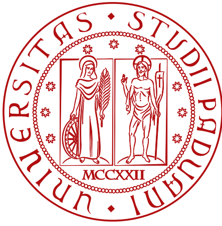
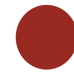


Università degli Studi di Padova



Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Sviluppo di software di Direct Device Integration per dispositivi IoT Android nell'ambito del progetto Eclipse hawkBit

Laureando

Alberto Battiston

Matricola n. 1221821

Relatore

Prof. Carlo Fantozzi

Università di Padova

Anno Accademico
2021/2022

Sommario

Eclipse hawkBit è un server che permette a dispositivi embedded di ricevere ed installare aggiornamenti software. Nell'implementazione presa in esame, l'interazione client-server viene realizzata da due progetti open source. La libreria Kotlin hara-ddiclient, facente parte del progetto Eclipse Hara, facilita lo sviluppo di client che si interfacciano a server hawkBit tramite le Direct Device Integration API. UF Android Client rappresenta l'applicativo lato client che, utilizzando la libreria hara-ddiclient, si occupa di gestire la connessione al server e le operazioni di aggiornamento nei dispositivi Android. Le pratiche di integrazione continua automatizzano le operazioni di verifica della funzionalità del codice e consentono di aumentare l'efficienza del processo di sviluppo software e il livello di robustezza del codice. Su queste motivazioni si basano le attività svolte, in particolare lo sviluppo di workflow, tramite GitHub Actions. Il processo di automazione comprende le operazioni di build e test per le tre piattaforme principali (Linux, macOS e Windows) e di produzione della documentazione relativa ai progetti. Il feedback degli utenti dell'applicazione Android ha inoltre permesso di identificare un caso d'uso che ha portato allo sviluppo di una nuova funzionalità. La funzionalità, sviluppata in Kotlin, affronta il problema delle diverse time zone associate ai dispositivi distribuiti in tutto il mondo. Il caso d'uso richiede che l'utente possa specificare in quali finestre temporali è possibile l'installazione di aggiornamenti, in modo da far sì, ad esempio, che i dispositivi siano aggiornati solamente negli orari notturni, in generale non lavorativi.

Indice

1 Introduzione **1**
1.1 Kynetics srl . . . 1
1.2 Il progetto Eclipse hawkBit . . . 2
1.3 La piattaforma Update Factory . . . 3
1.3.1 Interfaccia web . . . 4
1.4 Obiettivi del tirocinio . . . 5

2 Concetti e strumenti **6**
2.1 Integrazione continua . . . 6
2.2 Git . . . 7
2.2.1 Concetti chiave . . . 7
2.3 Github . . . 9
2.3.1 Github Actions . . . 9
2.4 Gradle . . . 10
2.4.1 Aspetti tecnici . . . 11
2.5 Kotlin coroutine . . . 12
2.6 Container e Docker . . . 12

3 La libreria Hara-ddiclient **14**
3.1 Panoramica . . . 14
3.2 Struttura . . . 15
3.2.1 Architettura: il modello ad attori . . . 16
3.2.2 Implementazione degli attori . . . 18
3.3 Il processo di build . . . 20
3.4 Sviluppo: pipeline per build e test . . . 20
3.4.1 Analisi del codice . . . 21
3.4.2 Commenti . . . 22
3.5 Sviluppo: pipeline per la documentazione . . . 23

3.5.1 Analisi del codice 24
3.5.2 Commenti 25

4 Update Factory Android Client **26**
4.1 Panoramica 26
4.1.1 Accesso alle API di sistema 27
4.2 Aggiornamenti supportati 27
4.3 Third-party integration 28
4.3.1 Aspetti tecnici 28
4.4 Il processo di build 32
4.5 Sviluppo: pipeline per build e test 34
4.5.1 Analisi del codice 35
4.5.2 Commenti 35
4.6 Sviluppo: schedulare gli aggiornamenti 36
4.6.1 Punti di intervento 36
4.6.2 Autorizzare l'aggiornamento 37
4.6.3 Salvataggio delle preferenze 40

5 Conclusioni **42**

Bibliografia **43**

Introduzione

In questo capitolo viene presentata Kynetics srl, azienda ospitante, e il progetto su cui si basano le attività svolte durante il tirocinio.

1.1 Kynetics srl

Kynetics [1] nasce nel 2006 con uffici a Padova e Santa Clara (California) e ad oggi si occupa di sviluppo software per i più diffusi processori per sistemi embedded, focalizzandosi in particolare sui SoC NXP. In collaborazione con produttori di hardware di livello mondiale, come Boundary Devices, Toradex e TechNexion, Kynetics lavora allo sviluppo di applicazioni e personalizzazione di sistemi operativi Android o Linux, distribuiti in settori come la sanità, l'ospitalità, il consumo, le scienze di laboratorio e la sicurezza. La qualità dei prodotti è garantita da un processo di integrazione continua che va dalla gestione del codice sorgente, alla compilazione e alla distribuzione sui dispositivi di destinazione. Kynetics, sfruttando un solido background nello sviluppo back-end, fornisce ai suoi clienti, distribuiti in tutto il mondo, un supporto completo nella pipeline di sviluppo dei prodotti nello spazio IoT.

Kynetics è partner di Eclipse Foundation e collabora allo sviluppo del progetto Elicpse hawkBit.

1.2 Il progetto Eclipse hawkBit

L'aggiornamento di componenti software su una rete più o meno complessa di dispositivi, in grado di garantire un alto livello di efficienza e sicurezza, è un requisito comune nello scenario IoT. Esistono tecnologie che offrono servizi di distribuzione di aggiornamenti, implementati direttamente nella rete di dispositivi. Tuttavia, tale approccio espone, spesso, punti di debolezza che impattano sull'organizzazione della distribuzione, in termini di efficienza. Inoltre questi servizi di "device management system" introducono un livello di complessità elevato e non necessario.

Elicpse hawkBit [2] è un framework back-end indipendente dal dominio per il roll-out degli aggiornamenti software ai dispositivi. HawkBit si pone l'obiettivo di snellire il processo di aggiornamento nello spazio IoT, aumentare l'efficienza di distribuzione per gruppi di dispositivi, offrendo funzionalità di report e monitoraggio dei processi in esecuzione, anche attraverso moderne interfacce web. I dispositivi possono connettersi direttamente al server hawkBit oppure a proprie istanze riconosciute, attraverso gateway connessi ad un'infrastruttura di rete basata sul protocollo IP. I server supportano aggiornamenti di software applicativo, sistemi operativi, tipicamente Linux, e firmware per sistemi embedded.

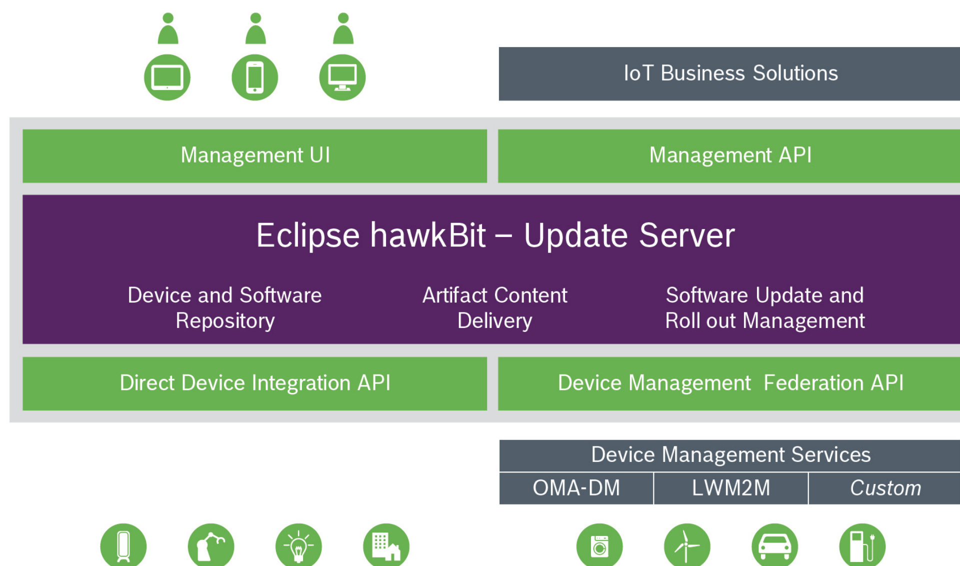


Figura 1.1: Architettura di Eclipse hawkBit [3]

L'interazione con i dispositivi è realizzata mediante due servizi di connessione esposti da Eclipse hawkBit: le Direct Device Integration API, basate sullo standard HTTP e risorse di tipo REST, e le Device Management Federation API, basate sul formato JSON. Gli utenti possono interfacciarsi con il server attraverso le Management API, utilizzando anche l'interfaccia grafica Management UI, per distribuire gli aggiornamenti ai dispositivi, monitorandone lo stato.

1.3 La piattaforma Update Factory

Update Factory [4] costituisce un'istanza di Eclipse hawkBit, sviluppata interamente da Kynetics. Update Factory è una piattaforma al servizio di ecosistemi IoT per la distribuzione di aggiornamenti software: permette di assegnare aggiornamenti a uno o più dispositivi contemporaneamente, tenendo traccia del progresso dell'aggiornamento e monitorando, istante per istante, lo stato di ciascun dispositivo. L'aggiornamento software e la gestione dei dispositivi fanno parte dello stesso processo. Per questo motivo Update Factory include funzionalità che consentono ai dispositivi di comunicare le informazioni necessarie ad organizzare le distribuzioni di aggiornamenti a gruppi di dispositivi. Al giorno d'oggi esistono diverse piattaforme che si occupano della distribuzione di aggiornamenti software, tuttavia per lo più focalizzate su microcontrollori in larga scala. Update Factory, invece, rappresenta una piattaforma neutrale per la distribuzione software a dispositivi embedded con sistemi operativi Android o Linux. Kynetics fornisce un'implementazione open source per client Android (vedasi cap.4), la quale sfrutta le Direct Device Integration API per la connessione ad Update Factory.

Dal punto di vista dell'utente, Update Factory si struttura in due componenti principali.

- Management UI: l'interfaccia web che consente l'interazione con l'utente.
- Update Server: il backend che consente ai dispositivi l'interazione con il server, rappresentato da Android Client o Linux Client.

1.3.1 Interfaccia web

Management UI è l'interfaccia web che consente all'utente di gestire la distribuzione degli aggiornamenti, visualizzando lo storico delle azioni applicate a ciascun dispositivo, e organizzare i dispositivi connessi alla piattaforma, creando dei gruppi che aderiscono a particolari filtri specificati attraverso l'apposita sezione. Tra le varie sezioni di cui è composta l'interfaccia, degna di nota è la sezione *Deployment*, la quale consente di visualizzare la lista di dispositivo (*Targets*) e la lista di artefatti (*Distributions*). Per assegnare un aggiornamento è sufficiente trascinare l'artefatto desiderato sopra il nome del dispositivo da aggiornare, specificando, tramite l'apposita finestra di dialogo, come dovrà essere applicato l'aggiornamento al dispositivo.

- **Forced**: la distribuzione viene applicata immediatamente, appena il dispositivo effettua il poll al server.
- **Soft**: l'utente deve autorizzare l'installazione dell'aggiornamento.
- **Time Forced**: la distribuzione viene applicata al dispositivo in una certa data e ora.

Abilitando l'opzione "Use maintenance window" l'utente può applicare ciclicamente una distribuzione.

Nella sezione *System Configuration* è possibile configurare l'interazione tra i dispositivi e il server. In particolare, la piattaforma implementa un particolare sistema di sicurezza, il quale prevede che i dispositivi, per poter comunicare con il server, debbano fornire un HTTP-Authorization header associando uno tra i seguenti tipi di token.

- **Target token**: ciascun dispositivo possiede il suo target token privato, che dev'essere specificato in anticipo.
- **Gateway token**: tutti i dispositivi appartenenti alla stessa rete (ossia allo stesso utente) vengono riconosciuti tramite lo stesso gateway token.

In questa sezione è possibile, inoltre, specificare l'intervallo di polling, cioè il processo con cui i client contattano ciclicamente il server per richiedere eventuali azioni da applicare al dispositivo.

1.4 Obiettivi del tirocinio

Le attività concordate riguardano la libreria Hara-ddiclient, facente parte del progetto Hara di Eclipse Foundation [2], e l'applicazione UF Android Client (vedasi cap.4). Hara-ddiclient è una libreria che facilita lo sviluppo di client permettendo a dispositivi di connettersi ad Update Factory. UF Android Client è un applicazione Android in grado di connettersi ad Update Factory e ricevere gli aggiornamenti per dispositivi con sistema operativo Android, grazie all'utilizzo della libreria Hara-ddiclient. Le attività svolte consistono interamente nello sviluppo software, centrato sull'integrazione continua (vedasi cap.2) e sulla realizzazione di una nuova funzionalità. Nella prima parte del tirocinio, in seguito ad una fase di studio per familiarizzare con gli strumenti di sviluppo, sono stati sviluppati gli script aderenti ai principi dell'integrazione continua per le pipeline di build e test e, solo nell'ambito di Hara-ddiclient, per la produzione della documentazione. Nella seconda parte, invece, l'attività è stata focalizzata sulla progettazione e sullo sviluppo di una funzionalità che estende il comportamento di UF Android Client. Nonostante la maggior parte dello sviluppo della funzionalità abbia riguardato UF Android Client, è stato necessario apportare delle modifiche anche alla libreria Hara-ddiclient, vista la forte interazione tra le parti.

Concetti e strumenti

Questo capitolo fornisce una panoramica riguardo le tecnologie e gli strumenti software utilizzati durante l'attività di tirocinio.

2.1 Integrazione continua

L'integrazione continua [5] è una pratica che si applica nel contesto dello sviluppo software collaborativo. L'ambiente condiviso, chiamato *mainline*, viene sincronizzato frequentemente con gli ambienti di lavoro dei singoli sviluppatori. L'allineamento avviene grazie ad una serie di test automatici eseguiti prima del rilascio del contributo degli sviluppatori verso l'ambiente condiviso. Lo scopo dei test è quello di garantire che le modifiche non introducano errori nel software già esistente. Un requisito fondamentale, ma non necessario, è la presenza di procedure predefinite di build e test per il software condiviso, le quali facilitano le pratiche di integrazione continua. Si riportano di seguito alcuni dei princìpi su cui si basa l'integrazione continua.

Automatizza il build Il processo di build converte il codice sorgente in eseguibile. Tale processo deve poter essere eseguito automaticamente, senza necessità dell'intervento umano.

Ogni commit fa partire una build Per evitare di introdurre ogni minimo errore, è necessario eseguire il build ad ogni modifica del codice sorgente. In

questo modo si può intervenire immediatamente e prevenire la generazione a catena di errori.

Eseguire i test in un clone dell'ambiente di produzione È fondamentale eseguire i test nello stesso ambiente di produzione, assicurando che eventuali errori siano scoperti prima della fase di produzione.

Ognuno può vedere i risultati dell'ultimo build In un prodotto software collaborativo è importante garantire uniformità nel codice. La trasparenza dei risultati di ciascun build stabilisce il livello di qualità del software, evidenziando i moduli che richiedono intervento.

2.2 Git

Git è un software libero e open source utilizzato per il controllo di versione distribuito. Il suo scopo è tenere traccia delle modifiche apportate a file di qualsiasi natura. Tuttavia, è utilizzato principalmente per coordinare lo sviluppo software collaborativo tra programmatori. Tra i suoi principi vi sono l'efficienza e la velocità di utilizzo, l'integrità dei dati e il supporto per flussi di lavoro distribuiti e non lineari, ovvero con la possibilità di gestire numerosi branch, eseguibili in diversi sistemi. Git viene ideato e realizzato da Linus Torvalds nel 2005 per lo sviluppo del kernel Linux: inizialmente dotato di funzionalità basilari, negli anni si evolve fino a diventare un software completo, efficiente e indipendente per il controllo di versione.

2.2.1 Concetti chiave

Snapshot Git tiene traccia dell'evoluzione del codice tramite gli snapshot, ovvero la registrazione delle modifiche apportate ad uno o più file in un determinato istante temporale. Spetta all'utente decidere quando catturare e quali file associare ad uno snapshot. In questo modo è anche possibile navigare nella storia del codice, visitando gli snapshots catturati

Commit L'atto di catturare uno snapshot si chiama Commit. Essenzialmente un progetto è composto da un insieme di commit. Un commit consente di ricavare tre importanti informazioni:

- quali modifiche sono state apportate ai file in esame
- un riferimento al commit precedente, chiamato anche parent commit
- l'hash code, codice univoco che identifica il commit

Repository All'interno del Git repository viene realizzata la storia del progetto: tiene traccia di tutti i file e le relative versioni, ovvero registra tutti i commit effettuati. Tale repository può risiedere nella macchina locale o in un server remoto (GitHub). È possibile copiare un repository da un server remoto, tramite l'operazione di "cloning", realizzando quindi l'idea della collaborazione. Inoltre è possibile sincronizzare un repository remoto con un repository locale attraverso le operazioni di "pulling", scaricando i commit che non esistono nella macchina locale, e "pushing", caricando nel repository remoto le modifiche locali.

Branch In Git un branch è un puntatore ad un insieme di commit. Un repository può avere più branch. Il branch di default è chiamato master branch. Il puntatore master branch è associato sempre all'ultimo commit realizzato.

Tag Come molti software di controllo versione, Git offre la possibilità di aggiungere una tag in uno specifico punto della storia del progetto, che solitamente identifica una nuova versione dello stesso. Le tag vengono aggiunte tramite operazioni di 'push', come avviene con i commit.

Pull request Il concetto di branch consente di isolare un processo di lavoro, come lo sviluppo di feature, bug fix o esperimenti che potrebbero compromettere il corretto funzionamento del prodotto finale. Ciascun repository possiede un branch di default ma può possedere più branch. L'operazione di integrazione di un branch in un altro branch viene chiamata "merging" e, nel caso di progetti collaborativi, può essere richiesto il merge di un branch tramite una "pull request". Una pull request permette di visualizzare un riassunto delle modifiche apportate al proprio branch rispetto al branch di base del repository. Ogni nuovo commit aggiunto a questo branch verrà integrato automaticamente alla pull request. I collaboratori del progetto possono valutare le modifiche, aggiungere commenti e, infine, decidere se unire la pull request al branch base.

2.3 Github

GitHub è un servizio di hosting per lo sviluppo software e il controllo di versione. Importanti funzionalità messe a disposizione sono quelle di controllo di versione distribuita di Git, bug tracking, task management e integrazione continua. Comunemente è utilizzato per ospitare software di natura open source, contando ad oggi oltre 83 milioni di sviluppatori e più di 200 milioni di repositories, rappresentando il più grande servizio di software hosting al mondo.

I progetti sono accessibili e manutenibili mediante l'interfaccia commandline di Git. Qualsiasi comando standard Git è compatibile con GitHub. È possibile navigare i repository pubblici attraverso l'interfaccia grafica del sito web. Chiunque può navigare e scaricare repository pubblici ma solo gli utenti registrati possono contribuire ai contenuti degli stessi. Con un account utente registrato è possibile interagire con discussioni, gestire repository e contribuire allo sviluppo richiedendo review delle modifiche apportate.

Documentazione In ciascun repository viene incluso il README: è un file, renderizzato secondo il formato Markdown, contenente informazioni più o meno dettagliate rigurardanti il progetto.

GitHub Actions GitHub Actions è un insieme di strumenti per lo sviluppo di pipeline per testing, releasing e deploying del software, senza necessità di strumenti esterni.

GitHub Pages GitHub Pages è un servizio di static web hosting utilizzato dagli utenti GitHub per lo sviluppo di blog o documentazione relativa al progetto. Il contenuto è salvato nel repository Git e, grazie all'integrazione con il generatore web statico Jekyll, ogni volta che il codice sorgente viene modificato, la documentazione o il sito web dedicato viene aggiornato di conseguenza. Spesso questa automazione viene realizzata tramite apposite GitHub Actions.

2.3.1 Github Actions

GitHub Actions [6] è una piattaforma di continuous integration (CI) che permette di automatizzare le operazioni di build, test e deployment. Offre la

possibilità di realizzare workflow che testano ogni pull request su uno specifico repository. GitHub fornisce macchine virtuali Linux, Windows, e macOS in cui eseguire i workflow.

Componenti

Il meccanismo di configurazione di un workflow permette di specificare quale evento associato al repository deve scatenarne l'esecuzione. Esempi di eventi che possono attivare un workflow sono l'apertura di una pull request, il push di un commit oppure l'apertura di una issue. Ciascun workflow contiene uno o più job. Ogni job è eseguito all'interno di uno specifico runner e consiste in uno o più step.

Workflow Un workflow è un processo automatizzato configurabile che esegue uno o più job. I workflow sono definiti da un file YAML, eseguibile manualmente, ad un istante programmato oppure in seguito al verificarsi di un determinato evento. Un repository può contenere diversi workflow, ognuno dei quali può svolgere operazioni diverse, dipendenti o indipendenti tra di loro.

Job Un job è un insieme di step in un workflow, eseguiti nello stesso runner. Ciascuno step può essere uno shell script che sarà eseguito oppure un'ulteriore action che sarà eseguita. Gli step sono eseguiti in ordine e sono dipendenti tra di loro. È possibile definire delle dipendenze tra job diversi, dal momento che, di default, i job sono tutti indipendenti tra di loro ed eseguiti in parallelo.

Runner Un runner è un server nel quale viene eseguito un workflow quando si verifica l'evento scatenante. Ciascun runner può eseguire un singolo job per volta. GitHub fornisce runner Ubuntu Linux, Microsoft Windows e macOS con predeterminate caratteristiche e software pre-installati.

2.4 Gradle

Gradle [7] è un tool open source utilizzato per automatizzare il processo di build di un'applicazione, rendendo consistenti le operazioni di compilazione, linking e packaging del codice sorgente. L'utilizzo più comune di Gradle si verifica nei progetti Android, in particolare nella costruzione di un APK a partire da

file Java e XML. In generale l'utilizzo di Gradle è vantaggioso nella compilazione di progetti software di grandi dimensioni. Gradle conserva le funzionalità di base che caratterizzano i suoi predecessori Apache Ant e Apache Maven[1], contenendo allo stesso tempo i limiti di entrambi.

Gradle risolve i limiti funzionali dei suoi predecessori, focalizzandosi sulla manutenibilità, usabilità e, soprattutto, sulle performance di compilazione. È noto per essere altamente personalizzabile quando si tratta di integrare progetti diversi che hanno a che fare con altrettanto differenti tecnologie. L'utilizzo di Gradle, inoltre, è facilitato dal supporto di plugin da parte di molti IDE. Gradle opera attraverso dei file situati nella root del progetto. All'interno di questi file è possibile definire dei *task*, ovvero delle unità di computazione contenenti le operazioni che Gradle deve eseguire, ad esempio compilare il codice sorgente. Questi task possono essere creati dinamicamente ed estesi a tempo di esecuzione.

2.4.1 Aspetti tecnici

Gradle offre dei vantaggi rispetto ad Apache Maven, quali ad esempio la possibilità di definire il meccanismo di build in linguaggio Groovy e di modificare il comportamento predefinito di alcune attività. Il processo di build in Gradle si suddivide in due fasi principali: la configurazione e l'esecuzione. Nella fase di configurazione, vengono delineati i passi dell'intero processo per generare il grafico delle dipendenze (DAG), contenente la sequenza di tutte le operazioni da eseguire. La seconda fase consiste nell'attraversamento del grafico precedentemente generato, eseguendo nell'ordine le operazioni necessarie. Entrambe le fasi avvengono per mezzo di istruzioni accessibili all'utente attraverso un'interfaccia di programmazione documentata. Il file che contiene tali istruzioni si chiama `build.gradle`, situato all'interno della directory del progetto. Altri file opzionali sono il `settings.gradle`, contenente la definizione dei moduli software di cui si compone il progetto, e il `gradle.properties`, un elenco di valori per l'inizializzazione delle proprietà di uno specifico progetto Gradle.

[1]Strumenti di gestione di progetti software basati su Java e build automation

2.5 Kotlin coroutine

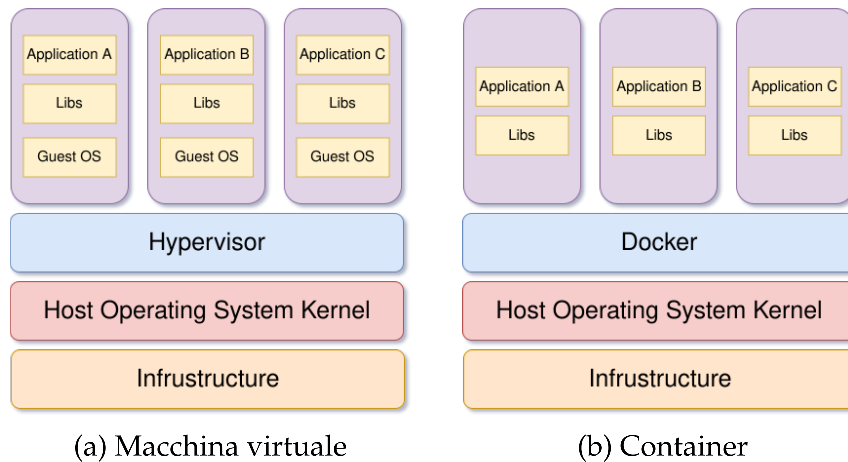
Nel linguaggio Kotlin una *coroutine* [8] rappresenta un'istanza di una computazione, la quale può essere sospesa. Una coroutine è concettualmente simile ad un thread, nel senso che è possibile assegnare un blocco di codice che verrà eseguito in maniera concorrente al resto del codice. Una coroutine non è associata in modo imprescindibile ad un thread, infatti, la sua esecuzione può essere interrotta in uno e ripresa in un altro thread. Le coroutine seguono un principio di concorrenza strutturata, il che significa che ciascuna coroutine può essere lanciata all'interno di un proprio *Coroutine Scope*. Il Coroutine Scope è generalmente utilizzato per la gestione del ciclo di vita delle coroutine e delle loro coroutine figlie, fornendo ogni informazione sul risultato delle esecuzioni, interrompendole in caso di errore. Le coroutine sono estremamente efficienti e risultano decisamente più vantaggiose rispetto ai thread, in quanto richiedono un numero di risorse molto più contenuto. Due o più coroutine possono comunicare tra di loro attraverso i *channel*.

2.6 Container e Docker

I *container* [9] rappresentano uno strumento che consente di raggruppare il software insieme a tutte le sue dipendenze in un pacchetto autonomo, in modo da poterlo eseguire evitando un fastidioso processo di configurazione, per rendere consistenti tra di loro diverse piattaforme. In particolare, è possibile sviluppare ed eseguire le applicazioni in un ambiente sicuro ed isolato, chiamato *container*, configurato con tutte le caratteristiche necessarie. L'applicazione è inserita all'interno di un singolo file, chiamato *immagine*, contenente anche le dipendenze e le informazioni necessarie alla distribuzione. L'immagine è condivisa con un server, chiamato *registro*, accessibile attraverso specifiche autorizzazioni. Questa tecnologia, quindi, permette la condivisione di applicazioni la cui esecuzione è riproducbile attraverso diverse piattaforme.

Docker è un'implementazione open-source della containerizzazione. Per comprenderne meglio il significato, possiamo considerare i container come la nuova generazione delle macchine virtuali. Infatti i container sono un ambiente totalmente isolato dall'host in cui risiedono, anche se richiedono un numero deci-

samente ridotto di risorse hardware rispetto alle tradizionali macchine virtuali. La principale differenza sta nel metodo di virtualizzazione.



(a) Macchina virtuale (b) Container

Figura 2.1: Meccanismi di virtualizzazione [9]

Un'applicazione eseguita all'interno di una macchina virtuale comunica mediante il sistema operativo ospite con l'hypervisor, il quale a sua volta comunica con il sistema operativo ospitante per allocare le risorse necessarie per l'esecuzione. Questo processo risulta inefficiente sia in termini di tempo che di risorse sprecate.

Il container, invece, realizza la virtualizzazione sfruttando direttamente il sistema operativo host, al posto di utilizzare un ulteriore sistema operativo all'interno di esso, come nel caso di una tradizionale macchina virtuale. Il ruolo dell'hypervisor è interpretato da un'istanza di un container, in questo caso Docker, il quale si occupa di richiedere le risorse necessarie al funzionamento della struttura, utilizzando il kernel del sistema operativo host. Il vantaggio di eliminare il livello del sistema operativo ospite sta nella quantità di risorse sprecate. Docker utilizza un'architettura client-server. Il *client* Docker comunica con il Docker *daemon*, il quale si occupa di eseguire e distribuire i containers. Sia il client che il daemon possono risiedere nello stesso sistema, ma è anche possibile la comunicazione con un daemon remoto. La comunicazione avviene usando REST API, mediante socket o interfacce di rete UNIX. Un altro tipo di client è *Docker Compose*, il quale permette di operare con applicazioni che fanno uso di molteplici containers.

La libreria Hara-ddiclient

In questo capitolo viene analizzata, dal punto di vista del codice, la struttura di Hara-ddiclient, una libreria che consente lo sviluppo di client, i quali, attraverso le DDI API, riescono a comunicare con il server hawkBit oppure Update Factory. Inoltre vengono descritte le attività di sviluppo durante il periodo in azienda, riguardanti l'integrazione continua.

3.1 Panoramica

Il termine *client* indica una componente software che accede alle risorse o ai servizi erogati da un'altra entità chiamata *server*, mediante uno o più protocolli di rete. Hara-ddiclient [10] è una libreria open source scritta in Kotlin che facilita e velocizza lo sviluppo di client che sfruttano le Direct Device Integration API per la connessione e la gestione dei dati inviati dal server hawkBit. L'update server hawkBit mette a disposizione risorse di tipo REST, utilizzate dai dispositivi per ricevere update task. Le API sono basate su un'architettura standard HTTP e sviluppano un meccanismo di polling, ovvero una comunicazione continua e regolare tra client e server che consente di verificare la presenza di informazioni utili per il client. Le DDI API consentono ai client di ricevere messaggi di feedback circa le azioni che il server sta compiendo su un determinato update task. Esempi di messaggi di feedback sono Canceled, Rejected, Downloaded, Resumed.

3.2 Struttura

La libreria è strutturata in tre moduli principali. Il ddi-consumer è un'implementazione di un client REST che fa uso delle DDI API di hawkBit per lo scambio dei dati. L'hara-ddiclient rappresenta il cuore pulsante della libreria: implementa la logica di comunicazione vera e propria tra client e server, gestendo gli update task con i relativi file da scaricare e gli aggiornamenti da applicare al dispositivo. Questa componente della libreria fa utilizzo del modello ad attori (vedi sez. 3.2.1), una tecnologia che permette di sfruttare al massimo i vantaggi della programmazione concorrente e rende il complesso software estremamente efficiente. Infine, il virtual-device costituisce un esempio di applicazione che fa uso della libreria. Il suo scopo è fornire un "dispositivo virtuale" configurabile e delle indicazioni su come utilizzare la libreria. Questa applicazione di esempio è in grado di connettersi al server e gestire i dati inviati dal server stesso, utilizzando il modulo hara-ddiclient. Il modulo virtual-device è anche utilizzato in fase di sviluppo, ad esempio per testare le funzionalità o per operazioni di debug. La Figura 3.1 riporta uno schema del meccanismo con cui Hara-ddiclient interagisce con il server hawkBit: il ddi-consumer funge da intermediario tra il client e il server, utilizzando le DDI API come mezzo di comunicazione con il server.

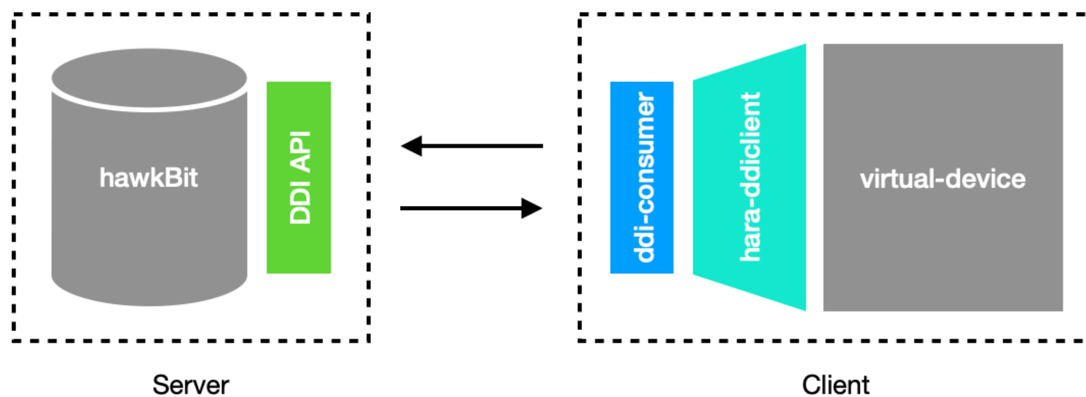


Figura 3.1: Architettura della libreria Hara-ddiclient

Lo sviluppo di client, che si adattano ai casi d'uso degli utenti, è possibile grazie alle interfacce Kotlin esposte dalla libreria. In particolare, l'implementazione di tali interfacce permette ai client di assumere comportamenti specifici,

come conseguenza ai dati inviati dal server. L'interfaccia *DownloadBehaviour*, ad esempio, specifica il comportamento del client durante la fase di download, ovvero le azioni da compiere prima, durante e dopo lo scaricamento degli artefatti. L'attività di sviluppo, descritta nel successivo capitolo, si focalizza sull'interfaccia *DeploymentPermitProvider*, la cui implementazione consente di personalizzare il meccanismo di autorizzazione allo scaricamento e applicazione di aggiornamenti.

3.2.1 Architettura: il modello ad attori

Il modello ad attori si discosta dal normale concetto di programmazione concorrente. Il classico modello di concorrenza si basa sulla condivisione della memoria da parte dei thread. La conseguenza è la necessità di sincronizzare i thread in modo tale da garantire la consistenza dei dati. Tuttavia, questo sistema espone vari punti di debolezza: deadlock, starvation, race-condition sono esempi classici di errori in cui è possibile incorrere al crescere della complessità logica dell'applicazione.

Un attore è un'entità fondamentale di computazione che incapsula al proprio interno uno stato, una coda di messaggi (mailbox) e un comportamento che può assumere in completa autonomia, in base al suo stato e indipendentemente dall'esterno.

Messaggi Le computazioni svolte da ciascun attore producono dei risultati che vengono incapsulati in pacchetti di informazione, chiamati messaggi, e inviati ad altri attori in attesa di essere processati. Una caratteristica importante è il fatto che un attore può inviare messaggi a se stesso.

Mailbox La mailbox rappresenta una coda di messaggi in attesa di essere processati. Normalmente ciascun attore possiede una mailbox. Per ogni messaggio presente nella mailbox, ciascun attore compie delle operazioni di computazione sulla base del contenuto del messaggio, cambiando il suo stato di conseguenza e inviando a sua volta altri messaggi ad altri attori.

Comportamento degli attori

Dopo aver analizzato l'interazione tra attori, è necessario descrivere il comportamento interno di ciascun attore. Come già detto ciascun attore effettua delle computazioni sulla base dei messaggi ricevuti, indipendentemente dall'esterno. Ciò significa che i contenuti dei messaggi possono far variare il risultato della computazione. Questa caratteristica spiega il significato del termine attore: assume dei comportamenti sulla base dei messaggi che riceve e quindi del suo stato, emulando le caratteristiche di un essere umano. Oltre ad inviare messaggi e cambiare il suo stato, un attore può compiere un'altra operazione fondamentale: la creazione di un nuovo attore. Tale caratteristica prevede la possibilità per ogni attore di distruggere ciascun attore figlio. I dettagli di questa operazione sono specifici per ogni linguaggio di programmazione che implementa il modello generale.

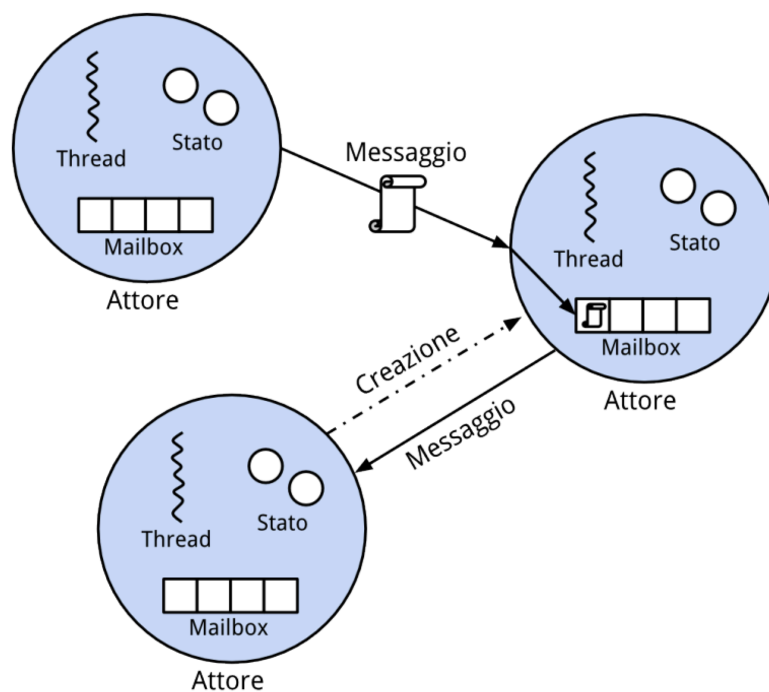


Figura 3.2: Invio di messaggi nel modello ad attori

Vantaggi del modello

Il modello ad attori è particolarmente adatto alla programmazione concorrente poichè permette di aggirare problemi classici che la caratterizzano. Il meccanismo di memoria condivisa tra i vari thread viene sostituito dal sistema di messaggistica tra gli attori e permette quindi di sviluppare software con un alto livello di parallelismo ma mantenendo un'alta efficienza e integrità dei dati.

3.2.2 Implementazione degli attori

In Hara-ddiclient il modello ad attori è implementato in Kotlin. In Kotlin un attore è un'entità formata da una combinazione di una coroutine, uno stato, incorporato all'interno della coroutine stessa, e un cosiddetto channel, utilizzato per lo scambio di messaggi. Un attore semplice può essere rappresentato con una singola funzione, mentre strutture più complesse necessitano di una classe dedicata. Nel caso in esame, ciascun attore estende `AbstractActor`, classe astratta che implementa la funzionalità di creazione di attori figli. Ciascun attore è figlio di un attore padre chiamato `RootActor`. Di seguito la struttura completa.

- **Connection Manager**: gestisce la connessione al server, riceve da Action Manager i messaggi di feedback sullo stato del client e li inoltra al server;
- **Action Manager**: avvia la procedura di aggiornamento da applicare al dispositivo. Invia messaggi di feedback a Connection Manager sulla base del risultato dell'update;
- **Deployment Manager**: gestisce il download, tramite Download Manager, e l'update, tramite Update Manager. Invia feedback ad Action Manager circa lo stato della procedura di aggiornamento;
- **Download Manager**: avvia la procedura di download dei file di update tramite File Downloader. Invia feedback sul risultato del download a Deployment Manager;
- **File Downloader**: effettua il download vero e proprio. Esiste un File Downloader per ogni file da scaricare. Il risultato del download viene comunicato a Download Manager;
- **Update Manager**: applica l'aggiornamento al dispositivo. Il risultato della procedura viene comunicato a Deployment Manager.

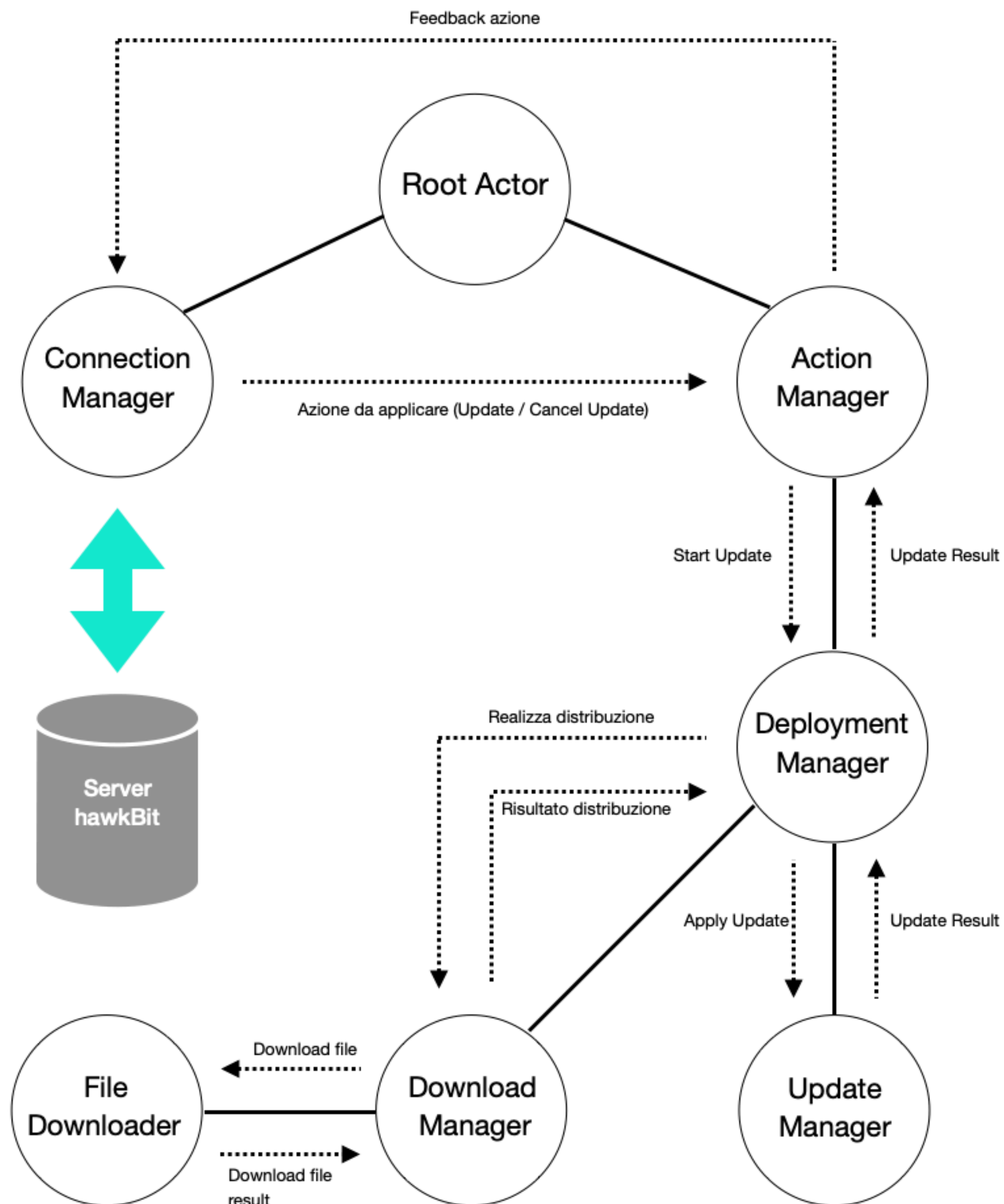


Figura 3.3: Struttura gerarchica degli attori in Hara-ddiclient

3.3 Il processo di build

Sfruttando le potenzialità del sistema Gradle, Hara-ddiclient possiede un meccanismo di build configurato per eseguire dei test automatici di corretta comunicazione tra client e server ad ogni operazione di build; in particolare viene verificato che i client riescano a scaricare e applicare gli aggiornamenti richiesti, comunicando messaggi di feedback al server. Questa procedura è realizzata da Docker Compose, utilizzando due immagini Docker:

- un'immagine di mariaDB, che carica un dump del database per configurare gli aggiornamenti da applicare;
- un'immagine di hawkBit Update Server, che contiene la directory con gli artefatti che i dispositivi scaricheranno durante i test. Questo servizio si basa sul database di mariaDB.

La comunicazione è implementata attraverso le Management API esposte da hawkBit Update Server. Il risultato del build dipende anche dal risultato di questi test.

3.4 Sviluppo: pipeline per build e test

La prima parte del tirocinio è stata dedicata allo sviluppo del workflow che implementa una pipeline per il build e i test della libreria, integrato nel repository GitHub. Lo script seguente contiene il codice completo del workflow: è necessario, prima di tutto, configurare il runner per la corretta esecuzione di Gradle e Docker; una volta configurato si procede al build vero e proprio.

```
1  name: build-action
2  on:
3      push:
4        branches-ignore:
5          - 'gh-pages'
6      pull_request:
7        branches-ignore:
8          - 'gh-pages'
9
10 jobs:
11     build:
12         strategy:
```

```
            matrix:
                os: [ubuntu-latest, macos-latest, windows-latest]
        runs-on: ${{matrix.os}}
        steps:
        - name: Check repository
          uses: actions/checkout@v3

        - name: Setup Gradle
          uses: gradle/gradle-build-action@v2

        #Run only on macOS
        - name: Setup Docker
          if: matrix.os == 'macos-latest'
          uses: docker-practice/actions-setup-docker@master

        #Build without test only on Windows
        - name: Gradle build
          run: |
              if [ "$RUNNER_OS" == "Windows" ]; then
                  ./gradlew build -x test
              else
                  ./gradlew build
              fi
          shell: bash
```

3.4.1 Analisi del codice

I workflow presentano dei costrutti principali che corrispondono agli elementi fondamentali della piattaforma GitHub Actions, ovvero gli eventi che causano l'esecuzione del workflow e i job contenenti la sequenza di step da eseguire.

name Il nome del workflow.

on Lista di eventi che scatenano l'esecuzione del workflow. In questo caso l'esecuzione avviene in seguito ad ogni 'push' di un commit e in seguito all'apertura di una 'pull request'. In entrambi i casi si è deciso di ignorare il caso in cui il push o la pull request avviene sul branch 'gh-pages', branch particolare utilizzato per la pubblicazione della documentazione.

jobs Lista di job da eseguire. In questo caso è presente un unico job chiamato 'build'. Il campo 'strategy' è utilizzato per definire delle variabili che vengono specificate nella sezione 'matrix'. Tali variabili indicano il nome dei runner sul quale verrà eseguito il workflow corrente. Nonostante sia specificato un unico job, questo verrà diviso a tempo di esecuzione in tre job separati, uno per ciascun runner specificato nel campo 'matrix'. L'esecuzione del workflow è considerata fallita quando almeno un job fallisce.

steps Sequenza di step da eseguire per ciascun job. Ciasuno step possiede un nome (facoltativo) e l'operazione che deve svolgere, specificata da 'uses'. In primo luogo avviene il checkout del repository, necessario per poter accedere ai file del progetto. Successivamente avviene la fase di setup di Gradle, durante la quale viene configurato l'ambiente per l'esecuzione del build e un meccanismo di cache per velocizzare le successive esecuzioni. Nel caso in cui il runner sia macOS avviene il setup di Docker, dal momento che non è presente nella lista dei software pre-installati. Infine viene eseguito il build con il comando `./gradlew build`, che comprende l'esecuzione dei test eseguiti con Docker Compose.

Figura 3.4: Alcuni risultati di esecuzione del workflow 'build-action'

3.4.2 Commenti

Lo sviluppo si è rivelato particolarmente ostico nella fase di integrazione tra Docker e i runner di macOS e Windows. Infatti Docker non risulta nativo per questi due sistemi operativi, come nel caso di Linux, e per questo motivo è necessario installarlo direttamente in fase di esecuzione del workflow. A causa delle licenze, l'unico modo per poter utilizzare Docker su questi due sistemi operativi è la versione Desktop, inutilizzabile con i runner di GitHub dal momento che l'interazione con l'utente avviene solamente tramite la shell. Questo problema è stato parzialmente risolto per macOS, grazie ad una action

che consente il setup di Docker mediante una particolare procedura. Visto l'eccessivo sforzo per arrivare ad una soluzione comunque lenta ed inefficiente, si è deciso di non eseguire i test che fanno uso di Docker nel runner Windows. Nonostante questi ostacoli il risultato desiderato, la soluzione si può considerare comunque valida.

3.5 Sviluppo: pipeline per la documentazione

Per una libreria software è fondamentale essere accompagnata da una documentazione sempre aggiornata all'ultima versione. La produzione della documentazione è spesso affidata a tool come, ad esempio, Dokka per Kotlin o javadoc per Java. Questi tool riescono a intepretare commenti con sintassi particolare, inseriti all'interno del codice, trasformandoli in documenti tipicamente in formato Markdown o HTML. Durante il tirocinio è stato sviluppato il seguente script che si occupa di generare la documentazione per la libreria: ad ogni push di una nuova tag viene richiamato il comando per generare la documentazione con il tool Dokka. Si è deciso di mantenere all'interno del repository la documentazione di tutte le versioni della libreria. L'ultima versione aggiornata è situata all'interno della directory 'latest'.

```
1  name: kdoc-action
2
3  #Triggered when a new tag is pushed
4  on:
5    push:
6      tags:
7        - '*'
8
9  jobs:
10   generate-doc:
11     runs-on: ubuntu-latest
12     steps:
13       - name: Check repository
14         uses: actions/checkout@v3
15
16       - name: Generate Html documentation
17         run: ./gradlew dokkaHtml
18
19       - name: Override index.html
```

```
20          run: |
21            echo '<html xmlns="http://www.w3.org/1999/xhtml">
22            <head>
23                <meta http-equiv="refresh" content="0;URL='${{ github.
    ref_name }}'" />
24            </head>
25            <body>
26            </body>
27            </html>' > 'index.html'
28        #Directory with index.html and documentation (only one commit)
29        - name: Setup publish directory
30          run: |
31            mkdir -p latest_doc/${{ github.ref_name }}
32            mv ./hara-ddiclient-api/build/dokka/html/* latest_doc/${{
    github.ref_name }}
33            mv index.html latest_doc
34        - name: Setup branch gh-pages
35          uses: peaceiris/actions-gh-pages@v3
36          with:
37            github_token: ${{ secrets.GITHUB_TOKEN }}
38            publish_dir: ./latest_doc
39            keep_files: true
```

3.5.1 Analisi del codice

name Il nome del workflow.

on Il workflow viene eseguito come conseguenza all'operazione di push di una nuova tag. La sintassi '*' specifica quali caratteristiche deve avere la stringa che costituisce la tag per essere considerata come evento scatenante.

jobs Ancora una volta viene eseguito un solo job, chiamato 'generate-doc'. In questo caso non è necessario effettuare alcun test multipiattaforma, dunque è sufficiente eseguire il workflow su un unico runner, in particolare ubuntu-linux. La sequenza di step inizia con il solito checkout del repository. Successivamente avviene la produzione della documentazione con il comando `./gradlew dokkaHtml` e l'aggiornamento del file 'index.html'. Il file 'index.html', situato all'interno del repository, contiene un riferimento alla documentazione generata ed è utilizzato dalla piattaforma GitHub Pages per la generazione del sito web

statico. Infine viene configurato il branch gh-pages che si occupa di aggiungere al repository la documentazione della versione appena prodotta. Di default questa operazione elimina tutti i file presenti, tuttavia l'opzione 'keep-files' consente di mantenere tutti i file già presenti, aggiornando la directory con i nuovi contenuti.

3.5.2 Commenti

Lo sviluppo è stato decisamente meno difficoltoso della pipeline per build e test. Non sono state incontrate notevoli criticità.

4 Update Factory Android Client

In questo capitolo viene presentato UF Android Client, un'applicazione Android che fa uso della libreria Hara-ddiclient per la distribuzione degli aggiornamenti. Le attività di sviluppo correlate riguardano la pipeline per build e test e l'implementazione di una funzionalità.

4.1 Panoramica

UF Android Client [11] è un'applicazione Android che, sfruttando le funzionalità della libreria Hara-ddiclient, si occupa di installare aggiornamenti software di tipo Applicazione (APK) o Sistema (OTA), ricevuti da server hawkBit o UpdateFactory, nel dispositivo in cui risiede. UFAndroidClient è un progetto open source sviluppato interamente da Kynetics. Esso è composto dai seguenti elementi.

UF Android Client Service Un APK che, una volta installato nei dispositivi Android, implementa un Android Service in background in grado di gestire tutte le comunicazioni da e verso la piattaforma Update Factory e controllare la procedura di installazione di file di aggiornamento.

UF third-party integration APIs Una libreria AAR che consente ad applicazioni terze di configurare e visualizzare le attività di UF Android Client Service.

UF Android Client UI example Un APK che permette, attraverso UF third-party integration APIs, di configurare, gestire e visualizzare UF Android Client Service, utilizzando una semplice interfaccia grafica di Android. Analogamente a virtual-device in Hara-ddiclient, questa componente rappresenta un esempio di applicazione per la configurazione di UF Android Client Service. In fase di sviluppo, UI example viene utilizzata per scopi di debug.

4.1.1 Accesso alle API di sistema

Il servizio UF Android Client Service richiede una serie di permessi di sistema, per avere accesso alle API che consentono di apportare modifiche al dispositivo, ad esempio installando gli APK forniti dall'aggiornamento. I permessi di sistema venogno forniti alle app che sono firmate con la chiave *platform* dell'OS Android. Questa chiave viene utilizzata durante il build del sistema operativo. La procedura di firma è descritta in maniera dettagliata nella sezione 4.4.

4.2 Aggiornamenti supportati

Android Client supporta due tipi di aggiornamenti: Application e System.

Application Il client installa (o aggiorna) applicazioni Android. Un modulo software di tipo applicazione contiene un Application type, ovvero un insieme di informazioni riguardanti l'aggiornamento (il nome, la versione, la descrizione,...) e una o più Android Application, ossia il file con estensione ".apk" da installare. L'installazione di un aggiornamento è considerata fallita quando almeno un aggiornamento di un applicazione fallisce oppure quando nessuno dei moduli software è di tipo APK. Gli aggiornamenti Application non richiedono il riavvio del sistema.

System Il client installa aggiornamenti del sistema operativo Android. Un modulo software di tipo system deve includere un OS Type, analogo ad Application Type, e un singolo file OTA. Esistono due tipi di aggiornamento System: *single-copy*, meccanismo utilizzato da versioni meno recenti di Android nelle quali la partizione di sistema non è duplicata, *double-copy*, utilizzato nelle ul-

time versioni di Android nelle quali le partizioni di sistema sono interamente duplicate. L'aggiornamento System richiede un riavvio del sistema.

4.3 Third-party integration

L'intero progetto UF Android Client si compone di tre moduli software: *uf-client-service*, *uf-client-ui-example* e *uf-ddiclient*. Il componente principale di UF Android Client è Android Service (uf-client-service), il quale però, operando in background, non fornisce un feedback diretto delle attività compiute al'utente del dispositivo. Con applicazioni third-party si intendono tutte le implementazioni analoghe a uf-client-ui-example che consentono di visualizzare, grazie ad un'interfaccia grafica, le attività in background del servizio. L'interazione tra le applicazioni terze e uf-client-service avviene grazie ad un insieme di API (uf-client-service-api) esposte direttamente da uf-client-service. Una volta integrate queste API, l'applicazione terza connessa al servizio sarà in grado,ad esempio, di acconsentire o meno all'applicazione di aggiornamenti di tipo soft (vedasi sez. 1.3.1), visualizzare lo stato del servizio oppure configurarne il comportamento specificando delle informazioni utili per la connessione al server, come il nome del dispositivo o l'URL del server.

4.3.1 Aspetti tecnici

Uf-android-service è un servizio Android di tipo *bound*[12]: le componenti delle applicazioni (ad esempio le activity) possono quindi associarsi, inviando richieste e ricevendo risposte, emulando il comportamento dell'architettura client-server. In particolare avviene un meccanismo di inter-process communication (IPC), realizzato da Android Messenger, il quale si occupa autonomamente di gestire lo scambio di informazioni in maniera efficiente attraverso oggetti Message. In questo punto intervengono le sopracitate API contenute nel modulo uf-client-service-api.

Il modulo uf-client-service-api contiene tutte le classi che l'applicazione terza deve includere per poter comunicare con il servizio. Le classi sono descritte di seguito.

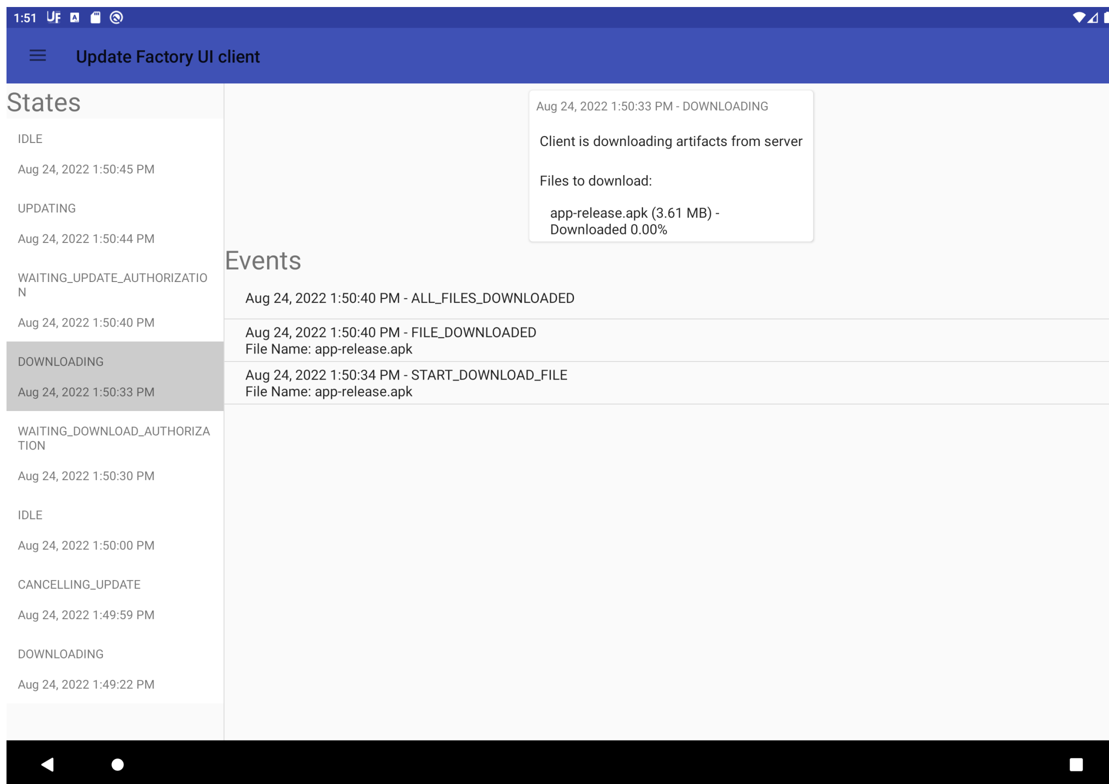


Figura 4.1: Esempio di comunicazione tra Android Client e Update Factory

UFServiceInfo Classe contenente le informazioni principali riguardanti il servizio.

UFServiceConfiguration Classe *data* contenente la configurazione corrente del servizio.

```
UFServiceConfiguration(
                tenant = <tenant_name>,
                controllerId = <controller_id>,
                url =  <uf_server_url>,
                targetToken = <target_token>,
                gatewayToken = <gateway_token>,
                isApiMode = true,
                isEnable = true,
                isUpdateFactoryServe = true,
                targetAttributes = mutableMapOf("DeviceOS" to "
   Android")
            )
```

- tenant: il nome utente per l'accesso alla piattaforma.
- controllerId: il nome identificatore del dispositivo.
- url: URL del server Update Factory.
- targetToken: token associato al singolo dispositivo per la comunicazione con il server (vedasi sez. 1.3.1).
- gatewayToken: token che autentica un gruppo di dispositivi per la comunicazione con il server (vedasi sez. 1.3.1).
- isApiMode: opzione per visualizzazione di finestre di dialogo per l'autorizzazione all'applicazione di aggiornamenti.
- targetAttributes: insieme di informazioni utili per il server riguardanti il dispositivo, come, ad esempio, la versione di Android installata.

Communication Insieme di classi che gestiscono i messaggi per la comunicazione con il servizio. In particolare i messaggi inviati al servizio vengono generati con la classe *In*, mentre i messaggi ricevuti dal servizio vengono interpretati con la classe *Out*. Entrambe le classi citate possiedono a loro volta delle sottoclassi per la gestione dei messaggi.

La sottoclassi della classe *In* sono:

- ConfigureService: messaggio per una nuova configurazione per il servizio
- RegisterClient: messaggio per registrare un nuovo dispositivo da associare al servizio
- UnregisterClient: messaggio per dissociare un dispositivo dal servizio
- AuthorizationResponse: messaggio che comunica l'autorizzazione o meno all'installazione di un aggiornamento nel dispositivo
- Sync: messaggio per richiedere una sincronizzazione con il servizio, che risponderà con la sua configurazione e il suo stato corrente.
- ForcePing: messaggio che comunica l'azione di poll del dispositivo al server

Un esempio di utilizzo di queste classi è riportato di seguito.

```
//mService:Messenger;
mService!!.send(RegisterClient(mMessenger).toMessage())
```

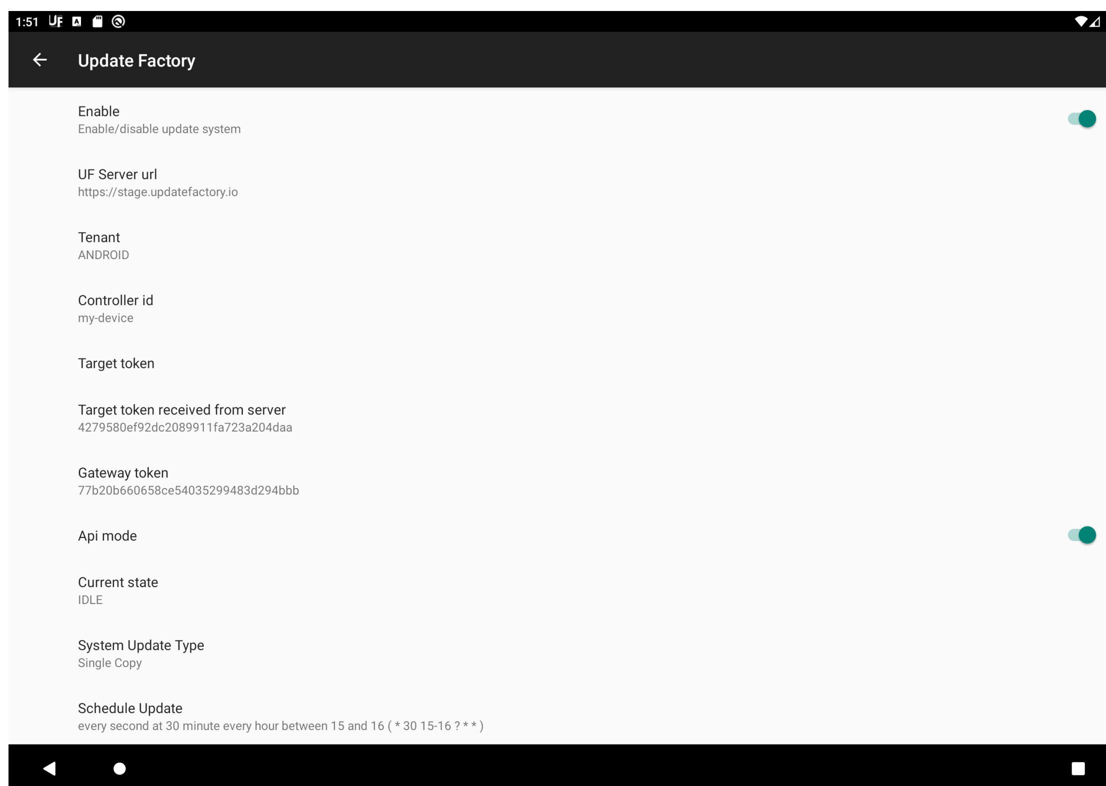


Figura 4.2: interfaccia per la configurazione di UF Android Service

Le sottoclassi della classe *Out* sono:

- ServiceStatus: contiene lo stato corrente del servizio
- AuthorizationRequest: contiene il tipo di autorizzazione (download oppure update)
- CurrentServiceConfiguration: contiene la configurazione corrente del servizio.

UFServiceMessageV1 Classe che mappa gli stati e gli eventi del servizio Update Factory.

Il client può assumere i seguenti stati.

- Downloading: il client sta scaricando artefatti dal server. Viene specificata la lista di artefatti nel corpo di un messaggio di Log.
- Updating: il processo di installazione dell'aggiornamento è cominciato. Non è possibile interromperlo.
- CancellingUpdate: L'ultima richiesta di aggiornamento è stata annullata.

- WaitingDownloadAuthorization: il client è in attesa di ricevere l'autorizzazione per procedere con il download.
- WaitingUpdateAuthorization: il client è in attesa di ricevere l'autorizzazione per procedere con l'installazione dell'aggiornamento.
- Idle: il client è in attesa di nuove richieste dal server.
- ConfigurationError: si è verificato un errore in fase di configurazione del servizio.

Il client può gestire i seguenti eventi.

- Polling: il client sta contattando il server per ottenere eventuali azioni da eseguire.
- StartDownloadFile: il download di un file è cominciato.
- FileDownloaded: il download di un file è terminato con successo.
- DownloadProgress: percentuale di file scaricato.
- AllFilesDownloaded: tutti i file necessari sono stati scaricati correttamente.
- UpdateFinished: l'aggiornamento è stato applicato con successo.
- Error: si è verificato un errore durante la procedura.
- UpdateAvailable: un aggiornamento è disponibile nel cloud.
- UpdateProgress: percentuale di aggiornamento applicata.

4.4 Il processo di build

UF Android Client può essere installato una volta generata la corrispondente immagine Android OS, ossia resa applicazione di sistema. Ciò avviene firmando l'APK con la chiave *platform*. La disponibilità della chiave platform è una componente fondamentale della procedura, in particolare sono richiesti due file: *platform.pk8* e *platform.x509.pem*. Esistono due principali metodi per firmare l'applicazione:

- manualmente mediante un'applicazione esterna;
- automaticamente durante il processo di build con Gradle.

Nel caso di UF Android Client, la firma avviene automaticamente grazie alla configurazione dello script `build.gradle`. Ciò risulta conveniente durante lo sviluppo, in quanto permette le operazioni di test e debug dell'applicazione senza necessità di ulteriori step. Lo script Gradle necessita, però, di un'ulteriore componente, il keystore. Per esempio, i comandi

```
1 openssl pkcs8 -in platform.pk8 -inform DER -outform PEM -out platform
    .priv.pem -nocrypt
2 openssl pkcs12 -export -in platform.x509.pem -inkey platform.priv.pem
     -out platform.pk12 -name MyAndroidKey
```

consentono di incapsulare le due chiavi platform.pk8 e platform.x509.pem in un'unica chiave *platform.pk12*, con alias MyAndroidKey. Una volta generata questa nuova chiave, per inserirla all'interno del keystore è sufficiente utilizzare il comando

```
keytool -importkeystore -destkeystore MyKeystore.jks -srckeystore
    platform.pk12 -srcstoretype PKCS12 -srcstorepass
    MyKeystorePassword -alias MyAndroidKey
```

Il keystore è un file che risiede all'interno della directory del progetto. Lo script build.gradle utilizza il keystore per firmare l'applicazione con le chiavi contenute al suo interno. La sezione *signingConfigs* contiene le informazioni necessarie per la procedura di firma, ad esempio le password per accedere al keystore stesso e alle singole chiavi.

La chiave platform dovrebbe essere tenuta segreta, ed essere diversa per ogni dispositivo, ma esistono alcune chiavi pubbliche utilizzate per build di debug di Android. Sono chiavi che non forniscono alcuna sicurezza, dato che sono reperibili online, ma sono utilizzate da molte build dimostrative di Android. La chiave AOSP è usata ad esempio dall'emulatore, mentre la chiave NXP è usata per le build di prova delle immagini Android con chip NXP.

4.5 Sviluppo: pipeline per build e test

I princìpi dell'integrazione continua sono rispettati anche da UF Android Client, grazie alla pipeline sviluppata. Il seguente script contiene il codice completo del workflow che configura il runner ed esegue il build del progetto. La sequenza delle istruzioni è analoga a quella per la libreria Hara-ddiclient, ad eccezione della procedura di firma che avviene durante l'esecuzione del comando `./gradlew build`.

```
name: build-action

on:
  push:
    branches-ignore:
      - 'gh-pages'
  pull_request:
    branches-ignore:
      - 'gh-pages'

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Setup gradle
        uses: gradle/gradle-build-action@v2

      - name: Override Android SDK
        run: ./install_android-hidden-api-jar.sh

      - name: Gradle build
        env:
          KYNETICS_KEYSTORE_PASS: ${{ secrets.KYNETICS_KEYSTORE_PASS
    }}
          KYNETICS_KEY_PASS: ${{ secrets.KYNETICS_KEY_PASS }}
          KYNETICS_KEY_ALIAS: ${{ secrets.KYNETICS_KEY_ALIAS }}

        run: ./gradlew build
```

4.5.1 Analisi del codice

name Il nome del workflow.

on Lista di eventi che scatenano l'esecuzione del workflow. In questo caso l'esecuzione avviene in seguito ad ogni 'push' di un commit e in seguito all'apertura di una 'pull request'. In entrambi i casi si è deciso di ignorare il caso in cui il push o la pull request avviene sul branch 'gh-pages', branch particolare utilizzato per la pubblicazione della documentazione.

jobs Lista di job da eseguire. L'unico job da eseguire è chiamato 'build'. UF Android Client non necessita di eseguire dei test multipiattaforma in fase di compilazione, di conseguenza è sufficiente testare il processo di build su un solo runner, in questo caso ubuntu-linux.

steps Sequenza di step da eseguire per ciascun job. In primo luogo avviene il checkout del repository. Come nel caso di Hara-ddiclient, avviene la configurazione dell'ambiente Gradle e del proprio meccanismo di caching per velocizzare le successive esecuzioni. Successivamente avviene una particolare configurazione dell'ambiente Android SDK già pre-installato nel runner. Le Android Hidden API sono un insieme di classi, metodi e risorse che Google, di norma, nasconde per motivi di stabilità. Dal momento che UF Android Client fa uso di tali API, è stato necessario rinconfigurare Android SDK pre-installato. Infine avviene il processo di build. Tale processo è caraterizzato da una procedura di firma particolare: l'app UI example è firmata con una chiave segreta di Kynetics, che però non ha alcuna peculiarità, ovvero non fornisce alcun permesso speciale. Questa chiave viene passata al runner nel costrutto *env*, ovvero come variabile d'ambiente. La parola chiave 'secrets' indica che il contenuto delle variabili è segreto, ovvero viene specificato come informazione privata interna al repository, il cui accesso è riservato solamente al proprietario dello stesso.

4.5.2 Commenti

In fase di sviluppo non sono state riscontrate particolari criticità.

4.6 Sviluppo: schedulare gli aggiornamenti

Il feedback degli utenti che fanno utilizzo di UF Android Client ha permesso di identificare un insieme di requisiti, i quali hanno portato alla progettazione e allo sviluppo di una nuova funzionalità. Un aggiornamento di tipo Force potrebbe causare un'interruzione forzata dell'attività lavorativa nel dispositivo per effettuare l'aggiornamento, con conseguenze più o meno gravi a seconda dello specifico caso di applicazione. Una prima soluzione al problema in questione, è la distribuzione di artefatti in specifiche finestre temporali, implementate lato server; tale soluzione però non tiene conto del fatto che i dispositivi sono distribuiti in tutto il mondo e, quindi, si pone il problema delle diverse time zone. Nasce l'idea di offrire all'utente (ovvero lato client) la possibilità di definire le finestre temporali personalizzate, all'interno delle quali i dispositivi sono considerati in uno stato non operativo, ossia disponibili ad applicare aggiornamenti senza causare problemi alla normale attività lavorativa.

4.6.1 Punti di intervento

Per poter implementare questa nuova funzionalità è necessario effettuare delle modifiche sia alla libreria Hara che all'app UF Android Client. L'idea di sviluppo è di impedire che gli aggiornamenti di tipo Force non vengano applicati senza prima aver ottenuto una specifica autorizzazione, la quale viene concessa solamente se il dispositivo si trova in uno stato non operativo.

Dal punto di vista della libreria Hara, è necessario operare sull'attore *DeploymentManager* (vedi cap. 3), che si occupa di avviare la procedura di download e applicazione dell'aggiornamento. In particolare, anche nel caso di aggiornamento Force, dovrà richiedere l'autorizzazione al dispositivo per il download degli artefatti, sempre concessa dal momento che non provoca interruzioni all'attività del dispositivo. Una volta ottenuta l'autorizzazione e scaricati gli artefatti, *DeploymentManager* dovrà richiedere l'autorizzazione per applicare l'aggiornamento. Solo dopo che quest'ultima verrà concessa, l'attore figlio *UpdateManager* potrà procedere con le operazioni del caso.

La maggior parte dello sviluppo si concentra sull'app UF Android Client. L'applicazione dovrà essere in grado di concedere l'autorizzazione all'aggior-

namento solamente all'interno di un determinato intervallo temporale. L'utente avrà la possibilità di personalizzare l'intervallo temporale, specificando l'istante iniziale, attraverso un'espressione *cron*, e la sua durata. È necessario, inoltre, un meccanismo di validazione dell'espressione inserita e di salvataggio delle preferenze. Ulteriori dettagli vengono forniti di seguito.

4.6.2 Autorizzare l'aggiornamento

Il meccanismo con cui vengono concesse le autorizzazioni agli aggiornamenti è implementato in Kotlin mediante le Coroutine, in particolare utilizzando l'oggetto `Deferred` [13].

Kotlin: interfaccia Deferred

In Kotlin, un'istanza dell'interfaccia Job rappresenta un'esecuzione di un blocco di codice in background, avente un proprio ciclo di vita che culmina con il suo completamento. L'interfaccia Deferred estende l'interfaccia Job, implementando metodi che consentono di ottenere il risultato dell'esecuzione del blocco di codice associato ad un Job. Si può istanziare un oggetto Deferred attraverso il costruttore della classe CompletableDeferred. L'oggetto rimane attivo fintanto che non viene completato. Per comprendere meglio il funzionamento, si consideri una struttura semplificata del meccanismo con cui avviene la gestione delle autorizzazioni, mostrata nel seguente esempio.

```
//Hara
interface DeploymentPermitProvider {
    ...
    /*Ritorna una Deferred di tipo Boolean che verrà completata
    quando l'utente decide di autorizzare l'update*/
    fun updateAllowed(): Deferred<Boolean>
}

class DeploymentManager(...) : AbstractActor(...) {
    ...
    private fun onAuthorizationReceive(){
        //Deferred posta in stato di attesa di completamento
        if(deploymentPermitProvider.updateAllowed().await()){
            //Autorizzazione concessa!
        } else {
```

```
            //Autorizzazione negata!
        }
    }
}

//UF Android Client
class DeploymentPermitProviderImpl : DeploymentPermitProvider{

    val completableDeferred = CompletableDeferred<Boolean>()

    override fun updateAllowed() : Deferred<Boolean> {
        /*La Deferred viene completata con valore 'true',
        ovvero l'autorizzazione è concessa*/
        completableDeferred.complete(true)
        return completableDeferred
    }
}
```

Il metodo `await` pone lo stato della Deferred in attesa di completamento, rimanendo tale fino all'esecuzione del metodo `complete`, il quale completa la Deferred con il valore passato come parametro. Il valore passato dev'essere compatibile con il tipo Boolean, come specificato dal costruttore di `CompletableDeferred`. Una volta completata la Deferred, l'aggiornamento viene applicato al dispositivo.

Espressione cron

L'utente definisce l'inizio dell'intervallo di tempo in cui il dispositivo è considerato non operativo, ovvero disponibile per applicare aggiornamenti, attraverso un'espressione Cron. Un'espressione Cron è una stringa che consiste di una serie di campi (solitamente dai cinque ai sette) che, individualmente, descrivono i dettagli dello schedule. Esempi di espressioni cron sono:

- 0 15 10 * * ? - ogni giorno alle 10:15 AM
- 0 15 10 ? * MON-FRI - alle 10:15 AM ogni Lunedì, Martedì, Mercoledì, Giovedì e Venerdì
- 0 15 10 ? * 6L - alle 10:15 AM di ogni ultimo Venerdì del mese

Una volta specificato l'inizio e la durata, qualsiasi richiesta di aggiornamento di tipo Force, rimarrà in stallo, in attesa che il dispositivo entri nello stato non

operativo. Dal punto di vista del codice, come visto sopra, ciò significa che l'oggetto Deferred (CompletableDeferred) verrà completato con valore *true* solo quando il dispositivo entra nello stato non operativo.

Attivazone della finestra temporale

Per gestire lo scheduling dell'aggiornamento, è stata implementata la classe `CronScheduler`, aderente al pattern Singleton, implementato in Kotlin attraverso il type class `object`. Questa speciale struttura consente di avere una sola istanza della classe, in quanto Kotlin, in fase di compilazione, rende il costruttore della classe privato ed assegna un riferimento statico all'unica istanza generata. La classe `CronScheduler` possiede un metodo `schedule`, che utilizza l'espressione cron specificata dall'utente per calcolare l'inizio della finestra temporale. Nel seguente script è mostrato come avviene il calcolo dell'istante iniziale della finestra temporale, partendo dalla sola espressione cron inserita dall'utente.

```
private var authJob: Job? = null

fun schedule(expression: String, authorize: () -> Unit){
    //parsing dell'espressione
    ...
    if(isNowOnValidTimeWindow()){
        authorize()
    } else {
        authJob = GlobalScope.launch(Dispatchers.IO){
            val triggerMillis = startMillis - now()
            delay(triggerMillis)
            authorize()
        }
    }
}
```

Come primo passo, utilizzando la libreria esterna `cronutils` [14], viene effettuato un parsing della stringa cron, per ricavare l'inizio della finestra. Successivamente viene controllato se l'istante corrente si trova in una finestra temporale valida, quindi se il dispositivo non è operativo. In questo caso l'aggiornamento viene autorizzato, altrimenti viene lanciata una coroutine, la quale si occupa di calcolare la differenza tra l'istante di inizio specificato (`startMillis`) e l'istante corrente (`now`), in millisecondi. Il risultato di questa operazione verrà passato

al metodo `delay`, una funzione che si occupa di sospendere l'esecuzione della coroutine a cui è associata, generata dal costrutto `GlobalScope.launch`, per un tempo pari al suo argomento. Dopo questo tempo lo stato del dispositivo è considerato non operativo, quindi l'aggiornamento può essere applicato. Nel metodo `authorize` viene richiamata la funzione `complete`, il quale completa la Deferred, concedendo l'autorizzazione.

4.6.3 Salvataggio delle preferenze

Android mette a disposizione meccanismi per la collocazione di dati persistenti su file di vario tipo. Shared Preferences [15], in particolare, consente di memorizzare dati primitivi (string, int, float, boolean,...) o informazioni di configurazione elementari in un file XML, situato nello storage locale del dispositivo. L'oggetto Shared Preferences rappresenta un riferimento a questo file, contenente un insieme relativamente contenuto di dati, sotto forma di coppie chiave/valore e accessibile tramite specifiche API. Questo meccanismo risulta comodo, ad esempio, nel caso in cui si voglia memorizzare le preferenze utente di un'applicazione condivise tra le varie activity di cui l'app è composta, senza ricorrere a basi di dati complesse. I dati salvati nelle Shared Preferences sono persistenti tra diverse esecuzioni dell'applicazione a cui fanno riferimento.

UF Android Client fa uso delle Shared Preferences per salvare le preferenze dell'utente, in particolare le informazioni di configurazione del servizio, viste in precedenza. L'implementazione della funzionalità prevede l'inserimento nell'activity Settings di due voci, *Schedule Update* e *Duration*, le quali consentono di specificare rispettivamente l'inizio della finestra temporale, tramite espressione cron, e la sua durata, espressa nel formato 'hh:mm:ss'. Una volta completati questi campi tramite le apposite finestre di dialogo, i dati inseriti vengono salvati attraverso le Shared Preferences.

Validazione dell'input

La libreria citata in precedenza, utilizzata per il parsing dell'espressione cron, offre dei metodi che controllano la sintassi della stringa cron, generando delle eccezioni in caso di errore. Sfruttando tali funzionalità è stato implementato un meccanismo di validazione dell'input, applicato alle finestre di dialogo che consentono di inserire i dati, in particolare accedendo all'oggetto 'EditText',

ossia il box in cui è possibile inserire il testo. In questo modo, nel caso in cui, per esempio, l'espressione cron non rispetti la sintassi, verrà generato un messaggio di errore e la preferenza non verrà aggiornata, come mostrato in Figura 4.3.

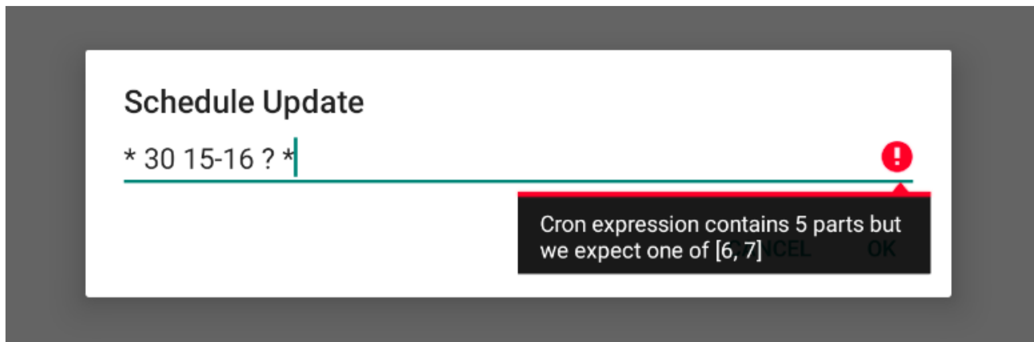


Figura 4.3: Esempio di errore di sintassi nell'espressione cron

Conclusioni

Il concetto di "Internet Of Things" (IoT) si è diffuso esponenzialmente negli ultimi decenni ed è ormai parte essenziale della nostra vita quotidiana. Ad oggi si contano più di 40 miliardi di dispositivi connessi e operativi a livello globale, l'equivalente di oltre 5 dispositivi per ogni persona, ed è un numero destinato a crescere nei prossimi anni. L'avanzata potenzialità delle tecnologie IoT trova spazio in numerosi settori, tra cui residenziale, sanitario, sicurezza pubblica e trasporti, industriale, energetico e ambientale. Oltre ai vantaggi pratici, la crescita esponenziale del numero di dispositivi connessi alla rete comporta anche una valutazione dal punto di vista tecnico. Dato l'alto grado di interazione reciproca, questi prodotti necessitano di tenere il passo dell'evoluzione tecnologica, adottando sistemi hardware di ultima generazione e soluzioni software moderne ed efficienti. Questo elaborato ha inteso sottolineare l'importanza di distribuire, in modo controllato ed efficiente, aggiornamenti software a sistemi embedded facenti parte di complesse reti IoT. L'abilitazione delle pipeline fornisce una garanzia di robustezza e uniformità al complesso software per la comunicazione con Update Factory, fondamentale vista la sua natura open source. Inoltre, le richieste di estensione delle funzionalità dell'app UF Android Client hanno permesso di comprendere l'importanza di rendere il servizio uniforme a livello mondiale, dimostrando la sconfinata diffusione della tecnologia IoT.

Bibliografia

[1] Kynetics. *Kynetics: enjoy the art of coding*. URL: https://www.kynetics.com/. [Consultato il 30.08.2022].

[2] Eclipse Foundation. *Eclipse hawkBit project*. URL: https://projects.eclipse.org/proposals/hawkbit. [Consultato il 25.08.2022].

[3] Eclipse.org. *Eclipse hawkBit*. URL: https://www.eclipse.org/hawkbit/. [Consultato il 5.09.2022].

[4] Update Factory. *Update Factory Documentation*. URL: https://docs.updatefactory.io/. [Consultato il 01.09.2016].

[5] Wikipedia. *Integrazione Continua*. URL: https://it.wikipedia.org/wiki/Integrazione_continua. [Consultato il 22.08.2022].

[6] GitHub Actions. *Understanding GitHub Actions*. URL: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions. [Consultato il 22.08.2022].

[7] simplilearn.com. *What is Gradle?* URL: https://www.simplilearn.com/tutorials/gradle-tutorial/what-is-gradle. [Consultato il 31.08.2022].

[8] Kotlin. *Kotlin: Coroutine basics*. URL: https://kotlinlang.org/docs/coroutines-basics.html. [Consultato il 26.08 2022].

[9] The Docker Handbook. *Learn Docker for beginners*. URL: https://www.freecodecamp.org/news/the-docker-handbook/. [Consultato il 30.08.2022].

[10] Eclipse. *Eclipse Hara: Hara-ddiclient*. URL: https://github.com/eclipse/hara-ddiclient. [Consultato il 23.08.2022].

[11] Kynetics. *UFAndroidClient*. URL: https://github.com/Kynetics/uf-android-client. [Consultato il 24.08.2022].

[12] Android Developers. *Android Service overview*. URL: https://developer.android.com/guide/components/bound-services. [Consultato il *.09.2022].

[13] Kotlin. *Kotlin: Coroutine Deferred*. URL: https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-deferred/. [Consultato il 26.08.2022].

[14] GitHub.com. *cron-utils*. URL: https://github.com/jmrozanec/cron-utils. [Consultato il *.08.2022].

[15] Android Developers. *Save key-value data*. URL: https://developer.android.com/training/data-storage/shared-preferences. [Consultato il 25.08.2022].