza avere poteri aggiuntivi sul sistema host [55].
Sebbene questa funzionalità sia supportata nativamente dal kernel Linux per la maggior parte delle distribuzioni, alcune di queste - al fine di garantire una maggiore sicurezza - applicano delle restrizioni aggiuntive. Ad esempio, su host Ubuntu 23.10 o di versione >= 24.04 LTS, il kernel è compilato di default con l'opzione AppArmor `apparmor_restrict_unprivileged_userns` settata a `0`, impedendo così agli utenti non privilegiati di creare user namespaces [56].
Purtroppo in casi particolari come quest'ultimo, l'abilitazione degli user namespaces per utenti non privilegiati richiede necessariamente l'intervento dell'admin [56].

Il tool che incarna al meglio l'uso degli user namespaces per l'esecuzione di processi isolati, e che è stato impiegato nella soluzione finale di Rootless sshlirpCI, è `unshare`, il quale, con un ampia gamma di opzioni, permette di creare nuovi namespaces non privilegiati per differenti risorse di sistema [57].

Combinazione delle soluzioni adottate: rootless-debootstrap-wrapper

La fusione di `fakeroot` e `debootstrap` combinata alla sostituzione di `chroot` con `unshare` ha portato all'impiego di un nuovo flusso di operazioni per la creazione di ambienti rootfs isolati e non privilegiati, che ha costituito la base per lo sviluppo di Rootless sshlirpCI.
Grazie allo sviluppo di Alex Bradbury [58, 59] è stato infatti possibile riciclare, all'interno di Rootless sshlirpCI, una sua ambiziosa e brillante implementazione della soluzione, di cui si è poc'anzi discusso, "`fakeroot + debootstrap + unshare - chroot`", racchiusa in un wrapper script denominato
`rootless-debootstrap-wrapper` [59].

Il cuore di questo componente, integralmente riportato nei sorgenti di Rootless sshlirpCI, si occupa di [58, 59]:

1. Avviare la prima fase di debootstrap (`--foreign`) wrappato da `fakeroot -s` all'interno della directory e per l'architettura, la release e il mirror dati in input;
2. Estrarre le variabili di ambiente di `fakeroot`, salvate durante lo step precedente, all'interno della directory target;
3. Creare uno script `_enter` interno alla directory target, il quale permetta all'utente di accedere a un usernamespace radicato nella directory target;
4. Accedere al rootfs parziale e completarne la costruzione attraverso la seconda fase di debootstrap (`--second-stage`).

```
#!/bin/sh
# Copyright Muxup contributors.
# Distributed under the terms of the MIT-0 license,
# see LICENSE for details.
# SPDX-License-Identifier: MIT-0
TARGET_DIR=""
SUITE=""
MIRROR=""
ARGSTR=""
...
echo "@@@@@@@@@@ [1] Starting first stage debootstrap @@@@@@@@@@"
TMP_FAKEROOT_ENV=$(mktemp)
fakeroot -s "$TMP_FAKEROOT_ENV" debootstrap $ARGSTR || error "Stage 1
  debootstrap failed"
mv "$TMP_FAKEROOT_ENV" "$TARGET_DIR/.fakeroot.env"
...
echo "@@@@@@@@@@ [2] Extracting fakeroot for target @@@@@@@@@@"
cd "$TARGET_DIR" || error "cd failed"
fakeroot -i .fakeroot.env -s .fakeroot.env bash -e <<'EOF' ||
  error "Failed to extract fakeroot for target"
for deb in ./var/cache/apt/archives/{libfakeroot_,fakeroot_}*.deb; do
...
done
ln -s fakeroot-sysv ./usr/bin/fakeroot
EOF
```

```
cd "$OLDPWD" || error "cd failed"

echo "@@@@@@@@@@ [3] Creating _enter script @@@@@@@@@@"
cat <<'EOF' > "$TARGET_DIR/_enter"
#!/bin/sh
export PATH=/usr/sbin:$PATH
FAKEROOTDONTTRYCHOWN=1 unshare -fpr --mount-proc -R
  "$(dirname -- "$0")" \
  fakeroot -i .fakeroot.env -s .fakeroot.env "$@"
EOF
...
echo "@@@@@@@@@@ [4] Starting second stage debootstrap @@@@@@@@@@"
"$TARGET_DIR/_enter" debootstrap/debootstrap --second-stage
  --keep-debootstrap-dir || error "Stage 2 debootstrap failed"
...
```

L'adozione di questo wrapper ha permesso a Rootless sshlirpCI di soddisfare il requisito di rootlessness e, grazie alla sua forma pulita e robusta, è stato d'ispirazione per l'avvio di un processo di semplificazione e ristrutturazione dell'architettura.

3.3.2 Ottimizzazioni e architettura finale di Rootless sshlirpCI

Il superamento delle vulnerabilità e limitazioni di sshlirpCI legate all'uso di `sudo`, descritte nella sezione 3.2.2, ha acceso l'ambizione di risolvere anche gli altri vincoli architetturali e implementativi, portando a un considerevole miglioramento generale del motore di continuous integration per la cross-compilazione di sshlirp.
Infatti, sebbene la struttura triangolare portante *Loader* - *Main Loop* - *Thread Builder* alla base di sshlirpCI sia rimasta invariata anche in questa evoluzione, molti dettagli sono stati re-ingegnerizzati a favore di un disegno e una realizzazione più robusti, sicuri e performanti.
Oltre alla rootlessness nativa, le principali features di Rootless sshlirpCI, che sono state sviluppate in parallelo ai punti deboli di sshlirpCI, riguardano principalmente 4 aspetti:

1. Architettura monolitica e compatta;
2. Gestione sicura, coerente e idempotente delle risorse di build;

3. Configurabilità esterna del guscio di build e sviluppo interno delle strutture innestate;
4. Introduzione di un killer per l'interruzione sicura del demone in qualsiasi fase di esecuzione;
5. Semplificazione e sicurezza del sistema di esecuzione degli script.

Compattezza dell'architettura

La struttura di Rootless sshlirpCI si basa sull'idea che *Quae sunt Thread-is, Thread-i*, ovvero tutto ciò che sia di competenza dei thread builder debba essere svolto da questi il più compattamente possibile e con minor consumo di risorse.
In altre parole tutte le operazioni connesse all'esecuzione dei worker e "privatizzabili" in un loro corrispondente ambiente di lavoro, non devono risiedere all'interno del main, il quale, a causa della sua natura serializzata, le svolgerebbe più lentamente impiegando mezzi aggiuntivi non necessari.
Questa linea guida, assunta all'inizio della fase di progettazione, ha portato ha due effetti immediati:

1. Eliminazione della logica di gestione dei sorgenti basata sulla combinazione di clone iniziale sull'host, copia e rimozione post-build nei rootfs;
2. Unificazione delle operazioni di creazione dei rootfs, reperimento in essi dei sorgenti o di loro aggiornamenti, build e delivery dei binari statici di sshlirp in un unico script invocato da ogni thread builder.

Il primo punto, sebbene privi Rootless sshlirpCI di una gestione centralizzata ed economica dal punto di vista dello storage e del traffico di rete, solleva l'intero sistema non solo da inutili operazioni di I/O ma anche da tutte le vulnerabilità e pericolosità di incoerenza discusse ampiamente nella sezione 3.2.2.

Il secondo punto, invece, aggrega sì molte operazioni in un'esecuzione poco granulare e quindi meno scalabile, ma evita che vengano aperti e chiusi troppo frequentemente processi figlio per l'esecuzione dei componenti `.sh` e al coltempo che sia il main a occuparsi sia del reperimento dei sorgenti di `sshlirp` e `libslirp` che dello spostamento dei binari finali nella target directory versionata, entrambi compiti di competenza dei thread in quanto coinvolgono directory e file "privatizzabili" e contestuali alla loro esecuzione.
Questa ristrutturazione architetturale, visibile dal diagramma nella figura 3.3, è stata quindi implementata attraverso lo spostamento di tali operazioni in un unico

componente script denominato `cross_compilation_engine.sh` che contenesse anche il setup dei rootfs tramite l'invocazione di `rootless-debootstrap-wrapper` e l'esecuzione della build di sshlirp e libslirp.

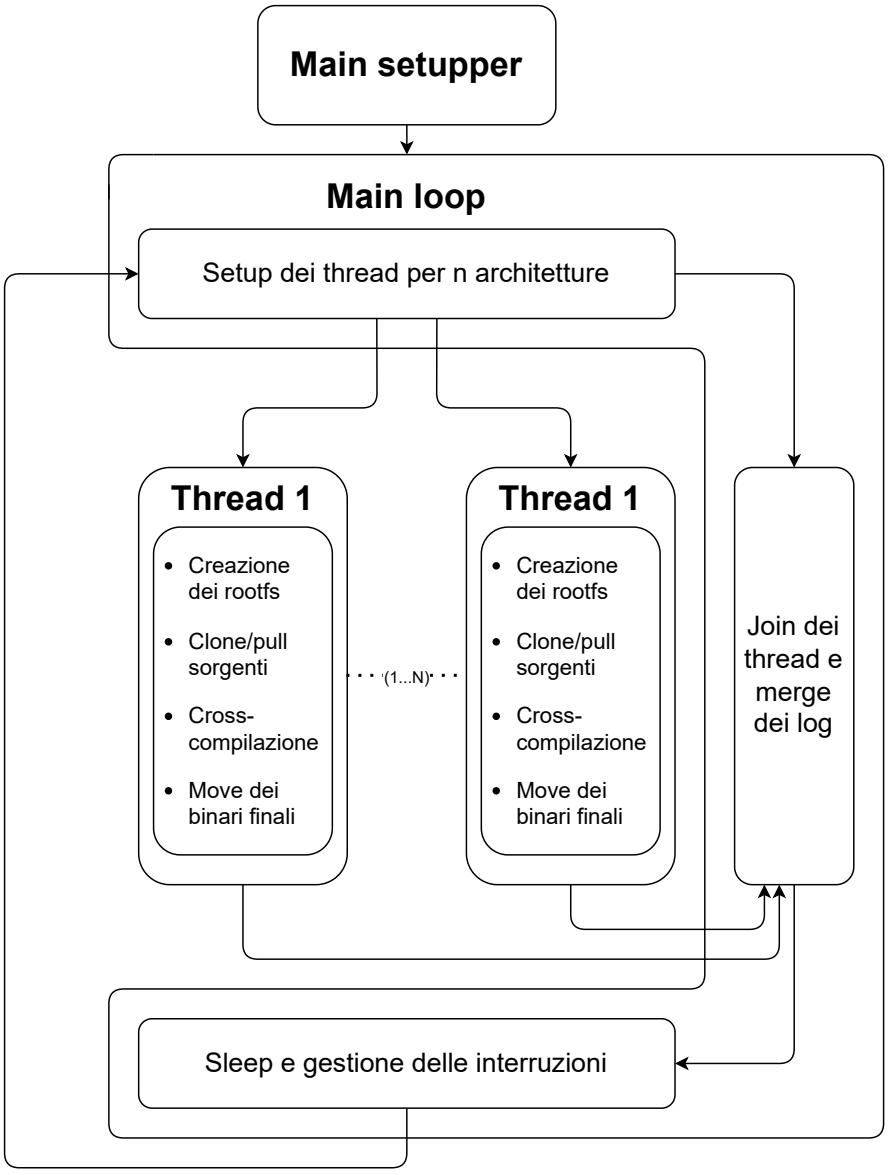

Figura 3.3: Diagramma architetturale di Rootless sshlirpCI

La drastica unificazione delle operazioni ha poi permesso di apportare ulteriori ottimizzazioni, quali:

- far convergere in un unico log file sull'host, dedicato a un singolo thread, i

logs prodotti dall'esecuzione di tale worker sia interna che esterna al rootfs corrispondente;

- gestire il sistema di sincronizzazione per la creazione dei rootfs con un lock file globale (piuttosto che con un mutex `c`), generato appena un'istruzione prima dell'invocazione di `rootless-debootstrap-wrapper` e rilasciato subito dopo la sua conclusione, permettendo così ad ogni thread di minimizzare il tempo di attesa per l'acquisizione del lock;
- amministrare la logica di versionamento della directory target attraverso l'estrazione del tag git corrente di sshlirp all'interno dello stesso script di build, evitando così di dover passare tale informazione da un componente di update `.sh` a un file di versioning, al thread builder e infine al main.

Tutti questi perfezionamenti sono stati quindi implementati all'interno di `cross_compilation_engine.sh`, il cui codice principale è riportato di seguito:

```
#!/bin/bash
...
debian_arch=$1
sshlirp_build_dir=$2
target_host_dir=$3
thread_log_file=$4
...
pull_round=1
...
exec >> "$thread_log_file" 2>&1
...
# Directory assoluta in cui risiede questo script
SCRIPT_DIR="$(cd -- "$(dirname -- "${BASH_SOURCE[0]}")"
  >/dev/null 2>&1 && pwd)"
WRAPPER="$SCRIPT_DIR/rootless-debootstrap-wrapper.sh"
...
# Lock file globale per la sola fase di creazione rootfs
LOCKFILE="$sshlirp_build_dir/.chroot_setup.lock"
if [ ! -d "$sshlirp_build_dir/$debian_arch-chroot" ]; then
  exec {lockfd}> "$LOCKFILE"
  flock "$lockfd"
  echo "[From cross_compilation_engine.sh for $debian_arch arch]
    Creating rootfs at $sshlirp_build_dir/$debian_arch-chroot"
```

```
  "$WRAPPER" --target-dir="$sshlirp_build_dir/$debian_arch-chroot"
    --arch="$debian_arch" --suite "$suite" --include=build-essential
  ...
  pull_round=0
  exec {lockfd}>&-
fi
if [ $pull_round -eq 0 ]; then
  $sshlirp_build_dir/$debian_arch-chroot/_enter <<EOF
  echo "[From cross_compilation_engine.sh inside $debian_arch-chroot
    rootfs] First pull round: Installing required packages"
  apt install -qq --assume-yes git meson cmake pkg-config
    libglib2.0-dev libvdeplug-dev
EOF
  ...
  echo "[From cross_compilation_engine.sh for $debian_arch arch]
    Cloning sshlirp and libslirp repositories"
  (cd $sshlirp_build_dir/$debian_arch-chroot/root;
    git clone https://gitlab.freedesktop.org/slirp/libslirp.git
  )
  ...
  (cd $sshlirp_build_dir/$debian_arch-chroot/root;
    git clone https://github.com/virtualsquare/sshlirp.git
  )
  ...
fi
if [ $pull_round -eq 1 ]; then
  $sshlirp_build_dir/$debian_arch-chroot/_enter <<EOF
  echo "[From cross_compilation_engine.sh inside
    $debian_arch-chroot rootfs] installing updates"
  apt-get update
  apt-get upgrade -qq --assume-yes
EOF
  ...
  echo "[From cross_compilation_engine.sh for $debian_arch arch]
    Pulling latest changes for sshlirp and libslirp"
  (cd $sshlirp_build_dir/$debian_arch-chroot/root/sshlirp; git pull)
  ...
  (cd $sshlirp_build_dir/$debian_arch-chroot/root/libslirp; git pull)
```

```
  ...
fi
cd $sshlirp_build_dir/$debian_arch-chroot/root/sshlirp
current_tag=$(git describe --tags --abbrev=0)
if [ ! -n "$current_tag" ]; then
  current_tag="unstable"
fi
echo "[From cross_compilation_engine.sh for $debian_arch arch]
  Starting build process"
$sshlirp_build_dir/$debian_arch-chroot/_enter << TAG
echo "[From cross_compilation_engine.sh inside $debian_arch-chroot
  rootfs] Building libslirp"
cd /root/libslirp
meson build . --default-library=both
ninja -C build install
TAG
...
$sshlirp_build_dir/$debian_arch-chroot/_enter << TAG
echo "[From cross_compilation_engine.sh inside $debian_arch-chroot
  rootfs] Building sshlirp"
mkdir -p /root/sshlirp/build
cd /root/sshlirp/build
cmake ..
make
TAG
...
# Verifica che il binario sia stato staticamente linkato
...
echo "[From cross_compilation_engine.sh for $debian_arch arch]
  Copying sshlirp binaries to $target_host_dir/v-$current_tag"
if [ ! -d "$target_host_dir/v-$current_tag" ]; then
  mkdir -p "$target_host_dir/v-$current_tag"
fi
cp $binary "$target_host_dir/v-$current_tag"
exit 0
```

Lo spostamento di tutta la logica di esecuzione dei thread builder in questo singolo componente `.sh` ha permesso di riflesso di alleggerire notevolmente il corpo "esterno"

dei worker, contenuto nel file `worker.c`.

```
void *build_worker(void *arg_ptr) {
  thread_args_t* args = (thread_args_t*)arg_ptr;
  int *result = malloc(sizeof(int));
  ...
  int script_status = execute_build_script_for_thread(
    args->arch,
    args->build_dir,
    args->target_dir,
    args->thread_log_file,
    test_enabled,
    thread_log_fp
  );
  if (script_status != 0) {
    fprintf(thread_log_fp, "[Thread %s] Compile script failed
      with status: %d\n", args->arch, script_status
    );
    *result = 1;
  }
  ...
  return result;
}
```

Persistenza delle risorse

L'immediato side effect della ristrutturazione dell'architettura di Rootless sshlirpCI è stata la risoluzione delle criticità legate all'amministrazione non coerente delle risorse di build in sshlirpCI, già criticata nella sezione 3.2.2.
Infatti, lo spostamento della maggior parte delle operazioni "thread-ful" - compresa quella di setup dei rootfs - all'interno di un unico script esguito interamente da ogni worker e per ogni iterazione del main loop, sebbene abbia introdotto una ridondanza logica nei check di esistenza e validità delle risorse, ha permesso di garantire l'integrità di quest'ultime anche per le iterazioni di esecuzione avviata, le quali, sempre grazie all'uso del `pull_round` simulato dai thread e grazie ai check su di esso e sull'esistenza dei rootfs, rimangono anche idempotenti.

Non modificabilità dell'albero del filesystem interno alle risorse di build

Sulla scia della semplificazione architetturale e dell'ottimizzazione delle delle risorse introdotta dalle due precedenti sottosezioni, si è deciso di scremare anche il grado di configurabilità applicabile dall'utente esterno al sistema di continuous integration.
Infatti, come già descritto nella sezione 3.2.2, sshlirpCI permetteva, troppo lascamente, di configurare l'ambiente di build modificando arbitrariamente le variabili contenute nel file `ci.conf`, le quali però sarebbero poi state utilizzate per costruire file e directory anche innestate.
Questo approccio, sebbene avesse il pregio di essere estremamente flessibile, presentava il difetto di esporre il sistema a potenziali errori di configurazione e a incongruenze tra le variabili settate e l'effettiva struttura del filesystem dell'host.
Per questo motivo, in Rootless sshlirpCI, si è deciso di limitare la configurabilità esterna alle sole variabili `BUILD_DIR`, `TARGET_DIR`, `POLL_INTERVALL` e `ARCHITECTURES`.

```
BUILD_DIR=/home/user/rootless_sshlirpCI
TARGET_DIR=/home/user/rootless_sshlirpCI/binaries
POLL_INTERVAL=3600 # secondi -> 1 ora
ARCHITECTURES=amd64,arm64,armhf,riscv64
```

. La costruzione dei path delle risorse di build più "interne" è stata invece delegata all'esecuzione dei setupper del main e dei threads.

```
int main() {
  // 0. Caricamento delle variabili dal file di configurazione
  ...
  if(conf_vars_loader(
      archs_list,
      &num_archs,
      build_dir,
      target_dir,
      &poll_interval) != 0
  ) {
    fprintf(stderr, "Failed to load configuration
      variables. Exiting.\n"
    );
    return 1;
```

```
}
printf("Configuration loaded successfully.\n");
// Variabili da passare ai threads e costruibili dalle
// precedenti
char release_file[CONFIG_ATTR_LEN];
char log_dir[CONFIG_ATTR_LEN];
char log_file[CONFIG_ATTR_LEN];
char thread_log_dir[CONFIG_ATTR_LEN];
snprintf(release_file, sizeof(release_file), "%s/release.txt",
  build_dir
);
snprintf(log_dir, sizeof(log_dir), "%s/log", build_dir);
snprintf(log_file, sizeof(log_file), "%s/log/main_sshlirp.log",
  build_dir
);
snprintf(thread_log_dir, sizeof(thread_log_dir), "%s/log/threads",
  build_dir
);
...
daemonize();
...
// 3. Creazione dei file principali (se non esistono):
// - la directory fondamentale
//    (/home/user/rootless_sshlirpCI)
// - il file di release
//    (/home/user/rootless_sshlirpCI/release.txt)
// - la directory dei log principali
//    (/home/user/rootless_sshlirpCI/log)
// - il file di log principale
//    (/home/user/rootless_sshlirpCI/log/main_sshlirp.log)
// - la directory dei log dei thread
//    (/home/user/rootless_sshlirpCI/log/threads)
...
// 5. Avvio del loop principale nel demone
while (1) {
  ...
  // 7.1. Preparazione dei thread
  pthread_t threads[num_archs];
```

```
    thread_args_t args[num_archs];
    // 7.2. Avvio dei thread di build
    for (int i = 0; i < num_archs; i++) {
      // Copia sicura del nome dell'architettura
      strncpy(args[i].arch, archs_list[i], sizeof(args[i].arch) - 1);
      args[i].arch[sizeof(args[i].arch) - 1] = '\0';
      // Copia sicura del build_dir
      snprintf(args[i].build_dir, sizeof(args[i].build_dir),
        "%s", build_dir
      );
      // Copia sicura della target_dir
      snprintf(args[i].target_dir, sizeof(args[i].target_dir),
        "%s", target_dir
      );
      // Copia sicura del thread_log_file (ossia il log file su cui
      // scrivera' il thread)
      snprintf(args[i].thread_log_file, sizeof(args[i].thread_log_file),
        "%s/%s-thread.log", thread_log_dir, archs_list[i]
      );
      // Creazione del file di log del thread
      FILE* thread_log_fp = fopen(args[i].thread_log_file, "a");
      ...
      if (pthread_create(&threads[i], NULL, build_worker, &args[i])
        != 0
      ) {
        fprintf(log_fp, "Error: Error creating thread for
          architecture %s.\n", args[i].arch
        );
        return 1;
      } else {
        fprintf(log_fp, "Thread created successfully for
          architecture %s.\n", args[i].arch
        );
      }
    }
    ...
  }
}
```

Implementazione di `rootless_sshlirp_ci_instant_killer`

Un'altra importante aggiunta a Rootless sshlirpCI riguarda l'introduzione di un meccanismo sicuro per l'interruzione del demone in qualsiasi fase della sua esecuzione.
Infatti, come seposto nella sezione 3.2.2, sshlirpCI non prevedeva alcun modo per terminare in sicurezza il demone una volta avviato, se non attraverso l'uccisione forzata del processo stesso tramite `kill -9 <PID>`.
Questo approccio, oltre a essere poco elegante, non permetteva di eseguire preliminarmente operazioni di cleanup o di rilascio delle risorse in uso, rischiando così di impedire il riavvio del demone.
Per colmare questa lacuna è stato sviluppato un componente accessorio, la cui compilazione tramite `cmake` dà origine all'eseguibile
`rootless_sshlirp_ci_instant_killer`, e che, dopo un primo tentativo di interruzione tramite `SIGTERM`, ricorre all'invio del segnale `SIGKILL` al demone di Rootless sshlirpCI e ne attende, per un numero di secondi fissato, la terminazione definitiva.

```
#define TERM_WAIT_SECONDS 10
#define CHECK_INTERVAL_MS 200
#define KILL_WAIT_SECONDS 2
...
int main(void) {
  ...
  // Primo tentativo: SIGTERM
  // (terminazione sicura -> permette al codice di chiudere
  // risorse se intercetta il segnale)
  if (kill(daemon_pid, SIGTERM) != 0) {
    fprintf(stderr, "Errore nell'invio di SIGTERM a %d: %s\n",
      daemon_pid, strerror(errno)
    );
  } else {
    printf("SIGTERM inviato. Attendo fino a %d secondi...\n",
      TERM_WAIT_SECONDS
    );
    struct timespec ts;
    ts.tv_sec = 0;
    ts.tv_nsec = CHECK_INTERVAL_MS * 1000000L;
    int waited_ms = 0;
```

```
    int max_wait_ms = TERM_WAIT_SECONDS * 1000;
    while (waited_ms < max_wait_ms) {
      if (!process_alive(daemon_pid)) {
        printf("Daemon terminato dopo SIGTERM (%d ms).\n", waited_ms);
        goto cleanup;
      }
      nanosleep(&ts, NULL);
      waited_ms += CHECK_INTERVAL_MS;
    }
    printf("Il daemon non e' terminato entro %d secondi dopo
      SIGTERM.\n", TERM_WAIT_SECONDS
    );
  }
  // Escalation: SIGKILL
  // Se il processo e' ancora vivo, ripeto il processo di invio
  // del segnale SIGKILL e attesa di KILL_WAIT_SECONDS
  ...
cleanup:
  // Pulizia pid e state file
  ...
  return 0;
}
```

Esecuzione diretta degli script tramite `system_safe()`

Per eliminare le vulnerabilità connesse alla generazione a run-time di script, al loro posizionamento in `/tmp` e alla loro esecuzione tramite `system()`, di cui si è già ampiamente discusso nella sezione 3.2.2, si è deciso di tornare a considerare l'esecuzione diretta degli script, anche se a discapito dei vantaggi offerti dalla precedente soluzione e resi noti nella sezione 3.2.1.
Infatti, anche a seguito della drastica riduzione di script esterni, si è pensato che la loro esecuzione diretta e il loro salvataggio in file `.sh` permanenti all'interno della stessa directory dei sorgenti di Rootless sshlirpCI, avrebbero non solo potenziato le prestazioni del demone ma anche assicurato che non potessero essere manomessi da altri utenti o processi in esecuzione sullo stesso host ma privi dei permessi necessari (assumendo che l'utente che intenda eseguire Rootless sshlirpCI abbia clonato il repository corrispondente in una directory di sua proprietà e non abbia spostato

la directory `script/` in una destinazione accessibile ad altri utenti).
Per implementare questa soluzione si è reso necessario l'impiego di variabili globali, che contenessero il path assoluto del principale script di build `cross_compilation_engine.sh`, definite all'interno del file `src/includes/types/types.h`:

```
#ifndef TYPES_H
#define TYPES_H
#define ROOTLESS_SSHLIRPCI_SOURCE_DIR "path/to/rootless_sshlirpCI"
...
#define DEFAULT_CONFIG_PATH ROOTLESS_SSHLIRPCI_SOURCE_DIR "/ci.conf"
#define CROSS_COMPILATION_SCRIPT_PATH
  ROOTLESS_SSHLIRPCI_SOURCE_DIR "/script/cross_compilation_engine.sh"
...
#endif // TYPES_H
```

Inoltre, mentre l'impiego di `system()` introduceva ulteriori fragilità in sshlirpCI, in Rootless sshlirpCI la garanzia di non incorrere in attacchi di *command injection* o in errori dovuti ad ambienti "sporchi" o corrotti, è stata data dall'uso di una funzione di utilità denominata `system_safe()`.
Questo metodo, definito nel file `execs.h` del progetto Virtualsquare `s2argv-execs` [60] e implementato come macro del metodo `_system_common()`, imita il comportamento di `system()` seguendo però i suggerimenti guida di Debian per l'esecuzione sicura di comandi di shell da codice C, ossia tramite la creazione di `argv` e l'invocazione diretta di `execv()` [48].

Sebbene l'architettura finale di Rootless sshlirpCI e i suoi perfezionamenti abbiano permesso di superare tutti i limiti di sshlirpCI, questo sistema di cross-compilazione aggiornato e semplificato rimaneva ancora privo di un'importante componente utile alla verifica e alla distribuzione dei binari prodotti: il testing automatico.

3.4 Testing dei binari

La crescente aderenza di Rootless sshlirpCI all'obbiettivo di un'infrastruttura che permettesse la consegna di binari statici di sshlirp per port Debian multipli, pronti per essere distribuiti e impiegati su host remoti per usufruire delle funzionalità

native di sshlirp stesso, ha spinto a integrare nel sistema già esistente un componente aggiuntivo che testasse automaticamente gli eseguibili prodotti, al fine di garantire non solo portabilità e compabilità, ma anche correttezza funzionale.

3.4.1 L'uso di vdens

Come già anticipato nella sezione 1.2, sshlirp, una volta copiato sul server remoto, che fungerà quindi da gateway, NAT e VPN provider, è pensato per essere "pluggato", per mezzo di `libvdeplug4`, a un network namespace creato sul client VDE tramite `vdens`.
L'immagine sottostante rappresenta lo scenario di utilizzo di `vdens` combinato a `slirp`, pensato per dare connettività esterna a un nodo virtuale "pluggandone" un network namespace VDE a un'istanza di `slirp` in esecuzione sul medesimo host.

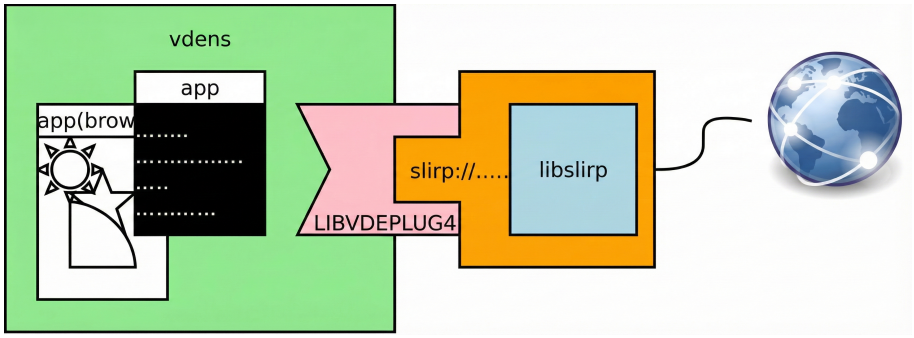

Figura 3.4: Scenario di utilizzo di slirp con vdens per connettività esterna su uno stesso host

Sshlirp non fa altro che spostare il provider, replicando quindi lo scenario rappresentato in figura, con la sola differenza di esecuzione su un server remoto piuttosto che sullo stesso host del client VDE, fornendo conseguentemente un servizio di VPN e NAT istantanei.
Normalmente, per ottenere questo risultato, come indicato dal repository ufficiale di `sshlirp` [7], si svolgono in sequenza le seguenti operazioni:

```
# [1] Copia dell'eseguibile statico di sshlirp sull'host remoto:
$ scp sshlirp-x86_64 remote.mydomain.org:/tmp/sshlirp
# [2] Creazione del network namespace VDE sul client e
# connessione di esso al server sshlirp:
$ vdens -R 9.9.9.9 cmd://"ssh remote.mydomain.org /tmp/sshlirp"
# [3] Configurazione dell'indirizzo ip (garantita dal servizio di
```

```
# DHCP offerto sempre da sshlirp):
$ /sbin/udhcpc -i vde0
# [4] Verifica della connettivita' esterna:
$ ping -c 4 8.8.8.8
```

3.4.2 Considerazioni sulle difficoltà di testing dei binari in Rootless sshlirpCI

Lo scopo di automatizzare la fase di testing dei binari prodotti da Rootless sshlirpCI, non ha trovato una soluzione immediata.
La natura stessa del progetto prevede infatti che Rootless sshlirpCI stesso possa essere eseguito da chiunque, richiedendo i soli prerequisiti di possesso di un host Debian-based e di connettività a internet.
La necessità di testare i binari prodotti invece, nello scenario standard esposto nella precedente sottosezione, richiede anche l'accessibilità a un host remoto tramite SSH.
Per aggirare questo vincolo, si è deciso di inserire testing automatico di sshlirp tramite una procedura più "(vde + slirp)-like", ossia mantenendo lo stack simulato da sshlirp sullo stesso host su cui esegue Rootless sshlirpCI e avviandolo tramite l'interfaccia `cmd://` dopo la creazione di un network namespace VDE tramite `vdens`, connesso all'interfaccia stessa.
Infatti, come lasciato intuire nella wiki di Virtualsquare [1], l'avvio di un network namespace VDE collegato, grazie all'interfaccia `cmd://`, a sshlirp, permette di far fluire i pacchetti della rete virtuale sul canale di stdin/stdout verso il processo di sshlirp stesso, il quale procederà a raccoglierli e restituire le risposte corrispondenti.

Questa scelta ha introdotto un problema addizionale, che però è stato risolto più rapidamente e con il riciclo di tecnologie già impiegate.
Il tentativo di avviare più eseguibili di sshlirp cross-compilati per port multipli su uno stesso host, infatti, ha in un primo momento portato a pensare che fosse necessario svolgere la fase di test per ogni architettura internamente al rootfs corrispondente, in modo da garantire la corretta esecuzione del binario nel suo ambiente nativo.
Questa supposizione però si è subito scontrata con quanto detto, nella sezione 3.3.1, riguardo le limitazioni trascurabili introdotte da `fakeroot` durante l'invocazione di `debootstrap`.
Infatti, l'assenza di device di rete "reali" negli ambienti chroot non privilegiati ha

impedito l'invocazione di `vdens` e le operazioni di test basate su `ping`; d'altra parte il requisito imprescindibile di rootlessness ha costretto a escludere la creazione di questi in un secondo momento.

Tutti questi apparenti limiti sono stati però superati dalla memoria di Qemu User-Mode Emulation.
Infatti, il semplice riutilizzo dei binari `qemu-<arch>-static` e la loro registrazione come interpreti per mezzo di `binfmt_misc` hanno permesso di dedurre che la fase di testing di sshlirp potesse essere eseguita direttamente sull'host, senza dover ricorrere ai rootfs.

3.4.3 Soluzione adottata

A seguito delle precedenti scelte e deduzioni, si è quindi deciso di inserire nel file `src/includes/types/types.h` una variabile aggiuntiva `TEST_ENABLED` e, in caso di abilitazione del testing tramite questa, di eseguire le operazioni esposte nella precedente sotto-sezione direttamente a termine del componente principale dell'esecuzione dei thread worker `cross_compilation_engine.sh`.

```
#!/bin/bash
...
debian_arch=$1
sshlirp_build_dir=$2
target_host_dir=$3
thread_log_file=$4
test_enabled=$5
...
pull_round=1
...
exec >> "$thread_log_file" 2>&1
...
# Fase di testing sull'host
if [ "$test_enabled" == "true" ]; then
  echo "[From cross_compilation_engine.sh for $debian_arch arch]
    Running test for $binary"
  # Create a vde network namespace and connect it
  # to sshlirp through cmd://
  vdens cmd://"$binary" /bin/bash <<EOF
```

```
  ip a
  # Configure the vde0 interface with a static IP
  ip addr add 10.0.2.15/24 dev vde0
  ip link set vde0 up
  # Ping the sshlirp default gateway to test connectivity
  ping -c 4 10.0.2.2
EOF
  if [ $? -eq 0 ]; then
    echo "[From cross_compilation_engine.sh for $debian_arch arch]
      Test for $binary passed"
  else
    echo "[From cross_compilation_engine.sh for $debian_arch arch]
      Test for $binary failed"
  fi
fi
echo "[From cross_compilation_engine.sh for $debian_arch arch]
  Copying sshlirp binaries to $target_host_dir/v-$current_tag"
...
exit 0
```

Il raggiungimento di quest'ultimo obbiettivo ha completato il sistema di continuous integration di sshlirp, portandolo a un alto livello di maturità e di aderenza agli scopi prefissati.

La tentazione di scalare orizzontalmente Rootless sshlirpCI al fine di ottenere un motore di cross-compilazione sincrono per più progetti Virtualsquare - che avessero gli stessi requisiti di build - e altamente disponibile, performante e sicuro è stata soddisfatta dal successivo sviluppo di Rootless V^2CI.

Capitolo 4

Rootless V^2CI e integrazione ELK

In quest'ultima fase di sviluppo ha preso forma, a partire dal lavoro svolto precedentemente, il disegno di un sistema di continuous integration generalizzato per progetti Virtualsquare multipli, in grado di produrne in parallelo binari cross-compilati staticamente, secondo le medesime logiche e linee guida adottate durante l'assemblaggio dell'antenato Rootless sshlirpCI.
Questo sistema, denominato Rootless V^2CI, predisposto alla scalabilità e alla distribuibilità sin dalla prima progettazione, è stato poi affiancato dall'ambiziosa idea di rendere il suo impiego e, in particolare, il suo monitoraggio di esecuzione più "user-friendly", attraverso un'integrazione con uno stack containerizzato e distribuito addetto alla trasmissione, all'ingestion e alla creazione di visualizzazioni costum dei log prodotti, basato sulle tecnologie ELK.
Il risultato finale di queste aspirazioni, raggiunto dopo un lungo processo evolutivo che ha spaziato dallo studio teorico dell'architettura alla scelta di strumenti e tecniche implementative che ne permettessero la più alta affidabilità, ha incarnato il sogno di rendere fruibile e per di più intuitivo l'uso di un sistema di continuous integration avanzato, il quale non solo è stato pensato per garantire massima persistenza, configurabilità e performance, ma anche per essere distribuito e avviato su più server, comunque con la possibilità di una supervisione centralizzata.

Come quanto fatto per le fasi evolutive precedenti, in particolar modo per questo consistente snodo terminale, risulta utile rappresentare il processo di disegno e sviluppo seguito attraverso un diagramma riassuntivo.

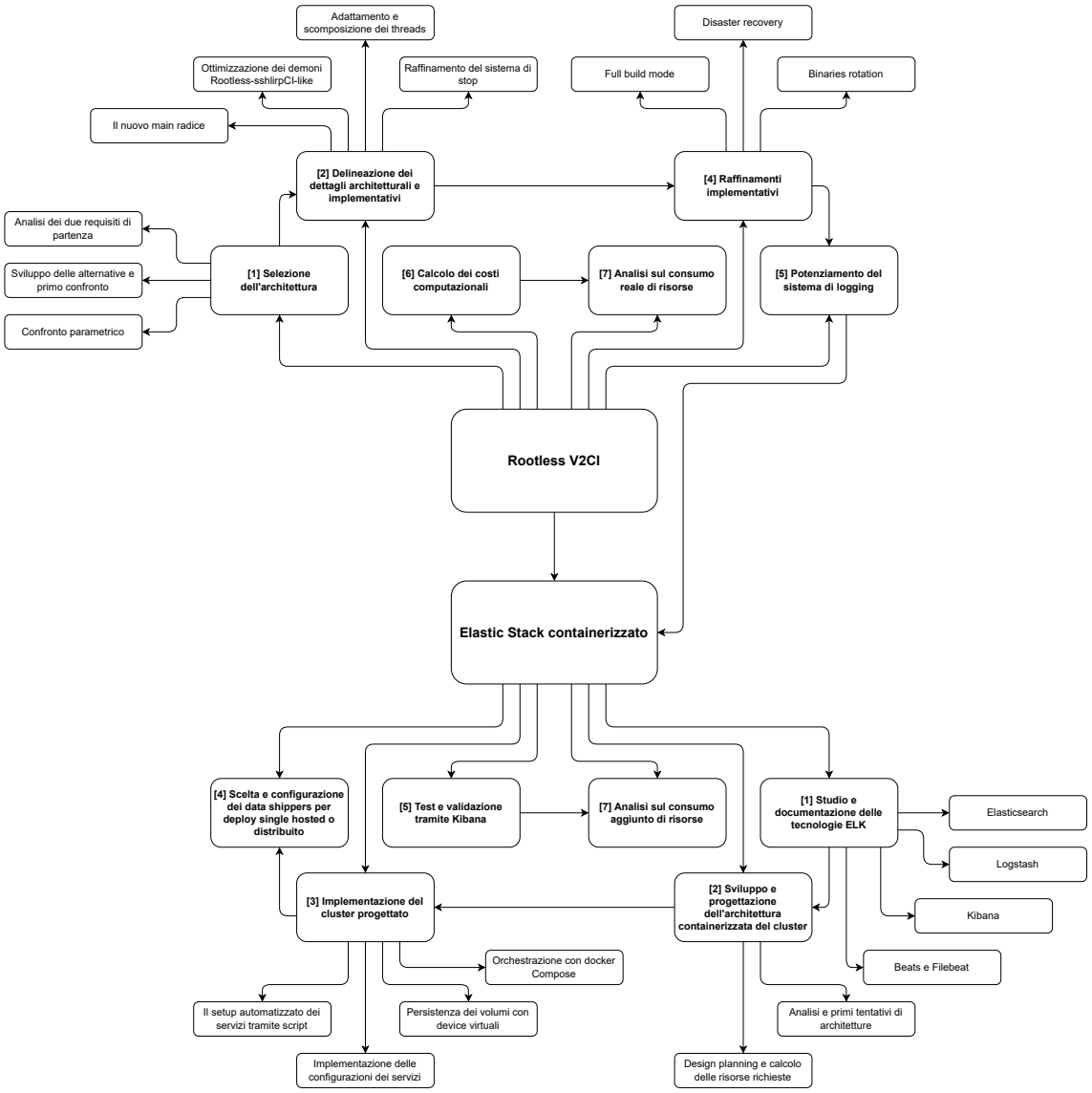

Figura 4.1: Diagramma riassuntivo del processo di design e sviluppo di Rootless V^2CI e della sua integrazione con ELK

4.1 Rootless V^2CI: potenziamento ed espansione di Rootless sshlirpCI

Il primo passo verso la realizzazione di questo complesso ecosistema che integra cross-compilazione e delivery di binari statici con monitoraggio e visualizzazione avanzata dei log, è stato il perfezionamento e l'espansione orizzontale di Rootless sshlirpCI.

Come già anticipato nel precedente capitolo, Rootless sshlirpCI, o per lo meno le

sue fondamenta, hanno costituito i mattoni su cui è stato edificato Rootless V^2CI. Questo potente motore è infatti basato sull'idea di avviare in parallelo, a seguito di una lettura e di un parsing di un denso file di configurazione, in cui siano specificati i progetti su cui l'utente intende avviare la build e per ognuno di essi dettagli "privati" e scelte su come svolgere quest'ultima, processi "Rootless sshlirpCI-like" multipli, ognuno "angelo custode" di uno dei progetti dati in input dall'utente.
Un'espansione orizzontale e sincrona di questo calibro è figlia dello scontro con numerose difficoltà e di svariate scelte sia nel disegno che nell'implementazione del sistema, in quanto, per preservare sicurezza e performance, si è reso necessario considerare nuovi parametri di sviluppo quali la condivisione di risorse e il parallelismo innestato.

4.1.1 Il disegno: scelte strutturali e difficoltà affrontate

Il primo grande ostacolo fronteggiato durante lo sviluppo di Rootless V^2CI è stato quello di definizione dell'architettura.
Prima che quest'ultima fosse tracciata, infatti, non era ancora chiaro quali componenti avrebbero svolto quali compiti e in che momento dell'esecuzione sarebbero stati avviati.
Due soli requisiti erano certi:

1. l'utente finale sarebbe dovuto essere in grado di impostare in modo semplice e granulare il comportamento di Rootless V^2CI, avviarlo e ottenere i binari statici per le architetture specificate per ogni progetto di input, circa nello stesso tempo di esecuzione di Rootless sshlirpCI;
2. il nuovo ecosistema di Rootless V^2CI avrebbe dovuto riutilizzare i componenti sviluppati per Rootless sshlirpCI, appoggiandosi alle sue solide fondamenta e al suo collaudato flusso di esecuzione.

Ciò che ha risieduto alla base delle difficoltà legate alla progettazione concettuale e che ha al coltempo guidato le scelte che ne hanno permessa la risoluzione, è stata infatti la volontà di garantire all'utente finale un grado di configurabilità tale da poter vedere Rootless V^2CI come incarnazione di un sistema di CI generalizzato per qualsiasi sorgente.
Inoltre, il desiderio di riciclare le idee e lo sviluppo alla base di Rootless sshlirpCI ha automaticamente spinto a prediligere una pianificazione che è rimasta alla base del risultato finale e che si è dimostrata, anche a seguito di valutazioni successive, più aderente logicamente al primo requisito e maggiormente performante rispetto

ad alternative che sembravano inizialmente paritarie.

Riassumendo, è possibile affermare che il processo di selezione dell'architettura di Rootless V^2CI si è basato sulle segenti fasi:

1. Deduzione delle conseguenze logiche del primo requisito di configurabilità: forma dell'input, ossia del file di configurazione;
2. Delineazione dello scheletro generale dell'architettura a partire dal tipo di composizione dell'input;
3. Progettazione concettuale di due possibili alternative architetturali che avrebbero permesso la corretta elaborazione dell'input, guidata dalla forma di quest'ultimo e dalla struttura dello scheletro generale di Rootless V^2CI;
4. Analisi comparativa delle possibili soluzioni sulla base dei parametri di performance, coerenza logica con l'input e re-use dei componenti di Rootless sshlirpCI.

È inoltre possibile riassumere e illustrare tale procedimento analitico nel seguente diagramma.

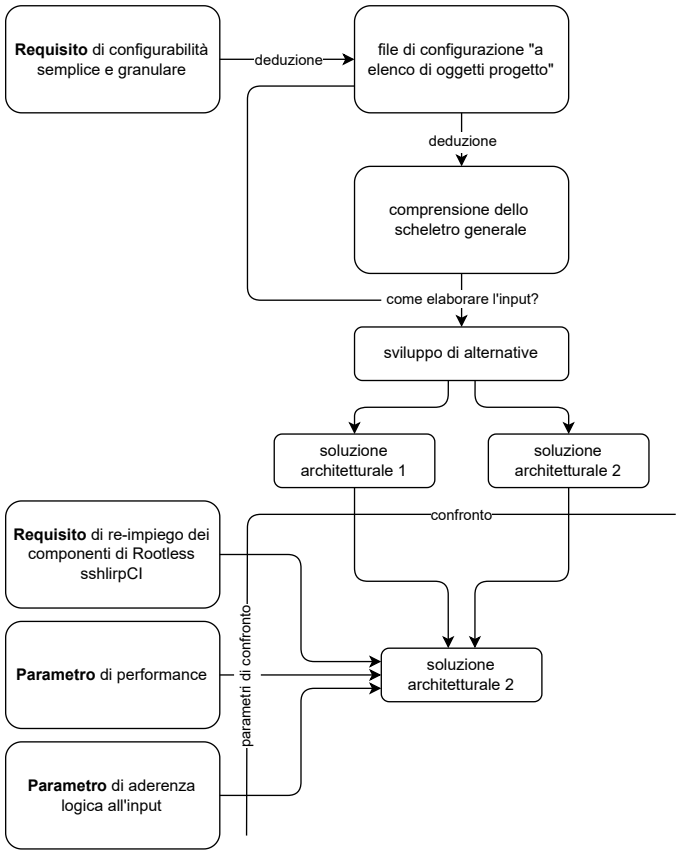

Figura 4.2: Diagramma riassuntivo del processo di design architetturale di Rootless V^2CI

Le origini del progetto architetturale: configurabilità e forma dell'input

Scendendo nei dettagli del primo requisito, ancor prima del disegno architetturale si era deciso che il cliente di Rootless V^2CI dovesse essere in grado di specificare il path della build directory e per ogni progetto di cui intendeva ottenere i binari, non solo le architetture di destinazione, il poll interval e il path della target directory, ma anche un elenco di sorgenti da compilare - che comprendeva l'URL del repository principale del progetto stesso e quelli delle sue eventuali dipendenze "manuali" -, i relativi sistemi di build (`cmake`, `make`, `meson`, ecc.), una lista di pacchetti aggiuntivi da installare preliminarmente nei rootfs e il tipo di trigger per l'avvio della compilazione - che si poteva configurare come dipendente da aggiornamenti al sorgente principale o ai progetti linkati.
Da tutto ciò è stato immediato intuire che il formato che avrebbe garantito all'utente massima facilità di configurazione era `yml` e che quindi l'eventuale file di input

risultante avrebbe avuto la forma di un elenco di oggetti "progetto" contenenti numerosi componenti innestati, come mostra il seguente esempio di `config.yml` per la sola cross-compilazione di `sshlirp`:

```
# Directory in cui verranno salvati gli ambienti rootfs,
# i log e gli artefatti di build
# (l'utente deve avere permessi di scrittura qui)
build_dir: /home/francesco/v2ci_build
projects:
  - name: sshlirp
    # Directory in cui verranno salvati i binari statici finali
    # (l'utente deve avere permessi di scrittura qui)
    target_dir: /home/francesco/sshlirp_build/target_binaries
    source:
      main_repo:
        git_url: https://github.com/virtualsquare/sshlirp
        build_system: cmake
      dependencies:
        - pkg-config
        - libglib2.0-dev
        - libvdeplug-dev
      dependency_repos:
        - git_url: https://gitlab.freedesktop.org/slirp/libslirp.git
          build_system: meson
          # Lista dei pacchetti APT da installare all'interno di
          # ogni ambiente rootfs
          # per questa dipendenza
          dependencies:
            - meson
            - ninja-build
            - libssl-dev
            - libglib2.0-dev
            - libexpat1-dev
            - libcap-ng-dev
            - libseccomp-dev
    build_config:
      # Modalita' di build supportate: main
      # (build solo se il repo principale ha nuovi commit),
```

```
    # dep (build se un qualsiasi repo di dipendenza ha nuovi commit)
    build_mode: main
    # Intervallo di tempo (in secondi) tra due controlli consecutivi
    # per nuovi commit
    poll_interval: 3600
  architectures:
    - amd64
    - arm64
    - armhf
    - riscv64
```

Prime deduzioni generali e questioni cardine per la scelta del disegno

Sia dall'idea iniziale di realizzare un sistema di CI per cross-compilazione sincrona per architetture e progetti multipli che dal primo requisito di configurabilità e dalle conseguenti deduzioni sulla forma dell'eventuale file di configurazione, si è intuito che Rootless V^2CI avrebbe di certo assunto la forma di un processo, dall'esecuzione limitata, che avrebbe avviato iterativamente sotto-processi, ognuno dei quali padre a sua volta di threads multipli, identificati quindi da una coppia - non ancora ordinata - `<prj, arch>`, ossia dal loro compito personale di cross-compilazione di un progetto per una specifica architettura.
Le domande a cui però mancava ancora una risposta erano:

- A quale "contenitore concettuale" sarebbero corrisposti i worker? Alle architetture coinvolte in tutto il processo di cross-build sincrono o ai progetti specificati dall'utente?
- I thread "foglia" di questo albero di esecuzione invece sarebbero stati avviati in parallelo per ogni progetto, occupandosi quindi della sua cross-compilazione per ogni suo port specificato, o per ogni architettura, cross-compilando perciò progetti diversi all'interno del medesimo rootfs?
- In che momento dell'esecuzione sarebbero stati setuppati gli ambienti chroot? Durante la vita del main o all'interno di ogni worker? Oppure sarebbe stato compito dei singoli thread figli della combinazione `<prj, arch>`?

E proprio dai primi due interrogativi sono nate le principali due alternative architetturali concrete, su ognuna delle quali si è svolta un'analisi comparativa servendosi dei parametri introdotti dal terzo interrogativo e, come anticipato, dal vincolo di re-use dei componenti di Rootless sshlirpCI e dall'ambizione di performance ottimizzate.

Prima soluzione: worker per architettura e threads per progetto

Il seguente seguente diagramma concettualizza alla perfezione la prima alternativa strutturale presa in considerazione per Rootless V^2CI.

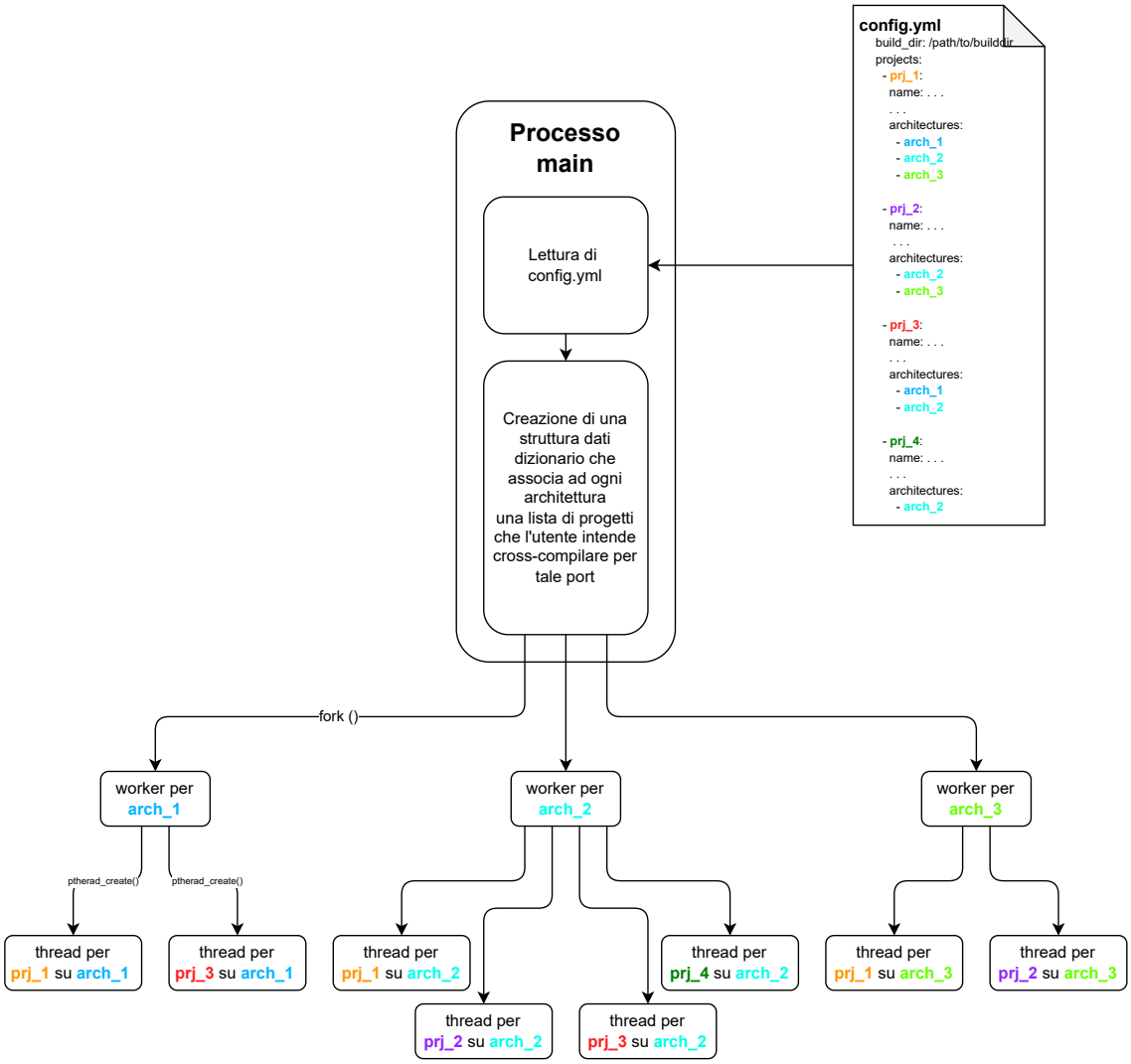

Figura 4.3: Schema concettuale della prima alternativa architetturale di Rootless V^2CI: worker per architettura e threads per progetto

Questa soluzione architetturale si basava sull'idea di un processo main che, una volta completata la lettura del file di configurazione - di cui è ora evidente la forma "object oriented" - avviasse un numero di demoni corrispondenti al numero di architetture totali coinvolte nel processo di build. Questi si sarebbero poi occupati di lanciare un thread per ogni progetto che avrebbe quindi acquisito il ruolo di cross-compiler per quel solo sorgente all'interno di quel solo rootfs cross-debootstrap-ato. L'idea di un'architettura di questo tipo è stata inizialmente sviluppata e apprezzata

in quanto permetteva di isolare logicamente e implementativamente la gestione di ogni rootfs, affidandola a un demone dedicato ed evitando che processi diversi, o ancora peggio thread figli di worker diversi, potessero interferire tra loro e operare in ambienti condivisi. Infatti il timore di dover amministrare concorrenza e gestione delle risorse di disco tra processi non comunicanti tra loro - ossia non interni allo scope di un medesimo processo padre - aveva spinto a prediligere quest'architettura che spostava al livello di "threads fratelli" le problematiche derivanti dal parallelismo.

Sebbene un albero di esecuzione di questo tipo permettesse massimo isolamento e "privatizzazione", una sua valutazione più approfondita, riguardante in particolare lo studio delle scelte implementative che ne avrebbero permessa la realizzazione, ha fatto emergere scogli di contraddizione logica e di abbattimento delle prestazioni che hanno portato a ricercare una soluzione addizionale.
In particolare, l'analisi ha evidenziato le seguenti criticità:

- **Gestione di strutture dati complesse e ridondanti**: la forma "contro natura" della gestione dell'input - che considerava infatti le architetture come "contenitori concettuali" di progetti - non solo avrebbe violato le aspettative logiche dell'utente, ma avrebbe costretto, come si evince dal diagramma 4.3, a dover costruire strutture dati complesse per cui un eventuale oggetto `architecture`, input per ogni demone, avrebbe dovuto contenere una lista di oggetti `project` con i loro dettagli compreso un ridondante riferimento alle architetture per cui sarebbero stati cross-compilati. Da ciò si è dedotto che l'impiego di risorse di allocazione e di calcolo sarebbe stato eccessivo, specialmente in scenari in cui molti progetti avrebbero richiesto una cross-build per architetture diverse: in quest'ultimo caso si sarebbe infatti dovuta gestire l'allocazione di oggetti `project` identici o, ancora peggio, il riferimento a liste di progetti condivise;

- **Parallelismo per progetti con `poll_interval` diversi**: l'idea di eseguire in parallelo per una stessa architettura progetti diversi si scontrava con il requisito della configurabilità: progetti diversi sarebbero dovuti essere configurabili per avere `poll_interval` diversi e questo avrebbe complicato di gran lunga la gestione del parallelismo. Le seguenti due possibili soluzioni a questo scenario infatti avrebbero solo aumentato complessità e tagliato le prestazioni:

  - **Soluzione 1: rimozione del multi-threading**: questa alternativa avrebbe previsto un'esecuzione del demone associato all'architettura "salte-

rina", ossia temporizzata tramite più intervalli di sleep, uno per ogni progetto interno allo scope del demone stesso. In tal caso non sarebbe stato più necessario avviare threads diversi ma sarebbe stato sufficiente eseguire in un ciclo di build il processo di cross-compilazione più volte a distanza di differenze di Δ secondi e ripetuto con `project` di input diversi. Questa soluzione, oltre ad avere minimo grado di eleganza, avrebbe portato a ritardi e sfasamenti tra le operazioni di build dovuti al tempo richiesto per ogni cross-compilazione, sì approssimabile ma comunque dipendente da fattori non sempre prevedibili (latenza di rete e risorse computazionali di sistema);

- **Soluzione 2: sleep di attesa interno ad ogni thread**: questa opzione, sebbene più elegante e di facile implementazione della precedente, si sarebbe basata sul concetto di un main "greedy" che avrebbe lanciato tutti i thread ogni volta che il minor `poll_interval` tra i progetti di sua competenza scadeva. Dopo di che ogni thread con `poll_interval` maggiore sarebbe entarto in uno stato di sleep per una differenza di Δ secondi dal `poll_interval` minore. Un'alternativa di questo tipo non solo avrebbe comportato l'aggiunta di un attributo `delta` alla struttura dati di input di ogni thread, ma avrebbe costretto il main a trascurare il join dei figli lanciati, andando così incontro a rischi per niente accettabili nel contesto di sviluppo di un sistema di CI robusto e ottimizzato.

Grazie a queste valutazioni - aggiuntive a quelle che veranno compiute durante l'ultima fase di confronto "parametrico" tra le due alternative architetturali - si è deciso di spostare l'attenzione su un'opzione dalla logica invertita.

Seconda soluzione: worker per progetto e threads per architettura

Come fatto per la prima soluzione, prima di qualsiasi valutazione o descrizione, risulta utile illustrare quest'opzione tramite un diagramma concettuale.

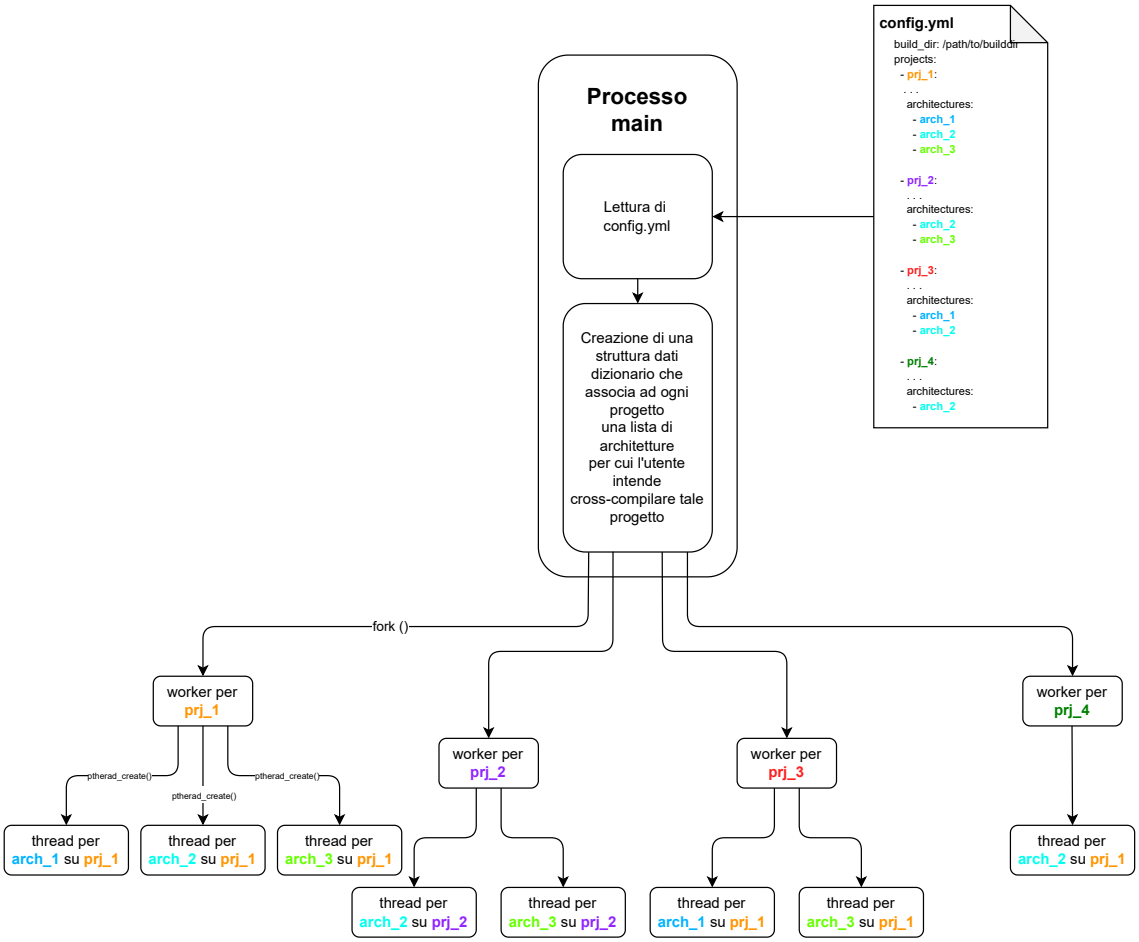

Figura 4.4: Schema concettuale della seconda alternativa architetturale di Rootless V^2CI: worker per progetto e threads per architettura

Questa configurazione architetturale di Rootless V^2CI inverte l'esecuzione dei componenti "padre" e "figlio" rispetto alla prima alternativa, affidando a ogni demone lanciato dal main il compito di cross-compilare un progetto per tutte le architetture specificate nel file di configurazione.
Tale scelta avrebbe permesso di risolvere le criticità emerse in precedenza, in particolare, per quanto riguarda la gestione degli input, non sarebbe stato più necessario comporre strutture dati complesse e ridondanti, bensì i thread avrebbero ricevuto in input direttamente un argomento contentente un riferimento al progetto padre - il quale sarebbe stato l'unico parametro di esecuzione anche per il demone - e all'architettura di destinazione. Anche per quanto riguarda la gestione degli intervalli di esecuzione, questa logica avrebbe semplificato drasticamente l'esecuzione del main loop per ogni worker essendo quest'ultimo responsabile della build di un solo progetto, associato a un solo `poll_interval`.

Nonostante tali vantaggi, ciò che era garantito dalla logica a "scompartimenti su disco" della soluzione precedente, in questa era negato: i threads fratelli non avrebbero operato su risorse di storage in comune, bensì avrebbero dovuto regolare la loro esecuzione con altri eventuali threads associati a `project` diversi ma che operavano sullo stesso rootfs. Ciò avrebbe chiaramente richiesto l'implementazione di un sistema di locking per la gestione della concorrenza sui file system chroot condivisi, escludendo la possibilità di gestire le interferenze tramite semplici mutex `pthread`.

Sebbene già da questa prima analisi fosse emerso che di entrambe le soluzioni architetturali proposte nessuna fosse perfetta e computazionalmente snella quanto la logica alla base di Rootless sshlirpCI, era comunque già possibile intuire che la seconda alternativa era la "favorita" in quanto, con un leggero aumento di complessità nella gestione del parallelismo, avrebbe rimosso criticità più profonde che avrebbero invece portato a cali evidenti di performance e imprecisioni inaccettabili. Nonostante la competizione tra queste due opzioni strutturali avesse quindi già una "vincitrice", un'ultima analisi comparativa, basata come anticipato sui tre parametri di re-use dei componenti di Rootless sshlirpCI, performance e coerenza logica con l'input, arricchiti con valutazioni sul setup degli ambienti rootfs e sulla scalabilità, ha permesso di confermare questa scelta.

Ultima analisi comparativa e parametrica

Come accennato poc'anzi, dalla precedente delineazione di alto livello delle due possibili soluzioni architetturali, oltre ai tre parametri comparativi già introdotti, sono emersi altri due fattori di valutazione che hanno permesso di completare il quadro analitico e di giungere a una decisione definitiva.
In particolare, si è deciso di considerare l'impatto che avrebbe avuto su ogni alternativa il setup degli ambienti chroot, in quanto operazione maggiormente costosa, e il grado di scalabilità orizzontale che ogni opzione avrebbe permesso.
Riassumendo quindi in cinque punti i criteri di valutazione adottati, è possibile svolgere le seguenti considerazioni:

- **Criticità del chroot setup**: come anticipato, l'operazione di cross-debootstrap è la più dispendiosa in termini di tempo e risorse di esecuzione, e per questo motivo si è reso necessario valutare, per ogni alternativa, in che momento dell'esecuzione sarebbe stato più opportuno svolgerla in modo da escludere sia ridondanze e sprechi dovuti ad eventuali setup multipli dello stesso rootfs che

ritardi nell'avvio dei worker o complicazioni nella gestione della concorrenza.
In tale contesto, il principale vantaggio della prima soluzione architetturale, ossia l'isolamento totale tra i rootfs, avrebbe intuitivamente permesso di eseguire tale operazione una sola volta, assegnandola al demone responsabile dell'architettura corrispondente. Questo sì avrebbe richiesto un sistema di locking aggiuntivo per il corretto assegnamento delle risorse computazionali, ma avrebbe evitato che processi diversi potessero tentare di eseguire la medesima operazione.
Se si fosse voluto poi escludere totalmente l'utilizzo di locking tra processi nella prima soluzione, si sarebbe potuti ricorrere sempre a setup iterativi nel main, ma questo avrebbe inevitabilmente svuotato di utilità funzionale i worker per architetture diverse.
D'altra parte, il punto critico della seconda soluzione avrebbe influenzato direttamente la sicurezza e la stabilità del processo di setup. Infatti il momento in cui sarebbe stato più logico eseguire il cross-debootstrap, volendo sfruttare il già esistente parallelismo del programma, sarebbe stato all'inizio dell'esecuzione di ogni thread foglia.
In sostanza, ogni thread per una certa coppia `<prj, arch>` avrebbe potuto inizialmente verificare la presenza del rootfs di sua competenza e, se necessario, svolgerne il setup. Considerando però che il rischio di setup concomitante di uno stesso rootfs da parte di threads figli di progetti diversi - e quindi non regolabili tramite mutex - sarebbe stato elevato, si sarebbe reso comunque necessario un sistema di locking tra threads.
Anche per questa seconda alternativa sarebbe stato possibile pensare di adottare un setup serializzato nel main. In questo caso non solo nessun componente avrebbe perso di funzionalità, in quanto ogni worker avrebbe ricoperto il solo ruolo di padre dei threads a prescindere, ma si sarebbe anche alleggerita l'esecuzione dei threads ed eliminati i problemi di conocorrenza.

Andando a confrontare quindi i due posizionamenti migliori del chroot setup per ogni alternativa, si è notato che, nonostante la seconda soluzione partisse svantaggiata a causa di un parallelismo di gestione intrinsecamente meno facile, l'opzione di una costruzione serializzata dei rootfs nel main per la seconda alternativa architetturale garantiva un'esecuzione dei demoni più leggera e un'amministrazione della concorrenza basata su locking tra threads. L'opzione

di setup concorrente per la prima alternativa avrebbe invece ritardato l'esecuzione dei worker e richiesto l'impiego di locks, che si sarebbe aggiunto alla già esistente necessità di mutex tra threads operanti nello stesso rootfs.

- **Scalabilità**: in questo contesto, con il termine "scalabilità" si vuole intendere il grado di flessibilità da parte del sistema di CI ad ammettere l'aggiunta di più progetti nel file di configurazione da parte dell'utente. Infatti, mentre il numero massimo di architetture per cui un utente può richiedere la cross-compilazione rimane limitata superiormente da 4 (`amd64`, `arm64`, `armhf` e `riscv64`), il numero di progetti che possono essere specificati è virtualmente illimitato.
  In questo senso, la prima alternativa architetturale avrebbe garantito minimo grado di scalabilità. La definizione di un progetto aggiuntivo da parte dell'utente avrebbe infatti implicato che:
  1. il main avrebbe allocato un nuovo oggetto `project`
  2. per ogni struttura dati `architecture` passata in input ai demoni, il main avrebbe aggiunto alla lista di progetti per quella architettura una copia allocata dell'oggetto `project` o, ancora peggio, un riferimento a quello già allocato, causando nel primo caso ridondanze e sprechi e nel secondo condivisione di puntatori tra processi diversi.

  D'altra parte, la seconda alternativa avrebbe garantito massimo grado di scalabilità. Infatti l'aggiunta di un progetto nel file di configurazione avrebbe richiesto al main la sola allocazione di un nuovo oggetto `project`, il quale sarebbe stato passato in input a un nuovo demone, senza dover appesantire strutture dati esistenti e gestire ridondanze o puntatori condivisi.

- **Re-use dei componenti di Rootless sshlirpCI**: come anticipato, uno dei requisiti per la progettazione di Rootless V^2CI era quello di minimizzare lo sforzo di re-implementazione di componenti già sviluppati, dando quindi priorità a un'implementazione modulare.
  Una struttura basata su workers per architettura e threads per progetto avrebbe totalmente stravolto la logica alla base di Rootless sshlirpCI, arricchendo il contesto di esecuzione di ogni demone con l'eventualità di sorgenti diversi e privandolo del concetto di cross-compilazione per architetture multiple.
  La seconda alternativa invece rispettava alla perfezione il requisito di impiego di processi "Rootless sshlirpCI-like", permettendo quindi di reimpiegare molte delle idee genitori dell'implementazione dell'antenato di Rootless V^2CI.
  Infatti, come è possibile dedurre dalle analisi svolte poc'anzi circa la gestione

della concorrenza e il setup dei rootfs, si sarebbe reso necessario solo aggiungere un servizio di locking per l'amministrazione di threads di progetti diversi per una stessa architettura e spostare la creazione degli ambienti chroot in un ciclo interno al main, eliminando la necessità di locking per questa fase.

- **Performance**: tutte le considerazioni svolte in precedenza hanno condotto a dedurre che le prestazioni di una qualsiasi configurazione della prima opzione architetturale sarebbero state inferiori rispetto a quelle della seconda.
  L'allocazione di strutture dati duplicate o condivise avrebbe di certo avuto un impatto negativo sul tempo di esecuzione e sulle risorse richieste dai demoni. Anche entrambe le possibili soluzioni al problema di concorrenza tra threads di progetti diversi, con `poll_interval` diversi, avrebbe portato all'abuso di cicli di sleep. Per di più, il mancato riciclaggio dei componenti di Rootless sshlirpCI avrebbe inevitabilmente privato il sistema di ottimizzazioni già collaudate. Infine l'impiego sia di locks per la fase di chroot setup concorrente che di mutex per la cross-compilazione in ambienti condivisi da threads fratelli, avrebbe appesantito ulteriormente l'esecuzione della prima opzione architetturale, rispetto alla sola necessità di servizi di locking tra threads "cugini" per la cross-compilazione in ambienti condivisi, necessaria nella seconda alternativa.

- **Coerenza logica con l'input**: come già accennato, un'architettura del primo tipo sarebbe stata "contro la natura" dell'input e, oltre a introdurre tutte le limitazioni di cui si è già discusso, avrebbe aumentato la curva di apprendimento per l'utente finale, oltre a rallentare la fase di sviluppo del sistema stesso.
  La seconda alternativa invece avrebbe aderito perfettamente alla forma dell'input fornito dall'utente e alle aspettative logiche di quest'ultimo, facilitando così il passaggio da configurazione a strutture dati nell'implementazione, l'esecuzione del sistema stesso e la sua comprensione.

Al termine di questa lunga analisi comparativa, è stata evidente la superiorità della seconda alternativa architetturale, che, sebbene presentasse lo svantaggio di non essere a "scompartimenti su disco", è stata comunque adottata come struttura portante di Rootless V^2CI.
Nonostante questa prima selezione abbia permesso di escludere molti scenari che si sarebbero dimostrati problematici, il lavoro di progettazione e implementazione del sistema era appena iniziato.

4.1.2 Dettagli architetturali e aspetti implementativi

La lunga fase implementativa di Rootless V^2CI ha visto in parallelo anche la traccia dei dettagli strutturali del sistema stesso. In particolare il primo sviluppo è stato guidato da quattro fasi principali, successive al primo disegno architetturale:

1. Progettazione e implementazione del nuovo main radice, in modo che potesse svolgere i compiti assegnatigli dalla precedente delineazione generale;
2. Ottimizzazione di alcuni aspetti dei demoni "Rootless sshlirpCI-like" in modo che sgravassero i thread figli da check ridondanti e da operazioni non di loro competenza;
3. Adattamento e scomposizione dei threads builder al fine di un'esecuzione per progetto meno monolitica e ridondante e allo stesso tempo più performante e facilmente interrompibile, mantenendo sicurezza, coerenza e idempotenza;
4. Raffinamento del sistema di stop, mirato all'interruzione sia di un eventuale processo main preparatorio che di tutti i demoni in esecuzione.

Il nuovo main: interfaccia tra utente e demoni

Il processo main di Rootless V^2CI avrebbe svolto pochi e semplici compiti, assumendo solo alcune delle responsabilità del main dell'antenato Rootless sshlirpCI e garantendo una vita più leggera ai demoni figli.
In particolare questo nuovo processo radice avrebbe avuto la seguente forma:

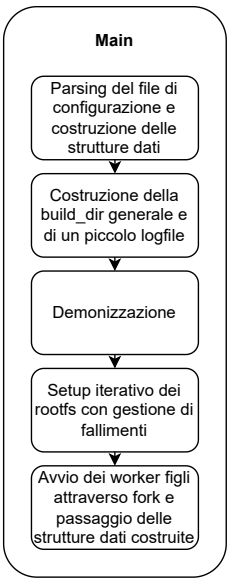

Figura 4.5: Schema concettuale del main di Rootless V^2CI

I due componenti più complessi di questo processo dall'esecuzione limitata sarebbero stati il parser del file di configurazione e il sistema di setup dei rootfs.

Per quanto riguarda il **parser**, la difficoltà principale è stata quella di trovare un metodo di tokenizzazione che fosse flessibile per la lettura di un file dalla struttura "object oriented" e innestata.
Un input di tipo `.yml` infatti permetteva sì all'utente di impostare il comportamento di Rootless V^2CI in modo granulare e ordinato, configurandosi come miglior opzione di interfaccia tra il motore di cross-compilazione e l'utilizzatore finale, ma comportava anche la necessità di un parser completo e robusto in grado di leggere correttamente ogni sezione, elenco, chiave e valore del file e tradurre il tutto in strutture dati C adeguate.

Il primo passo naturale, importante e preliminare alla composizione di questo strumento essenziale è stato appunto quello di plasmare tali strutture dati sulla forma dello stesso file di configurazione. Analizzando il contenuto ipotetico di quest'ultimo, riportato precedentemente, è stata evidente la necessità di impiego di tre strutture dati principali: `Config`, `project` e `manual_dependency`.

```
typedef struct manual_dependency {
  char git_url[MAX_CONFIG_ATTR_LEN];
  char build_system[MIN_CONFIG_ATTR_LEN];
  char *dependencies[MAX_DEPENDENCIES];
  int dep_count;
  struct manual_dependency *next;
} manual_dependency_t;

typedef struct project {
  char name[64];
  // <cfg.build_dir>/<project.name>
  char main_project_build_dir[CONFIG_ATTR_LEN];
  // <main_project_build_dir>/logs/worker.log
  char worker_log_file[MAX_CONFIG_ATTR_LEN];
  // Path assoluto ricavato dal file di configurazione
  char target_dir[CONFIG_ATTR_LEN];
  char repo_url[MAX_CONFIG_ATTR_LEN];
  char main_repo_build_system[CONFIG_ATTR_LEN];
```

```
  char build_mode[MIN_CONFIG_ATTR_LEN];
  int  poll_interval;
  char *architectures[MAX_ARCHITECTURES];
  int arch_count;
  char *dependency_packages[MAX_DEPENDENCIES];
  int dep_count;
  manual_dependency_t *manual_dependencies;
  int manual_dep_count;
  struct project *next;
} project_t;

typedef struct {
  char build_dir[MIN_CONFIG_ATTR_LEN];
  char main_log_file[CONFIG_ATTR_LEN];
  project_t *projects;
  int project_count;
} Config;
```

A partire da queste strutture dati e dalla forma del file di configurazione è stato poi necessario ricercare un metodo di tokenizzazione dell'input che fosse in grado di popolare correttamnte i dati di esecuzione sopra illustrati.
Per fare ciò è stato deciso di adottare la libreria `libyaml` [61], la quale permette di gestire la lettura di complessi file `.yml` attraverso l'emissione e la cattura di eventi. Senza entrare nei dettagli di questo tool di parsing, è possibile mostrarne l'utilità, e l'impiego nello sviluppo del programma di riconoscimento dei pattern interni al `config.yml`, riportando alcuni estratti principali della funzione `load_config()` che si sarebbe appunto occupata di posizionare in una struttura dati `Config` il path della `build_dir` principale e una lista di `project`, caricati tramite la funzione ausiliaria `load_project`.

```
// Funzione helper per il parsing di un singolo oggetto 'project'
static int load_project(project_t *prj, yaml_parser_t *parser) {
  // Variabili di stato per tracciare sezioni (es. source, git) e chiavi
  int depth = 1; // Profondita' di annidamento corrente
  yaml_event_t ev;
  // Ciclo annidato: consuma eventi finche' non si chiude il project
  while (depth > 0) {
    // La funzione yaml_parser_parse fa avanzare il parser e popola 'ev'
```

```
    if (!yaml_parser_parse(parser, &ev)) { ... }
    switch (ev.type) {
      case YAML_SCALAR_EVENT:
        // L'evento SCALAR indica la lettura di una chiave o di un valore
        char *val = (char *)ev.data.scalar.value;
        // Se siamo dentro una sequenza (es. architectures),
        // aggiungiamo il valore alla lista
        if (seq == SEQ_ARCH) {
          add_string(prj->architectures, ..., val);
        }
        // Altrimenti controlliamo l'ultima chiave letta per assegnare
        // il valore al campo corretto
        else {
          if (strcmp(last_key, "name") == 0) {
            snprintf(prj->name, sizeof(prj->name), "%s", val);
          } else if (strcmp(last_key, "repo_url") == 0) {
            snprintf(prj->repo_url, sizeof(prj->repo_url), "%s", val);
          }
          // ... gestione altri campi (build_mode, ecc.) ...
        }
        break;
      case YAML_MAPPING_START_EVENT:
        depth++; // Entrata in una sotto-sezione (es. 'source:')
        // ... logica per aggiornare lo stato della sezione corrente ...
        break;
      case YAML_MAPPING_END_EVENT:
        depth--; // Uscita da una sotto-sezione
        break;
      case YAML_SEQUENCE_START_EVENT:
        // Inizio di una lista (es. 'architectures:', 'dependencies:')
        // ... aggiornamento stato sequenza attiva ...
        break;
      // ...
    }
    yaml_event_delete(&ev);
  }
  return 0;
}
```

```
// Funzione principale di caricamento della configurazione
int load_config(Config *cfg) {
  // ... apertura file e inizializzazione strutture ...
  // Inizializzazione del parser libyaml
  yaml_parser_t parser;
  yaml_parser_initialize(&parser);
  yaml_parser_set_input_file(&parser, config_file);
  // Ciclo principale di parsing del file
  while (!done) {
    yaml_parser_parse(&parser, &ev); // Fetch del prossimo evento
    switch (ev.type) {
      case YAML_SCALAR_EVENT:
        // Lettura chiavi globali (es. 'build_dir')
        if (!in_projects && strcmp(top_last_key, "build_dir") == 0) {
          snprintf(cfg->build_dir, ..., val);
        }
        break;
      case YAML_SEQUENCE_START_EVENT:
        // Rilevamento inizio lista 'projects'
        if (strcmp(top_last_key, "projects") == 0) {
          in_projects = 1;
        }
        break;
      case YAML_MAPPING_START_EVENT:
        // Se siamo dentro la lista 'projects', inizia un nuovo oggetto
        if (in_projects) {
          project_t *prj = calloc(1, sizeof(project_t));
          // ... linking del nuovo progetto alla lista in cfg ...
          // Delega il parsing del contenuto del progetto alla funzione,
          // helper passando il parser per continuare la lettura
          load_project(prj, &parser);
        }
        break;
      case YAML_STREAM_END_EVENT:
        done = 1; // Fine del file
        break;
    }
    yaml_event_delete(&ev);
```

```
    }
    // ... cleanup parser e chiusura file ...
    return 0;
}
```

Grazie allo sviluppo di questo parser potenziato e flessibile, il main era ora dotato della capacità di estrarre correttamente, in una variabile `Config cfg`, i dati di esecuzione forniti dall'utente, creare la `cfg.build_dir`, il suo log file `cfg.main_log_file`, demonizzarsi esattamente come veniva fatto in Rootless sshlirpCI - settando quindi anche un handler temporaneo per catturare eventuali `SIGTERM` durante successive operazioni consistenti - e procedere con il setup dei rootfs.

La seconda operazione più densa, dopo il parsing del file `config.yml` e il load delle variabili di configurazione, era appunto il **setup degli ambienti chroot**, che, a seguito delle valutazioni svolte nella sottosezione 4.1.1, si era deciso di eseguire in modo serializzato nel main, attraverso l'invocazione diretta di uno script `chroot_setup.sh`.
Chiaramente, per realizzare tale operazione, è stato necessario inserire, tra la fase di demonizzazione e quella di costruzione dei rootfs, l'estrazione delle architetture, richieste da tutti i progetti della lista `cfg.projects` e non ripetute. Quest'operazione addizionale, rispetto a quelle richieste da un alternativo setup sincrono svolto dai singoli threads, ha aggiunto un minimo overhead in termini di tempo di computazione, ma ha anche alleggerito drasticamente l'esecuzione dei demoni figli.
La scelta di un setup iterativo nel main ha però fatto risorgere i problemi di coerenza legati alla costruzione di ambienti chroot una tantum, che rendevano sshlirpCI soggetto a errori in scenari di rimozione a run-time delle risorse di build.
Tale criticità è stata risolta solo in un secondo momento, grazie a raffinamenti e accorgimenti implementativi che verranno discussi nella sezione 4.1.4.
Anche in questo primo sviluppo però si è tentato di migliorare la sicurezza del sistema e garantire che non ci fossero contraddizioni tra l'esito delle operazioni del main e l'esecuzione dei demoni figli. Infatti, durante il chroot setup iterativo, si è deciso di tener traccia delle architetture per cui il setup fosse fallito, provvedendo in seguito a rimuoverle dalla lista di architetture di ogni progetto. In questo modo si è quanto meno garantito che nessun demone figlio avrebbe tentato di operare in un ambiente chroot non correttamente costruito.

Le successiva e conclusiva operazione di avvio iterativo dei worker per progetto, sempre interna al main, avrebbe portato a termine il compito di quest'ultimo, passando il controllo ai processi forkati, attraverso riferimenti ai corrispondenti `project`. Questa prima versione del main di Rootless V^2CI, integrando in essa i ruoli di "interfaccia utente", loader, setupper e orchestratore, ha quindi assunto la seguente forma:

```
int main() {
  // 0. Caricamento delle variabili dal file di configurazione
  Config cfg;
  if(load_config(&cfg) != 0) { ... }
  // 1. Creazione di directory e file principali (se non esistono)
  // ...
  // 2. Demonizzazione del processo
  daemonize();
  // 3. Unione delle architetture necessarie da tutti i progetti
  // in una singola lista di architetture uniche
  char *archs_list[MAX_ARCHITECTURES];
  int num_archs = 0;
  project_t *current = cfg.projects;
  for (int i = 0; i < cfg.project_count; i++) {
      // ... logica di unione architetture ...
  }
  // 4. Setup iterativo del chroot per ogni architettura
  char *failed_chroots[MAX_ARCHITECTURES];
  int num_failed_chroots = 0;
  for (int i = 0; i < num_archs; i++) {
    // I setup del chroot sono le operazioni piu' dispendiose,
    // quindi se  viene ricevuto un segnale di terminazione,
    // esce immediatamente
    if (terminate_main_flag) { ... }
    // ... preparazione path chroot ...
    if (chroot_setup(archs_list[i], chroot_dir, cfg.main_log_file,
      log_fp) != 0
    ) {
      // ... gestione errore setup ...
      failed_chroots[num_failed_chroots] = archs_list[i];
      num_failed_chroots++;
```

```
    }
  }
  // ... controllo se tutti i setup sono falliti ...
  // 5. Avvio iterativo dei builder di progetto tramite fork
  // (nuovo processo per ogni progetto)
  current = cfg.projects;
  for (int i = 0; i < cfg.project_count; i++) {
    // Prima di lanciare un nuovo worker, controlla se e' stato
    // ricevuto un segnale di terminazione
    if (terminate_main_flag) { ... }
    pid_t pid = fork();
    if (pid < 0) {
      // ... errore fork ...
    } else if (pid == 0) {
      // Processo figlio: lancia il worker del progetto solo con
      // le architetture per cui il setup del chroot ha
      // avuto successo
      int failed_removal = remove_failed_archs_from_project(current,
        failed_chroots, num_failed_chroots
      );
      // ...
      int result = project_worker(current, cfg.build_dir);
      exit(result);
    } else {
      // Processo padre: continua al prossimo progetto
      current = current->next;
    }
  }
  // ... chiusura log e rimozione pid file ...
  return 0;
}
```

I demoni per progetto: processi Rootless sshlirpCI ottimizzati

Meno rivoluzionarie sono state le modifiche apportate al corpo del main loop di Rootless sshlirpCI, quasi integralmente reimpiegato nei demoni figli di Rootless V^2CI.
Infatti, sebbene la fase di demonizzazione e di setup degli ambienti chroot risiedesse

ora all'interno del nuovo main generale, la forma dei demoni di Rootless V^2CI ha permesso di rispettare ampiamente il requisito di re-use dei componenti antenati. Come nella struttura portante di sshlirpCI e Rootless sshlirpCI, anche alla base dei worker di questo nuovo motore risiedevano:

- un setup iniziale del sotto-albero di filesystem dedicato al progetto specifico di ogni worker;
- un loop infinito che eseguisse periodicamente l'avvio dei threads builder per ogni architettura richiesta dal progetto.

Tre sole sono state le ottimizzazioni e le modifiche necessarie apportate a questa struttura di base, precedentemente collaudata, e riassumibili nel seguente schema comparativo che mostra a confronto i main di sshlirpCI e Rootless sshlirpCI con il componente "demone per progetto", impiegato invece in Rootless V^2CI:

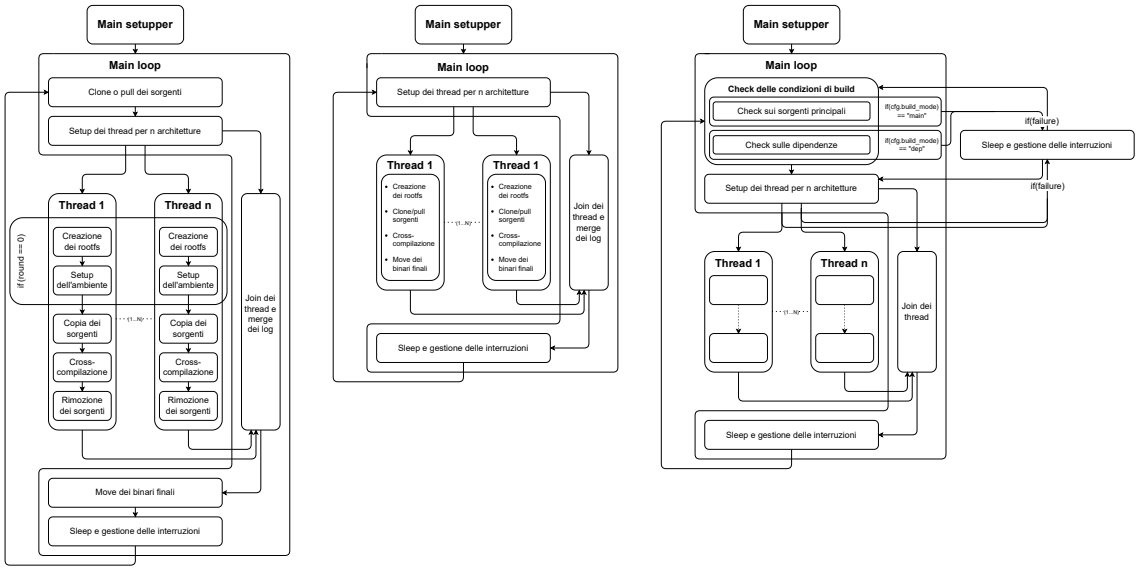

Figura 4.6: Confronto tra rispettivamente il main di sshlirpCI, quello di Rootless sshlirpCI e il demone per progetto di Rootless V^2CI

In particolare, come suggerito dal diagramma soprastante, è possibile ampliare la descrizione delle novità e delle variazioni introdotte in tre punti principali:

- **Spostamento del check delle condizioni di build dallo script di esecuzione dei thread al main**: per quanto questa operazione di trasferimento dei controlli di presenza o aggiornamento dei sorgenti possa rievocare la struttura scomposta e frammentata di sshlirpCI, è bene sottolineare che in questo nuovo contesto non sarebbe più convenuto eseguire in un unico script tutte le operazioni di competenza di un singolo thread. Il vantaggio di un'esecuzione

monolitica in Rootless sshlirpCI derivava infatti dalla possibilità di delegare ai singoli thread anche gli stessi setup dei rootfs. In Rootless V^2CI invece l'esecuzione dei thread è separata dalla costruzione degli ambienti di build ed è quindi più logico cercare di estrarre tutte le operazioni "generalizzabili" appena successive.
Infatti un semplice check esterno alla fase di cross-compilazione vera e propria, non comprensivo di alcun clone effettivo ed eseguito sul primo rootfs setup-ato dal main, non solo non trasgredisce la linea guida *Quae sunt Thread-is, Thread-i*, ma permette anche di regolare l'avvio del processo di cross-compilazione in base al nuovo parametro di modalità di build. Avendo infatti il demone accesso diretto alla variabile `prj->build_mode`, risulta conveniente che sia lui stesso a invocare un metodo `check_for_updates_inside_chroot()` con gli argomenti corretti, i quali comunque verranno poi passati a un componente `check_updates.sh` eseguito all'interno del chroot.

- **Gestione degli errori e delle interruzioni**: il main process sia di sshlirpCI che di Rootless sshlirpCI, in caso di fallimento di una qualsiasi sua operazione, ritornava errore e terminava la sua esecuzione. In Rootless V^2CI invece si è deciso di dotare i demoni della capacità di entrare anticipatamente nel ciclo di sleep tramite una funzione `sleep_and_handle_interrupts()` e, al suo termine, ritentare direttamente l'ultima operazione fallita. Questo approccio è stato adottato solo nel contesto di operazioni maggiormente *network-dependent* come il check delle condizioni di build.
  Contestualmente a questo potenziamento, che avrebbe permesso al demone di un certo progetto di proseguire la sua esecuzione in caso di errori saltuari, si è deciso inoltre di irrobustire il sistema di cattura dei segnali `SIGTERM`. La predisposizione alla ricezione e all'elaborazione di tali segnali è stata infatti aggiunta in corrispondenza di ogni attesa, come è possibile intuire dal nome della funzione poc'anzi citata. Questa ottimizzazione è conseguenza logica della presenza più diffusa delle fasi di sleep: di per sé un ciclo di esecuzione del demone con esito positivo non impiega troppo tempo se non appunto per l'attesa del join dei tread figli - la cui predisposizione all'interruzione tramite `SIGTERM` verrà discussa nella sottosezione successiva -, per un eventuale loop infinito generatosi da un bug persistente e per il ciclo di sleep conclusivo di ogni iterazione. Queste ultime due sorgenti di attesa occupano probabilmente più del 50% della vita del demone e quindi sono intrinsecamente più inclini a

ricevere segnali di interruzione da parte di un utente.
Quindi, quest'ultimo, grazie a tale accorgimento avrebbe potuto servirsi più frequentemente del programma di stop piuttosto che di un eventuale killer, garantendo al demone l'opportunità di una terminazione sì pulita ma anche coerente.

- **Rimozione del concatenazione dei logs**: parallelamente alla riscrittura del main loop interno ad ogni demone, si è pensato che l'operazione di concatenazione dei logs prodotti dai threads figli in un unico log file "personale" del demone padre, avrebbe aggiunto vantaggi minimi per l'utente rispetto al più consistente overhead introdotto dalle frequenti operazioni di I/O, specialmente nel contesto di un'esecuzione generale con due gradi di concorrenza e, conseguentemente, caratterizzata da un elevato numero di processi in esecuzione contemporaneamente.
  Per questo motivo si è deciso di abbandonare completamente la gestione e il popolamento di un unico log file di riferimento per un certo progetto, mantenendo invece su disco quello personale di ciascun thread lanciato.

La ripresa quasi in toto dell'antenato `main.c` di Rootless sshlirpCI con l'aggiunta di queste modifiche marginali, ha portato alla creazione di un file `project_worker.c` dalla seguente forma:

```
// ... include, gestori di segnali e helper per file di stato ...
static void sleep_and_handle_interrupts(int poll_interval,
  char *STATE_FILE, FILE *log_fp, const char *project_name
) {
  update_worker_state(PROJECT_WORKER_STATE_SLEEPING, STATE_FILE);
  unsigned int time_left = poll_interval;
  while(time_left > 0) {
    time_left = sleep(time_left);
    // Controllo segnale di terminazione durante lo sleep
    if (terminate_worker_flag) {
      break;
    }
  }
  update_worker_state(PROJECT_WORKER_STATE_WORKING, STATE_FILE);
}
int project_worker(project_t *prj, char *main_build_dir) {
```

```
// ... Setup logging, PID file, gestori segnali, directory ...
// Loop principale
while (1) {
  if (terminate_worker_flag) break;
  // 1. Controllo aggiornamenti direttamente nel demone
  int need2update = 0;
  // ... setup percorsi chroot ...
  if (strcmp(prj->build_mode, "main") == 0) {
    // ... estrazione nome repo ...
    int update_result = check_for_updates_inside_chroot(...,
      &need2update, ...
    );
    if (update_result != 0) {
        // 2. Gestione errori tramite sleep e riprova
        sleep_and_handle_interrupts(prj->poll_interval, ...);
        continue;
    }
  } else if (strcmp(prj->build_mode, "dep") == 0) {
    // ... logica simile per le dipendenze ...
  }
  if (!need2update) {
    sleep_and_handle_interrupts(prj->poll_interval, ...);
    continue;
  }
  // 3. Avvio thread (nessuna concatenazione log, logging diretto)
  pthread_t threads[prj->arch_count];
  thread_arg_t args[prj->arch_count];
  for (int i = 0; i < prj->arch_count; i++) {
    // ... setup argomenti ...
    if (pthread_create(&threads[i], NULL, build_thread,
      &args[i]) != 0
    ) {
      // Gestione errore creazione
      sleep_and_handle_interrupts(prj->poll_interval, ...);
      continue;
    }
  }
  // Attesa dei thread
```

```
    for (int j = 0; j < i; j++) {
      pthread_join(threads[j], ...);
      // ... log risultato thread ...
    }
    // Sleep prima della prossima iterazione
    sleep_and_handle_interrupts(prj->poll_interval, ...);
  }
  // ... Pulizia ...
  return 0;
}
```

Adattamento e scomposizione dei thread builder

I riflessi più importanti dell'architettura di Rootless V^2CI, scalata orizzontalmente rispetto a quella dell'antenato Rootless sshlirpCI, si sono maggiormente notati nell'adattamento dei thread builder al nuovo contesto "variabile".
Sia in sshlirpCI che in Rootless sshlirpCI infatti, il contesto di esecuzione dei thread per architettura era tracciato: essendo il sorgente da cross-compilare fissato su `sshlirp`, dipendenze, tool di compilazione e pacchetti richiesti erano noti a priori e l'esecuzione del processo di build in base a questi era quindi costante.
Una sequenza di operazioni fissa e di competenza dei soli threads aveva senso di essere racchiusa in un unico script monolitico, come era stato fatto in Rootless sshlirpCI; ma in Rootless V^2CI, con la moltiplicazione dei progetti, la variabilità delle loro caratteristiche e l'aggiunta di sorgenti multipli per ognuno di essi, questa rigidità avrebbe costretto a una manipolazione degli argomenti di input per l'ipotetico script centrale per niente immediata, che avrebbe previsto:

1. una traduzione integrale da parte del demone, o di ogni suo thread, della complessa struttura dati `project` in una serie di argomenti stringa;
2. l'invocazione dello script di build con la complessa lista di parametri;
3. la decodifica, da parte dello script, di questi argomenti in variabili locali e strutture dati interne, per poter eseguire correttamente le operazioni di cross-compilazione.

Queste operazioni sarebbero state evidentemente molto costose in termini di tempo di sviluppo e di performance, oltre a essere soggette a errori di traduzione e interpretazione dei dati.
Per questo motivo si è deciso di scomporre il monolite di Rootless sshlirpCI in una

serie di funzioni C, ognuna delle quali si sarebbe occupata di un singolo compito all'interno del processo di build, tornando alla struttura più modulare e flessibile di sshlirpCI, senza però aumentare l'overhead dovuto alle operazioni di I/O su file di script - che come in Rootless sshlirpCI sarebbero stati comunque eseguiti direttamente - né abbattere le performance a causa dell'abuso di `system()` - che sarebbe stato sostituito da `system_safe()` come quanto fatto per Rootless sshlirpCI.
Perciò, per ogni operazione che avrebbe avuto un alto grado di variabilità negli argomenti di input, in quanto non solo sarebbe stata invocata su progetti diversi ma anche su sorgenti di input multipli internamente allo scope di uno stesso `project`, si è deciso di creare uno script `.sh` dedicato e invocarlo come utility pittosto che motore comprensivo di tutte le funzionalità.
In particolare, ogni thread builder di Rootless V^2CI avrebbe eseguito in ordine:

- `install_packages_in_chroot.sh`, invocato sia per l'installazione delle dipendenze di sistema richieste dal progetto principale che per l'installazione di quelle delle sue dipendenze "manuali";
- `clone_or_pull_for_project.sh`, lanciato per ogni sorgente da cross-compilare, sia esso il repository principale o uno di quelli delle dipendenze manuali;
- `cross_compiler.sh` che, come nei precedenti casi, sarebbe stato eseguito per ogni sorgente, occupandosi della vera e propria compilazione incrociata per il repository di input, attraverso il metodo di build specificato dal parametro `build_system`. Per questo componente è bene specificare che è stato necessario inserire anche una fase di selezione euristica del toolchain da utilizzare per la compilazione e un processo di scelta dei binari statici finali, se prodotti dalla build. Inoltre questo script, nel caso in cui il sorgente fosse stato una dipendenza, avrebbe proceduto a una sua installazione all'interno del rootfs, In caso contrario, avrebbe spostato l'eseguibile selezionato in una directory di staging interna al chroot e poi nella destinazione target versionata, sull'host.

In base a quanto appena descritto, è possibile rappresentare la forma generale di un thread builder di Rootless V^2CI con il seguente diagramma:

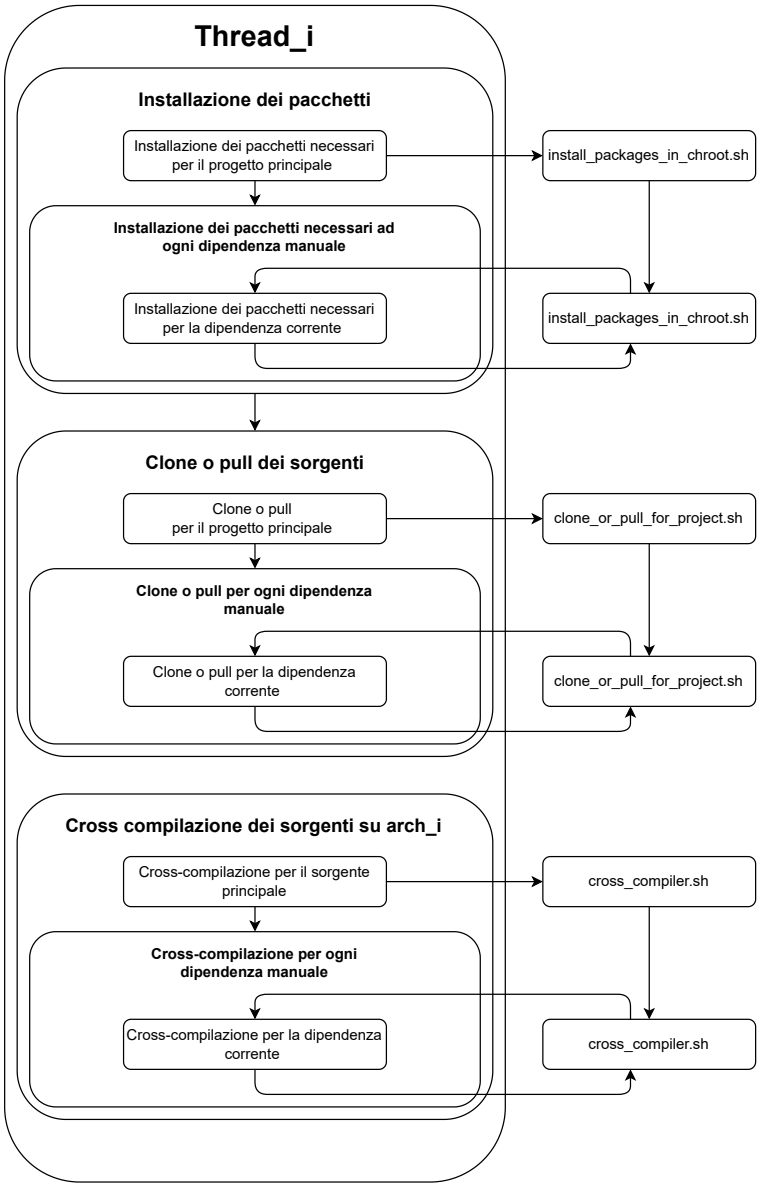

Figura 4.7: Schema concettuale di un thread builder di Rootless V^2CI

Come è possibile evincere dallo schema soprastante, ciò che ha permesso l'interazione tra questi script di "utility" e la necessità dei thread builder di portare a termine una cross-compilazione completa, sono state funzioni intermedie, dedite a estrarre i dati necessari dalla struttura `project` e a eseguire iterativamente su ogni "input atomico" i componenti `.sh` sopracitati, che non avrebbero quindi dovuto occuparsi di alcun parsing o interpretazione di argomenti complessi.

Questa scelta, quasi obbligata, di scomporre il monolite di Rootless sshlirpCI ha portato con sé due vantaggi addizionali, figli anche di arricchimenti e ottimizzazioni

implementative:

- **Maggior interrompibilità dei thread builder tramite stop**: sebbene non sia stato precedentemente specificato, in questo nuovo contesto anche la struttura dati di input di ogni thread `thread_arg_t` è stata modificata non solo affinché contenesse un riferimento necessario al suo `project` padre, ma anche per includere la flag di terminazione condivisa con il demone genitore. Infatti, come accennato nella sottosezione precedente, una delle operazioni più time-consuming del worker per progetto sarebbe stata l'attesa della terminazione dei suoi thread figli. L'assenza di un metodo di interruzione ordinato e granulare per quest'ultimi avrebbe costretto l'utente a servirsi direttamente di un eventuale killer, senza garanzia di terminazione coerente. Invece, con la condivisione della `terminate_worker_flag`, definita nel file `project_worker.c`, e grazie alla nuova struttura modularmente scomposta dei threads builder, questi avrebbero potuto controllare periodicamente la presenza di un segnale di interruzione per il genitore, interrompendo la loro esecuzione prima di una qualsiasi operazione consistente o *network-dependant*. La precedente esecuzione monolitica adottata in Rootless sshlirpCI invece non avrebbe permesso di usufruire di questo meccanismo, costringendo l'utente ad attese più lunge o terminazioni forzate.

- **Tracciabilità degli errori e dello stato di avanzamento**: un secondo vantaggio derivante dalla scomposizione dell'esecuzione dei threads per architettura è stata la possibilità di amministrare gli scenari di terminazione in modo più completo, ritornando al project worker non solo un exit code - che nel caso di Rootless sshlirpCI assumeva i soli valori di 0 o 1 rispettivamente in caso di successo o fallimento - ma anche un messaggio e un nuovo attributo `char *stats`, in modo che la scomodità derivante dalla mancanza di un log file centralizzato per ogni worker fosse compensata da una tracciabilità di eventuali errori potenziata. Ovviamente per permettere ciò è stato ripreso e arricchito da sshlirpCI l'impiego della struttura dati `thread_result`.

L'ultima modifica sostanziale al corpo dei threads builder di Rootless V^2CI è stata l'aggiunta del sistema di locking per l'amministrazione di operazioni sui rootfs condivisi con threads "cugini", come introdotto nella sottosezione 4.1.1, dedicata alla descrizione della seconda alternativa architetturale.
Infatti, sebbene threads per la stessa architettura e figli di `project` differenti non avrebbero mai condiviso directory o file all'interno del rootfs comune, l'operazione

di installazione delle dipendenze di sistema tramite `apt`, se eseguita contemporaneamente da più processi, avrebbe portato quest'ultimi a tentare di scrivere simultaneamente su `/usr/local/var/lib/dpkg/status`. `dpkg`, per escludere incoerenze e corruzioni del database dei pacchetti, impone un lock esclusivo su questo file durante le operazioni di installazione [62], e un qualsiasi processo che tenti di acquisirlo mentre è già detenuto da un altro fallisce immediatamente.
Quindi, per evitare terminazioni precoci da parte dei threads builder a causa di questa eventualità, in Rootless V^2CI si è deciso di atomizzare l'operazione di installazione dei pacchetti tramite due funzioni `lock_package_manager_in_chroot()` e `unlock_package_manager_in_chroot()`, che si sarebbero entrambe servite di `flock()`.

La nuova carrozzeria dei thread builder di Rootless V^2CI, che ha preso il posto del monolite di Rootless sshlirpCI, ha quindi assunto la seguente forma finale:

```
// ... include e funzioni helper per il locking ...
void *build_thread(void *arg) {
  // Estrazione argomenti e preparazione risultato
  thread_arg_t *targ = (thread_arg_t *)arg;
  project_t *prj = targ->project;
  volatile sig_atomic_t *terminate_flag = targ->terminate_flag;
  thread_result_t *result = malloc(sizeof(thread_result_t));
  // ... inizializzazione result e logging ...
  // Creazione directory e file necessari nel chroot
  // ... (mkdir ricorsivi per build_dir, log_file, target_dir) ...
  // 1. Installazione dipendenze con locking
  if (*terminate_flag) { ... return (void *)result; }
  // Acquisizione lock per evitare conflitti apt/dpkg tra processi
  int lock_fd = lock_package_manager_in_chroot(
    targ->thread_chroot_dir, ...
  );
  if (lock_fd == -1) { ... return (void *)result; }
  // Installazione dipendenze principali
  if (install_packages_list_in_chroot(prj->dependency_packages, ...)
    != 0
  ) {
    // ... gestione errore ...
```

```
    return (void *)result;
  }
  // Installazione dipendenze manuali (iterativa)
  manual_dependency_t *cur_manual = prj->manual_dependencies;
  while (cur_manual) {
    if (install_packages_list_in_chroot(
        cur_manual->dependencies, ...
      ) != 0
    ) {
      // ... gestione errore ...
      return (void *)result;
    }
    cur_manual = cur_manual->next;
  }
  // Rilascio lock
  unlock_package_manager_in_chroot(lock_fd, ...);
  // 2. Clone o pull dei sorgenti (principale e dipendenze)
  if (*terminate_flag) { ... return (void *)result; }
  if (clone_or_pull_sources_inside_chroot(targ, log_fp) != 0) {
    // ... gestione errore ...
    return (void *)result;
  }
  // 3. Avvio processo di build (compilazione dipendenze e progetto)
  if (*terminate_flag) { ... return (void *)result; }
  if (build_in_chroot(targ, log_fp) != 0) {
    // ... gestione errore ...
    return (void *)result;
  }
  // Successo
  result->stats = "Progress: 100%";
  result->status = 0;
  return (void *)result;
}
```

Potenziamento del sistema di stop

Ultima modifica sostanziale all'ecosistema di Rootless sshlirpCI è stata la rimozione del metodo di terminazione tramite killer e l'implementazione di un sis-

tema di stop ordinato e granulare, che avrebbe permesso all'utente di interrompere l'esecuzione del motore in modo pulito e coerente.
Infatti, come deducibile dalle tre sottosezioni precedenti, la possibilità di interrompre in modo granulare il breve processo di "decollo" - ossia il nuovo main genitore dei worker -, i processi demoni e gli stessi threads "foglia", grazie all'impiego più consistente di flags di terminazione e a check più frequenti per le operazioni maggiormente time-consuming, ha permesso lo sviluppo di un programma `stop.c` che, per qualsiasi stadio di esecuzione, fosse in grado di terminare tutti i processi di cross-compilazione avviati minimizzando il tempo di attesa richiesto e massimizzando la pulizia e coerenza delle risorse lasciate "inattive" sul disco.
In particolare il nuovo programma di stop si occupa di:

1. inviare un segnale `SIGTERM` all'eventuale processo main ancora vivo;
2. caricare le variabili di configurazione dal file di input `config.yml`;
3. ricostruire il percorso dei pid file di ogni demone per `project` e inviare a ognuno di questi un segnale di terminazione senza attenderne la loro reale morte con pesanti verifiche iterative.

Quest'ultimo punto conferma che la predisposizione di ogni processo avviato a una ricezione granulare di interrupt, consente all'utente di avere un riscontro dell'esito del processo di stop in tempo reale e la garanzia di terminazione ordinata e rapida di Rootless V^2CI.

```
int main() {
  // 1. Tentativo di terminazione del processo main (se esiste)
  const char *MAIN_PID_FILE = "/tmp/rootless_v2ci.pid";
  if (access(MAIN_PID_FILE, F_OK) == 0) {
    // ... lettura pid da file ...
    kill(main_pid, SIGTERM);
  }
  // 2. Caricamento configurazione per ricostruire i path dei PID file
  Config cfg;
  if(load_config(&cfg) != 0) return 1;
  // 3. Iterazione sui progetti e invio SIGTERM ai worker
  project_t *current = cfg.projects;
  for (int i = 0; i < cfg.project_count; i++) {
    char PID_FILE[256];
    snprintf(PID_FILE, sizeof(PID_FILE), "/tmp/%s-worker.pid",
      current->name
```

```
    );
    if (access(PID_FILE, F_OK) == 0) {
      // ... lettura pid del worker ...
      kill(pid, SIGTERM);
      printf("Sent termination signal to project %s.\n",
        current->name
      );
    }
    current = current->next;
  }
  // ... cleanup memoria ...
  return 0;
}
```

4.1.3 Diagramma finale

Le scelte architetturali e implementative prese durante l'intero processo di sviluppo di Rootless V^2CI, hanno dato vita a un'infrastruttura di CI completa, altamente scalabile, configurabile e sicura, che combina all'interno dei suoi componenti distribuiti l'assenza di ridondanze e la granularità caratterizzanti sshlirpCI con l'idempotenza e la performance del suo successore Rootless sshlirpCI, introducendo anche nuovi meccanismi di gestione degli errori e delle interruzioni.
I seguenti diagrammi concettuali riassumono la forma finale di questo ecosistema, illustrandone rispettivamente la struttura generale e la composizione interna di un suo componente worker.

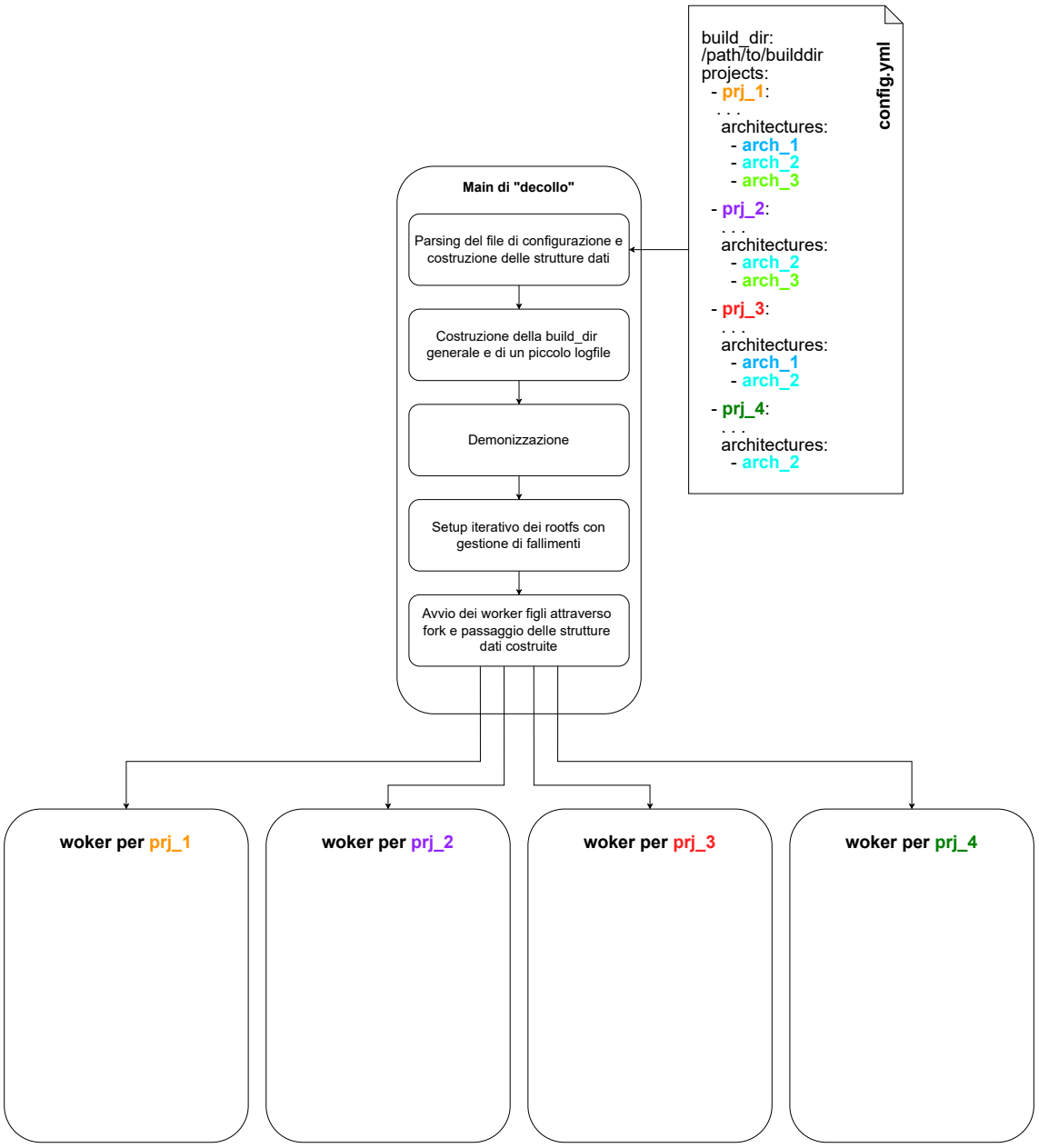

Figura 4.8: Schema concettuale generale di Rootless V^2CI, con main espanso e demoni per progetto rappresentati come contenitori logici opachi

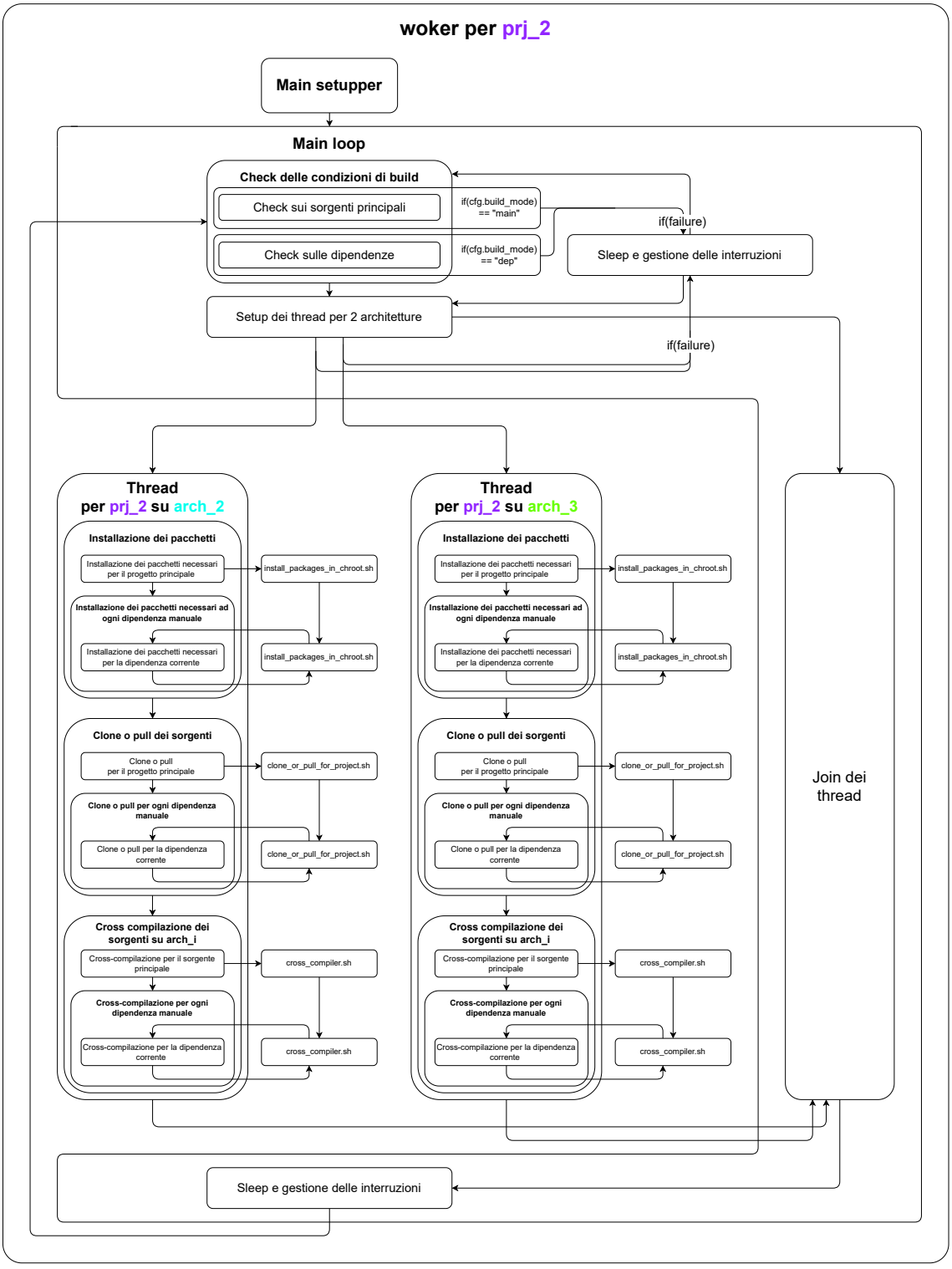

Figura 4.9: "Scorcio interno" del demone per il secondo progetto di esempio ereditato dalla rappresentazione generale di Rootless V^2CI precedente

4.1.4 Raffinamenti e completezza: full build mode, disaster recovery e rotazione dei binari

Sebbene la forma raggiunta da Rootless V^2CI attraverso il processo di sviluppo e assemblaggio, descritto nella sottosezione 4.1.2 e riassunto dai diagrammi 4.8 e 4.9, fosse già sufficientemente completa, funzionale e robusta, al fine di affinare questo sistema di CI distribuito fino a renderlo potenzialmente adatto a un contesto di produzione reale, si è deciso di implementare tre ulteriori funzionalità secondarie, ma non meno importanti: il *full build mode*, un meccanismo di *disaster recovery* e un sistema di *rotazione dei binari* prodotti.
Questi meccanismi, dall'implementazione piuttosto semplice ma dai vantaggi consistenti, sono stati introdotti per ottimizzare tre aspetti principali di Rootless V^2CI, rispettivamente il grado di configurabilità, la persistenza delle risorse e la gestione dello spazio su disco.

Full build mode

Una delle prime ottimizzazioni implementative che un contesto di produzione avrebbe richiesto per Rootless V^2CI, sarebbe stata la possibilità di avviare il processo di cross-compilazione nel caso di aggiornamenti sia ai sorgenti del progetto principale che a quelli delle sue dipendenze, e non esclusivamente nel caso di una sola condizione delle due.
Per maggiore chiarezza, nella versione finale di Rootless V^2CI si intendeva permettere all'utente di specificare, per l'attributo `build_mode` del file di configurazione, un terzo possibile valore `full`, attraverso il quale il demone del progetto corrispondente avrebbe eseguito i check di aggiornamento sia per il repository principale che per quelli delle dipendenze manuali, avviando il processo di build nel caso in cui almeno uno di questi fosse risultato assente o superato da commit più recenti.
L'implementazione di questa funzionalità si è rivelata piuttosto semplice, richiedendo esclusivamente l'inserimento di minimi controlli aggiuntivi nel file `project_worker.c`:

```
int project_worker(project_t *prj, char *main_build_dir) {
  // ... Setup logging, PID file, gestori segnali, directory ...
  // Main loop
  while (1) {
    // ... creazione dei percorsi e setup variabili ...
    need2update = 0;
```

```
    if (strcmp(prj->build_mode, "main") == 0 ||
      strcmp(prj->build_mode, "full") == 0
    ) {
      // ... estrazione nome repo ...
      int update_result = check_for_updates_inside_chroot(
        ..., &need2update, ...
      );
      // ... gestione errori ...
    }
    if (strcmp(prj->build_mode, "dep") == 0 ||
      (strcmp(prj->build_mode, "full") == 0 && !need2update)
    ) {
      manual_dependency_t *cur_manual = prj->manual_dependencies;
      while (cur_manual) {
        // ... estrazione nome repo ...
        int update_result = check_for_updates_inside_chroot(
          ..., &need2update, ...
        );
        // ... gestione errori ...
        if (need2update) {
          break;
        }
        cur_manual = cur_manual->next;
      }
      if (terminate_worker_flag) {
        break;
      }
    }
    if (!need2update) {
      sleep_and_handle_interrupts(...);
      continue;
    }
    // ... avvio thread builder ...
  }
  // ...
}
```

Disaster recovery

Questo secondo miglioramento implementativo è stato quello che ha apportato maggior stabilità, persistenza, garanzia di idempotenza e di coerenza, rispetto a tutti i precedenti accorgimenti introdotti nell'architettura di Rootless V^2CI.
Infatti, come accennato nella sottosezione 4.1.2, la scelta, quasi obbligata, di assegnare il compito di un setup dei rootfs iterativo al main process, piuttosto che all'esecuzione del main loop di ogni demone figlio, avrebbe privato Rootless V^2CI della capacità di ripristinare autonomamente gli ambienti chroot qualora fossero stati corrotti o rimossi tra un'iterazione e l'altra del ciclo di build, capacità invece in dotazione all'antenato monolitico Rootless sshlirpCI.

Per questo motivo, e ispirandosi all'alto grado di coerenza già introdotto all'interno dei demoni dalla funzionalità aggiuntiva di gestione dei fallimenti della costruzione dei rootfs, discussa nella sottosezione 4.1.2, si è deciso di implementare un meccanismo di *disaster recovery* che fosse in grado di ripetere l'intero processo di setup dei chroot al presentarsi di fallimenti critici durante le operazioni che ne avrebbero richiesto l'accesso.
Questa accortezza ha inevitabilmente aumentato il livello di difficoltà nella gestione della concorrenza: esattamente come sarebbe stato necessario per regolare un eventuale chroot setup a livello dei threads foglia, anche in questo caso si è reso indispensabile evitare, tramite locking, che più demoni tentassero di eseguire contemporaneamente un'operazione di ricostruzione dello stesso rootfs.
Nonostante ciò, il guadagno apportato da questo meccanismo di disaster recovery, in termini di persistenza, idempotenza e sicurezza a run-time, ha permesso di superare anche l'alto grado di affidabilità garantito già da Rootless sshlirpCI, il quale appunto mancava di una vera e propria gestione degli errori tramite recovery delle risorse.

L'implementazione di questa ottimizzazione, sebbene un po' più laboriosa rispetto a quella del full build mode, ha previsto la sola aggiunta di due funzioni statiche `handle_recovery()` e `recovery()` internamente al file `project_worker.c`, con l'inserimento di loro invocazioni nelle sezioni di gestione degli errori a seguito di operazioni sui rootfs.

```
// Funzione core di recovery: ricrea directory e rilancia chroot_setup
static int recovery(project_t *prj, FILE **log_fp, char *main_build_dir)
```

```
{
  // 1. Ricreazione directory fondamentali (build_dir, log_file...)
  // ... (omesso per brevita') ...
  // 2. Per ogni architettura, esegue il setup del chroot se mancante
  for (int i = 0; i < prj->arch_count; i++) {
    if (terminate_worker_flag) break;
    // ... setup path chroot ...
    if (chroot_setup(prj->architectures[i], chroot_dir,
      prj->worker_log_file, *log_fp) != 0
    ) {
      // Log errore setup
      return 1;
    }
  }
  return 0;
}

// Wrapper per gestire la concorrenza tra worker durante il recovery
static int handle_recovery(FILE **log_fp, project_t *prj,
  char *main_build_dir
) {
  // Lock tramite file di stato per evitare recovery simultanei
  // su risorse condivise o sovraccarico host
  char recovery_state_file[] = "/tmp/worker.recovery";
  while (access(recovery_state_file, F_OK) == 0) {
    // Attesa attiva se un altro worker sta facendo recovery
    sleep(60);
  }
  // Creazione lock file
  FILE *recovery_fp = fopen(recovery_state_file, "w");
  if (!recovery_fp) return 1;
  // Avvio operazioni di recovery
  int recovery_result = recovery(prj, log_fp, main_build_dir);
  // Cleanup lock file
  fclose(recovery_fp);
  remove(recovery_state_file);
  return recovery_result;
}
```

```
int project_worker(project_t *prj, char *main_build_dir) {
  // ... Setup logging, PID file, gestori segnali ...
  while (1) {
    // ...
    // 1. Controllo aggiornamenti (logica full/main/dep)
    // In caso di errore nel check (es. chroot corrotto), tenta recovery
    if (check_for_updates_inside_chroot(...) != 0) {
      while (handle_recovery(&log_fp, prj, main_build_dir) == 1) {
        // Se il recovery fallisce, riprova dopo poll_interval
        sleep_and_handle_interrupts(prj->poll_interval, ...);
        if (terminate_worker_flag) break;
      }
      if (terminate_worker_flag) break;
      // Se recovery ok, riprova il check immediatamente
      continue;
    }
    // ... Se non ci sono aggiornamenti, sleep e continue ...

    // 2. Avvio thread builder e join
    // ... (creazione thread e join come in precedenza) ...

    // 3. Verifica esito build
    if (failed_builds > 0) {
      // Se le build falliscono (es. dipendenze rotte nel chroot),
      // tenta il recovery dell'ambiente
      while (handle_recovery(&log_fp, prj, main_build_dir) == 1) {
        sleep_and_handle_interrupts(prj->poll_interval, ...);
        if (terminate_worker_flag) break;
      }
      if (terminate_worker_flag) break;
      // Riavvia il ciclo per ritentare la build
      continue;
    }
    // ... Successo e sleep finale ...
  }
  // ... Cleanup ...
  return 0;
```

```
}
```

Rotazione dei binari

Un'eventualità a cui un contesto di produzione reale sarebbe potuto andare incontro è la saturazione dello spazio su disco a causa dell'accumulo di versioni multiple dei binari prodotti dai processi di cross-compilazione, specialmente nel caso in cui questi fossero stati eseguiti con una frequenza elevata e su numerosi sorgenti, tutti soggetti a costanti aggiornamenti e mirati a un numero di architetture target massimo.
Non tanto per evitare il verificarsi di questo scenario - che nel contesto accademico di Rootless V^2CI sarebbe comunque stato piuttosto raro - quanto per conferire ulteriore robustezza e flessibilità a questo ecosistema distribuito, si è deciso di ristrutturare il sotto-albero di fs interno alla target directory di ogni progetto e implementarvi un meccanismo di rotazione dei binari prodotti, basato su politiche di *daily, weekly, monthly, yearly retention*, *frequency interval* e *disk usage limit* configurabili dall'utente, sempre tramite il file di input `config.yml`.

In particolare, per ogni progetto l'utente avrebbe potuto definire una macrosezione `binaries_config` contenente a sua volta due sottosezioni:

- `interval`, la quale avrebbe contenuto per ogni scaglione temporale (`weekly`, `monthly` e `yearly`) l'intervallo minimo in minuti tra due versioni consecutive da conservare - non necessario per lo scaglione `daily`, il cui valore sarebbe stato semanticamente già implicito nel `poll_interval`;
- `mem_limit`, che avrebbe invece specificato, sempre per tutti gli scaglioni temporali, una soglia massima di spazio su disco occupabile dai binari conservati in quella categoria, espressa in kilobyte.

```
build_dir: /home/francesco/v2ci_build
projects:
  - name: sshlirp
    target_dir: /home/francesco/sshlirp_build/target_binaries
    binaries_config:
      interval:
        weekly: 1440    # e.g. 1440 minutes = 1 day
        monthly: 10080  # e.g. 10080 minutes = 7 days
        yearly: 43200   # e.g. 43200 minutes = 30 days
```

```
      mem_limit:
        daily: 10000        # 10000 in KB -> 10 MB
        weekly: 50000       # 50 MB
        monthly: 200000     # 200 MB
        yearly: 1000000     # 1 GB
    source:
    # ...
```

A partire da queste nuove direttive si sarebbe dovuto modificare lo script di compilazione `cross_compiler.sh` e arricchire il comportamento del demone di Rootless V^2CI, integrandolo con un *cronjob*.

Nello specifico della **fase di spostamento dei binari** statici finali, interna allo script di compilazione invocato da ogni thread builder per i sorgenti del progetto di sua competenza, non si sarebbe più potuta applicare la logica di versioning ereditata da sshlirpCI tramite Rootless sshlirpCI che, come descritto nella sottosezione 3.2.1, prevedeva il salvataggio degli eseguibili compilati a partire da un certo `tag` dei sorgenti in una sotto-directory della destinazione target, nominata appunto `target_dir/tag`. Bensì si sarebbe reso necessario, al termine dell'esecuzione di `cross_compiler.sh` sui sorgenti principali di un certo progetto, svolgere le seguenti operazioni:

1. creare, se non esistente, la directory `target_dir/daily`;
2. installarvi il binario selezionato, rinominandolo `target_dir/prj_name-release_version-debian_arch`;
3. eliminare iterativamente gli eseguibili meno recenti, solo se necessario, fino a quando il `mem_limit` su `daily` non fosse rispettato;

Queste ottimizzazioni, riportate di seguito, avrebbero, per le prime iterazioni di Rootless V^2CI, solo impostato l'ambiente di salvataggio dei binari, trascurando ancora la necessità di una rotazione ordinata e rispettosa dei vincoli imposti dall'utente.

```
#!/bin/bash
  # ... (parsing degli argomenti e setup del logging) ...

  # Ingresso nel chroot per eseguire la build
  $thread_chroot_dir/_enter <<EOF
```

```
  # ... (rilevamento del sistema di build: cmake,
  # autotools, meson, makefile) ...
  # ... (compilazione e installazione delle dipendenze) ...

  # Solo per il progetto principale: selezione del binario candidato
  if [ "$main_project" = "yes" ]; then
    # ... (euristica per trovare l'eseguibile linkato staticamente) ...
    cp -f "\$selected_binary" \
      "$thread_chroot_target_dir/$repo_name-$debian_arch"
  fi
EOF

# Solo per il progetto principale, rotazione e salvataggio
if [ "$main_project" = "yes" ]; then
  cd "$thread_chroot_dir$thread_chroot_build_dir/$repo_name"
  current_tag=$(git describe --tags --abbrev=0 2>/dev/null)
  release_version=${current_tag:-"unstable"}

  # 1. Creazione della directory daily
  mkdir -p "$project_target_dir/daily"

  # 2. Installazione del binario con nome versionato
  install -m 0755 \
  "$thread_chroot_dir$thread_chroot_target_dir/$repo_name-$debian_arch" \
  "$project_target_dir/daily/$repo_name-$release_version-$debian_arch"

  # 3. Applicazione del limite di memoria per le build giornaliere
  total_daily_size=$(du -s "$project_target_dir/daily" | cut -f1)
  while [ "$total_daily_size" -gt "$mem_limit" ]; do
    oldest_file=$(ls -t "$project_target_dir/daily" | tail -n 1)
    # Evita di cancellare il file appena aggiunto
    if [ "$oldest_file" = \
         "$repo_name-$release_version-$debian_arch" ]; then
      break
    fi
    oldest_file_size=$(du -s \
      "$project_target_dir/daily/$oldest_file" | cut -f1)
    rm -f "$project_target_dir/daily/$oldest_file"
```

```
    total_daily_size=$((total_daily_size - oldest_file_size))
  done
fi
exit 0
```

Per quanto riguarda invece la fase di **rotazione dei binari ad esecuzione avviata**, si è deciso di servirsi del demone `cron` [63], addetto all'esecuzione periodica di comandi o script a intervalli di tempo predefiniti, istruendolo attraverso l'editing del file `/var/spool/cron/crontabs/<user>` [64] personale dell'utente, tramite il comando nativamente rootless `crontab -u <user> <file>` [65].
Per automatizzare questo processo di setup e creazione di un work temporizzato per la rotazione dei binari, si è scelto di creare uno script bash `binaries_rotation_cronjob.sh`, nel quale sarebbe stata contenuta la logica di gestione degli eseguibili salvati, e sviluppare una funzione `set_binaries_rotation_cronjob`, all'interno del file `project_worker.c`, la quale, una volta invocata durante la fase di setup iniziale del demone, avrebbe richiesto un lock esclusivo tramite `flock()` - per non generare corruzioni sul file `crontab` dell'utente, possibili e probabili in un contesto di demoni multi-threaded concorrenti - e svolto le seguenti operazioni, solo al seguito delle quali avrebbe rilasciato le risorse detenute:

1. composizione - secondo i pattern definiti dallo stesso manuale [64] - e salvataggio in una variabile stringa dell'entry di crontab, la quale avrebbe invocato lo script di rotazione dei binari ogni giorno a mezzanotte;
2. copia dei cronjobs esistenti in un file temporaneo, escludendo ogni entry identica a quella appena composta;
3. aggiunta dell'entry di rotazione dei binari al file temporaneo;
4. sostituzione del crontab dell'utente con il file temporaneo, tramite il comando `crontab -u <user> <temp_crontab_file>`.

Queste operazioni sono state implementate come segue:

```
static int set_binaries_rotation_cronjob(project_t *prj, FILE *log_fp) {
  // 0. Acquisizione di un lock globale su /tmp per
  // evitare race conditions tra i worker che tentano
  // di modificare il crontab contemporaneamente
  int lock_fd = open("/tmp/cronjob_lock.lock", O_CREAT | O_RDWR, 0644);
  flock(lock_fd, LOCK_EX);
```

```
// 1. Preparazione dell'entry cronjob per
// eseguire lo script ogni notte
/* Crontab entries have the following format:
* * * * * command to be executed
- - - - -
| | | | |
| | | | +----- Day of week (0 - 7) (Sunday=0 or 7)
| | | +------- Month (1 - 12)
| | +--------- Day of month (1 - 31)
| +----------- Hour (0 - 23)
+------------- Minute (0 - 59)
*/
char cronjob_entry[MAX_COMMAND_LEN];
snprintf(cronjob_entry, sizeof(cronjob_entry),
  "0 0 * * * %s %s %s %d %d %d %d %d %d\n",
  expand_tilde(CRONJOB_SCRIPT_PATH),
  prj->name, prj->target_dir, prj->cronjob_log_file,
  prj->binaries_limits->weekly_mem_limit, ...
);

// 2. Lettura dei cronjob esistenti tramite 'crontab -l' e copia in un
// file temporaneo, escludendo eventuali duplicati dell'entry corrente
FILE *cron_pipe = popen("/usr/bin/crontab -u $USER -l", "r");
FILE *cron_fp = fopen(temporary_crontab_file, "w");
while (fgets(line, sizeof(line), cron_pipe)) {
  if (strcmp(line, cronjob_entry) != 0) {
    fputs(line, cron_fp);
  }
}
// ... chiusura pipe e file ...

// 3. Aggiunta della nuova entry al file temporaneo
cron_fp = fopen(temporary_crontab_file, "a");
fputs(cronjob_entry, cron_fp);
fclose(cron_fp);

// 4. Installazione del nuovo crontab
```

```
    char command[MAX_COMMAND_LEN];
    snprintf(command, sizeof(command), "/usr/bin/crontab -u %s %s",
      getenv("USER"), temporary_crontab_file
    );
    system_safe(command);

    // 5. Rilascio del lock e pulizia
    flock(lock_fd, LOCK_UN);
    close(lock_fd);
    remove(temporary_crontab_file);
    return 0;
}
```

Infine, lo script di rotazione dei binari `binaries_rotation_cronjob.sh` sarebbe stato responsabile, per ogni sua invocazione, di operazioni di spostamento ed eliminazione "a cascata", guidato dalle politiche definite dall'utente e procedendo secondo una logica di funzionamento riassumibile nei seguenti passi:

1. Eliminazione dalla directory `target_dir/yearly` dei binari prodotti più di un anno fa, a partire dalla data di esecuzione dello script;
2. Spostamento, tramite una funzione di utility `rotation_engine()`, dei binari più vecchi di un mese dalla directory `target_dir/monthly` a `target_dir/yearly`, con controlli e operazioni preliminari per il rispetto dei vincoli sugli intervalli e sulle soglie di memoria. In particolare, tale funzione avrebbe eliminato iterativamente i binari mensili meno recenti fino a quando la distanza temporale tra il file annuale più recente e quello mensile più vecchio non fosse stata almeno pari all'intervallo minimo definito per lo scaglione temporale `yearly`; inoltre, prima di effettuare lo spostamento di un binario candidato, avrebbe anticipatamente verificato che la soglia di memoria definita per lo scaglione `yearly` non sarebbe stata superata, eliminando in tal caso i file annuali meno recenti che impedivano il rispetto del vincolo.
3. Ripetizione del passo precedente per lo spostamento dei binari più vecchi di una settimana dalla directory `target_dir/weekly` a `target_dir/monthly`;
4. Rotazione dei binari più vecchi di un giorno dalla directory `target_dir/daily` a `target_dir/weekly`, seguendo sempre la stessa logica.

A proposito dell'ordine di esecuzione delle operazioni sopra citate, è importante sottolineare che l'inversione della sequenza di rotazione, partendo cioè dallo spostamento dei binari giornalieri fino ad arrivare a quelli annuali, nonstante la sua maggior

logicità apparente, avrebbe portato a effettuare eliminazioni premature di eseguibili che invece, dopo un preventivo "cleanup" della directory successiva a quella di loro appartenenza, sarebbero potuti essere ruotati comunque nei vincoli di memoria e di intervallo temporale.

L'intero processo di rotazione vincolata è stato poi trascritto nel corrispondente file bash come segue:

```
#!/bin/bash
# ... (setup variabili e funzioni helper per calcolo date) ...

rotation_engine() {
  local rotation_type=$1
  local current_dir=$2
  local later_dir=$3
  # ... (impostazione limiti
  # MEM_LIMIT,
  # INTERVAL_LIMIT,
  # MIN_AGE
  # in base al tipo) ...

  while true; do
    oldest_current_file=$(ls -t "$current_dir" | tail -n 1)
    # ... (check esistenza file e verifica eta' minima per rotazione) ...

    # Verifica vincolo intervallo temporale con il file piu'
    # recente nella directory successiva
    recent_later_file=$(ls -t "$later_dir" | head -n 1)
    if [ -n "$recent_later_file" ]; then
      # ... (calcolo differenza temporale) ...
      if [ "$minutes_diff" -lt "$INTERVAL_LIMIT" ]; then
        # Vincolo non rispettato: elimino il file corrente
        # (troppo frequente)
        rm -f "$current_dir/$oldest_current_file"
        continue
      fi
    fi
```

```
    # Verifica vincolo memoria nella directory di destinazione
    # ... (calcolo dimensioni file e directory) ...
    while [ "$forcasted_later_dir_mem_kb" -gt "$MEM_LIMIT" ]; do
      # Elimina i file piu' vecchi nella directory successiva per
      # fare spazio
      oldest_later_file=$(ls -t "$later_dir" | tail -n 1)
      rm -f "$later_dir/$oldest_later_file"
      # ... (aggiornamento stima memoria occupata) ...
    done

    # Spostamento effettivo del file
    mv "$current_dir/$oldest_current_file" "$later_dir/"
  done
}

# ... (setup logging e check directory daily) ...

# 1. Pulizia directory YEARLY (elimina file piu' vecchi di un anno)
# ... (ciclo di rimozione basato su eta' file) ...

# 2. Rotazione MONTHLY -> YEARLY
mkdir -p "$yearly_dir"
rotation_engine "monthly" "$monthly_dir" "$yearly_dir"

# 3. Rotazione WEEKLY -> MONTHLY
mkdir -p "$monthly_dir"
rotation_engine "weekly" "$weekly_dir" "$monthly_dir"

# 4. Rotazione DAILY -> WEEKLY
mkdir -p "$weekly_dir"
rotation_engine "daily" "$daily_dir" "$weekly_dir"
```

È possibile infine specificare che, essendo un processo non esente da bug o scenari di fallimento, si è deciso di distribuire il sistema di logging di Rootless V^2CI - ancora retaggio di Rootless sshlirpCI - anche all'interno di questo componente aggiuntivo, che avrebbe quindi tracciato la sua esecuzione temporizzata su un log file dedicato e unico per l'esecuzione dell'intera infrastruttura.

L'implementazione di questi tre raffinamenti ha terminato il lungo percorso di sviluppo, originato dallo scopo di una semplice pacchettizzazione di sshlirp e confluito in un sistema avanzato e ottimizzato di continuous integration per sorgenti multipli e port target eterogenei.
La completezza di Rootless V^2CI, e la sua supremazia rispetto ai propri antenati in termini di configurabilità, estensione, scalabilità, persistenza e sicurezza, è dovuta però anche all'assunzione della disponibilità di una certa mole di risorse di sistema computazionali e di storage, capaci di tollerare costi significativi.

4.1.5 Calcolo dei costi di Rootless V^2CI

Parallelismo innestato, progetti multipli, architetture multiple, dipendenze di sistema e sorgenti numerosi, sono tutti fattori variabili, dipendenti dalla configurazione di input dell'utente, che possono determinare per Rootless V^2CI un'esecuzione controllata o, al contrario, estremamente onerosa sia in termini di tempi di esecuzione e di costi computazionali, che di consumo delle risorse di storage.

Analisi dei costi computazionali

Per lo svolgimento della seguente analisi computazionale e algoritmica di **un solo ciclo** di build, si è ovviamente assunto lo scenario di *Worst Case Execution Time* (WCET), ovvero che la `build_mode` sia impostata al valore `full`, da configurazione, e che vengano rilevati aggiornamenti, innescando effettivamente il processo di build per tutti i target.
Per questo motivo non è da intendersi come una *media* dei costi computazionali, bensì come una stima del loro massimo globale.

Definizioni delle Variabili Siano date le seguenti variabili di input definite nel file di configurazione `config.yml`:

- n: numero totale di progetti;
- m: numero di architetture target per progetto;
- k: numero di dipendenze manuali per progetto;
- j: numero di pacchetti di sistema per il progetto;
- y: numero di pacchetti di sistema per ogni dipendenza manuale;
- u: numero di architetture *uniche* nell'unione di tutti i progetti ($1 \leq u \leq n \times m$).

1. Analisi della fase di inizializzazione Il processo sequenziale di setup iniziale, eseguito dal `main.c` prima del fork dei processi worker, come già precedentemente descritto, è composto dalle seguenti sotto-fasi:

1. **Parsing della configurazione**: trascurabile rispetto al resto, in quanto limitato dalla dimensione del solo file `config.yml`.
2. **Calcolo delle architetture Uniche**: dal momento che il sistema itera su tutti gli n progetti e le loro m architetture per creare una lista unica `archs_list`, possiamo stimare un costo di $O(n \cdot m)$.
3. **Chroot setup sequenziale**: in questo ciclo `for`, il `main.c` itera u volte chiamando `chroot_setup.sh` e invocando `debootstrap`; identificando il costo delle operazioni di quest'ultimo T_{setup}, possiamo stimare una complessità di $O(u \cdot T_{setup})$.

Costo totale dell'inizializzazione main: $W_{init} \approx O(n \cdot m + u \cdot T_{setup})$

2. Analisi della vita dei demoni per progetto Una volta completato il setup, il main fork-a n processi worker, il cui flusso di esecuzione con `build_mode:full`, dopo un'inizializzazione della porzione di fs dedicata alla build del progetto, di costo approssimabile a $O(1)$, prevede la fase di controllo degli aggiornamenti sia per il repository principale che per quelli delle sue dipendenze sorgenti.
Quindi, con un totale di $1 + k$ chiamate a `check_updates.sh` e un costo temporale assunto di T_{git_check} per le operazioni di `git remote update` e `git rev-parse`, ipotizzando WCET, ossia esecuzione avviata e non check costante causa rootfs assente, abbiamo:
Costo totale per progetto pre-thread: $W_{check} \approx O((1 + k) \cdot T_{git\_check})$.

3. Analisi della vita dei thread builder Considerando sempre il *worst case scenario*, ognuno degli n demoni per progetto rileva un aggiornamento e procede quindi a lanciare m thread builder, i quali si occupano di eseguire la build per ogni architettura target, ognuno aggiungendo consistenti costi per le seguenti operazioni:

1. **Installazione delle dipendenze di sistema**: in questa fase vengono installati j pacchetti per il progetto e y pacchetti per ogni dipendenza, per un totale di $j + k \cdot y$ pacchetti. Stimando un costo di installazione tramite `apt` di T_{apt}, otteniamo quindi $O((j + k \cdot y) \cdot T_{apt})$;
2. **Clone/pull dei sorgenti**: il thread esegue il clone o il pull - di cui stimiamo il costo con T_{clone} - del repository principale e di ogni dipendenza, per un costo totale di $O((1 + k) \cdot T_{clone})$;

3. **Cross-compilazione**: infine, il thread esegue la compilazione delle k dipendenze e del progetto principale. Ipotizzando che il costo medio di una compilazione da sorgente sia C_{build}, abbiamo un costo computazionale totale di $O(k \cdot C_{build} + C_{build}) = O((k+1) \cdot C_{build})$.

Costo totale per thread builder: $W_{thread} = W_{install} + W_{clone} + W_{compilation} \approx O((j + k \cdot y) \cdot T_{apt} + (1+k) \cdot T_{clone} + (k+1) \cdot C_{build})$.

4. Formula globale della complessità computazionale Possiamo ora formulare il calcolo del lavoro totale W_{total} eseguito dal sistema in un ciclo completo:

$$W_{total} = W_{init} + \sum_{p=1}^{n} \left(W_{check,p} + \sum_{a=1}^{m} W_{thread,p,a} \right)$$

Espandendo i termini in base alle variabili $n, m, k, j, y, u, T_{setup}, T_{git\_check}, T_{apt}, T_{clone}, C_{build}$, otteniamo:

$$W_{total} \approx O\left(n \cdot m + u \cdot T_{setup}\right) + \sum_{p=1}^{n} \left(O\left((1+k_p) \cdot T_{git\_check}\right) + \sum_{a=1}^{m} \left(O\left((j_{a,p} + k_{a,p} \cdot y_{a,p}) \cdot T_{apt} + (1+k_{a,p}) \cdot T_{clone} + (k_{a,p}+1) \cdot C_{build}\right) \right) \right)$$

Considerando costanti i tempi di esecuzione $T_{setup}, T_{git\_check}, T_{apt}, T_{clone}, C_{build}$ e uniformi le variabili k, j, y per tutti i progetti e architetture, possiamo semplificare la formula come segue:

$$W_{total} \approx O\left(n \cdot m + u\right) + n \cdot O\left(1+k\right) + n \cdot m \cdot O\left((j + k \cdot y) + (1+k) + (k+1)\right)$$

Semplificando ulteriormente:

$$\begin{aligned} W_{total} &\approx O\left(n \cdot m + u\right) + O\left(n \cdot (1+k)\right) + O\left(n \cdot m \cdot (j + k \cdot y + 2k + 2)\right) \\ &\approx O\left(n \cdot m\right) + O\left(n \cdot k\right) + O\left(n \cdot m \cdot (k \cdot y + j + k)\right) \\ &\approx O\left(n \cdot m \cdot (k \cdot y + j + k)\right) \\ &\approx O\left(n \cdot m \cdot k \cdot y + n \cdot m \cdot j + n \cdot m \cdot k\right) \end{aligned}$$

Infine supponendo $y \approx j$:

$$W_{total} \approx O\left(n \cdot m \cdot k \cdot y\right)$$

Analisi dei costi di storage

Più brevemente, è possibile stimare i costi di storage richiesti da Rootless V^2CI ereditando le variabili definite nella precedente valutazione dei costi computazionali e assumendo che S_{chroot} sia lo spazio su disco occupato da un singolo rootfs debootstrap-ato (considerabile uniforme per tutte le architetture), e che $S_{build\_artifacts}$ sia la quota richiesta dai sorgenti clonati e dalle dipendenze installate all'interno dell'ambiente chroot.

$$S_{total} \approx (u \cdot S_{chroot}) + \left(n \cdot m \cdot S_{build\_artifacts}\right)$$

Come prima, considerando il caso pessimo di $u \approx n \cdot m$:

$$\begin{aligned} S_{total} &\approx (n \cdot m \cdot S_{chroot}) + \left(n \cdot m \cdot S_{build\_artifacts}\right) \\ &\approx O\left(n \cdot m \cdot (S_{chroot} + S_{build\_artifacts})\right) \end{aligned}$$

4.1.6 Analisi sui consumi reali delle risorse di sistema

Per quanto Rootless V^2CI possa essere specchio di una perfetta armonia tra *HA* e performance, la sua esecuzione, come intuito dalla precedente sottosezione, richiede inevitabilmente l'impiego di risorse di calcolo e disco consistenti.
Infatti, al di là dei costi computazionali dipendenti da parametri di configurazione variabili, nella maggior parte degli scenari d'uso, la fetta più significativa di risorse di sistema indispensabili per il funzionamento di Rootless V^2CI è da attribuire a quei fattori, sì considerabili costanti, ma decisamente impattanti sul consumo totale (T_{setup}, T_{git_check}, T_{apt}, T_{clone} e C_{build}).
Conducendo test di esecuzione per un solo progetto su host Ubuntu 24.04.3 LTS con processore 11th Gen Intel® Core™ i5-1155G7 × 8 e 16 GB di RAM, si sono infatti potute svolgere diverse considerazioni aggiuntive e trarre conclusioni pratiche circa i seguenti aspetti:

- **Consumo delle risorse di calcolo durante il setup**: come per Rootless sshlirpCI, una delle principali fonti di spesa di risorse di sistema computazionali durante l'esecuzione di Rootless V^2CI è legata alla fase di setup iterativo dei rootfs; lo spostamento di questa da un contesto di concorrenza serializzata a uno di esecuzione ciclica isolata ha però sia abbassato il consumo percentuale di RAM che ridotto, anche se minimamente, l'impiego delle CPUs; in particolare, i test condotti hanno mostrato per Rootless sshlirpCI un'occupazione

aggiuntiva di circa un 2% di memoria (equivalente a 328MB sull'host di esecuzione) rispetto a un solo $1,2\%$ (circa 197MB) richiesto invece dal main setupper di Rootless V^2CI. Per quanto riguarda invece l'utilizzo delle CPUs i due processi hanno mostrato consumi molto simili, corrispondenti a valori che variavano per ogni core tra il 100% - quando il carico di lavoro veniva distribuito dal sistema su una o due sole unità di calcolo - e il 22% - quando invece le operazioni richieste dall'esecuzione di `debootstrap` venivano bilanciate equamente tra tutti gli 8 core disponibili sull'host di test.
Queste osservazioni hanno portato a concludere che l'esecuzione della costruzione dei rootfs in concomitanza all'esistenza in memoria di più threads builder - come accadeva in Rootless sshlirpCI - comportava un leggero sovraccarico di memoria, e che invece il suo spostamento in una fase di setup iterativo avrebbe garantito un'economia più saggia e conveniente delle risorse computazionali. Inoltre, essendo questa deduzione indipendente dal numero di `project` per cui esegue Rootless V^2CI, si dimostra di valenza "globale" la supremazia dell'architettura di quest'ultimo ecosistema sviluppato rispetto sia alla sua versione alternativa, pensata nella sezione 4.1.1, che al monolite "thread-level" di Rootless sshlirpCI.
Detto ciò, il consumo di risorse di calcolo durante questa fase è rimasto comunque non indifferente, a causa della natura intrinseca di `debootstrap`, portando l'host a completare l'operazione per 4 port target in circa 19 minuti.

- **Consumo delle risorse di calcolo durante la build**: sebbene si possa pensare che il setup serializzato dei rootfs sia di gran lunga l'operazione più dispendiosa, la preparazione degli ambienti di build e le cross-compilazioni in parallelo hanno mostrato consumi di CPU e RAM ben più elevati, a causa dell'elevato grado di parallelismo raggiunto con l'esecuzione multi-threaded dei builder, sia in Rootless sshlirpCI che in Rootless V^2CI.
  Infatti, se si fosse tentato di impiegare tale grado di concorrenza anche per i chroot setup, almeno uno di questi sarebbe fallito, mostrando lo stesso comportamento documentato per la prima versione di sshlirpCI nella sottosezione 3.2.1.
  I processi di build invece, in quanto più "leggeri" presi singolarmente, sono stati eseguiti integralmente in parallelo senza una regolazione delle risorse computazionali, mostrando però un consumo di CPU e RAM molto elevato.
  In particolare, i test condotti - sempre su un solo progetto e 4 architetture target, ossia in un contesto ancora Rootless sshlirpCI-like - hanno generato,

appunto durante la fase di build, un Δ di occupazione RAM massimo del $9,6\%$ (raggiunto durante la cross-compilazione), minimo del $4,5\%$ (durante la fase di setup dei builder) e medio del $7,1\%$, equivalente a circa $1,16$GB di memoria aggiuntiva richiesta sull'host di test.
La percentuale di occupazione RAM si è inoltre rivelata estremamente connessa al grado di impiego dei core, i quali, in modo del tutto simile, hanno tutti contemporaneamente raggiunto picchi di utilizzo dell'80%, minimi del 12% e medi del 53%, operando sempre in modo bilanciato.
Questi risultati hanno confermato come la natura altamente concorrente di Rootless V^2CI sia sì in grado di completare le operazioni di build in appena 6 minuti, ma anche bisognosa di consistenti risorse di calcolo, in particolar modo quando il numero di progetti e port target aumenta.

- **Impiego del disco**: a differenza di quanto detto per le risorse computazionali, per quelle di storage è possibile affermare che il consumo più consistente deriva dalla persistenza degli ambienti chroot sul filesystem. Quest'ultimi infatti, indipendentemente dalle loro origini - ovvero, sia che siano stati costruiti da Rootless sshlirpCI che da Rootless V^2CI - richiedono in totale e in media circa $4,8$GB di spazio su disco, con rootfs che occupano da un minimo di 990MB (per port più "semplici" come `armhf`) a un massimo di $1,5$GB (registrato per `riscv64`).
  A questa "spesa fissa" si aggiunge poi lo spazio richiesto per i sorgenti e i binari prodotti, il quale però, rispetto a quello necessario per gli ambienti chroot, risulta decisamente trascurabile, in caso di progetti "leggeri" e non troppo numerosi.

I risultati dei test genitori di queste considerazioni possono inoltre essere riassunti nella seguente tabella:

Aspetto	Rootless sshlirpCI			Rootless V^2CI			Note
	Max	Min	Medio	Max	Min	Medio	
RAM (Setup)	N/A	N/A	~2% (328MB)	N/A	N/A	~1,2% (197MB)	Valori aggiuntivi rispetto all'idle.
CPU (Setup)	100%	5%	22%	100%	5%	22%	Bilanciamento variabile: 100% su 1-2 core, 22% se distribuito su 8 core.
RAM (Build)	9,6%	4,5%	7,1% (~1,16GB)	9,6%	4,5%	7,1% (~1,16GB)	Valori identici per singolo progetto.
CPU (Build)	80%	12%	53%	80%	12%	53%	Carico bilanciato equamente su tutti gli 8 core.
Storage (Rootfs)	1,5GB	990MB	~1,2GB	1,5GB	990MB	~1,2GB	Totale per 4 architetture: ~4,8GB.

Tabella 4.1: Tabella riassuntiva dei consumi di risorse di sistema rilevati durante i test.

Da queste analisi approfondite si è tornati a confermare la natura di Rootless V^2CI: un potente motore estremamente avanzato ma, sebbene ottimizzato sotto diversi aspetti e categorizzabile come prodotto finito, anche affetto da consumi di "carburante" computazionale e di storage non trascurabili, e quindi non alla portata di tutti gli ambienti di esecuzione.

4.1.7 Perfezionamento del sistema di logging: scopi dell'omogeneizzazione e arricchimento dei pattern

L'ultimo gradino aggiunto alla "scala evolutiva" di Rootless V^2CI ha consentito di conquistare "user-friendliness" e monitorabilità avanzata.
Un'integrazione e una connessione con uno stack ELK dedicato all'ingestion, il parsing, l'enrichment e la visualizzazione dei log prodotti, infatti non sarebbe stato possibile senza una normalizzazione di quest'ultimi.
Questa operazione, per quanto cruciale nel percorso verso il raggiungimento del prodotto finale, è stata di facile implementazione e di minimo impatto sui sorgenti di Rootless V^2CI.
Nello specifico, è stato sufficiente sostituire ogni `fprintf` di log interno ai componenti `.c` - o analogo `echo` degli script `.sh` - con una funzione di utility `formatted_log()` che si sarebbe occupata di arricchire ogni stampa con dati aggiuntivi, componendoli in un log finale dalla notazione JSON-like, come mostra il seguente estratto di codice del componente `utils.c`:

```
void formatted_log(FILE *log_file, const char *log_level,
  const char *source_file, int line_number, const char *project_name,
```

```
    const char *thread_arch, const char *format, ...) {

    // Recupero informazioni sull'host (IP, OS, Arch, Modello)
    // tramite esecuzione di comandi shell (curl, uname, hostnamectl)
    char ip[128], os[128], arch[128], agent[128];
    get_client_stats("curl ... https://api.ipify.org", ip, ...);
    get_client_stats("uname -o", os, ...);
    // ... (altre chiamate per architettura e modello hardware) ...

    // Formattazione del messaggio variabile
    char message_buffer[2048];
    va_list args;
    va_start(args, format);
    vsnprintf(message_buffer, sizeof(message_buffer), format, args);
    va_end(args);

    // Stampa del log con timestamp e metadati in formato JSON-like
    log_time(log_file);
    fprintf(log_file, "[%s] source: { client: { ip: %s, os: %s, ... }, "
      "location: { file: %s, line: %d } }, project: %s, "
      "thread_arch: %s, message: %s\n",
      log_level, ip, os, source_file, line_number,
      project_name ? project_name : "N/A",
      thread_arch ? thread_arch : "N/A",
      message_buffer
    );
}
```

Dopo quest'ultimo perfezionamento, Rootless V^2CI non solo aveva raggiunto la sua forma definitiva, ma si era anche dotato di un sistema di logging strutturato e omogeneo, pronto per essere ingerito e processato da stack di monitoraggio esterni.

4.2 Integrazione con stack ELK

La scoperta delle tecnologie ELK - acronimo di *ElasticSearch*, *Logstash* e *Kibana* - ha permesso di ambire a un grado di usabilità di Rootless V^2CI che, con i soli strumenti impiegati fino a questo momento, sarebbe stato impossibile da raggiungere.

L'adozione di un'architettura containerizzata distribuita per l'esecuzione di questi servizi, la scelta degli agenti di monitoring più adatti al contesto di Rootless V^2CI e il processo di configurazione per l'invio, l'elaborazione e la visualizzazione dei log prodotti, hanno sì introdotto la necessità innegabile di ulteriori risorse di sistema e di privilegi elevati, ma hanno anche permesso di completare il quadro di un sistema di continuous integration user-friendly, monitorabile in tempo reale e distribuibile.
Senza quest'accessoria fase di sviluppo infatti, Rootless V^2CI non avrebbe potuto vantare di un metodo di controllo e analisi di esecuzione centralizzato in scenari di istanziazione contemporanea su più server o macchine virtuali; e anche se eseguito su un singolo host, la scelta ottimizzata di non unificare in un unico log file tutte le stampe prodotte dai suoi componenti, avrebbe reso estremamente complessa la lettura e l'interpretazione dei dati di esecuzione.
L'architettura compatta dei servizi elastic ha quindi permesso sia di rendere più accessibile la verifica a run-time dell'infrastruttura di CI, che di facilitarne la raccolta e l'analisi di log generati da sue istanze multiple, in un'ottica di scalabilità orizzontale e di orchestrazione semplificata.

4.2.1 Cenni a ELK

Lo stack ELK è una suite di strumenti open-source [66] sviluppata da Elastic N.V., progettata per semplificare drasticamente la gestione e l'analisi di dati time-series, come logs e metriche [67].
Il fulcro e la fonte di sviluppo di questi strumenti è rappresentato da *Elasticsearch*, il quale funge da indicizzatore distribuito per i dati raccolti.
Inoltre, la sua integrazione con *Logstash* consente di ingerire, trasformare e arricchire i dati provenienti da molteplici fonti, mentre quella con *Kibana* offre un'interfaccia grafica intuitiva per la visualizzazione e l'analisi interattiva di tali dati.
Infine, un tassello fondamentale per la raccolta dei log in ambienti distribuiti è rappresentato da *Beats*, una collezione di agenti leggeri progettati per inviare dati direttamente alle pipelines di Elasticsearch o per farli transire preliminarmente da Logstash.

ElasticSearch

Impiegato da importanti community e piattaforme come Wikimedia [68], Mozzilla [69], GitHub [70] e Netflix [71], Elasticsearch è un motore di ricerca e analisi distribuito, open source e sviluppato in Java, costruito sulla libreria Apache Lucene

[66, 72].
Rilasciato inizialmente nel 2010, si è affermato come standard industriale per la ricerca full-text, l'analisi di log strutturati e non strutturati e la gestione di dati geospaziali. A differenza dei tradizionali RDBMS, Elasticsearch è classificato come un database NoSQL orientato ai documenti con un'architettura progettata per offrire scalabilità orizzontale, HA e capacità di gestione di grandi volumi di dati in NRT (*Near Real-Time*), con una latenza tipica di un secondo tra l'indicizzazione del documento e la sua disponibilità per la ricerca [72].

Per comprendere il funzionamento dell'**architettura** intrinsecamente distribuita di Elasticsearch, è necessario definire i concetti logici e fisici che governano il sistema [67]:

- **Nodi**: quest'unità alla base dell'infrastruttura è sostanzialmente una singola istanza del server Elasticsearch in esecuzione, la cui composizione multipla prende il nome di *Cluster*.
  Ogni cluster è identificato da un nome univoco e orchestra automaticamente la distribuzione dei dati e delle query tra i nodi disponibili.
  Inoltre ogni nodo può essere configurato per svolgere ruoli specifici (come `master`, `data_content`, `data_hot`, `data_warm`, `data_cold`, ecc.), ottimizzando così le prestazioni e la gestione delle risorse in base ai carichi di lavoro previsti e alle politiche di *retention* e *rollover* dei dati.
- **Indici e Documenti**: a livello logico, i dati sono organizzati in Indici. Un indice è una collezione di documenti che condividono caratteristiche simili ed è analogo a una "tabella" in un database relazionale, sebbene questa analogia sia puramente funzionale.
  All'interno di un indice, i dati sono memorizzati sotto forma di Documenti, serializzati in formato JSON.
- **Shard primari e Replica**: per garantire sia scalabilità orizzontale che HA, Elasticsearch implementa il concetto di Sharding. Un indice può essere suddiviso in più frammenti chiamati shards. Ogni shard, che è essenzialmente un'istanza autonoma e completa di Apache Lucene, può essere classificato in due tipi:
  - **Primary Shard**: dove avvengono le operazioni di scrittura originali.
  - **Replica Shard**: copie dei primary shard, utilizzate per aumentare la disponibilità del sistema (failover) e per parallelizzare le operazioni di

lettura, migliorando il throughput delle ricerche, a discapito però di un leggero aumento del carico di scrittura e dello spazio richiesto su disco.

In questo contesto architetturale, è fondamentale specificare che i Nodi sono componenti che non necessariamente devono essere deploy-ati su server multipli. Chiaramente in un contesto di produzione aziendale, pratiche di sharding e replicamento mirate all'HA perderebbero di senso se eseguite su uno stesso host, ma nel contesto di sviluppo di Rootless V^2CI, dato lo scope intrinsecamente accademico, si è deciso di distribuire i nodi Elasticsearch come container Docker su un host centralizzato.

Una caratteristica rilevante di Elasticsearch per lo sviluppo dell'integrazione ELK a Rootless V^2CI, riguarda la sua usabilità: **l'interazione** con questo motore di ricerca avviene tramite un'API RESTful completa. In aggiunta, per le interrogazioni complesse, lo strumento ELK fornisce un potente linguaggio dedicato chiamato **DSL** (*Domain Specific Language*), basato su JSON, che permette di combinare filtri, query full-text e aggregazioni analitiche in un'unica richiesta [67, 72].

Logstash

Logstash è un motore di elaborazione dati che funge da componente di ingestion nella moderna architettura di gestione dei log, sviluppato in Java e Ruby [66].
La sua dinamica di funzionamento è accostabile a quella di una pipeline ETL (*Extract, Transform, Load*): acquisisce dati da molteplici sorgenti simultaneamente, li trasforma per normalizzarli e arricchirli, e infine li invia a una o più destinazioni, tipicamente Elasticsearch [73]. In sostanza, Logstash è generalmente impiegato per risolvere il problema della normalizzazione di dati eterogenei: dal momento che i log di sistema, i messaggi applicativi e le metriche di rete vengono generati in formati disparati, esso agisce come strato di mediazione logica prima dell'archiviazione persistente. In Rootless V^2CI invece, questo strumento è stato impiegato principalmente per permettere l'ingestion e il parsing dei log già strutturati prodotti dall'infrastruttura di CI, al fine di renderli interrogabili e visualizzabili in Kibana.

Per quanto riguarda l'architettura della **pipeline** di Logstash, ognuno dei suoi tre stadi è gestito da plugin specifici che operano all'interno del ciclo di vita dell'evento. Con il primo stadio di *Input*, responsabile dell'ingestion vera e propria, Logstash si occupa di covertire i dati in un formato interno basato su eventi, "mettendosi in ascolto" su una o più fonti di dati [73]:

- **File**: lettura in *tailing* di file di log, simile al comando Unix `tail -f`;
- **Syslog**: ascolto su porte standard (es. 514) per messaggi di sistema conformi all'RFC3164 [74];
- **Beats**: ricezione di dati inviati da *data shippers* leggeri installati sugli edge node;
- **TCP/UDP**: gestione generica di socket di rete.

Lo stadio di *Filter* invece rappresenta il nucleo computazionale di Logstash. Qui avvengono la strutturazione e l'arricchimento dei dati grezzi tramite filtri che vengono applicati condizionalmente e in sequenza [73]:

- **Grok**: è il filtro più rilevante per la strutturazione di testo non strutturato. Utilizza pattern basati su RegEx per estrarre campi semantici (es. indirizzi IP, timestamp) da stringhe di testo arbitrarie;
- **Mutate**: permette di rinominare, rimuovere, sostituire e modificare i campi dati;
- **GeoIP**: arricchisce i dati contenenti indirizzi IP aggiungendo coordinate geografiche (latitudine/longitudine) consultando database interni (es. MaxMind);

Infine lo stadio terminale di *Output* instrada i dati elaborati verso la destinazione specificando l'identificativo dell'host target, il formato con cui salvare i dati (*index*, per targettizzare un indice specifico, o *data stream* per delegare a Elasticsearch la gestione di piccoli *backing indexes* tramite l'incapsulamento in un unico flusso) ed eventuali configurazioni di connessione ssl.
Sebbene Elasticsearch sia l'output primario nell'ecosistema Elastic, Logstash supporta l'invio verso email, file system, servizi cloud o altri broker di messaggistica come *Kafka* [73].

Kibana

Kibana, ultimo componente cardine dello stack ELK, è un'applicazione frontend, sviluppata principalmente in Node.js [66], che funge da interfaccia utente per l'Elastic Stack [75]. Il suo ruolo non è limitato alla semplice presentazione grafica; Kibana agisce come un client amministrativo e analitico per Elasticsearch, traducendo le interazioni visive dell'utente in query RESTful complesse inviate al cluster sottostante, permettendo l'esplorazione operativa dei dati, l'analisi dei time-series e la gestione della sicurezza del cluster.
Nello sviluppo di Rootless V^2CI, Kibana è stato uno strumento fondamentale per le fasi di verifica degli indici, di testing e di creazione delle dashboards.

Beats

Nelle moderne architetture distribuite, la centralizzazione dei log richiede *data shippers* installati direttamente sui nodi *edge* di origine.
La piattaforma Beats è una famiglia di agenti leggeri, scritti in Go [66].
Mentre Logstash opera come aggregatore server-side, i Beats operano secondo il paradigma di *forwarder*: raccolgono i dati localmente e li inviano verso Logstash (per un'elaborazione complessa) o direttamente verso Elasticsearch [76].
Sebbene esistano diversi Beat specializzati (come *Metricbeat* per le metriche e *Packetbeat* per i dati di rete), il componente più diffuso e cruciale per la gestione dei log è **Filebeat**.

Questo agente leggero, progettato per inoltrare e centralizzare log e file di testo, sostituisce tradizionali strumenti di monitoraggio pesanti o script artigianali, offrendo una gestione robusta degli errori e della rotazione dei file [77].
Il suo **funzionamento** interno si basa su un'**architettura** di tipo *producer-consumer*, governata da due componenti logici principali:

- **Input**: è il componente responsabile della gestione dei localizzatori. L'Input scansiona i percorsi definiti nella configurazione (es. `/var/log/*.log`) per rilevare nuovi file o modifiche a file esistenti;
- **Harvester**: avviato per ogni file rilevato dall'Input, questo componente apre il documento, ne legge il contenuto e invia i dati al buffer di uscita (*Spooler*). Un aspetto tecnico fondamentale è che l'Harvester, finché attivo, mantiene ogni fd aperto, mantenendo il file allocato su disco e garantendo la persistenza dei log in lettura.

Inoltre, per garantire che nessun dato venga perso o duplicato, Filebeat mantiene un file di stato locale chiamato **Registry** [77]. Il Registry mappa il percorso di ogni suo file di input al corrispondente inode e all'offset, permettendo a Filebeat di gestire correttamente eventuali operazioni terze di renaming (dovute ad esempio a log rotation).

Infine, per semplificare l'ingestione di formati comuni, Filebeat introduce il concetto di **Moduli**, ossia pacchetti preconfigurati per la localizzazione, il preparsing, la selezione delle pipelines interne a Elasticsearch e la visualizzazione in Kibana di logs comuni (come quelli di Apache, Nginx o PostgreSQL) [77].

Nel contesto di sviluppo presente, l'impiego di data shippers distribuibili ha consentito di separare (facoltativamente) i pesanti servizi di ingestion, indexing e visualization dall'altrettanto dispendiosa cross-compilazione multi-process e multi-threaded condotta da Rootless V^2CI, permettendo per quest'ultima l'esecuzione diretta su server distribuiti e la sorveglianza centralizzata su un host separato.

Diagramma riassuntivo dell'infrastruttura ELK

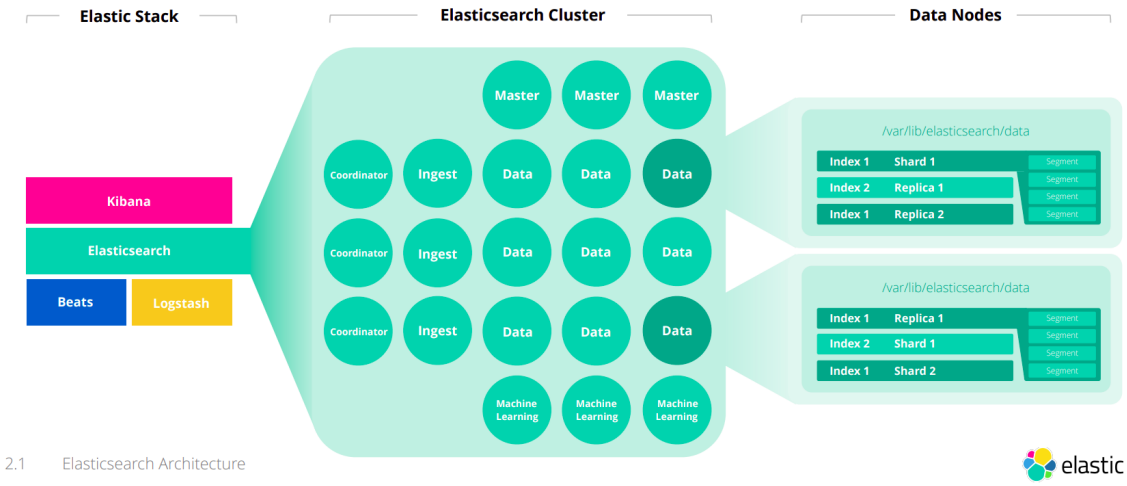

Figura 4.10: Diagramma riassuntivo della composizione dell'ELK stack

4.2.2 Scelta di architettura containerizzata distribuita e design planning del cluster

Per l'allestimento di un Elastic Stack completo dedicato alla monitorabilità di Rootless V^2CI, si è deciso di adottare un'architettura distribuita su più container Docker - uno per ogni servizio e nodo del cluster Elasticsearch - e orchestrata tramite *Docker Compose*.

Tale scelta, sconsigliata per ambienti di produzione aziendale e suggerita per scopi accademici di sviluppo e test locale su singoli server [67], ha aderito perfettamente all'obbiettivo di permettere all'utente utilizzatore sia di istanziare agevolmente il motore di cross-compilazione e lo stack ELK sul medesimo host, che di monitorare i log generati da più istanze distribuite di Rootless V^2CI da un unico server supervisore.

Nonostante ciò, la scelta, quasi obbligata, di un'infrastruttura ELK compattizzata per un'esecuzione integrale su singolo host ha imposto non poche limitazioni progettuali.

Trade-off del deploy containerizzato su singolo host

Come anticipato nella precedente sottosezione 4.2.1, il deploy dei nodi Elasticsearch - così come dei container associati ai servizi di Logstash e Kibana - su un unico host, ha di certo mantenuto limitati i prerequisiti di funzionamento dell'intera infrastruttura, ma ha anche escluso a priori la possibilità - o per lo meno il senso - di adottare tecniche di replicamento degli shard, compromettendo così l'HA e la tolleranza ai guasti dei container aventi il ruolo di `data_content`.
Questo compromesso però, considerando il contesto di sviluppo e di probabile applicazione locale unificata a Rootless V^2CI, si è considerato del tutto accettabile.

Allo stesso tempo, la più verosimile eventualità di eseguire Rootless V^2CI con "plugin" ELK su un unico server, ha però costretto a una progettazione del cluster Elastic estremamente economica che ha propagato l'assenza di HA anche a livello di servizi.
Nello specifico, una prima progettazione over-stimata ha previsto l'adozione per Elasticsearch di un totale di 7 nodi, di cui:

- 3 necessariamente con ruolo di `master_eligible`: necessario per tollerare guasti del cluster e garantire l'elezione automatica del nodo master in caso di crash di uno di essi, escludendo quindi scenari di *split-brain* [78];
- 2 con ruolo di `data_hot`: per consentire ai log ingeriti più recentemente di godere di una permanenza sicura (ossia caratterizzata da replicas e HA) su nodi ottimizzati per l'indicizzazione, prima di ruotare su nodi efficienti per lo stoccaggio [67];
- 2 con ruolo di `dat_warm`: per ospitare i log meno recenti e loro repliche, con minori requisiti di performance, ma comunque ancora soggetti a query frequenti [67].

Tale architettura, era sì tecnicamente valida, aderente alle best-practices di Elastic e altamente disponibile grazie alla ridondanza dei nodi, ma, in esecuzione combinata ai servizi di Logstash, Kibana e Rootless V^2CI, ha avuto l'immediato effetto collaterale di saturare tutte le risorse computazionali al punto da causare trashing a livello di sistema operativo host.
Questo fallimento dei test su hardware apprezzabile ma comunque limitato (processore 11th Gen Intel® Core™ i5-1155G7 × 8 e 16 GB di RAM), ha inevitabilmente indotto a una revisione dell'architettura che permettesse un drastico abbattimento dei servizi in esecuzione, attraverso un pedante calcolo delle risorse strettamente

necessarie per il funzionamento di ogni container.
Per questo motivo però, come verrà esposto nella prossima sottosezione, la progettazione del cluster Elastic per l'ingestion dei log di Rootless V^2CI ha escluso il setup di nodi ridondanti, rinunciando quindi a qualsiasi forma di HA, non solo a livello di host ma anche di servizi containerizzati.

Design planning

La fase di progettazione concettuale e la pianificazione dei volumi del cluster Elastic ha seguito un approccio top-down: partendo da stime verosimili di produzione di log da parte di Rootless V^2CI, si è proceduto a calcolare - seguendo le indicazioni pratiche suggerite dall'architetto Elasticsearch Dave Moore [79] - le risorse minime necessarie per ogni servizio containerizzato.
L'unica decisione progettuale impostata a priori, come anticipato, è stata quella di economizzare al massimo il numero di `data_content`. Pertento, a seguito di una valutazione del contesto e dello scope pratico di questa integrazione ELK, in aggiunta a previsioni e stime derivanti dai test svolti e falliti sulla precedente architettura over-ingegnerizzata, si è deciso di dotare il cluster Elasticsearch di una struttura minimale composta da un totale di 4 nodi:

- 3 con ruolo di `master_eligible`, per garantire comunque il corretto funzionamento del cluster e un minimo di fault tolerance;
- 1 solo con ruolo di `data_content`, in *hot tier*, in modo da minimizzare le ridondanze e al coltempo permettere indicizzazione ottimizzata, per 30 giorni di *retention*, valore che si è pensato fosse sufficiente all'utente per stimare in modo completo una media degli esiti delle operazioni di Rootless V^2CI, eseguite per la compilazione dei binari più recenti. In particolare questa scelta è figlia anche dell'assunzione che esecuzioni legate alla build di binari presenti in `target_dir/yearly` siano poco rilevanti a scopo di monitorabilità della fornitura di versioni aggiornate e corrette.

L'approccio migliore per illustrare le successive operazioni meticolose di size planning è quello di mostrare le seguenti tabelle riassuntive delle stime dei volumi iniziali, delle definizioni di eventuali requisiti di usabilità, dei calcoli del dimensionamento del cluster e delle risorse finali da assegnare ad ogni container Docker.

Attributo	Calcolo	valore per hot data node	nota
log/giorno	$3750 \cdot 2 \cdot 15$	112500	Basato sull'assunzione di un massimo di 1/4 delle righe totali prodotte (circa 15000) per ogni progetto e per ogni ciclo (i log sono multilinea). Considerando un caso pessimo di 2 cicli al giorno per progetto, comprensivi di chroot setup (primo avvio o recovery) e build, e 15 progetti.
Dimensione log (KB)	–	1	Stima molto rilassata (solo per log multilinea molto grandi)
Dati grezzi al giorno (GB)	$\text{Logs/Day} \cdot \text{Log Size}/1024^2$ $112500 \cdot 1/1024^2$	0.1125	Calcolato secondo le linee guida [79]
Fattore JSON	–	1.5	–
Fattore Indice	–	1.1	–
Fattore Compressione	–	0.3	–
Fattore espansione netta	$\text{JSON} \cdot \text{Indexing} \cdot \text{Compression}$ $1.5 \cdot 1.1 \cdot 0.3$	0.495	Calcolato secondo le linee guida [79]
Memoria per data node (GB)	–	1	–
Rapporto Memoria:dati	–	0.03333333333	Ricavato dal rapporto memoria:dati di 1 : 30 delle linee guida [79]
Retention (giorni)	–	30	In questa architettura leggera si assume una retention massima di 30 giorni
Repliche	–	0	Nessuna replica (i log rimarranno disponibili sugli host)

Tabella 4.2: Assunzioni e Stime dei Dati

	Calcolo	Volume dati (GB)	Volume storage (GB)	Nodi dati
hot tier	$D_{tot} = \text{Raw} \cdot \text{Days} \cdot \text{Exp} \cdot (\text{Repl} + 1)$ $D_{tot} = 0.1125 \cdot 30 \cdot 0.495 \cdot 1$ $S_{tot} = D_{tot} \cdot (1 + 0.15 + 0.05)$ $S_{tot} = 1.670625 \cdot 1.2$ $N = \lceil S_{tot}/\text{Mem}/\text{Ratio} \rceil + 1$ $N = 2.00475 \cdot 0.0333/1 + 1$	1.670625	2.00475	1.066825
totale	–	1.670625	2.00475	1.066825

Tabella 4.3: Dati, Storage e Nodi

	Ruoli	Risorse computazionali (vCPU)	Memoria principale (GB RAM)	Storage (GB)
Servizio 1	Master dedicato	1	1	2
Servizio 2	Master dedicato	1	1	2
Servizio 3	Master dedicato	1	1	2
Servizio 4	Dati Hot	2	1	2.879174185
Servizio 5	Logstash	2	1	1
Servizio 6	Kibana	2	1	1
Totale	–	**≈4/5cores con hyperthreading**	**6**	**10.87917419**

Tabella 4.4: Ruoli dei Servizi e Risorse

Come è possibile evincere da queste tre tabelle fondamentali per la progettazione del cluster ELK, i calcoli cardine hanno validato l'iniziale preclusione progettuale di servirsi di un solo `data_content`, per l'indicizzazione in hot tier.
Infatti, partendo da assunzioni sul volume dei dati prodotti da Rootless V^2CI in un contesto di impiego intensivo, per 30 giorni di retention in hot tier si è dimostrato necessario esattamente un solo nodo, validando così l'architettura scelta.
Infine, come è possibile osservare dall'ultima tabella, sia i calcoli esatti che le assunzioni su risorse computazionali e di disco (valutate empiricamente grazie al fallimento dei test precedentemente svolti sull'architettura over-dimensionata), hanno mostrato la reale possibilità di eseguire questa ambiziosa integrazione ELK anche su host

limitati.

Al fine di illustrare l'architettura finale di questo cluster è stato realizzato il seguente diagramma.

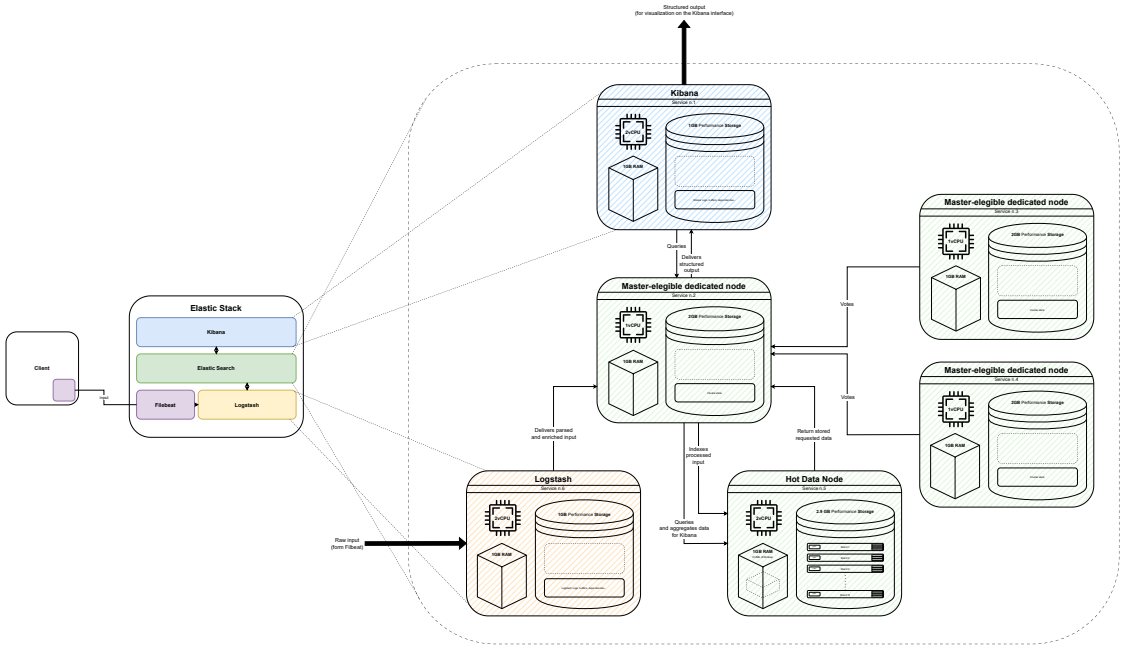

Figura 4.11: Diagramma architetturale del cluster Elastic impiegato per il monitoring di Rootless V^2CI

4.2.3 Aspetti implementativi del setup, delle configurazioni e della gestione delle risorse per i servizi ELK containerizzati

Come anticipato in precedenza, l'integrazione ELK per Rootless V^2CI si basa sull'orchestrazione tramite Docker Compose di 5 container, generati a partire dalle immagini Docker ufficiali di Elastic [80], regolati da file di configurazione `.yml` e setup-ati - con linking ai propri volumi e agli altri container in esecuzione - per mezzo di script `.sh`.
Tale orchestrazione è stata inoltre progettata per essere incapsulata in un unico servizio `systemd` - `elastic-stack.service` - che prevede anche fasi di preparazione preliminare e pulizia conclusiva all'esecuzione dei servizi ELK, garantendo così all'utente che intenda utilizzare ELK associato a Rootless V^2CI massima comodità e resilienza.

Orchestrazione con Docker Compose

Il file `docker-compose.yml` contiene la logica per un avvio ordinato e funzionale dei servizi ELK [81].
Nello specifico, per il corretto raggiungimento dello stato `healthy` del cluster, sarebbero state necessarie non solo attente configurazioni ai singoli servizi - di cui verrà discusso nelle prossime sezioni - ma, primariamente, anche caute sequenze di operazioni per l'avvio dei container stessi. In particolare, qualsiasi cluster Elastic [67] necessita di start-are in ordine:

1. Tutti i nodi `master_eligible`, per permettere l'elezione del nodo master;
2. I nodi `data_content`, che si registrano al cluster una volta che il master è stato eletto;
3. Infine, i servizi di Logstash e Kibana, che si connettono al cluster Elasticsearch solo dopo che quest'ultimo è operativo.

Per ogni container, durante la fase di avvio, Docker Compose ha anche l'essenziale compito di montare i volumi di persistenza dati, definire variabili di ambiente - ereditate da un file `.env` dedicato e trasferite ai volumi dei servizi - e specificare un `healthcheck` che, a intervalli regolari e limitato da un numero massimo di tentativi e da un timeout, verifichi lo stato di salute del container, in modo che, una volta completate con successo le operazioni di sua competenza, possa "sbloccare" l'avvio dei container dipendenti [81].

Per il caso d'uso specifico di monitoring di Rootless V^2CI, è stato quindi composto il file `docker-compose.yml` che, a seguito della redazione dei file di configurazione delle tecnologie ELK, dell'archivio `.env` delle variabili di ambiente, e degli script bash per il setup dei nodi Elastic, ha assunto la seguente forma:

```
services:
  # Servizio effimero di setup per la generazione dei certificati SSL
  # e la configurazione iniziale delle policy di retention (ILM)
  setup:
    image: elasticsearch:${STACK_VERSION}
    volumes:
      - ./certs:/usr/share/elasticsearch/config/certs
      - ./scripts/cluster_setup.sh:/usr/share/
          elasticsearch/cluster_setup.sh:ro
```

```
    # ... (variabili d'ambiente per credenziali) ...
    command: [
      "bash", "-c", "/usr/share/elasticsearch/cluster_setup.sh \
      && tail -f /dev/null"
    ]
    healthcheck:
      # Prima di considerarsi sano attende la
      # generazione dei certificati SSL
      test:
      [
      "CMD-SHELL", "[
        -f config/certs/v2ci-es-master-1/v2ci-es-master-1.crt
        ]"
      ]
      interval: 1s
      timeout: 5s
      retries: 120

  # Nodo Master 1 (Rappresentativo per i 3 nodi master del cluster)
  v2ci_es_master_1:
    depends_on:
      setup:
        condition: service_healthy
    image: elasticsearch:${STACK_VERSION}
    volumes:
      # Montaggio certificati generati dal setup
      - ./certs:/usr/share/elasticsearch/config/certs
      # Configurazione specifica del nodo master
      - ./elasticsearch/master/config/elasticsearch.yml:
          /usr/share/elasticsearch/config/elasticsearch.yml:ro
      # Volume dati persistente
      - v2ci_es-master_1_data:/usr/share/elasticsearch/data
    ports:
      - ${ES_PORT}:${ES_PORT}
    environment:
      - NODE_NAME=v2ci-es-master-1
      - CLUSTER_NAME=${CLUSTER_NAME}
      # Rispetto della progettazione delle risorse di memoria
```

```
      # tramite variabili d'ambiente
      - ES_JAVA_OPTS=-Xms${MASTER_HEAP} -Xmx${MASTER_HEAP}
      # ...
    healthcheck:
      # Verifica connettivita' sicura al nodo master
      test:
        [
          "CMD-SHELL",
          "curl -s --cacert config/certs/ca/ca.crt \
            https://$$NODE_NAME:$$ES_PORT | grep -q \
            'missing authentication credentials'",
        ]
        # ...

  # ... (v2ci-es-master-2, v2ci-es-master-3, v2ci-es-hot-1
  # v2ci-logstash e v2ci-kibana, con volumi e healthcheck
  # omessi per brevita' e per configurazione simile) ...

volumes:
  # Esempio di definizione volume con driver locale bind-mount
  v2ci_es_master_1_data:
    driver: local
    driver_opts:
      type: "none"
      device: "/var/lib/elastic-stack/es-master-1-data"
      o: "bind"
  # ... (altri volumi omessi) ...
```

Come anticipato e constatabile dal codice YAML soprastante, i tre aspetti cardine - non ancora discussi - attorno ai quali ruota tutta l'esecuzione di `docker compose up -d` - che avrebbe permesso di avviare l'intero stack -, riguardano la gestione dei volumi persistenti, il setup iniziale, regolato dal servizio temporaneo `setup` per mezzo di `cluster_setup.sh`, e i file di configurazione `.yml` dei singoli servizi Elastic.

Persistenza dei volumi

Uno dei prerequisiti fondamentali per l'avvio del processo di orchestrazione dei container Docker è la disponibilità di volumi persistenti per ogni servizio che ne

necessiti.
Dal momento che lo stack ELK per il monitoring di Rootless V^2CI, come anticipato, è stato pensato per essere avviato come servizio systemd, si è stabilito che il momento più aderente alla necessità di una fase di setup dei volumi fosse l'`ExecStartPre` dello stesso `elastic-stack.service`.
In corrispondenza di questa direttiva è infatti invocato uno script `LVM_setup.sh` che, con lo scopo accessorio di garantire persistenza, coesione e flessibilità nell'organizzazione dei dati, si serve di *LVM* (*Logical Volume Manager*) [82], al fine di dimensionare preliminarmente i volumi da associare ai servizi Elastic, seguendo la progettazione svolta nella sottosezione 4.2.2.

Nello specifico, dal momento che LVM necessita nativamente di un device dedicato e formattato su cui operare per creare e amministrare volumi logici, `LVM_setup.sh` procede come mostrato di seguito:

1. per prima cosa si tenta di rilevare il **block device** secondario (e.g. `/dev/sdb`) dichiarato dalla variabile di ambiente `ESDATA_PV_DEVICE`; in caso di sua mancanza, lo script crea un'immagine disco `/var/lib/elasticsearch-disk.img` - di capienza pari alla somma dello storage richiesto da ogni servizio Elastic - e la collega a un ***loop-back device*** in modo che anche gli host dotati di un singolo disco possano comunque imitare un data driver dedicato [82];

2. successivamente viene aggiornata la **cache del *device mapper***, in modo che eventuali modifiche precedenti ai block device vengano riconosciute;

3. si inizializza il ***physical volume*** - solo quando necessario, ossia se non è già stato svolto un LVM setup sulla stessa `.img` in precedenza;

4. viene assemblato il ***volume group*** `esdata-vg`, garantendo riesecuzioni idempotenti;

5. per ogni componente dell'Elastic Stack, lo script predispone poi un ***logical volume*** dimensionato secondo la pianificazione delle risorse precedente, lo formatta con **XFS** (il filesystem raccomandato da Elastic per carichi di lavoro ad alta concorrenza [67]) e aggiunge **un'entry persistente in** `/etc/fstab` con quote utente abilitate;

6. dopo il mount, vengono applicati **limiti di quota** per-servizio per l'UID 1000, impedendo a Elasticsearch, Logstash o Kibana di sottrarre risorse agli altri

servizi ELK, e infine viene impostata correttamente l'**ownership** in modo che i servizi containerizzati possano scrivere senza escalation di privilegi.

Queste operazioni, eseguite in sequenza, garantiscono che ogni servizio Elastic disponga di un volume logico dedicato, persistente e dimensionato secondo le necessità progettuali.

```
#!/bin/bash
# Definizione variabili (device paths, nome volume group, mount root)
DATA_PV_DEVICE=${ESDATA_PV_DEVICE:-/dev/sdb}
DISK_IMAGE=${ESDATA_LOOP_IMAGE:-/var/lib/elasticsearch-disk.img}
DATA_VG_NAME=esdata-vg
MOUNT_ROOT=/var/lib/elastic-stack
# ...

# [1] Se il device specificato non esiste, crea e
# collega un loopback device
if [ ! -b "${DATA_PV_DEVICE}" ]; then
  # ... (creazione file .img e associazione tramite losetup) ...
  DATA_PV_DEVICE=${LOOP_DEVICE}
fi

# [2] Aggiornamento cache LVM
sudo pvscan --cache "${DATA_PV_DEVICE}" >/dev/null 2>&1 || true

# Definizione dei Logical Volumes in base alla pianificazione
declare -A LV_SPECS=(
  [es-master-1-data]=2G
  [es-master-2-data]=2G
  [es-master-3-data]=2G
  [es-hot-data]=3G
  [logstash-data]=1G
  [kibana-data]=1G
)

# [3] & [4] Inizializzazione Physical Volume e Volume Group
if ! sudo vgs ...; then
  # ... (pvcreate se necessario) ...
  sudo vgcreate "${DATA_VG_NAME}" "${DATA_PV_DEVICE}"
```

```
fi
sudo vgchange -ay "${DATA_VG_NAME}" >/dev/null 2>&1 || true

# [5] Creazione Logical Volumes, formattazione XFS e aggiornamento fstab
for lv_name in "${!LV_SPECS[@]}"; do
  # ... (setup variabili size e path) ...

  # Creazione LV se non esiste
  sudo lvcreate -L "${lv_size}" -n "${lv_name}" "${DATA_VG_NAME}"

  # Formattazione XFS se necessaria
  sudo mkfs.xfs -f "${lv_path}" >/dev/null

  # Aggiunta entry persistente in fstab con quote abilitate
  fstab_entry="${lv_path} ${mount_point} xfs defaults,uquota 0 2"
  echo "${fstab_entry}" | sudo tee -a /etc/fstab >/dev/null
done

sudo mount -a

# [6] Configurazione quote XFS e ownership per UID 1000
for lv_name in "${!LV_SPECS[@]}"; do
  # ...
  sudo xfs_quota -x -c "limit -u bsoft=${lv_size}" \
    "bhard=${lv_size} 1000" \
    "${mount_point}"
done

sudo chown -R 1000:0 "${MOUNT_ROOT}"
echo "Volume setup completed."
```

Chiaramente, la scelta di collegare l'immagine disco contenente i volumi logici a un dispositivo a blocchi virtuale che risiede in memoria, causa inevitabilmente la perdita del collegamento device↔immagine ogni qualvolta che l'host si riavvia. Senza ulteriori salvaguardie quindi, il sistema potrebbe fallire il mount dei logical volumes al boot successivo, in quanto, sebbene l'immagine `/var/lib/elasticsearch-disk.img` sia persistente, il loop-back che ne garantiva l'accessibilità come block device, non esiste più.

Per evitare questo problema, è stata aggiunta la direttiva `ExecStopPost` all'unit `elastic-stack.service`, la quale si occupa di invocare `LVM_teardown.sh`.
Questo script, fratello del precedente, assicura arresti e riavvii del servizio idempotenti e puliti, eseguendo le seguenti operazioni:

1. **smonta i logical volumes** sotto `/var/lib/elastic-stack`, rimuovendo i mount points corrispondenti e le entries in `/etc/fstab`;
2. **disattiva il volume group** `esdata-vg`, ossia lo scollega dal block device virtuale;
3. **scollega il loop-back** associato all'immagine disco.

Eseguiti in questo ordine, questi passaggi garantiscono uno smantellamento bottom-up che preserva l'integrità dei dati e la coerenza del sistema.

```
#!/bin/bash
# ... (definizione variabili e funzione di logging) ...

# [1] Smontaggio dei volumi e rimozione persistenza in fstab
for lv in "${LV_NAMES[@]}"; do
  mount_point="${MOUNT_ROOT}/${lv}"
  lv_path="/dev/${DATA_VG_NAME}/${lv}"

  # Unmount se montato
  if mountpoint -q "${mount_point}"; then
    umount "${mount_point}" || true
  fi

  # Rimozione entry da /etc/fstab per evitare errori al boot
  if [[ -f /etc/fstab ]]; then
    sed -i "\|^${lv_path}[[:space:]]\+${mount_point}[[:space:]]|d" \
      /etc/fstab
  fi
  rm -rf "${mount_point}"
done

# [2] Disattivazione del Volume Group
if vgdisplay "${DATA_VG_NAME}" >/dev/null 2>&1; then
  vgchange -an "${DATA_VG_NAME}" >/dev/null 2>&1 || true
```

```
fi

# [3] Distacco del loop device associato all'immagine disco
if [[ -f "${DISK_IMAGE}" ]]; then
  loopdev=$(losetup -j "${DISK_IMAGE}" | cut -d: -f1 | head -n1)
  if [[ -n "${loopdev}" ]]; then
    losetup -d "${loopdev}" >/dev/null 2>&1 || true
  fi
fi
```

I file di configurazione

L'altro prerequisito essenziale al corretto avvio di tutti i servizi ELK, chiaramente, è la presenza di file di configurazione `.yml` specifici per ogni tecnologia impiegata.
Questi file, montati come volumi di sola lettura all'interno dei container Docker, contengono le direttive necessarie per l'abilitazione della comunicazione sicura tramite SSL, l'autenticazione mutua tra client e server Elastic, la definizione dei ruoli dei nodi del cluster, le pipeline di ingestion di Logstash e le impostazioni di Kibana.

Configurazione di Elasticsearch e Kibana In un contesto containerizzato, la redazione dei file di configurazione `elasticsearch.yml` - per ognuno dei nodi del cluster - e `kibana.yml`, ha richiesto minimo sforzo.
Astraendo infatti dal sistema host sottostante, è stato possibile concentrarsi principalmente sulla mera configurazione delle impostazioni di sicurezza e connessione.

Pipeline di Logstash Il file `logstash/pipeline/logstash.conf` si può considerare il cuore pulsante della fase di ingestion e parsing dei log di Rootless V^2CI.
Esso definisce infatti le *pipeline* che Logstash deve eseguire per processare i log in arrivo, specificando gli *input*, i *filter* e gli *output*.
Trascurando per il momento la sezione che regola la comunicazione con eventuali data shippers - di cui verrà discusso invece al paragrafo 4.2.4 -, prima di illustrare gli altri componenti della pipeline Logstash, si riporta un esempio di log monolinea generato da Rootless V^2CI:

```
[2025-10-30 23:52:32] [INFO] source: { client: { ip: 151.82.40.78,
os: GNU/Linux, arch: x86_64, agent: HP Laptop 15s-fq4xxx },
```

```
location: { file: /home/francesco/Scrivania/terzo_anno/Tesi
/rootless_V2CI/src/build_thread.c, line: 167 } }, project: sshlirp,
thread_arch: arm64, message:
Starting build process for architecture arm64 for project sshlirp...
```

A partire da questo formato di log, strutturato secondo quanto documentato nella sottosezione 4.1.7, è stato necessario comporre una pipeline Logstash flessibile che fosse in grado, non solo di ingerire e tipizzare gli attributi secondo valori standard, ma anche di arricchire i documenti stoccati in Elasticsearch con informazioni aggiuntive come la derivazione geografica del log in base all'indirizzo IP del client e flag/tag personalizzati per una più agevole categorizzazione.
Il risultato di questo obbiettivo, figlio di un processo di debugging e test non trascurabile, è converso nella seguente pipeline `logstash.conf`:

```
# ... input ...

# FILTER
filter {
  grok {
    match => {
      # Impiego Grok per il parsing del log
      "message" => [
        "^\[%{TIMESTAMP_ISO8601:[log][timestamp]}\]
        \[(?<[log][level]>%{LOGLEVEL}|INTERRUPT)\]
        source: { client: { ip: %{IPORHOST:[source][client][ip]}
        ...}, ... }, ..., message: %{GREEDYDATA:[log][message]}"
      ]
    }
    # Aggiungo un tag in caso di fallimento del parsing con Grok
    tag_on_failure => [ "_grokparsefailure" ]
  }

  # Date filter per convertire [log][timestamp] in @timestamp
  if [event][timezone] and [event][timezone] != "" {
    date {
      match => ["[log][timestamp]", "YYYY-MM-dd HH:mm:ss"]
      target => "@timestamp"
      timezone => "%{[event][timezone]}"
```

```
    }
  } else {
    date {
      match => ["[log][timestamp]", "YYYY-MM-dd HH:mm:ss"]
      target => "@timestamp"
      timezone => "${LOG_TIMEZONE:UTC}"
    }
  }

  # GeoIP enrichment solo se l'IP e' valido
  # in qual caso lo inserisco in [source][ip] e faccio il lookup
  if [source][client][ip] and
    [source][client][ip] =~
    /^(?:\d{1,3}\.){3}\d{1,3}$|^[0-9A-Fa-f:]+$/
  {
    mutate {
      copy => { "[source][client][ip]" => "[source][ip]" }
    }
    geoip {
      source => "[source][ip]"
      target => "source"
      tag_on_failure => ["_geoip_lookup_failure"]
    }
  }

  # ... normalizzazione di altri campi ...
  }
}

# OUTPUT
output {
  elasticsearch {
    # Connessione sicura al nodo data_hot del cluster
    hosts => ["https://v2ci-es-hot-1:${ES_PORT:9200}"]
    user => "elastic"
    password => "${ELASTIC_PASSWORD}"
    ssl_enabled => true
    ssl_certificate_authorities =>
```

```
      "/usr/share/logstash/config/certs/ca/ca.crt"

    # Specifico che i log vadano nel data stream di cui
    # configurero' il template in ILM_setup.sh
    data_stream => true
    data_stream_type => "logs"
    data_stream_dataset => "compiler"
    data_stream_namespace => "default"
  }
}
```

Il setup iniziale

Il file di script bash `cluster_setup.sh`, eseguito come primo servizio da Docker Compose e basato sull'assunzione di volumi persistenti accessibili e file di configurazione corretti, ha il compito di preparare l'ambiente del cluster Elastic, integrando dentro di sé 3 principali funzionalità:

- La generazione dei **certificati SSL** per la comunicazione sicura tra i nodi del cluster e per l'autenticazione mutua tra client e server Elastic;
- L'impostazione di **regole globali** ereditate dalla progettazione concettuale, quali l'esclusione di repliche per gli shard - per evitare che il cluster tenti di allocare copie ridondanti di indici di sistema, per comportamento predefinito al suo avvio - e la definizione delle policy di *Index Lifecycle Management* (ILM) per la retention dei log, entrambe svolte tramite le API RESTful di Elasticsearch;
- La creazione - sempre per mezzo delle API - degli **utenti di sistema predefiniti**, con le relative password, per l'accesso autenticato ai servizi Elastic.

Il seguente estratto di codice bash riassume l'implementazione di tali funzionalità all'interno di `cluster_setup.sh`:

```
#!/usr/bin/env bash
# ... (variabili e funzioni helper per logging
# e check variabili da .env) ...

# Generazione della CA se non presente, tramite certutil e in
```

```
# formato .pem, al fine di ottimizzare una gestione
# trasparente dei .crt e .key
if [[ ! -f "${CERT_DIR}/ca.zip" ]]; then
  bin/elasticsearch-certutil ca --silent --pem -out "${CERT_DIR}/ca.zip"
  unzip "${CERT_DIR}/ca.zip" -d "${CERT_DIR}"
fi

# Generazione dei certificati per le istanze se non presenti
if [[ ! -f "${CERT_DIR}/certs.zip" ]]; then
  bin/elasticsearch-certutil cert --silent --pem \
    -out "${CERT_DIR}/certs.zip" --in "${CERT_DIR}/instances.yml" ...
  unzip "${CERT_DIR}/certs.zip" -d "${CERT_DIR}"
fi

# ... (impostazione permessi file e attesa disponibilita'
# Elasticsearch tramite curl --ca-cert) ...

log "Setting cluster default index.number_of_replicas=0"
curl --cacert "${CA_CERT}" -u "elastic:${ELASTIC_PASSWORD}" \
  -H "Content-Type: application/json" \
  -X PUT "${ES_HOST}/_template/default" \
  -d '{"index_patterns": ["*"], "settings": {"number_of_replicas": 0}}'

log "Setting kibana_system password"
until curl -s -X POST --cacert "${CA_CERT}" \
  -u "elastic:${ELASTIC_PASSWORD}" \
  -H "Content-Type: application/json" \
  "${ES_HOST}/_security/user/kibana_system/_password" \
  -d "{\"password\":\"${KIBANA_PASSWORD}\"}" | grep -q '^{}'; do
  sleep 10
done

log "Set ILM policies with ILM_setup.sh script"
bash /usr/share/elasticsearch/ILM_setup.sh
```

Lo script `ILM_setup.sh`, a cui si appoggia in modo evidente l'esecuzione dell'inizializzatore bash sopra riportato, oltre a creare una vera e propria policy di ciclo di vita degli indici con una configurazione completa di *retention* e *rollover*, si occupa

anche di comporre l'essenziale *index template* per i log di Rootless V^2CI, il quale ne definisce il mapping e la struttura all'interno di Elasticsearch.
Per questo specifico obbiettivo si è deciso di optare per una configurazione che prevedesse l'uso di *data stream* - a cui si ha accennato nella sottosezione 4.2.1 - e *dynamic mapping*. La prima scelta ha permesso di semplificare la gestione degli indici, delegando a Elasticsearch la creazione automatica di nuovi indici fisici e la rotazione dei precedenti in *backing indexes* [67], mentre la seconda ha evitato di dover definire a priori il mapping di ogni singolo campo dei log, lasciando che fosse Elasticsearch a dedurlo automaticamente in fase di ingestion.

L'applicazione tramite API di queste politiche al cluster Elasticsearch ha quindi conferito a `ILM_setup.sh` la seguente struttura:

```
#!/bin/bash
# ... (setup variabili d'ambiente e check credenziali) ...

# Definizione della policy JSON: rollover giornaliero o a 1GB,
# delete dopo 30 giorni
cat >/tmp/compiler-logs-ilm.json <<'EOF'
{
  "policy": {
    "phases": {
      "hot": {
        "actions": {
          "rollover": {
            "max_age": "1d",
            "max_primary_shard_size": "1gb"
          }
        }
      },
      "delete": {
        "min_age": "30d",
        "actions": { "delete": {} }
      }
    }
  }
}
```

```
EOF
# Applicazione della policy tramite API PUT _ilm/policy
curl -s -o /dev/null -u "elastic:${ELASTIC_PASSWORD}" \
  --cacert "${CA_CERT}" \
  -H "Content-Type: application/json" -X PUT \
  "${ES_URL}/_ilm/policy/compiler-logs-ilm" \
  -d @/tmp/compiler-logs-ilm.json

# Definizione del template JSON: pattern per data stream,
#  mapping dinamico e associazione alla policy ILM creata sopra
cat >/tmp/logs-compiler-default-template.json <<'EOF'
{
  "index_patterns": [ "logs-compiler-*", ".ds-logs-compiler-*" ],
  "data_stream": {},
  "template": {
    "settings": {
      "index.lifecycle.name": "compiler-logs-ilm",
      "index.number_of_shards": 1,
      "index.number_of_replicas": 0,
      "index.codec": "best_compression"
    },
    "mappings": { "dynamic": true }
  }
}
EOF
# Applicazione del template tramite API PUT _index_template
curl -s -o /dev/null -u "elastic:${ELASTIC_PASSWORD}" \
  --cacert "${CA_CERT}" \
  -H "Content-Type: application/json" -X PUT \
  "${ES_URL}/_index_template/logs-compiler-default-template" \
  -d @/tmp/logs-compiler-default-template.json
```

4.2.4 Scelta degli agenti di monitoring e configurazioni per architettura sia distribuita che centralizzata

Un passo successivo - all'implementazione dell'`elastic-stack.service` - nell'integrazione del sistema di Continuous Integration con l'Elastic Stack, è stata la selezione e la configurazione degli agenti di monitoring incaricati di raccogliere e

inoltrare i log generati dal motore di build verso Logstash.
La scelta di escludere nella sezione *input* di Logstash il semplice meccanismo standard di *File* - introdotto nella sottosezione 4.2.1 - è figlia dell'ambizione già ampiamente anticipata di permettere all'utente utilizzatore di distribuire il sistema di monitoraggio in due scenari distinti: *All-in-One* (singolo host) e *Distributed* (Builder + Monitor Remoto).
Quindi la constatata essenzialità di data shippers, per l'implementazione di questo progetto, si è andata immediatamente a combinare con la natura dei documenti prodotti da Rootless V^2CI da ingerire, escludendo a priori forwarder di metriche e prediligendo **Filebeat** per la gestione computazionalmente impercettibile dei time-series dell'infrastruttura di CI.

Questa scelta, che ha permesso di potenziare enormemente la predisposizione allo *scaling-out* della combinazione Rootless V^2CI + ELK, ha richiesto prima di tutto la strutturazione della sezione di input di `logstash.conf` per l'accettazione di connessioni sicure provenienti da Filebeat, come mostrato di seguito:

```
input {
    beats {
    port => "${LOGSTASH_PORT:5044}"
    ssl_enabled => true
    ssl_certificate => "/usr/share/logstash/config/
      certs/v2ci-logstash/v2ci-logstash.crt"
    ssl_key => "/usr/share/logstash/config/certs/
      v2ci-logstash/v2ci-logstash.key"
    }
}
```

Successivamente è stato necessario preimpostare un file `.env` per le variabili di ambiente di Filebeat - essenziale per parametri quali l'indirizzo ip raggiungibile dell'host su cui è in esecuzione `elastic-stack.service`, password condivisa per l'autenticazione sicura e path assoluto dei certificati da copiare tramite `scp` -, redigere il file di configurazione `filebeat.docker.yml` e infine, in modo del tutto analogo a quanto fatto per i servizi ELK, comporre degli script di setup, avvio e stop che si occupassero di eseguire in modo ordinato tutte le operazioni necessarie al linking di Filebeat a Logstash e alla raccolta dei log di Rootless V^2CI.

Mentre risulta superfluo mostrare gli script `start.sh`, `stop.sh` e `install.sh` - in quanto di logica simile a quelli illustrati nella sezione precedente e di implementazione standard secondo la documentazione Elastic [67] -, è fondamentale specificare che una prima fase di categorizzazione e parsing avviene proprio nel file `filebeat.docker.yml`, il quale appunto si occupa di definire i 5 tipi di log prodotti da Rootless V^2CI, arricchirli con campi distintivi e inviarli al cluster Elasticsearch passando per Logstash.

```
filebeat.inputs:

# --- INPUT 1: Main process logs ---
- type: filestream
  id: main-process-logs
  paths:
    - ${V2CI_BUILD_DIR}/logs/main.log
  # ... (impostazioni standard: scan_frequency, recursive_glob) ...
  parsers:
    - multiline:
        type: pattern
        # Pattern per timestamp [YYYY-MM-DD HH:mm:ss]
        pattern: '^\[\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}\]'
        negate: true
        match: after
  fields:
    log_type: main_process
    service_name: main
    'event.timezone': ${FILEBEAT_TIMEZONE:UTC}
  fields_under_root: true

# --- INPUT 2..4: Project, Cronjob & Host Thread logs ---
# Configurazione analoga a INPUT 1, con paths specifici per
# ogni componente:
# - ${V2CI_BUILD_DIR}/*/logs/worker.log
# - ${V2CI_BUILD_DIR}/*/logs/binaries_rotation_cronjob.log
# - ${V2CI_BUILD_DIR}/*/logs/*-worker.log

# --- INPUT 5: Threads logs inside chroot ---
- type: filestream
```

```
  id: project-chroot-thread-logs
  paths:
    - ${V2CI_BUILD_DIR}/*-chroot/home/*/logs/worker.log
  # ... (parser multiline e impostazioni standard come sopra) ...
  fields:
    log_type: project_chroot_thread
    service_name: project-thread
    'event.timezone': ${FILEBEAT_TIMEZONE:UTC}
  fields_under_root: true

# ================== Outputs ===================
output.logstash:
  hosts: ["v2ci-logstash:${LOGSTASH_PORT:5044}"]
  ssl.enabled: true
  ssl.certificate_authorities: ["/usr/share/filebeat/certs/ca.crt"]

setup.kibana:
  host: "https://v2ci-kibana:${KIBANA_PORT:5601}"
  ssl.enabled: true
  ssl.certificate_authorities: ["/usr/share/filebeat/certs/ca.crt"]
```

4.2.5 Testing e risultati: visualizzazione su Kibana

Sebbene, come anticipato, lo sviluppo containerizzato di un'infrastruttura ELK dalle dimensioni limitate sia stato estremamente più agevole rispetto a una vera e propria installazione locale dei servizi distribuita su cluster multi-server, è stato comunque necessario affrontare considerevoli sfide di debugging e testing, soprattutto per quanto riguarda la corretta integrazione con Rootless V^2CI.

Una volta però "plug-ato" correttamente Filebeat sia alla produzione di log del motore di cross-compilazione, che all'interfaccia di parsing ed enrichment fornita da Logstash, è stato possibile interagire attraverso Kibana con un cluster, sì semplice e poco persistente, ma healthy e perfettamente funzionante anche per macchine dalle risorse limitate.

Un riflesso tangibile e apprezzabile - dal punto di vista di un eventuale utente utilizzatore - di questo corretto funzionamento dell'intera infrastruttura, più che nei

log di sitema dei servizi Elastic o nell'analisi dei consumi delle risorse, si trova nell'-effettiva visualizzazione dello stato del cluster e dei log, e nella tangibile possibilità di creare dashboard complete - attraverso l'uso di tool avanzati e integrati in Kibana quali *Canvas* e *Lens* - e ottimizzate per lo sfruttamento di tutte le funzionalità di strutturazione dei documenti offerte dalle configurazioni di Filebeat e Logstash, e dalla grezza formattazione iniziale operata dallo stesso Rootless V^2CI.
I seguenti sreenshots mostrano quindi due interfacce di overview dello stato del cluster Elastic, quella di discover dei logs e un esempio di dashboard creata per monitorare l'attività di build.

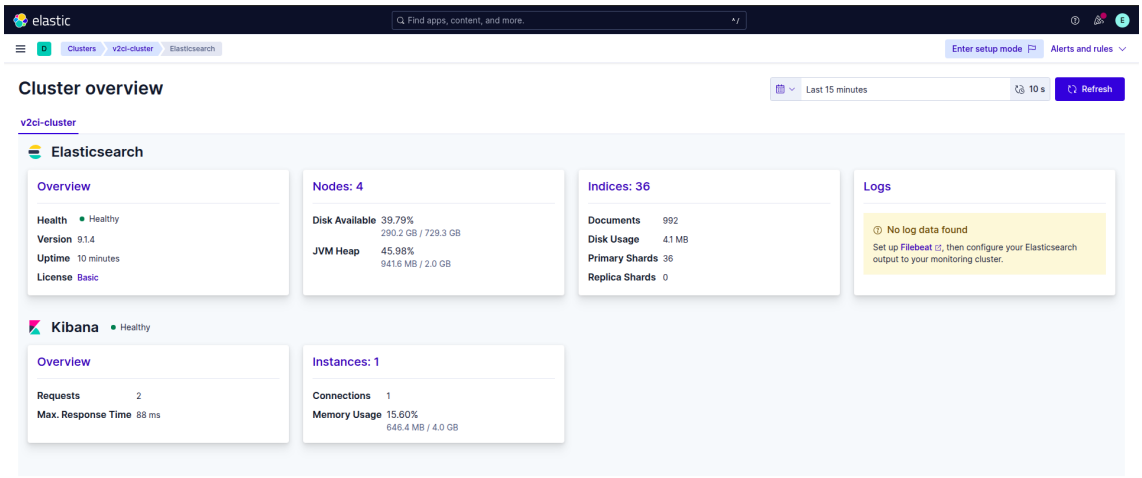

Figura 4.12: Panoramica dello stato del cluster Elasticsearch su Kibana > Stack Monitoring

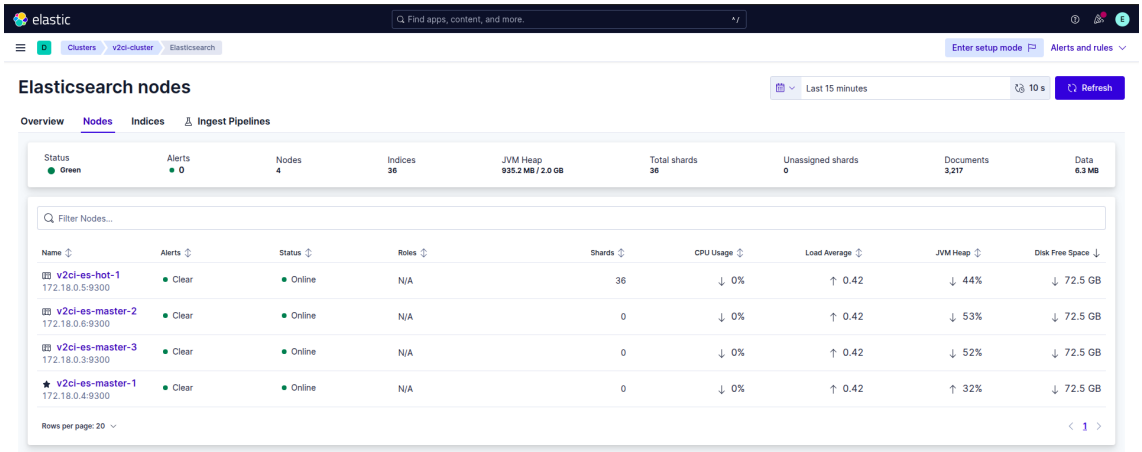

Figura 4.13: Dettagli sui nodi del cluster Elasticsearch visualizzabili tramite la funzionalità di stack monitoring su Kibana

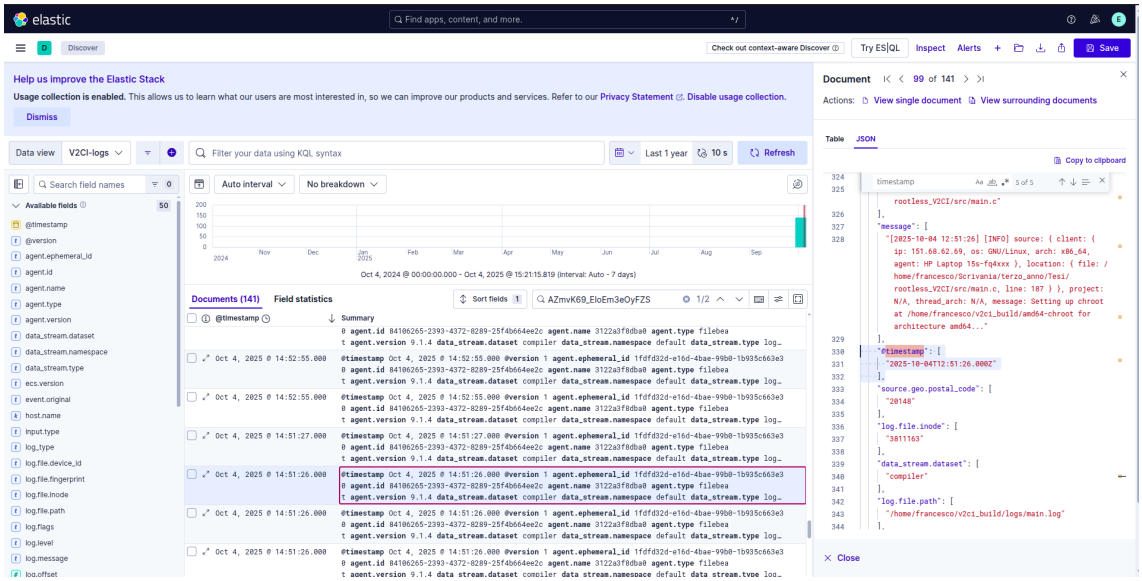

Figura 4.14: Interfaccia di Discover di Kibana per l'esplorazione diretta dei log strutturati di Rootless V^2CI

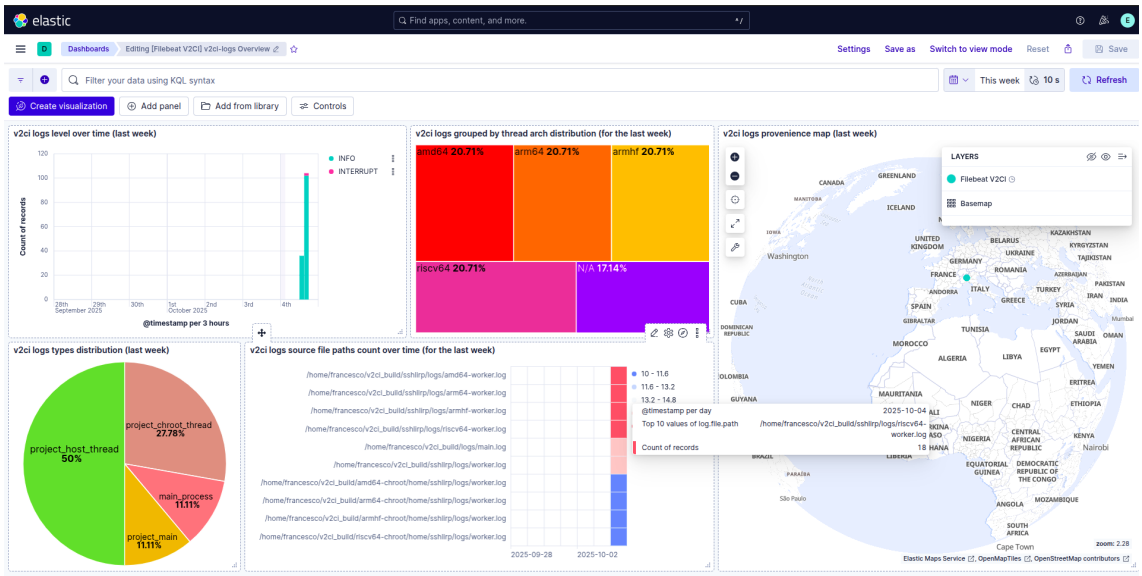

Figura 4.15: Esempio di dashboard personalizzata per il monitoraggio delle build di Rootless V^2CI su Kibana Canvas; realizzata tramite Lens e rappresentativa di vari grafici e mappe geografiche

4.2.6 Analisi dei requisiti di sistema per l'integrazione ELK

La coesistenza di container Docker multipli mirati al funzionamento dell'Elastic Stack, in aggiunta alla già dispendiosa esecuzione del motore di Rootless V^2CI, ha chiaramente portato a scenari di consumo di risorse computazionali per niente

trascurabili.
In scenari di test "All-in-One", in cui sia il motore di Continuous Integration che l'infrastruttura di monitoraggio risiedevano sulla stessa macchina fisica impiegata già precedentemente per le analisi sui consumi di sistema da parte di Rootless V^2CI - discussi nella sottosezione 4.1.6 -, si è potuto osservare come l'overhead aggiuntivo introdotto dall'Elastic Stack incida in media per un ulteriore 30% di CPU per ogni core e 25% di RAM.
Questi Δ di consumo mostrano come su un host con processore 11th Gen Intel® Core™ i5-1155G7 × 8 e 16 GB di RAM, l'esecuzione totale non lasci molto margine in termini di risorse disponibili.
In scenari più stressful - come ad esempio la cross-compilazione multi-threaded per 4 architetture per un solo progetto contemporanea al bootstrap del cluster ELK - si è infatti raggiunto il 100% di utilizzo della CPU su tutti gli 8 core logici e si sono toccati picchi di consumo di RAM fino a 14.7 GB su 16 GB totali, con conseguente swap intensivo e rallentamenti percepibili.

Nonostante ciò, sebbene lo scenario di impiego "Distributed" della combinazione Rootless V^2CI + ELK sia raccomandabile rispetto a quello su singolo host, l'use case generale di quest'ultimo prevede - come verrà illustrato nel tutorial conclusivo - prima un'esecuzione preliminare del sistema di CI e solo in seguito il setup dell'infrastruttura di monitoraggio, consentendo così di mitigare in parte l'impatto sulle risorse computazionali e assorbire in modo del tutto accettabile l'overhead introdotto dall'Elastic Stack.

4.3 Valutazioni totali

In quest'ultima sezione conclusiva, si intende fornire una valutazione complessiva dell'integrazione del sistema di Continuous Integration Rootless V^2CI con l'Elastic Stack, analizzando i principali aspetti qualitativi e quantitativi che ne caratterizzano l'implementazione e l'efficacia.
Come verrà illustrato dalle seguenti sottosezioni, da questa "somma" di considerazioni è emerso che, pur con limiti di natura tecnica e computazionale, Rootless V^2CI con integrazione ELK rappresenta una soluzione valida e promettente per l'automazione e il monitoraggio dei processi di cross-compilazione in ambienti eterogenei, caratterizzata, lato infrastruttura CI, da resilienza, sicurezza, portabilità, scalabilità, configurabilità e architettura ottimizzata, e, lato monitoring, da stan-

dardizzazione, persistenza, gestione ottimizzata delle risorse, usabilità, accessibilità, scalabilità e capacità di raccolta, analisi e visualizzazione avanzata e user-friendly dei log.
Di questi aspetti, in particolare, si analizzeranno quelli comuni ad entrambe le tecnologie, i quali si configurano quindi come i principali punti di forza dell'intera soluzione integrata.

4.3.1 Resilienza delle risorse

Il primo pregio di questa complesssa architettura risiede nella sua resilienza, intesa sia come capacità di adattarsi e rispondere efficacemente a variazioni impreviste dell'ambiente, che come predisposizione a un'operatività idempotente, continua e priva di incoerenze.
Da un lato, Rootless V^2CI, grazie sia all'attenta gestione delle risorse contese tramite `flock()` e file di stato, che alla garanzia di interruzione coerente per mezzo di un programma `v2ci_stop` granulare ma permissivo, esclude scenari di corruzione delle risorse condivise e assicura che ogni build possa essere ripresa o riavviata senza rischi di incoerenza.
Inoltre, con i suoi ultimi aggiornamenti inclusivi di meccanismi di *disaster recovery* e *binaries rotation*, si configura come un sistema di cross-CI "bullet-proof", capace di mantenere l'integrità dei dati e la continuità operativa grazie alla sua massima fault tolerance e configurabilità.

Dall'altro lato, l'Elastic Stack, anche se privato di HA in favore di una gestione economica delle risorse di sistema, in scenari, probabilisticamente parlando, frequenti e standard, grazie al salvataggio dei volumi persistenti su disco simulato e alla garanzia di accesso ad essi tramite loop-back devices, permette di recuperare lo stato del cluster e i dati in esso contenuti anche in caso di crash improvvisi o riavvii non pianificati, assicurando così la persistenza e l'integrità delle informazioni raccolte.

4.3.2 Architettura ottimizzata

Sia Rootless V^2CI che il suo plug-in ELK, sono figli di un'attenta progettazione, derivata da lunghe fasi di studio e sperimentazioni di alternative architetturali, testing e analisi delle risorse.
Infatti, la struttura di Rootless V^2CI basata su *chroot setup iterativo + demoni per progetto + thread per architettura* ha consentito, rispettando ogni singolo requisito

progettuale, di sorpassare problemi di overhead computazionale e di parallelismo da cui sarebbe stata invece affetta una qualsiasi altra configurazione dei medesimi componenti.

Analogamente, lo studio architetturale pedantemente condotto per l'integrazione con l'Elastic Stack, ha portato alla definizione di un sistema di monitoraggio estremamente contenuto, economico e, conseguentemente, abbastanza portabile.

4.3.3 Scalabilità

Uno dei principali punti forti dell'intero ecosistema risiede nella predisposizione allo *scaling-out*.
Rootless V^2CI, grazie alla sua natura modulare e alla gestione indipendente dei progetti e delle architetture, permette di aggiungere facilmente nuovi input di grandezza variabile garantendo comunque tempi di esecuzione ridotti.
L'Elastic Stack, d'altro canto, grazie alla sua architettura distribuita e alla capacità di bilanciare il carico tra i nodi del cluster, consente di scalare orizzontalmente l'infrastruttura su host più performanti e, possibilmente, dedicati.

Ma il vero ventaggio dell'integrazione di questi due servizi, risiede appunto nella loro combinazione che consente a un utente proprietario di più server o macchine virtuali sufficientemente prestanti, di distribuire il carico di lavoro in modo ottimale attraverso il deploy di istanze Rootless V^2CI su nodi dedicati alla compilazione, l'assegnamento agli stessi server dei leggeri data shippers Filebeat, e l'aggregazione centralizzata dei log sull'host dell'Elastic Stack.
In questo scenario, la facilità di distribuzione dei vari servizi si dimostra anche il miglior modo per l'uso ottimizzato delle risorse di sistema.

4.3.4 Usabilità

Un altro aspetto di rilievo dell'architettura creata è la sua usabilità.
Come verrà mostrato nel tutorial conclusivo, l'inizializzazione di Rootless V^2CI richiede all'utente di interfacciarsi principalmente solo con il file `config.yml`, strutturato e leggibile, consetendogli di personalizzare il comportamento del sistema in modo semplice e intuitivo.
Anche il setup dei servizi containerizzati Elastic è stato pensato come un semplice eseguibile unico, capace di impostare anche servizi systemd - sia per Elastic Stack

che per Filebeat - in modo da permettere all'utente di avviare, fermare e monitorare l'infrastruttura con comandi standard e familiari.
Infine, l'interfaccia web di Kibana, con i suoi tool integrati per la visualizzazione dello stato del cluster e l'analisi dei log, rende l'interazione con i dati raccolti semplice, accessibile e anche gradevole.

4.4 Tutorial

Quest'ultima sezione fornisce le istruzioni operative per l'installazione, la configurazione e l'integrazione "plug and play" del sistema di Continuous Integration Rootless V^2CI con il sistema di monitoraggio Elastic Stack.
Verranno coperti entrambi gli scenari operativi: distribuzione su singolo host (All-in-One) e distribuzione distribuita (Builder + Monitor Remoto).

4.4.1 Prerequisiti comuni

Indipendentemente dallo scenario scelto, il primo passo è preparare l'ambiente per il motore di compilazione.
Questo infatti deve essere installato ed eseguito almeno una volta per generare la struttura delle directory di log necessarie a Filebeat.

4.4.2 Preparazione del sistema host builder

Eseguire i seguenti comandi sull'host destinato alla compilazione:

```
sudo apt update
sudo apt upgrade
sudo apt install debootstrap \
  qemu-user-static binfmt-support build-essential \
  cmake git libexecs-dev libyaml-dev cron
```

Accortezza per utenti Ubuntu $\geq$ 24.04:
A causa delle restrizioni di AppArmor sui namespace utente, è necessario eseguire il seguente comando prima di procedere:

```
sudo sysctl -w kernel.apparmor_restrict_unprivileged_userns=0
```

Compilazione di Rootless_V2CI

1. Clonare ed entrare nella directory (si assume che il sorgente sia scaricato in `rootless_V2CI`):

```
cd rootless_V2CI
```

2. Modificare i percorsi di default in `src/include/types/types.h` con i path assoluti corretti per il proprio ambiente:

```
#define DEFAULT_CONFIG_PATH
  "/path/assoluto/a/.config/v2ci/config.yml"
#define SCRIPTS_DIR_PATH
  "/path/assoluto/a/rootless_V2CI/scripts"
```

3. Compilare il progetto:

```
mkdir build && cd build
cmake ..
make
```

Troubleshooting per la compilazione: Se si verificano errori di linking con `libyaml` o `libexecs`, modificare il `CMakeLists.txt` specificando i percorsi assoluti alle librerie statiche (`.a`).

Prima esecuzione obbligatoria

Prima di installare lo stack Elastic, avviare il motore per generare i log:

```
chmod +x ../scripts/*
./v2ci_start
```

Attendere che il sistema completi almeno un ciclo di inizializzazione o build, quindi, se si desidera, fermarlo:

```
./v2ci_stop
```

4.4.3 Scenario 1: deploy su singolo host (All-in-One)

In questo scenario, sia il motore di CI (`Rootless_V2CI`) che lo stack di monitoraggio (`Elastic Stack`) risiedono sulla stessa macchina fisica o virtuale.

Requisiti specifici

Assicurarsi che l'host abbia:

1. Docker e Docker Compose installati;
2. Risorse minime: 4 CPU, 12 GB RAM, 30 GB spazio disco libero.

Step 1: configurazione variabili di ambiente

Navigare nelle directory del repository `rootless_v2ci_logs_ingestion_system`, quindi:

1. Modificare `elastic-log-monitoring/.env`:
   - Impostare le porte e le credenziali desiderate per Kibana/Elasticsearch.
2. Modificare `filebeat-log-monitoring/.env`:
   - Impostare il path assoluto della directory di build di V2CI (dove risiedono i log generati al punto 1.3);
   - Poiché siamo su singolo host, assicurarsi che Filebeat punti a `localhost` o al nome del servizio Docker di Logstash.

Step 2: avvio Elastic Stack (Backend)

Avviare Elasticsearch, Logstash e Kibana:

```
cd elastic-log-monitoring
sudo chmod +x ./*.sh ./scripts/*.sh
sudo ./install.sh
sudo systemctl enable elastic-stack
# Verifica stato
sudo systemctl status elastic-stack
```

Step 3: avvio Filebeat (Agent)

Avviare l'agente che legge i log locali e li invia allo stack:

```
cd ../filebeat-log-monitoring
sudo chmod +x ./*.sh
sudo ./install.sh
sudo systemctl enable v2ci-filebeat
```

```
# Verifica stato
sudo systemctl status v2ci-filebeat
```

Step 4: accesso

Accedere a Kibana via browser: `https://localhost:5601` (o IP dell'host). Accettare il certificato SSL autofirmato se richiesto, o, se segnalata connessione non sicura, inserirlo tra i certificati browser

4.4.4 Scenario 2: deploy su host remoti ma comunicanti (Distributed)

In questo scenario, `Rootless_V2CI` gira sull'Host A (Builder), mentre `Elastic Stack` gira sull'Host B (Monitor). Filebeat verrà installato sull'Host A per inviare i log all'Host B.

Host B (Monitor): installazione Elastic Stack

Eseguire questi comandi sul server dedicato al monitoraggio:

1. Configurare `elastic-log-monitoring/.env` assicurandosi che le porte di Logstash (solitamente 5044) siano esposte per accettare connessioni dall'esterno;
2. Avviare lo stack:

```
cd elastic-log-monitoring
sudo chmod +x ./*.sh ./scripts/*.sh
sudo ./install.sh
sudo systemctl enable elastic-stack
```

Host A (Builder): installazione Filebeat

Eseguire questi comandi sulla macchina dove gira `Rootless_V2CI`:

1. **Configurazione fondamentale:** modificare il file `filebeat-log-monitoring/.env`:
   - `LOG_DIR`: Path assoluto dei log di V2CI sull'Host A.;
   - `LOGSTASH_HOST`: Inserire l'indirizzo IP dell'Host B (es. `192.168.1.50`).

2. Avviare Filebeat:

```
cd filebeat-log-monitoring
sudo chmod +x ./*.sh
sudo ./install.sh
sudo systemctl enable v2ci-filebeat
```

4.4.5 Gestione operativa

Rotazione binari

Lo script `v2ci_start` installa automaticamente un cronjob per la pulizia dei binari vecchi. Per rimuoverlo:

```
crontab -e
# Rimuovere la riga relativa a v2ci
```

Reset dei servizi

Se necessario resettare l'intero stack di monitoraggio (**attenzione: cancella i dati**):

- **Su Host Monitor (o unico host):**

```
cd elastic-log-monitoring
sudo ./reset.sh
sudo systemctl restart elastic-stack
```

- **Su Host Builder (o unico host) per Filebeat:**

```
cd filebeat-log-monitoring
sudo systemctl stop v2ci-filebeat
sudo docker volume rm registry
sudo systemctl start v2ci-filebeat
```

Conclusioni

In questa trattazione si è cercato di illustrare, con rigore metodologico e attenzione ai dettagli tecnici, il processo evolutivo che ha condotto alla realizzazione dell'infrastruttura di Rootless V^2CI e alla sua successiva integrazione con i servizi di monitoraggio ELK. L'analisi ha ripercorso le tappe fondamentali dello sviluppo, partendo dalle prime sperimentazioni manuali di pacchettizzazione fino al consolidamento di un'architettura software distribuita e resiliente, ponendo l'accento sulle scelte progettuali dettate da vincoli di sicurezza, performance e usabilità.

La genesi di Rootless V^2CI, radicata nella necessità di espandere la frontiera di connettività dei progetti Virtualsquare tramite la distribuzione capillare di sshlirp, ha dimostrato come un'istanza specifica e circoscritta possa evolvere in un sistema generalizzato. Il passaggio dalle soluzioni intermedie - sshlirpCI e la sua variante Rootless - ha segnato un percorso di maturazione tecnica guidato dalla volontà di superare i limiti imposti dai privilegi di sistema. L'adozione di primitive quali user namespaces e l'uso combinato di `fakeroot` e `unshare` hanno permesso di disaccoppiare definitivamente il processo di costruzione del software dai permessi amministrativi dell'host, raggiungendo l'obiettivo primario di una "rootlessness" nativa e sicura.

L'architettura finale, basata su un modello concorrente di demoni per progetto e thread per architettura, ha validato l'efficacia di un approccio modulare. Attraverso test empirici e misurazioni dei consumi di sistema, si è comprovato come Rootless V^2CI riesca a garantire scalabilità e velocità di esecuzione, pur mantenendo un'impronta computazionale gestibile. L'introduzione di meccanismi avanzati quali il *disaster recovery* autonomo e la rotazione intelligente dei binari conferisce al sistema un grado di fault tolerance che lo rende idoneo a operare in continuità.

Infine, la decisione di arricchire l'ecosistema con un layer di osservabilità basato sullo stack Elastic ha colmato il divario tra la complessità delle operazioni di basso livello e l'accessibilità utente. L'orchestrazione di container Docker per l'ingestion e la visualizzazione dei log, unita all'impiego di Filebeat come data shipper leggero,

ha trasformato Rootless V^2CI da mero strumento di compilazione a piattaforma monitorabile in tempo reale, capace di fornire insight immediati sullo stato delle build distribuite.

In definitiva, sebbene Rootless V^2CI presenti requisiti di risorse che potrebbero limitarne l'impiego in contesti hardware estremamente vincolati, l'infrastruttura realizzata soddisfa pienamente gli obiettivi accademici e pratici prefissati. Essa rappresenta una soluzione valida e innovativa nel panorama della Continuous Integration, proponendosi come un'alternativa sicura, portabile e trasparente per la produzione automatizzata di software cross-compilato, fedele alla filosofia open source e ai principi di libertà e accessibilità promossi da Virtualsquare.

Bibliografia

[1] R. Davoli *et al.*. "Virtual Square wiki (2.0)", `http://wiki.virtualsquare.org/`.

[2] R. Davoli, M. Goldweber (2005). *Virtual Square (V^2) in Computer Science Education*, `http://www.cs.unibo.it/renzo/papers/2005.iticse-v2.pdf`.

[3] Landley, Rob (2009-12-16). "Rob's quick and dirty UML howto", `http://www.landley.net/code/UML.html`.

[4] The QEMU Project Developers (2025). "Qemu docs", `http://www.qemu.org/docs/master/`.

[5] Alessandro Tarasio - INFN - ReCaS (2014). *Concetti di base sulla Virtualizzazione.*

[6] George Ferguson (2006-01-08). "Ubuntu slirp manpages - Description", `http://manpages.ubuntu.com/manpages/focal/man1/slirp.1.html#description`.

[7] R. Davoli *et al.* (2024). "sshlirp github repository", `http://github.com/virtualsquare/sshlirp`.

[8] R. Davoli (2012-12-27). "Virtualsquare wiki - Design Guidelines for Virtual Square", `http://wiki.v2.cs.unibo.it/wiki/index.php%3Ftitle=Introduction.html#Design_guidelines_of_Virtual_Square`.

[9] R. Davoli (2021-07-23). "vdeplug4 github repository", `http://github.com/rd235/vdeplug4`.

[10] R. Davoli (2019-11-15). "vdens github repository", `http://github.com/rd235/vdens`.

[11] R. Davoli *et al.* (2016-08-23). "`libvdeplug_vxvde` manpage", `http://wiki.virtualsquare.org/#/man/man1/libvdeplug_vxvde.1.html`.

[12] R. Davoli (2019-12-26). "libslirp github repository", `http://github.com/rd235/libslirp`.

[13] R. Davoli *et al.* (2019-12-29). "libvdeslirp github repository", `http://github.com/virtualsquare/libvdeslirp`.

[14] Michael K. Johnson (1996-04-01). "Choosing an Internet Service Provider", Linux Journal (24), Specialized Systems, Seattle, USA, ISSN 1075-3583, `http://www.linuxjournal.com/article/1233`.

[15] Jim Knoble (1996-04-01). "Almost Internet with SLiRP and PPP", Linux Journal (24), Specialized Systems, Seattle, USA, ISSN 1075-3583, `https://www.linuxjournal.com/article/1174`.

[16] Glen Reesor (1997-08-07). "SLIP/PPP Emulator mini-HOWTO", `http://www.pluto.it/sites/default/files/ildp/HOWTO/SLIP-PPP-Emulator/SLIP-PPP-Emulator.html`.

[17] R. Davoli (2025-09-06). "sshlirp: create an instant VPN using VDE and slirp", ESC Endsummer Summer Camp, Parco della Scultura in Architettura, San Donà di Piave (VE), 2-7 Settembre 2025.

[18] Debian Foundation (2017-08-19). "Salsa - Debian wiki", `http://wiki.debian.org/Salsa`.

[19] Otto Kekäläinen (2025-07-14). "DEP-18: Encourage Continuous Integration and Merge Request based Collaboration for Debian packages", `http://dep.debian.org/deps/dep18`.

[20] Salsa CI Team (2018-06-11). "Salsa CI Pipeline - GitLab repository", `http://salsa.debian.org/salsa-ci-team/pipeline#salsa-continuous-integration-ci--quality-assurance-for-debian-packaging`.

[21] Debian developers (2025-07-24). "PackagingWithGit - Debian wiki", `http://wiki.debian.org/PackagingWithGit`.

[22] I. Jackson, C. Schwarz (1996, 1197, 1998). "Source Packages - Debian Policy Manual", `http://www.debian.org/doc/debian-policy/ch-source.html`.

[23] Raphaël Hertzog (2020-11-29). "DEP-14: Recommended layout for Git packaging repositories", http://dep-team.pages.debian.net/deps/dep14/.

[24] Developer's Reference Team (2019). "Buone pratiche per la pacchettizzazione - Debian developers-reference", http://www.debian.org/doc/manuals/developers-reference/best-pkging-practices.

[25] Guido Günther (2025-04-11). "Configuration Files", http://honk.sigxcpu.org/projects/git-buildpackage/manual-html/gbp.cfgfile.html.

[26] Joey Hess (2025-01-02). "dh_install(1) — Linux manual page", http://man7.org/linux/man-pages/man1/dh_install.1.html.

[27] Debian developers (2025). "File list of package libslirp-dev in trixie of architecture amd64", http://packages.debian.org/trixie/amd64/libslirp-dev/filelist.

[28] Virtualsquare team (2025). "Debian VisrtalSquare Team - Debian Salsa GitLab official page", http://salsa.debian.org/virtualsquare-team.

[29] Debian community (2025). "DebianInstaller Qemu", http://wiki.debian.org/DebianInstaller/Qemu.

[30] Debian community (2025). "Network install from a minimal USB, CD", http://www.debian.org/CD/netinst/index.en.html.

[31] TianoCore EDK II Developers (2025). "edk2 - official github repository", http://github.com/tianocore/edk2.

[32] Debian Community (2025). "Source Package: edk2 (2025.08.01-1)", http://packages.debian.org/source/sid/edk2.

[33] Mark McLoughlin (2008-09-11). "The QCOW2 Image Format", http://web.archive.org/web/20121004073848/http://people.gnome.org/~markmc/qcow-image-format.html.

[34] The QEMU Project Developers (2025). "Qemu docs - System Emulation - Direct Linux Boot", http://www.qemu.org/docs/master/system/linuxboot.html.

[35] Debian community (2025). "Index of /cdimage/archive/12.12.0/armel - iso-cd/", https://cdimage.debian.org/cdimage/archive/12.12.0/armel/iso-cd/.

[36] Ryutaroh Matsumoto (2020-12-11). "Bug#977126: linux: No armel kernel can be booted by grub-efi-arm:armel", http://www.mail-archive.com/debian-bugs-dist@lists.debian.org/msg1779262.html.

[37] The QEMU Project Developers (2025). "Qemu docs - User Mode Emulation - QEMU User space emulator - System call translation", http://www.qemu.org/docs/master/user/main.html.

[38] Debian community (2025). "Debootstrap - Debian wiki", http://wiki.debian.org/Debootstrap.

[39] Matt Kraai - Debian Project (2001-04-27). "debootstrap - Bootstrap a basic Debian system - DEBOOTSTRAP(8) manpage", http://manpages.debian.org/trixie/debootstrap/debootstrap.8.en.html.

[40] Debian community (2025). "Package: binfmt-support (2.2.2-7 and others)", http://packages.debian.org/it/sid/binfmt-support.

[41] Debian community (2025). "pbuilder - Debian wiki", http://wiki.debian.org/pbuilder.

[42] Debian community (2025). "qemubuilder - Debian wiki", http://wiki.debian.org/qemubuilder.

[43] Debian community (2025). "Salsa Doc - Debian wiki", http://wiki.debian.org/Salsa/Doc.

[44] Salsa CI Team (2025). "Salsa CI Pipeline README - GitLab repository", http://salsa.debian.org/salsa-ci-team/pipeline/-/blob/master/README.md.

[45] Sudo Project (2025). "Commit history - Sudo github repository", http://github.com/sudo-project/sudo/commits/main/?after=02ff3afc1b71f602b62ce0be5efc35892d95910a+174.

[46] National Vulnerability Database (06/30/2025). "CVE-2025-32463", http://nvd.nist.gov/vuln/detail/cve-2025-32463.

[47] Sudo Project (2025). "Commit fdafc2c - Revert pivot_root and go back to prepending the new root directory. - Sudo github repository", http://github.com/sudo-project/sudo/commit/fdafc2ceb36382b07e604c0f39903d56bef54016.

[48] Michael Kerrisk (2025-05-17) "system(3) — Linux manual page", http://man7.org/linux/man-pages/man3/system.3.html.

[49] Ansgar Burchardt (2016-10-20) "debootstrap/functions file - debootstrap github repository", http://github.com/aburch/debootstrap/blob/master/functions.

[50] Joost Witteveen, Clint Adams, Timo Savola (2014-10-05). "fakeroot(1) Debian manual", http://manpages.debian.org/stretch/fakeroot/fakeroot.1.en.html.

[51] Michael Kerrisk (2025-09-06). "capabilities(7) — Linux manual page", http://man7.org/linux/man-pages/man7/capabilities.7.html.

[52] Debian community (2023-05-13). "PROOT(1) General Commands Manual", http://manpages.debian.org/experimental/proot/proot.1.en.html.

[53] GitHub user 'NTheCuteDrone' (2025-06-19). "[Bug]: cannot install distro #527 - termux/proot-distro github repository", http://github.com/termux/proot-distro/issues/527.

[54] GitHub user 'sylirre' (2023-05-09). "[INFO | BUG]: QEMU no longer works with PRoot Distro #299 - termux/proot-distro github repository", http://github.com/termux/proot-distro/issues/299.

[55] Michael Kerrisk (2023-06-06). *Understanding user namespaces*, http://man7.org/conf/corec%2B%2B2023/understanding-Linux-user-namespaces--CoreC%2B%2B-2023--Kerrisk.pdf.

[56] Maxime Bélair (2025-03-27). "Understanding AppArmor User Namespace Restriction", http://discourse.ubuntu.com/t/understanding-apparmor-user-namespace-restriction/58007.

[57] Michael Kerrisk (2025-09-06). "unshare(1) — Linux manual page", http://man7.org/linux/man-pages/man1/unshare.1.html.

[58] Alex Bradbury (2024). "Muxup - Rootless cross-architecture debootstrap", http://muxup.com/2024q4/rootless-cross-architecture-debootstrap.

[59] Alex Bradbury (2024-12-03). "medley/rootless-debootstrap-wrapper - muxup github repository", http://github.com/muxup/medley/blob/main/rootless-debootstrap-wrapper.

[60] Virtualsquare team (2025). "s2argv-execs github repository", http://github.com/virtualsquare/s2argv-execs/tree/master.

[61] Kirill Simonov (2006-2016). "LibYAML wiki", http://pyyaml.org/wiki/LibYAML.

[62] Michael Kerrisk (2025-09-06). "dpkg(1) — Linux manual page", http://man7.org/linux/man-pages/man1/dpkg.1.html.

[63] Michael Kerrisk (2025-09-06). "cron(8) — Linux manual page", http://man7.org/linux/man-pages/man8/cron.8.html.

[64] Michael Kerrisk (2025-09-06). "crontab(5) — Linux manual page", http://man7.org/linux/man-pages/man5/crontab.5.html.

[65] Michael Kerrisk (2025-09-06). "crontab(1) — Linux manual page", http://man7.org/linux/man-pages/man1/crontab.1.html.

[66] Elastic community (2025). "elastic - official github repository", http://github.com/elastic.

[67] Elasticsearch N.V. (Naamloze Vennootschap) (2025). "Elastic Docs", http://www.elastic.co/docs/.

[68] Chad Horohoe (2014-01-06). "Wikimedia moving to Elasticsearch - Wikimedia blog", http://blog.wikimedia.org/2014/01/06/wikimedia-moving-to-elasticsearch/.

[69] Anurag Phadke (2010-12-10). "Flume, Hive and realtime indexing via ElasticSearch - Blog of Data", http://web.archive.org/web/20150304213251/https://blog.mozilla.org/data/2010/12/30/flume-hive-and-realtime-indexing-via-elasticsearch-2/.

[70] Tim Pease (2013-01-23). "A Whole New Code Search - GitHub Blog", https://github.blog/news-insights/product-news/a-whole-new-code-search/.

[71] Sagar Loke and Christos Kalantzis (2014-11-10). "Introducing Raigad - An Elasticsearch Sidecar - Netflix TechBlog", http://netflixtechblog.com/introducing-raigad-an-elasticsearch-sidecar-350c7e01339f.

[72] Clinton Gormley and Zachary Tong (2015). *Elasticsearch: The Definitive Guide (A DISTRIBUTED REAL-TIME SEARCH AND ANALYTICS ENGINE)*, O'Reilly Media, ISBN: 978-1-449-35854-9.

[73] Elasticsearch N.V. (Naamloze Vennootschap) (2025). "Logstash Reference", http://www.elastic.co/guide/en/logstash/index.html.

[74] Elasticsearch N.V. (Naamloze Vennootschap) (2025). "Syslog input plugin - Elastic Docs/ Reference/ Ingestion tools /Logstash Plugins /Input plugins", http://www.elastic.co/docs/reference/logstash/plugins/plugins-inputs-syslog.

[75] Elasticsearch N.V. (Naamloze Vennootschap) (2025). "Kibana Guide", http://www.elastic.co/guide/en/kibana/index.html.

[76] Elasticsearch N.V. (Naamloze Vennootschap) (2025). "Beats Platform Reference", http://www.elastic.co/guide/en/beats/libbeat/index.html.

[77] Elasticsearch N.V. (Naamloze Vennootschap) (2025). "Filebeat Reference", http://www.elastic.co/guide/en/beats/filebeat/index.html.

[78] Elasticsearch N.V. (Naamloze Vennootschap) (2025). "Quorum-based decision making - Elastic Docs /Deploy and manage /Distributed architecture /Discovery and cluster formation", http://www.elastic.co/docs/deploy-manage/distributed-architecture/discovery-cluster-formation/modules-discovery-quorums.

[79] Dave Moore (2019-05). *Elasticsearch - Size and Capacity Planning.*

[80] Elastic community (2025). "Elastic Docker Hub", http://hub.docker.com/u/elastic.

[81] Eddie Mitchell (2023-05-17). "Getting started with the Elastic Stack and Docker Compose - Elastic Blog", http://www.elastic.co/blog/getting-started-with-the-elastic-stack-and-docker-compose.

[82] Red Hat Inc. (2025). "Logical Volume Manager Administration - Red Hat Enterprise Linux 5 - Guida per l'amministratore LVM", http://docs.redhat.com/it/documentation/red_hat_enterprise_linux/5/html/logical_volume_manager_administration/index.

[83] Ubuntu community (2020-02-09). "loop, loop-control - loop devices - Ubuntu manpage", http://manpages.ubuntu.com/manpages/focal/man4/loop.4.html.

Ringraziamenti

Ringrazio innanzitutto di cuore il mio relatore, il professore Renzo Davoli, che non solo ha acceso in me la passione verso gli argomenti che ho amato trattare in questa tesi, ma che è anche stato una guida etica che ho avuto la fortuna di conoscere e da cui ho avuto l'onore di imparare come studente e soprattutto come persona.

Ringrazio con amore la mia famiglia, senza la quale non sarei mai riuscito ad innamorarmi delle difficoltà che mi hanno cresciuto come uomo durante questo percorso universitario. Alla mia mamma dedico la gratitudine di avermi spinto a credere in me stesso e di avermi condotto sempre verso l'impegno. A mia sorella voglio dire grazie per tutti i giochi e gli scherzi che mi ha permesso di condividere con lei anche nei momenti più ardui. Al mio babbo: grazie per essere stato paziente come nessun altro, grazie per avermi sempre rincoraggiato e grazie per non aver mai permesso che mi sentissi solo.

Un ringraziamento speciale anche a tutti i miei amici.
Le superiori sono state fantastiche. Il Copernico mi ha lasciato tanto alle spalle, anche memorie difficili. Ma con voi mi sono sempre sentito bene, mi avete sempre valorizzato come nessun altro e non avete mai smesso di credere in me come il ciofo simpatico e divertente che adoro sentirmi quando sono con voi. Tra i miei ex compagni di classe voglio soprattuto ringraziare Zhou. Sei una persona molto profonda, dalla bontà e dalla saggezza illuminanti. Voglio rimanere tuo amico per sempre, perché non mi sono mai sentito tanto ascoltato né tanto libero come lo sono stato con te. Ringrazio in particolare anche la mia migliore amica, la Mati. Le nostre risate a tarda sera non hanno mai avuto fine nella mia testa e quando sento che voglio di uscire, la prima persona a cui penso sei tu.
All'università invece ho conosciuto sei ragazzi, tutti di un'intelligenza incredibile e dalle ambizioni brillanti. Mi avete insgnato cos'è l'amicizia adulta e con voi non ho mai avuto paura di perdere il sorriso. Siete persone fantastiche a cui voglio augurare il meglio nella vita e che spero di non perdere mai.
Poi, un ringraziamento dedicato a Matteo, e ovviamente a tutte le belle persone

che mi ha fatto conoscere. Sei il quinto componente della mia famiglia. Mi hai accompagnato in mille follie e sostenuto in centomila investimenti. Ma sopratutto, mi hai sempre dato la motivazione, il coraggio, la speranza e la spensieratezza che non avrei mai potuto ricevere da altri e che credo di aver vissuto solo quando ero piccolo e già giocavamo assieme.

Infine, ringrazio chi mi ha disegnato la "P" che mi porterò dietro una vita intera. Mi sono commosso molte volte pensandoti e trovato il coraggio dove non c'era dentro di me. Mi hai dato tutto quello che sono e la speranza di quello che vorrei essere.