UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA DELL'INFORMAZIONE

"Esecuzione e adattamento di applicazioni GNU/Linux su piattaforma Android"

Relatore: Dott. Carlo Fantozzi

Laureando: Luca Fabbian

ANNO ACCADEMICO 2020–2021

Data di laurea 24/09/2021

Ringraziamenti

Ringrazio innanzitutto i miei genitori, che in questi ultimi mesi e in tutti i ventun anni precedenti non mi hanno mai fatto mancare il loro supporto e affetto. Grazie per essermi sempre stati vicini e per avermi sostenuto quando ero in difficoltà.

Ringrazio i colleghi universitari per i momenti di divertimento e le conversazioni mai banali. In particolare, ringrazio Filippo Bragato, Paolo Bresolin, Filippo Simonetti ed Enrico Tosi, con cui ho trascorso le ultimi lezioni in presenza, e i miei coinquilini e amici Gianluigi Galeotafiore e Diego Antignati.

Ringrazio i compagni di liceo Giosuè Sardo, Maxim Kovalkov, Antonio Napolitano, Filippo Casarin, Andrea Zigiotti, Lorenzo Botter, Gabriele Saracco, Matteo Secco e il resto del gruppo con cui ho affrontato il concorso “Zerorobotics”, un’esperienza che mi ha fatto riscoprire la passione per l’informatica, in un momento in cui ero incerto sul percorso da intraprendere.

Ringrazio inoltre il Dottor Emanuele Rocco, il Dottor Andrea Saiani, il laureando Simone Nascivera e gli altri membri dello staff del laboratorio Witted, che mi hanno accompagnato in molteplici esperienze di stage, contribuendo in modo fondamentale alla mia formazione personale e professionale. Sarò sempre loro grato per i preziosi consigli e per avermi continuato ad aiutare anche al di fuori dell’esperienza lavorativa.

Ringrazio infine il professor Riccardo Colpi, per avermi aiutato nell’iscrizione a Ingegneria dell’Informazione, e il professor Carlo Fantozzi, per l’impegno costante con cui ha svolto il ruolo di relatore, seguendomi e dimostrandosi sempre competente, affidabile e puntuale.

Sommario

Abstract

I cellulari Android moderni presentano un potere computazionale paragonabile a quello dei PC, ma non è possibile eseguirvi con facilità software come LibreOffice, Gcc, Python o Java, tipicamente disponibili nei sistemi operativi GNU/Linux.

Dopo aver analizzato le principali incompatibilità tra Android e i sistemi GNU/Linux, a partire dal kernel fino alle librerie ad alto livello, questa tesi descriverà come sia possibile superarle, evidenziando però i difetti degli approcci esistenti. Saranno considerate per il confronto le app IBochs, Limbo, Termux, AnLinux, Andronix e Debian noroot. Verrà quindi proposto un nuovo approccio al problema: uso di web app con a disposizione una potente API fornita da un server locale, che rende possibile sfruttare allo stesso tempo la libreria di sistema Android e gli eseguibili installabili all'interno di Alpine Linux.

Per dimostrare la realizzabilità dell'approccio proposto verrà creata un'app Android chiamata Unyw (https://github.com/unyw) che permetta l'installazione ed esecuzione di tali web app. Il prototipo ottenuto consentirà di evidenziare i vantaggi della soluzione proposta, che non solo rende possibile eseguire applicazioni GNU/Linux arbitrarie così come sono (incluse quelle a interfaccia grafica), ma consente anche di usare le tecnologie web (HTML/CSS/JavaScript/Svelte) per creare dei wrapper che adattino tali programmi al sistema Android, senza che ci sia bisogno di intervenire sui binari o, in generale, di compilare una singola riga di codice.

Per completare il progetto si realizzeranno alcune web app di esempio, fra cui quelle che permettono di interfacciarsi con il package manager di Alpine Linux, l'emulatore di terminale XTerm, il browser Chromium, e la funzione di conversione file di Libreoffice. Verranno inoltre forniti librerie e script che facilitino la scrittura e la distribuzione di altri wrapper, rendendo possibile la realizzazione di questi ultimi con poche righe di codice.

Unyw, l'app Android ottenuta, si dimostrerà ottima sia per quanto riguarda la dimensione su disco (circa 86Mb al termine dell'installazione, un risultato migliore di tutte le altre soluzioni esaminate), sia per le prestazioni raggiunte (paragonabili o superiori a tutte le altre soluzioni esaminate), sia per l'estrema facilità con cui è possibile espanderla, aggiungendo non solo programmi a linea di comando, ma nuove schermate con le proprie interfacce grafiche. Nella conclusione, si elencheranno le modifiche da effettuare al prototipo per prepararlo alla distribuzione.

1. Introduzione

Gli smartphone di oggi sono per molti aspetti equiparabili a PC, al punto che esistono anche cellulari con una potenza di calcolo paragonabile a quella dei computer di fascia media/alta.

Vista l'assenza di limiti hardware, ci si aspetterebbe che il parco software dei computer desktop sia utilizzabile con semplicità anche dai cellulari. In particolare i dispositivi Android, basati sullo stesso kernel dei sistemi desktop GNU/Linux, dovrebbero avere facile accesso alle numerose applicazioni disponibili su questi ultimi. Ciò include:

- l'intera collezione di utility GNU, inclusi il compilatore Gcc, la shell Bash e comandi come Gzip;
- la Java Virtual Machine e i programmi scritti in Java;
- applicazioni con interfaccia grafica come LibreOffice, Firefox, Chromium e Audacity.

Solo alcune delle applicazioni elencate hanno un porting su Android e nella maggior parte dei casi si tratta di un porting parziale: ad esempio Firefox per Android non include gli strumenti sviluppatore. Vi è quindi interesse nell'avere un metodo generale per eseguire la versione PC di queste applicazioni; tuttavia, mentre esistono da anni soluzioni valide per eseguire app Android su desktop (ad esempio Anbox **[48]**), solo recentemente si è posto il problema inverso e tutt'oggi ci sono sfide ancora aperte.

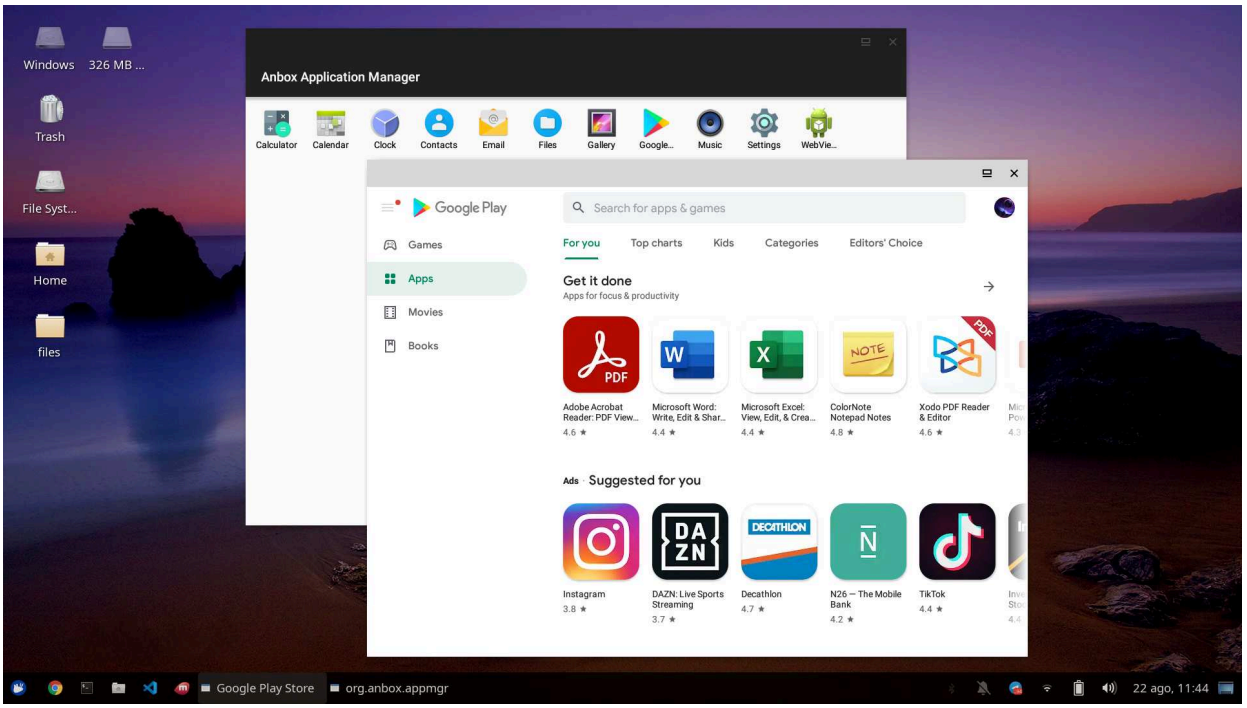

Figura 1: *Screenshot di Anbox perfettamente integrato all'interno di XUbuntu 20.04*

Android, infatti, presenta molte scelte architetturali incompatibili con i sistemi GNU/Linux e impone numerose limitazioni di sicurezza che rendono più difficile progettare un emulatore e integrarlo adeguatamente con il sistema. La quasi totalità delle soluzioni esistenti si limita quindi ad eseguire un sistema GNU/Linux così com'è, senza offrire alcun tipo di controllo migliorato per i dispositivi mobili; dato che i cellulari sono sprovvisti di tastiera e hanno schermo piccolo, questo rende alcuni programmi inutilizzabili. Questa tesi, dopo aver analizzato i problemi presenti, si occuperà pertanto proporre un nuovo approccio in grado di risolvere molte delle criticità riscontrate.

1.1 Distro prese in esame e metriche considerate

Essendo sia Android che GNU/Linux di base open-source, esistono numerose varianti, chiamate usualmente “distro”, di questi sistemi operativi; il sito DistroWatch, ad esempio, conta più di 500 varianti di sistemi operativi GNU/Linux. Alcune distro sono molto simili fra loro, ma altre presentano caratteristiche inusuali e in antitesi con quelle delle distro più popolari. Trattare tutte le distro esistenti sarebbe inutile e controproducente ai fini di questa tesi, visto che introdurrebbe numerosi problemi che non si pongono con la grande maggioranza del software GNU/Linux.

Quando non specificato altrimenti, pertanto, ci si limiterà a prendere come modello rappresentativo i sistemi GNU/Linux più popolari, come Ubuntu **[49]**, Debian **[50]**, Arch Linux **[51]** e Alpine Linux **[52]**. A volte verrà citato anche Tinycore Linux **[53]**, come esempio di distro “live” (cioè che non necessita di essere installata perché in grado di funzionare interamente dalla RAM) che riesce ad offrire un ambiente grafico in soli 12 Mb. Per i dispositivi Android, si considereranno le sole caratteristiche presenti su uno smartphone generico e disponibili all’utente senza bisogno di manomettere il dispositivo sbloccando il bootloader o il root user.

Inoltre, considerando che, in pratica, la prestazione di un eseguibile è influenzata da numerosi fattori, nella maggior parte dei casi ci si concentrerà sulle funzionalità presenti nelle soluzioni analizzate. In alcuni casi, tuttavia, sarà necessario argomentare l'estrema inefficienza di alcune soluzioni rispetto ad altre; per farlo, verranno realizzate delle benchmark utilizzando un dispositivo reale. Considerando inoltre che i dispositivi mobile spesso hanno memoria limitata, ci si preoccuperà di confrontare le soluzioni anche sulla base dello spazio su disco che richiedono.

1.2 Argomenti e struttura della tesi

Questa tesi si occuperà di:

- analizzare le incompatibilità tra i sistemi Android e quelli GNU/Linux, partendo da quelle a basso livello fino ad alto livello ed enunciando di volta in volta le possibili strategie per risolvere i problemi emersi (capitolo 2).
- Confrontare le soluzioni già esistenti, dividendole tra basate su emulazione hypervisor e basate su meccanismi chroot. Di ciascuna verranno elencati pregi e difetti e si effettueranno dei test per misurarne le prestazioni (capitolo 3).
- Proporre un nuovo approccio al problema, partendo dalle considerazioni raccolte in precedenza: un'app Android che consenta di eseguire applicazioni sviluppate con tecnologie web ma con a disposizione una potente API per integrarsi sia con il sistema Android che con un sistema Alpine Linux eseguito tramite meccanismi chroot. Tale app verrà descritta dal punto di vista teorico e, allo stesso tempo, ne verrà fornita un’implementazione di riferimento a dimostrazione dei concetti espressi (capitolo 4).
- Analizzare i risultati ottenuti. Si concluderà effettuando un confronto con le altre soluzioni prese in esame e considerando le possibili estensioni alla soluzione proposta (capitolo 5).

2. Incompatibilità tra sistemi Android e GNU/Linux

2.1 Kernel

I sistemi Android e i sistemi GNU/Linux hanno come base lo stesso kernel. Al kernel Android viene però applicata una patch **[1]**, diversa da dispositivo a dispositivo, che mira ad aggiungere funzionalità. Questo include:

- generici miglioramenti e feature sviluppati in anteprima dal team Android. In questi casi, di solito, entro qualche anno le aggiunte vengono introdotte come standard nel kernel Linux principale. Questo è successo, ad esempio, per l'EAS **[2]** (Energy Aware System, un modulo che aiuta nel risparmio energetico), alcuni moduli riguardanti il networking, e alcuni moduli per la gestione del filesystem come F2FS **[3]** (filesystem sviluppato da Samsung come competitor a ext4).
- Funzionalità pensate specificatamente per Android, come un modulo per la gestione del logging, l'Android Shared Memory e un sistema di Wakelocks e di sveglie.
- Driver di terze parti, che spesso vengono inseriti come bloat binari, cioè come software già compilati a parte in precedenza, e che quindi possono essere closed-source.

Molte di queste funzionalità sono così a basso livello che vengono sfruttate solo dal sistema operativo o dal kernel stesso. Non generano quindi incompatibilità nelle applicazioni, che anzi ne beneficiano implicitamente. Ad esempio, la presenza del modulo F2FS consente di scrivere su filesystem F2FS con la generica chiamata per scrivere file, senza che l'applicazione sia cosciente di aver chiamato internamente il modulo o di star scrivendo su filesystem F2FS.

Vi è però una differenza sostanziale su come vengono gestiti i moduli del kernel.

Molti sistemi GNU/Linux consentono all'utente di scegliere se installare un kernel completamente open-source (Linux-Libre) o uno con alcuni driver proprietari closed-source. Viene inoltre data all'utente la possibilità di installare moduli aggiuntivi in un secondo tempo.

Al contrario, i dispositivi Android vengono venduti con un kernel preinstallato che, in mancanza dei permessi amministratore, non è modificabile e che presenta binari di cui non è noto il codice sorgente. Questo rende impossibile eseguire specifici programmi GNU/Linux che per funzionare hanno bisogno di installare moduli kernel, come ad esempio v4l2loopback, un software per usare come telecamera uno stream video.

Una possibile soluzione per risolvere tutte le incompatibilità è emulare l'intero kernel, ma ciò comporta che vada emulato anche il resto del sistema operativo, con evidenti costi in termini di prestazioni. Le soluzioni di questo tipo verranno analizzate nel capitolo 3.1

Una soluzione più raffinata consiste nell'emulare solo i moduli kernel necessari, sfruttando gli strumenti di debug presenti nel kernel stesso. In Linux, infatti, un programma, può chiamarne un altro e chiedere di intervenire in alcuni punti della sua esecuzione grazie alla chiamata ptrace **[4]**. Questa funzione, pensata inizialmente per permettere agli sviluppatori di monitorare e testare i programmi, consente a tutti gli effetti di rimappare alcune istruzioni, incluse chiamate a sistema di solito gestite dal kernel. Le soluzioni di questo tipo verranno analizzate nel capitolo 3.2

2.2 Architettura CPU

Android **[5]** e GNU/Linux **[6]** sono entrambi compatibili sia con i processori x86/64 che ARM (inclusa la versione 64 bit). Tuttavia, in pratica, la quasi totalità dei device in cui è installato Android è ARM, mentre la maggior parte dei sistemi GNU/Linux è nata per computer desktop e per questo molti programmi vengono rilasciati solo per x86 o x86/64 (ne è un esempio il software di videoconferenze Zoom). Ciò comporta che, in genere, i binari disponibili su desktop non possono essere utilizzati se non tramite emulazione. La quasi totalità dei software open-source (tutti quelli scritti in linguaggi portable, cioè che non includono chiamate assembly specifiche), può tuttavia essere ricompilata per adattarsi al nuovo set di istruzioni.

Fortunatamente, la popolarità dell'architettura ARM anche al di fuori dei cellulari, come ad esempio nei dispositivi embedded (come Raspberry PI) e nei server, ha spinto le principali distro GNU/Linux (incluse Debian, Ubuntu e Arch Linux) a rilasciare un porting ARM sia del sistema base che di tutti i software open-source disponibili nelle loro repository.

Si segnala, inoltre, che i kernel Linux moderni includono una funzionalità chiamata "binfmt_misc" che consente di far registrare al sistema operativo degli emulatori, in modo da far coesistere programmi appartenenti ad architetture diverse nello stesso sistema operativo. Quando l'utente proverà ad eseguire un eseguibile non compatibile con l'architettura del processore, il kernel lo riconoscerà in automatico e tenterà di trovare un emulatore opportuno; in questo modo l'intervento dell'emulatore viene limitato al minimo necessario.

2.3 Librerie dinamiche e Linker

Per motivi di copyright e di prestazioni **[7]**, i sistemi Android usano un'implementazione della libreria C, chiamata Bionic, diversa da glibc, l'implementazione standard GNU. Oltre ad avere header diversi, Bionic non implementa del tutto fedelmente lo standard POSIX, né include alcune feature sperimentali presenti in glibc.

Tutti gli eseguibili linkati dinamicamente alla libreria glibc (o a una qualsiasi delle librerie che utilizzano la libreria C al loro interno) non possono, pertanto, essere eseguiti così come sono su Android.

A questo si aggiunge che, mentre in GNU/Linux il linker si trova all'interno di `/lib`, in Android si trova in `/system/bin`. Questo implica che non è possibile chiamare i programmi direttamente (ad esempio `./myprogram`) ma è necessario invocare esplicitamente il linker (`/system/bin/linker` `` `pwd`/myprogram``) o si riceverà un "Error: file not found".

Con i dovuti accorgimenti, le applicazioni GNU/Linux dinamiche possono essere ricompilate per funzionare con Bionic e cercare il linker nella posizione corretta.

In alternativa è possibile distribuire i programmi con una copia di glibc e delle altre librerie necessarie. Per indicare al linker di usare le librerie fornite anziché quelle di sistema, Linux mette a disposizione numerosi strumenti, tra cui le variabili shell LD_LIBRARY_PATH e LD_PRELOAD.

Bisogna tenere a mente, però, che nel momento in cui si sceglie glibc come libreria C a cui linkare il programma, non è più possibile linkarlo alle librerie di sistema Android, visto che sono compatibili con Bionic. Ciò comporta, ad esempio, che non sia possibile usare i driver closed-source di Android distribuiti come librerie dinamiche. In alcuni sistemi GNU/Linux

embedded viene usata una libreria chiamata libhybris **[54]** che, supportata da una patch kernel, rende possibile far coesistere librerie Bionic e glibc. Come detto in 2.1, però, i sistemi Android standard non prevedono la possibilità di inserire moduli nel kernel: usare libhybris è quindi fuori dallo scopo di questa tesi.

2.4 API OpenGL

Tipicamente le applicazioni su sistemi Unix-like, incluse quelle su dispositivi Apple, utilizzano la scheda grafica attraverso un'API ad alto livello scritta in C chiamata OpenGL. Di default, usano OpenGL sia app come Chrome, Firefox e Blender, sia componenti di sistema come Compiz, il gestore di animazioni delle finestre in molti sistemi GNU/Linux.

Esistono altre API come Vulkan o Metal (mantenuta da Apple), più difficili da programmare ma che offrono maggior controllo a basso livello e che pertanto trovano impiego nello sviluppo di videogiochi 3D complessi o altre applicazioni intensive.

Android non implementa l'intera API OpenGL, ma un subset chiamato OpenGL ES. Questo rende impossibile eseguire app GNU/Linux su Android sfruttando la scheda grafica.

Esiste la libreria gl4es **[55]** (succeduta a glshim) in grado di tradurre, eventualmente emulando, le chiamate di OpenGL 2 su schede grafiche che hanno solo driver OpenGL ES 2. Non esistono, però, piani per aggiornarla alle versioni di OpenGL successive alla 2, che è obsoleta dal 2008.

Una diversa soluzione, di più facile implementazione, è usare il software rendering, cioè implementare l'API OpenGL con il solo processore, senza ricorrere alla scheda grafica. Questo approccio consente di scrivere un software cross-platform indipendente dai driver della GPU. Lo svantaggio principale è che la CPU è nettamente più lenta nell'eseguire operazioni grafiche rispetto alla GPU, a volte anche di ordini di grandezza.

Fra le soluzioni per implementare OpenGL tramite software rendering, si segnala LLVMpipe **[8]**, un driver open-source disponibile in molte distribuzioni GNU/Linux tra cui Debian, Ubuntu e Arch Linux.

2.5 Rendering

Android si basa sul concetto di "surface", cioè superfici in cui è possibile renderizzare elementi grafici. Tali superfici vengono gestite da "SurfaceFlinger", che fa anche da window manager. È possibile interfacciarsi a SurfaceFlinger tramita la libreria HWUI (succeduta a Skia), che fornisce un'API per disegnare elementi 2D, o usando l'API di OpenGL ES.

I sistemi GNU/Linux prevedono invece un server grafico che implementa il protocollo X, la cui ultima versione X11 risale al 1987. Il server riceve richieste dalle varie applicazioni, i "client", e comunica con il sistema per renderizzare quanto ricevuto; allo stesso tempo si occupa di inoltrare input come tastiera e mouse. Negli anni, si sono consolidate estensioni al protocollo base, fra cui Xkb (per la gestione dei layout delle tastiere), Xinerama (per gli schermi affiancati) e GLX (per usare OpenGL su server X).

Un'interessante proprietà di X11 è che client e server sono indipendenti: le stesse applicazioni funzionano, infatti, con qualunque server che implementi il protocollo. Nei computer desktop, oltre all'implementazione di riferimento Xorg, trovano spazio anche Xvesa (meno performante, ma con

dimensioni estremamente ridotte, adatta ai sistemi live come Tinycore Linux) e XQuartz (l'implementazione gestita da Apple, attualmente installata sui dispositivi OS X).

Il protocollo è inoltre trasparente alla rete. È pertanto possibile eseguire un'applicazione su una macchina e farla renderizzare su un'altra macchina collegata tramite Internet o rete locale. In Unix questo si ottiene semplicemente impostando la variabile shell

DISPLAY="IP DELLA MACCHINA: ID DEL SERVER"

Tipicamente ID DEL SERVER = 0 per lo schermo principale.

Esistono anche soluzioni più avanzate per eseguire lo streaming, come ad esempio il software Xpra, in grado di salvare finestre in una sessione virtuale e trasmetterle ad altri dispositivi tramite SSH o su browser tramite WebSocket.

In generale, i server X non includono anche un window manager, che va installato separatamente. È il window manager a gestire come sono posizionate le applicazioni ed eventualmente, la grafica dei bordi e delle finestre. Nei sistemi GNU/Linux, i window manager possono essere di due tipi **[9]** o un misto tra i due.

- Stacking/floating: in cui cioè si hanno delle vere e proprie finestre, come avviene nella maggior parte dei computer desktop (Windows incluso), che possono essere mosse sullo schermo e sovrapposte fra di loro. Fra i più popolari GNOME, KDE, Xfce, Lxde, Openbox, Twm.
- Tiling: questi window manager dividono lo spazio in aree rettangolari, in cui è possibile visualizzare le applicazioni, senza che queste si sovrappongano. Questo consente di gestire le finestre più efficacemente anche senza l'uso del mouse. Fra i più popolari i3, bspwm, xmonad e Awesome. Questo approccio è simile a quello utilizzato in Android.

Sebbene non sia possibile interfacciare le applicazioni GNU/Linux direttamente con SurfaceFlinger, esistono dei server X11 per Android che sono in grado di ricevere richieste dai client X11 e renderizzarle usando come base una superficie.

- X11 Server: un server molto leggero scritto puramente in Java e pensato specificatamente per Android. Il progetto originale non riceve aggiornamenti dal 2014, ma un fork **[10]** continua a mantenere vivo il software correggendo vari bug e rendendolo compatibile con le nuove versioni di Android.
- XServer XSDL: porting di un server X11 per desktop che si basa su Simple DirectMedia Layer, una libreria cross-platform per gestire grafiche, audio e periferiche input. L'app è stata programmata prima che esistesse un porting ufficiale di SDL, pertanto usa un porting non-ufficiale realizzato dallo stesso programmatore e chiamato Commandergenius **[11]**.

In alternativa, è possibile usare Xvfb (X virtual framebuffer), un server X11 nato per il testing che implementa tutte le operazioni grafiche all'interno della memoria ram senza renderizzarle a schermo. Utilizzando applicazioni per il controllo desktop remoto è possibile trasmettere all'esterno il contenuto dello schermo emulato da Xvfb, mostrandolo quindi su una superficie. Tipicamente, per lo streaming, si utilizza il protocollo VNC **[12]** (Virtual Network Computing). Un popolare server VNC che consente di effettuare lo streaming di uno schermo X è x11vnc. Per visualizzare streaming VNC su Android esistono numerose app, fra cui RealVNC (ufficiale, closed source) e bVNC (open source). Esiste inoltre un client in grado di funzionare completamente da browser

chiamato noVNC.js **[70]** , che richiede però una versione di x11vnc abbastanza recente da supportare i WebSocket.

Non esiste, ad oggi, alcuna implementazione di X11 completa dell'estensione GLX funzionante su un generico Android. Questo sia perché i driver per la GPU sono compilati con Bionic (vedi 2.3), sia perché Android implementa solo il subset OpenGL ES (vedi 2.4), sia perché le "Surface" di Android consentono una sola istanza dell'oggetto "Context", mentre nei sistemi GNU/Linux è necessario un Context diverso per ciascuna finestra **[13]**.

Per completezza, si segnala che X11 non è l'unico protocollo esistente nel panorama GNU/Linux. Le applicazioni recenti vengono infatti compilate per supportare anche il protocollo Wayland, un'alternativa nata nel 2008. Wayland è meno stabile e ha meno supporto di X11, ma per la sua efficienza e sicurezza è impostato come predefinito in alcuni sistemi, come ad esempio Ubuntu 21. Non verrà trattato in questa tesi, sia perché X gode ancora di maggior compatibilità, sia perché Wayland richiede OpenGL già nel nucleo del codice, ed è quindi più difficile da emulare.

2.6 Audio

Anche il sistema audio usato dai sistemi operativi GNU/Linux segue la stessa filosofia server/client vista per il rendering nella sezione precedente. Il server, in questo caso, è chiamato PulseAudio e comunica con un modulo kernel chiamato ALSA che a sua volta comunica con i driver. Anche PulseAudio è trasparente alla rete e può essere sovrascritto usando una variabile d'ambiente (*export PULSE_SERVER*). A differenza di X, PulseAudio è però organizzato in moduli che vengono caricati durante l'esecuzione. È addirittura possibile evitare di eseguire tutti i moduli collegati ad ALSA e reindirizzare l'audio su un socket di rete, ad esempio usando in combinazione i moduli module-null-sink e module-simple-protocol-tcp. Il formato in cui i dati vengono inviati è progettato per essere di facile decodifica, tanto che esistono app Android in grado di riprodurre l'audio scritte in meno di 200 righe Java **[46]**.

2.7 App ed ecosistema software

Nei sistemi GNU/Linux gli eseguibili vengono compressi e raggruppati in "pacchetti". L'installazione e la gestione dei pacchetti è affidata a un programma apposito, detto "Package Manager", utilizzabile da linea di comando e diverso da distro a distro (le distro Debian-based usano dpkg+apt, altre opzioni sono yum e PacMan). Non esiste un formato universale per i pacchetti (apt usa il formato .deb, yum il formato .rpm, PacMan il formato .pkg).

Il Package Manager, oltre a consentire di installare pacchetti locali, offre l'accesso alle repository fornite dalla distro, ossia una collezione di pacchetti liberamente scaricabili, già compilati e testati. Vi è inoltre la possibilità di collegarsi a repository di terze parti, ad esempio per scaricare pacchetti proprietari o pacchetti sperimentali.

Ogni pacchetto ha un suo "Manifest", cioè un file che ne descrive il contenuto, la versione, l'autore e così via.

Tipicamente, gli eseguibili vengono compilati dinamicamente. Quando dipendono da librerie presenti nelle repository, anziché distribuire una copia delle librerie, si preferisce indicare nel Manifest tale dipendenza: durante l'installazione il Package Manager si occuperà automaticamente

di scaricare e installare i pacchetti delle librerie necessarie. Per i linguaggi interpretati come Java, Python e JavaScript/Node.js, è necessario indicare l'interprete fra le dipendenze.

Nel caso in cui un software non venga distribuito per il package manager della distro, l'utente può comunque installarlo tramite servizi alternativi, come ad esempio Snap **[56]**, Flatpak **[57]** o AppImage **[58]**, o compilarlo e inserirlo manualmente nelle cartelle di sistema: per tale scopo sono disponibili delle cartelle apposite, con suffisso "local". I linguaggi interpretati spesso presentano anche un proprio sistema di gestione dipendenze, alternativo a quello della distro, che, non essendo sottoposto all'approvazione dello staff che gestisce la distro, consente di avere aggiornamenti in tempi più rapidi.

I pacchetti che contengono un'applicazione con interfaccia grafica possono includere un file .desktop e un'icona, così da poter essere avviati cliccando sull'icona sul desktop.

In Android, l'utente non ha la possibilità di eseguire programmi binari, né ha modo di installare file binari direttamente. Gli unici pacchetti installabili, i pacchetti "apk", devono avere come punto di ingresso all'esecuzione un file costituito da bytecode Java. È quindi necessario scrivere parte dell'app nel linguaggio Java o in uno dei linguaggi in grado di produrre codice pseudocompilato Java (attualmente il sito ufficiale consiglia il linguaggio Kotlin).

Le apk devono essere generate tramite un insieme di programmi denominato "Android SDK" (Software Development Kit). Per ogni nuova versione di Android viene pubblicata una nuova versione dell'SDK. Le apk compilate con una versione dell'SDK sono in genere compatibili anche con i dispositivi in cui è installata una versione di Android successiva; esiste inoltre un insieme di librerie chiamato AppCompat per rendere retrocompatibili le apk fornendo un'implementazione delle nuove funzioni tramite le vecchie funzioni.

Per eseguire codice C all'interno di un'apk, è possibile compilarlo usando l'Android NDK (Native Development Kit) e usare il Java Native Bridge per interfacciarsi ad esso. Generalmente, tuttavia, è sconsigliato ricorrere direttamente a codice binario. Le app, infatti, non necessitano di interagire direttamente con le librerie C per via di una vastissima libreria di sistema scritta in Java che mette a disposizione numerosi binding ad alto livello.

Per eseguire codici in linguaggi interpretati diversi da Java, è necessario includere nell'app un interprete. Questa è, ad esempio, la soluzione utilizzata dal framework React Native che consente di scrivere app in JavaScript **[14]**.

Esiste, inoltre, la possibilità di eseguire file binari, che siano scaricati da Internet o inseriti come risorsa all'interno dell'apk, tramite istruzioni come Runtime.getRuntime().exec("command") e la classe java.lang.Process. Tuttavia, come verrà spiegato nella sezione 2.8, questa possibilità è stata estremamente ridotta per tutte le apk compilate con SDK 29 (Android 10) o successivi. È pertanto necessario compilare con SDK 28 le apk che intendono fare uso di questa funzionalità.

Per sviluppare le app viene fornito un editor ufficiale, chiamato Android Studio, già configurato per supportare Kotlin e il build system Gradle.

Generalmente, le app vengono gestite e aggiornate dal servizio di distribuzione digitale Google Play Store, presente come app di sistema nei dispositivi, ma è possibile installarle manualmente o da altri servizi come F-Droid. Google Play non consente però di distribuire apk che eseguono o installano codice Java o binari arbitrari **[15]**; tale clausola fa sì che tutte le app disponibili su Google Play non possano auto-aggiornarsi, perché installerebbero codice Java arbitrario.

Un'altra clausola del Google Play Store, inoltre, pone severe limitazioni sulla versione dell'SDK da utilizzare per compilare le apk, impedendo che siano pubblicate apk compilate con versioni superate dell'SDK come la 28. Per questi motivi, molte delle apk che saranno esaminate nel capitolo 3 saranno disponibili sono tramite servizi alternativi.

2.8 Permessi e accesso alle risorse

Tutti i sistemi Linux (e più in generale quelli Unix-like) prevedono la presenza di più utenti, di cui uno, chiamato sempre "root", svolge il ruolo di amministratore. Gli utenti possono appartenere a uno o più gruppi, così da ottenere i privilegi relativi a quel gruppo. Ad esempio, gli utenti appartenenti al gruppo sudoers, hanno il permesso di utilizzare il programma "sudo" che consente loro di diventare temporaneamente l'utente root.

Ogni file presente ha un proprietario "owner" e un gruppo proprietario "group-owner". Per ciascuna delle tre operazioni di lettura/scrittura/esecuzione, ai file sono associati tre bit che indicano se tale operazione è concessa a tutti, all'owner o al group owner, per un totale di nove bit.

Nei sistemi GNU/Linux, normalmente non è necessario chiedere autorizzazioni particolari per accedere alle periferiche o scrivere nei file dell'utente. Esistono tuttavia distro e metodi di distribuzione, come Snap **[56]**, in cui i programmi contenuti nei pacchetti sono eseguiti in ambiente protetto, e devono dichiarare nel manifest le risorse di cui hanno bisogno.

Android utilizza una struttura simile, riservando l'utente root alle sole app/programmi di sistema. L'utilizzatore del dispositivo non può usufruirne, a meno di compiere una procedura di manomissione del dispositivo chiamata "rooting" che comporta, oltre alla perdita della garanzia del dispositivo, la totale formattazione dei dati all'interno (e quindi esula da questa tesi).

Le normali app, vengono eseguite senza privilegi di amministratore e in un ambiente protetto, in cui hanno accesso solo a determinate risorse. Per poter effettuare alcune operazioni, come leggere i contatti nella rubrica dell'utente, usare le fotocamere, collegare il dispositivo alla rete wifi, è necessario dichiararle nel Manifest dell'app. A partire da Android 6.0, alcune autorizzazioni, ritenute particolarmente pericolose ("dangerous"), devono essere confermate dall'utente anche durante l'esecuzione: l'app potrà chiedere al sistema di mostrare una finestra popup in cui si chiede all'utente se accetta che l'app compia una certa operazione, come ad esempio registrare tramite il microfono.

Analogamente, le app hanno a disposizione capacità limitate di eseguire processi in background, a meno che non informino l'utente della propria presenza usando una notifica prioritaria **[16]**.

Ogni app ha a sua disposizione uno spazio privato all'interno del filesystem, in cui può leggere, scrivere ed eseguire senza bisogno di autorizzazioni. Esiste poi uno spazio esterno condiviso e accessibile all'utente, in cui le app possono scrivere, a patto di chiedere un'autorizzazione "dangerous". I file di questo spazio non hanno un proprietario e i bit di autorizzazione come nei tradizionali sistemi Linux: da un lato ciò implica che possono essere scritti e letti da tutte le app con l'autorizzazione per accedere allo storage esterno, dall'altro non possono però essere eseguiti. Questo impone tassativamente che ogni programma GNU/Linux, per poter essere eseguito dall'app, debba essere installato nello spazio interno. Per ragioni di sicurezza, Android non consente più di eseguire binari nello spazio interno di apk compilate con Android SDK 29 o successivi: questo vuol

dire che, se si sceglie di compilare apk con la versione attuale dell'SDK, non esiste più alcun modo per installare ed eseguire binari.

Si segnala. inoltre, che, a partire da Android 11, le app hanno una cartella in cui scrivere liberamente anche nello spazio condiviso esterno, e Google Play diverrà più selettivo nel pubblicare app che richiedono l'accesso globale ai file.

2.9 Comunicazione inter-processo tra app

Tradizionalmente, le app Linux utilizzano un meccanismo chiamato Unix domain Socket per consentire la comunicazione fra eseguibili diversi. Gli Unix domain Socket sono canali, gestiti interamente dal kernel, che vengono identificati da un path, come se fossero file a tutti gli effetti. È possibile evitare la creazione di un file usando path speciali, slegate dal filesystem perché gestite direttamente dal kernel; in questo caso si parla di Abstract Namespace Unix Socket **[17]**.

Molte app si affidano, in alternativa, a protocolli di rete usati in locale, ad esempio i socket TCP. Il kernel Linux è in grado di gestire efficacemente queste connessioni, rimappandole prima che raggiungano la rete, ottenendo così ottime prestazioni, paragonabili a quelle degli Unix domain Sockets (anche se più lente **[18]**). Il vantaggio dei protocolli di rete è che sono universali, più supportati e che consentono al programma di interfacciarsi anche con macchine collegate tramite rete locale o Internet.

Android implementa sia gli Unix domain Socket che i socket TCP, ma aggiunge nuovi metodi, integrati con la libreria Java di sistema, per facilitare la comunicazione tra app, come ad esempio:

- Intent, delle chiamate che le app effettuano indicando più o meno genericamente un'azione da delegare; il sistema operativo si occuperà di trovare l'app più adatta a svolgere l'azione. Ad esempio, nel caso in cui l'app chieda di aprire un documento pdf, il sistema operativo sceglierà il lettore pdf predefinito o chiederà all'utente quale lettore pdf preferisce. Per poter ricevere un Intent, l'app deve dichiarare di essere in grado di gestirlo nel Manifest. È possibile indirizzare un Intent verso un'app precisa, specificandone l'ID. Tipicamente, gli Intent vengono chiamati da codice Java usando la libreria di sistema, esiste tuttavia un programma chiamato "am" (Activity Manager), con cui lanciare Intent da linea di comando.
- Broadcast, simili agli Intent, ma vengono ricevuti da tutte le app compatibili anziché da una singola app. I Broadcast vengono usati dal sistema stesso per comunicare alle app che si è verificato un determinato avvenimento. Ad esempio, il Broadcast CONNECTIVITY_CHANGE viene inviato ogni volta che l'utente modifica la connessione Internet. Come per gli Intent, è possibile lanciare Broadcast usando am.
- Il "binding", un'operazione con cui due componenti creano un legame e possono richiamare l'uno le funzioni Java dell'altro. Per consentire il binding fra componenti di app diverse, Android mette a disposizione l'AIDL (Android Interface Definition Language) e l'interfaccia Messenger, il cui ruolo è simile a quello dei file header delle librerie C. AIDL offre più flessibilità, ma Messenger è più facile da implementare.

2.10 Estetica e User Experience

Le distribuzioni GNU/Linux non hanno un'estetica precisa, lo testimonia il fatto che il solo Ubuntu venga distribuito in 7 diversi flavour ufficiali **[19]**, ciascuno con un'estetica diversa. Fra le librerie di componenti grafici, si citano GTK (open-source) e Qt (proprietaria, ma utilizzabile liberamente per progetti open-source). Tipicamente, l'utente si aspetta che, come su Windows o Mac Os, in uno dei quattro lati dello schermo ci sia una barra delle applicazioni in cui è possibile vedere le app in esecuzione e avviarne di nuove. Le app vengono progettate per schermi medi o grandi, si presuppone che l'utente navighi usando mouse e tastiera. Come già detto in 2.5, solitamente, le app vengono gestite da un window manager a finestre. L'utente si aspetta che su ogni finestra siano presenti i pulsanti per ridimensionarla, massimizzarla o chiuderla.

Al contrario dei sistemi GNU/Linux, Android presenta delle linee guida ufficiali su come realizzare lo stile grafico delle app. Attualmente, è in vigore l'estetica "Material Design", con la sua evoluzione "Material Design You", concentrata sulla personalizzazione dell'utente, prevede, fra le altre cose, che lo sfondo impostato modifichi il tema di sistema.

Come detto in 1, le soluzioni di emulazione attuali non compiono alcuno sforzo per adattare l'estetica dei sistemi GNU/Linux ad Android, visto che questa incompatibilità non è un aspetto critico.

Per un corretto funzionamento delle app, va tuttavia assicurato che l'utente sia in grado di inserire gli stessi input che avrebbe potuto inserire su PC, dove sono sempre disponibili sia mouse che tastiera, e in cui la tastiera presenta anche tasti come Ctrl o Esc. Ciò avviene tipicamente o fornendo una tastiera a schermo sempre attiva, o includendo un pulsante per mostrare/nascondere la tastiera a schermo e dei pulsanti con i tasti da integrare.

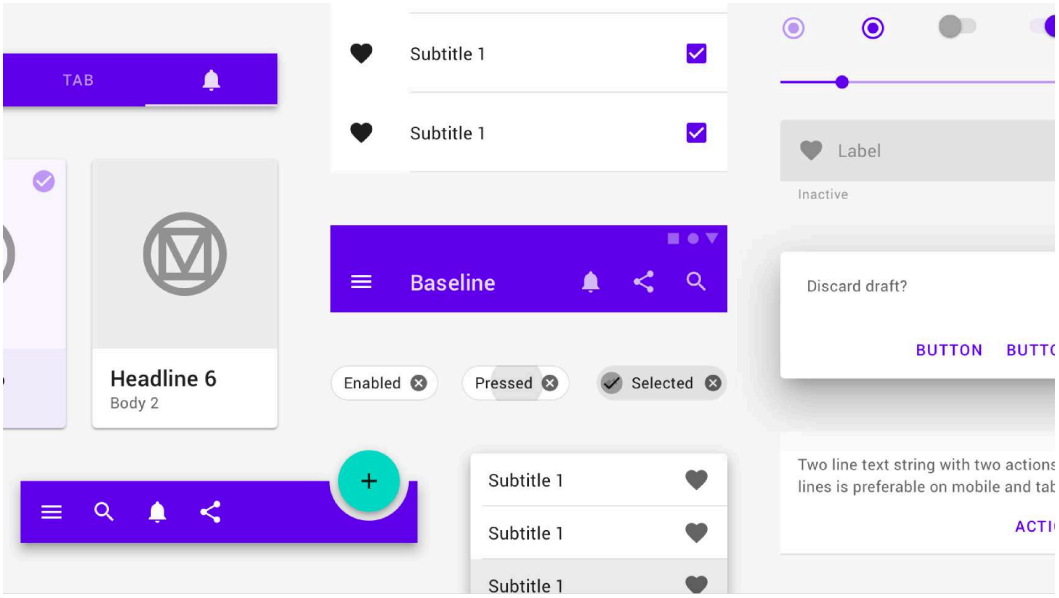

Figura 2: *Componenti app realizzati con l'estetica Material Design*

3. Soluzioni esistenti

In questo capitolo si esamineranno le più popolari tra le soluzioni esistenti per eseguire programmi GNU/Linux su Android, raggruppandole in due categorie. Verranno considerate le app Android:

- IBochs, Limbo (basate su emulatori),
- Termux, AnLinux, Andronix, UserLAnd e Debian noroot (native o basate su meccanismi Chroot).

Ciascuna app verrà presentata individualmente, indicando le strategie che utilizza e come risolve le incompatibilità enunciate nel capitolo 2. Verranno quindi effettuati dei test per confrontare le app sia dal punto di vista qualitativo che quantitativo; verranno confrontate prestazioni e spazio utilizzato su disco.

3.1 Soluzioni basate su emulatori (hosted hypervisor)

In queste soluzioni, il sistema GNU/Linux non viene eseguito direttamente dal processore, ma viene letto da un programma chiamato "emulatore" o “ipervisore” (hypervisor **[20]**), in grado di interpretare il codice macchina. Non vengono eseguite le sole app di interesse, ma il sistema operativo nella sua interezza, kernel incluso. Per consentire al sistema operativo emulato di comunicare con l’esterno, si agisce a basso livello, usando periferiche virtuali.

Il vantaggio principale di questo approccio è che l'emulatore ha completa libertà su come gestire il sistema operativo in esecuzione. Ad esempio, quando il programma vuole scrivere un file su disco, l'emulatore rimappando la chiamata fa sì che scriva all'interno di un file disco virtuale, senza quindi intaccare il filesystem Android. Allo stesso modo è possibile eseguire sistemi operativi di qualsiasi architettura (x86 inclusa), usare kernel diversi e avere controllo maggiore sui programmi.

Vi sono tuttavia numerosi svantaggi.

- È evidente che il programma emulato avrà prestazioni ridotte, in quanto per ogni istruzione presente nel programma, l'emulatore dovrà eseguire più istruzioni. È possibile mitigare il problema grazie a tecniche come la compilazione Just-In-Time, ma sempre con risultati poco soddisfacenti. Per consentire di ottenere prestazioni vicine a quelle originarie, la maggior parte dei processori include un modulo hardware in grado di svolgere la funzione di hypervisor (si parla di hypervisor hardware). In questo modo le operazioni più onerose vengono delegate all’elettronica e si riescono ad ottenere velocità paragonabili a quelle dei sistemi nativi, non emulati. La principale limitazione di questo approccio è che il processore è in grado di fare da hypervisor solo per la propria architettura: non è possibile, quindi, usare l’hardware hypervisor per sistemi x86 che emulano ARM e viceversa.
- L'emulatore deve essere in grado di riprodurre fedelmente l'architettura del sistema che sta eseguendo, un compito non banale (i soli processori x86 hanno oltre 1500 **[21]** istruzioni).
- Per emulare un programma, è necessario avviare un intero kernel e sistema operativo, spesso con tempi molto lunghi.
- Per ottenere prestazioni migliori, l'emulatore deve allocare in anticipo le risorse a disposizione del sistema. Si tratta di risorse non trascurabili, visto che corrispondono al necessario per eseguire un intero sistema operativo.

3.1.1 IBochs (Bochs)

IBochs **[59]** è un porting proprietario dell'emulatore open-source Bochs, scritto in C puro.

L'app consente di avviare sistemi sia partendo da file disco virtuale, che da file .iso (il formato può popolare per distribuire sistemi desktop, visto che può essere masterizzato direttamente su disco). L'app è disponibile sia su architetture x86 e x86/64 che su ARM e ARM64, ma è in grado di emulare solo sistemi x86 e x86/64. Consente di emulare la presenza di tastiere, mouse e componenti vari, incluso uno schermo che poi è possibile visualizzare all'interno dell'app. Per renderizzare il contenuto dello schermo, viene usato il porting ufficiale di SDL.

Oltre allo schermo, l'interfaccia grafica include una tastiera virtuale completa di tutti i tasti, inclusi quelli tipicamente non presenti su cellulare. Non è però possibile usare la propria tastiera personale.

Esiste anche un altro porting di Bochs **[22]**, ma non verrà trattato in questa tesi, perché, sui dispositivi considerati, non è stato in grado di avviare correttamente nessuno dei sistemi testati.

3.1.2 Limbo (Qemu)

Limbo PC Emulator **[60]** è un porting di una parte dell'emulatore QEMU **[61]** (Quick EMUlator).

Il vero punto di forza di Limbo è che si basa su QEMU, un software estremamente versatile, che può essere usato in numerosi scenari diversi, anche al di fuori dell'emulazione hypervisor, come vedremo in 3.2.

Come per 3.1.1, l'app consente di avviare sistemi partendo sia da file disco virtuale che da file .iso, e consente di emulare la presenza di tastiere, mouse e altri componenti.

Contrariamente a IBochs, supporta l'emulazione di sistemi ARM e ARM64 (e molti altri) oltre che x86 e x86/64. Per non appesantire il file apk, l'app è stata spezzata in diversi file apk, uno per ciascuna architettura disponibile.

In aggiunta alle funzioni disponibili su IBochs, Limbo possiede multipli motori grafici e consente anche lo streaming dello schermo virtuale tramite protocollo VNC, così da consentire all'utente di visualizzare il sistema operativo emulato tramite il Client VNC che preferisce.

Si segnala che, installando un apposito modulo kernel (KVM **[62]**, kernel-based virtual machine), Limbo è in grado di accedere all'hardware hypervisor, e ciò consente di eseguire sistemi operativi (non solo Linux, ma anche Windows) con prestazioni elevatissime, paragonabili a quelle native. Per i motivi enunciati in 2.1, QEMU+KVM non verrà trattato in questa tesi; tuttavia, il confronto fra le prestazioni di QEMU e QEMU+KVM sarà utile per stimare in prima approssimazione, quanto incide l'emulazione software nelle prestazioni del sistema operativo.

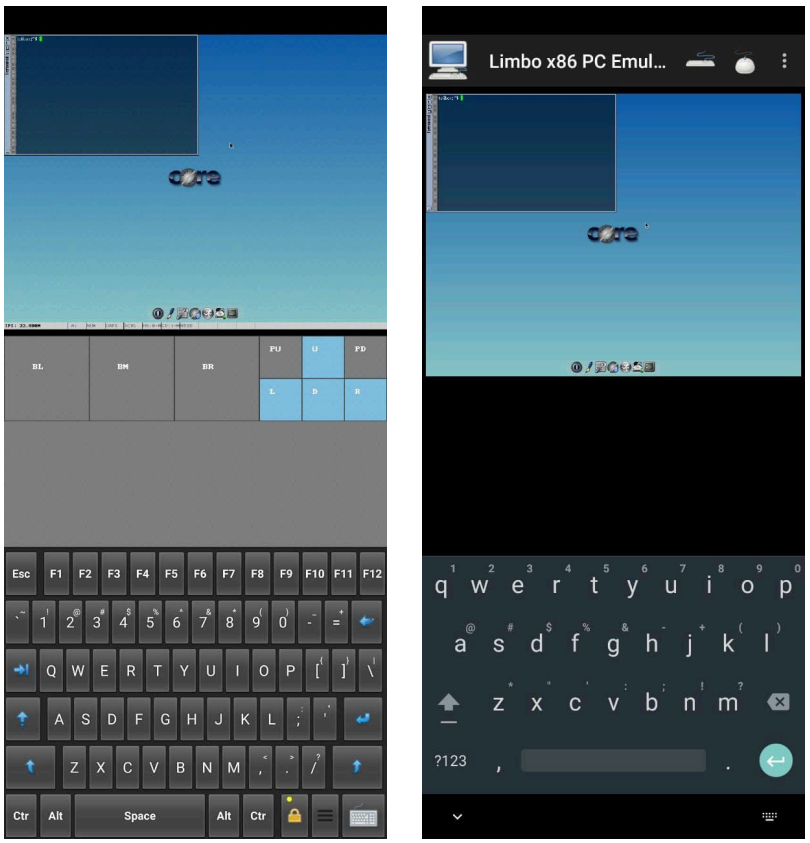

Figura 3: *Confronto tra IBochs (sinistra) e Limbo (destra). Mentre Limbo consente di usare la propria tastiera, IBochs ne implementa una all'interno della schermata.*

3.2 Soluzioni native o basate su meccanismi Chroot (OS-Level Virtualization)

In queste soluzioni, all'interno della cartella privata dell'app viene scaricato l'insieme di programmi e librerie che compongono una distribuzione GNU/Linux mantenendo del tutto o in parte la struttura del filesystem (a questo insieme di file generalmente si dà il nome di root filesystem, abbreviato "rootfs"). Poi, attraverso vari meccanismi, si rimappano tutte le chiamate a sistema in modo che siano compatibili con Android e, anziché usare i file di sistema Android, usino quelli nella cartella privata dell'app. Ciò può essere fatto sia intervenendo a priori sul codice sorgente (e quindi i binari ottenuti saranno in tutto e per tutto nativi e non avranno alcuna perdita di performance), sia durante l'esecuzione attraverso funzionalità del kernel. In questi casi è improprio parlare di "emulazione", ma si preferisce il termine "virtualizzazione" **[23]**, in quanto emulazione sottintende che si stia riproducendo il comportamento di un componente hardware.

Il kernel Linux include un comando specifico per rimappare le chiamate durante il runtime: "chroot".

Chroot consente di eseguire programmi usando come base una sottocartella anziché la cartella base del sistema. Ad esempio, se dopo aver usato `chroot /home/myfolder` si avvia un programma che scrive un file su `/tmp/file.txt`, grazie a chroot il file verrà invece scritto su `/home/myfolder/tmp/file.txt`. Allo stesso modo, se il programma tenta di leggere la libreria `/lib/mylib.so`, caricherà invece `/home/myfolder/lib/mylib.so`.

I file all'interno di un ambiente chroot (chiamato chroot jail), sono isolati e non possono accedere al resto del dispositivo. È possibile rendere disponibili cartelle esterne tramite il comando "mount".

Il comando chroot, così come il comando mount, richiede i privilegi root per poter essere usato direttamente (e quindi il suo uso diretto non sarà trattato in questa tesi), ma esistono delle alternative che, sfruttando gli strumenti di debug del kernel (vedi 2.1), permettono di ottenere le stesse funzionalità senza i permessi di amministratore.

La più comune è il software PRoot **[24]**, in grado non solo di emulare chroot, ma anche di fornire le funzioni svolte dal comando mount e dal modulo binfmt_misc (vedi 2.2). A differenza di chroot, PRoot non garantisce il completo isolamento dei programmi (un programma malevolo potrebbe effettuare chiamate a sistema per uscire dalla PRoot jail), ma ciò non è un problema perché tali programmi sono comunque sottoposti alle numerose limitazioni imposte da Android. Fra le funzionalità offerte da da PRoot vi è anche la possibilità di fingere che l'utente abbia i permessi di root senza che li abbia davvero; ciò è utile perché alcuni programmi (come il package manager apt) rifiutano a priori di avviarsi se non sono eseguiti con i privilegi di amministratore. Usando PRoot come binfmt_misc e installando l'emulatore QEMU è inoltre possibile eseguire programmi compilati per architetture diverse da quella del dispositivo, anche se con prestazioni più scarse.

In generale, il vantaggio di usare sistemi basati su meccanismi chroot, è che non necessitano di emulare una cpu o un intero kernel né di intervenire su molte delle istruzioni eseguite. Per tale ragione, questi approcci (chiamati OS-Level Virtualization) sono ritenuti i più veloci, anche comparati con gli hypervisor hardware **[25]**.

La possibilità di eseguire singoli binari senza bisogno di avviare il sistema operativo (impiegando tempo e risorse computazionali), inoltre, rende queste soluzioni utili per distribuire singole applicazioni in ambienti perfettamente riproducibili; ciò ha fatto nascere software come Docker

(che però richiede permessi di root, e quindi non sarà utilizzabile per i fini di questa tesi), per semplificare e automatizzare la distribuzione e la gestione di applicazioni eseguite in chroot.

Lo svantaggio principale è che l'assenza di un hypervisor limita le possibilità di emulazione: ad esempio, non è possibile avviare sistemi GNU/Linux incompatibili con il kernel Android. Queste soluzioni, inoltre, richiedono di eseguire dei file binari all'interno dello spazio privato dell'app: ciò comporta, come spiegato in 2.7 e 2.8, che non sia possibile compilarle con le versioni più recenti dell'Android SDK e quindi distribuirle sul Google Play Store.

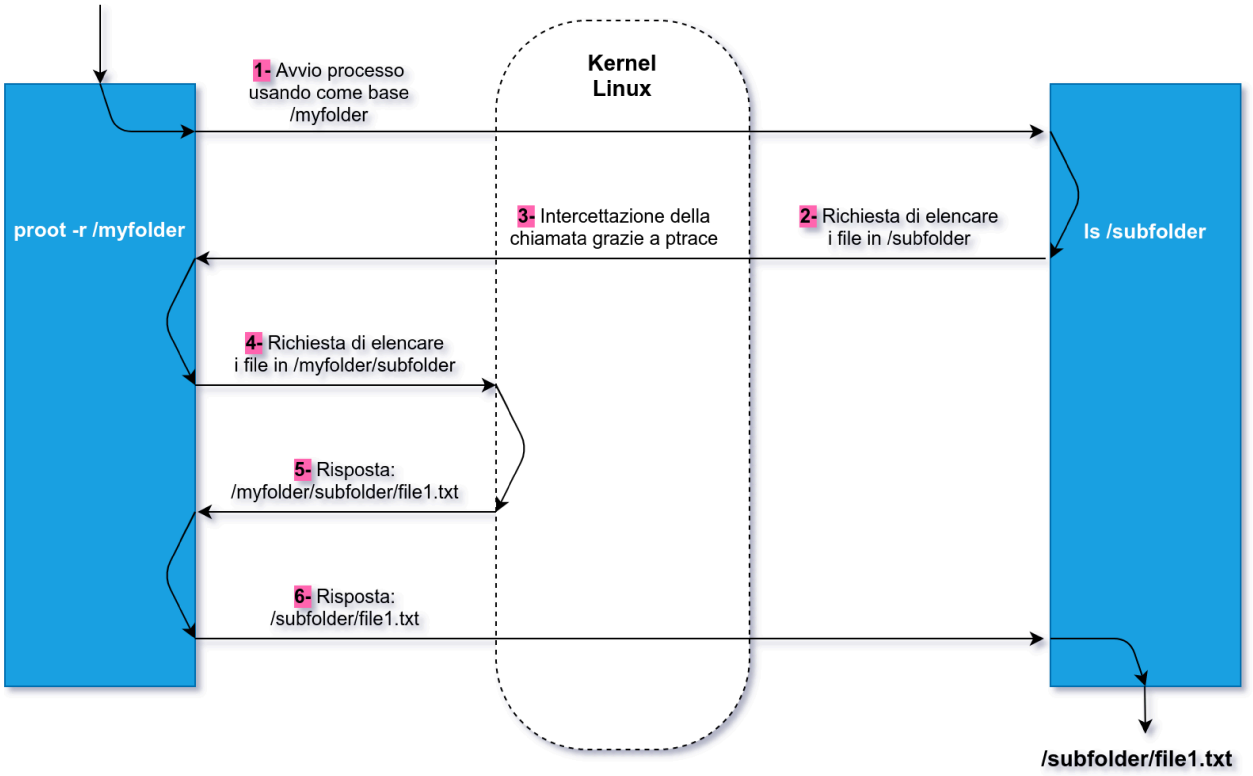

Figura 4: *Esecuzione di ls (list file) utilizzando PRoot per rimappare la chiamata*

3.2.1 Termux

Termux **[63]** un'app per installare ed eseguire, tramite un terminale, porting delle principali applicazioni a linea di comando GNU/Linux; include il package manager pkg (basato su apt **[26]**) e delle repository specifiche, create appositamente per l'app.

Per poter funzionare con Termux, i programmi vengono patchati opportunamente e ricompilati in modo da essere compatibili con Android. Le modifiche consentono ai binari di:

- adeguarsi al filesystem Android, in particolare limitare la propria lettura e scrittura alla sola cartella privata di Termux (vedi 2.8);
- funzionare con un solo utente e senza privilegi di root (vedi 2.8);
- cercare il linker in `/system/bin` e essere linkati con Bionic anziché glibc (vedi 2.3).

Le modifiche effettuate rimuovono la necessità di eseguire i binari all'interno di una chroot jail. In questo senso, si può dire che più che un sistema GNU/Linux completo, Termux è da considerarsi un "Linux Environment" che coesiste nativamente con Android **[27]**.

Scaricando un'app ausiliaria, i programmi e gli script eseguibili all'interno di Termux possono interfacciarsi con il sistema Android grazie a un set di eseguibili, chiamato termux-api. Questi programmi comunicano con l'app usando gli Abstract Namespace Unix Socket (la path viene comunicata all'app tramite un Broadcast lanciato da am, vedi 2.9) e consentono di effettuare operazioni come verificare lo stato della batteria, scattare una fotografia, leggere gli sms ricevuti.

Oltre a varie utility da linea di comando, Termux fornisce una serie di applicazioni con interfaccia grafica, organizzate in una repository a parte chiamata x11-repo, che include anche Xvfb e x11vnc. Termux non presenta un modo per visualizzare un server X all'interno dell'app; per poter interagire con i programmi a interfaccia grafica è quindi necessario scaricare un server X o un client VNC separatamente.

Le repository di Termux contengono solo una parte ridotta del software GNU/Linux **[28]** esistente, in particolare mancano molti programmi grafici, fra cui LibreOffice, Audacity, Chromium e Firefox. Vi è però la possibilità di installare PRoot e avviare un chroot jail, in modo da usare un sistema come Debian, Alpine o Arch Linux. Questo può essere fatto anche grazie all'aiuto del pacchetto proot-distro, che consente di scaricare proot e dei rootfs minimali (sprovvisti di interfaccia grafica) delle distro più popolari. proot-distro **[64]** fornisce anche una serie di comandi che aiutano con l'aggiornamento di tali distro e il loro backup, ma non include dei comandi per semplificare l'installazione di ambienti grafici.

Termux è disponibile sia sul Google Play Store che sul market alternativo per software open-source F-Droid, tuttavia la versione su Google Play è obsoleta (non può più essere aggiornata per le restrizioni indicate in 2.7). In particolare, l'app su Google Play è impostata su una repository non più attiva; se l'utente non interviene e la modifica, non potrà installare alcun pacchetto **[29]**.

3.2.1.1 AnLinux e Andronix

Per rendere più facile il processo di installazione del rootfs di una distro e di PRoot all'interno di Termux, esistono alcune app che guidano passo passo l'utente. Le due più popolari disponibili sono AnLinux e Andronix.

Entrambe forniscono un'interfaccia da cui è possibile scegliere la distro desiderata, entrambe presentano dei comandi pronti per essere copia-incollati all'interno di Termux ed entrambe hanno un proprio mirror da cui prendere la base dei rootfs.

Entrambe supportano le principali distro linux (Debian, Ubuntu, Arch Linux e Alpine) e la possibilità di scegliere il proprio window manager.

AnLinux è open-source (e quindi disponibile anche su F-Droid **[30]**), mentre Andronix è open-source solo parzialmente **[31]**.

Andronix include, oltre alle versioni base delle distro, delle versioni closed source acquistabili che, secondo gli sviluppatori, sono state ottimizzate e testate per funzionare meglio su vari dispositivi Android. Queste distro non verranno trattate in questa tesi perché non è chiaro in cosa consistono le modifiche apportate: alcuni test preliminari non sono riusciti a rilevare alcuna differenza significativa di prestazioni rispetto, ad esempio, ai sistemi PRoot eseguiti da UserLAnd (vedi 3.2.2), e non è stato possibile eseguire analisi più accurate per via della natura closed-source del progetto.

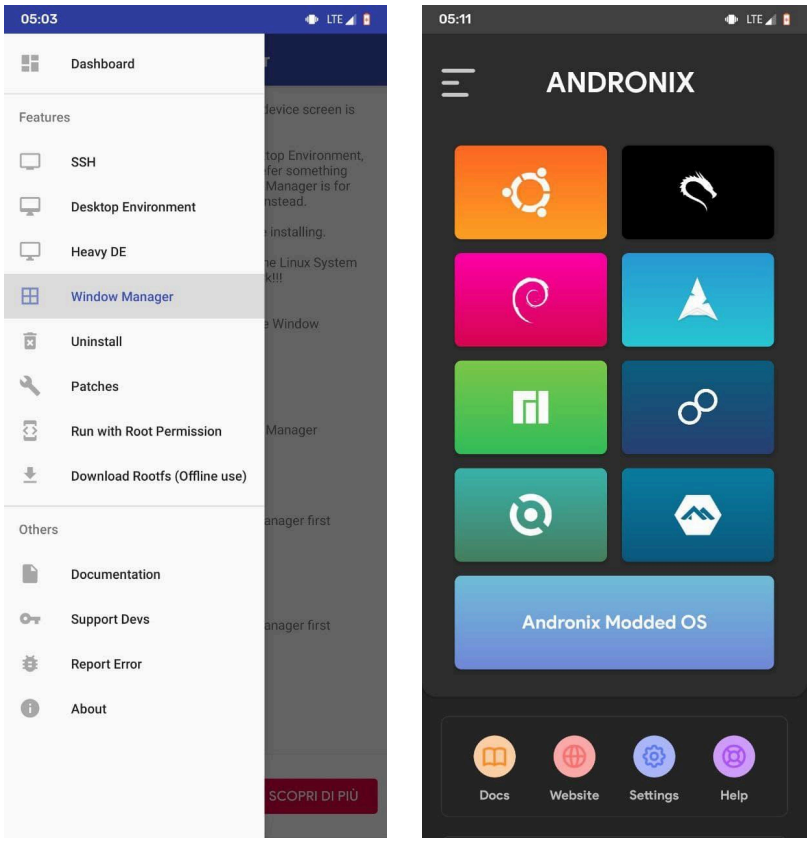

Figura 5: *Confronto tra le schermate con cui si aprono AnLinux (sinistra) e Andronix (destra). Si nota che Andronix è nettamente più curato dal punto di vista grafico.*

3.2.2 UserLAnd

UserLAnd **[65]** è un'app che permette di installare i rootfs delle distro più popolari (fra cui Alpine, Debian, Ubuntu e Arch Linux) e avviarli usando PRoot.

Tutte le distro vengono scaricate da un sito apposito, già preconfigurate in modo da avere installato all'interno Xvfb e x11vnc (per lo streaming dello schermo tramite protocollo VNC), il window manager minimale twm e un terminale.

È possibile avviare interagire con ciascuna distro in tre modi:

- tramite un'interfaccia a terminale inclusa nell'app, identica a quella di Termux perché basata sul codice sorgente di quest'ultima **[32]**;
- avviandola in modo che le applicazioni vengano renderizzare all'interno del server X XServer XSDL (questa opzione è disabilitata per i dispositivi con Android 10 o successivi, in quanto non più funzionante);
- visualizzandola tramite un client VNC, non incluso nell'app.

Fra i vantaggi dell'app, ci sono sicuramente la facilità di utilizzo e l'estrema stabilità, visto che tutte le distro vengono distribuite già preconfigurate e testate.

Ci sono però anche delle criticità.

- L'app non è in grado di gestire sessioni multiple con VNC, inoltre presenta dei bug se si avviano più sessioni usando l'interfaccia a terminale inclusa.
- Avviare una sessione VNC è particolarmente scomodo: innanzitutto richiede di installare un'app esterna. Inoltre per impedire la connessione da parte di altri programmi non autorizzati, l'app protegge tale connessione con una password che l'utente deve inserire a ogni nuova sessione.
- Il window manager preinstallato, twm, è estremamente leggero, ma anche estremamente scomodo da utilizzare, soprattutto usando il touch screen: deficita, ad esempio, di una barra delle applicazioni e richiede di tenere premuto il tasto destro del mouse per mostrare/nascondere le schermate.
- L'app fornisce un solo rootfs per distro; è quindi obbligatorio scaricare i pacchetti per la trasmissione vnc anche nel caso in cui si volessero usare solo le funzionalità a linea di comando.

3.2.3 Debian noroot

Scritta dallo stesso sviluppatore di XServer XSDL, Debian noroot **[66]** consente di utilizzare una distro Debian (ambiente grafico Xfce) eseguita tramite PRoot e renderizzata tramite XServer XSDL.

Debian Noroot rappresenta un'alternativa alle soluzioni basate su Xvfb e x11vnc rispetto a cui, potenzialmente, potrebbe ottenere risultati migliori: è l'unica app a integrare una vero e proprio porting del server X e ciò dà modo di ottenere ottime performance e numerose funzionalità native, ad esempio ha un supporto per l'audio già integrato nel server X **[33]**. Tuttavia, questo punto di forza è anche il punto di debolezza: XServer-XSDL, per la mole di codice da cui è composto, richiede molto sforzo per essere mantenuto e adattato agli aggiornamenti che Android riceve nel tempo. Se storicamente Debian Noroot è stata per lungo tempo l'opzione migliore per installare Debian, oggi, vista la mancanza di supporto dall'autore originale, è diventata inutilizzabile sui dispositivi più recenti: l'app presenta numerosi glitch grafici, spesso non riesce a gestire gli input da tastiera e alcuni programmi non si avviano. Pertanto, non verrà trattata oltre in questa tesi.

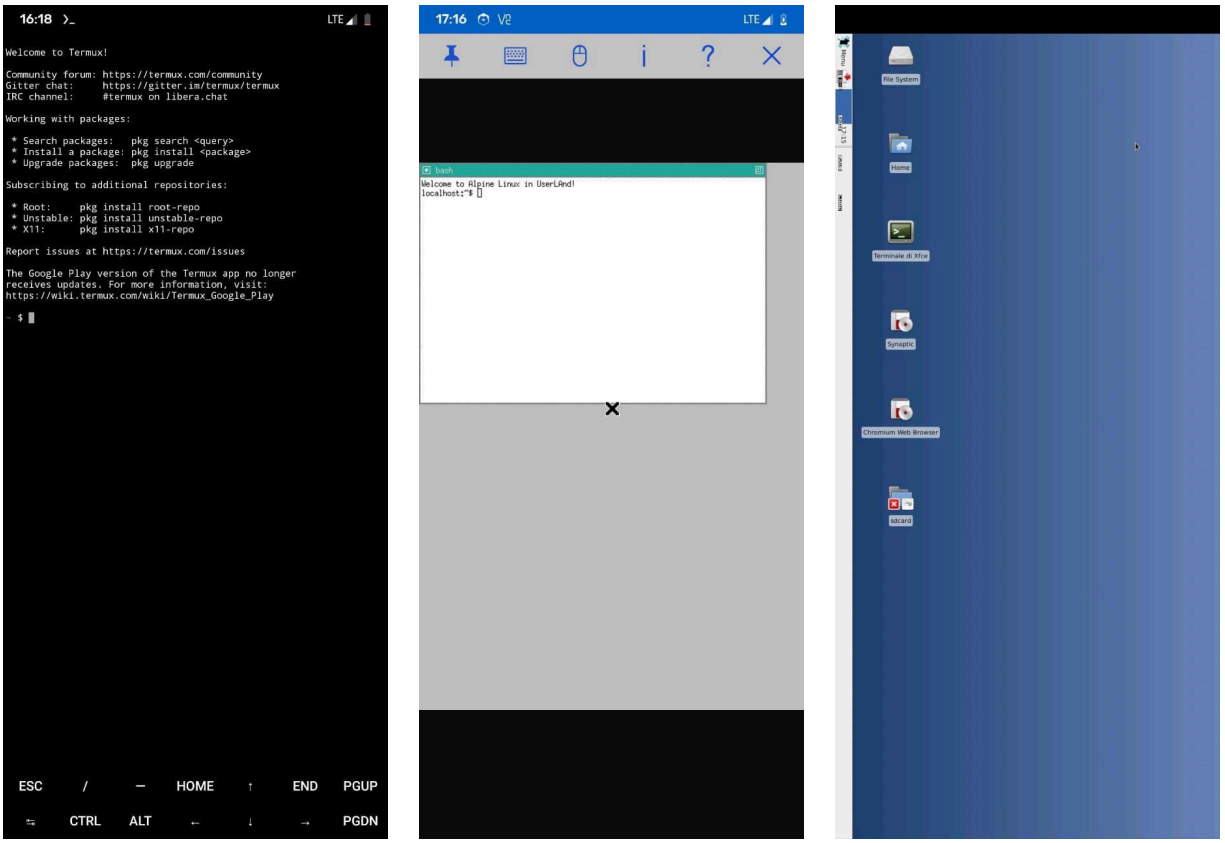

Figura 6: *Confronto tra tre diversi modi di visualizzare i programmi GNU/Linux: l'emulatore di terminale fornito da Termux (sinistra), una distro UserLAnd visualizzata usando RealVNC (centro) e il server X (XServer-XSDL) integrato in Debian noroot (destra).*

3.3 Test comparativi

Questa sezione contiene dei test svolti appositamente per questa tesi. Per rendere più immediato il confronto, si è scelto di riportare sia i dati delle app analizzate in questo capitolo (sezioni 3.1 e 3.2), sia i risultati raggiunti dalla soluzione proposta nel capitolo 4.

3.3.1 Funzionalità presenti

Per il confronto sono state considerate le seguenti caratteristiche:

1. Funziona sulle versioni recenti di Android
2. Consente di eseguire programmi GNU/Linux a linea di comando
3. Consente di eseguire semplici programmi grafici
4. Consente di eseguire programmi grafici complessi, come il browser Chromium
5. Include il supporto audio
6. Include supporto per l'accelerazione hardware
7. Funziona a velocità native o quasi native (senza bisogno di root)
8. Permette l'installazione di distro GNU/Linux complete
9. È compatibile con i file .iso (kernel incluso)
10. Funziona ed è in grado di visualizzare programmi grafici senza bisogno di installare app esterne/installando 1 app/installando 2 app
11. Fornisce un'API per interagire con le librerie di sistema Android
12. Ha un'interfaccia grafica espandibile tramite programmi esterni

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
IBochs	Sì	Sì	Sì	Sì	?*	No	No	Sì	Sì	0	No	No
Limbo	Sì	Sì	Sì	Sì	?*	No	No	Sì	Sì	0	No	No
Termux (senza PRoot)	Sì	Sì	Sì	No	Sì	No	Sì	No	No	1	Sì	No
Termux (proot-distro)	Sì	Sì	Sì	Sì	Sì	No	Sì	Sì	No	1	No	No
Termux (AnLinux o Andronix)	Sì	Sì	Sì	Sì	Sì	No	Sì	Sì	No	2	No	No
UserLAnd	Sì	Sì	Sì	Sì	No	No	Sì	Sì	No	1	No	No
Debian noroot	No	Sì	Sì	Sì	Sì	No	Sì	Sì	No	0	No	No
Unyw (soluzione proposta)	Sì	Sì	Sì	Sì	Sì	No	Sì	Sì	No	0	Sì	Sì

* in entrambi i casi, sono presenti delle opzioni per regolare l'audio e la documentazione sembra indicare che il supporto sia presente, tuttavia nonostante numerosi tentativi non si è riusciti ad attivarlo con nessuna delle configurazioni testate e su nessun dispositivo

3.3.2 Benchmark delle prestazioni

In questo paragrafo sono raccolti alcuni test da me svolti per valutare quantitativamente le prestazioni delle soluzioni analizzate. Per via della natura del problema trattato, non è stato possibile affidarsi agli strumenti di benchmark tradizionali come sysbench: questi, infatti, in ambienti emulati, non sempre forniscono risultati coerenti e a volte le loro misurazioni sono completamente errate. Ne è un esempio questa discussione online **[67]**, in cui degli utenti, tentando di confrontare le prestazioni di Termux e PRoot con sysbench, ottengono tre risultati in antitesi tra loro, di cui uno matematicamente impossibile.

Si è pertanto scelto di ricorrere a dei test pratici, pensati specificatamente per il problema, in cui ci si è limitati ad eseguire le azioni su un dispositivo reale e cronometrare il tempo richiesto (nella pratica si è filmato il dispositivo con un dispositivo esterno e si è poi misurata la durata di tale filmato). Questo approccio fornisce ovviamente risultati meno precisi, più difficili da riprodurre, e affetti maggiormente da fattori imprevedibili, ma garantisce la correttezza del risultato. Come evidente dai dati stessi, i risultati ottenuti sono stati comunque coerenti e non hanno riportato eccessive discrepanze. Tutti i dati ottenuti sono riportati nella forma minuti:secondi. Per ogni test sono riportate 10 misurazioni e la loro media. Per limitare l'influenza di errori, le misurazioni sono state effettuate a distanza di tempo l'una dell'altra e dopo un riavvio del dispositivo.

I test sono stati eseguiti su un cellulare Redmi Note 7 (4G di RAM, processore Octa-Core Snapdragon 660 SoC, LineageOS 19) e su un computer HP (8GB di RAM, processore AMD Ryzen 5 3500U, XUbuntu 20.04 LTS).

3.3.2.1 Booting di Tinycore Linux

In questo test si è misurato il tempo necessario ad avviare il sistema operativo Tinycore Linux 12 da file .iso.

PC - Qemu:

0:41	0:45	0:45	0:38	0:45	0:42	0:45	0:43	0:39	0:44	(Avg) 0:43

PC - Qemu+kvm:

0:07	0:08	0:09	0:07	0:07	0:08	0:08	0:07	0:08	0:07	(Avg) 0:08

Cellulare - IBochs: i dati sono stati misurati dopo aver attivato l'opzione "Unlock speed", che consente di ottenere metriche migliori, e utilizzando 1024Mb di RAM

2:11	2:12	2:53	2:05	2:15	2:09	1:59	2:07	2:10	2:06	(Avg) 2:13

Cellulare - Limbo: i dati sono stati misurati dopo aver attivato l'opzione "High priority" e utilizzando 1024Mb di RAM

2:14	2:17	2:16	7:53	6:11	2:18	2:15	2:21	4:54	2:38	(Avg) 3:32

3.3.2.2 Esecuzione di fibonacci.sh

In questo test viene eseguito uno script (**Codice 1**) che calcola il numero 20 della successione di Fibonacci usando un algoritmo particolarmente inefficiente (è addirittura esponenziale). Il test è particolarmente indicativo nel testare l'efficacia delle soluzioni analizzate nel creare nuovi processi, visto che obbliga il sistema a crearne due per ogni sottoistanza.

```
#!/bin/bash
fibonacci()
{
  if (( $1 < 3 ))
  then
      echo 1
  else
      echo $(( $(fibonacci $(( $1 - 1 ))) + $(fibonacci $(( $1 - 2 ))) ))
  fi
}
date
fibonacci 20
date
```

Codice 1: *Script fibonacci.sh*

PC - nativo:

0:10	0:11	0:11	0:10	0:10	0:10	0:10	0:10	0:09	0:10	(Avg) 0:10

PC - comando chroot, rootfs Debian:

0:12	0:11	0:12	0:12	0:12	0:12	0:12	0:12	0:12	0:12	(Avg) 0:12

PC - PRoot, rootfs Debian:

0:22	0:20	0:21	0:20	0:23	0:22	0:22	0:21	0:22	0:23	(Avg) 0:22

Cellulare - strumento di debug Adb:

1:22	1:02	0:52	1:24	0:48	0:55	1:00	1:00	0:58	1:15	(Avg) 1:04

Cellulare - app IBochs: i dati sono stati misurati all'interno di un terminale di Tinycore Linux dopo aver attivato l'opzione "Unlock speed", che consente di ottenere metriche migliori, e utilizzando 1024Mb di RAM

16:44	16:05	18:10	18:23	17:32	18:01	16:40	17:54	18:22	20:21	(Avg) 17:49

Cellulare - app Limbo: i dati sono stati misurati all'interno di un terminale di Tinycore Linux dopo aver attivato l'opzione "High priority" e utilizzando 1024Mb di RAM

25:36	27:12	21:11	24:12	21:31	27:13	26:18	22:40	24:39	21:31	(Avg) 24:12

Cellulare - app Termux:

1:34	1:37	1:23	1:34	1:26	1:34	1:40	1:25	1:28	1:36	(Avg) 1:32

Cellulare - app Andronix, rootfs Debian:

1:46	1:51	1:59	1:47	1:59	1:55	1:58	1:47	1:56	1:58	(Avg) 1:54

Cellulare - app Andronix, rootfs Alpine Linux:

2:11	2:15	2:11	2:11	2:05	2:09	2:05	2:11	2:08	2:07	(Avg) 2:09

Cellulare - app AnLinux, rootfs Debian:

1:48	1:49	1:57	1:49	1:55	1:57	1:48	1:55	1:55	1:53	(Avg) 1:53

Cellulare - app AnLinux. rootfs Alpine Linux:

2:12	2:14	2:09	2:13	2:08	2:09	2:03	2:14	2:08	2:10	(Avg) 2:10

Cellulare - app UserLAnd, rootfs Debian:

1:39	1:39	1:41	1:38	1:38	1:41	1:39	1:39	1:38	1:39	(Avg) 1:39

Cellulare - app UserLAnd, rootfs Alpine Linux:

2:01	1:58	1:58	1:59	1:58	2:02	1:58	2:01	1:59	1:59	(Avg) 1:59

Cellulare - app Unyw (soluzione proposta):

2:03	2:01	1:57	1:58	2:00	2:01	1:58	2:01	2:02	1:58	(Avg) 2:00

Dal confronto si nota che le uniche soluzioni con performance inaccettabili sono quelle basate su hosted hypervisor, circa 10 volte più lente anche di quelle basate su PRoot. Le soluzioni basate su PRoot ottengono prestazioni leggermente peggiori, ma comunque comparabili a quelle native.

Andronix e Anlinux ottengono prestazioni simili, così come avviene per UserLAnd e la soluzione proposta (esattamente ciò che ci si aspetta, visto che utilizzano gli stessi binari). Non è stato

possibile effettuare delle misurazioni su Debian noroot con il Redmi Note 7, ma altre misurazioni su dispositivi in cui l'app funzionava suggeriscono che abbia prestazioni simili a quelle di UserLAnd.

Si nota che, in generale, le soluzioni basate su Alpine Linux performano leggermente peggio delle corrispettive basate su Debian. Una possibile spiegazione è che l'implementazione di Bash presente su Alpine è compilata con la libreria C musl: musl è di dimensioni minori ma in alcuni ambiti ha prestazioni peggiori di glibc **[45]**.

3.3.3 - Dimensioni app

3.3.3.1 Soluzioni basate su emulatori (hosted hypervisor)

IBochs occupa 26Mb, mentre Limbo x86 occupa 33Mb. A questi dati vanno aggiunti i pesi delle immagini ISO ed eventualmente lo spazio occupato dal disco virtuale.

Tipicamente, le immagini ISO fornite sono più pesanti dei rootfs perché includono una copia del kernel. A titolo di confronto, si segnala, ad esempio, che la versione ISO di alpine Linux specificatamente ottimizzata per gli emulatori pesa da sola 44Mb. Anche in distro come Tinycore Linux, in cui si è compresso il kernel il più possibile, occupa comunque 5Mb, circa un terzo del peso complessivo dell'ISO.

3.3.3.2 Soluzioni native o basate su meccanismi Chroot (OS-Level Virtualization)

In questa sezione viene riportato lo spazio occupato dalle app analizzate in 3.2 dopo il primo avvio.

I dati vengono riportati così come calcolati dal sistema operativo Android. Lo spazio è diviso fra peso dell'apk in sé e peso dei file all'interno delle cartelle riservate all'app, il quale include sia la cartella privata interna che la cartella assegnata all'app sullo spazio condiviso, chiamata tipicamente `./Android/data/ID_DELL'APP`.

Nome app	**Dimensioni app (Mb)**	**Dati utenti (Mb)**	**Totale (Mb)**
Termux (terminale)	110	75	185
UserLAnd (Alpine, terminale o VNC)	30	171	201
UserLAnd (Arch, terminale o VNC)	30	1220	1250
UserLAnd (Debian, terminale o VNC)	30	526	557
UserLAnd (Ubuntu, terminale o VNC)	30	430	461
Debian noroot	201	980	1181
Unyw (soluzione proposta, senza il supporto Pulseaudio)	10	48	58
Unyw (soluzione proposta, includendo Pulseaudio)	15	71	86

3.4 Considerazioni finali

Dai test effettuati, appare evidente che, quando possibile, sono sempre preferibili gli approcci nativi o basati su virtualizzazione OS-Level, sia perché più performanti, sia perché più integrati col sistema, sia perché in grado di evitare un lungo tempo di avvio. Anche in situazioni critiche come quella presentata in 3.3.2, le applicazioni che utilizzano PRoot riescono ad ottenere comunque tempi di esecuzione paragonabili a quelli nativi, mentre le soluzioni hypervisor si dimostrano pesantemente inefficienti, anche 10-20 volte più lente.

Le due soluzioni hypervisor prese in considerazione hanno performance comparabili fra loro (i test suggeriscono che IBochs sia leggermente più performante, ma tranne in rari casi, si tratta di una differenza pari a meno del 30%). Nelle situazioni in cui la distro considerata sia compatibile con entrambe le app, pertanto, non esiste un vero motivo per usare l'una o l'altra. IBochs è più immediato da installare (si può scaricare dal Play Store) e ha un'interfaccia grafica più pulita; Limbo presenta, tuttavia, molte opzioni assenti in IBochs, come la possibilità di utilizzare KVM o eseguire sistemi ARM (anche se quest'ultima funzionalità richiede di scaricare un'app a parte).

Per eseguire le applicazioni nativamente, l'unica soluzione esistente, Termux, si è dimostrata pratica e funzionale: nelle sue repository sono presenti la maggior parte dei programmi a linea di comando tipici dei sistemi GNU/Linux, ed è l'unica fra queste soluzioni che presenta un'API, seppur limitata, per interagire con il sistema Android. Lo svantaggio principale è che l'app è orientata per l'utilizzo da linea di comando, e quindi deficita della maggior parte delle applicazioni ad interfaccia grafica; non è neppure immediato installare ed eseguire le poche applicazioni a interfaccia grafica presenti.

Per le soluzioni OS-Level basate su PRoot, l'esperienza migliore è stata quella ottenuta con UserLAnd. Sia Andronix che AnLinux, infatti, richiedono diversi passaggi per configurare il sistema; entrambi, inoltre, non forniscono rootfs completi e devono quindi compiere parte dell'installazione all'interno del dispositivo: può essere un'operazione lunga. Si segnala, inoltre, che sia Andronix che AnLinux sono installabili dal Google Play Store, e questo è un grave problema, perché l'utente ignaro può non sapere che sono incompatibili con la versione di Termux presente nel Google Play Store (**nota:** recentemente, Andronix ha risolto in parte il problema aggiungendo un avviso che spiega all'utente la situazione). L'unico vantaggio nell'usare queste app è che il sistema viene installato all'interno di Termux; ciò consente di avere più controllo sui file e di poter usare i programmi di Termux su di essi (per esempio, usando l'API termux-job-scheduler, è possibile impostare una sveglia perché avvii PRoot a una determinata ora).

Riassunto delle considerazioni fatte:

Tipologia della soluzione considerata	**Ha prestazioni accettabili?**	**Consente di accedere all'intero parco software delle distro GNU/Linux?**
Software hypervisor	No	Sì
Porting nativi (Termux)	Sì	No
PRoot	Sì	Sì

In definitiva, appare chiaro che le soluzioni basate su PRoot hanno il potenziale per essere le più vantaggiose. Tuttavia, ci sono delle criticità che non vengono risolte da nessuna delle app analizzate.

- I programmi GNU/Linux grafici presentano la stessa interfaccia che avevano su PC, e ciò li rende inadatti all'utilizzo su cellulare.
- L'avvio dell'app Android richiede diversi passaggi, incluso aprire un'app esterna per visualizzare lo schermo trasmesso con VNC.
- Non c'è una vera e propria integrazione con il sistema Android. Si consideri ad esempio, un utente che vuole convertire un file pdf ricevuto via mail utilizzando LibreOffice: effettuare questa semplice operazione richiede come minimo di aprire 4 app Android diverse (l'app da cui proviene il file, l'app che esegue PRoot, l'app che visualizza VNC, un gestore file che consenta di aprire il file convertito) e non c'è alcun modo di semplificare il processo.
- Non è possibile estendere tali soluzioni se non aggiungendo programmi Linux all'interno dell'ambiente PRoot; si tratta di estensioni in ogni caso limitate e difficili da condividere con altri utenti.

4. Soluzione proposta

In questo capitolo si descriverà una soluzione che, pur basandosi sulla virtualizzazione PRoot, presenta un approccio alternativo nella gestione dei programmi. Tale soluzione verrà descritta in parallelo sia dal punto di vista teorico che pratico: per dimostrare la realizzabilità di quanto esposto ne è infatti stata creata un'implementazione concreta e funzionante, il cui codice è liberamente scaricabile dalle repository presenti su github.com/unyw (il nome è una contrazione delle parole Unix e Anywhere). La ragione per cui non si tratta separatamente la parte teorica dall'implementazione è che, in molti casi non sarà possibile procedere senza considerazioni empiriche. Allo stesso modo, a volte, la scelta progettuale più sensata porrà dei problemi teorici che altrimenti non avrebbero ragione di essere trattati in questa tesi.

L'obiettivo che si vuole raggiungere è di mantenere i vantaggi della virtualizzazione PRoot evidenziati in 3.3, fra cui una velocità quasi nativa e la possibilità di accedere a tutto il parco software di una distro, ma fornire un nuovo tipo di interfacce e nuove funzionalità ai programmi esistenti, senza che questi debbano essere modificati tramite patch o ricompilati. La soluzione proposta mirerà inoltre a risolvere alcune delle criticità comuni a UserLand, AnLinux e Andronix, ad esempio renderà più semplice installare e avviare i programmi GNU/Linux.

L'approccio proposto sarà basato sull'uso di "wrappers", cioè di web app (programmi locali scritti in HTML, CSS e JavaScript) in grado da un lato di incapsulare i programmi GNU/Debian (eseguendoli, visualizzandoli e rimappando le loro funzionalità in modo da renderli fruibili più facilmente da Android), e dall'altro di comunicare con Android stesso, diventando così dei ponti che consentono ai due sistemi di interagire fra loro. Per rendere possibile l'installazione e l'esecuzione di tali web app, sarà necessario programmare un'applicazione Android (unyw.apk) che le gestisca e consenta loro di comunicare con le librerie di sistema. Per completare il progetto verranno inoltre forniti degli strumenti di sviluppo che semplifichino il processo di scrittura di tali web app e il processo di pubblicazione delle medesime. (**nota:** nel seguito, i termini "wrapper" e "web app" verranno trattati essenzialmente come sinonimi).

Vi sono numerosi vantaggi nell'utilizzare wrappers programmati con tecnologie web:

- le interfacce create possono essere visualizzate facilmente e con ottime performance all'interno di un'app Android, tramite il componente "webview" della libreria di sistema. Tale componente è basato su Chromium e viene aggiornato automaticamente in tutte le versioni di Android moderne (Android 4.4 e successivi **[34]**).
- Le web app sono composte da semplici file HTML, CSS e JavaScript: per installarle, è sufficiente scaricarle e copiarle nei file privati dell'applicazione Android, senza che ci sia alcun bisogno di far intervenire Android o chiedere particolari permessi al sistema. Questo fa sì che possano essere gestite con facilità da un qualsiasi file package manager, incluso quello della distro eseguita con PRoot; possono quindi essere distribuite tramite le repository della distro e possono essere facilmente sincronizzate con i programmi che vanno a incapsulare (basta che indichino tali programmi come dipendenze nel manifest).
- Le web app sono generalmente immediate da scrivere, sia perché richiedono un post processing minimo (a differenza delle app Android che, come spiegato in 2.7, necessitano di essere pseudocompilate tramite Android SDK, un'operazione lunga e complessa), sia perché le tecnologie web sono più ad alto livello se confrontate con Java.

- In quanto a dimensione su disco, tali app sono anche molto leggere sia rispetto alle app Android che soprattutto ai programmi grafici GNU/Linux.
- È possibile visualizzare un programma GNU/Linux all'interno di un'interfaccia web usando il client VNC "noVNC.js" **[70]**. Ciò rende possibile comporre con facilità interfacce ibride, che mostrano i programmi incapsulati, ma aggiungono controlli e nuovi elementi grafici. Va sottolineato come noVNC.js sia un'implementazione completa del protocollo e consenta, quindi, di interagire in maniera automatica con lo schermo che si sta visualizzando, ad esempio inviando da codice una sequenza di tasti.
- Alcuni noti programmi come Visual Studio Code o Jupyter Notebook, utilizzano come predefinita un'interfaccia web; usare interfacce web all'interno della soluzione proposta fa sì che tali programmi siano facilmente visualizzabili praticamente senza modifiche grafiche.

A questo va aggiunto che i siti web sono un formato universale; questo non rappresenta un vantaggio per l'utente, ma semplifica notevolmente il processo di sviluppo e debugging, perché fa sì che le app possano essere visualizzate e testate anche su PC senza bisogno di emulatore.

Lo svantaggio principale dell'approccio scelto è che le web app vengono eseguite in un ambiente protetto e controllato (la webview, paragonabile a un browser) e ciò fa sí che abbiano accesso a meno risorse e che in generale abbiano meno possibilità di interagire con il sistema. Un sito web, ad esempio, non è in grado di leggere e scrivere file, né può accedere liberamente ai processi o ai dati relativi ai sensori.

Per risolvere tale problema è possibile istituire un canale che metta in comunicazione le web app con l'applicazione Android, in modo che le web app possano chiedere all'applicazione Android di eseguire codice al di fuori dell'ambiente protetto (si parla in questo caso di "hybrid web app"). Tale canale (che verrà formalizzato in 4.2), se reso disponibile ai programmi GNU/Linux, può fornire anche a loro un modo semplice per accedere a funzionalità Android, similmente a quanto fa la Termux-api descritta in 3.2.1.

Si specifica fin da subito che l'app creata, pur essendo tecnicamente utilizzabile, non è pensata per essere un prodotto distribuibile a un vero utente, ma va intesa come un prototipo realizzato esclusivamente per questa tesi e il cui scopo è dimostrare le potenzialità della soluzione proposta. Ad esempio, non ci si concentrerà nel creare wrapper per numerosi programmi GNU/Linux, ma ci si limiterà a crearne alcuni che esemplifichino le potenzialità dell'API e illustrino alcune logiche con cui possono essere progettati.

4.1 Filosofia di sviluppo

Oltre ai requisiti imposti dalla soluzione cercata, si è deciso di progettare e sviluppare seguendo anche ulteriori criteri, che, pur essendo soggettivi e non strettamente inerenti al problema in esame, hanno lo scopo di ottenere un codice finale più ordinato, coerente e facile da gestire.

- **Assenza di compilazione:** si è scelto di evitare nel modo più assoluto di inserire nella soluzione proposta codice che necessitasse la compilazione. La soluzione proposta è stata quindi interamente realizzata con linguaggi interpretati/pseudocompilati fra cui shell Ash, Kotlin, JavaScript/CSS/HTML/Svelte; tutti i binari presenti come PRoot e Busybox sono stati presi precompilati da fonti esterne. Questo approccio ha un duplice vantaggio: da un lato semplifica il processo di generazione dell'app (che, come si vedrà in 4.4.1, è già

piuttosto elaborato), dall'altro rende più stabile il risultato finale, perché fa sì che alcune parti critiche siano delegate a progetti terzi con alle spalle ore di lavoro e debugging.
- **Astrazione del sistema Android:** si è scelto, ove possibile, di non esporre direttamente i wrapper alla complessità del sistema Android, ma di astrarre le funzionalità e le risorse fornite da quest'ultimo. Sarebbe stato possibile compiere la scelta opposta. Tale possibilità verrà vagliata in 4.4.2.1, spiegando i vantaggi e gli svantaggi che comporta e il perché non è stata adottata.
- **Unica app Android apk:** si è scelto di produrre come risultato un unico file apk funzionante su tutte le architettura, in modo che l'app risultante sia più facile da distribuire, ma comporta, fra le altre cose, che i rootfs debbano essere distribuiti separatamente dall'apk principale.

4.2 Struttura del progetto

Complessivamente, si è scelto di dividere il codice prodotto in tre parti, ciascuna con una diversa repository Github:

- i rootfs, uno per ciascuna architettura cpu supportata (ossia x86, x86_64, ARM e ARM64) (repository: github.com/unyw/android-rootfs, verranno trattati in 4.3);
- l'applicazione Android (repository: github.com/unyw/android-apk, verrà trattata in 4.4);
- i wrappers (repository: github.com/unyw/repo-main, verranno trattati in 4.5).

È stata poi creata una repository chiamata github.com/unyw/template-app che può essere clonata e usata come base per lo sviluppo di un wrapper.

Per semplificare il processo di distribuzione dell'app, si è scelto di includere in ciascuna repository sia il codice sorgente che la versione di quest'ultimo preparata per la distribuzione. Github mette a disposizione un servizio chiamato "Github Pages" **[35]**, che consente di configurare una repository in modo che i suoi file siano scaricabili tramite HTTPS: ciò rende possibile usare le repository stesse per la distribuzione dell'app, senza bisogno di affidarsi a servizi esterni.

La divisione scelta è una naturale conseguenza del fatto che i tre gruppi producono ciascuno una risorsa che necessita di essere scaricata separatamente dalle altre, infatti:

- mantenere un'unica app valida per tutte le architetture cpu, comporta che essa debba scaricare il rootfs a parte; l'alternativa sarebbe inglobare all'interno del file apk i rootfs per tutte le architetture possibili, rendendolo estremamente e inutilmente pesante.
- Includere all'interno dei rootfs tutti i wrapper e le applicazioni che incapsulano, li renderebbe estremamente pesanti; si è pertanto scelto di includere nei rootfs solo una selezione minimale dei wrapper/delle web app e poi lasciare all'utente la possibilità di installare i restanti.

Tale divisione ha anche il vantaggio di separare il codice sorgente Kotlin, contenuto interamente nella repository android-apk, e il codice sorgente web, contenuto per la quasi totalità nella repository repo-main.

4.3 Scelta della distro e generazione dei rootfs

Dopo aver considerato Debian, Ubuntu, Arch Linux e Alpine, si è scelto Alpine come distro, sia per via delle dimensioni estremamente ridotte, sia per la velocità del package manager **[36]**. A posteriori tale scelta si è rivelata particolarmente felice, considerando che le dimensioni dell'app ottenuta sono inferiori a quelle di tutte le altre soluzioni considerate.

Si è quindi passati a creare uno script che automatizzasse il processo di generazione dei rootfs. Tale script si occupa di:

- scaricare un rootfs minimale dal sito ufficiale di Alpine ed estrarlo in una cartella;
- entrare nel sistema usando PRoot;
- all'interno, aggiungere la repository github.com/unyw/repo-main/stable e scaricare tutti i pacchetti necessari per un'installazione minimale, cioé **unyw-bridge-android**, **unyw-app-home** e **unyw-app-xterm** (vedi 4.5.2);
- comprimere nuovamente l'archivio (si è utilizzato xz perché ha fornito il rapporto di compressione migliore);
- generare un checksum del file ottenuto.

L'operazione viene ripetuta per ciascuna della quattro architetture supportate (nella armv7, aarch64, x86, x86_64), usando qemu quando necessario.

4.4 App Android

Per consentire l'esecuzione di programmi GNU/Linux in background anche quando l'utente non sta più utilizzando attivamente l'app Android, è necessario che questa sia composta da almeno due componenti:

- un'*activity*, cioè il componente attivo, che si occupa di visualizzare le web app tramite una webview;
- un *service*, cioè un componente in background che si occupa di gestire i processi GNU/Linux anche quando l'utente chiude l'activity o visualizza un'altra app. Come spiegato in 2.8, tale service non può essere completamente in background, ma deve informare l'utente della propria presenza tramite una notifica prioritaria (si parla di "foreground service") o rischia di essere comunque terminato dal sistema Android per liberare risorse.

All'apertura, l'activity controlla se il service è in esecuzione e in caso sia necessario lo avvia. Si procede quindi a creare un legame fra l'activity e il service, in modo che possano effettuare inter process communication. Questo in pratica viene fatto inizialmente tramite Intent e poi tramite "binding" (vedi 2.9). Tale legame persiste fino al momento in cui l'activity è visualizzata a schermo e viene revocato ogni volta in cui l'utente chiude l'activity o la mette in background.

Nel momento in cui l'utente avvia nuovamente l'activity, questa nota che il service è già in esecuzione e procede pertanto a creare nuovamente il legame con esso. In questo modo non c'è bisogno di avviare nuovamente processi come x11vnc e le web app possono collegarsi ai processi aperti in precedenza.

Mentre l'activity è composta da un solo componente (che si occupa essenzialmente di avviare il service e visualizzare la webview), il service è a sua volta diviso in tre parti, ciascuna con una funzionalità specifica e ciascuna con il proprio thread.

- Il server API, che si occupa rendere disponibili le funzionalità e risorse di Android ai wrapper e ai programmi GNU/Linux. Tale API verrà descritta nel dettaglio in 4.4.1.
- Il modulo che si occupa del processo PRoot principale; tale modulo si occupa di installare il rootfs se assente, e poi di avviare i processi che dovrebbero essere sempre attivi in background, come ad esempio Xvfb e x11vnc. I processi di installazione ed avvio verranno trattati nel complesso in 4.4.2.
- Il modulo che si occupa di gestire gli altri processi e di terminarli. In 4.4.3 verranno trattate due possibili implementazioni.

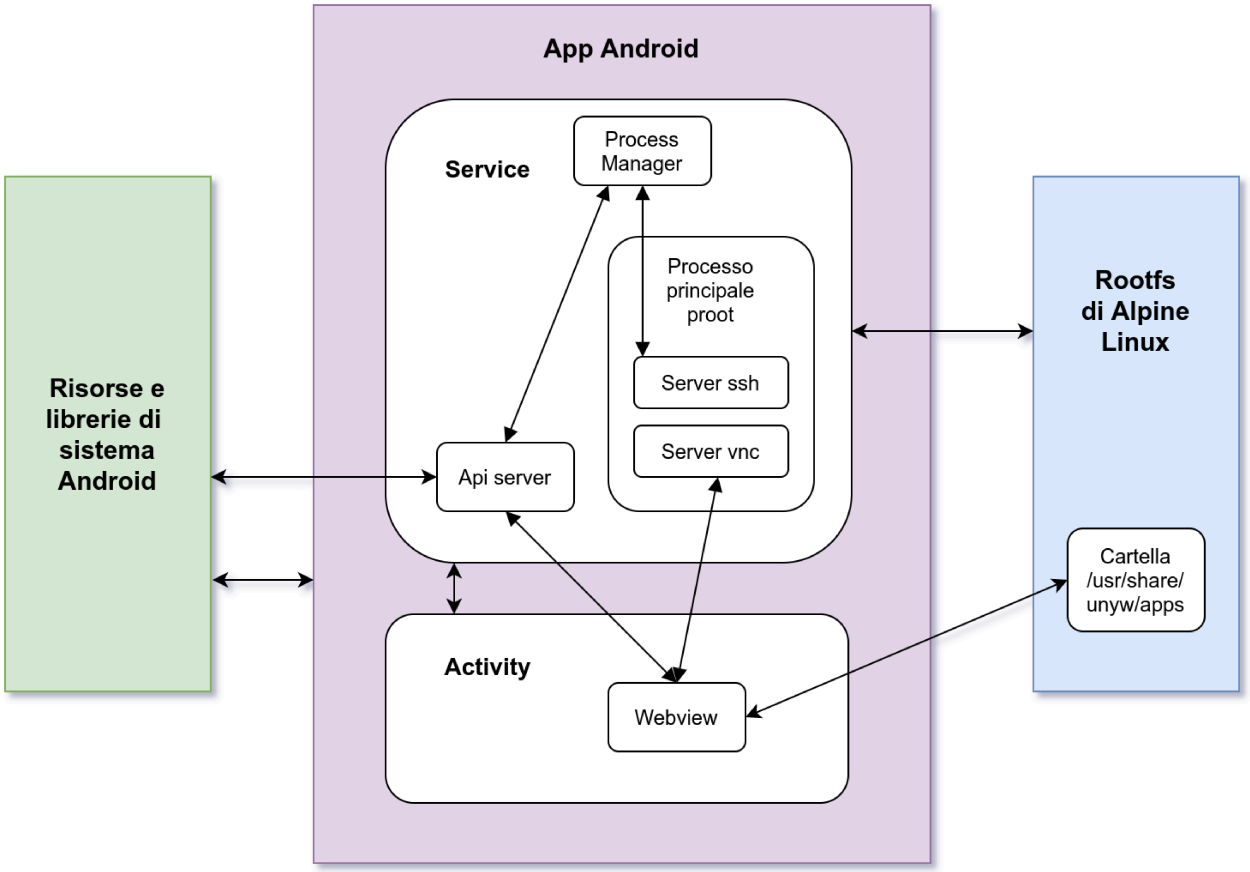

Figura 7: *Componenti principali dell'app e loro interazioni*

4.4.1 Installazione e avvio dell'app

Perché l'app funzioni, è necessario che fra i suoi file interni siano correttamente installati PRoot e il rootfs.

Come anticipato in 4.2, non è possibile inglobare il rootfs all'interno dell'apk per motivi di spazio; è quindi necessario scaricarlo dalla repository github.com/unyw/android-rootfs.

Il processo di installazione viene effettuato dal service, perché deve continuare anche quando l'app è in background e consiste in tre fasi:

- vengono copiati alcuni file dall'interno del pacchetto apk alla cartella dei file privati dell'apk; questo include due script (install.sh e start.sh), Busybox (una collezione di utility linux a linea di comando) PRoot e le librerie dinamiche usate da quest'ultimo;
- viene scaricato il rootfs utilizzando una libreria di sistema Android;
- viene chiamato lo script "install.sh", che si occupa, chiamando opportunamente busybox, di decomprimere il rootfs, e completa l'installazione con le altre piccole operazioni necessarie, come generare un identificativo casuale all'interno del file `/etc/machine-id`.

Sarebbe più elegante pensare di effettuare tutto il processo di installazione usando esclusivamente le librerie Android o esclusivamente lo script anziché mischiare i due, tuttavia nella pratica non è possibile:

- busybox non include il supporto per scaricare usando HTTPS, perché richiederebbe la verifica dei certificati e tale operazione obbligherebbe ad inglobare nel binario la pesante libreria OpenSSL;
- le librerie di sistema Android non includono un modo per decomprimere i file xz e in generale, non prevedono un modo affidabile per decomprimere archivi in modo ricorsivo e mantenendo inalterati i bit di autorizzazione.

Per indicare che l'installazione è andata a buon fine, viene creato un file chiamato "unyw_installed.txt", che servirà anche a segnalare che il rootfs è già stato installato alle successive istanze. In caso di assenza di tale file alla fine del processo, l'installazione viene ripetuta dall'inizio. Non è previsto un sistema di recupero di installazioni completate parzialmente: non avrebbe senso di esistere visto che, nella pratica, il processo dura meno di cinque minuti.

Durante tutto il processo di installazione, se viene aperta l'activity, questa visualizzerà una semplice pagina HTML speciale, programmata direttamente all'interno del codice Kotlin, che include una generica barra di progresso e raccoglie i log generati dal service. Tali log vengono trasmessi utilizzando sempre il binding ma sfruttano un canale diverso da quello fornito dall'API, in modo da essere indipendenti da quest'ultima.

Una volta che il processo di installazione è terminato, l'app procede eseguendo un normale avvio

Se non specificato altrimenti, l'activity si apre su un wrapper speciale chiamato **unyw-app-home** e corrispondente all'URL `/apps/home/index.html`; da tale web app è possibile accedere a tutte le altre o installarne di nuovi.

È tuttavia possibile che l'app non si sia aperta come risultato di un semplice click dell'utente, ma di un Intent più specifico, fornito da un'altra app o da uno shortcut. In questo caso:

- se l'activity viene lanciata con un Intent che ha un parametro URL, viene aperto quell'URL anziché quello predefinito;
- se l'activity viene lanciata con un Intent che coinvolge l'apertura file, cioè Intent.ACTION_SEND o Intent.ACTION_VIEW, il service si occupa di risolvere l'Intent e di copiare tale file in un percorso accessibile all'applicazione. Viene quindi chiesto all'utente con quale web app tra quelle installate voglia aprire il file. Si procede infine ad aprire l'URL relativo, a cui viene aggiunto parametro URL speciale, del tipo *?unyw=DATA*, che indica alla web app che dovrebbe aprire il file le cui specifiche sono contenute all'interno di DATA.

Nel caso in cui il service non sia attivo, l'activity attende volutamente qualche secondo prima di procedere con il caricamento dell'URL per dare modo al service di eseguire uno script chiamato "start.sh" che si occupa di avviare alcuni processi in background necessari al corretto funzionamento, fra cui:

- un server SSH, per entrare nel sistema da remoto, utile ad esempio per motivi di debug, e utilizzato dal server API;
- Xvfb, x11vnc e il window manager bspwm (verranno trattati più approfonditamente in 4.4.3).

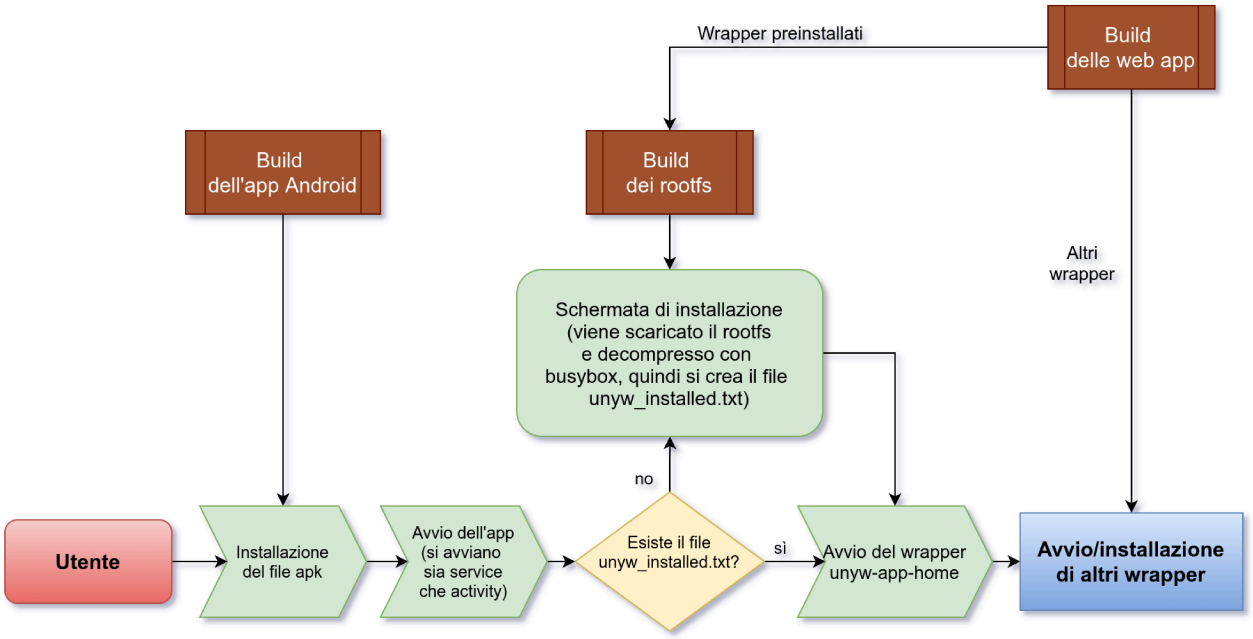

Figura 8: *Processi di build, distribuzione, installazione e avvio dell'app*

4.4.2 Server API

Come già anticipato, per consentire l'uso di funzioni di Android, è necessario fornire un'api, ossia un insieme di funzioni chiamabili dall'esterno tramite un protocollo predefinito. Dato che tale api dovrebbe essere disponibile sia ai wrapper che ai programmi GNU/Linux, si è scelto di realizzarla utilizzando il protocollo HTTP. Nello specifico si è implementato un server utilizzando la libreria Ktor; tale server vive all'interno di un thread ed è sempre raggiungibile attraverso la porta 12080.

I vantaggi del protocollo HTTP sono che si tratta di un protocollo request/reply universale, stateless (non richiede di istituire un collegamento, ogni chiamata è indipendente dalle altre), facilmente implementabile e monitorabile.

Gli svantaggi principali di questa scelta, invece, possono essere riassunti come segue.

- Se non si limitano connessioni esterne, un server HTTP può aprire una falla nella sicurezza del dispositivo. Ciò può essere mitigato consentendo solo connessione da localhost e utilizzando un token di accesso (vedi 4.6).
- Il protocollo HTTP in generale aggiunge header pesanti alle comunicazioni, questo lo rende particolarmente inefficiente per i programmi che hanno bisogno di effettuare molte chiamate **[37]**. Ciò non è un problema nel caso in esame, perché nessuna delle API previste ha bisogno di essere chiamata in modo ripetuto o necessita di tempi di risposta real-time. Si specifica, per completezza, che tale protocollo è comunque più veloce di un ordine di grandezza rispetto alla soluzione implementata da Termux-api: una chiamata HTTP può impiegare meno di 120 millisecondi, mentre tutte le API Termux richiedono almeno 3 o 4 secondi per essere eseguite, probabilmente per via della chiamata ad am.
- Il protocollo è completamente sincrono. Questo può essere un problema: non consente, ad esempio, a un wrapper di essere informato tramite evento se l'applicazione viene terminata. Va tuttavia considerato che, nel caso in esame, sarebbe stato complicato utilizzare un protocollo asincrono, questo sia perché la soluzione proposta prevede componenti che possono essere terminati e riavviati in continuazione, sia perché non tutti i contesti hanno a disposizione la possibilità di gestire adeguatamente eventi asincroni (es: bash). Inoltre, alcuni moduli dell'API stessa consentiranno di ovviare parzialmente a questo problema, fornendo dei meccanismi di callback ad eventi.

4.4.2.1 Approccio basato su funzioni e moduli predefiniti

In accordo con quanto enunciato in 4.1, si è deciso di astrarre l'accesso alle librerie Android, creando un'API con delle funzioni specifiche e predefinite. Per semplicità di implementazione, si sono raggruppate tali funzioni in moduli e si è fatto sì che tutte le chiamate avessero la struttura fissa, con URL della forma:

http://localhost:12080/MODULO/FUNZIONE?token=TOKEN

e body in formato JSON (TOKEN è il token di accesso, vedi 4.6). Si è scelto di utilizzare solo chiamate di tipo "post". La scelta di un unico tipo di chiamate è giustificata dal fatto che, in questo modo, la funzione chiamata è identificata univocamente dai soli nomi di modulo e della funzione stessa. L'uso di "post" anziché, ad esempio, "get" è per via dell'assenza di cache (assolutamente deleteria nel caso in esame - si vuole che ogni chiamata abbia un risultato aggiornato) e la possibilità di inviare body molto lunghi.

Si è scelto di far ricevere sempre una risposta in formato JSON, che conterrà sempre un campo "status": "success" se la richiesta è andata a buon fine.

La scelta del formato JSON ha il vantaggio intrinseco di rendere l'API robusta ad aggiornamenti, consentendo l'aggiunta di campi sia nelle richieste che nelle risposte senza per questo far smettere di funzionare i wrapper pensati per versioni precedenti. Tale formato non è particolarmente indicato per l'uso con i programmi GNU/Linux a linea di comando, che di solito preferisco utilizzare testo puro; è tuttavia possibile utilizzare tool come il programma a linea di comando jc per estrarre campi da un JSON con facilità.

I moduli sono raggruppabili in due categorie: moduli funzionanti anche in background ("unyw", "device", "file", "process", "job" e "notification"), e moduli che richiedono che l'activity sia aperta e in primo piano ("webview" e "intent"). È possibile utilizzare le funzioni del modulo "notification" per avvisare l'utente che un processo in background è terminato, tramite una vibrazione, un messaggio toast o una notifica vera e propria, mentre il modulo "webview" consente ai programmi GNU/Linux di interagire con le web app se queste sono attive.

```
POST REQUEST:
localhost:12080/process/run?token=debugapi
{
 "command": "echo Hello world!"
}

REPLY:
{
 "status": "success",
 "result": "Hello world!\n"
}
```

Codice 2: *Esempio request-reply*

Moduli e funzioni implementate
Modulo unyw: include funzioni legate a unyw stesso /unyw/info: fornisce informazioni sull'app, come una lista delle API disponibili e un elenco dei token di sicurezza
Modulo device: include funzioni per gestire il dispositivo e il suo hardware /device/torch: per accendere/spegnere la torcia /device/battery: per ottenere informazioni sulla batteria

Modulo file: include funzioni per leggere, scrivere ed elencare i file

/file/read: per leggere un file come stringa

/file/write: per scrivere un file di testo

/file/list: per ottenere informazioni sui file presenti all'interno di una cartella

/process: funzioni per gestire i processi PRoot

/process/run: per eseguire un processo ed ottenere l'esito dell'esecuzione

/process/dtach: per lanciare un processo in background

/process/screen: per lanciare un processo

/process/list: per avere una lista dei processi in background attivi

/process/kill: per terminare un processo in background

Modulo job: gestisce azioni programmate da Android per essere eseguite a una certa ora o ripetutamente

/job/schedule: consente di creare un job e avviarlo

/job/stop: consente di terminare un job

/job/list: consente di verificare i job in esecuzione

Modulo notification: include funzioni per notificare all'utente quando il processo è in background

/notification/toast: visualizza un messaggio "toast" (sono i messaggi neri a scomparsa)

/notification/vibrate: fa vibrare il dispositivo

/notification/show: mostra una notifica nella barra delle applicazioni o modifica una notifica esistente

/notification/list: restituisce una lista delle notifiche create e attualmente visualizzate

Modulo webview: funzioni per interagire con la webview; funziona solo se l'activity è visualizzata a schermo

/webview/info: fornisce informazioni sulla webview, come la webapp in esecuzione

/webview/eval: esegue codice all'interno della webview e restituisce il risultato

/webview/event: chiama un evento all'interno della webview (si può rispondere agli eventi ricevuti utilizzando la funzione standard window.addEventListener)

Modulo intent: include una serie di funzioni per interagire con le altre app installate all'interno del dispositivo; funziona solo se l'activity è visualizzata a schermo

/intent/authenticate: consente all'utente di autenticarsi con il pin o con il sensore di impronte

/intent/photo: apre la telecamera e chiede di scattare la foto

/intent/open: apre un file presente nelle cartelle dell'app (ad esempio un documento) con l'app predefinita di sistema

/intent/filepicker: chiede all'utente di scegliere un file

***nota:** tecnicamente, la richiesta di autenticazione non è classificata come "Intent" all'interno della libreria di sistema Android, ma siccome fondamentalmente ne segue le stesse regole e ha bisogno che l'activity sia visualizzata per andare a buon fine, è parso naturale inserire la funzione in questo modulo e non in "device"

Trattare in maniera approfondita l'implementazione, il protocollo e le specifiche delle funzioni implementate, sarebbe ridondante ed esulerebbe dallo scopo di questa tesi, visto che l'obiettivo è dimostrare come sia possibile far coesistere l'API con i programmi GNU/Linux e non dimostrare le potenzialità dell'API in sé; ci si limiterà, pertanto, a fare delle considerazioni generali sul perché sono state compiute alcune scelte architetturali di interesse. A tal proposito, si ricorda che l'applicazione presentata è un prototipo ideato per dimostrare le potenzialità delle idee espresse e non per essere realmente funzionale a un utente. Pertanto, ad esempio, sono stati sviluppati solo un sottoinsieme ristretto di moduli device; prima di un effettivo rilascio dell'app sarebbe necessario, tra le altre cose, implementare funzioni per la gestione di sensori come accelerometro e gps o per la gestione della fotocamera in background. Si è comunque cercato di creare un raggruppamento in moduli plausibile anche in vista di una distribuzione futura.

Nella maggior parte dei casi, si è sviluppato evitando di fornire funzioni ridondanti. Vi sono però alcune eccezioni.

- Il modulo file tecnicamente è superfluo, visto che è possibile effettuare tali operazioni con programmi a linea di comando Linux invocati dal modulo process; tuttavia, nella pratica, le operazioni sui file sono molto frequenti e quindi ha senso ottimizzarle, inoltre non sempre è facile effettuarle con programmi GNU/Linux ed ottenere un risultato utile per una macchina (ad esempio, ls non ha un output standard: è un comando utile per un umano ma rischioso se usato all'interno di un processo automatico).
- `/webview/event` può essere eseguito anche utilizzando `/webview/eval`. In questo caso si voleva incentivare uno specifico pattern per l'interazione fra i programmi GNU/Linux e le web app.

Tutte le funzioni sono state inoltre programmate per essere "stateless" (non richiedere chiamate pregresse) e robuste a chiusure improvvise dell'entità chiamante. La maggior parte delle funzioni, ad esempio, si esaurisce con la chiamata stessa; nel caso in cui le chiamate portino alla creazione di risorse che sopravvivono al termine della chiamata (come avviene ad esempio con `/process/dtach`, che crea un processo in background, e `/notification/show`), tali risorse hanno sempre un codice identificativo che non può essere smarrito perché è sempre possibile chiedere una lista dei codici attivi (ad esempio, una web app che viene chiusa e riaperta può controllare se aveva già lanciato un processo in precedenza utilizzando `/process/list`).

Per rendere più agevole chiamare l'API all'interno di Alpine Linux, è stato creato uno script (**Codice 3**) chiamato unyw-api, che al proprio interno chiama il software Linux wget per effettuare una chiamata HTTP.

```
#!/bin/sh
wget -O - -q --post-data="$2" \
 localhost:12080"$1"?token=`cat /tmp/UNYW_TOKEN_API`
```

Codice 3: *Script unyw-api per collegarsi all'API utilizzando wget*

4.4.2.2 Approccio alternativo basato su Java Reflection

In questo paragrafo si descriverà un approccio in antitesi con quello presentato nel paragrafo precedente, in cui anziché astrarre le funzionalità della libreria di sistema, le si rende disponibili per intero così come sono. Ciò è possibile grazie alla libreria di sistema "Java Reflection" **[38]**, che consente di effettuare meta-programmazione e di comporre ed eseguire codice Java arbitrario durante l'esecuzione. Con Java Reflection è possibile, ad esempio, verificare quali classi siano presenti all'interno di una libreria, invocare una funzione conoscendone il nome come stringa, importare una libreria e istanziare una classe on-the-fly **[71]**.

Tecnicamente, fornendo un modo per interfacciarsi con Java Reflection, si fornisce l'accesso a tutte le funzioni Java disponibili sul dispositivo, esattamente come dare accesso alla funzione eval in JavaScript/Python fornisce un modo per richiamare qualsiasi funzione di quel linguaggio.

L'uso di Java Reflection a questo scopo trova un utilizzo pratico anche in casi reali, ad esempio è alla base del framework NativeScript **[39]** per la realizzazione di app Android tramite JavaScript.

Tale approccio ha principalmente due vantaggi:

- il primo è che consente con un minimo sforzo di avere una mappatura completa di tutte le funzionalità disponibili all'interno di Android;
- il secondo è che tale mappatura si estende automaticamente anche alle funzionalità uscite successivamente al rilascio dell'app Android, rendendo addirittura, sotto opportune ipotesi, superfluo aggiornare l'apk e limitare l'aggiornamento alle sole web app.

È facile argomentare come quest'ultimo vantaggio sia relativo: un'API ben progettata, infatti, non dovrebbe richiedere aggiornamenti frequenti; inoltre, non c'è un reale beneficio ad aggiornare le web app, che sono numerose, rispetto ad aggiornare l'app Android, che anzi è una sola. Tra l'altro si ricorda che le app distribuite al di fuori del Google Play Store, hanno la possibilità di essere programmate per auto-aggiornarsi senza bisogno di intervento esterno.

Al contrario, questa scelta porta problematiche difficilmente risolvibili:

- innanzitutto costringe i programmatori delle web app a interfacciarsi, direttamente o meno, con la complessità di Android, che spesso richiede protocolli precisi e laboriosi per accedere alle risorse, ad esempio, per richedere l'autenticazione biometrica sono necessarie 30 righe di codice Kotlin, mentre grazie all'astrazione dell'API, ai programmatori delle web app ne sono richieste solo 2. Consentire l'esecuzione di codice arbitrario elimina, in generale, ogni possibilità di intervenire facilmente sul codice stesso, ad esempio colmando le lacune di Android o appianando le differenze fra le varie versioni del sistema operativo.
- L'ambiente fornito da Reflection richiede un dispositivo Android (reale o emulato) per essere eseguito o testato efficacemente; al contrario un'API più astratta può essere testata utilizzando implementazioni alternative. Ciò comporta, ad esempio, che sia possibile progettare dei test automatici molto più leggeri. Addirittura, si potrebbe reimplementare l'API nella sua interezza per piattaforme diverse da Android, per consentire l'esecuzione delle web app su altri sistemi operativi, come il Windows Subsystem for Linux.
- Vi è inoltre un problema più sottile sul modo in cui gestire la memoria all'interno della Java Virtual Machine. Il linguaggio Java, infatti, implementa il paradigma ad oggetti, e questo fa sì che per svolgere delle azioni sia necessario istanziare delle classi, classi che in generale non sono serializzabili (non è possibile trasferirle su JavaScript) e necessitano di essere mantenute in memoria per operazioni successive. La soluzione classica a questo problema

(che è anche quella implementata da NativeScript) è l'utilizzo di una "pool", ossia un contenitore virtuale in cui vengono inserite tutte queste risorse, ciascuna con un token identificativo. La web app può quindi chiedere di creare una risorsa, ottenere in cambio il token e utilizzare tale token per interfacciarsi con la risorsa in futuro. In questo modo si perde però traccia dell'ambito in cui gli oggetti sono stati creati, e ciò impedisce al garbage collector (il meccanismo automatico di gestione della memoria di Java) di eliminarli una volta che hanno esaurito la loro funzione. Nello specifico, la perdita del token all'interno della web app fa sì che non esista può alcun modo per interfacciarsi con la risorsa, che rimane attiva all'interno del service anche dopo che la web app è stata chiusa. Per via della grande mole di oggetti con cui devono ragionare i programmatori Java, ciò introduce nell'API una serie indiscriminata di elementi non stateless, che la web app deve accuratamente gestire con il rischio costante di intasare la pool e sovraccaricare la memoria o, al contrario, di tentare l'accesso a risorse non più esistenti (si parla di "dangling reference"). Si pensi ad esempio ad una web app che, ogni secondo, controlla il disco per vedere se sono stati aggiunti dei file ad una cartella: se non venisse implementata correttamente la cancellazione degli oggetti creati, l'utente potrebbe ritrovarsi in poco tempo con decine di migliaia di istanze dell'oggetto "File" in memoria, e sarebbe costretto a riavviare il dispositivo. La situazione peggiora ulteriormente se si considera che l'API dovrebbe essere accessibile anche ai programmi GNU/Linux in background e che le web app potrebbero ritrovarsi terminate in qualunque istante, visto che l'utente potrebbe chiudere l'activity in qualunque momento. La soluzioni esistenti in letteratura, non sono applicabili, perché richiederebbero delle modifiche all'interprete JavaScript (come fa ad esempio NativeScript, che comunque si pone nella semplicistica condizione di un'API accessibile da un'unica activity che non presenta service).

A ciò si aggiunge che un'API del genere richiederebbe molte più chiamate di quella attuale e ciò, in teoria, potrebbe portare ad evidenziare i limiti di prestazioni del protocollo HTTP.

Per queste ragioni nell'implementazione fornita si è pertanto scelto di concentrarsi unicamente sull'approccio presentato nel paragrafo 4.4.2. Comunque, nulla vieta, in futuro, di adottare un approccio ibrido, implementando all'interno dell'API un modulo per accedere alle librerie Android tramite Java Reflection. Questo consentirebbe di mantenere tutti i benefici di un'API astratta, ma fornire ai programmatori delle web app un modo alternativo per accedere alle librerie Android in mancanza di un modulo API che implementi quella funzione specifica. Per risolvere il problema degli oggetti in memoria, si potrebbe pensare a una divisione della pool in "sessioni", in modo da consentire al service di eliminare le sessioni inattive. Si tratta, in ogni caso, di un problema ben più esteso (basti pensare che lo studio della gestione dei puntatori è alla base di intere branche dell'informatica **[68]**), che meriterebbe una trattazione in una tesi a parte e, pertanto, non verrà approfondito ulteriormente, rimanendo quindi un problema aperto.

4.4.3 Modulo “process” e gestore dei processi

Una parte cruciale del progetto presentato è la possibilità per le web app di interagire con il sistema GNU/Linux, lanciando programmi di quest’ultimo. Perché questo sia possibile, si è realizzato un gestore di processi all’interno del service. Nel tentativo di ottenere le migliori performance possibili, si sono create due implementazioni intercambiabili: nel primo caso si è utilizzata direttamente la classe Java ProcessBuilder, nel secondo ci si è collegati al server SSH interno tramite la libreria. JSch. Il vantaggio di usare SSH è che, in questo modo, non è necessario caricare ogni volta PRoot, come mostrato in figura. La prima implementazione sembra presentare momentanei cali di prestazione dopo il mancato utilizzo dell’app per qualche minuto; non si è tuttavia riusciti a riprodurre questo fenomeno in modo consistente, né a misurarlo. Per il resto, i risultati ottenuti con le due implementazioni sono paragonabili.

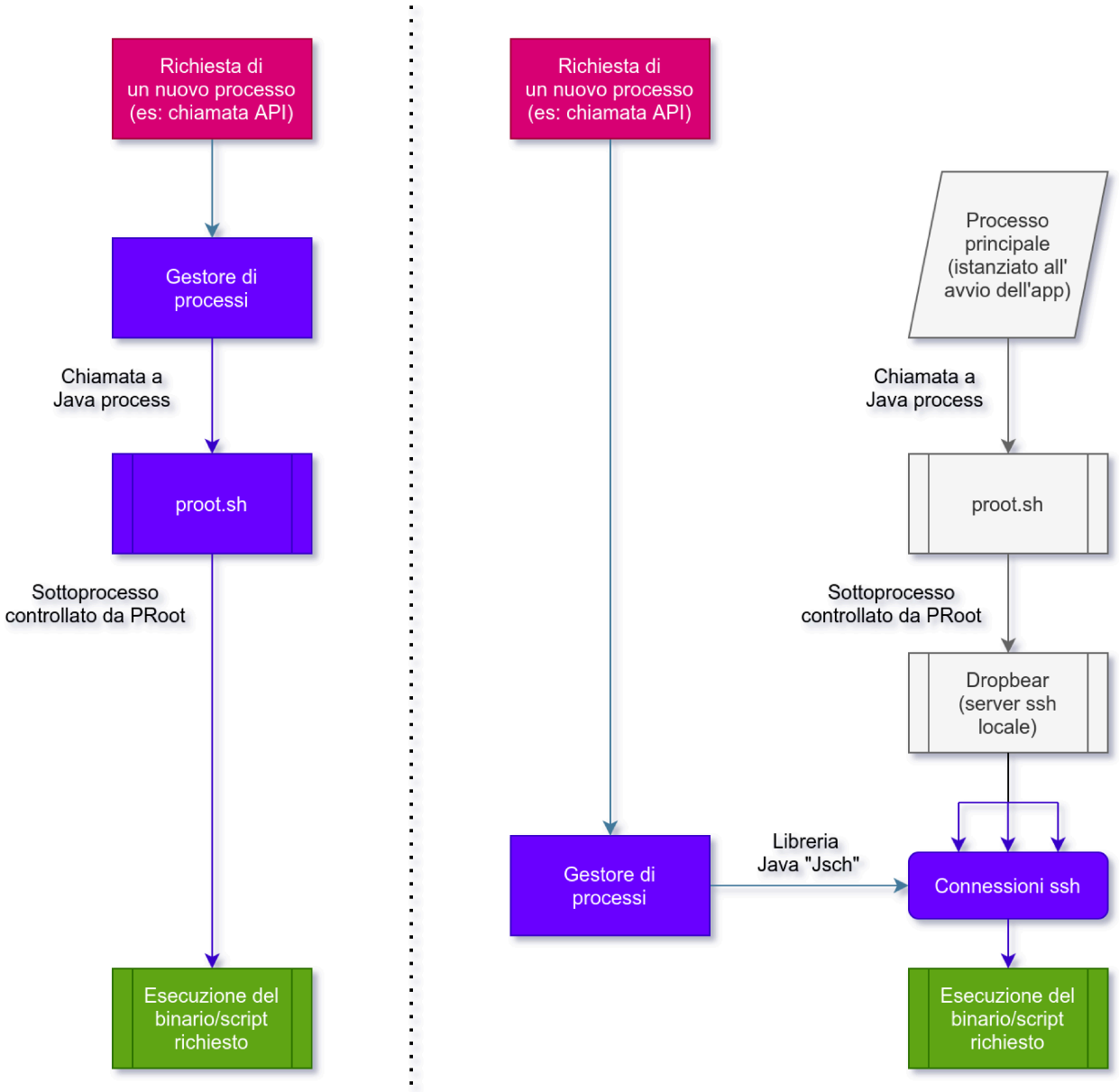

Figura 9: *Confronto fra le due implementazioni del gestore di processi (a destra quella che utilizza un server SSH locale)*

Per consentire alle web app di accedere al gestore di processi si è creato un modulo “process” all’interno dell’API. Si è scelto di fornire due tipi di accesso ai processi GNU/Linux:

- il primo, ottenibile utilizzando /process/run, è pensato per i processi rapidi che devono restituire in fretta un risultato e hanno pertanto un timeout fisso di 60 secondi per terminare la loro esecuzione, o verranno terminati dal sistema. Ad esempio, una web app potrebbe chiamare /process/run con body {"command": "ls /folder"} per ottenere la lista dei file nella cartella /folder .
- Il secondo, utilizzabile invocando /process/dtach è pensato per programmi che, una volta lanciati, non devono restituire un risultato, ma rimanere attivi in background. Tali processi vengono istanziati sfruttando il software Linux Dtach **[40]**, per l’esecuzione di programmi in background, che consente loro di essere controllati in futuro utilizzando un socket Unix. Dtach non include un meccanismo per terminare i processi lanciati, per questo l’API, ogni volta che lancia un processo, si preoccupa di salvare il suo process id in un file nella stessa cartella del socket Unix, così da poter usare il comando kill in seguito.

Viene inoltre fornita una funzione /process/screen, che consente di avviare un processo con Dtach e fare sì che sia visualizzato in primo piano all’interno dello schermo virtuale fornito da Xvfb, così da poterlo visualizzare nella web app tramite noVNC.js.

Per motivi di performance, si è scelto di utilizzare un’unica istanza di Xvfb (che viene avviata con l’avvio del service) e di gestire i processi grafici grazie al window manager bspwm. Si è scelto bspwm in quanto controllabile interamente da linea di comando. Con la configurazione opportuna, bspwm consente di far coesistere più programmi a interfaccia grafica, mantenendo quello desiderato come unico processo visibile e a schermo intero.

Web app diverse potrebbero desiderare di visualizzare app all’interno di schermi con dimensioni differenti: ad esempio, una web app potrebbe decidere di mostrare un terminale in modo che occupi metà schermo. In teoria, le versioni recenti di Xvfb includono un’estensione che consente di cambiare le dimensioni dello schermo emulato **[41]**. Nella pratica, però, tale operazione si è rivelata troppo lenta; si è quindi scelto di mantenere le dimensioni di Xvfb fisse e pari allo schermo del dispositivo, e di regolare di volta in volta lo spazio disponibile ai programmi GNU/Linux utilizzando la funzione “margini” di bspwm.

4.5 Web app (“Wrappers”)

Si è scelto di non imporre vincoli particolari sulla struttura delle web app; esse sono da considerarsi normali siti web e possono essere realizzate con qualsiasi tecnologia/framework esistente per creare web app. Lo stesso server API è agnostico: come spiegato in 4.4.2 per accedervi è sufficiente fare una chiamata HTTP a *localhost:12080* .

Come già anticipato, si è scelto di distribuire tali app utilizzando il package manager di Alpine linux: questo consente agli sviluppatori di indicare fra le dipendenze del pacchetto tutti i programmi che vogliono siano installati assieme alla web app (ad esempio, un wrapper che vuole utilizzare LibreOffice, può indicare “libreoffice” fra le sue dipendenze, e ha garantito che, al momento della sua esecuzione, LibreOffice sarà disponibile). Si è scelto di imporre la convenzione sul fatto che i wrapper vadano chiamati **unyw-app-NOMEAPP**, e vengano installati all’interno del rootfs nella cartella /usr/share/unyw/apps/NOMEAPP. Ciascuna web app deve avere inoltre un punto d’ingresso chiamato “index.html” (come da tradizione dei siti web) e un’icona chiamata “icon.png”.

Nella realizzazione del prototipo presentato, si è scelto di utilizzare l'approccio moderno alla scrittura delle web app, che prevede di utilizzare Node.js e Npm (Node Package Manager) per gestire le librerie JavaScript, Rollup.js per il processo di ottimizzazione del codice, a cui si aggiunge un framework che semplifichi la dichiarazione di interfacce HTML dinamiche. Nello specifico, si è scelto il framework Svelte, un framework minimale particolarmente indicato per la scrittura di web app a singola pagina (SPA, single page app) e che sta acquisendo molta popolarità negli ultimi anni **[42]**. Per facilitare la scrittura dei wrapper, si sono creati degli script e delle librerie JavaScript che automatizzino alcune operazioni ripetute; tali script sono descritti in 4.5.1, mentre i wrapper in sé sono trattati in 4.5.2.

4.5.1 Ambiente di sviluppo

La creazione di pacchetti per Alpine è particolarmente complessa e poco documentata (la wiki ufficiale, ad esempio, riporta informazioni obsolete e presenta addirittura errori di ortografia **[43]**). Si è deciso pertanto di creare uno script chiamato "build.js" che eseguisse le operazioni necessarie automaticamente per tutte le web app presenti. Tale script si occupa di:

- scaricare un rootfs all'interno del computer se non ci si trova all'interno di Alpine Linux e installare tramite PRoot tutte le dipendenze necessarie. Questo è necessario perché alcuni software richiesti per generare i pacchetti di Alpine Linux (come quello che firma i pacchetti digitalmente) sono disponibili solo nelle repository di Alpine Linux stesso.
- Se assente, generare una coppia di chiavi crittografiche pubblica/privata per consentire la firma digitale dei pacchetti. La chiave privata viene memorizzata all'interno del computer locale, mentre quella pubblica viene inserita all'interno della repository.
- Per ciascuno dei wrapper, si crea un manifest compatibile con il gestore di pacchetti di Alpine Linux a partire dalle informazioni nel file "package.json" (il manifest che hanno i progetti creati con Node.js).
- Si procede alla creazione dei pacchetti e alla loro firma digitale. Alpine Linux richiede che vengano fornite 4 versioni dei pacchetti, una per ogni architettura, anche se i pacchetti non contengono software binario specifico per una determinata architettura.
- Si conclude generando i file APKINDEX.tar.gz, che servono al gestore di pacchetti di Alpine per avere un elenco dei pacchetti disponibili all'interno della repository.

Si sono inoltre create due librerie JavaScript, che è possibile installare e importare tramite Node.js + Rollup; Tali librerie consentono di sfruttare con poche righe di codice tutto il potenziale dell'API:

- **@unyw/api**, che rende più immediato effettuare chiamate HTTP al server API, in particolare rende automatica l'operazione di richiesta del token (vedi 4.6). Si è scelto di progettare tale libreria in modo che sfruttasse tutte le potenzialità del linguaggio JavaScript, in particolare le JavaScript Promises, un modello per la gestione di codice asincrono introdotto nella revisione JavaScript ES2015 e attualmente standard. Tale libreria include anche un meccanismo, utile ai fini di debug, per collegarsi a un server API al di fuori di localhost: questo consente di lanciare e testare una web app sul proprio computer facendola collegare a un dispositivo remoto.
- **@unyw/ui**: una collezione di componenti grafici Svelte creati secondo i principi di Material Design. Oltre a componenti generici come barre di navigazione, pulsanti, icone, … si sono creati anche componenti già integrati con @unyw/api, tra cui VncViewer, un componente che consente di visualizzare lo schermo trasmesso da x11vnc tramite noVNC.js. Vi è inoltre

un componente, chiamato Keyboardbar, che include un set di tasti extra come CTRL, ALT, TAB, … che se cliccati verranno inviati direttamente allo schermo Xvfb tramite noVNC.js, andando così ad ampliare la tastiera dell'utente.

L'uso di tali librerie, unite all'efficienza di Svelte e in generale del JavaScript, rendono estremamente conciso il codice sorgente necessario per scrivere le web app. Ciò consente di realizzare wrapper con pochissime righe di codice, come evidenziato dal codice sottostante (**Codice 4**) che, in sole 32 linee, racchiude l'intero codice sorgente necessario per avviare chromium e visualizzarlo con noVNC.js, logica e stili inclusi.

```
<script>
 import {Keyboardbar, VncViewer} from '@unyw/ui'
 import Unyw from '@unyw/api'
  const socket = `chromium-app`

 Unyw().then( ({process}) => process.screen({
       socket,
       command: 'chromium-browser --no-sandbox'
   }))
</script>

<main>
 <VncViewer {socket}/>
 <Keyboardbar/>
</main>

<style>
 :global(body){
   margin: 0px;
   padding: 0px;
   font-family: 'Roboto', 'Arial', sans-serif;
   --color-primary: #000;
   -webkit-tap-highlight-color: transparent;
   overflow: hidden;
 }

 main {
   display: flex;
   flex-direction: column;
   height: 100%;
 }
</style>
```

Codice 4: *Wrapper* ***unyw-app-chromium***

4.5.2 Wrapper demo sviluppati

unyw-app-home: questa web app consente di avviare altri wrapper e di interfacciarsi con il gestore di pacchetti di Alpine Linux, in modo da aggiungere repository esterne e installare nuovi wrapper. Consente inoltre di visualizzare informazioni sulla versione di unyw attualmente in esecuzione e di modificare le impostazioni dell'app.

unyw-app-xterm: consente di avviare e utilizzare il terminale xterm, che viene visualizzato tramite noVNC.js. È in grado di gestire istanze multiple del programma, consentendo così all'utente di lanciare più processi contemporaneamente.

unyw-app-chromium: consente di avviare e utilizzare il browser chromium, che viene visualizzato tramite noVNC.js. Tale web app è stata realizzata come esempio di wrapper minimale.

unyw-app-vscode-server: avvia la versione server del popolare editor di codice VSCode, e visualizza l'interfaccia avviata tramite la webview (**nota:** allo stato attuale tale web app non funziona su dispositivi ARM64 con la libreria per un problema interno al pacchetto code-server che è in fase di risoluzione).

unyw-app-libreoffice-converter: questa web app consente di aprire un documento tramite il selettore file di Android, convertirlo in un formato a piacere usando il convertitore fornito da LibreOffice e condividere il risultato, facendolo aprire dall'app che l'utente preferisce. L'obiettivo è quello di mostrare un'interazione più avanzata fra i programmi GNU/Linux e l'API Android.

4.6 Esempio sviluppo wrapper: **unyw-app-libreoffice-converter**

In questa sezione si descriverà il processo di sviluppo di un wrapper prendendo ad esempio il wrapper **unyw-app-libreoffice-converter**. Il risultato sarà poi pubblicato nella repository repo-main, già esistente.

Come spiegato nella sezione 4.5.2, il wrapper **unyw-app-libreoffice-converter** ha lo scopo di fornire un'interfaccia grafica alla funzione "convertitore" inclusa in LibreOffice.

Per iniziare il progetto è conveniente partire dalla base fornita nella repository template-app, clonandola grazie a degit **[69]**, un software che viene distribuito attraverso Npm (Node Package Manager) e che consente di clonare le repository template con più facilità.

```
# Usa come cartella base quella di repo-main
cd repo-main

# Scarica una copia della repository attraverso degit
# Verrà creata una cartella "unyw-app-libreoffice-converter"
# all'interno di repo-main/packages
npx degit unyw/template-app packages/unyw-app-libreoffice-converter
```

Codice 5: *Clonazione di template-app*

Si modifica quindi il file manifest unyw-app-libreoffice-converter/package.json, inserendo il nome dell'app e aggiungendo LibreOffice fra le dipendenze.

```
{
 "name": "unyw-app-libreoffice-converter",
 ...
 "unyw-pkginfo": {
   "depend": [
     "unyw", "libreoffice"
   ]
 },
 ...
}
```

Codice 6: *Modifiche da effettuare a package.json*

Si modificano anche altre informazioni, come descrizione e autore della web app, e si aggiunge il file icona unyw-app-libreoffice/app/icon.png.

Si procede quindi a definire codice e interfaccia grafica all'interno del file unyw-app-libreoffice-converter/src/App.svelte, aggiungendo i vari componenti della web app:

- un pulsante "Open file" che quando cliccato chiama l'API /intent/filepicker, la quale chiede all'utente di scegliere un file dal proprio dispositivo e salva il file scelto come /storage/unyw/libreoffice-converter/document.
- Un componente "select" che permette di scegliere il formato in cui convertire il file.
- Un pulsante "Convert" che avvia l'applicazione xterm con /process/screen ed esegue lo script che si occupa di convertire il file scelto e di aprire il file convertito (vedi **Codice 7**).
- Un client VNC, realizzato con noVNC.js, che permette di visualizzare xterm mentre viene eseguito lo script, mostrando così eventuali errori. Tale componente è importato dalla libreria **@unyw/ui**, già inclusa nel template.

```
# Converte il documento nel formato pdf
libreoffice --headless --convert-to pdf                  \
  /storage/unyw/libreoffice-converter/document           \
  --outdir /storage/unyw/libreoffice-converter;

# Apre il risultato
unyw-api /intent/open \
 '{
    "file":"document.pdf",
    "folder": "libreoffice-converter",
    "mimetype":"application/pdf"
  }';
```

Codice 7: *Script per convertire il documento in PDF*

```
<script>
 import {Topbar, VncViewer} from '@unyw/ui'
 import Unyw from '@unyw/api'

 let format = 'pdf'
 const socket = `libreoffice-convert-app`

 const bashStr = str => `'${str.replaceAll(`'`, `'"'"'`)}'`

 const mimetypes = {
   "pdf": "application/pdf",
   "odt": "application/vnd.oasis.opendocument.text",
   "doc": "application/msword",
   "docx":
"application/vnd.openxmlformats-officedocument.wordprocessingml",
   "html": "text/html",
   "txt": "text/plain",
 }

</script>

<main>
 <Topbar>
   <h3>LibreOffice converter</h3>
 </Topbar>
 <div style="flex: 1 1 0; text-align:center; margin: 40px;">
   <div><button on:click={ () => Unyw().then( ({intent}) => {
     intent.filepicker({
       file: 'document',
       mimetype: 'application/*',
       folder: 'libreoffice-converter'
     })
   })}>Open file</button></div>
   <div><select bind:value={format}>
     <option value="pdf">pdf</option>
     <option value="odt">odt</option>
     <option value="doc">doc</option>
     <option value="docx">docx</option>
     <option value="html">html</option>
     <option value="txt">txt</option>
   </select></div>
   <div><button on:click={ () => {
     Unyw().then( async({process, intent}) => {
       await process.kill({
```

```
          socket
        })

        await process.screen({
          socket,
          command: `xterm -e  sh -c ${bashStr(`echo "Converting
document..."; libreoffice --headless --convert-to ${format}
/storage/unyw/libreoffice-converter/document --outdir
/storage/unyw/libreoffice-converter; `+
          `unyw-api /intent/open ${bashStr(`{"file":"document.${format}",
"folder": "libreoffice-converter", "mimetype":"${mimetypes[format]}"}`)};
echo ""; echo "Done!"; read`)}`
        })
      })
    }}>Convert</button></div>
 </div>
 <h3 style="margin: 4px 16px;"><b><u>Logs:</u></b></h3>
 <div style="height: 40vh; box-sizing: border-box; margin: 10px">
   <VncViewer {socket}/>
 </div>
</main>

<style>
 :global(body){
   --color-primary: rgb(22, 131, 0);
   overflow: hidden;
 }

 main {
   display: flex;
   flex-direction: column;
   height: 100%;
   align-content: center;
 }

 button {
   margin: 5px 0;
   padding: 10px 20px;
   border-radius: 20px;
 }
</style>
```

Codice 8: *Wrapper* ***unyw-app-libreoffice-converter***

Una volta terminata la scrittura del wrapper, è possibile prepararlo alla distribuzione, eseguendo lo script "build.js" (vedi sezione 4.5.1) di repo-main, e poi pubblicarlo grazie a Github Pages **[35]**, come mostrato nel **Codice 9**.

Il wrapper verrà immediatamente visualizzato all'interno della schermata Home dell'app Unyw e l'utente potrà installarlo senza ulteriori passaggi.

```
# Usa come cartella base quella di repo-main
cd repo-main

# Crea un pacchetto Alpine a partire dal wrapper
node build.js packages/unyw-app-libreoffice-converter

# Pubblica il pacchetto su Github
git add .
git commit -m "New libreoffice-converter package"
git push
```

Codice 9: *Preparazione e pubblicazione di* ***unyw-app-libreoffice-converter***

4.7 Cenni sulla sicurezza dell'app

Non è nello scopo di questa tesi effettuare una revisione accurata della sicurezza della soluzione proposta. Ci si limiterà, pertanto, a dare dei cenni generali su quali siano le criticità più importanti e si proporrà una possibile strategia per risolverle.

Tutti i programmi eseguiti all'interno dell'app proposta sono soggetti alle misure di sicurezza di Android. Pertanto, l'esecuzione dei programmi in sé non costituisce rischio per la sicurezza del sistema.

L'app utilizza però protocolli di rete per comunicare internamente, in particolare:

- l'API viene fornita come server HTTP, alla porta 12080;
- Xvfb crea uno schermo accessibile alla porta 6000;
- VNC e SSH sono comunque basati sul protocollo di rete tcp, rispettivamente alle porte 12081 e 12082.

Si evidenzia come fra queste comunicazioni non sia presente il meccanismo che consente di inviare alla Webview i file che vi devono essere caricati: tale meccanismo è interno a Java e non apre porte di rete.

La presenza di queste porte aperte potrebbe esporre il sistema a falle di sicurezza, perché un programma al di fuori dell'app potrebbe connettersi ai canali elencati per prendere il controllo e/o danneggiare l'app e le risorse a cui questa ha accesso. Questo è particolarmente grave dato che l'app, visto che deve fornire una API varia, all'installazione deve chiedere l'autorizzazione ad accedere a numerose risorse.

Per risolvere il problema, innanzitutto, è necessario sincerarsi che nessuna delle porte elencate sia accessibile al di fuori di localhost. La soluzione proposta può facilmente adeguarsi a questo requisito, perché in tutti i casi elencati esiste un'opzione per limitare le connessioni a quelle in arrivo dal solo dispositivo ("localhost"); ad esempio per Xvbf è sufficiente iniziare il processo con --no-listen tcp. Un vantaggio delle connessioni localhost è che sono gestite interamente dal kernel, non è quindi possibile intercettarle se il dispositivo non è manomesso. Questo rende sicura la trasmissione di dati sensibili.

Evitare le connessioni esterne non è sufficiente a garantire la sicurezza, perché i canali localhost potrebbero essere usati impropriamente se app malevole sono presenti all'interno del dispositivo: ad esempio un'app che non ha l'autorizzazione a scrivere file su disco potrebbe sfruttare l'API per farlo (si parla in questi casi di Confused Deputy Problem **[44]**). È quindi necessario che tutti i protocolli usati per comunicare internamente non siano accessibili alle altre app sul dispositivo:

1) I programmi GNU/Linux eseguiti dall'app devono essere certi di comunicare con i server dell'app e non con un'altra app del dispositivo, magari intenzionata a rubare le informazioni ricevute. Ciò è facilmente ottenibile, perché tutte le comunicazioni avvengono attraverso porte del protocollo IP, e tali porte, una volta riservate, appartengono a un processo finché non le libera o viene interrotto. È sufficiente che l'app, all'avvio, inizializzi i vari servizi chiedendo di usare porte fisse: nel caso siano tutte disponibili, essere apparterranno all'app finché non viene terminata. Nel caso in cui alcune di queste porte dovessero risultare non disponibili (ad esempio perché un'altra app tenta di intercettarle o le sta usando), l'app proposta dovrebbe evitare di avviarsi, spiegando all'utente la situazione; in quel caso non ci sarebbero, quindi, processi che tentano di collegarsi.
2) I vari server offerti dall'app devono essere certi di soddisfare solo e unicamente le richieste provenienti da altri componenti dell'app stessa. Ciò si può realizzare mediante l'uso di token, ossia password momentanee generate casualmente che consentono l'accesso a una risorsa. Nel prototipo proposto, all'avvio del service, questo genera un token e lo scrive all'interno del file `/tmp/UNYW_TOKEN_API`. Essendo all'interno dei file privati dell'app, solo i programmi dell'app stessa potranno accedervi (la webview non vi ha accesso nativo, ma essendo un protocollo interno a Java, è possibile rendere disponibile tale file in sicurezza rimappandolo sull'url `localhost:12079/UNYW_TOKEN_API.json`). A quel punto, si chiede ai programmi di leggere tale file e di inviare il suo risultato come parametro ogni volta che devono effettuare una chiamata all'API. Una volta resa sicura l'API, è possibile usare l'API stessa per distribuire gli altri token (come ad esempio le password di VNC e SSH). Xvfb non prevede la possibilità di usare una password, ma non è un problema rilevante, perché il massimo ottenibile inserendosi al suo interno è avviare programmi a interfaccia grafica non richiesti.

Nella pratica, è impossibile ottenere risultati migliori e più granulari, a meno di cambiare radicalmente la struttura dell'app. Questo perché, come già anticipato, PRoot non permette realmente di imporre dei permessi sul filesystem: una web app malevola potrebbe sostituire i programmi di sistema con la propria versione infetta, radicandosi così nel sistema.

4.8 Test comparativi

Per validare la soluzione proposta a livello di funzionalità offerte e prestazioni, si sono usati gli stessi test già descritti nella sezione 3.3. I risultati ottenuti sono stati riportati anche in 3.3, così da rendere più immediato il confronto.

Funzionalità presenti (vedi 3.3.1):

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Unyw (soluzione proposta)	Sì	Sì	Sì	Sì	Sì	No	Sì	Sì	No	0	Sì	Sì

La soluzione proposta, pertanto, è quella con il maggior numero di funzionalità fra quelle considerate. Delle due funzionalità mancanti:

- la 6 (supporto per l'accelerazione hardware) è attualmente un problema praticamente impossibile da risolvere;
- la 9 (compatibilità con i file .iso, kernel incluso) per definizione non può essere ottenuta dalle soluzioni OS-Level come PRoot, che, come discusso in 3.2, hanno come prerogativa di evitare l'emulazione del kernel.

Esecuzione di fibonacci.sh (vedi 3.3.2.2):

2:03	2:01	1:57	1:58	2:00	2:01	1:58	2:01	2:02	1:58	(Avg) 2:00

L'app ottiene performance comparabili o superiori a tutte le altre soluzioni considerate. In particolare, il risultato è identico a quello ottenuto dalle altre soluzioni che utilizzano PRoot e Alpine Linux.

Dimensioni app (vedi 3.3.3 e 3.3.3.2):

Nome app	**Dimensioni app (Mb)**	**Dati utenti (Mb)**	**Totale (Mb)**
Unyw (soluzione proposta, senza il supporto Pulseaudio)	10	48	58
Unyw (soluzione proposta, includendo Pulseaudio)	15	71	86

Nonostante non sia stata compiuta alcuna ottimizzazione sulle dimensione del file apk, in particolare per quanto riguarda Pulseaudio, Unyw richiede solo 86Mb per essere installata e risulta quindi più leggera di tutte le soluzioni analizzate in precedenza. Questo può essere spiegato sia considerando che è stato scelto Alpine Linux, sia perché usare i wrapper uniti a bspwm consente di evitare l'installazione di un window manager più pesante.

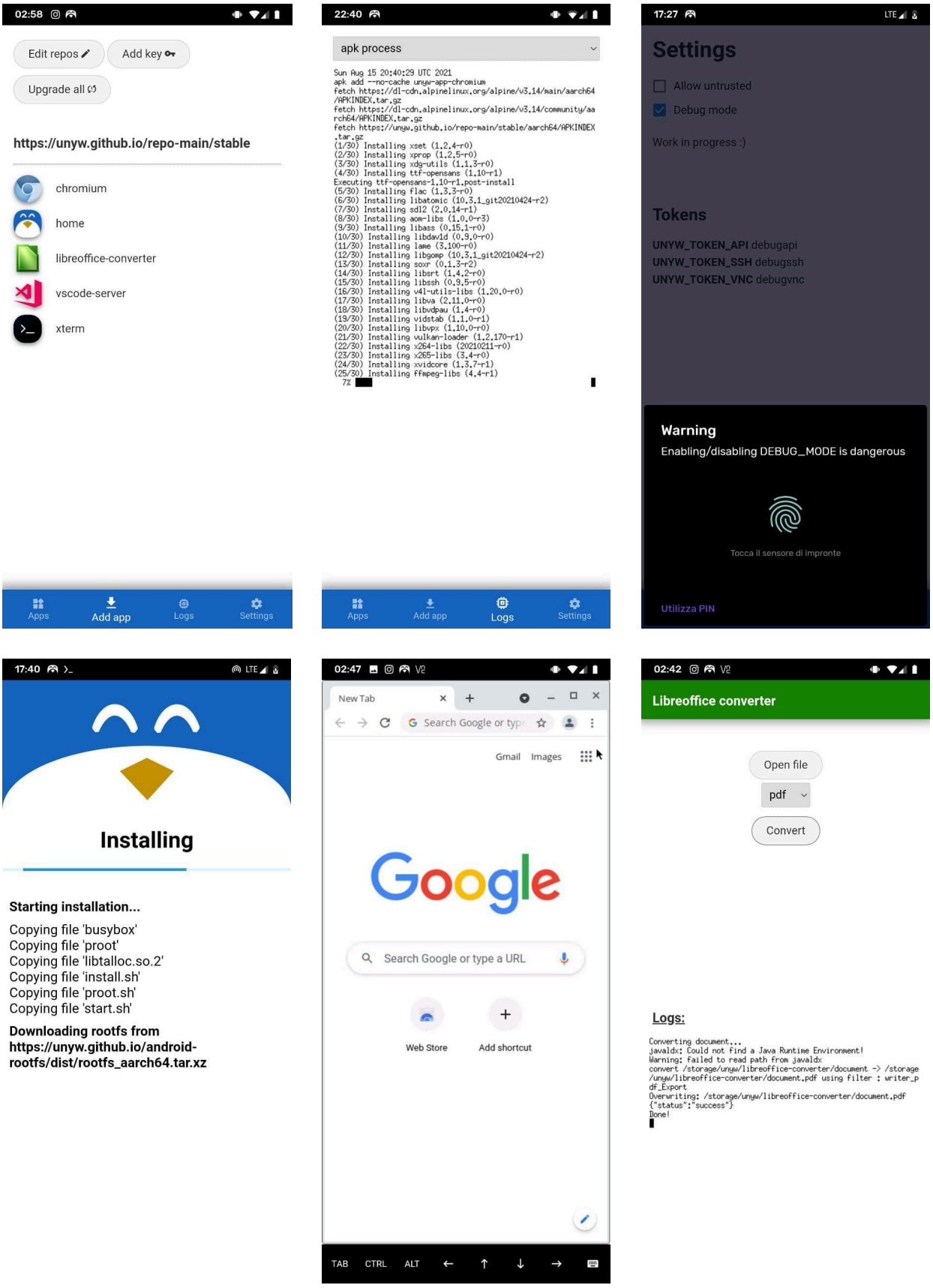

Figura 10: *Screenshot di Unyw:* ***unyw-app-home*** *(in alto), schermata di installazione (in basso a sinistra)* ***unyw-app-chromium*** *(in basso al centro) e* ***unyw-app-libreoffice-converter*** *(in basso a destra)*

5. Conclusioni

5.1 Risultati ottenuti

Lo scopo di questa tesi era fornire un modo per eseguire programmi GNU/Linux adattandoli al sistema Android. Tale obiettivo è stato raggiunto, infatti, si è riusciti a programmare un prototipo che rispetta tutte le specifiche richieste:

- è installabile con facilità e tempi minimi;
- offre prestazioni comparabili a quelle delle soluzioni più efficienti analizzate, richiede il minor spazio su disco in assoluto e presenta il maggior numero di funzionalità fra quelle considerate;
- permette l'esecuzione di programmi GNU/Linux sia a linea di comando che a interfaccia grafica, senza bisogno di utilizzare app esterne;
- è espandibile (e sono già stati creati strumenti e librerie che aiutino nella scrittura di estensioni, come visto in 4.5.1);
- consente di realizzare nuove interfacce ai programmi GNU/Linux tramite poche linee di codice.

Si è inoltre argomentato come nella quasi totalità dei casi le funzionalità non ottenute, come la distribuzione tramite Google Play Store e l'implementazione dell'accelerazione hardware, siano impossibili da ottenere dal punto di vista pratico, con l'eccezione dell'approccio presentato in 4.4.2.1, che rimane un problema aperto.

5.2 Modifiche necessarie in vista di una distribuzione

Come già ripetuto, il prototipo realizzato è incompleto, in quanto ha il solo scopo di dimostrare i concetti espressi in questa tesi. Il progetto può pertanto considerarsi concluso solo a livello teorico. Prima di una eventuale distribuzione sarebbe infatti necessario:

- creare numerosi wrapper oltre a quelli di esempio presentati in 4.5.2;
- formalizzare uno standard per l'API, e implementare tutte le funzionalità mancanti, fra cui un'estesa mappatura dei componenti hardware con cui Android può interagire e aggiungendo in generale moduli più estesi;
- creazione di una dettagliata documentazione sul progetto, che consenta a terzi di implementare le proprie web app;
- testare l'app ottenuta su vari dispositivi e varie versioni Android, correggendo le eventuali incongruenze riscontrate. In particolare, va indagata maggiormente come varia la gestione degli input provenienti da tastiera: la soluzione attuale utilizza esclusivamente le opportunità offerte dall'HTML, il cui standard tuttavia non gestisce in maniera univoca elementi come il correttore automatico, generando risultati diversi da dispositivo a dispositivo; per garantire un'esperienza omogenea potrebbe pertanto essere necessario delegare la gestione di tali eventi al codice Kotlin dell'activity.
- Risolvere il problema presentato 4.4.2.1, in modo da integrare un modulo che consenta di sfruttare Java Reflection

Sitografia

[1] “Android Kernel Features - eLinux.org.” https://elinux.org/Android_Kernel_Features (accessed Aug. 22, 2021).

[2] A. Ltd., “Linux Kernel,” *Arm Developer*. https://developer.arm.com/tools-and-software/open-source-software/linux-kernel/energy-aware-scheduling (accessed Aug. 22, 2021).

[3] “Filesystems in the Linux kernel — The Linux Kernel documentation.” https://www.kernel.org/doc/html/latest/filesystems/index.html (accessed Aug. 22, 2021).

[4] Contributors to Wikimedia projects, “Ptrace,” *Wikipedia*, Jun. 08, 2021. https://en.wikipedia.org/wiki/Ptrace (accessed Aug. 22, 2021).

[5] “Android ABIs ,” *Android Developers*. https://developer.android.com/ndk/guides/abis (accessed Aug. 22, 2021).

[6] “SupportedArchitectures - Debian Wiki.” https://wiki.debian.org/SupportedArchitectures (accessed Aug. 22, 2021).

[7] Contributors to Wikimedia projects, “Bionic (software),” *Wikipedia*, May 31, 2021. https://en.wikipedia.org/wiki/Bionic_(software) (accessed Aug. 22, 2021).

[8] “LLVMpipe — The Mesa 3D Graphics Library latest documentation.” https://docs.mesa3d.org/drivers/llvmpipe.html (accessed Aug. 22, 2021).

[9] “Window manager - ArchWiki.” https://wiki.archlinux.org/title/window_manager (accessed Aug. 22, 2021).

[10] nwrkbiz, “GitHub - nwrkbiz/android-xserver: Maintaining the original project to make it work again with new Android versions.,” *GitHub*. https://github.com/nwrkbiz/android-xserver (accessed Aug. 22, 2021).

[11] pelya, “GitHub - pelya/commandergenius: Port of SDL library and several games to the Android OS.,” *GitHub*. https://github.com/pelya/commandergenius (accessed Aug. 22, 2021).

[12] Contributors to Wikimedia projects, “Virtual Network Computing,” *Wikipedia*, Apr. 22, 2021. https://en.wikipedia.org/wiki/Virtual_Network_Computing (accessed Aug. 22, 2021).

[13] pelya, “What about opengl support? · Issue #45 · pelya/xserver-xsdl,” *GitHub*. https://github.com/pelya/xserver-xsdl/issues/45#issuecomment-127371602 (accessed Aug. 22, 2021).

[14] “React Native Internals.” https://www.reactnative.guide/3-react-native-internals/3.1-react-native-internals.html (accessed Aug. 22, 2021).

[15] “Device and Network Abuse - Play Console Help.” https://support.google.com/googleplay/android-developer/answer/9888379 (accessed Aug. 22, 2021).

[16] “Background Execution Limits ,” *Android Developers*. https://developer.android.com/about/versions/oreo/background (accessed Aug. 22, 2021).

[17] golem, “What does the @ symbol denote in the beginning of a unix domain socket path in Linux?,” *Unix & Linux Stack Exchange*. https://unix.stackexchange.com/questions/206386/what-does-the-symbol-denote-in-the-beginning-of-a-unix-domain-socket-path-in-l (accessed Aug. 22, 2021).

[18] “TCP loopback connection vs Unix Domain Socket performance,” *NewbeDEV*. https://newbedev.com/tcp-loopback-connection-vs-unix-domain-socket-performance (accessed Aug. 22, 2021).

[19] “Ubuntu flavours,” *Ubuntu*. https://ubuntu.com/download/flavours (accessed Aug. 22, 2021).

[20] Contributors to Wikimedia projects, “Hypervisor,” *Wikipedia*, Jul. 23, 2021. https://en.wikipedia.org/wiki/Hypervisor (accessed Aug. 22, 2021).

[21] “How many x86 instructions are there?,” *The ryg blog*, Aug. 25, 2016. https://fgiesen.wordpress.com/2016/08/25/how-many-x86-instructions-are-there/ (accessed Aug. 22, 2021).

[22] L. Lisen, “Bochs.” https://play.google.com/store/apps/details?id=net.sourceforge.bochs (accessed Aug. 22, 2021).

[23] “Difference between Emulation and Virtualization - javatpoint,” *www.javatpoint.com*. https://www.javatpoint.com/emulation-vs-virtualization (accessed Aug. 22, 2021).

[24] “PRoot — chroot, mount --bind, and binfmt_misc without privilege/setup.” https://proot-me.github.io/ (accessed Aug. 22, 2021).

[25] J. M. Mbongue, D. T. Kwadjo, and C. Bobda, “Performance Exploration of Virtualization Systems,” *arXiv.org*, Mar. 12, 2021. https://arxiv.org/abs/2103.07092 (accessed Aug. 22, 2021).

[26] “Package Management - Termux Wiki.” https://wiki.termux.com/wiki/Package_Management (accessed Aug. 22, 2021).

[27] “Differences from Linux - Termux Wiki.” https://wiki.termux.com/wiki/Differences_from_Linux (accessed Aug. 22, 2021).

[28] termux, “x11-packages/packages at master · termux/x11-packages,” *GitHub*. https://github.com/termux/x11-packages/tree/master/packages (accessed Aug. 22, 2021).

[29] termux, “[README] Bintray repositories are down now,” *GitHub*. https://github.com/termux/termux-packages/issues/6726 (accessed Aug. 22, 2021).

[30] “AnLinux.” https://f-droid.org/en/packages/exa.lnx.a/ (accessed Aug. 22, 2021).

[31] AndronixApp, “GitHub - AndronixApp/AndronixOrigin” *GitHub*. https://github.com/AndronixApp/AndronixOrigin (accessed Aug. 22, 2021).

[32] CypherpunkArmory, “GitHub - CypherpunkArmory/UserLAnd: Main UserLAnd Repository,” *GitHub*. https://github.com/CypherpunkArmory/UserLAnd (accessed Aug. 23, 2021).

[33] pelya, “No audio · Issue #35 · pelya/debian-noroot,” *GitHub*. https://github.com/pelya/debian-noroot/issues/35 (accessed Aug. 23, 2021).

[34] “Migrating to WebView in Android 4.4 ,” *Android Developers*. https://developer.android.com/guide/webapps/migrating (accessed Aug. 23, 2021).

[35] “GitHub Pages,” *GitHub Pages*. https://pages.github.com/ (accessed Aug. 23, 2021).

[36] M. Stapelberg, “Linux package managers are slow (2019) - Michael Stapelberg.” https://michael.stapelberg.ch/posts/2019-08-17-linux-package-managers-are-slow/ (accessed Aug. 23, 2021).

[37] “Analysis of HTTP Performance Problems.” https://www.w3.org/Protocols/HTTP-NG/http-prob.html (accessed Aug. 23, 2021).

[38] “Using Java Reflection.” https://www.oracle.com/technical-resources/articles/java/javareflection.html (accessed Aug. 23, 2021).

[39] Contributors to Wikimedia projects, “NativeScript,” *Wikipedia*, Aug. 06, 2021. https://en.wikipedia.org/wiki/NativeScript (accessed Aug. 23, 2021).

[40] “dtach(1) - Linux man page.” https://linux.die.net/man/1/dtach (accessed Aug. 23, 2021).

[41] Erik, “Resizing an Xvfb display,” *Stack Overflow*. https://stackoverflow.com/questions/6356169/resizing-an-xvfb-display (accessed Aug. 23, 2021).

[42] “Stack Overflow Developer Survey 2021,” *Stack Overflow*. https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks (accessed Aug. 23, 2021).

[43] “Alpine package format - Alpine Linux.” https://wiki.alpinelinux.org/wiki/Alpine_package_format (accessed Aug. 23, 2021).

[44] Contributors to Wikimedia projects, “Confused deputy problem,” *Wikipedia*, Jul. 04, 2021. https://en.wikipedia.org/wiki/Confused_deputy_problem (accessed Aug. 23, 2021).

[45] “Comparison of C/POSIX standard library implementations for Linux.” http://www.etalabs.net/compare_libcs.html (accessed Aug. 23, 2021).

[46] dront78, “GitHub - dront78/PulseDroid: Android network audio player,” *GitHub*. https://github.com/dront78/PulseDroid (accessed Aug. 29, 2021).

[47] “Modules – PulseAudio.” https://www.freedesktop.org/wiki/Software/PulseAudio/Documentation/User/Modules/ (accessed Aug. 29, 2021).

[48] Anbox, “Anbox,” *Android in a Box*. https://anbox.io/ (accessed Sep. 16, 2021).

[49] “Enterprise Open Source and Linux,” *Ubuntu*. https://ubuntu.com/ (accessed Sep. 16, 2021).

[50] “Debian -- The Universal Operating System.” https://www.debian.org/ (accessed Sep. 16, 2021).

[51] “Arch Linux.” https://archlinux.org/ (accessed Sep. 16, 2021).

[52] “index,” *Alpine Linux*. https://www.alpinelinux.org/ (accessed Sep. 16, 2021).

[53] “Tiny Core Linux, Micro Core Linux, 12MB Linux GUI Desktop, Live, Frugal, Extendable.” http://tinycorelinux.net/ (accessed Sep. 16, 2021).

[54] libhybris, “GitHub - libhybris/libhybris: Hybris is a solution that commits hybris, by allowing us to use bionic-based HW adaptations in glibc systems,” *GitHub*. https://github.com/libhybris/libhybris (accessed Sep. 17, 2021).

[55] “GL4ES - The OpenGL driver for GLES Hardware,” *GL4ES - The OpenGL driver for GLES Hardware*. http://ptitseb.github.io/gl4es/ (accessed Sep. 17, 2021).

[56]“Snapcraft - Snaps are universal Linux packages,” *Snapcraft*. https://snapcraft.io/ (accessed Sep. 17, 2021).

[57] “Flatpak—the future of application distribution.” https://flatpak.org/ (accessed Sep. 17, 2021).

[58] “AppImage,” *Linux apps that run anywhere*. https://appimage.org/ (accessed Sep. 17, 2021).

[59] L. B., “lBochs PC Emulator,” *Apps on Google Play*. https://play.google.com/store/apps/details?id=lb.myapp.lbochs (accessed Sep. 17, 2021).

[60] limboemu, “Home · limboemu/limbo Wiki,” *GitHub*. https://github.com/limboemu/limbo/wiki (accessed Sep. 17, 2021).

[61] “QEMU.” https://www.qemu.org/ (accessed Sep. 17, 2021).

[62] “KVM.” https://www.linux-kvm.org/page/Main_Page (accessed Sep. 17, 2021).

[63] “Termux.” https://termux.com/ (accessed Sep. 17, 2021).

[64] termux, “GitHub - termux/proot-distro: An utility for managing installations of the Linux distributions in Termux.,” *GitHub*. https://github.com/termux/proot-distro (accessed Sep. 17, 2021).

[65] CypherpunkArmory, “GitHub - CypherpunkArmory/UserLAnd: Main UserLAnd Repository,” *GitHub*. https://github.com/CypherpunkArmory/UserLAnd (accessed Sep. 17, 2021).

[66] pelya, “GitHub - pelya/debian-noroot: Debian running on Android, without rooting your phone,” *GitHub*. https://github.com/pelya/debian-noroot (accessed Sep. 17, 2021).

[67] “r/termux - CPU Battle : Termux vs Linux proot,” *reddit*. https://www.reddit.com/r/termux/comments/hvzp9x/cpu_battle_termux_vs_linux_proot/ (accessed Sep. 17, 2021).

[68] Contributors to Wikimedia projects, “Garbage collection (computer science),” *Wikipedia*, Jun. 29, 2021. https://en.wikipedia.org/wiki/Garbage_collection_(computer_science) (accessed Sep. 17, 2021).

[69] Rich-Harris, “GitHub - Rich-Harris/degit: Straightforward project scaffolding,” GitHub. https://github.com/Rich-Harris/degit (accessed Sep. 18, 2021).

[70] “noVNC.” https://novnc.com/info.html (accessed Sep. 18, 2021).

[71] C. ai progetti Wikimedia, “Live coding,” Wikipedia, Sep. 07, 2021. https://it.wikipedia.org/wiki/Live_coding (accessed Sep. 18, 2021).