DOCUMENTO ESTERNO

Carbon7team

carbon7team@gmail.com

27 Giugno 2022

Organizzazione github: Carbon7team

Manuale dello Sviluppatore

v1.0.0

Redattori	**Revisori**
Andrea Polato	Adnan Latif Gazi
Filippo Brugnolaro	Leonardo Speranzon
Adnan Latif Gazi	Filippo Brugnolaro
Leonardo Speranzon	Andrea Polato
Damiano D'Amico	
Marco Odinotte	

Sommario

Documento gestionale Esterno relativo al *ManualeDelloSviluppatore*$_G$ del Carbon7team

Storico modifiche al documento

Legenda:

- +: Prima redazione di contenuto
- #: Estensione di contenuto
- [n]: Sezione n del documento

Versione	Operazione	Autore	Verficatore	Data
1.0.0	Approvazione Documento	Adnan Latif Gazi	Adnan Latif Gazi	2022/06/27
0.2.0	Revisione Documento	Adnan Latif Gazi	Adnan Latif Gazi	2022/06/27
0.1.5	+ View e ViewModel [5.1.2]	Marco Odinotte	Leonardo Speranzon	2022/06/27
0.1.4	+ Diagrammi di sequenza [6]	Leonardo Speanzon	Adnan Latif Gazi	2022/06/27
0.1.3	+ Nodo Server [5.3]	Damiano D'Amico	Leonardo Speranzon	2022/06/27
0.1.2	+ Virtual Display - Modello [5.1.1] + Virtual Display - Servizi [5.1.3]	Leonardo Speranzon	Adnan Latif Gazi	2022/06/26
0.1.1	+ Virtual Display [5.1]	Adnan Latif Gazi	Leonardo Speranzon	2022/06/26
0.1.0	Revisione Documento	Adnan Latif Gazi	Adnan Latif Gazi	2022/06/26
0.0.5	+ Model [5.2.1] + View [5.2.2]	Filippo Brugnolaro	Adnan Latif Gazi	2022/05/10
0.0.4	# Tecnologie adottate [2] # Installazione di Remote Support [4.3] + Remote Support [5.2]	Filippo Brugnolaro	Leonardo Speranzon	2022/05/09
0.0.3	+ Configurazione [3] + Installazione [4]	Andrea Polato	Filippo Brugnolaro	2022/04/05

Versione	Operazione	Autore	Verficatore	Data
0.0.2	+ Introduzione [1] + Tecnologie adottate [2]	Andrea Polato	Filippo Brugnolaro	2022/04/04
0.0.1	Generazione Documento	Andrea Polato	Andrea Polato	2022/03/24

Indice

1 Introduzione

1.1 Scopo del documento

Il documento descrive le scelte architetturali prese da Carbon7team nella realizzazione del prodotto in questione. Il fine del documento è quello di fornire una panoramica dettagliata delle tecnologie utilizzate e delle scelte implementative, per garantire la manutenibilità agli sviluppatori che in futuro prenderanno in carico questo prodotto.

1.2 Scopo del prodotto

Il capitolato C6 prevede lo sviluppo di un sistema di assistenza, che garantisca supporto tecnico all'utente utilizzatore di dispositivi UPS. Tale sistema prevede sia la creazione di un applicativo, per monitorare lo stato del dispositivo UPS, sia di una piattaforma di sostegno che aiuti il tecnico nello svolgimento della sua mansione di assistenza.

1.3 Glossario

Per assicurare la massima trasparenza e fruibilità del documento, il *Carbon7team* ha deciso di stilare il *Glossario*. Qui verranno inserti tutti i termini ambigui o relativi all'attività del progetto, che il gruppo individua come degni di nota. I termini qui presenti saranno identificati attraverso una 'G' a pedice.

1.4 Riferimenti

1.4.1 Riferimenti normativi

- *NormeDiProgetto_3.0.0*$_G$
- Regolamento del progetto didattico - dispense
- Capitolato C6: Smart4Energy

1.4.2 Riferimenti informativi

- Design pattern architetturali - dispense
- Design pattern creazionali - dispense
- Design pattern strutturali - dispense
- Design pattern comportamentali - dispense
- Pattern MVC e derivati - dispense
- SOLID programming - dispense

2 Tecnologie adottate

Tecnologia - Versione	**Descrizione**
Linguaggi	
Kotlin - x.x.x	Linguaggio di programmazione utilizzato per realizzare l'applicazione mobile Virtual Display
JavaScript - x.x.x	linguaggio di programmazione orientato agli eventi utilizzato nella realizzazione del nodo server e di Remote Support.
Framework	
React - 17.0.2	Libreria utilizzata per la realizzazione dell'interfaccia utente di Remote Support.
PeerJS - 1.3.2	
Socket.io-Client - 4.4.1	
Express - x.x.x	
Junit - x.x.x	Framework di testing utilizzato nello sviluppo di Virtual Display.
Jest - 27.5.1	Framework di testing utilizzato nello sviluppo di Remote Support.
Strumenti	
NPM - 7.24.2	Strumento di gestione dei pacchetti JavaScript utilizzati.
React-Select - 5.3.0	
Node.js - x.x.x	Runtime che permette di eseguire moduli JavaScript.
Heroku - x.x.x	
Gradle - x.x.x	Strumento per l'automazione dello sviluppo di Virtual Display.
PostrgreSQL - x.x.x	Database utilizzato per immagazzinare i dati degli utenti.

3 Configurazione

Di seguito vengono riportati i requisiti hardware e software necessari all'utilizzo degli strumenti sfruttati nella realizzazione del prodotto finale.

3.1 Requisiti hardware - Virtual Display

Si necessità di un PC con:

- Windows 8 (o successivo) 64-bit;
- 8 GB RAM;
- 8 GB minimo di spazio su disco (IDE + Android SDK + Android Emulator);
- CPU con supporto alla virtualizzazione;
- schermo con risoluzione minima di 1280x800.

3.2 Requisiti software - Virtual Display

- Android Studio Bumblebee 2021.1.1 - download;
- Simulatore Modbus - rilasciato da SOCOMEC®, link per il download non disponibile.

3.3 Requisiti hardware - Remote Support

-

3.4 Requisiti software - Remote Support

- Node.js - download;
- NPM - installazione tramite terminale.

3.5 Altri requisiti

- Una porta di rete libera per l'esecuzione del simulatore (predefinita: 8888);
- Un server raggiungibile da entrambi il Virtual Display e il Remote Support per instaurare la comunicazione tra le parti e recuperare i dati di login.

4 Installazione

Il prodotto finale è diviso in 3 repository differenti, in modo da fornire solo la parte interessata nel caso in cui si disponesse già delle altre. Di seguito sono riportati i link ad esse:

- Virtual Display;
- Remote Support;
- Nodo server.

4.1 Installazione di Virtual Display

- Clonare la relativa repository;
- Eseguire l'emulatore del modbus:
  - Dalla directory del simulatore, entrare nella cartella `bin` e avviare un terminale;
  - Digitare il comando `.\ModbusSlave.exe -d . --dump`;
  - Se si desidera avere informazioni sui comandi disponili, digitare: `.\ModbusSlave.exe --help`;
- Aprire Android Studio e aprire il progetto contenuto nella cartella di clonazione;
- Eseguire l'emulazione dell'applicazione;
- In alternativa, da Android Studio, generare il pacchetto .apk e installarlo su un dispositivo Android fisico o emulato;

4.2 Installazione del server

- Clonare la relativa repository;
- Aprire il terminale e posizionarsi nella cartella contente index.js;
- Eseguire il comando `node index.js`.

4.3 Installazione di Remote Support

- Clonare la relativa repository;
- Posizionarsi all'interno della cartella principale;
- Installare i moduli e le dipendenze necessari tramite il comando `npm install`;
- Posizionarsi all'interno della cartella `src`, contenente il file `index.js`;
- Eseguire il comando `npm start`.

5 Progettazione architetturale

5.1 Virtual Display

Per il Virtual Display è stato deciso di usare il design pattern **Model-View-View-Model** per definire la sua struttura architetturale. La scelta risieda nella facilità che questo design pattern offre nella separazione tra l'interfaccia grafica e la componente logica del programma, evitando in questo modo anche la dipendenza della grafica dal modello. L'impossibilità in Android di creare un'interfaccia grafica "intelligente", ovvero che in grado di comunicare direttamente con il modello per richiedere i dati necessari per la visualizzazione, ha portato alla scelta di questa variante del design pattern anzicché a quelli più comuni come il **Model-View-Controller** o il **Model-View-Presenter**.

Per semplificare al meglio l'interazione con l'interfaccia grafica, abbiamo utilizzato il **ViewBinding** che permete una scrittura più pulita e semplice della vista andando a sostituire il procedimento standard di reperimento degli elementi grafici (**FindViewById**). A livello di compilazione e stesura del codice il ViewBinding notifica immediatamente possibili errori di scrittura, evitando problemi durante l'esecuzione del programma.

La presenza di dati da osservare nel modello è necessario per la loro visualizzazione grafica e l'aggiornamento al loro variare. Perciò, è stato deciso di sfruttare il **LiveData**, ovvero una classe dei dati osservabili considerato una variante del design pattern dell'**Observer**, con la particolarità che il LiveData rispetta il ciclo di vita di altri componenti dell'app, come attività, frammenti o servizi, facendo sì che aggiorni solo gli osservatori dei componenti dell'app che si trovano in uno stato del ciclo di vita attivo. Oltretutto, implementa automaticamente il funzionamento di un normale Observer, gestisce il proprio ciclo di vita, ed è particolarmente efficiente a dialogare con le attività Android ai loro cambiamenti ed errori durante la loro esecuzione.

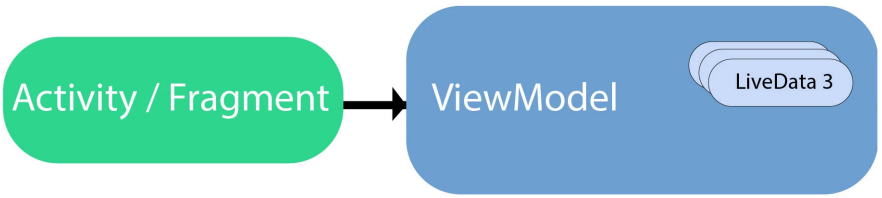

Figura 1: LiveData

Per garantire minor accoppiamento tra le componenti del programma, è stato usata la classe **EventBus** appositamente creata. Essa usa il design pattern **publisher/subscriber**, intermediando la comunicazione tra più componenti, che così sono più debolmente accoppiate. EventBus è efficiente e semplice da usare, e gestisce diversi tipi di comunicazione.

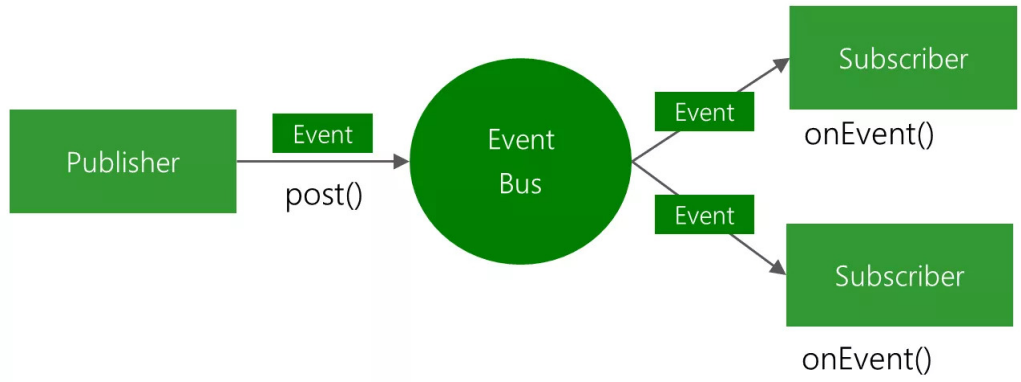

Figura 2: EventBus

Il Virtual Display prevede di eseguire spesso funzionalità parallelamente ad altre operazioni già in corso, come ad esempio l'aggiornamento dei dati relativi ad un UPS dopo che un utente ne ha modificato le informazioni salvate dall'apposita schermata. A tal proposito è stato deciso di usare le **Coroutine**, ovvero dei design pattern legati alla concorrenza e messi a disposizione da Android per semplificare l'esecuzione di funzioni asincrone. Oltre ad essere semplici da usare, risolvono tutti i problemi legati al parallelismo tra componenti, garantendo anche efficienza nella loro esecuzione.

In un ambiente in cui vengono sfruttate delle Coroutine, i Flow sono dei tipi che possono emettere multipli valori dello stesso tipo, rappresentando quindi uno stream asincrono. Vengono solitamenti usati per ricevere in tempo reale dati osservati man mano che essi si aggiornano. Vengono usati nel Virtual Display per l'aggiornamento dei dati nel **ViewModel** man mano che essi cambiano nel modello. Dal ViewModel tali dati vengono passati nella vista della selezione dell'UPS che così può rappresentarle sempre aggiornate.

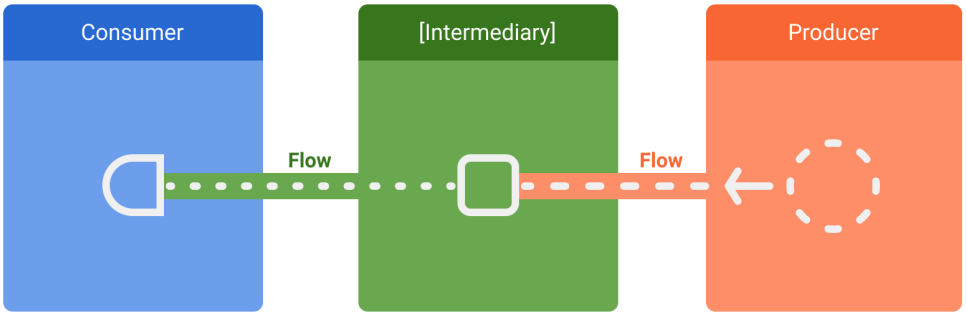

Figura 3: Flow

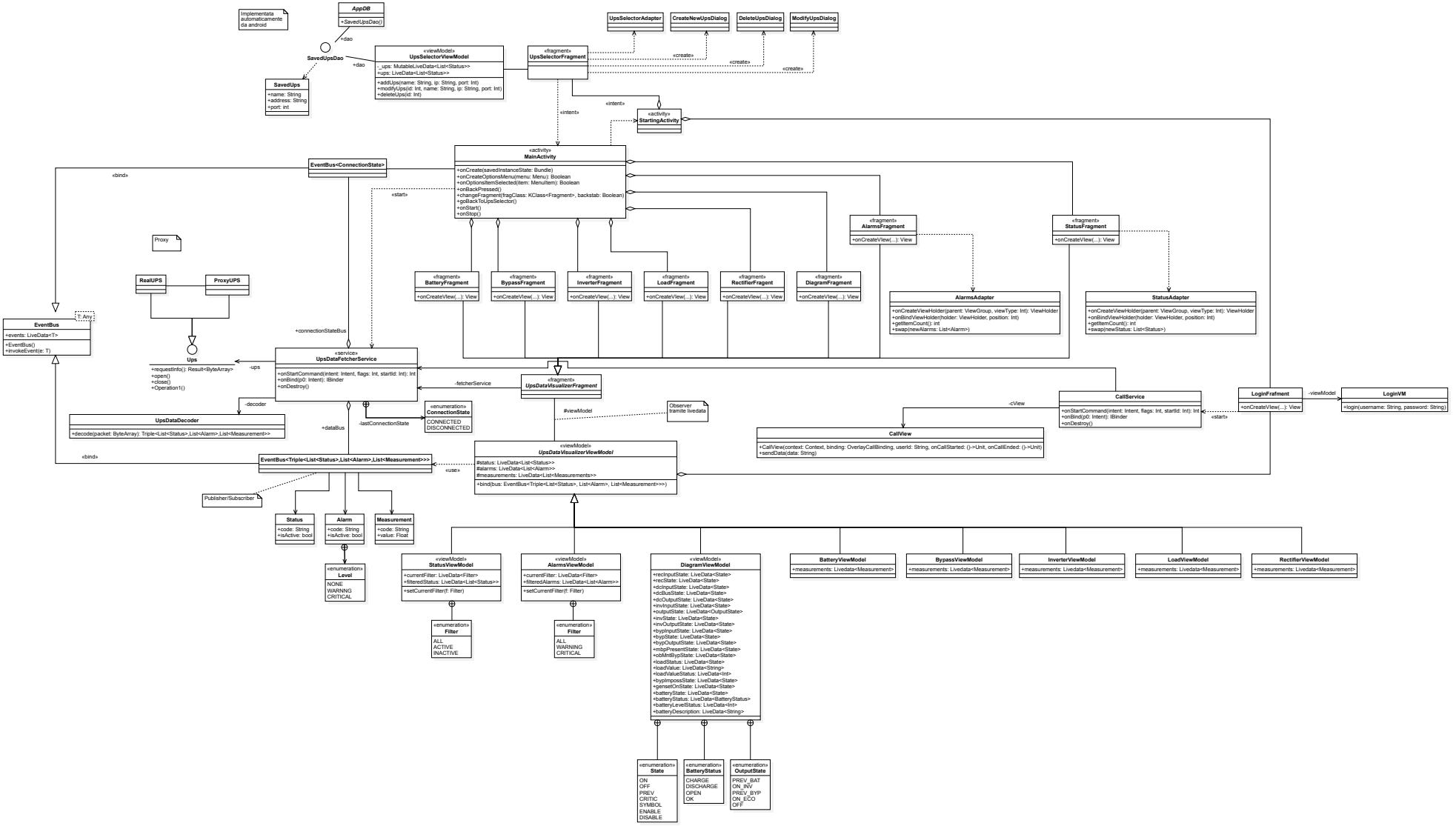

Figura 4: Diagramma delle classi Virtual Display

5.1.1 Model

Il modello è composto da:

- Il database Room;
- Le classi legate alle ricezione e decodifica dei dati dell'UPS;
- Le classi utilizzate per contenere i dati dell'UPS (Stati, Allarmi e Misurazioni).

5.1.1.1 Room

Per poter salvare in modo persistente gli ups (ip, porta e nome) all'interno dell'applicazione abbiamo optato per l'uso della libreria Room di Android che offre un layer di astrazione su SQLite che semplifica sia la sua creazione che l'uso.

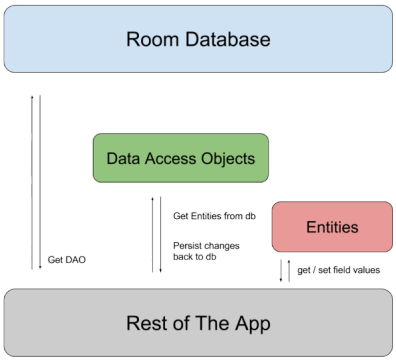

Figura 5: Schame Room

5.1.1.2 Classi resposabili della business logic

La business logic riguardante i dati provenienti dall'ups ha richiesto l'implementazione di due classi: una responsabile della ricezione (ProxyUps) e l'altra responsabile della decodifica dei dati grezzi. La classe ProxyUps fa parte di un pattern Proxy con lo scopo di impersonare l'ups reale.

Queste classi vengono coordinate da un servizio (UpsDataFetcherService [5.1.3.1]) che ha funzioni di modello anche se si tratta di un componente Android e quindi inutilizzabile al di fuori di applicazioni Android.

5.1.1.3 Data classes

Per lavorare coi dati prodotti dall'ups abbiamo deciso di modellare tre diverse classi (Status, Alarm e Measurement) con lo scopo di rappresentare quei dati. Rispettivamente contengono:

- **Status** contiene il codice (Sxxx) e se attualmente è attivo;
- **Alarm** contiene il codice (Axxx), se attualmente è attivo ed il livello di gravità (WARNING, CRITICAL o nessuna);
- **Measurement** contiene il codice (Mxxx) ed il valore rappresentato in virgola mobile.

Per agevolare la necessaria traduzione a runtime i nomi non sono salvati all'interno degli oggetti ma viene recuperato dalle risorse di Android partendo dal codice del dato.

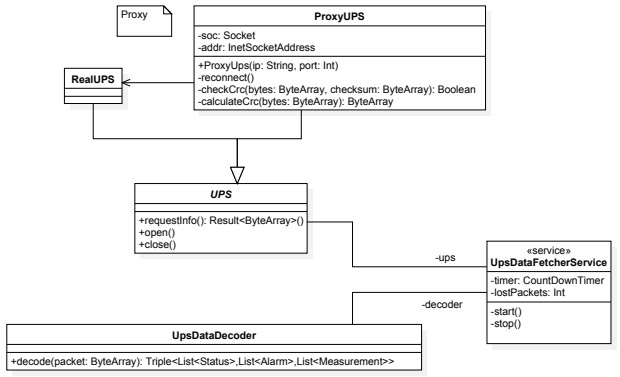

Figura 6: Classi del modello responsabile della comunicazione con l'Ups

5.1.2 View e ViewModel

La comunicazione tra il dato e la UI è rappresentato, come già accennato in precedenza, dal ViewModel che qui viene rappresentato dalla classe astratta **UpsDataVisualizerViewModel**. Tale classe è responsabile della preparazione e gestione dei dati provenienti da *Stati*, *Allarmi* e *Misurazioni* per conto di un Activity o di un Fragment; gestendo anche la comunicazione tra l'Activity/Fragment designato e il resto dell'applicazione. L' *UpsDataVisualizerViewModel*, tramite le sue varie specializzazioni nelle classi derivate, si occupa di acquisire e preservare l'informazione necessaria ad un determinato Activity o Fragment, permettendogli di monitorare eventuali cambiamenti al suo interno tramite il meccanismo dei *LiveData*. Avvalendoci di classi ausiliarie per la codifica dei vari stati che gli elementi della View possono assumere, il ViewModel immagazzina in LiveData osservabili tali stati o valori che permetteranno di provocare relative modificazioni alla UI ed evitare commistioni inutili tra componenti differenti.

A completamento del contesto sopracitato, la classe astratta **UpsDataVisualizerFragment**, derivante dalla classe *Fragment*, permette di osservare le informazioni racchiuse nei valori dell' *UpsDataVisualizerViewModel* e suscitare relativi cambiamenti nella View attraverso il meccanismo modulare dei **Fragment** messi a disposizione da Android. Questi rappresentano una porzione riutilizzabile dell' UI associata all'applicativo, definiscono e organizzano facilmente il proprio layout, gestendo gli input a loro personalmente diretti. Il loro punti forte sono proprio la modularità e riusabilità, che permettono di suddividere l'UI in un numero discreto di blocchi o moduli. Nel caso del nostro applicativo, ogni Fragment derivato da *UpsDataVisualizerFragment* permette di gestire e modificare il proprio modulo specifico di competenza, osservando i cambiamenti che avvengono nel corrispettivo ViewModel. Questo collegamento diretto si manifesta, poi, nel layout dell'applicativo attraverso il binding tra determinati elementi grafici e l'elaborazione dati proposta da View-ViewModel. Il binding comprende sia elementi puramente testuali come le *TextView*, ma anche elementi grafici come le *ImageView* che mutano il loro aspetto (colore, cambio di icona, livello della batteria ...) grazie al Fragment designato alla loro gestione.

Con lo stesso meccanismo di comunicazione funziona anche la componente **UpsSelector** che attraverso le classi *UpsSelectorViewModel* e *UpsSelectorFragment* permette di costruire l'UI designata alla scelta, modifica e gestione degli UPS registrati dall'applicativo. In base ai LiveData osservati, il Fragment corrispondente potrà operare le trasformazioni richieste e ospitare la *MainActivity* risultante da tutto questo processo di decisione operato dall'utente. Quest'ultima sarà anche la dimora di tutti i fragment sopracitati che si alterneranno nelle varie View in base alle necessità dell'utente.

5.1.3 Servizi

5.1.3.1 UpsDataFetcherService

Il servizio UpsDataFetcherService è più precisamente un Bound Service ovvero un servizio a cui, una volta avviato, molteplici "client" possono collegarsi ed utilizzarne la stessa istanza. Anche se all'apparenza molto simile ad un Singletone un Bound Service ha la caratteristica di avere un ciclo di vita dove viene inizializzato e poi distrutto. Lo scopo di UpsDataFetcherService è quello di richiedere i dati all'ups, decodificarli ed inviarli ai ViewModel o comunque ai suoi usufruitori ogni determinato periodo di tempo.

La richiesta e decodifica viene eseguita tramite le due classi del modello create a questo scopo, per l'invio dei dati ai client invece viene utilizzato la classe EventBus creata secondo il pattern Publisher/Subscriber in modo che non solo il Producer (servizio) non debba conoscere gli iscritti ma anche il contrario (i client non conoscono il producer). In questo modo i ViewModel possono osservare l'eventBus senza dover conservare al loro interno l'istanza del servizio che essendo un oggetto della libreria Android non sarebbe previsto dal pattern architetturale (per quanto riguarda i ViewModel) e li renderebbe più complicati da testare.

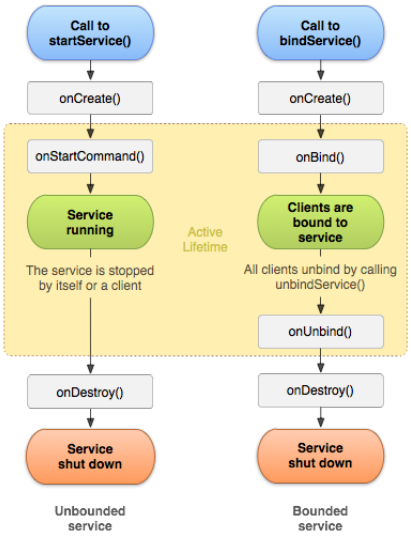

Figura 7: Lyfecicle dei diversi tipi di servizi

UpsDataFetcherService è sia di tipologia started (a destra 7) che bounded (a sinistra 7) e quindi il suo lyfecicle viene gestito dalla MainActivity che lo crea inizializzandolo con l'indirizzo dell'UPS e lo ferma quando si ritorna alla schermata di scelta dell'UPS. Dopo il suo avvio ogni client, che all'interno della nostra applicazione sono le viste ed il servizio per le chiamate, può indipendentemente accoppiarsi ed osservare gli eventBus

5.1.3.2 FloatingCallService

Il servizio FloatingCallService è un ForegroundService ovvero un servizio di cui l'utente nota l'effetto, in questo caso la chiamata ed il bottone per il suo controllo. Questa funzionalità è stata implementata tramite un servizio e non una normale Activity o Fragment in quanto doveva essere indipendente dall'Activity/Fragment corrente e dato che non possono esserci più Activity allo stesso momento l'uso del servizio è stata la soluzione migliore. Il servizio al suo interno è composto principalmente da tre parti:

- La **webview** che gestisce la chiamata WebRTC facendo uso della libreria PeerJS che semplifica di molto la sua gestione sia per la chiamata in sè che la comunicazione col signaling server;
- Il **layout** è gestito da una classe apposita ed è composto da un bottone floattante con un menu espandibile contente i pulsant per la gestione della chiamata;
- La parte relativa all'Il **encoding** dei dati dell'ups ed il loro invio consiste in un timer che ogni determinato intervallo di tempo attraverso un parser addattato traduce le liste di stati, allarmi e misurazioni in una stringa JSON e la passa alla webview per inviarla al tecnico.

5.2 Remote Support

Per quanto riguarda la parte del Remote Support, a livello prettamente architetturale, è stato utilizzato un approccio **Model-View**.
Il motivo sta nel fatto che, nella libreria di React, le componenti funzionali sono delle funzioni stateless dove si va a inserire tramite delle feature tipiche di React lo stato e le funzionalità della componente.
Inoltre è stata utilizzata una libreria esterna, *MobX*, che ha svolto il compito di implementare il **design pattern observer**, utile sia per la visualizzazione dei dati in real time ricevuti tramite la connessione peer-to-peer che per la condivisione di variabili di stato che legano componenti che non hanno nessuna correlazione tra loro.
Diretta conseguenza dell'utilizzo di questa libreria è l'attuazione del **RootStore pattern**, contenente i 2 contenitori rispettivamente di *DatasetStore* e *StateUIStore*.

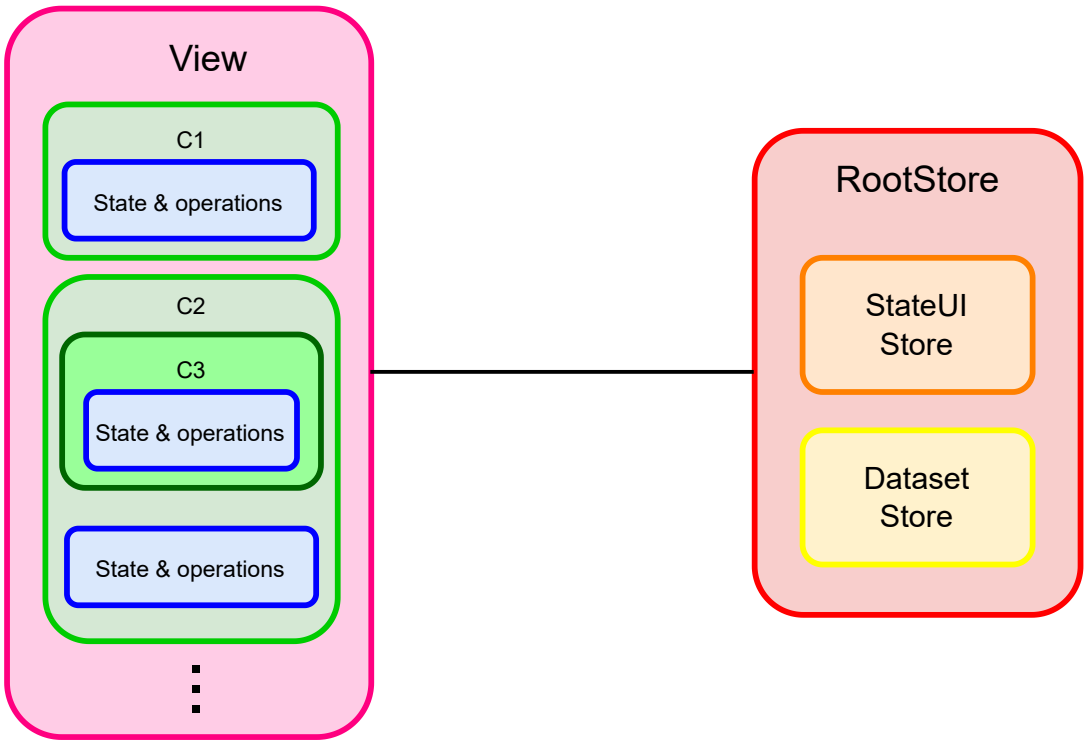

Figura 8: Architettura

Legenda:

- *Cx*: componente x, ognuna può essere formata da più componenti più piccole

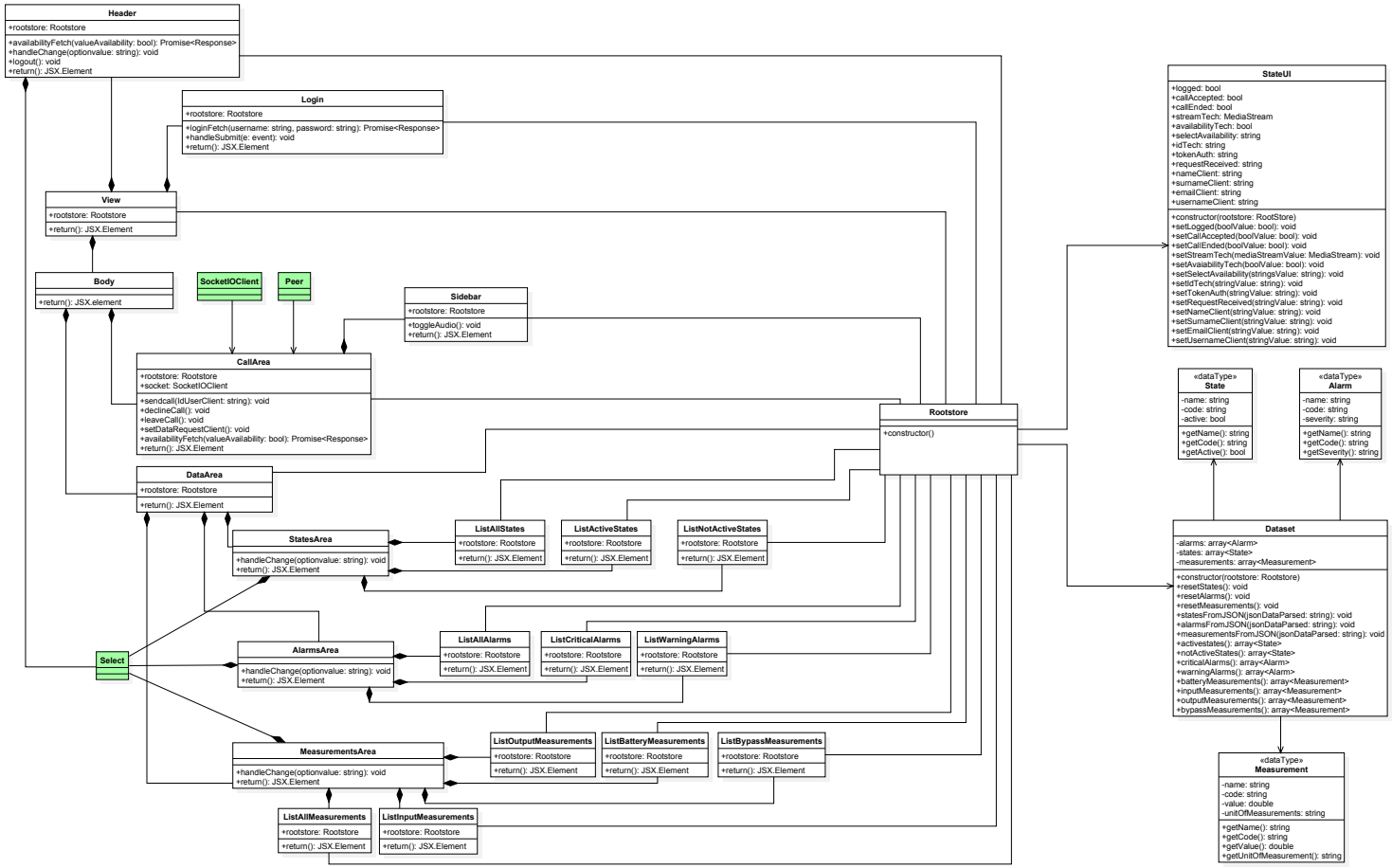

Figura 9: Diagramma delle classi Remote Support

5.2.1 Model

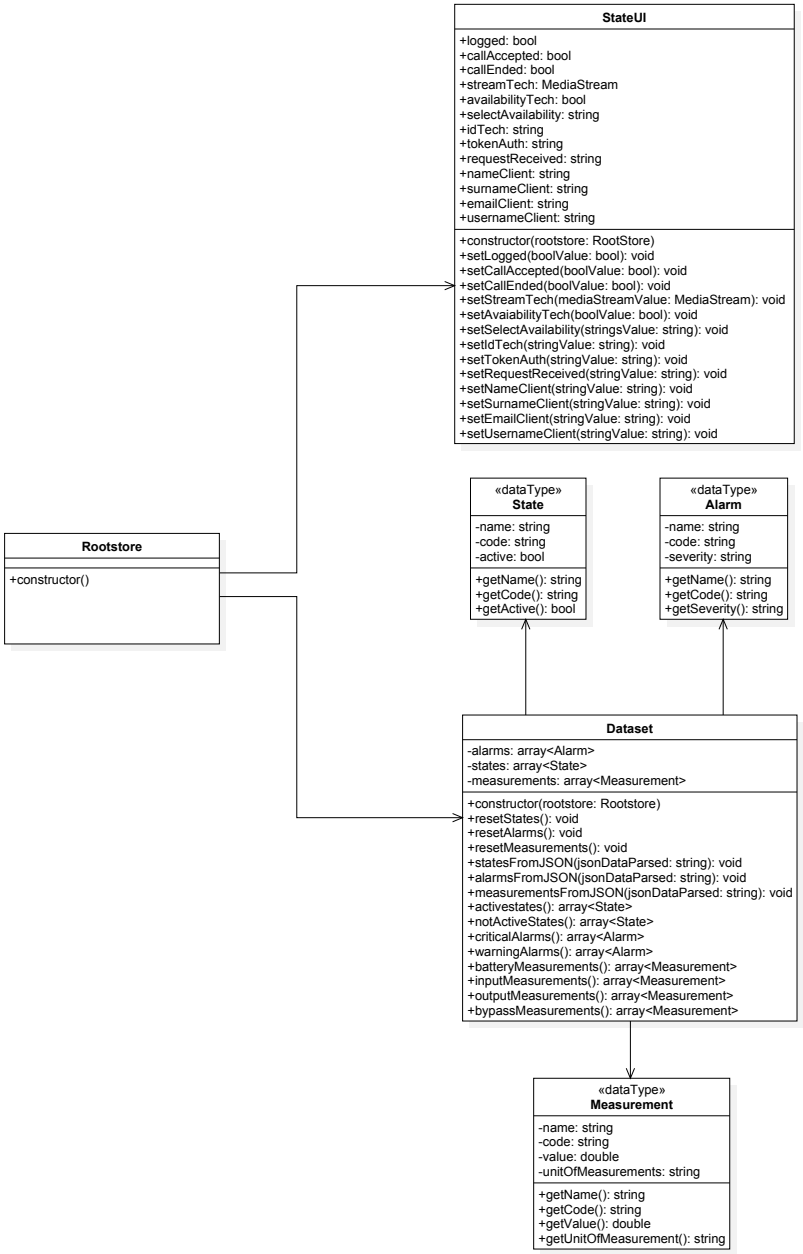

Figura 10: Diagramma delle classi Remote Support, zoom sul Model

Il diagramma delle classi del *Model* è costituito da un **RootStore**, il quale istanzia i due store utilizzati nel Remote Support:

- *StateUI Store*: contiene le variabili di stato che sono condivise tra componenti che non sono direttamente correlate tra loro (esempio: variabili per la renderizzazione condizionale).
- *Dataset Store*: contiene i dati che vengono presi dalle componenti della View, dotato di metodi per il recupero e l'aggiornamento degli stessi.

State, **Alarm** e **Measurement** sono i tipi che rappresentano le varie tipologie di dati.
Gli store contengono gli attributi observable che, nel momento in cui vengono modificati, grazie all'utilizzo di *MobX*, causano la re-renderizzazione dei componenti observer della vista.

5.2.2 View

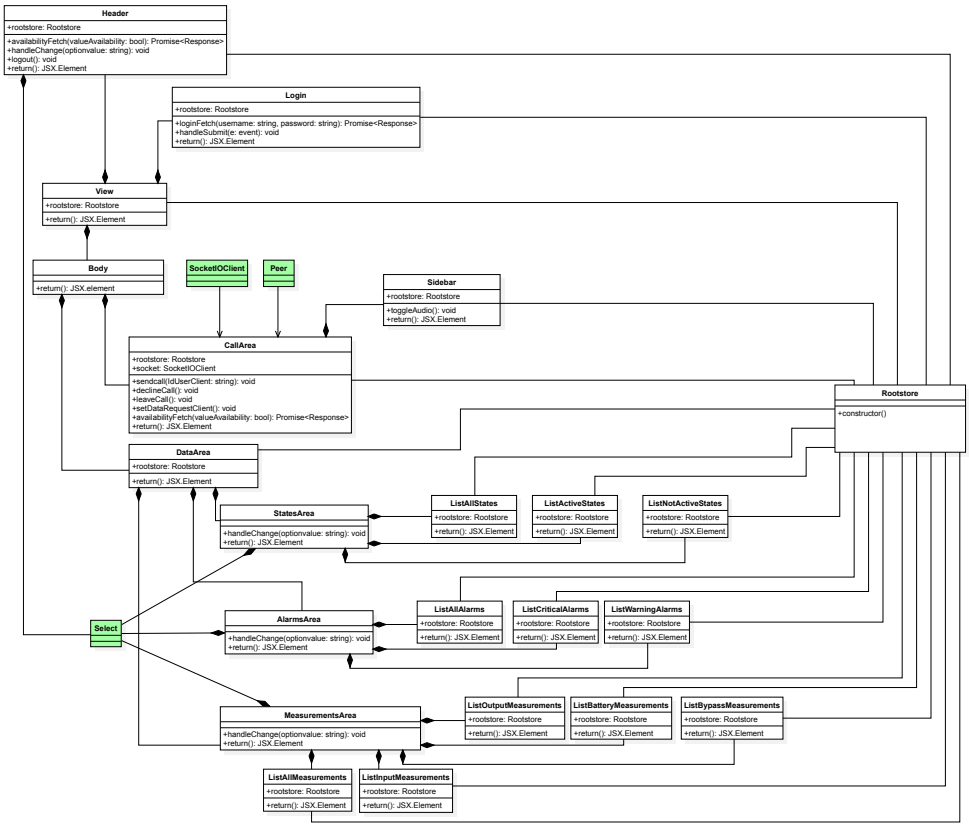

Figura 11: Diagramma delle classi Remote Support, zoom sulla View

Il diagramma delle classi della vista è costituito da tutti i componenti React che compongono l'interfaccia utente della UI. Visitando dall'alto la gerarchia di componenti, il padre è **View**, il quale crea l'**Header** e i due componenti principali **Login** e **Body**, tutti renderizzati condizionalmente in base al fatto che il tecnico abbia effettuato o meno l'accesso correttamente.
Riguardo a quest'ultimi due componenti:

- *Login*: rappresenta il punto di accesso per il tecnico
- *Body*: contiene una *DataArea* per la rappresentazione dei dati con i relativi filtri e una *CallArea* per la visualizzazione dello stato della chiamata

Tutte le classi sono degli **observer** verso il **Rootstore**, dunque faranno un re-render in caso le variabili cambino di valore.
In verde sono evidenziate le classi che hanno dipendenze che però provengono da librerie esterne.

5.3 Nodo Server

Il Nodo Server è un server javascript che gestisce le seguenti funzionalità:

- *Autenticazione*: il server gestisce l'autenticazione dei tecnici e dei clienti
- *Gestione Chiamate*: Per la gestione delle chiamate utilizziamo la libreria PeerJS. Viene inizializzato un server PeerJS che gestisce i vari id dei client e permette la connesisone tra di loro.
- *RestApi*: il server Rest è fondamentale per la gestione del tecnico, che può gestire la sessione e la propria disponibilità per ricevere chiamate di assistenza

Per quanto riguarda il modello abbiamo utilizzato l'ORM Sequelize che ci ha permesso di gestire i dati all'interno del database.

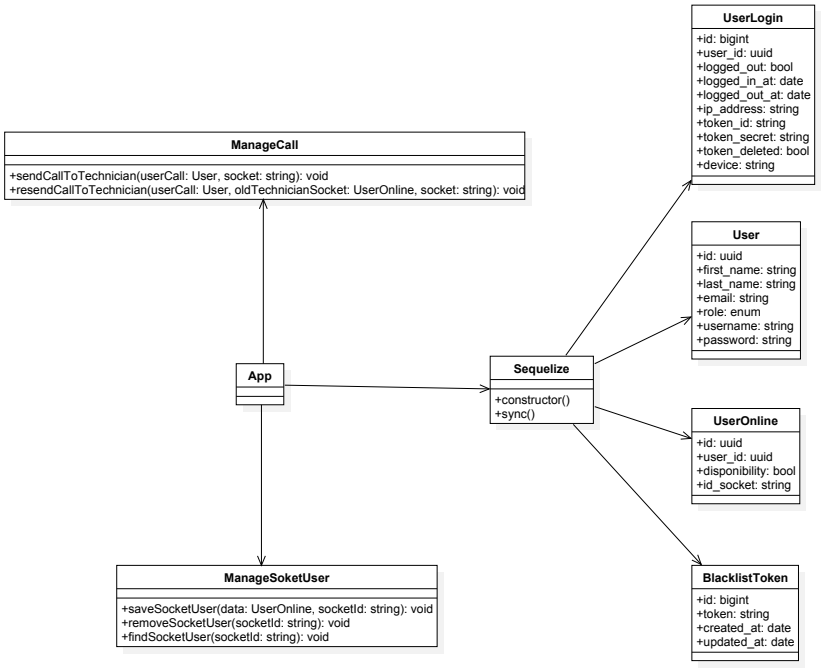

Figura 12: Diagramma delle classi Nodo Server

6 Diagrammi di sequnza

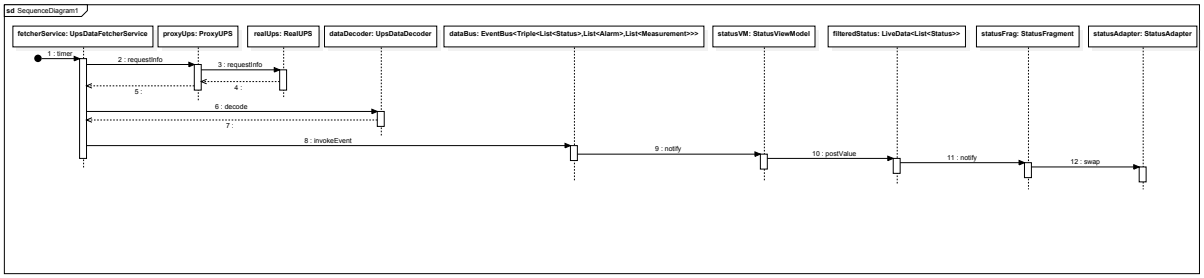

Figura 13: Primo Diagramma di Sequenza

Nel primo diagramma di sequnza viene mostrato come funziona l'aggiornamento dei dati provenieni dell'Ups sulla vista, in particolare la vista relativa aglli stati dell'Ups. Le varie operazioni che si succedono sono:

1. il fetcherService contiene un timer per eseguire le richieste ogni determinato periodo di tempo;
2. il fetcherService richiede al ProxyUps i dati relativi all'ups;
3. Il ProxyUps esegue una richiesta (tramite Socket) al vero ups;
4. Il vero ups ritorna i dati al ProxyUps;
5. Il ProxyUps ritorna al fetcherSevice i dati ancora in formato grezzo (bytes);
6. Il fetcherService esegue una chiamata a dataDecoder che si occupa di tradurre i dati grezzi in delle Liste di dati comprensibili(Status,Alarms e Measurement);
7. Le liste di dati vengono ritornate al fetcherService
8. Una volta che il fetcherServie ottiene i dati decodificati aggiorna il valore contenuto all'interno del dataBus;
9. statusVM che era iscritta al dataBus riceve l'evento con il nuovo valore;
10. statusVM aggiorna di conseguenza il liveData filteredStatus contenente la lista degli stati da visualizzare;
11. StatusFrag che osservava filteredStatus viene notifica dell'aggiornamento;
12. statusFrag cambia di conseguenza la lista mostrata a schermo con quella nuova.

Nel secondo diagramma viene invece mostrato il percorso dei dati dell'Ups dall'App del VirtualDisplay al RemoteSupport, infatti in questo diagramma callArea, DataSetStore e StatesArea sono implementati nel Remote Support, invece i restati nel VirtualDisplay. Ecco la lista delle varie operazioni:

1. All'interno di callService vi è un timer (con frequenza minore rispetto a quello del fetchService) che indica quando mandare i dati;
2. Il callService interroga connBus;
3. connBus ritorna l'ultimo valore;

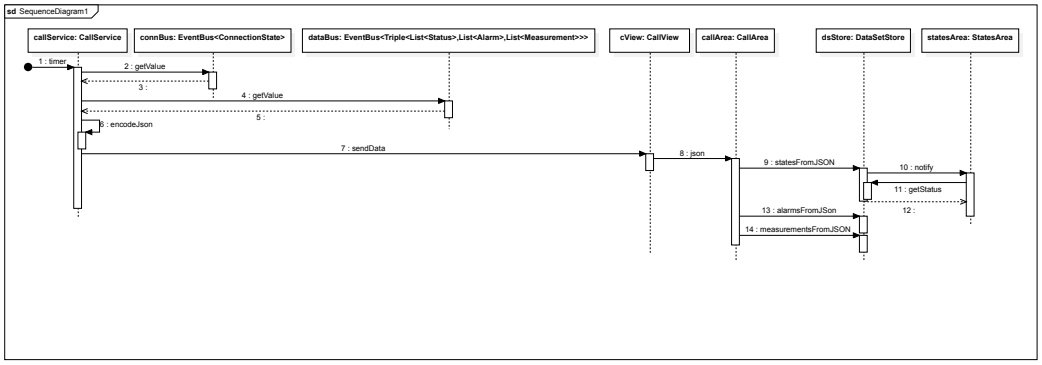

Figura 14: Secondo Diagramma di Sequenza

4. Il callService interroga dataBus;
5. dataBus ritorna l'ultimo valore;
6. Il callService tramite un encoder converte tutti le informazioni appena ricevute in un unico JSON;
7. Il callService manda alla CallView la stringa JSON da inviare al RemoteSupport;
8. Il callService, tramite la connessione webRTC, invia il JSONal RemoteSupport;
9. La callArea manda ala DatasetStore il Json da cui esttrarre e salvare i nuovi stati;
10. statesArea che osservava il DatasetStore viene notificata;
11. statesArea esegue un re-rendering e richiede i nuovi stati al DataSetStore;
12. il DataSetStore ritoran i nuovi stati;
13. La callArea manda ala DatasetStore il Json da cui esttrarre e salvare i nuovi allarmi;
14. La callArea manda ala DatasetStore il Json da cui esttrarre e salvare le nuove misurazioni.