UNIVERSITÀ DEGLI STUDI DI PERUGIA

DIPARTIMENTO DI INGEGNERIA

Corso di laurea in INGEGNERIA INFORMATICA ED ELETTRONICA

TESI DI LAUREA

STUDIO E SVILUPPO DI UN SISTEMA PER L'AGGIORNAMENTO SOFTWARE OVER-THE-AIR (OTA) PER APPLICAZIONI AUTOMOTIVE

LAUREANDO

Gabriele Fantini

RELATORE

Prof. Emilio Di Giacomo

Anno Accademico 2018-2019

“You should enjoy the little detours to the fullest. Because that's where you'll find the things more important than what you want.”

Ging Freecss

Indice

Introduzione

Questa tesi nasce da una proposta di collaborazione dell'azienda ART S.p.a. di Passignano sul Trasimeno con il Dipartimento di Ingegneria dell'Università degli Studi di Perugia. Lo scopo principale del progetto è lo studio di una soluzione che consenta l'aggiornamento over-the-air (OTA) di una delle centraline auto sviluppate e prodotte dall'azienda. In particolare l'aggiornamento software dell'MCU (MicroController Unit) della centralina, basato su OS Linux con una distribuzione personalizzata da ART stessa. Poiché tale sistema è altamente specializzato e complesso, lavorare su di esso richiede molto tempo: si è quindi optato per l'utilizzo di Raspberry Pi 3 b+ come piattaforma hardware e di una distribuzione linux come sistema operativo, creata mediante ambiente di cross-compilazione[1] Yocto. Questo ha permesso lo studio e la realizzazione di un sistema funzionante per aggiornamenti OTA e al contempo ha reso possibile un'eventuale operazione di porting sul sistema proprietario.

[1]**Cross-compilazione**: Tecnica mediante la quale si compila un codice sorgente con un cross-compilatore, ottenendo così un file binario eseguibile su di un elaboratore con architettura diversa da quella della macchina su cui è stato lanciato il cross-compilatore stesso.

Gli aggiornamenti OTA

Un sistema embedded identifica genericamente tutti quei sistemi elettronici di elaborazione a microprocessore progettati appositamente per un determinato utilizzo e non riprogrammabili dall'utente per altri scopi. La motivazione principale che spinge un produttore ad aggiornare il proprio sistema embedded è quello di fornire miglioramenti del sistema, aggiornamenti delle funzionalità, correzioni e patch di sicurezza. Esistono due metodologie diverse per effettuare l'aggiornamento di un dispositivo: quella manuale e quella "over-the-air" (OTA). L'aggiornamento manuale era l'unica alternativa prima dell'introduzione della tecnologia OTA ed è ancora utilizzato in alcune situazioni. Esso prevede il download del nuovo software in un dispositivo di archiviazione, come un hard disk o una chiavetta USB, per poi connetterlo all'apparecchio da aggiornare. In molti casi quest'ultimo va preso dal suo alloggiamento, riprogrammato, riassemblato e riposizionato così da funzionare nuovamente. Nonostante sia senza dubbio una procedura sicura e che garantisce il funzionamento ad aggiornamento terminato, ha numerosi svantaggi: il primo, e il più evidente, è il grande dispendio di tempo che occorre per aggiornare una singola unità e che ha come effetto collaterale la difficoltà nel gestire campagne di aggiornamento che riguardano flotte numerose di apparecchi. Il secondo è il fatto che il dispositivo va raggiunto fisicamente, causando problemi qualora questo si trovi in aree difficilmente raggiungibili. Oppure genera disagi, come nel caso delle centraline nel settore automotive, le quali richiedono al proprietario di recarsi in concessionaria ogni volta che occorre aggiornare la vettura. La metodologia OTA prevede l'aggiornamento da remoto. Il nuovo software viene consegnato tramite connessione wireless direttamente al dispositivo, senza necessitare di componenti hardware aggiuntivi. Gli aggiornamenti OTA sfruttano tipicamente connessioni internet a banda larga ma in alcuni casi possono usufruire anche di dati cellulari. Evitando tutto il processo richiesto dagli aggiornamenti manuali, gli OTA offrono la possibilità di aumentare la frequenza degli aggiornamenti, fornendo costantemente nuove features, risolvendo bug e migliorando il comportamento del prodotto anche mentre che questo è nelle mani del consumatore. Tutto questo riducendo i costi.

Problematiche da affrontare

Nonostante i numerosi vantaggi che gli OTA offrono, le problematiche che si vanno ad affrontare sono molte e di differente natura, specie quando si devono aggiornare sistemi critici il cui funzionamento corretto può risultare vitale, come ad esempio le centraline delle automobili.

Affidabilità

Per affidabilità si intende la garanzia che il sistema, in seguito al tentativo di aggiornamento, funzioni correttamente e svolga tutti i compiti più importanti per i quali è stato pensato, sia nel caso che questo sia andato a buon fine, sia che ci siano state delle complicazioni. Questo problema è risolto per la maggior parte dei casi mediante aggiornamenti a due o più partizioni. Gli aggiornamenti a due partizioni, chiamati “aggiornamenti A/B senza soluzione di continuità”, garantiscono che almeno un software funzionante ed eseguibile rimanga sul disco durante l’aggiornamento OTA. Come si può osservare dalla Figura 1, la partizione A è quella attualmente funzionante e che contiene il vecchio firmware[2]: il software che gestisce l’OTA scarica nella partizione B il nuovo firmware, si accerta che questo funzioni correttamente e solo allora segnala al bootloader[3] di cambiare partizione di avvio. Se tutto ciò non bastasse e all’avvio la partizione B dovesse avere problemi, la vecchia partizione A sarebbe ripristinata come predefinita, lasciando la B inattiva.

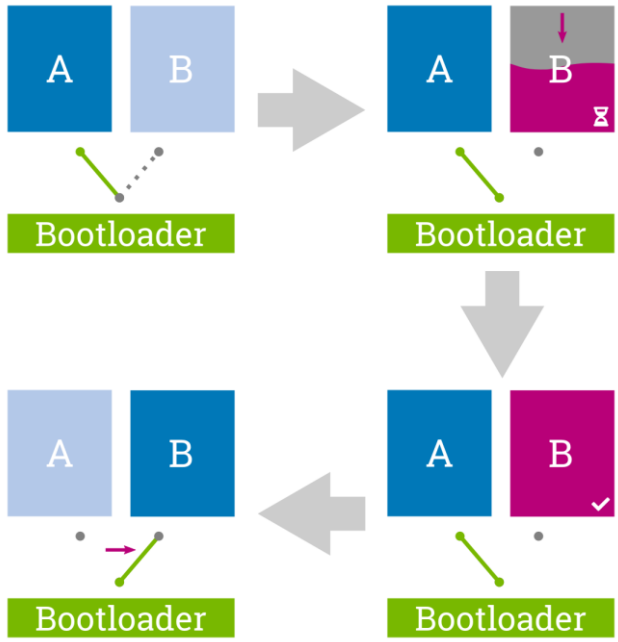


Figura 1

[2]**Firmware**: insieme delle istruzioni e delle applicazioni presenti permanentemente nella memoria di un sistema e che non possono essere modificate dall'utente.
[3]**Bootloader**: programma che in fase di boot del computer carica il kernel del SO dalla memoria secondaria a quella primaria, permettendone l’esecuzione da parte del processore.

Sicurezza

Soddisfare determinati criteri di sicurezza non solo protegge il sistema da manomissioni, ma evita che alcuni utenti riescano ad installare software non ufficiali e difende la proprietà intellettuale dell'azienda. Se negli aggiornamenti manuali è più facile garantire la sicurezza, negli OTA l'aggiornamento è trasmesso via Internet, con la possibilità di passare attraverso reti non adeguatamente protette, esponendosi così a molti più rischi di attacchi da parte di malintenzionati. Adottare meccanismi che ne garantiscano la sicurezza è dunque molto importante per un sistema di aggiornamenti OTA.

Autenticità

L'autenticità garantisce la provenienza del software che sarà installato. Spesso viene garantita assieme all'integrità, per mezzo di chiavi che "firmano" l'intero aggiornamento. Le chiavi vengono verificate per mezzo di certificati rilasciati da un'infrastruttura denominata PKI (public key infrastructure), la quale gestisce la loro validità e può revocarli in base alle necessità. La Figura 2 mostra un generico schema di una PKI.

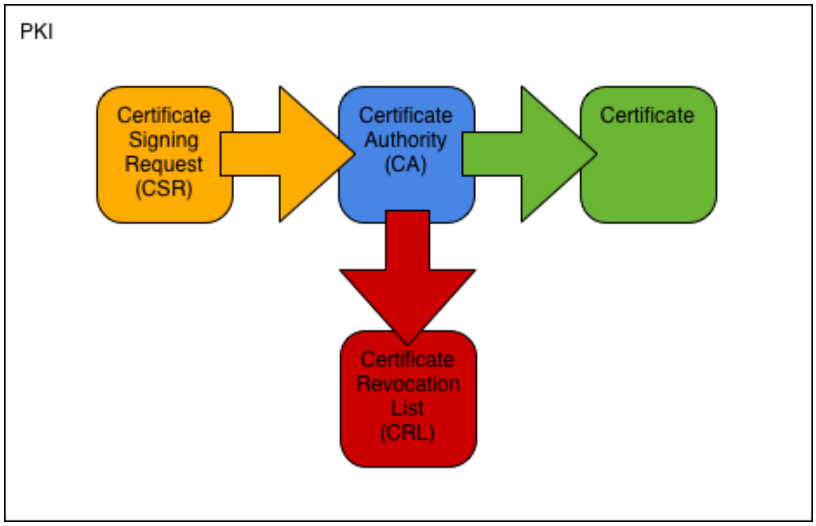


Figura 2

Questo meccanismo permette di gestire i permessi che gli sviluppatori hanno, per esempio rilasciando certificati per chiavi abilitate solo in fase di sviluppo e altre solo in fase di rilascio, oppure chiavi valide solo in periodi coincidenti alla durata di un incarico. L'azienda può così ritenersi certa della provenienza di ogni versione del firmware, evitando che versioni non certificate vengano installate nelle proprie vetture.

Integrità

L'integrità è la protezione dei dati e delle informazioni nei confronti di modifiche (volontarie o non) del contenuto. Ciò risulta essenziale al fine del corretto funzionamento del sistema ed evita l'installazione di firmware modificati da terzi. Garantire l'integrità è fondamentale per gli OTA poiché vi è la possibilità che gli aggiornamenti vengano effettuati sotto reti instabili, le quali potrebbero corrompere il firmware in fase di scaricamento. Come detto in precedenza,

in molti casi viene gestita assieme all'autenticazione per mezzo di certificati rilasciati tramite PKI.

Riservatezza

I dati scambiati tra server e dispositivo target dell'aggiornamento devono essere protetti da letture non autorizzate, risultare cioè accessibili solo agli utenti e ai processi che ne hanno il diritto. Questo protegge la proprietà intellettuale dell'azienda nei confronti del software prodotto. Possibili malintenzionati non avendo la possibilità di leggere in chiaro l'aggiornamento, avranno più difficoltà nel trovare delle falle attraverso le quali violare la sicurezza del sistema. Negli OTA la sicurezza è garantita dall'uso della crittografia negli aggiornamenti e da protocolli sicuri come HTTPS.

Motivazioni economiche

Sebbene gli aggiornamenti OTA vadano ad eliminare tutti i passaggi richiesti dagli aggiornamenti manuali, fornendo notevoli benefici in termini di costi e di tempo, d'altro canto essi introducono altre spese non trascurabili. Nel settore automotive gli aggiornamenti necessitano di connessioni stabili per non minare l'integrità del software. Attualmente vi sono due possibilità per raggiungere la vettura da aggiornare: la rete wireless del cliente stesso o una rete dati cellulare. La prima opzione è esente da costi ma prevede che il proprietario tenga la propria auto in una zona coperta dalla propria rete Wi-Fi. Non tutti però hanno questa possibilità e a differenza delle macchine di lusso, tenute principalmente in box auto, le utilitarie, che costituiscono la maggior parte delle vetture in circolazione, sono tenute in parcheggi esterni. Aggiornare una vettura via Wi-Fi quindi non si adatta facilmente a tutti i mercati. Optare per l'uso dei dati cellulari consente di raggiungere le vetture in qualsiasi momento ma ha dei costi che vanno in base agli accordi presi con i gestori delle reti e alla dimensione degli aggiornamenti.

Gli aggiornamenti differenziali

Negli aggiornamenti manuali la dimensione non è molto influente ed è quindi comune avere aggiornamenti comprendenti l'intero sistema, sia con le parti "nuove" che con le parti "vecchie". Negli aggiornamenti OTA, in particolar modo quando si sfrutta la rete dati cellulare, più le dimensioni scaricate sono piccole e minore è il costo per ciascuna campagna di aggiornamento. Al fine di ridurre la mole di dati scaricati, si ricorre alla strategia degli "aggiornamenti differenziali": il sistema di aggiornamento riconosce quali sono i cambiamenti da una versione del firmware ad un'altra e crea un aggiornamento contenente solo i dati modificati. Il target dovrà essere in grado di ricostruire la nuova versione del firmware partendo da quello attuale, integrandolo con le informazioni ricevute. Questa tecnica consente una notevole riduzione delle dimensioni dei singoli aggiornamenti, con efficienze diverse in base all'algoritmo e alla strategia utilizzata.

Soluzioni esistenti

In commercio esistono numerosi sistemi per gli aggiornamenti OTA, alcuni dei quali offrono anche ottime prestazioni. Scegliere di adottare un sistema closed source significa essere vincolati alle scelte di un’altra azienda. Ha senso quindi studiare una propria soluzione appoggiandosi su tecnologie open source, apportando miglioramenti per ottenere il prodotto desiderato.

Le tecnologie utilizzate

Tutte le tecnologie utilizzate si basano sulla filosofia open source[4]. Si ha così la possibilità di sviluppare un sistema in tempi ridotti usando software già pronti. Se lo studio andrà a buon fine si analizzeranno le criticità, aggiungendo modifiche ed integrazioni che porteranno al prodotto finale.

Yocto

Yocto è un progetto open source che fornisce un software di cross-compilazione[5] per la creazione di sistemi basati su Linux. Attraverso Yocto gli sviluppatori di sistemi embedded di tutto il mondo sono in grado di condividere tecnologie, stack software[6] e configurazioni che sono poi usate per la creazione di immagini Linux su misura per dispositivi IoT[7].

I concetti base

Yocto si basa su di un modello a strati per la creazione dei sistemi Linux. Tale modello è progettato per supportare sia la collaborazione che la personalizzazione.

Ricetta

Una ricetta è una forma di metadati che contiene una lista di impostazioni ed istruzioni per la creazione di un pacchetto. L'assemblamento di più pacchetti genera l'immagine binaria di tutta la distribuzione.

Layer

Un layer, ovvero uno strato, è un insieme di ricette. È un'entità logica che serve per isolare le parti di una stessa distribuzione. Un unico strato può essere riutilizzato per più progetti.

[4]**Open-source:** Software non protetto da copyright e liberamente modificabile dagli utenti.
[5]**Cross-compilazione:** tecnica mediante la quale si compila il codice sorgente in una macchina con architettura diversa da quella con cui sarà eseguito il codice compilato.
[6]**Stack software:** insieme di componenti software necessari per creare una piattaforma completa.
[7]**IoT:** "internet of things", estensione di Internet al mondo degli oggetti e delle cose.

Generazione dell’immagine

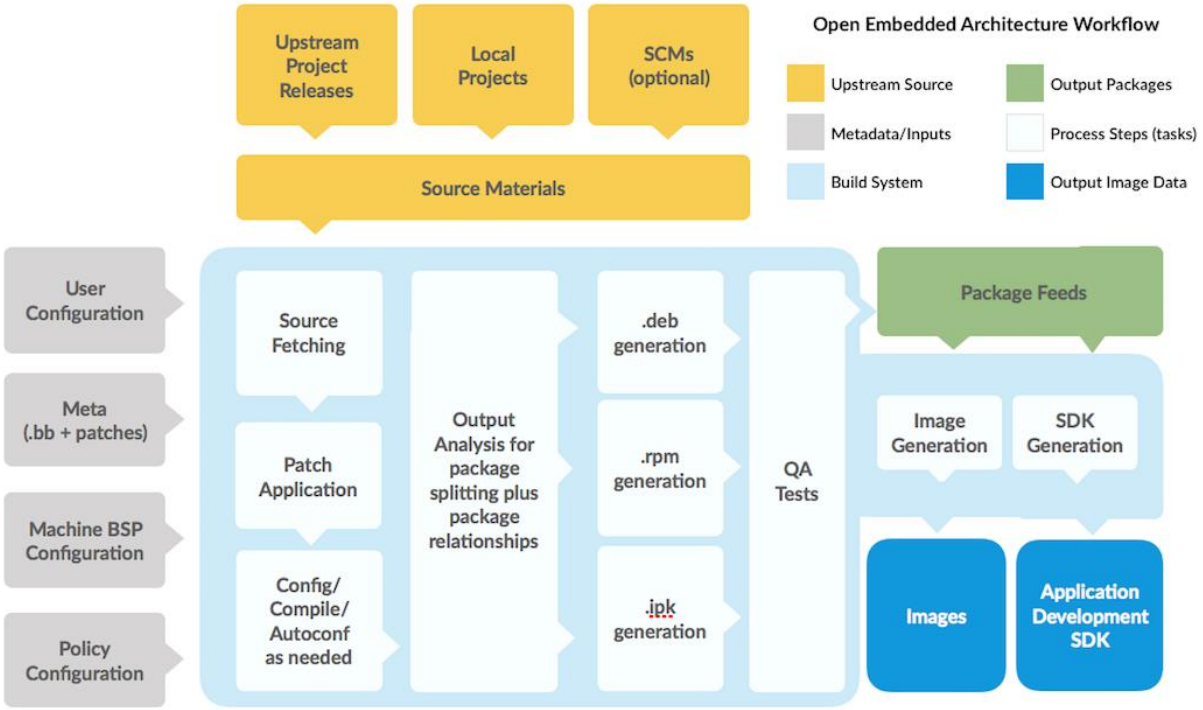


Figura 3

La Figura 3 illustra il processo di generazione delle immagini binarie. In primis vengono scaricati i layer già esistenti che contengono funzionalità utili alla distribuzione. Successivamente se necessario vengono creati i layer con all’interno le ricette per la personalizzazione del sistema. Infine viene eseguito il compilatore BitBake che partendo dai metadati forniti genera prima i relativi pacchetti e poi li assembla nell’immagine binaria. Tale immagine sarà poi caricata nel dispositivo embedded per il quale la distribuzione è stata sviluppata.

La struttura

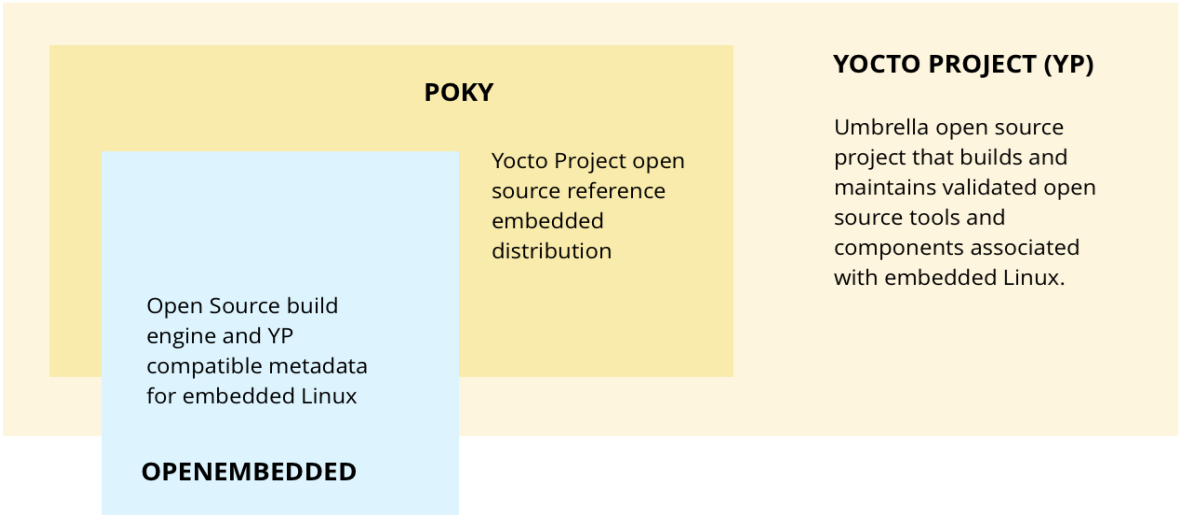


Figura 4

Yocto è un progetto detto ad “ombrello”. Esso comprende un vasto numero di progetti e risorse sviluppati da più aziende, le quali collaborano per creare infrastrutture per sistemi embedded Linux. Poky è uno di questi progetti, uno dei più grandi, ed è usato come esempio per mostrare come i vari tool lavorano assieme. Questo è costituito da un piccolo sistema operativo embedded, generato compilando l’insieme di metadati OpenEmbedded-Core attraverso BitBake (il sistema di build incluso). Il progetto OpenEmbedded fornisce oltre a quanto detto sopra, anche una delle ricette disponibili per la creazione di sistemi Linux, detta “meta-openembedded”. L’organizzazione generale di tutto l’ecosistema Yocto è riportata in Figura 4.

Rauc

RAUC controlla il processo di aggiornamento su un sistema embedded Linux. É sia una applicazione del dispositivo target dell'aggiornamento, eseguita come client, sia un tool che permette la creazione, l'ispezione e la modifica di artefatti per l'installazione. RAUC svolge quattro compiti fondamentali:

1. La generazione di artefatti software per gli aggiornamenti.
2. La firma e la verifica di tali artefatti.
3. La gestione robusta (affidabile) dell'installazione.
4. L'interazione con il processo di boot.

I Bundle

RAUC usa degli artefatti software per gli aggiornamenti chiamati bundle. Un bundle contiene dati per l'installazione sul sistema, come immagini di filesystem o archivi, e un "manifest", il quale indica le immagini da installare e contiene delle opzioni e meta-informazioni utili all'installazione stessa. Il tutto è impacchettato nel formato SqashFS che garantisce una buona compressione dei dati. RAUC rende obbligatoria la firma del bundle: una firma a chiave pubblica è applicata all'intera immagine del bundle ed è poi salvata nel formato CMS (Cryptographic Message Syntax). Prima dell'installazione la firma è verificata usando le chiavi già presenti nel sistema. Il processo di firma del bundle è gestito per mezzo di un'infrastruttura PKI. RAUC garantisce quindi l'autenticità e l'integrità del bundle ma non la sua riservatezza.

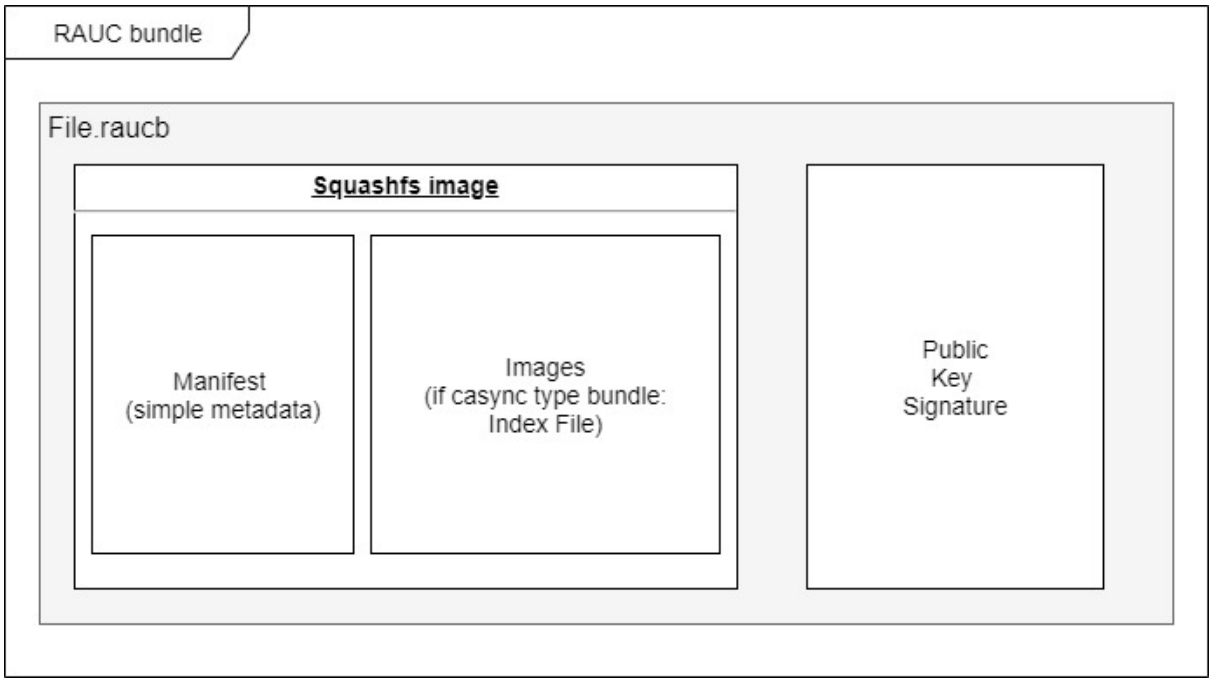


Figura 5

Gli Slot

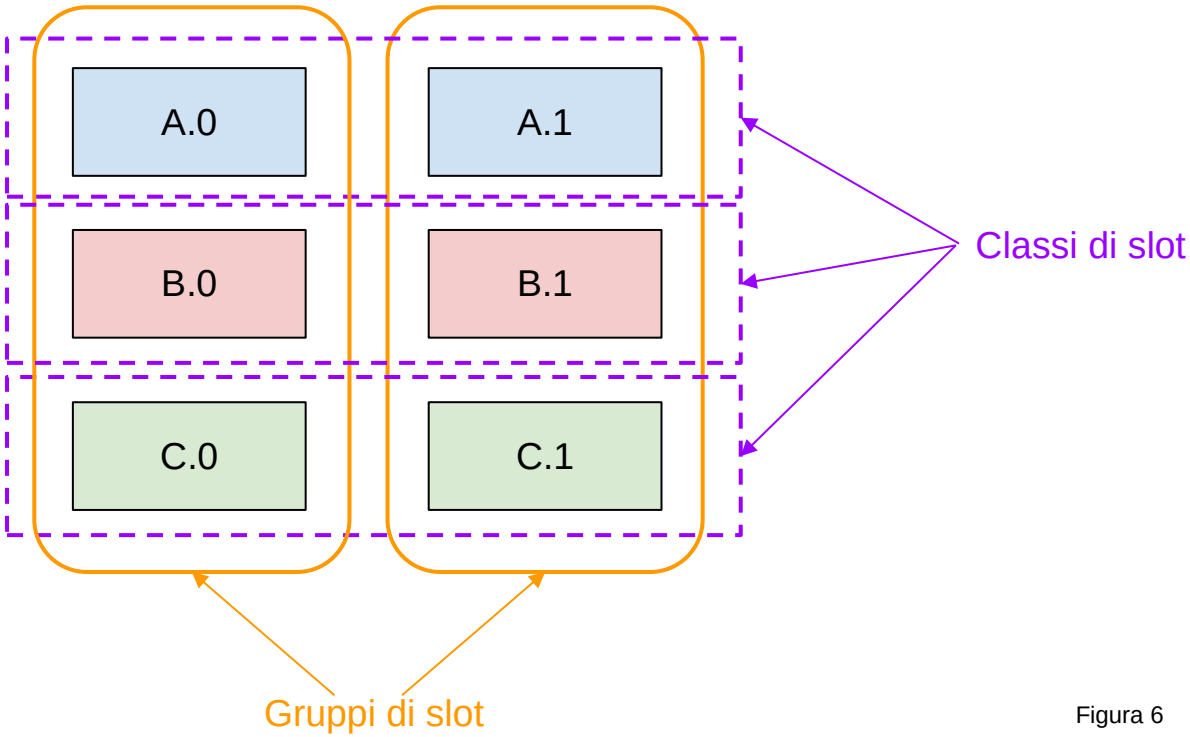


Figura 6

RAUC identifica come slot tutto ciò che può essere aggiornato. Uno slot può essere sia un intero dispositivo, una sua partizione o addirittura un file. Gli slot sono definiti dal file di configurazione del sistema presente nel dispositivo target. Più slot di classi diverse possono essere raccolti in gruppi di slot. La Figura 6 mostra due gruppi di slot, il primo relativo alla versione 0 del software ed il secondo alla versione 1. L'aggiornamento è installato nel gruppo di slot inattivi. Per identificare gli slot attivi, RAUC si avvale di informazioni reperite dal kernel o dalle informazioni di mount. RAUC crea un hash per ogni immagine o archivio nel momento in cui lo impacchetta nel bundle e lo salva nel manifest. Questo permette di identificare il contenuto dell'immagine. Al termine di un'installazione andata a buon fine, RAUC scrive un file contenente gli hash degli slot. La prossima volta che RAUC proverà ad installare un'immagine su quello slot, prima leggerà il file degli hash: se questi coincidono con quelli dell'immagine da installare, allora RAUC salterà l'aggiornamento di quel determinato slot. Questo ottimizza di molto le performance nel caso di aggiornamenti con numerosi slot ridondanti.

L’interazione con il processo di boot

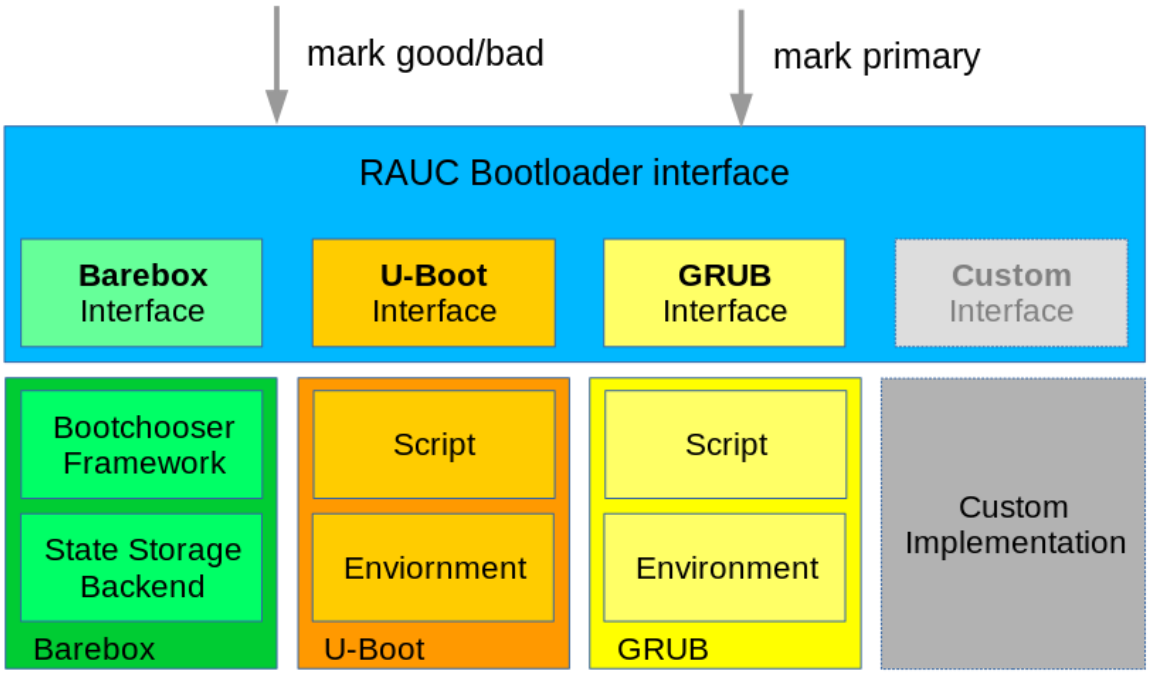


Figura 7

RAUC offre un'interfaccia per interagire con diversi tipi di bootloader esistenti e anche con possibili soluzioni di boot custom (Figura 7). Nel bootloader devono essere stabilite delle adeguate logiche di boot (ad esempio usando degli appositi script) e deve essere fornito un set di variabili modificabili dall’userspace[8] Linux, così da poter cambiare sia l’ordine sia la priorità di boot. Potendo interagire con il bootloader, RAUC può attuare la strategia di aggiornamento A/B o anche a più partizioni (indicate dagli slot).

[8]**Userspace:** Spazio per applicazioni eseguite in modalità utente esterno al kernel protetto attraverso la separazione dei privilegi.

Hawkbit

Hawkbit è un framework back-end per il rollout di aggiornamenti software su dispositivi IoT. Esso supporta complesse strategie di rollout, necessarie per progetti su larga scala e facilita l'organizzazione degli aggiornamenti mediante una classificazione dei vari artefatti software.

Le interfacce

È possibile l'interazione diretta tra dispositivo target e server Hawkbit per mezzo delle DDI API, delle API di architettura REST. Queste sono basate sugli standard HTTP e su meccanismi di polling[9]: esse consentono al dispositivo di comunicare il proprio stato, di verificare la presenza di aggiornamenti, scaricarli e comunicare l'esito dell'aggiornamento. Per quanto riguarda l'aspetto gestionale degli aggiornamenti è possibile usufruire dell'interfaccia grafica per interagire con il sistema, oppure possono essere usate le RESTful management API per interfacciarsi mediante adeguate soluzioni custom (Figura 8).

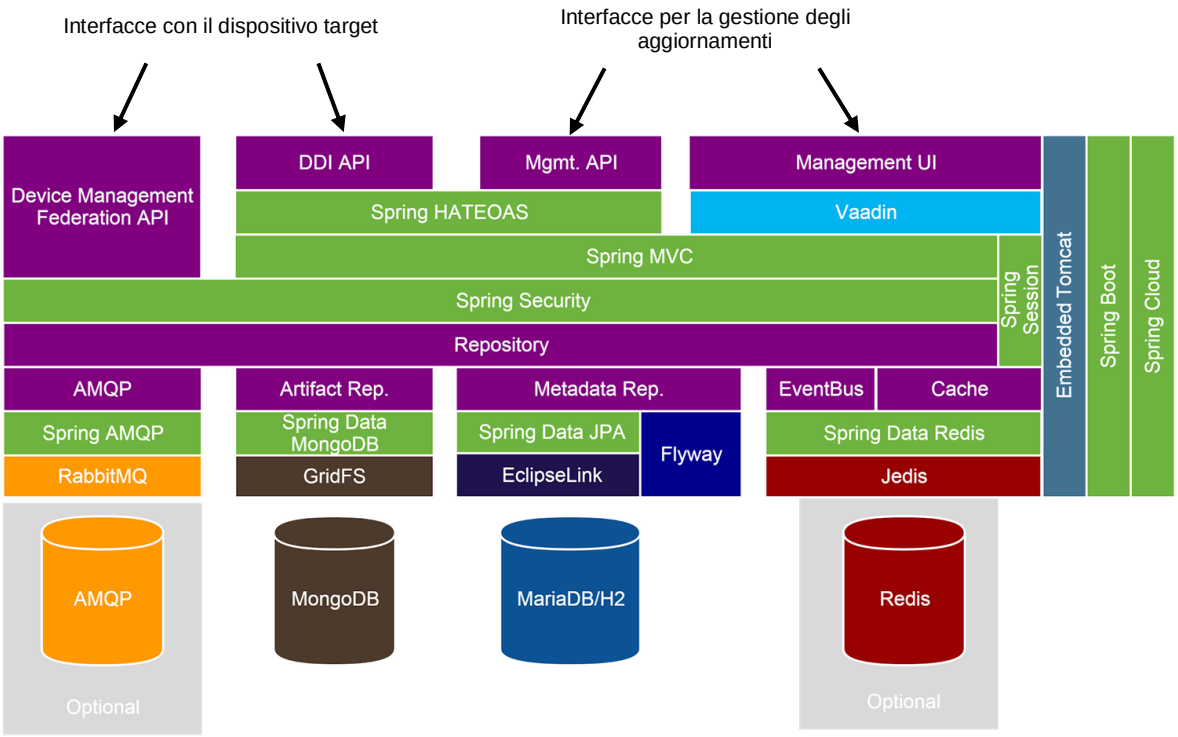


Figura 8

[9] **Polling:** Interrogazione ciclica.

I rollout

Hawkbit sopporta un'organizzazione facile e veloce di campagne di aggiornamento che consente di aggiornare un gran numero di dispositivi suddividendoli in gruppi. I gruppi vengono definiti da dei filtri che possono essere più o meno articolati a seconda delle esigenze. Tra le più importanti funzionalità vi sono:

- Aggiornamenti in cascata, ovvero l'inizio dell'aggiornamento di un gruppo in base allo stato di installazione del gruppo precedente.
- Interruzione di emergenza della campagna qualora in un gruppo si sia verificato un numero di errori superiore ad una determinata soglia.
- Monitoraggio del progresso dell'intera campagna e dei singoli gruppi.
- Monitoraggio dello stato dei singoli target in base all'ultima richiesta di poll.

Il modello del package

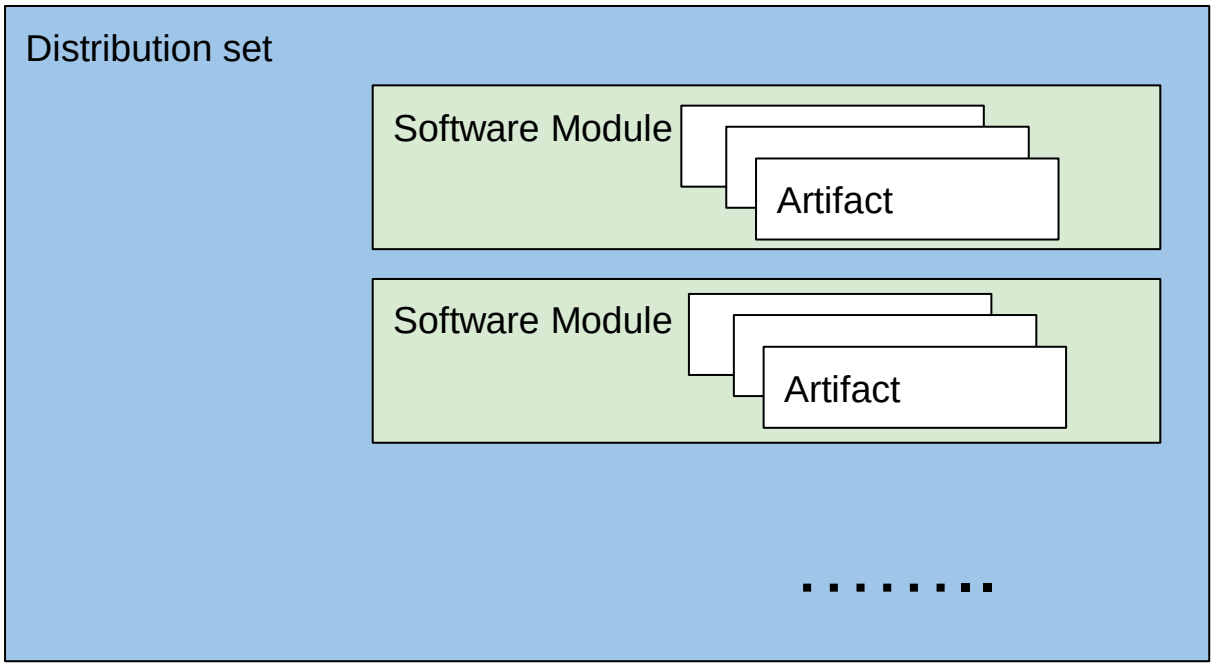


Figura 9

Hawkbit offre la possibilità di organizzare gli aggiornamenti definendo complesse strutture software (Figura 9). Quando si lancia una campagna di aggiornamento si assegna ad un gruppo, o ad un singolo target, una distribuzione. Questa è definita da un nome e una versione. Ogni distribuzione contiene uno o più moduli software, definiti a loro volta da nome e versione. Infine ogni modulo contiene uno o più artefatti. Un artefatto è una componente software non divisibile, come ad esempio un'immagine binaria, un archivio, etc.

Sicurezza

La comunicazione è resa sicura grazie alla possibilità di usare il protocollo HTTPs invece che l'HTTP. Inoltre ogni singolo target possiede un token di sicurezza che deve corrispondere a quello presente nel dispositivo e che ne garantisce l'autenticazione. La generazione dei token è gestita dal server Hawkbit. La firma e la codifica del contenuto degli aggiornamenti non è supportata e deve essere gestita da componenti esterni.

Casync

Casync è un software che offre un sistema per il salvataggio e la consegna di immagini di file system in maniera efficiente, ottimizzato per aggiornamenti ad alta frequenza via Internet. Per fare ciò prende ispirazione dall'algoritmo di rsync[10] e dal meccanismo di indirizzamento del file system usato da git[11].

Funzionamento

La Figura 10 mostra uno schema semplificato del funzionamento di Casync.

Codifica

Inizialmente vengono scelti o il device file[12] o la directory che si vogliono utilizzare. Questi vengono tramutati in uno stream binario, il quale viene suddiviso in chunk[13] grazie ad un algoritmo Buzhash, ossia una funzione hash ricorsiva che processa l'input tramite una finestra scorrevole sull'input stesso. L'algoritmo SHA256 viene utilizzato per generare i digest dei singoli chunk e l'algoritmo XZ per comprimere i chunk individualmente. I chunk compressi vengono quindi salvati in una directory (che in figura è chiamata "chunk store") e denominati usando i relativi digest. Viene generato un file (chiamato "chunk index" in figura) che contiene una lista di tutti i digest dei chunk più la relativa dimensione, il tutto sotto forma di un array lineare.

Decodifica

Partendo dal "chunk index" si ricostruisce lo stream binario concatenando i vari chunk decompressi presi dal "chunk store" grazie alle corrispondenze dei vari valori hash assegnati in precedenza.

[10] **Rsync:** Software per Unix che sincronizza file e cartelle da una posizione all'altra minimizzando il trasferimento di dati utilizzando quando possibile la codifica delta.
[11] **Git:** Software di controllo versione distribuito.
[12] **Device file:** Tipo speciale di file che rappresenta una periferica o un dispositivo virtuale.
[13] **Chunk:** Letteralmente "pezzetto".

I vantaggi

Tutto il procedimento elencato sopra apporta alcuni vantaggi:

- Un singolo “chunk store” può contenere più versioni di uno stesso software evitando le ridondanze per le parti comuni. Questo comporta una riduzione nello spazio occupato nei server adibiti alla distribuzione degli aggiornamenti.
- Un dispositivo IoT può calcolare il proprio “chunk index”, compararlo con quello della versione software successiva e scaricare solo i chunk che gli occorrono. Così facendo si riduce notevolmente la quantità di dati da scaricare.
- Poiché la funzione hash usata assegna valori univoci ai singoli chunk in base al loro contenuto, vi è un’unica rappresentazione valida per i dati consegnati. L’integrità dell’aggiornamento è quindi garantita.

Casync inoltre supporta nativamente i protocolli HTTP, HTTPS, FTP e SSH per il download del chunk index e dei chunk.

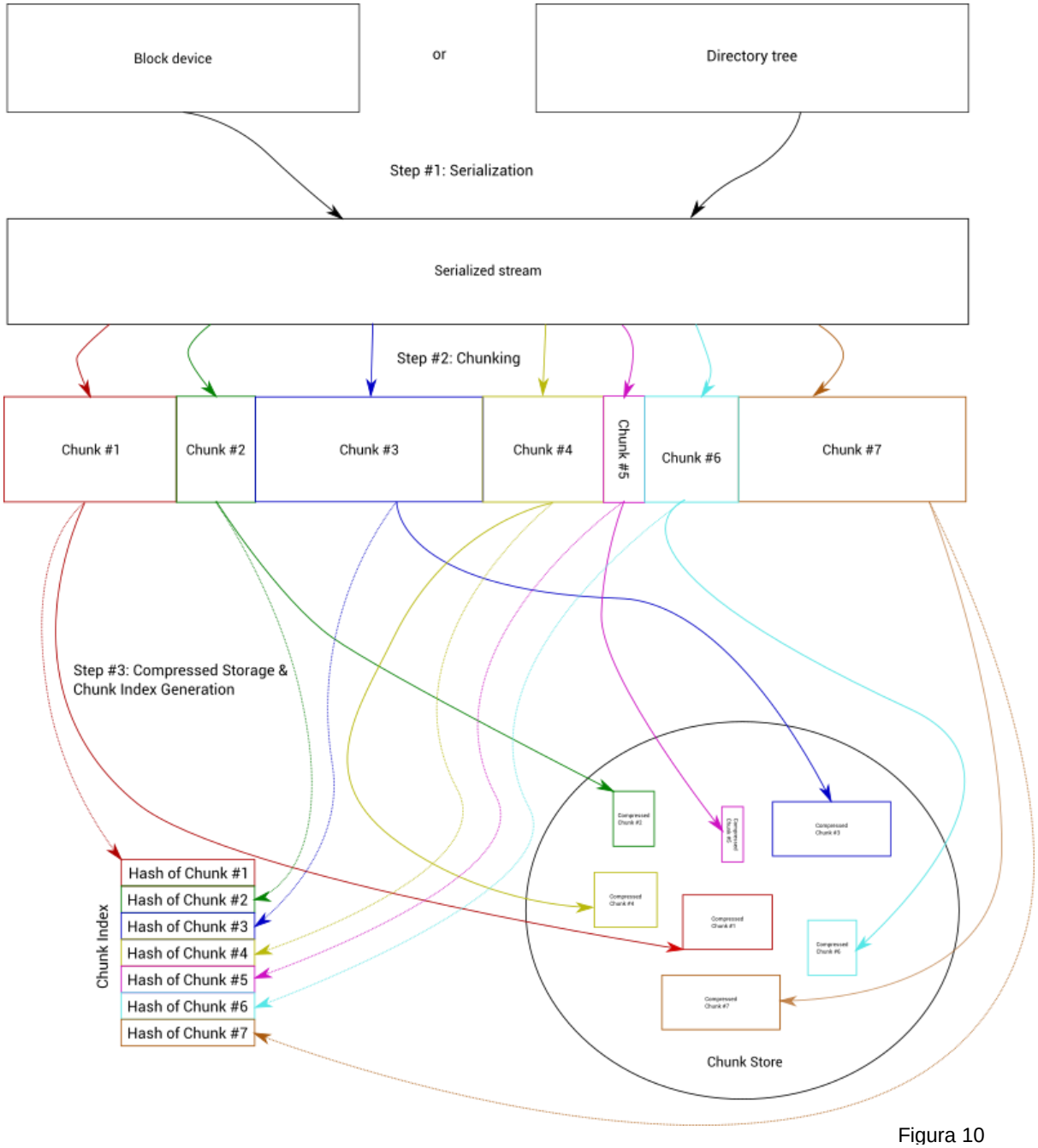

Block device
or
Directory tree
Step #1: Serialization
Serialized stream
Step #2: Chunking
Chunk #1
Chunk #2
Chunk #3
Chunk #4
Chunk #5
Chunk #6
Chunk #7
Step #3: Compressed Storage &
Chunk Index Generation
Chunk Index
Hash of Chunk #1
Hash of Chunk #2
Hash of Chunk #3
Hash of Chunk #4
Hash of Chunk #5
Hash of Chunk #6
Hash of Chunk #7
Chunk Store

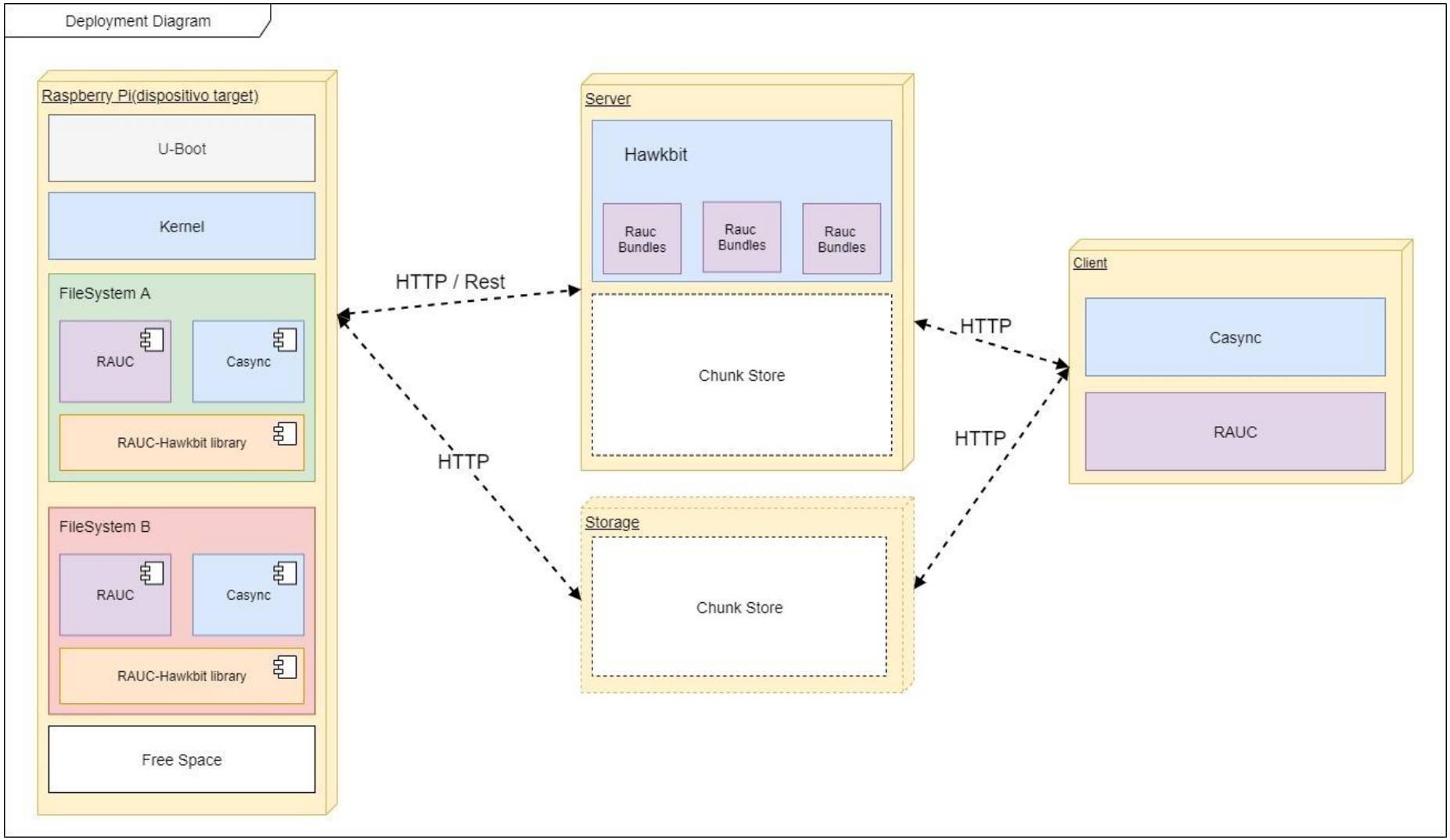
Figura 10

Implementazione e Sviluppo

Il sistema per gli aggiornamenti OTA si estende su quattro piattaforme separate:

- Il dispositivo target che riceve gli aggiornamenti.
- Il client che genera gli aggiornamenti.
- Il server che si occupa della distribuzione degli aggiornamenti.
- Il server nel quale sono salvati i chunk.

Gli ultimi due possono essere eseguiti anche sulla stessa macchina in base alle necessità. Il deployment diagram presente nella pagina successiva mostra le varie interazioni tra le piattaforme sopra elencate e dove risiedono le componenti software più importanti ai fini dell'aggiornamento.


Deployment Diagram
Raspberry Pi(dispositivo target)
U-Boot
Kernel
FileSystem A
RAUC
Casync
RAUC-Hawkbit library
FileSystem B
RAUC
Casync
RAUC-Hawkbit library
Free Space
HTTP / Rest
HTTP
Server
Hawkbit
Rauc Bundles
Rauc Bundles
Rauc Bundles
Chunk Store
Storage
Chunk Store
HTTP
HTTP
Client
Casync
RAUC

Software lato client

Come client è stato usato un computer con a bordo un sistema operativo Ubuntu contenente i software Casync e RAUC (Figura 11).

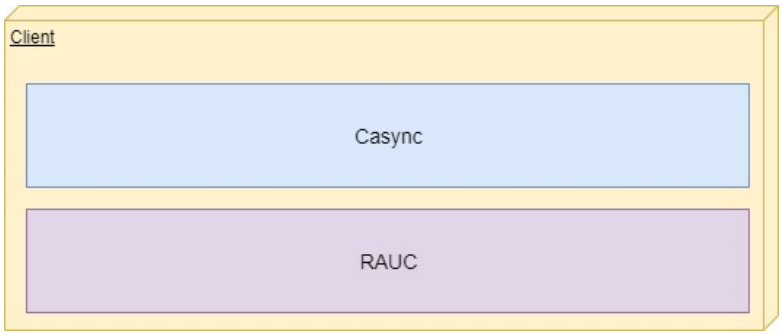


Figura 11

Per configurare il client è sufficiente scaricarli dalle relative repository Git (https://github.com/systemd/casync.git, https://github.com/rauc/rauc.git) e installare le librerie necessarie per il loro funzionamento.

Generazione degli aggiornamenti

La Figura 12 mostra i passaggi necessari per convertire un intero Filesystem o un qualsiasi altro tipo di file in artefatti software compatibili con il sistema di aggiornamento OTA. Mediante il comando da terminale

```
rauc --cert=<certfile> --key=<keyfile> bundle directory_filesystem/
nome_bundle.raucb
```

RAUC genera un bundle a partire dalla “directory_filesystem”, all’interno della quale vi sono un manifest, che contiene informazioni sull’immagine da selezionare e a quale classe di slot appartiene, e l’immagine del filesystem. Il bundle viene firmato utilizzando delle chiavi ottenute tramite un’infrastruttura PKI o, nel nostro caso, utilizzando le chiavi temporanee fornite da RAUC stesso. In seguito attraverso il comando

```
rauc convert --cert=<certfile> --key=<keyfile> --keyring=<keyring> nome-
bundle.raucb casync-bundle.raucb
```

RAUC partendo dal bundle originato in precedenza e sfruttando casync, produce un bundle chiamato “casync-bundle.raucb” contenente lo stesso manifest ma un index file al posto dell’immagine binaria e una directory chiamata “casync-bundle.castr” contenente i chunk. Se il chunk store è già presente e contiene chunk appartenenti ad altre versioni dello stesso software, casync vi aggiunge solo i chunk non ridondanti che consentiranno di ricostruire la versione del software processata (Figura 13). Il bundle e il chunk store vengono caricati sul server adibito alla distribuzione degli aggiornamenti.

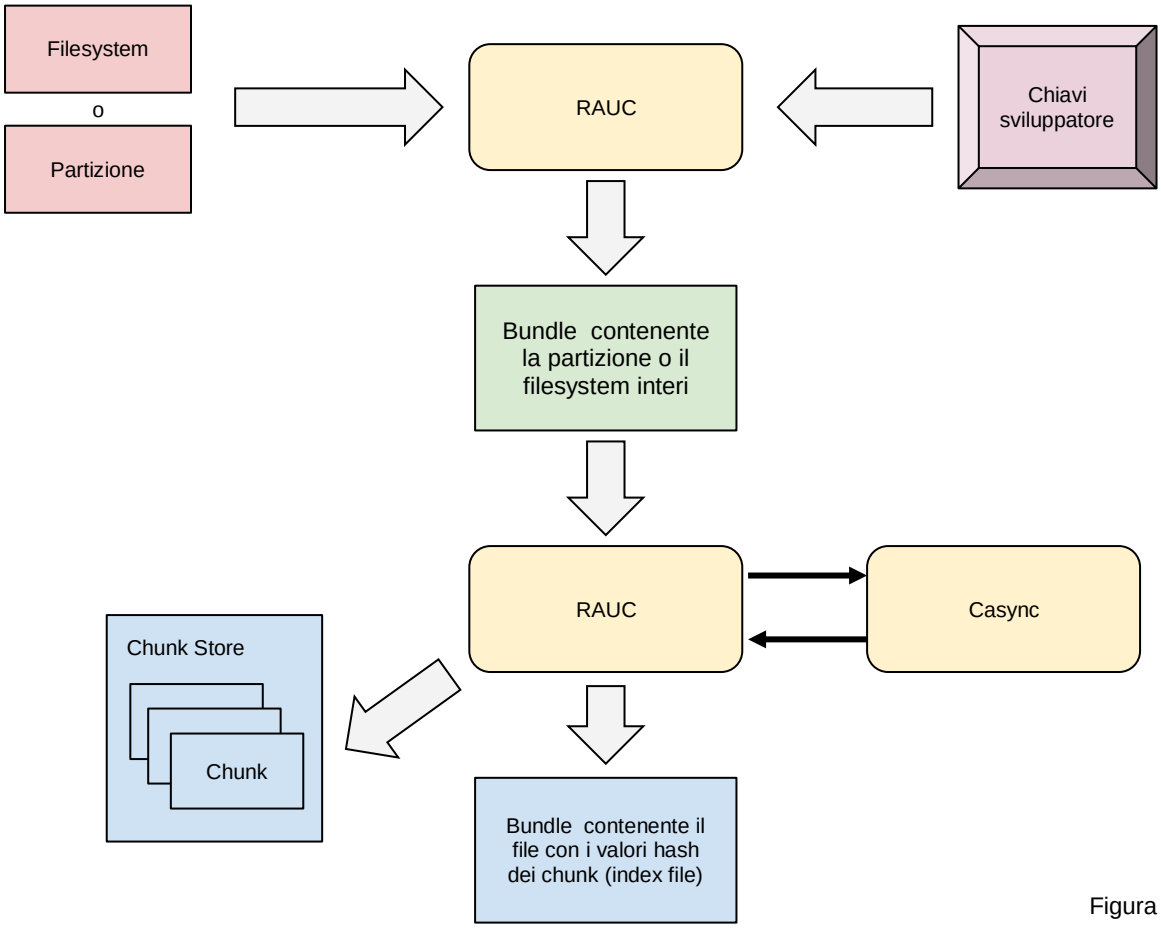


Figura 12

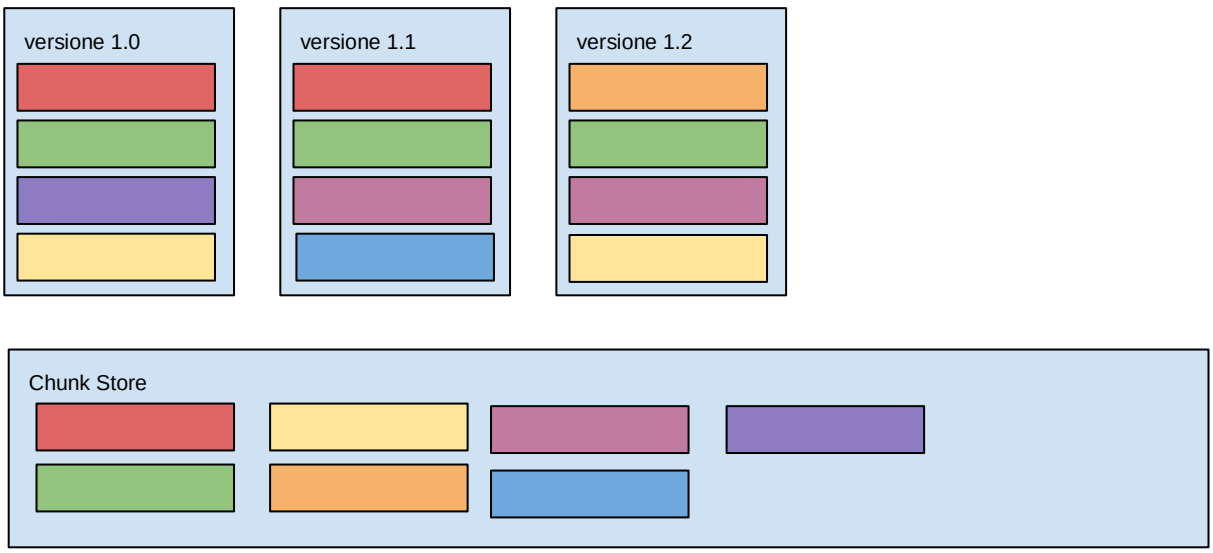


Figura 13

Interazione RAUC-casync

In seguito all'implementazione delle tecnologie sopra riportate, è stata portata avanti una fase di reverse engineering con il fine di comprendere i meccanismi di interazione tra RAUC e casync durante la conversione del bundle. Questa fase ha portato alla realizzazione di un diagramma UML di tipo structural (Figura 14) che evidenzia le principali componenti coinvolte nel processo di conversione del bundle.

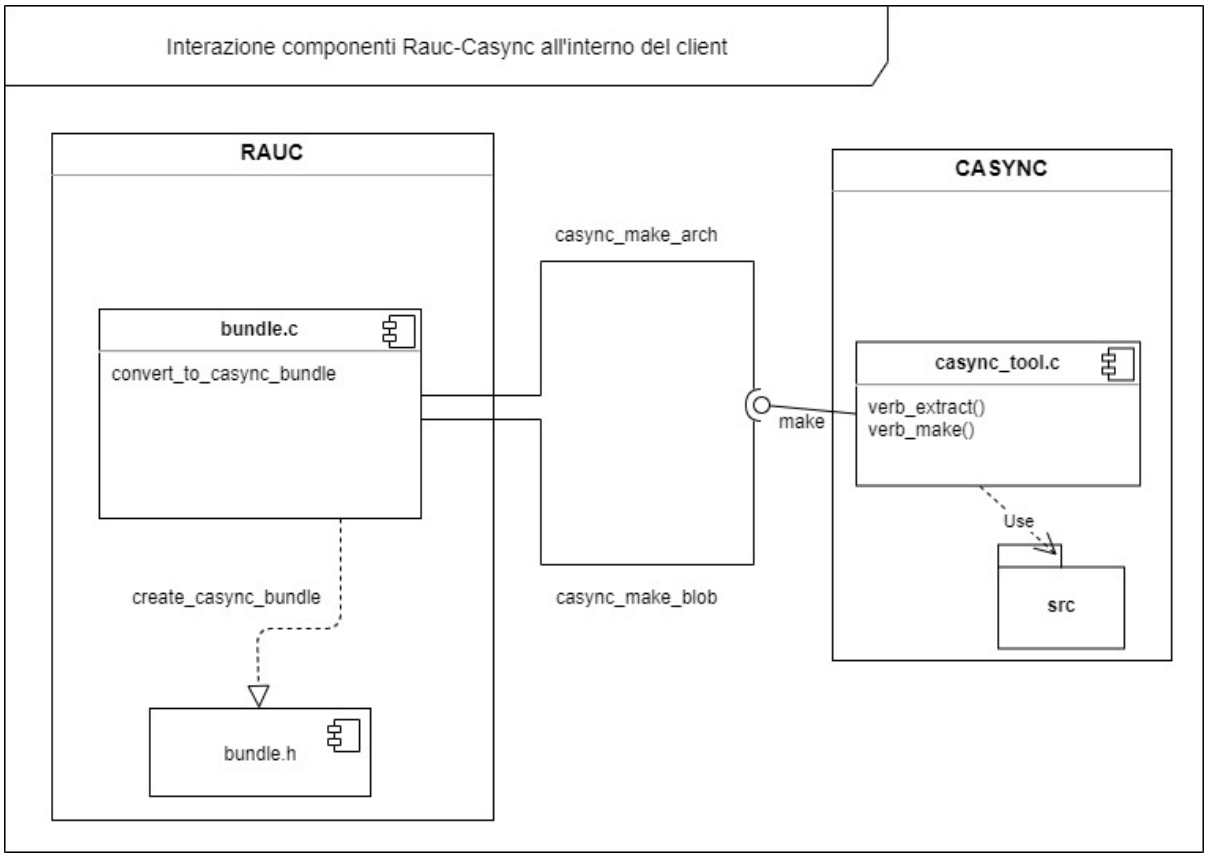


Figura 14

Di seguito è riportata la funzione convert_to_casync_bundle grazie alla quale è possibile convertire un bundle RAUC standard in uno che sfrutta la tecnologia di casync. Per brevità sono state selezionate solo le parti più significative che spiegano il processo di conversione del bundle.

```
static gboolean convert_to_casync_bundle(RaucBundle *bundle, const gchar
*outbundle, GError **error)
{
     ....

    /* Assure bundle destination path does not already exist */
    if (g_file_test(outbundle, G_FILE_TEST_EXISTS)) {
```

```
        g_set_error(error, G_FILE_ERROR, G_FILE_ERROR_EXIST, "Destination
bundle '%s' already exists", outbundle);
        res = FALSE;
        goto out;
    }

    if (g_file_test(storepath, G_FILE_TEST_EXISTS)) {
        g_warning("Store path '%s' already exists, appending new chunks",
outbundle);
    }
```

Inizialmente RAUC si assicura che non vi siano bundle con lo stesso nome di quello scelto per il bundle convertito. In seguito cerca la presenza di un chunk store sulla posizione definita dal comando di conversione: se lo trova aggiungerà ad esso soltanto i chunk aggiuntivi.

```
    /* Set up tmp dir for conversion */
    tmpdir = g_dir_make_tmp("rauc-casync-XXXXXX", &ierror);
    if (tmpdir == NULL) {
        g_propagate_prefixed_error(error, ierror,
                "Failed to create tmp dir: ");
        res = FALSE;
        goto out;
    }

    contentdir = g_build_filename(tmpdir, "content", NULL);
    mfpath = g_build_filename(contentdir, "manifest.raucm", NULL);

    /* Extract input bundle to content/ dir */
    res = extract_bundle(bundle, contentdir, &ierror);
    if (!res) {
        g_propagate_error(error, ierror);
        goto out;
    }
```

Successivamente crea una directory temporanea per la conversione e vi estrae il contenuto del bundle originale (formato da una o più immagini e da un manifest).

```
    /* Load manifest from content/ dir */
    res = load_manifest_file(mfpath, &manifest, &ierror);
    if (!res) {
        g_propagate_error(error, ierror);
```

```
        goto out;
    }

    /* Iterate over each image and convert */
    for (GList *l = manifest->images; l != NULL; l = l->next) {
        RaucImage *image = l->data;
        g_autofree gchar *imgpath = NULL;
        g_autofree gchar *idxfile = NULL;
        g_autofree gchar *idxpath = NULL;

        imgpath = g_build_filename(contentdir, image->filename, NULL);

        if (image_is_archive(image)) {
            idxfile = g_strconcat(image->filename, ".caidx", NULL);
            idxpath = g_build_filename(contentdir, idxfile, NULL);

            g_message("Converting %s to directory tree idx %s", image-
>filename, idxfile);

            res = casync_make_arch(idxpath, imgpath, storepath, &ierror);
            if (!res) {
                g_propagate_error(error, ierror);
                goto out;
            }
        } else {

            idxfile = g_strconcat(image->filename, ".caibx", NULL);
            idxpath = g_build_filename(contentdir, idxfile, NULL);

            g_message("Converting %s to blob idx %s", image->filename,
idxfile);
            /* Generate index for content */
            res = casync_make_blob(idxpath, imgpath, storepath, &ierror);
            if (!res) {
                g_propagate_error(error, ierror);
                goto out;
            }
        }
```

RAUC legge il contenuto del manifest copiato nella directory provvisoria. Ogni immagine indicata viene riconvertita in archivio o in qualsiasi altro file che rappresenta e poi processata da casync attraverso la funzione “make” che genera i chunk e il relativo index file.

```
        /* Rewrite manifest filename */
        g_free(image->filename);
        image->filename = g_steal_pointer(&idxfile);

        /* Remove original file */
        if (g_remove(imgpath) != 0) {
            g_warning("Failed removing %s", imgpath);
        }
    }

    /* Rewrite manifest to content/ dir */
    res = save_manifest_file(mfpath, manifest, &ierror);
    if (!res) {
        g_propagate_error(error, ierror);
        goto out;
    }

    res = create_bundle(outbundle, contentdir, &ierror);
    if (!res) {
        g_propagate_error(error, ierror);
        goto out;
    }

    res = TRUE;
out:
    /* Remove temporary bundle creation directory */
    if (tmpdir)
        rm_tree(tmpdir, NULL);
    return res;
}
```

Infine viene riscritto il manifest in base agli index file e ai chunk generati, creato il nuovo bundle e rimossa la directory temporanea.

Software lato server

Come server è stata usata una macchina virtuale con sistema operativo Ubuntu. I software presenti al suo interno sono:

- Hawkbit, utilizzato per la gestione delle campagne di aggiornamento, che consistono nella distribuzione dei singoli bundle ai vari dispositivi target.
- Docker, utilizzato per eseguire i container necessari al funzionamento di Hawkbit.
- Python, utilizzato per la creazione di un server HTTP. Questo permette a casync di scaricare i chunk dal chunk store al dispositivo target dell'aggiornamento.

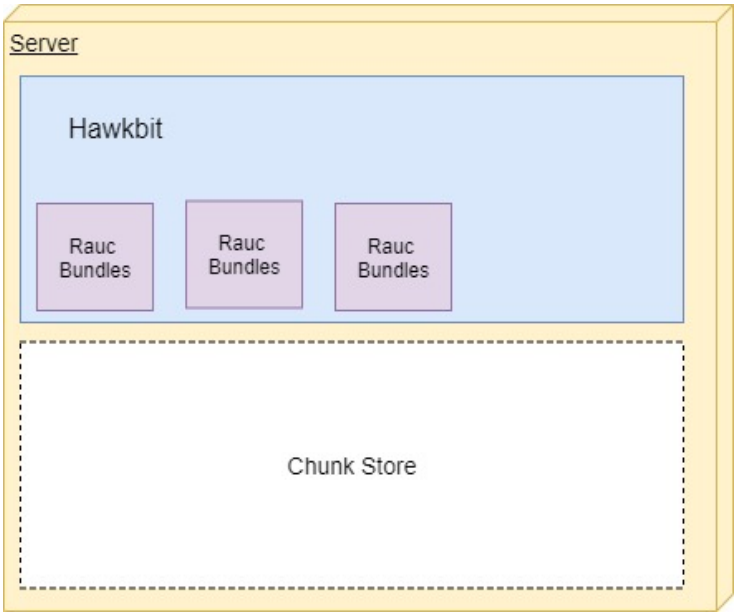


Figura 15

La figura 15 mostra la posizione dei bundle e dei chunk nel server.

Server Hawkbit

La repository Git https://github.com/eclipse/hawkbit.git offre sia la possibilità di installare manualmente le varie componenti software che compongono tutto il sistema Hawkbit, sia la possibilità di utilizzare la tecnologia di containerizzazione Docker. Quest'ultima è stata scelta per la sua caratteristica "plug and play", perché evita tutti i problemi di compatibilità tra l'ambiente di esecuzione ed il software. Inoltre è ampiamente utilizzata nei sistemi su vasta scala perché offre numerose funzionalità come ad esempio il clustering[14].

[14]**Clustering:** Tecnica per la scalabilità di un sistema.

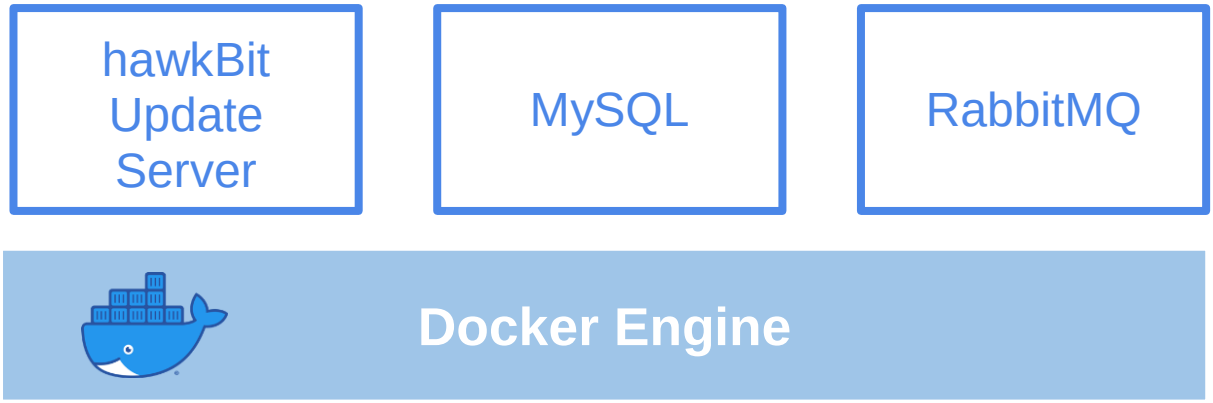


Figura 16

La Figura 16 mostra i servizi in esecuzione come container nel docker engine.

Server Python HTTP

Per la gestione dei chunk è stato usato un semplice server HTTP realizzato con Python. In particolare è stato usato il modulo `http.server`. Questo può essere invocato direttamente utilizzando il comando a terminale

```
python -m http.server 8000
```

servendo i file relativi alla directory corrente. Inoltre se non specificato dal comando `--bind` esso si lega a tutte le interfacce di rete presenti nella macchina. Questa è una soluzione provvisoria e in fase di produzione sarà opportuno sostituirla con soluzioni più affidabili e sicure.

Gestione degli aggiornamenti

Per gestire il caricamento degli aggiornamenti e la loro assegnazione ai vari dispositivi è stata utilizzata l'interfaccia grafica già presente. La maggior parte dei test è stata eseguita assegnando manualmente l'aggiornamento al dispositivo target, poiché l'utilizzo di rollout prevede un grande numero di dispositivi, mentre quelli a disposizione si aggiravano tra i 2 e i 4.

Software per dispositivo target

Come già accennato nell'introduzione, per semplicità si è optato per usare una Raspberry Pi 3 Model B+ come piattaforma rappresentante un dispositivo target di aggiornamento. Questa contiene una distribuzione Linux generata mediante YOCTO. La Figura 17 mostra il partizionamento della memoria secondaria della Raspberry e dove risiedono le componenti software responsabili degli aggiornamenti. Oltre alla partizione contenente il bootloader U-Boot e quella contenente il kernel, vi sono le partizioni ridondanti A e B che contengono i filesystem, e solo una delle due viene caricata in fase di boot.

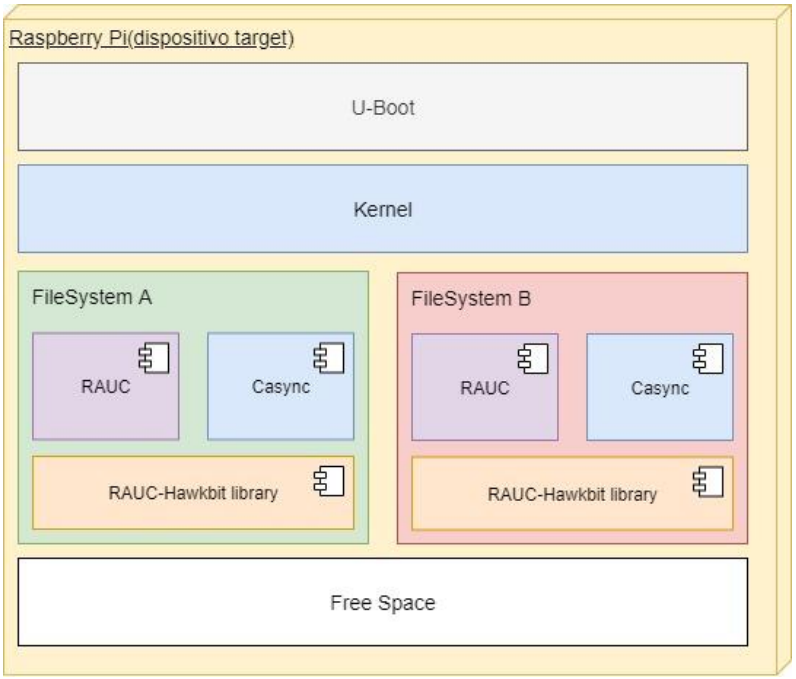


Figura 17

Generazione della distribuzione per la Raspberry

Utilizzando YOCTO è stata sviluppata una ricetta ad hoc grazie alla quale è stato possibile generare un'immagine binaria contenente tutto il software necessario. Mediante il file `bblayers.conf` sono stati selezionati i layer da installare presenti come directory nella cartella di poky. I layer meta-rasperrypi e meta-rauc sono stati scaricati dalle repository Git ufficiali (https://github.com/agherzan/meta-raspberrypi.git, https://github.com/rauc/meta-rauc.git), mentre il layer meta-gabriele è un layer custom creato appositamente per il progetto di tesi. Di seguito è riportato il file di configurazione nel quale sono selezionati tutti i layer utilizzati.

```
# POKY_BBLAYERS_CONF_VERSION is increased each time poky/build/conf/bblayers.c
onf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  /home/gabrielefantini/Desktop/poky/meta \
  /home/gabrielefantini/Desktop/poky/meta-poky \
  /home/gabrielefantini/Desktop/poky/meta-yocto-bsp \
  /home/gabrielefantini/Desktop/poky/meta-gabriele \
  /home/gabrielefantini/Desktop/poky/meta-raspberrypi \
  /home/gabrielefantini/Desktop/poky/meta-rauc \
  "
```

La Figura 18 mostra il contenuto del layer meta-gabriele.

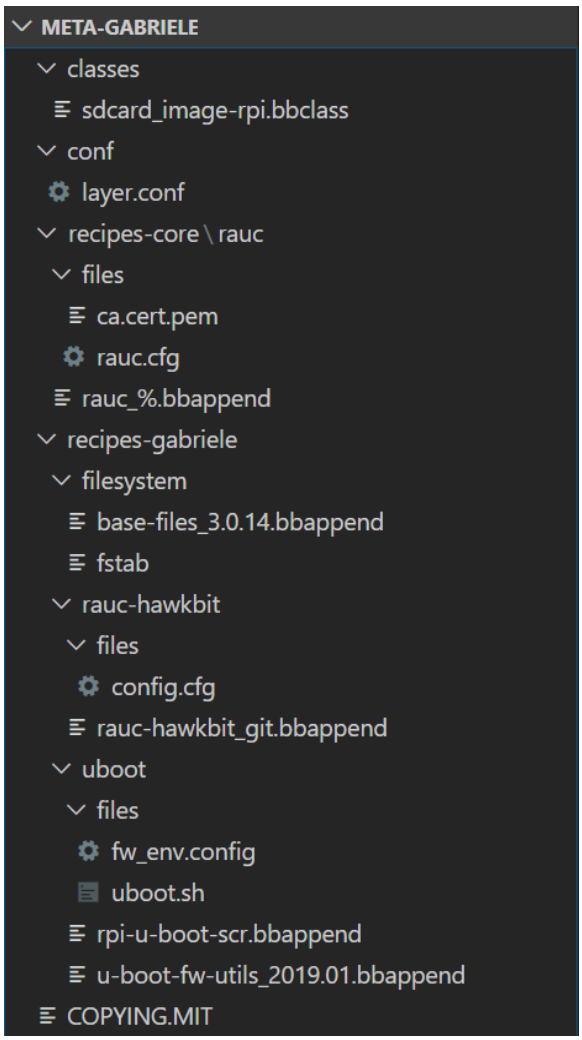


Figura 18

Attraverso il file `local.conf` presente nella directory `poky/build/conf/` è stata selezionata il tipo di macchina:

```
#defines the version of raspberry pi
MACHINE ?= "raspberrypi3"
```

e sono stati installati i software disponibili mediante i layer sopra citati, necessari al funzionamento del sistema di aggiornamento:

```
IMAGE_INSTALL_append = " rauc"
IMAGE_INSTALL_append += " u-boot-fw-utils"
IMAGE_INSTALL_append += " rauc-hawkbit"
IMAGE_INSTALL_append += " casync "
```

Infine è stato selezionato U-Boot come bootloader:

```
RPI_USE_U_BOOT = "1"
```

Layout del disco

La strategia di aggiornamento adottata è quella A/B. Si è scelto di avere due partizioni ridondanti da aggiornare solo per i filesystem, mentre partizioni uniche per kernel e bootloader. Questo per vari motivi:

- Il bootloader è una componente fondamentale per l'integrità del sistema, danneggiarla significherebbe perdere l'uso di tutto il dispositivo. Date queste premesse, una volta stabilito il suo corretto funzionamento raramente viene aggiornato e, qualora vi sia la necessità, lo si fa sempre in maniera manuale.
- Il kernel è molto leggero rispetto al filesystem ed è più complicato andarlo a modificare per testare l'efficacia di sistemi di aggiornamento differenziali. Si è scelto così di prendere in considerazione soltanto il filesystem, tenendo conto che eventuali strategie di aggiornamento sviluppate sarebbero valide anche per il kernel.

```
# The disk layout used is:
#
#    0                      -> IMAGE_ROOTFS_ALIGNMENT         - reserved for other data
#    IMAGE_ROOTFS_ALIGNMENT -> BOOT_SPACE                     - bootloader and kernel
#    BOOT_SPACE             -> SDIMG_SIZE                     - rootfs
#

#                                                Default Free space = 1.3x
#                                                Use IMAGE_OVERHEAD_FACTOR to add more space
#                                                <--------->
#           4MiB              40MiB          SDIMG_ROOTFS           SDIMG_ROOTFS
# <-----------------------> <----------> <---------------------><---------------------->
#  ------------------------ ------------ ----------------------- ----------------------
# | IMAGE_ROOTFS_ALIGNMENT | BOOT_SPACE | ROOTFSA_SIZE           | ROOTFSB_SIZE         |
#  ------------------------ ------------ ----------------------- ----------------------
# ^                       ^            ^                       ^                      ^
# |                       |            |                       |                      |
# 0                      4MiB      4MiB + 40MiB       4MiB + 40Mib + SDIMG_ROOTFS     4MiB + 40Mib + SDIMG_ROOTFS + SDIMG_ROOTFS
```

Figura 19

La Figura 19 è tratta dal file `sdcard_image-rpi.bbclass`, situato nella directory "classes" presente nel layer meta-gabriele, il quale si occupa della suddivisione della ROM della

Raspberry. In esso vengono definite le dimensioni delle varie partizioni, il loro allineamento e viene definita anche la tabella delle partizioni. Inoltre il file contiene dei comandi grazie ai quali il compilatore è in grado di costruire l'immagine binaria della build usando il partizionamento definito.

RAUC ed U-Boot

U-Boot è supportato nativamente da RAUC. Affinché i due software possano interagire però occorre:

- Creare uno script per U-Boot che gli consenta di gestire il boot in presenza di partizioni ridondanti.
- Installare i tool fw_printenv e fw_setenv nel filesystem così da poter leggere e scrivere le variabili di ambiente di U-Boot dall'userspace di Linux. La posizione delle variabili è indicata ai tools dal file di configurazione `fw_env.config`. Tale posizione deve coincidere con quella dove il file `fstab` ha montato l'intera partizione di boot.

`fw_env.config`,`fstab` e `uboot.sh` (lo script di avvio per U-Boot) sono tutti forniti all'interno del layer meta-gabriele. Lo script al momento dell'avvio del dispositivo cerca delle specifiche variabili di boot. Se non le trova le crea e gli assegna dei valori di default. Nel sistema sviluppato le partizioni dei due filesystem sono chiamate A e B: sono specificati, mediante le variabili di U-Boot, l'ordine di avvio, i tentativi di avvio rimasti e lo stato delle partizioni, che può essere "good" o "bad". Lo script quindi proverà ad avviare la prima partizione marcata "good". Se il tentativo fallisce passerà all'altra e diminuirà i tentativi di avvio dalla prima. Quando il valore dei tentativi di avvio arriva a 0 per entrambe le partizioni, lo script riporta tutti i valori delle variabili a quelli di default. Grazie ai tool nominati in precedenza, RAUC può sia risalire a quale partizione è attualmente attiva, sia marcare "bad" la partizione che andrà ad aggiornare e contrassegnarla "good" solo ad aggiornamento ultimato con esito positivo. Così anche nell'eventualità di una perdita di alimentazione, il bootloader saprà che un aggiornamento non è andato a buon fine e quindi continuerà ad avviare la vecchia partizione funzionante.

File di configurazione RAUC

Quanto segue è il contenuto del file `rauc.cfg` presente in `meta-gabriele/recipes-core/files/rauc.cfg`

```
[system]
compatible=rauc-raspberry
bootloader=uboot

[keyring]
path=ca.cert.pem


[slot.rootfs.0]
device=/dev/mmcblk0p2
type=ext3
bootname=A
```

```
[slot.rootfs.1]
device=/dev/mmcblk0p3
type=ext3
bootname=B

[casync]
storepath=10.0.200.153
```

In esso vengono definiti due slot di classe “rootfs”. Per ogni slot vengono definite le relative partizioni, il tipo di filesystem utilizzato e il nome di boot. Nel momento dell’installazione RAUC verifica le informazioni presenti nel manifest del bundle con quelle presenti nel file `rauc.cfg` per garantire che lo slot da aggiornare sia della stessa classe e utilizzi lo stesso filesystem di quello nel bundle. Infine viene selezionato lo “storepath” di casync, l’indirizzo dal quale scaricherà i chunk aggiuntivi.

Certificati RAUC

RAUC offre la possibilità di usare in fase di sviluppo un’alternativa alla firma dei bundle attraverso una “Public Key Infrastructure”: grazie ad uno script presente all’interno della repository ufficiale “rauc”, più specificatamente seguendo il percorso `rauc/test/openssl-ca.sh`, è possibile generare delle chiavi provvisorie. Una di queste chiavi va collocata sul dispositivo target, il quale la userà per identificare il bundle in fase di installazione. Per inserire la chiave direttamente nella build è sufficiente sostituirla al file `ca.cert.pem` presente in meta-gabriele.

La libreria rauc-hawkbit

Nella distribuzione è presente una libreria python che opera come interfaccia tra le API D-Bus di RAUC e le API DDI di Hawkbit. Oltre ad essere usata come libreria, essa può essere usata anche come una semplice applicazione client. Questa offre la funzione di polling degli aggiornamenti sul server Hawkbit e si occupa di scaricarli se presenti. Una volta ottenuto il bundle lo installa mediante RAUC, comandandolo tramite D-Bus. Per inserire il file di configurazione all’interno della build occorre sostituirlo al file situato su `meta-gabriele/recipes-gabriele/rauc-hawkbit/files/config.cfg`.

```
[client]
hawkbit_server =10.0.200.153:8080
ssl = false
tenant_id = DEFAULT
target_name = test-target
auth_token = bad3dfe83cad3eed6b467695f768c661
bundle_download_location = /tmp/bundle.raucb
log_level = debug
```

Nel file sono presenti l’indirizzo del server Hawkbit, il certificato ssl, la posizione dove verranno scaricati gli aggiornamenti, il nome e il token relativi alla Raspberry. Questi ultimi

due sono necessari per l'autenticazione del dispositivo e devono coincidere con quelli presenti nel server.

Casync

Casync è supportato dal layer meta-rauc: è sufficiente aggiungere la stringa "casync" alla variabile `IMAGE_INSTALL_append` affinché venga installato nella build. RAUC una volta ricevuto il bundle grazie all'applicazione rauc-hawkbit, lo esamina e se trova un file di tipo .caidx o .caibx (ovvero un "index file") esegue casync, il quale ricostruirà l'immagine binaria che sarà poi installata nello slot inattivo. I dettagli di questa procedura saranno approfonditi in seguito.

BitBake e generazione dell'immagine binaria

Una volta completata la configurazione di tutti i layer, per generare l'immagine binaria occorre eseguire, all'interno della cartella poky, i seguenti comandi da terminale:

```
source oe-init-build-env
bitbake core-image-base
```

Questo genererà sia l'intera immagine binaria contenente bootloader, kernel e i due filesystem ridondanti, sia l'immagine binaria di un solo filesystem. La prima sarà quella che verrà inizialmente caricata in maniera manuale sulla scheda micro sd della Raspberry, la seconda servirà per generare gli aggiornamenti OTA.

Architettura del sistema

Tramite una fase di reverse engineering è stato possibile ricostruire la struttura di tutto il sistema presente nel dispositivo target e di come le varie componenti interagiscono. Questo è mostrato nella Figura 20, un diagramma UML di tipo structural.

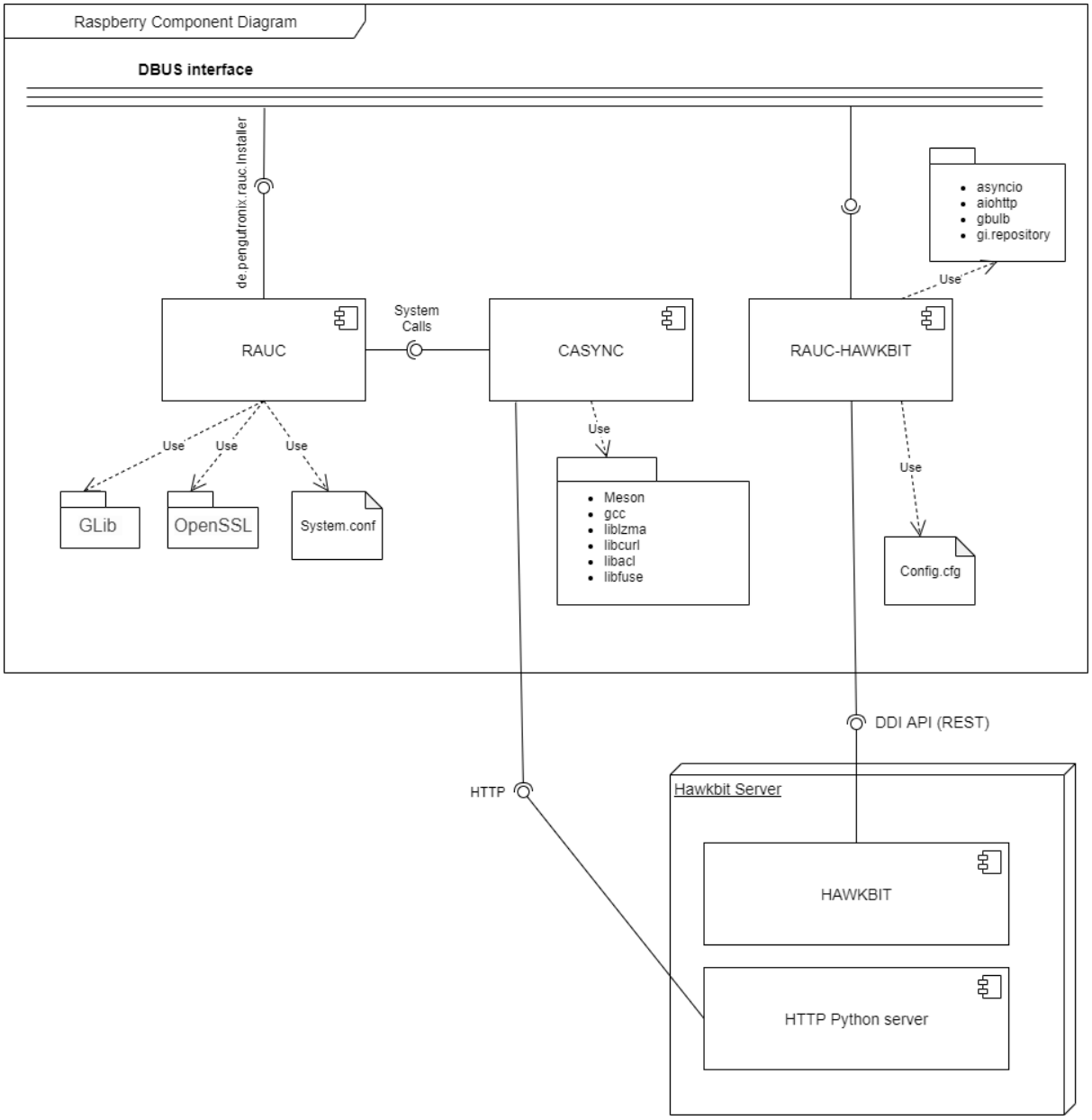


Figura 20

Interazione tra RAUC e la libreria RAUC-Hawkbit

La libreria python RAUC-Hawkbit svolge il ruolo di collegamento tra il server Hawkbit e il software RAUC presente nel dispositivo. Con il primo comunica mediante le “Direct Device Integration”, delle API HTTP di tipo RESTful, con il secondo mediante interfaccia D-Bus. Se eseguita come client mediante il comando da terminale

```
rauc-hawkbit-client -c config.cfg
```

questa inizierà a fare polling verso il server Hawkbit secondo le impostazioni passate dal file `config.cfg`. Le righe seguenti sono un estratto della classe `RaucDBUSDDIClient` presente all’interno della libreria.

```
        # DBUS proxy
        self.rauc = self.new_proxy('de.pengutronix.rauc.Installer', '/')

        # DBUS property/signal subscription
        self.new_property_subscription('de.pengutronix.rauc.Installer',
                                       'Progress', self.progress_callback)
        self.new_property_subscription('de.pengutronix.rauc.Installer',
                                       'LastError', self.last_error_callback)
        self.new_signal_subscription('de.pengutronix.rauc.Installer',
                                     'Completed', self.complete_callback)
```

Quando viene inizializzata, la classe crea una proxy D-Bus chiamandola rauc e si sottoscrive a dei segnali D-Bus mediante i quali RAUC comunica lo stato di installazione nel bundle. Una proxy D-Bus è un oggetto nativo creato per rappresentare un oggetto in un altro processo. Quando si invoca un metodo sulla proxy questo viene convertito in messaggio D-Bus e viene inviato al processo destinatario, il quale a sua volta risponde con un messaggio contenente i valori di ritorno del proprio metodo.

```
async def poll_base_resource(self):
        """Poll DDI API base resource."""
        while True:
            base = await self.ddi()

            if '_links' in base:
                if 'configData' in base['_links']:
                    await self.identify(base)
                if 'deploymentBase' in base['_links']:
                    await self.process_deployment(base)
                if 'cancelAction' in base['_links']:
                    await self.cancel(base)

            await self.sleep(base)
```

La funzione `poll_base_resource(self)` in intervalli di tempo definiti dalla funzione `sleep(base)` si identifica sul server mediante la funzione `identify(self, base)`.

```
async def identify(self, base):
        """Identify target against HawkBit."""
        self.logger.info('Sending identifying information to HawkBit')
        # identify
        await self.ddi.configData(
                ConfigStatusExecution.closed,
                ConfigStatusResult.success, **self.attributes)
```

La funzione `process_deployment(self, base)` controlla gli aggiornamenti, li scarica, verifica la loro integrità e avvia l'installazione tramite RAUC.

```
async def process_deployment(self, base):
        """
        Check for deployments, download them, verify checksum and trigger
        RAUC install operation.
        """
        if self.action_id is not None:
            self.logger.info('Deployment is already in progress')
            return

        # retrieve action id and resource parameter from URL
        deployment = base['_links']['deploymentBase']['href']
        match = re.search('/deploymentBase/(.+)\?c=(.+)$', deployment)
        action_id, resource = match.groups()
        self.logger.info('Deployment found for this target')
        # fetch deployment information
        deploy_info = await self.ddi.deploymentBase[action_id](resource)
        try:
            chunk = deploy_info['deployment']['chunks'][0]
        except IndexError:
            # send negative feedback to HawkBit
            status_execution = DeploymentStatusExecution.closed
            status_result = DeploymentStatusResult.failure
            msg = 'Deployment without chunks found. Ignoring'
            await self.ddi.deploymentBase[action_id].feedback(
                    status_execution, status_result, [msg])
            raise APIError(msg)

        try:
            artifact = chunk['artifacts'][0]
        except IndexError:
            # send negative feedback to HawkBit
            status_execution = DeploymentStatusExecution.closed
            status_result = DeploymentStatusResult.failure
            msg = 'Deployment without artifacts found. Ignoring'
            await self.ddi.deploymentBase[action_id].feedback(
                    status_execution, status_result, [msg])
            raise APIError(msg)
```

```
        # prefer https ('download') over http ('download-http')
        # HawkBit provides either only https, only http or both
        if 'download' in artifact['_links']:
            download_url = artifact['_links']['download']['href']
        else:
            download_url = artifact['_links']['download-http']['href']
```

In presenza di aggiornamenti disponibili, la funzione li scarica, controlla l'integrità, invia un feedback al server Hawkbit ed inizia l'installazione con RAUC. Per fare ciò invoca la funzione `self.install()`, la quale invoca il metodo Install sulla proxy D-Bus "rauc" definita nelle righe precedenti. Come argomento viene passata la posizione del bundle scaricato.

```
        # download artifact, check md5 and report feedback
        md5_hash = artifact['hashes']['md5']
        self.logger.info('Starting bundle download')
        await self.download_artifact(action_id, download_url, md5_hash)

        # download successful, start install
        self.logger.info('Starting installation')
        try:
            self.action_id = action_id
            # do not interrupt install call
            await asyncio.shield(self.install())
        except GLib.Error as e:
            # send negative feedback to HawkBit
            status_execution = DeploymentStatusExecution.closed
            status_result = DeploymentStatusResult.failure
            await self.ddi.deploymentBase[action_id].feedback(
                    status_execution, status_result, [str(e)])
            raise APIError(str(e))
```

La funzione tiene costantemente aggiornato il server Hawkbit sul progresso di aggiornamento e se si verifica un errore, questo viene notificato immediatamente.

Interazione RAUC-casync

La Figura 21 mostra un diagramma UML di tipo structural che evidenzia le principali relazioni tra le varie componenti dei due software all'interno del dispositivo target.

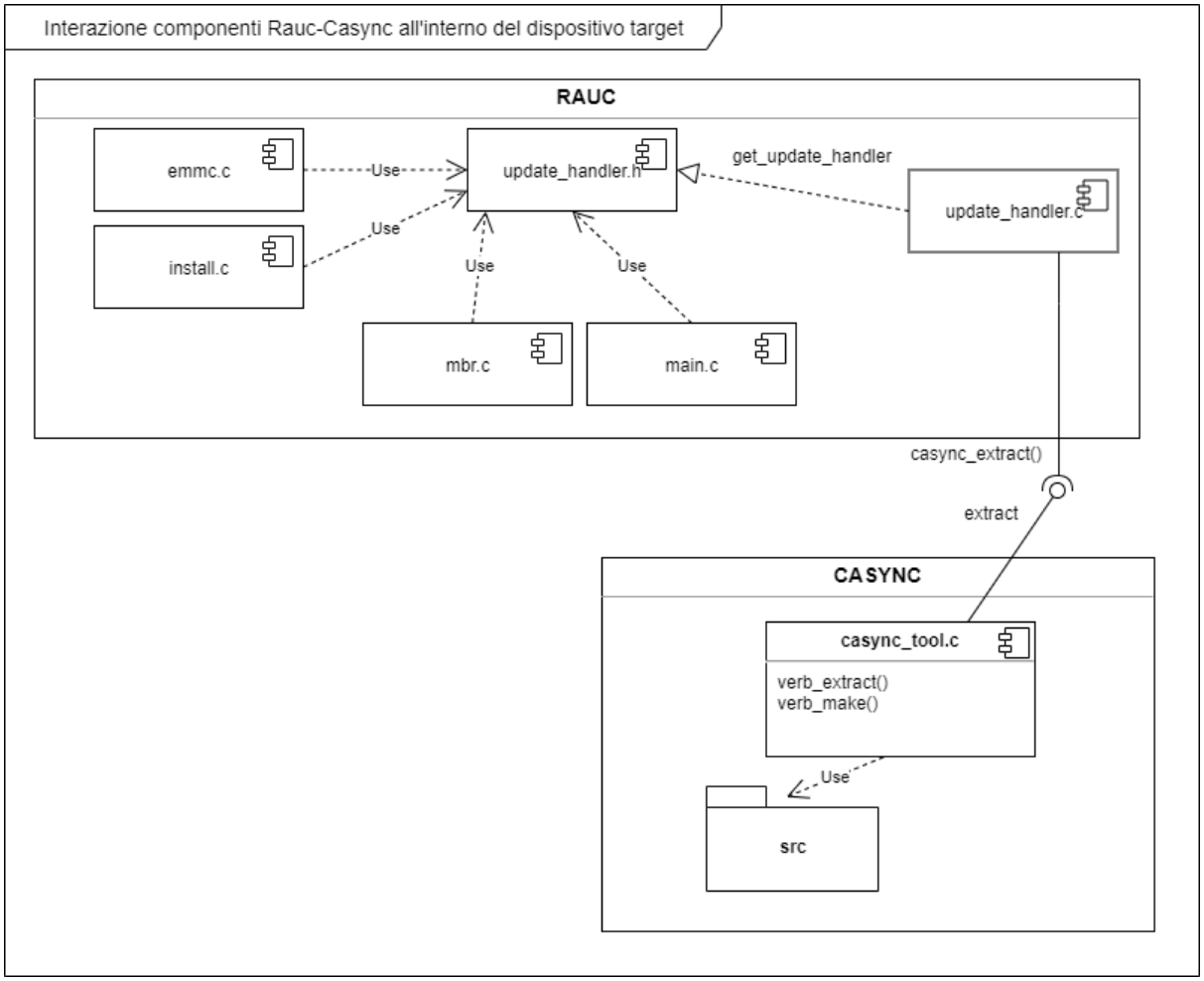


Figura 21

Una volta che il bundle è stato scaricato dal client RAUC-Hawkbit, RAUC riceve l'input di installarlo mediante messaggio D-Bus. Le righe che seguono sono un estratto del file `update_handler.c` e mostrano come viene gestita l'installazione nel caso in cui il bundle sia di tipo casync.

```
img_to_slot_handler get_update_handler(RaucImage *mfimage, RaucSlot
*dest_slot, GError **error)
{
    const gchar *src = mfimage->filename;
    const gchar *dest = dest_slot->type;
    img_to_slot_handler handler = NULL;

    /* If we have a custom install handler, use this instead of selecting an
existing one */
    if (mfimage->hooks.install) {
```

```
        return hook_install_handler;
    }

    g_message("Checking image type for slot type: %s", dest);

    for (RaucUpdatePair *updatepair = updatepairs; updatepair->handler !=
NULL; updatepair++) {
        if (g_pattern_match_simple(updatepair->src, src) &&
            g_pattern_match_simple(updatepair->dest, dest)) {
            g_message("Image detected as type: %s", updatepair->src);
            handler = updatepair->handler;
            break;
        }
    }

    if (handler == NULL)  {
        g_set_error(error, R_UPDATE_ERROR, R_UPDATE_ERROR_NO_HANDLER,
"Unsupported image %s for slot type %s",
                mfimage->filename, dest);
        goto out;
    }

out:
    return handler;
}
```

Il metodo `get_update_handler` prende l'immagine presente nel bundle e controlla che questa sia compatibile con lo slot nel quale deve essere installata. Se risulta compatibile l'immagine viene inviata ad un handler in base al proprio formato. Nel caso l'immagine sia di tipo casync (.catar, .cadix), questa viene processata dalla funzione `casync_extract_image`.

```
static gboolean unpack_archive(RaucImage *image, gchar *dest, GError **error)
{
    if (g_str_has_suffix(image->filename, ".caidx" ))
        return casync_extract_image(image, dest, error);
    else if (g_str_has_suffix(image->filename, ".catar" ))
        return casync_extract_image(image, dest, error);
    else
        return untar_image(image, dest, error);
}
```

```
static gboolean casync_extract_image(RaucImage *image, gchar *dest, GError
**error)
```

```
{
    ...

    /* Prepare Seed */
    seedslot = get_active_slot_class_member(image->slotclass);
    if (!seedslot) {
        g_warning("No seed slot available for %s", image->slotclass);
        goto extract;
    }

    if (g_str_has_suffix(image->filename, ".caidx" )) {
        /* We need to have the seed slot (bind) mounted to a distinct
         * path to allow seeding. E.g. using mount path '/' for the
         * rootfs slot seed is inaproppriate as it contains virtual
         * file systems, additional mounts, etc. */
        if (!seedslot->mount_point) {
            g_debug("Mounting %s to use as seed", seedslot->device);
            res = r_mount_slot(seedslot, &ierror);
            if (!res) {
                g_warning("Failed mounting for seeding: %s", ierror->message);
                g_clear_error(&ierror);
                goto extract;
            }
            seed_mounted = TRUE;
        }

        g_debug("Adding as casync directory tree seed: %s", seedslot-
>mount_point);
        seed = g_strdup(seedslot->mount_point);
    } else {
        g_debug("Adding as casync blob seed: %s", seedslot->device);
        seed = g_strdup(seedslot->device);
    }
```

Nella prima parte viene selezionato lo slot attivo della stessa classe dello slot da aggiornare. Questo viene montato per essere usato come “seed”.

```
extract:
    /* Set store */
    store = r_context()->install_info->mounted_bundle->storepath;
    g_debug("Using store path: '%s'", store);
```

RAUC imposta lo “storepath”, ovvero la posizione dove casync andrà a scaricare i chunk aggiuntivi non presenti nel dispositivo. Per default lo storepath è situato allo stesso indirizzo dove il bundle è stato scaricato. Nel nostro caso però, andando a modificare il file è stato impostato un indirizzo che puntasse al server HTTP realizzato con Python.

```
    /* Set temporary directory */
    tmpdir = r_context()->config->tmp_path;
    if (tmpdir)
        g_debug("Using tmp path: '%s'", tmpdir);
```

Qui viene impostata la directory temporanea dove

```
    /* Call casync to extract */
    res = casync_extract(image, dest, seed, store, tmpdir, &ierror);
    if (!res) {
        g_propagate_error(error, ierror);
        goto out;
    }
```

Viene invocata la funzione `casync_extract` che a sua volta effettua una system call per casync. Gli argomenti passati sono:

- image, contenente l’index file relativo alla nuova versione.
- dest, la posizione dove verrà ricostruita l’immagine. In questo caso è lo slot inattivo da aggiornare.
- seed, lo slot attualmente attivo.
- store, la posizione dalla quale casync attingerà i chunk aggiuntivi.
- tmdir, la directory temporanea dove casync ricostruirà l’index file relativo al seed.

Una volta ricreato l’index file partendo dal seed fornitogli, casync lo compara con quello proveniente dal bundle RAUC. Ricostruisce l’immagine del nuovo software, salvandola nello slot inattivo, mediante l’utilizzo dei chunk provenienti sia dal seed, sia dallo store, ovvero dal server HTTP.

```
    /* Cleanup seed */
    if (seed_mounted) {
        res = r_umount_slot(seedslot, &ierror);
        if (!res) {
            g_propagate_prefixed_error(error, ierror, "Failed unmounting seed
slot: ");
            goto out;
        }
    }
```

```
    res = TRUE;
out:
    return res;
}
```

Alla fine dell'operazione l'immagine della nuova versione del software sarà stata installata nello slot inattivo. RAUC modifica le variabili U-Boot affinché venga avviata la partizione aggiornata, e poi comunica l'avvenuta installazione mediante segnale D-Bus al client RAUC-Hawkbit che a sua volta manda un feedback al server Hawkbit usando le API DDI.

Risultati ottenuti

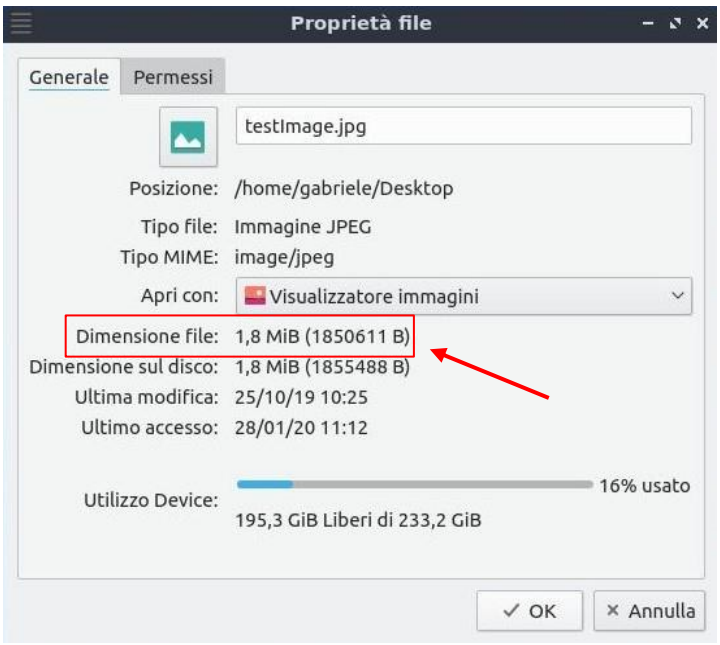
Esempio di un aggiornamento OTA

Di seguito verrà elencata la procedura che consente di effettuare un aggiornamento OTA sfruttando il sistema sviluppato.

Versioni software

Al fine di simulare aggiornamenti OTA di tipo differenziale, sono state prodotte 3 versioni del filesystem della Raspberry:

- Versione 0, il filesystem originale, privo di modifiche. La sua dimensione è di 1.304.428.544 byte.
- Versione 1, il filesystem originale con un'immagine jpg su /testimmagine.jpg.
- Versione 2, il filesystem originale con un'immagine jpg su /home/testimmagine.jpg.



È stata scelta un'immagine di tipo .jpg (Figura 22) poichè, essendo già di sua natura un file compresso, quando attraversa un processo di compressione la sua dimensione varia in maniera poco significativa. Questo ha permesso di stimare l'efficienza del sistema differenziale. Sono stati eseguiti dei test su filesystem di prodotti sviluppati da ART stessa oltre a quelli sulle versioni software sopra citate.

Figura 22

Creazione dei Bundle

All'interno del client sono stati prodotti i bundle RAUC di tutte le versioni software. La Figura 23 mostra la dimensione dei bundle relativi alle versioni del filesystem della Raspberry, mentre la Figura 24 mostra la dimensione dei bundle dei firmware ART. Nella prima immagine possiamo notare come la dimensione del primo bundle sia notevolmente compressa rispetto all'originale (da circa 10435 MiB a circa 58 MiB) mentre quelle degli altri due differiscono dalla prima per la dimensione dell'immagine aggiunta. I firmware ART invece sono passati da 1.3 GiB a 373 MiB uno e 356 MiB l'altro. Con questi dati alla mano si evince l'ottimo rapporto di compressione offerto da RAUC.

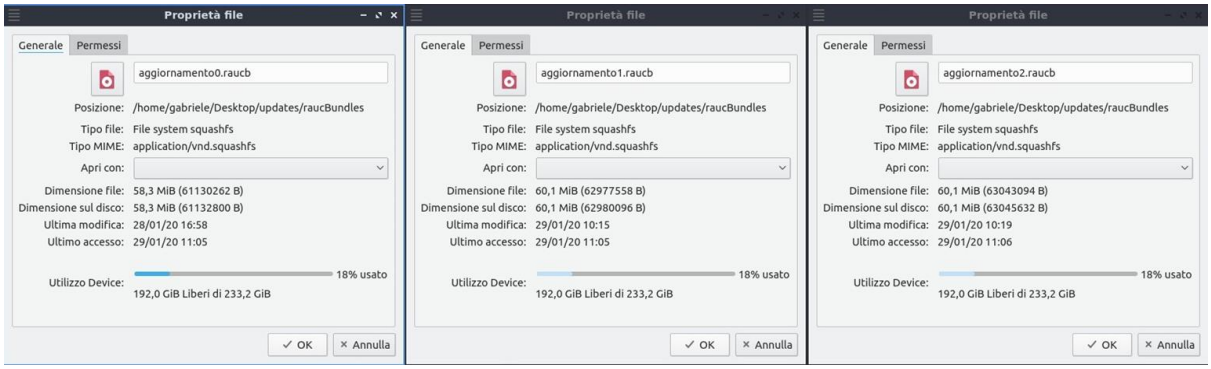


Figura 23

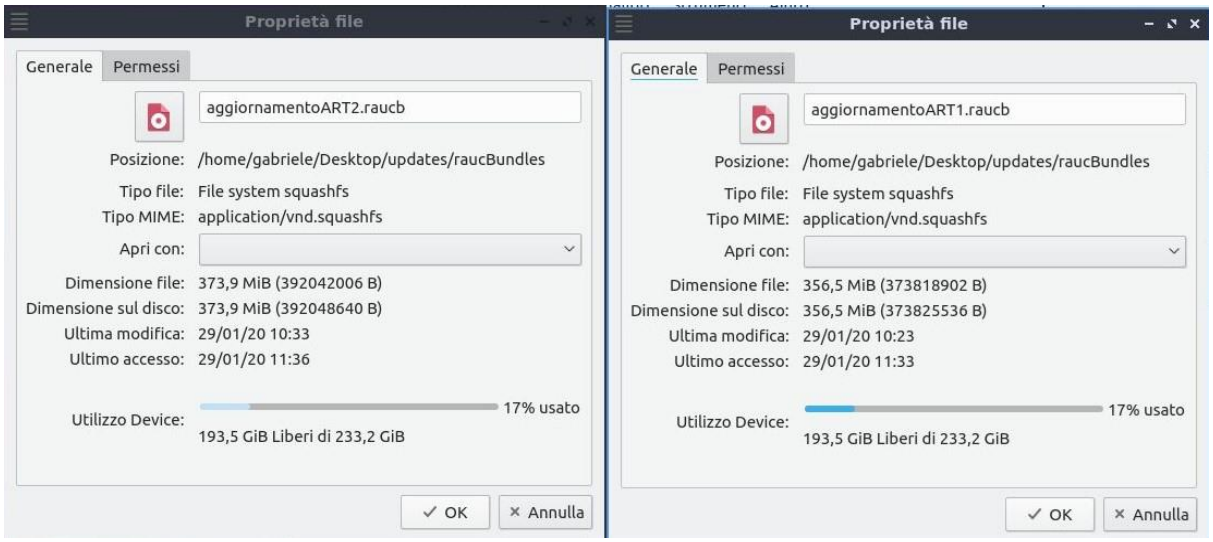


Figura 24

Conversione dei Bundle

```
gabriele@gabriele-pc:~/Desktop/rauc$ ./rauc convert --cert=development-1.cert.pem --key=development-1.key.pem --keyring=ca.ce
rt.pem  aggiornamento0.raucb aggiornamento0_casync.raucb
rauc-Message: 11:10:42.794: Reading bundle: aggiornamento0.raucb
rauc-Message: 11:10:42.800: Verifying bundle...
rauc-Message: 11:10:43.536: Converting rootfs.ext3.img to blob idx rootfs.ext3.img.caibx
rauc-Message: 11:11:00.029: Keyring given, doing signature verification
rauc-Message: 11:11:00.031: Keyring given, doing signature verification
Bundle written to aggiornamento0_casync.raucb
gabriele@gabriele-pc:~/Desktop/rauc$
```

Figura 25

Sono stati convertiti tutti i bundle creati in precedenza. Il processo genera altrettanti chunk store e relativi bundle contenenti gli index file (Figura 25). Nel caso in cui RAUC trova un chunk store preesistente, vi aggiunge solo i chunk non ridondanti (Figura 26).

```
gabriele@gabriele-pc:~/Desktop/rauc$ ./rauc convert --cert=development-1.cert.pem --key=development-1.key.pem --keyring=ca.ce
rt.pem  aggiornamento1.raucb aggiornamento1_casync.raucb
rauc-Message: 11:18:07.682: Reading bundle: aggiornamento1.raucb
rauc-Message: 11:18:07.687: Verifying bundle...

(rauc:5182): rauc-WARNING **: 11:18:07.819: Store path 'aggiornamento1_casync.raucb' already exists, appending new chunks
rauc-Message: 11:18:08.427: Converting rootfs.ext3.img to blob idx rootfs.ext3.img.caibx
rauc-Message: 11:18:23.844: Keyring given, doing signature verification
rauc-Message: 11:18:23.846: Keyring given, doing signature verification
Bundle written to aggiornamento1_casync.raucb
gabriele@gabriele-pc:~/Desktop/rauc$
```

Figura 26

La Figura 27 mostra la dimensione dei bundle contenenti l'index file. Nelle prove effettuate, sia per le versioni software della Raspberry, sia per le versioni software ART, la dimensione dei bundle RAUC-casync non ha mai superato 1 MiB.

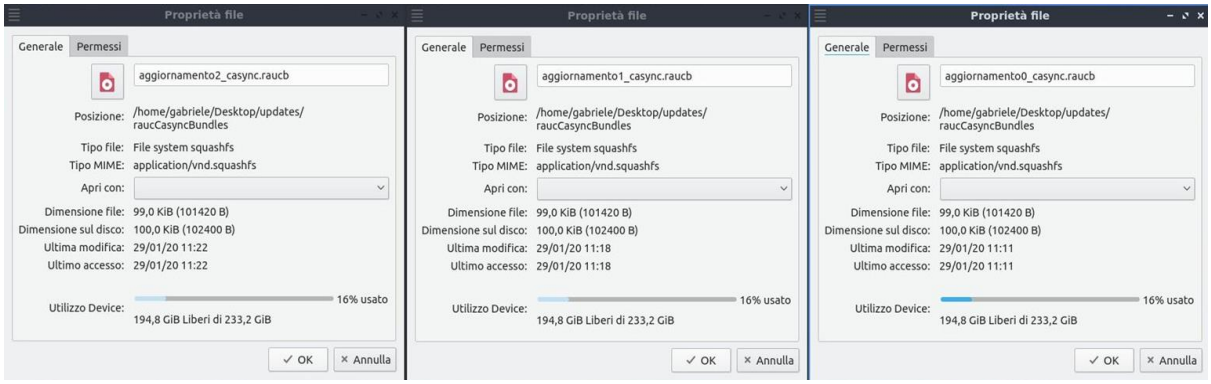


Figura 27

Dimensione del chunk store dopo la prima conversione (Figura 28).

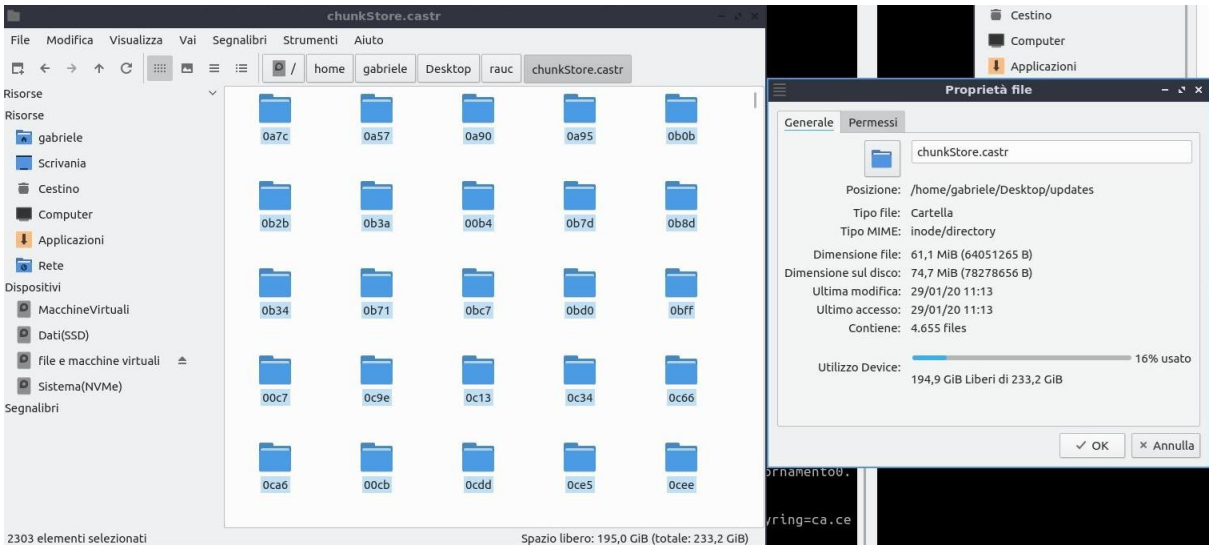


Figura 28

Dimensione del chunk store dopo la seconda conversione (Figura 29).

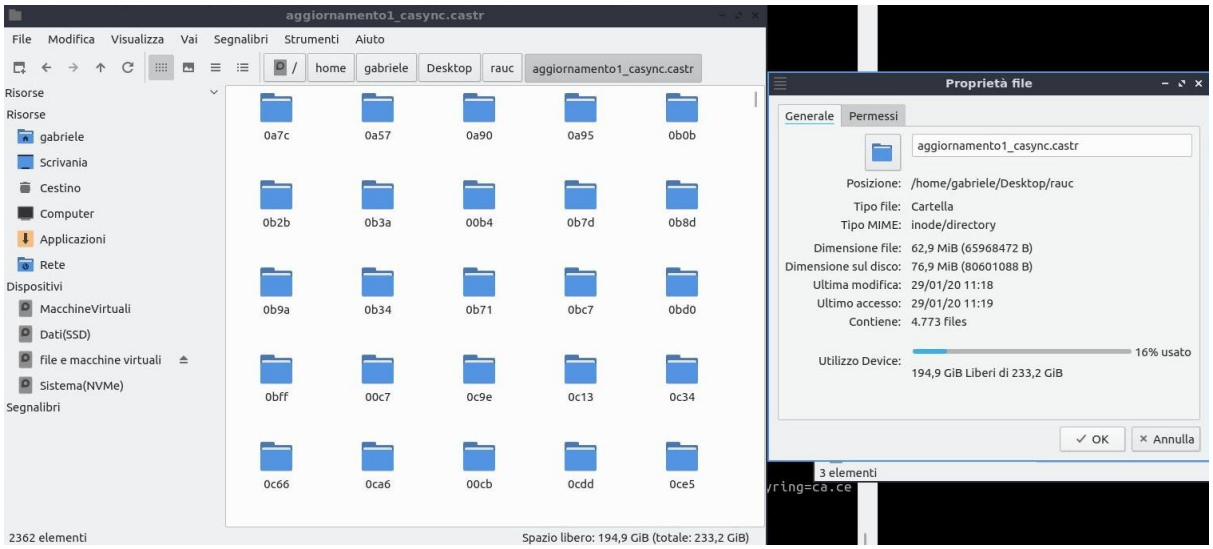


Figura 29

Dimensione del chunk store dopo la terza conversione (Figura 30).

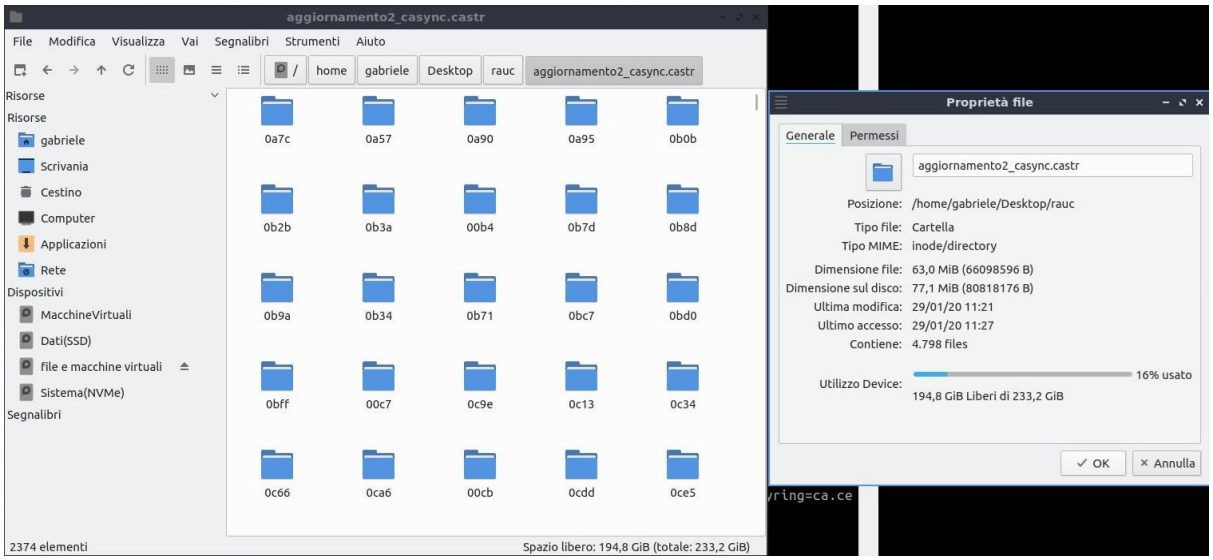


Figura 30

Arrivati alla terza conversione, notiamo che il chunk store contenente tutte e tre le versioni software è passato da una dimensione iniziale di 61 MiB ad una finale di 63 MiB. Questo perché il processo di conversione ha aggiunto al chunk store solo le parti “nuove” di ogni versione, lasciando i chunk comuni inalterati. Per quanto riguarda il rapporto di compressione, notiamo che questo è estremamente simile a quello offerto soltanto da RAUC: sia le dimensioni del chunk store sia le dimensioni dei bundle contenenti l’intera immagine si aggirano sui 60 MiB.

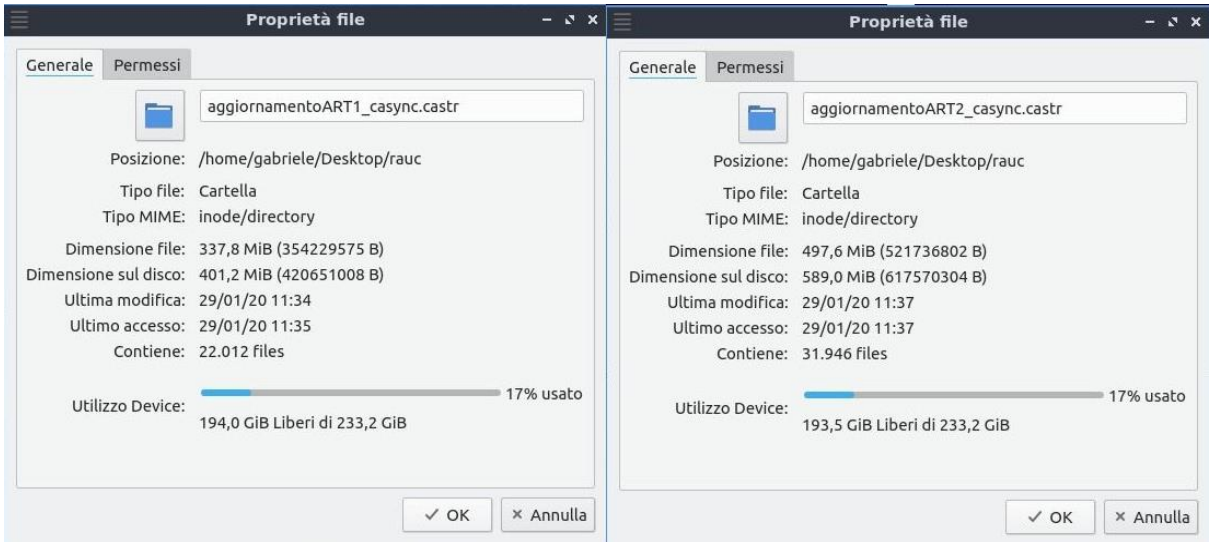


Figura 31

La Figura 31 mostra la dimensione del chunk Store contenente il software ART. La prima dimensione è relativa ad una versione software soltanto, mentre la seconda le contiene tutte e due: passando da 337 MiB a 497 MiB si evince che solo una parte della seconda versione è stata aggiunta.

Definizione del dispositivo target

Viene definito il dispositivo target (Figura 32, Figura 33) indicando il “controllerID”, che sarà il suo identificativo, ed il nome. Completata la procedura Hawkbit genererà il token attraverso il quale il dispositivo si autenticherà sul server assieme al proprio ID (Figura34).

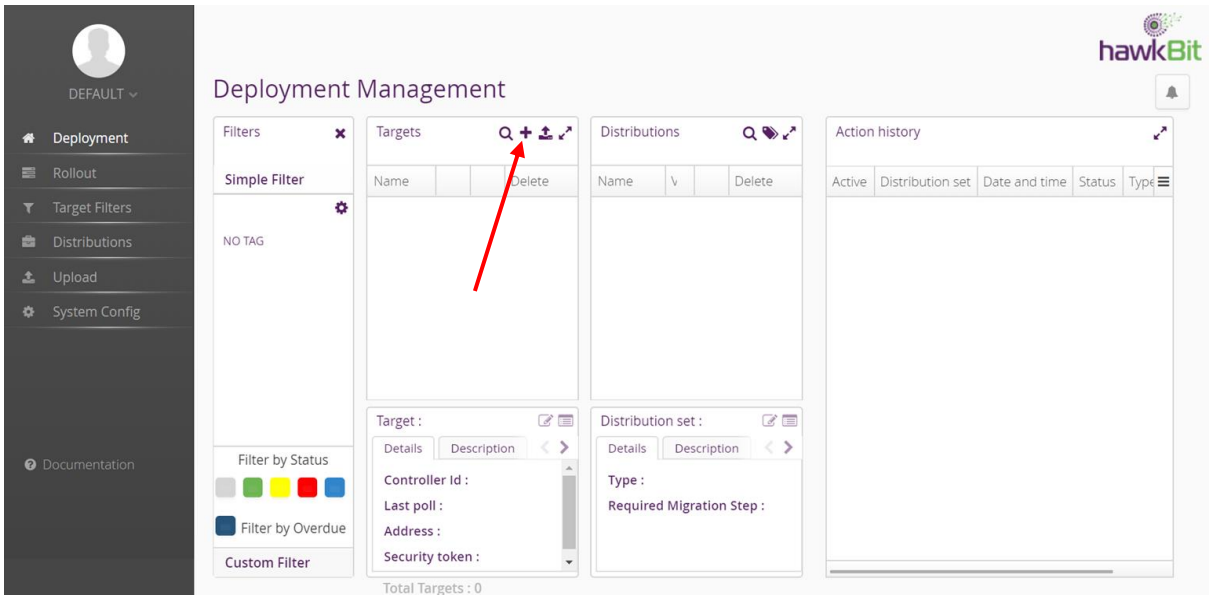


Figura 32

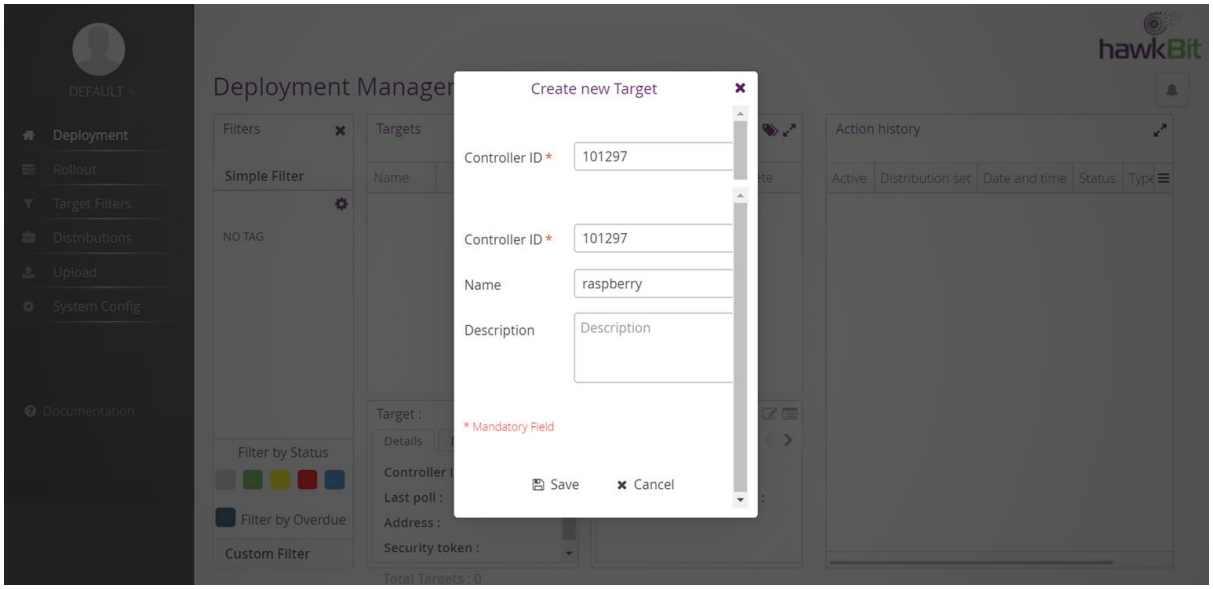


Figura 33

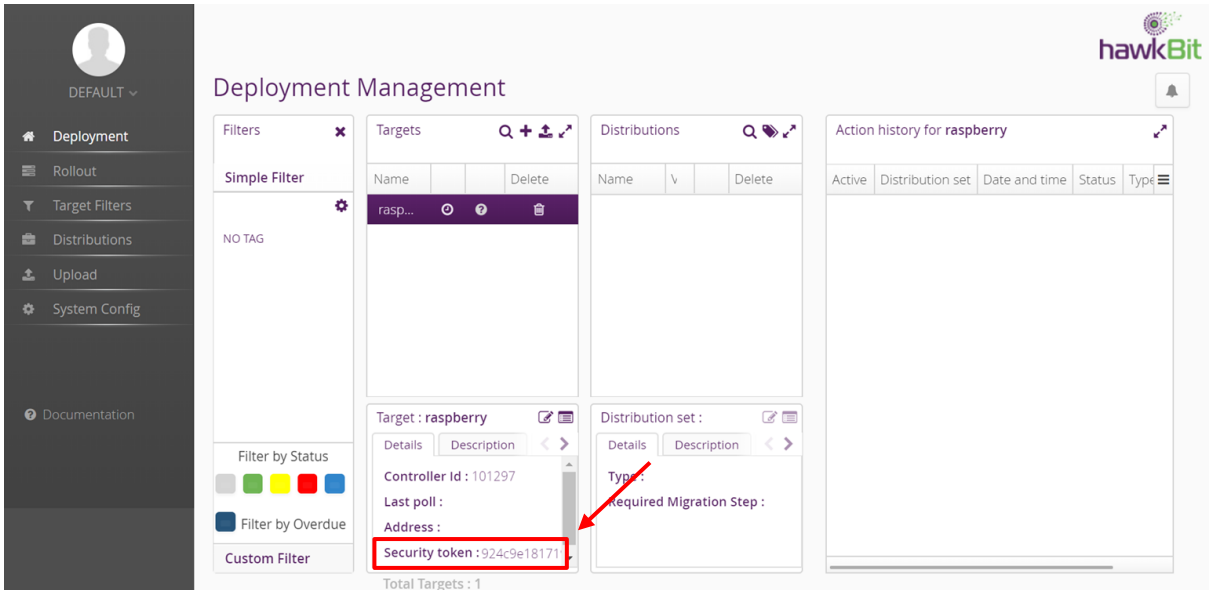


Figura 34

Caricamento dei bundle nel server Hawkbit

Viene creato un Modulo software per ciascun Bundle (Figura 35, Figura 36).

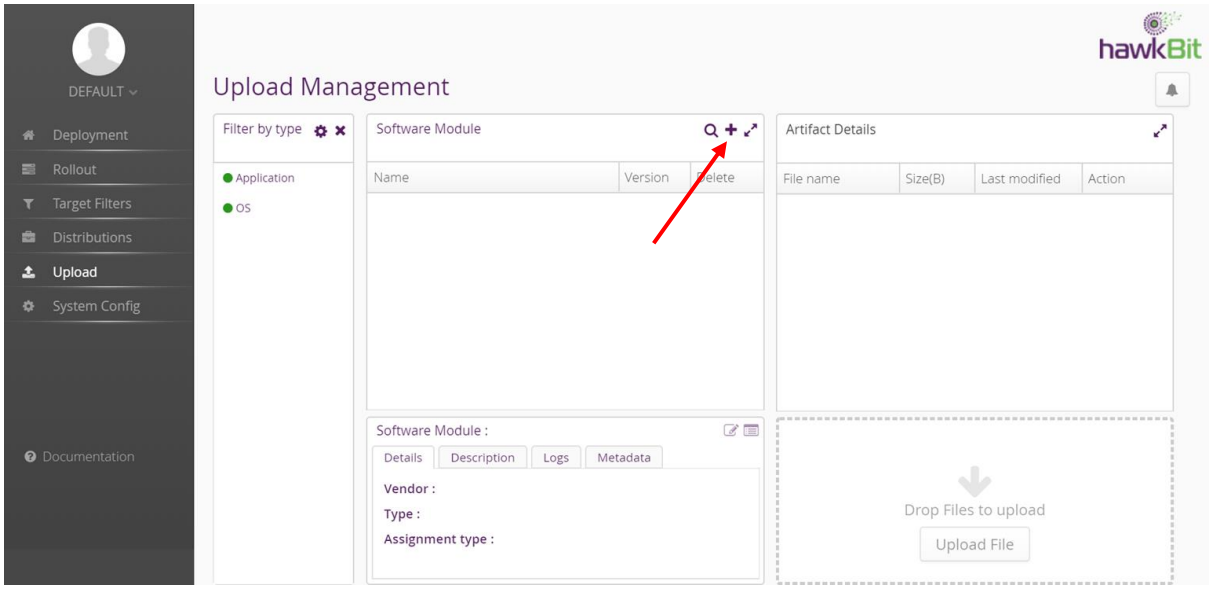


Figura 35

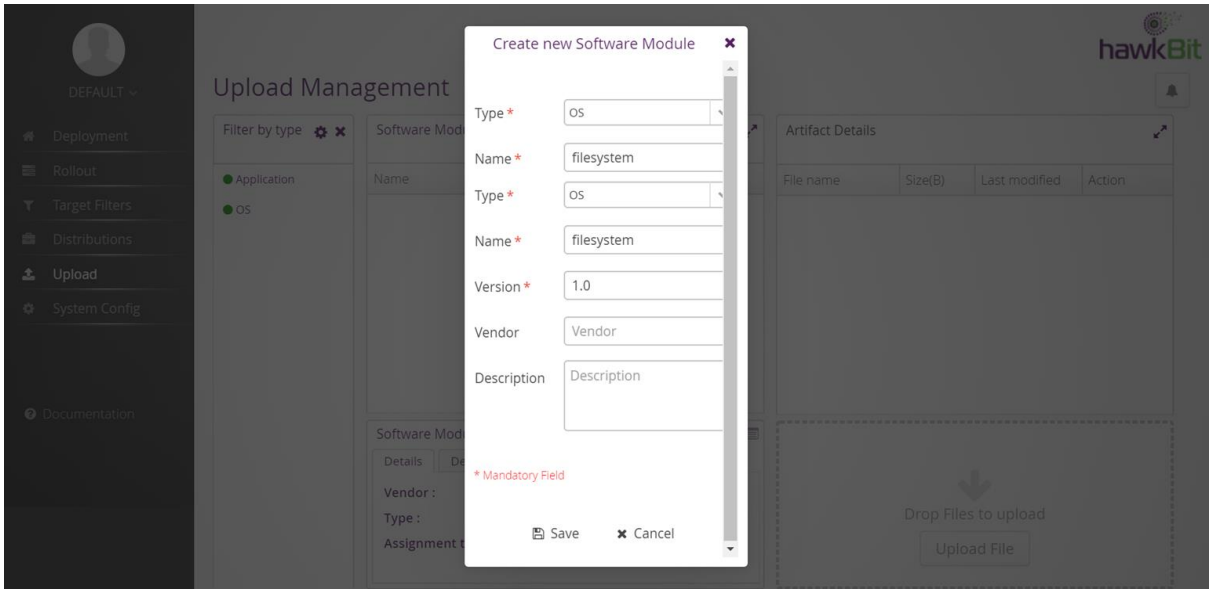


Figura 36

Successivamente viene caricato il bundle come artefatto appartenente al modulo software (Figura 37).

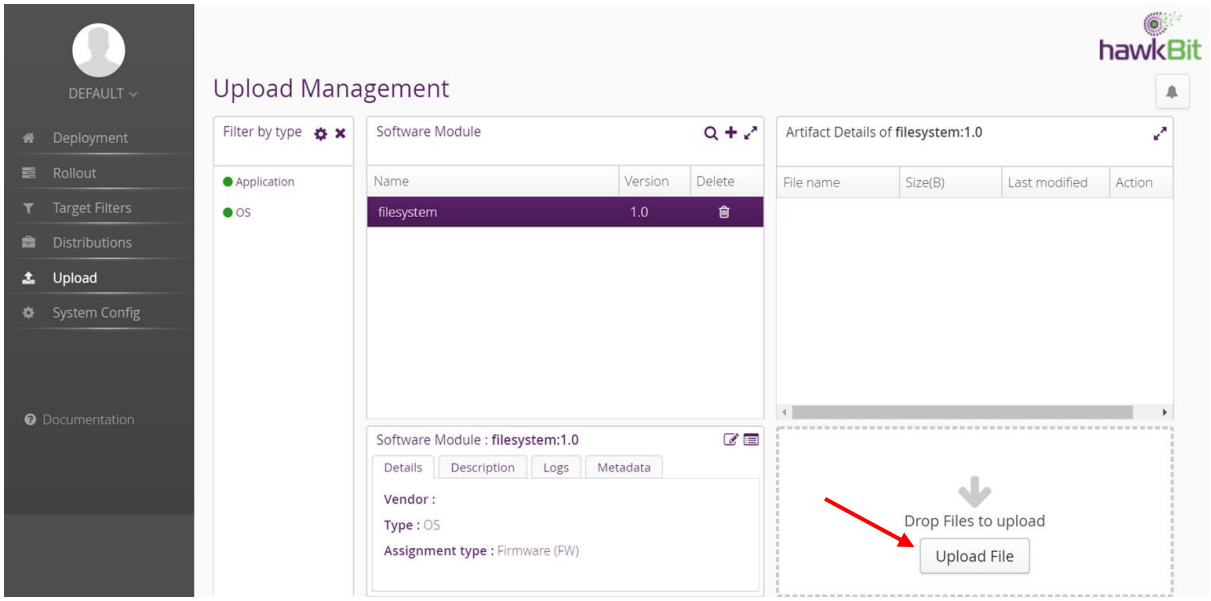


Figura 37

Poi viene creata una distribuzione ed assegnato il relativo modulo software (Figura 38, Figura 39).

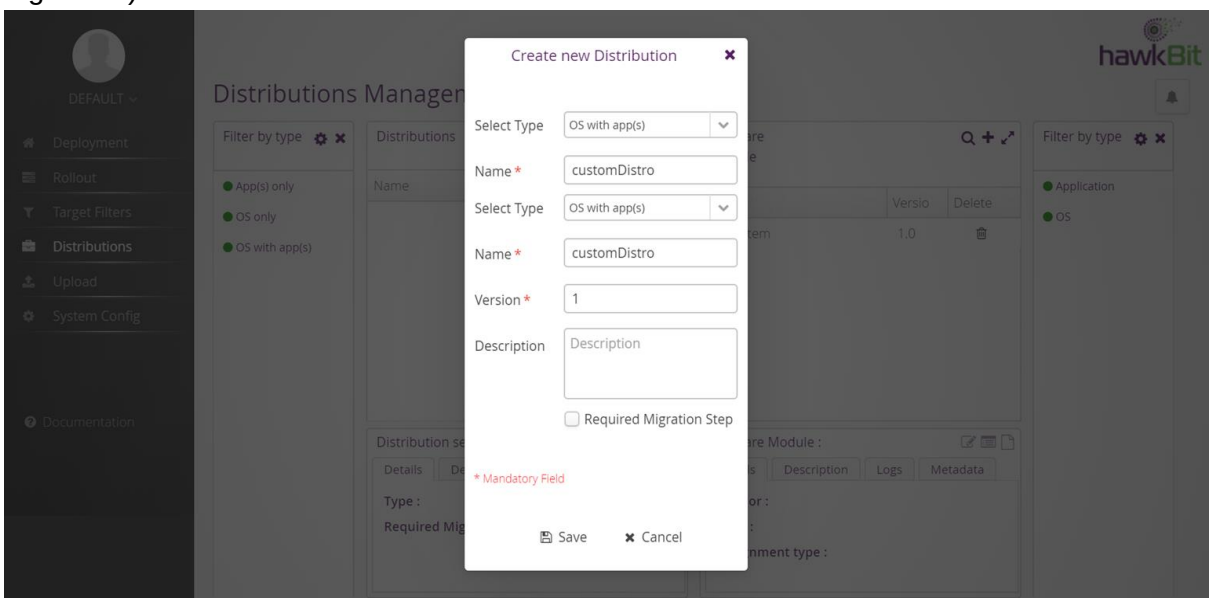


Figura 38

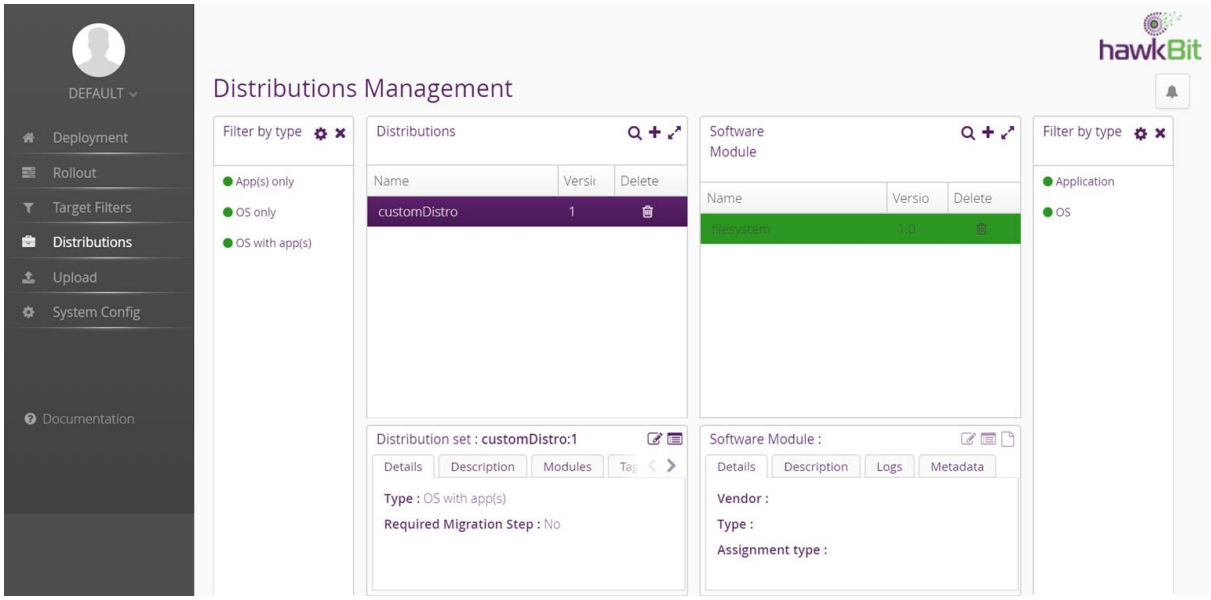


Figura 39

Assegnazione dell'aggiornamento

Una volta creato il target e caricati i bundle nel server è possibile assegnare un aggiornamento ad un determinato dispositivo (Figura 40).

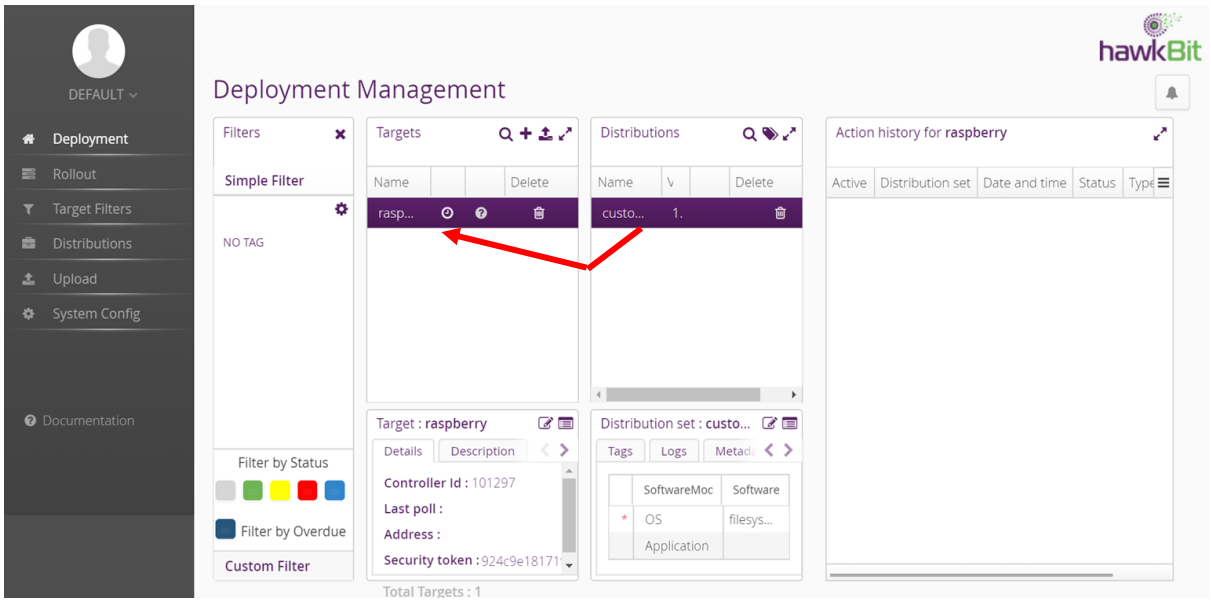


Figura 40

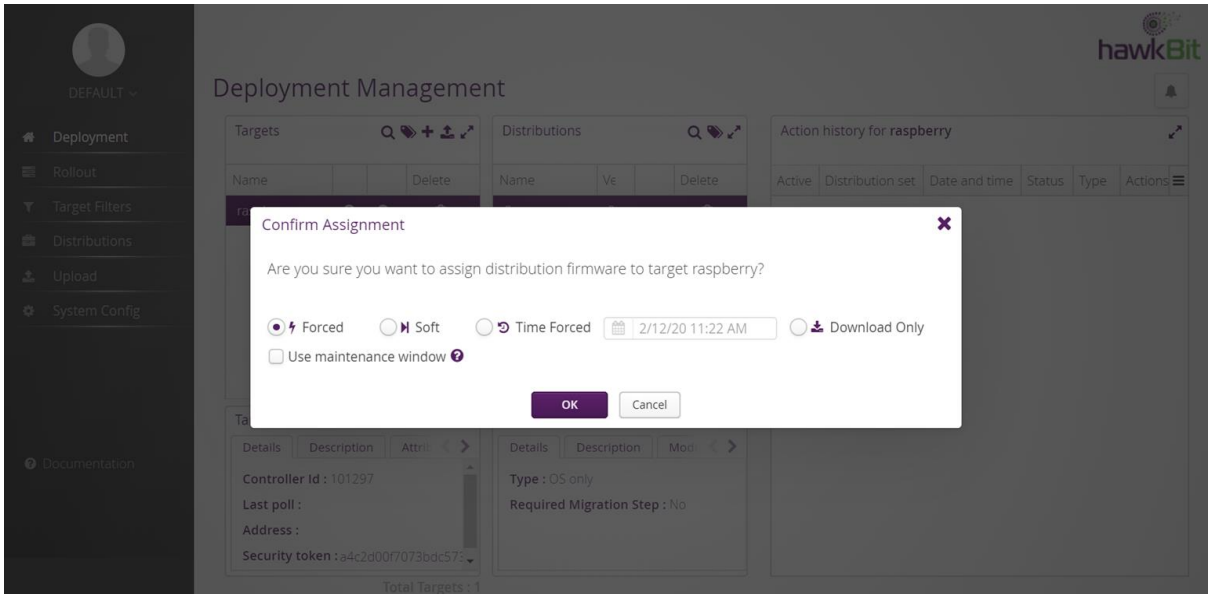


Figura 41

Apparirà una finestra pop-up di conferma dove è possibile selezionare il tipo di rilascio (Figura 41):

- Forced, appena il dispositivo ha scaricato l'aggiornamento inizia l'installazione.
- Soft, il dispositivo installa l'aggiornamento quando lo ritiene necessario.
- Time Forced, l'aggiornamento passa da soft a forced dopo un certo lasso di tempo.

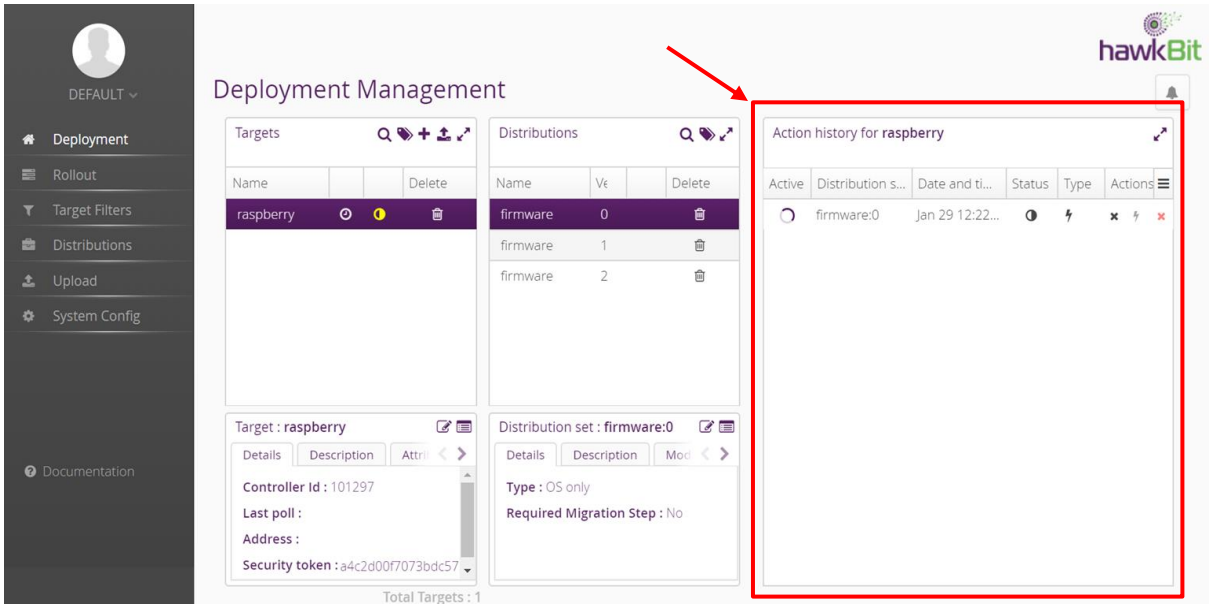


Figura 42

Quando viene assegnato l'aggiornamento al dispositivo, il suo stato è riportato nello storico del target (Figura 42). Se si espande tale sezione vengono riportati anche i singoli messaggi di feedback mandati dal dispositivo durante l'aggiornamento (Figura 43).

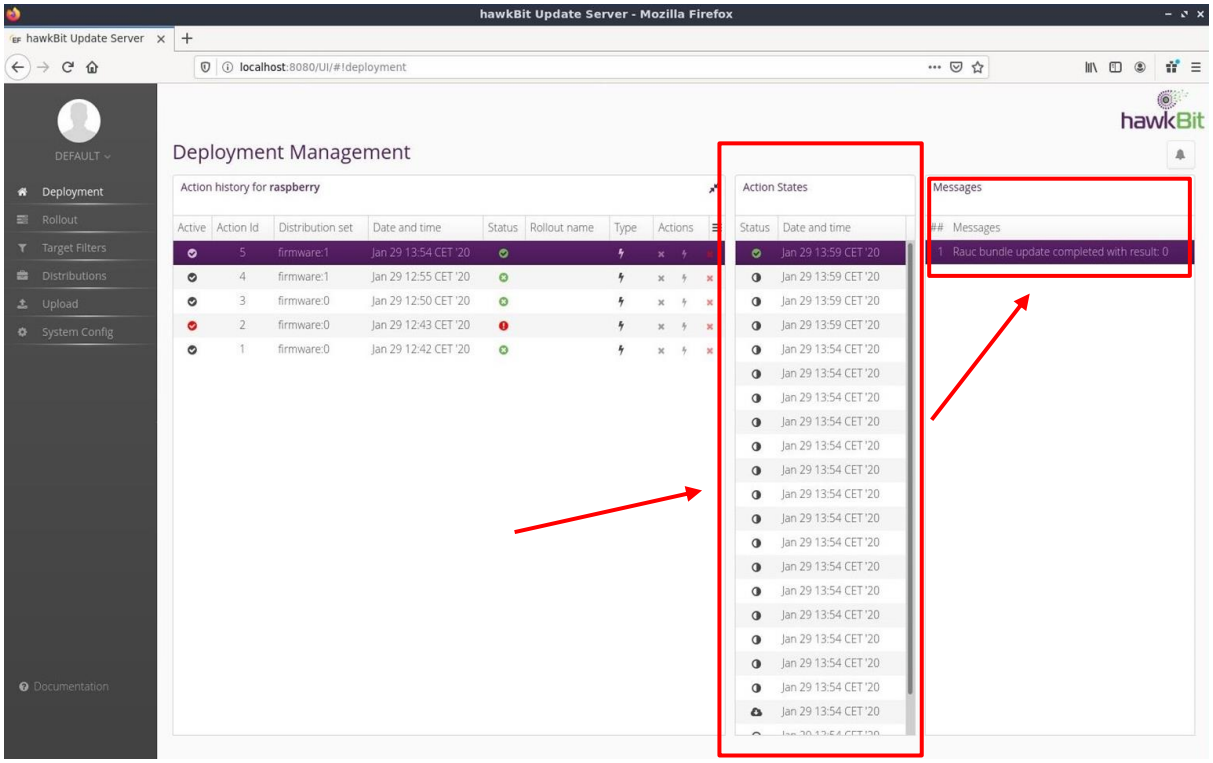

hawkBit Update Server - Mozilla Firefox
hawkBit Update Server
localhost:8080/UI/#!deployment
DEFAULT
Deployment
Rollout
Target Filters
Distributions
Upload
System Config
Documentation
hawkBit
Deployment Management
Action history for raspberry
Active
Action Id
Distribution set
Date and time
Status
Rollout name
Type
Actions
5 firmware:1 Jan 29 13:54 CET '20
4 firmware:1 Jan 29 12:55 CET '20
3 firmware:0 Jan 29 12:50 CET '20
2 firmware:0 Jan 29 12:43 CET '20
1 firmware:0 Jan 29 12:42 CET '20
Action States
Status
Date and time
Jan 29 13:59 CET '20
Jan 29 13:54 CET '20
Messages
Messages
1 Rauc bundle update completed with result: 0

Figura 43

Scaricamento e installazione dell’aggiornamento

Nella Raspberry viene avviata l’applicazione client RAUC-Hawkbit: questa inizia ad interrogare periodicamente il server Hawkbit, identificandosi e verificando la presenza di aggiornamenti disponibili. Nel caso in cui questi sono presenti, li scarica. La Figura 44 mostra un diagramma UML di tipo behavioral che riporta le principali fasi dell’aggiornamento all’interno del dispositivo target.

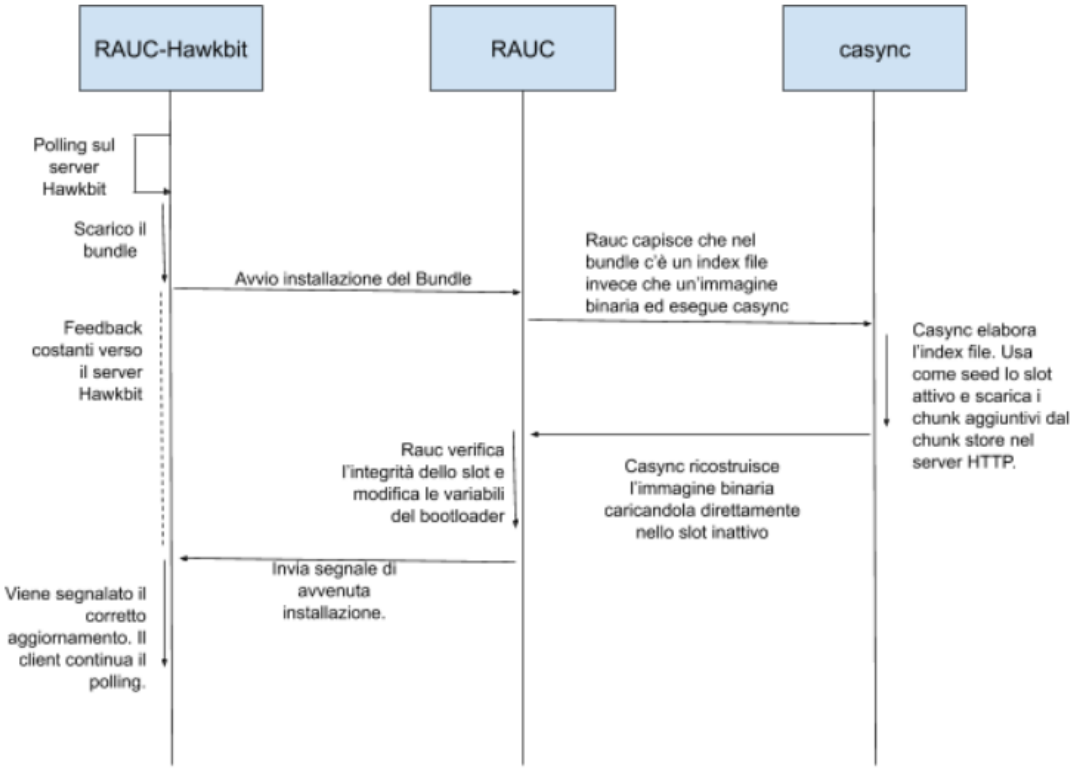


Figura 44

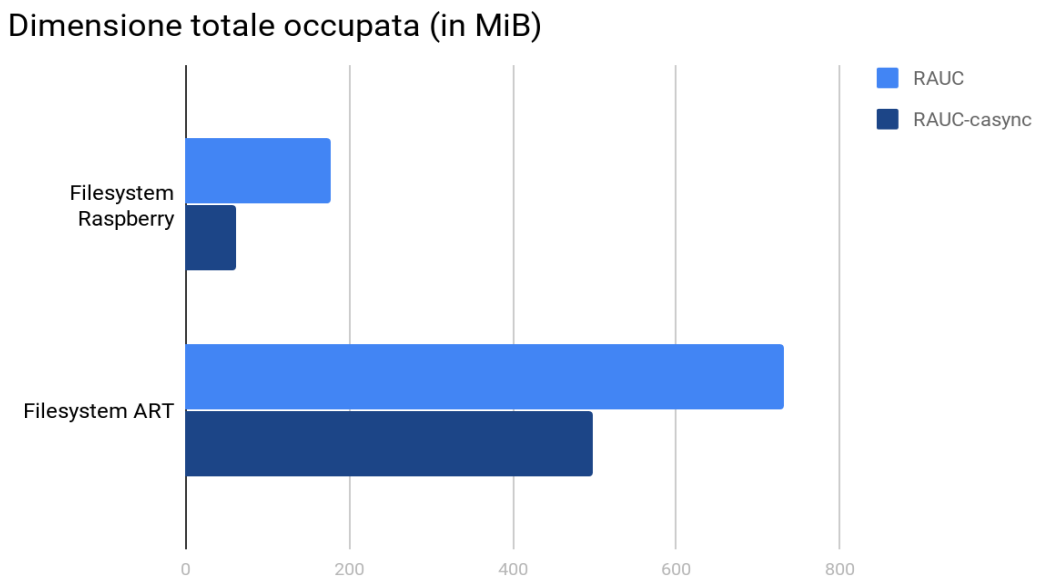
Considerazioni efficienza

Performance compressione

Utilizzando solo RAUC, la dimensione dei bundle passa da:

- Versione 0, da 10435 MiB a 58 MiB.
- Versione 1, da 10435 MiB a 60 MiB.
- Versione 2, da 10435 MiB a 60 MiB.
- Filesystem ART 1, da 1,3 GiB a 357 MiB.
- Filesystem ART 2, da 1,3 GiB a 374 MiB.

Per contenere tutti i bundle delle versioni software Raspberry sono quindi necessari 178 MiB, mentre per le due versioni dei Filesystem ART 731 MiB.
Utilizzando RAUC e Casync, la dimensione totale occupata dal software Raspberry è di 63 MiB, mentre quella occupata dal software ART è di 497 MiB. Tutto questo comporta un notevole risparmio di spazio all'interno del server. Tale risparmio è direttamente proporzionale al numero di versioni di uno stesso software, poiché le parti comuni non saranno duplicate.



Quantità di dati trasmessi

Al fine di testare l'efficienza del sistema sono state monitorate le trasmissioni tra il server e la Raspberry durante un aggiornamento OTA. Le rilevazioni sono state effettuate dal software Wireshark. La Figura 45 mostra il traffico avvenuto durante l'aggiornamento dalla versione 0 alla versione 1, mentre la Figura 46 riguarda l'aggiornamento dalla versione 1 alla versione 2. Il server ha indirizzo IP 192.168.100.100, mentre la Raspberry ha IP 192.168.100.105. Come è possibile vedere la quantità di dati trasmessa nel primo caso è 461 KByte, ovvero 3688 KiB, mentre nel secondo è 651 KByte, ovvero 5208 KiB. Considerando la dimensione degli header dei pacchetti, la dimensione del payload è molto vicina a quella dell'immagine aggiunta (1800 KiB). Le conclusioni sono:

- Il sistema risulta tanto più efficiente nella trasmissione dei dati quanto più simile è la nuova versione software rispetto alla precedente. Nel caso analizzato vengono trasmessi al massimo 5,2 MiB mentre le dimensioni del bundle standard RAUC si aggirano sui 60 MiB.
- Casync è sensibile alla posizione dei file nel filesystem. Non è stato capace di rilevare che la stessa immagine era stata spostata ed è stato necessario riscaricarla.

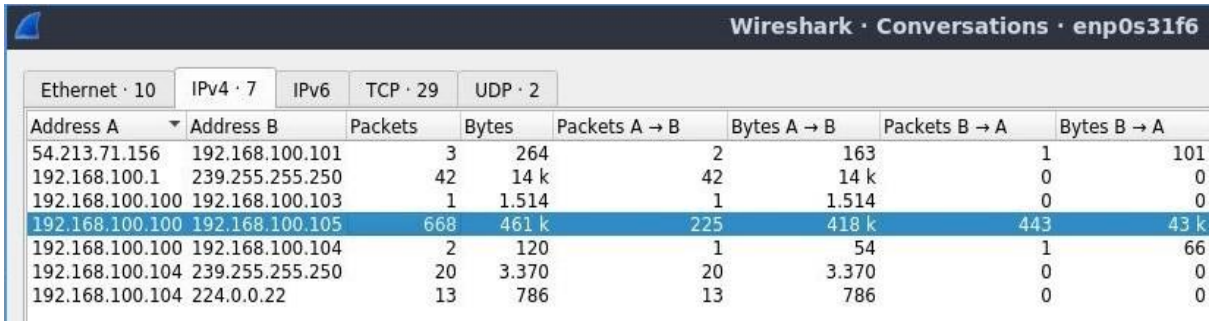


Figura 45

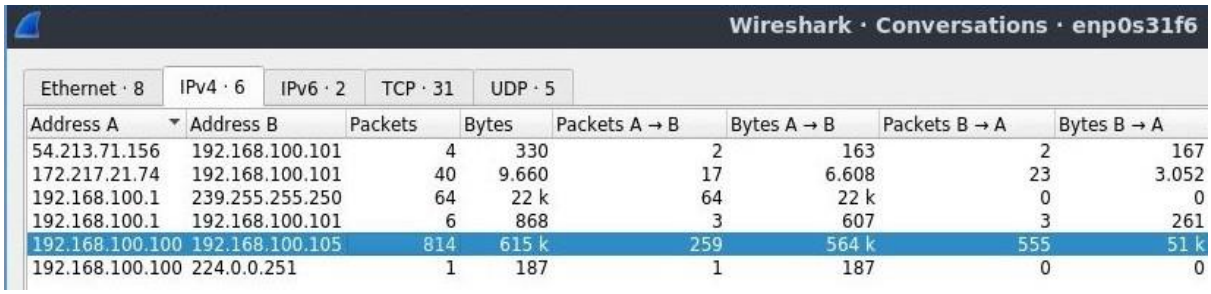


Figura 46

Conclusioni e sviluppi futuri

Il sistema sviluppato si è dimostrato una valida alternativa alle soluzioni già presenti in commercio. Esso offre ottime prestazioni: ottimizza il traffico dati necessario per gli aggiornamenti, riduce notevolmente lo spazio necessario per il salvataggio delle varie versioni e garantisce affidabilità e sicurezza, tuttavia manca ancora il supporto per la crittografia degli aggiornamenti. Inoltre occorre effettuare un'operazione di porting su sistema proprietario ART e risolvere tutti i vari problemi di compatibilità.

Modifiche da apportare al sistema

Affinché il sistema sia "production ready" occorre apportare alcune migliorie lato dispositivo target e lato server: la Figura 47 mostra in rosso le componenti software che sarebbe opportuno integrare al sistema attuale. In particolare:

- Migliorare l'interazione tra casync e RAUC rendendo più trasparente la comunicazione tra le due componenti.
- Creare una componente che gestisca gli aggiornamenti all'interno del dispositivo target. Questa deve contenere la logica con la quale il dispositivo dialoga con il server. Per fare ciò è necessario o creare un nuovo componente che comunichi con il client Rauc-Hawkbit o sostituirlo completamente, integrando le funzionalità.
- Creare un software che si interfacci con Hawkbit e che permetta l'automatizzazione dei rollout evitando di accedere all'interfaccia grafica.

Considerazioni sulla stabilità di casync

Tra i software utilizzati, casync è il più acerbo: ancora non è stata rilasciata una versione stabile e alcune delle sue funzionalità, come per esempio la possibilità di calcolare la dimensione necessaria per passare da una versione software ad un'altra, sono in fase di sviluppo. Nonostante non si sono mai verificati problemi durante i numerosi test, in uno sviluppo futuro del sistema è auspicabile contribuire a casync per il rilascio di una versione stabile e completa.

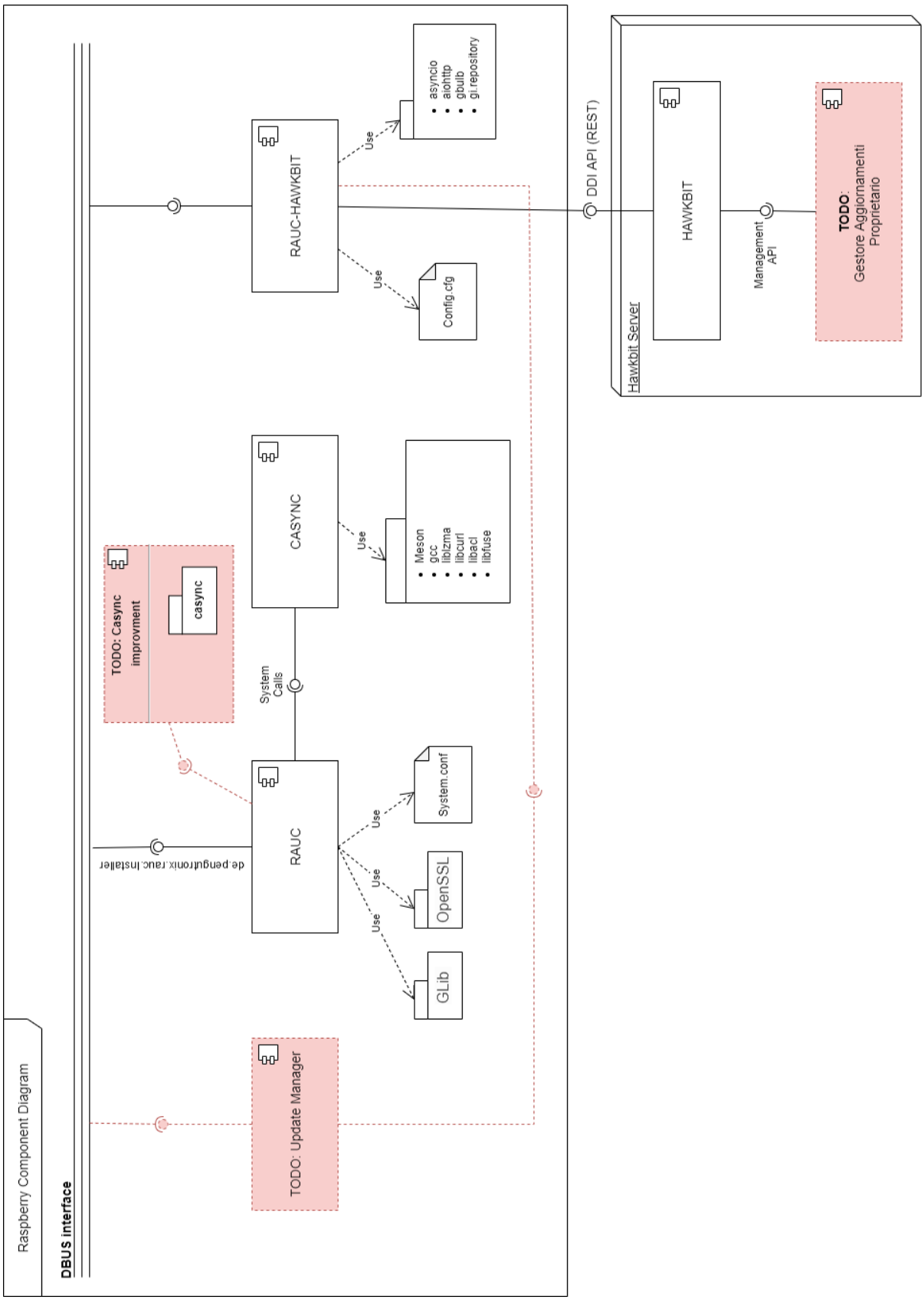

Raspberry Component Diagram
DBUS interface
RAUC-HAWKBIT
asyncio
aiohttp
gbulb
gi.repository
Use
Config.cfg
DDI API (REST)
Hawkbit Server
HAWKBIT
Management API
TODO:
Gestore Aggiornamenti
Proprietario
CASYNC
Meson
gcc
liblzma
libcurl
libacl
libfuse
TODO: Casync improvment
casync
System Calls
de.pengutronix.rauc.installer
RAUC
System.conf
OpenSSL
GLib
TODO: Update Manager

Figura 48

Riferimenti bibliografici

- https://www.yoctoproject.org/
- https://rauc.readthedocs.io/en/latest/index.html
- http://0pointer.net/blog/casync-a-tool-for-distributing-file-system-images.html
- https://github.com/systemd/casync
- https://www.docker.com/
- https://github.com/rauc/rauc-hawkbit
- https://www.eclipse.org/hawkbit/
- https://www.wireshark.org/
- https://docs.python.org/3/
- https://www.openssl.org/docs/
- https://pki-tutorial.readthedocs.io/en/latest/
- https://www.denx.de/wiki/U-Boot

Ringraziamenti

Ringrazio il Prof. Emilio Di Giacomo, l'Ing. Andrea Narciso e tutto il personale del settore Innovazione dell'azienda ART S.p.a per avermi permesso di produrre questa tesi di laurea e per avermi guidato durante tutto il percorso.

Ringrazio la mia famiglia, i miei amici e la mia ragazza per avermi sempre sostenuto e per essermi sempre stati vicini.