UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare și Tehnologia Informației
SPECIALIZAREA: Tehnologia Informației

Aplicație de monitorizare a centralelor termice, pentru dispozitive mobile

LUCRARE DE LICENȚĂ

Coordonator științific
ș.l.dr.ing. Robert Gabriel Lupu

Absolvent
Dragoș Popa

Iași, 2015

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII LUCRĂRII DE LICENȚĂ

Subsemnatul(a) DRAGOȘ POPA,
legitimat(ă) cu BI seria MX nr. 924878, CNP 1920902226703
autorul lucrării APLICAȚIE DE MONITORIZARE A CENTRALELOR TERMICE, PENTRU DISPOZITIVE MOBILE

elaborată în vederea susținerii examenului de finalizare a studiilor de licență organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi" din Iași, sesiunea IULIE a anului universitar 2015, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi" din Iași (Manualul Procedurilor, UTI.POM.02 – Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data Semnătura

Cuprins

Capitolul 1. Introducere....1
Capitolul 2. Fundamentare teoretică și documentare bibliografică....2
2.1. Java....2
2.2. XML....2
2.3. Android....3
2.3.1. Sistemul de operare....3
2.3.2. Dezvoltarea aplicațiilor....5
2.4. Bluetooth....8
Capitolul 3. Proiectarea aplicației....10
3.1. Arhitectura sistemului....10
3.2. Componente software....11
3.2.1. Fire de execuție....12
3.2.2. Modulele aplicației....14
3.2.3. Modulul de comunicații Bluetooth....17
3.2.4. Modulul protocol de parsare....18
3.2.4.1. Împachetarea datelor....18
3.2.4.2. StreamParser....24
3.2.4.3. MC1XPackProcesser....25
3.2.4.4. MC1XData....26
3.2.5. Modulul interfață grafică....26
Capitolul 4. Implementarea aplicației....28
4.1. Activitatea principală și firele de execuție....28
4.2. Conexiunea Bluetooth....29
4.3. Parsarea și reprezentarea datelor....30
4.4. Descrierea funcționării aplicației....32
Capitolul 5. Testarea aplicației....37
Concluzii....41
Bibliografie....42
Anexe....43
Anexa 1. Diagramele UML ale claselor....43
Anexa 2. Codul sursă....46

Aplicație de monitorizare a centralelor termice, pentru dispozitive mobile

Dragoș Popa

Rezumat

Proiectul își propune dezvoltarea unei aplicații pentru dispozitive mobile cu sistem de operare Android, care să monitorizeze comportamentul, stările și mărimile implicate în procesele unei centrale termice. Vor fi reprezentate atât valorile numerice aferente mărimilor, cât și evoluțiile acestora, prin intermediul unor reprezentări grafice, ce vor fi construite pe măsură ce sunt primite date de la unitatea de control a centralei termice.

Unele din mărimile ce sunt extrase și monitorizate în cadrul acestei aplicației sunt:

- temperatura pe circuitul de termoficare – tur și retur.
- temperatura măsurată în tancul de apă – pentru doi senzori diferiți.
- temperatura măsurată a gazelor de evacuare.
- viteza de rotație a ventilatorului.
- temperatura la exterior
- presiunea măsurată în circuitul de încălzire.
- tensiunea de ionizare a gazelor de evacuare.

Tipul de comunicație utilizat pentru transmiterea datelor este reprezentat de standardul Bluetooth, fiind prezent pe aproximativ toate dispozitivele cu sistem de operare Android din ziua de azi. Datele primite de la centrală sunt împachetate cu un anumit protocol, fiind necesară identificarea tipului de pachet și procesarea corespunzătoare a acestuia în vederea extragerii informațiilor. Transmiterea datelor este una serială, deci fiecare pachet de date primit este reconstruit conform protocolului cu care a fost împachetat. Pachetul este încadrat de delimitatori și urmat de codul CRC.

După primirea unui set complet de pachete, informațiile extrase din acestea sunt reprezentate corespunzător. Stările sunt afișate sugestiv prin aprinderea unor led-uri grafice aferente stărilor curente, valorile mărimilor sunt reprezentate tabelar și, de asemenea, sunt adăugate la graficele corespunzătoare acestora. Pentru ca experiența utilizatorului să fie plăcută, acesta poate alege ce grafice vor fi afișate sau poate schimba culoarea acestora.

Proiectul cuprinde 3 mari module:

- modul de comunicații Bluetooth, care se ocupă de trimiterea unui mesaj de control și de recepția pachetelor de date
- modul de parsare al datelor conform protocolului de împachetare
- modul interfață grafică, care reprezintă valorile și graficele corespunzătoare.

Pentru partea de testare am implementat un program în .NET(C#), care va trimite datele prin Bluetooth, de la un PC către aplicația de monitorizare. Datele sunt pachete primite de la centrală într-o sesiune anterioară.

Primul capitol al lucrării va introduce și va motiva necesitatea implementării unei astfel de aplicații. În Capitolul 2, sunt prezentate o serie de fundamente necesare implementării aplicației Android, Capitolul 3 prezintă arhitectura sistemului, modulele principale și comunicația între acestea, a căror implementare este descrisă mai în detaliu în Capitolul 4, iar Capitolul 5 descrie succint modulul folosit pentru testare, dar și rezultatele obținute pe parcursul rulării aplicației dezvoltate.

Capitolul 1. Introducere

Inovația și progresul tehnologic au remodelat în continuu activitățile de zi cu zi ale oamenilor. Evoluția tot mai rapidă a tehnologiei din ultimele decenii a favorizat mai întâi dezvoltarea calculatoarelor personale, iar, mai recent, a dispozitivelor portabile ce îndeplinesc o multitudine de funcții, de la telefonie mobilă, până la navigarea pe internet și localizarea automată prin GPS.

Progresul în aria dispozitivelor mobile(telefon inteligent și tabletă) și răspândirea pe o scară tot mai largă a acestora au determinat apariția unei tot mai vaste diversități de aplicații, cum ar fi cele pentru mesagerie și e-mail, muzică, înregistrări foto-audio-video, planificare, stocare și mai ales consultare a orice fel de informații (vreme, curs valutar, știri etc.), dar și aplicații pentru controlul și monitorizarea diferitelor electronice din casă sau dintr-o clădire. Astfel, telefoanele inteligente nu mai sunt telefoane, ci mai degrabă mini-calculatoare portabile ce pot fi considerate asistenți personali, întrucât sunt capabile să răspundă oricând la aproape orice fel de întrebări sau curiozități.

Deși, Apple a fost compania care a consacrat telefoanele inteligente, succesul acestora a determinat compania Google să dezvolte sistemul de operare Android pentru dispozitive mobile și, prin urmare, și a dispozitivelor care sa îl utilizeze. Răspândirea acestora a fost mult mai mare și mai rapidă, datorită software-ului open-source și a gamei mai largi de dispozitive: de la cele mai puțin performante, dar mai accesibile, la cele cu puteri de procesare și stocare mult mai ridicate, pentru performanțe mai bune. Astfel, numărul dispozitivelor ce rulează această platformă crește considerabil de la an la an, deci aplicațiile pot avea o răspândire masivă în rândul utilizatorilor, iar aceștia vor avea acces tot mai ușor la aplicații pentru dispozitive mobile cu sistem de operare Android.

Centralele termice sunt în prezent utilizate la o scară foarte largă, deoarece furnizează o modalitate convenabilă pentru încălzirea apei și a spațiilor închise. Pentru ca apa să fie încălzită foarte repede, aceasta trece prin mai multe etape și componente ale centralei, care utilizează un anumit combustibil pentru ardere(de obicei, gaz), dar și un agent termic.

Funcționarea corespunzătoare a unei centrale este determinată de mai mulți factori măsurabili. Monitorizarea acestora ar putea furniza informații vitale pentru funcționarea corectă a centralei. Astfel, cel mai prietenos mediu pentru reprezentarea acestor date ar fi printr-o aplicație mobilă. Platforma Android este un mediu propice pentru o astfel de aplicație, întrucât oferă conectivitate printr-o multitudine de tehnologii - deci transmiterea informațiilor se poate face cu ușurință - dar și pentru că poate oferi o interacțiune ușoară, elegantă și foarte prietenoasă cu utilizatorul.

Capitolul 2. Fundamentare teoretică și documentare bibliografică

Tema acestui proiect este realizarea unei aplicații software, pentru dispozitive mobile, care să monitorizeze comportamentul, stările și evoluția mărimilor(temperatura tancurilor de apă, temperatura circuitului de termoficare, tensiunea de ionizare a gazelor etc.) unei centrale termice. Monitorizarea se realizează prin intermediul unor reprezentări grafice ale evoluției valorilor primite în timp real. De asemenea, sunt reprezentate și stările centralei într-un mod cât mai sugestiv, dar și valorile numerice ale tuturor mărimilor monitorizate.

Platforma Android este un mediu ideal pentru o astfel de aplicație, întrucât oferă conectivitate printr-o multitudine de tehnologii(WiFi, Bluetooth, NFC) - deci transmiterea informațiilor se poate face cu ușurință - dar și pentru că poate oferi o interacțiune ușoară, elegantă și foarte prietenoasă cu utilizatorul. De asemenea, numărul dispozitivelor ce rulează această platformă crește considerabil de la an la an, deci aplicațiile pot avea o răspândire masivă în rândul utilizatorilor, iar aceștia vor avea acces tot mai ușor la aplicații pentru dispozitive mobile cu sistem de operare Android.

Pentru dezvoltarea aplicațiilor, Google pune la dispoziție și oferă suport pentru mediul de programare *Android Studio* și unelte care să ajute în procesul de dezvoltare. Aplicațiile pot fi create în acest mediu cu limbajul de programare Java și limbajul extensibil de marcare XML, utilizat pentru elementele grafice și resurse. Pe lângă acestea, pentru dezvoltarea aplicației de față este utilizat protocolul de comunicații Bluetooth pentru transmiterea datelor, întrucât acesta este prezentat pe aproximativ toate dispozitivele cu sistem de operare Android. De asemenea, Bluetooth utilizează puțină energie în comparație cu celelalte tehnologii fără fir, în special pentru transmiterea datelor la distanțe nu foarte mari.

2.1. Java

Java este o tehnologie inovatoare lansată de compania Sun Microsystems în 1995, care a avut un impact remarcabil asupra întregii comunități a dezvoltatorilor de software, impunându-se prin calități deosebite cum ar fi simplitate, robustețe și nu în ultimul rând portabilitate. Denumită inițial OAK, tehnologia Java este formată dintr-un limbaj de programare de nivel înalt pe baza căruia sunt construite o serie de platforme destinate implementării de aplicații pentru toate segmentele industriei software.[1]

Limbajul împrumută o mare parte din sintaxă de la C și C++, dar are un model al obiectelor mai simplu și prezintă mai puține facilități de nivel jos. Un program Java compilat, corect scris, poate fi rulat fără modificări pe orice platformă care e instalată o mașină virtuală Java (engleză Java Virtual Machine, prescurtat JVM). Acest nivel de portabilitate (inexistent pentru limbaje mai vechi cum ar fi C) este posibil deoarece sursele Java sunt compilate într-un format standard numit cod de octeți (engleză byte-code) care este intermediar între codul mașină (dependent de tipul calculatorului) și codul sursă.[2]

Cele mai multe aplicații distribuite sunt scrise în Java, iar noile evoluții tehnologice permit utilizarea sa și pe dispozitive mobile gen telefon, agenda electronică, palmtop etc. În felul acesta se creează o platformă unică, la nivelul programatorului, deasupra unui mediu eterogen extrem de diversificat. Acesta este utilizat în prezent cu succes și pentru programarea aplicațiilor destinate intranet-urilor.

2.2. XML

Extensible Markup Language (XML) este un meta-limbaj de marcare recomandat de Consorțiul Web pentru crearea de alte limbaje de marcare, cum ar fi XHTML, RDF, RSS,

MathML, SVG, OWL etc. Aceste limbaje formează familia de limbaje XML.

Meta-limbajul XML este o simplificare a limbajului SGML (din care se trage și HTML) și a fost proiectat în scopul transferului de date între aplicații pe internet, descriere structură date.

XML este acum și un model de stocare a datelor nestructurate și semi-structurate în cadrul bazelor de date native XML.

Datele XML pot fi utilizate în limbajul HTML, permit o identificare rapidă a documentelor cu ajutorul motoarelor de căutare. Cu ajutorul codurilor javascript, php etc. fișierele XML pot fi înglobate în paginile de internet, cel mai elocvent exemplu este sitemul RSS care folosește un fișier XML pentru a transporta informațiile dintr-o pagină web către mai multe pagini web.

Câteva avantaje ale XML sunt:

- extensibilitate (se pot defini noi indicatori dacă este nevoie)
- validitate (se verifică corectitudinea structurală a datelor)
- oferă utilizatorilor posibilitatea de a-și reprezenta datele într-un mod independent de aplicație
- XML este simplu și accesibil (sunt fișiere text create pentru a structura, stoca și a transporta informația)
- poate fi editat, modificat foarte ușor (necesită doar un editor text simplu precum notepad, wordpad etc.)

2.3. Android

2.3.1. Sistemul de operare

Android este o platformă software și un sistem de operare pentru dispozitive și telefoane mobile bazată pe nucleul Linux, dezvoltată inițial de compania Google, iar mai târziu de consorțiul comercial Open Handset Alliance. Android permite dezvoltatorilor să scrie cod gestionat în limbajul Java, controlând dispozitivul prin intermediul bibliotecilor Java dezvoltate de Google. Aplicațiile scrise în C și în alte limbaje pot fi compilate în cod mașină ARM și executate, dar acest model de dezvoltare nu este sprijinit oficial de către Google.

Arhitectura sistemului de operare Android este prezentată în Figura 2.1 și este reprezentată de următoarele componente:

1. Kernel-ul Linux (cu unele modificări) conține driver-ele pentru diferitele componente hardware (ecran, cameră foto, tastatură, antenă WiFi, memorie flash, dispozitive audio), fiind responsabil cu gestiunea proceselor, memoriei, perifericelor (audio/video, GPS, WiFi), dispozitivelor de intrare/ieșire, rețelei și a consumului de energie.
2. Bibliotecile (user-space) conțin codul care oferă principalele funcționalități ale sistemului de operare Android, făcând legătura între kernel și aplicații. Sunt incluse aici motorul open-source pentru navigare WebKit, biblioteca FreeType pentru suportul seturilor de caractere, baza de date SQLite utilizată atât ca spațiu de stocare cât și pentru partajarea datelor specifice aplicațiilor, biblioteca libc (Bionic), biblioteca de sistem C bazată pe BSD și optimizată pentru dispozitive mobile bazate pe Linux, biblioteci pentru redarea și înregistrarea de conținut audio/video (bazate pe OpenCORE de la PacketVideo), biblioteci SSL pentru asigurarea securității pe Internet și Surface Manager, bibliotecă pentru controlul accesului la sistemul de afișare care suportă 2D și 3D. Aceste biblioteci nu sunt expuse prin API, reprezentând detalii de implementare Android.
3. Motorul Android este reprezentat de:
    1. un set de biblioteci de bază care permit utilizatorilor să dezvolte aplicații Android folosind limbajul de programare Java; acestea includ acces la funcțiile telefonului

(telefonie, mesaje, resurse, locații), interfața cu utilizatorul, furnizori de conținut și gestiunea pachetelor (instalare, securitate)

2. mașina virtuală (Java) Dalvik este optimizată special pentru Android (dispozitive mobile alimentate de o baterie, resurse de procesare și de memorie limitate, sistem de operare fără swap). Arhitectura sa se bazează pe regiștri, fiind echipată cu un compilator JIT (just-in-time), executabilul obținut putând fi modificat când este instalat pe dispozitivul mobil. Întrucât este utilizată o bibliotecă proprie ce pornește de la un subset al implementării Java realizată de Apache Harmony, nu sunt conținute pachetele pentru AWT / Swing, imprimare sau alte componente speciale. Prin urmare, deși se poate utiliza versiunea curentă de Java pentru dezvoltarea aplicației, facilitățile ce pot fi folosite sunt limitate aproximativ la versiunea 6. Bytecode-ul este compilat în fișiere .dex - Dalvik Executable (în loc de .class), datele duplicate provenind din clase diferite (șiruri de caractere, alte constante) fiind incluse o singură dată, motiv pentru care un astfel de fișier necomprimat va avea o dimensiune mai mică decât o arhivă .jar (comprimată). De asemenea, se permite ca fiecare aplicație Android să ruleze în procesul propriu, într-o instanță a mașinii virtuale Dalvik.

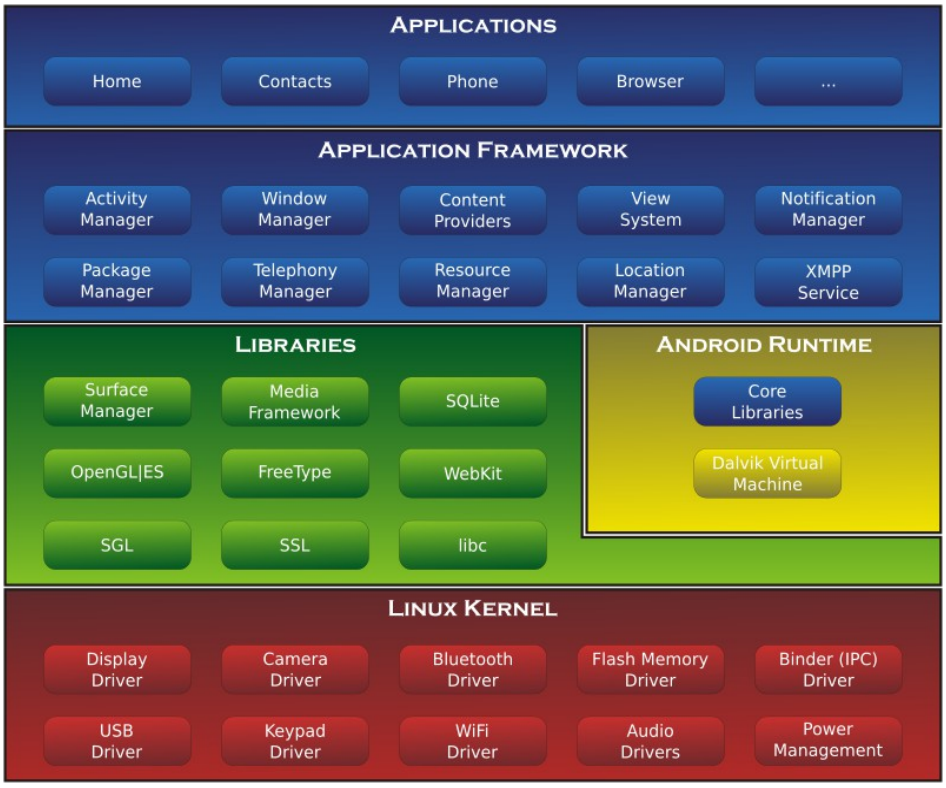


Figura 2.1: Arhitectura software a platformei Android

4. Cadrul pentru Aplicații expune diferitele funcționalități ale sistemului de operare Android către programatori, astfel încât aceștia să le poată utiliza în aplicațiile lor.
5. La nivelul de aplicații se regăsesc atât produsele împreună cu care este livrat dispozitivul mobil (Calculator, Camera, Contacts, Clock, FM Radio, Music Player, S Note, S Planner, Video Player, Voice Recorder), cât și produsele instalate de pe Play Store sau cele dezvoltate de programatori.

Unele din principalele caracteristici ale acestei platforme sunt:

- este adaptabilă la configurații mai mari, VGA, biblioteci grafice 2D, biblioteci grafice 3D bazate pe specificația OpenGL ES 1.0 și configurații tradiționale smartphone.

- Software-ul de baze de date SQLite este utilizat în scopul stocării datelor.
- Android suportă tehnologii de conectivitate incluzând GSM/EDGE, CDMA, EV-DO, UMTS, Bluetooth și Wi-Fi.
- SMS și MMS sunt formele de mesagerie instant disponibile, inclusiv conversații de mesaje text.
- Navigatorul de web disponibil în Android este bazat pe platforma de aplicații open source WebKit.
- Software-ul scris în Java poate fi compilat în cod mașină Dalvik și executat de mașina virtuală Dalvik, care este o implementare specializată de mașină virtuală concepută pentru utilizarea în dispozitivele mobile, deși teoretic nu este o Mașină Virtuală Java standard.
- Android acceptă următoarele formate media audio/video/imagine: MPEG-4, H.264, MP3, AAC, OGG, AMR, JPEG, PNG, GIF.
- poate utiliza camere video/foto, touchscreen, GPS, accelerometru, și grafică accelerată 3D.
- Include un emulator de dispozitive, unelte de depanare, profilare de memorie și de performanță, un plug-in pentru mediul de dezvoltare Eclipse.
- Similar cu App Store-ul de pe iPhone, piața Android este un catalog de aplicații care pot fi descărcate și instalate pe hardware-ul țintă prin comunicație fără fir, fără a se utiliza un PC. Inițial au fost acceptate doar aplicații gratuite. Aplicații contra cost sunt disponibile pe Piața Android începând cu 19 februarie 2009.
- Android are suport nativ pentru multi-touch, dar această funcționalitate este dezactivată (posibil pentru a se evita încălcarea brevetelor Apple pe tehnologia touch-screen).O modificare neoficială, care permite multi-touch a fost dezvoltată.[3]

2.3.2. Dezvoltarea aplicațiilor

Pentru a înțelege arhitectura unei aplicații mobile Android este nevoie de un minim de cunostințe cu privire la conceptele cheie ale aplicațiilor Android. Înțelegerea acestor elemente va permite programatorului sa controleze:

- componentele aplicației;
- ciclul de viață al aplicației;
- resursele aplicației.

Principalele caracteristici ale unei aplicații Android sunt:

- este o aplicație Java executată de catre o mașină virtuală Dalvik; mașina virtuală (VM) Dalvik execută fișiere .dex, obținute din fișiere .class Java;
- este distribuită într-un pachet Android (Android Package), care este un fișier .apk;
- este executată de către sistemul de operare Android (un Linux multi-utilizator), într-un sandbox (cutie de nisip); fiecare aplicație are un ID utilizator unic și resursele sale pot fi accesate numai de către acel utilizator; fiecare aplicație rulează în procesul Linux propriu și în propria instanță de mașină virtuală; în ciuda acestui fapt, două aplicații diferite pot avea același ID utilizator pentru a partaja resursele lor;
- operațiile critice (acces la Internet, citire/scriere date de contact, monitorizare SMS, acces la modulul GPS), pot fi restrictionate sau se poate solicita permisiunea utilizatorului, utilizând fișierul manifest de configurare a aplicației, AndroidManifest.xml;
- o aplicație Android înseamnă una sau mai multe activități (o activitate poate fi asociată cu un ecran sau o fereastră, dar este mai mult decât atât) și procesul Linux; ciclul de viață al unei activități nu este legat de ciclul de viață al procesului; activitatea poate rula chiar în

cazul în care procesul său nu mai există (acesta situație este diferită de aplicațiile C, C++ sau Java în cazul cărora o aplicație este un proces);

- o aplicație poate executa o componentă (activitate) din altă aplicație; componenta (activitatea) este executată de procesul de care aparține;
- o aplicație Android nu are un punct unic de intrare, cum ar fi metoda main() in aplicațiile Java, C sau C++.[4]

Cele mai importante componente ale unei aplicații Android sunt: Activity(Activitatea), Intent(Intenție), Service(Serviciu), Content provider (Furnizor sau manager de conținut), Broadcast Receiver.

Activity(Activitatea) are următoarele caracteristici:

- reprezintă o interfață cu utilizatorul, fereastră sau formular;
- o aplicație Android poate avea una sau mai multe activități; de exemplu o aplicație de tip Agenda poate avea o activitate pentru a gestiona contactele, o activitate pentru a gestiona intâlniri și una pentru a edita o intrare in agenda;
- fiecare Activitate are propriul său ciclu de viață, independent de ciclul de viață al procesului asociat aplicației;
- fiecare activitate are propria stare și datele acesteia pot fi salvate sau restaurate;
- activitățile pot fi pornite de aplicații diferite (dacă este permis);
- are un ciclu de viață complex deoarece aplicațiile pot avea activități multiple și doar una este in prim-plan; utilizând managerul de activități, sistemul Android gestionează o stivă de activități care se găsesc în diferite stări (pornire, in execuție, întreruptă, oprită, distrusă);
- în SDK, Activitatea este implementată folosind o subclasă a clasei Activity care extinde clasa Context;

Intent(Intenția) are următoarele caracteristici:

- reprezintă o entitate folosit pentru a descrie o operațiune care urmează să fie executată; este un mesaj transmis către o altă componentă pentru a anunța o operațiune;
- oarecum similar cu conceptul de event-handler din .NET sau Java.;
- un mesaj asincron utilizat pentru a activa activități sau servicii;
- gestionată de o instanță a clasei Intent;[5]

Activitatea este una dintre cele mai importante componente (alături de servicii și broadcast receivers) ale unei aplicații Android deoarece este strâns legată de interfața cu utilizatorul. O activitate este utilizată pentru a gestiona interfața cu utilizatorul și este echivalentă cu fereastra sau formularul din aplicațiile desktop.

Înțelegerea modului în care se controlează activitatea vă permite să:

- utilizați ciclul de viață al activității pentru a crea, vizualiza, utiliza și a opri activitățile;
- salvați datele utilizatorului înainte ca activitatea să fie oprită și le restaurați atunci când activitatea este readusă în prim-plan;
- creați aplicații cu mai multe formulare sau activitați;

Ciclul de viata al unei Activitati descrie starea in care o activitate poate fi la un moment dat:

- *Running* - Activitatea a fost creată (onCreate()), pornită (onStart()) și este afișată pe ecranul aparatului; în cazul în care activitatea a mai fost utilizată și aplicația a salvat starea acesteia (onSaveInstanceState()), activitatea este reluată din acel punct (onRestoreInstanceState() si onResume()); în această stare utilizatorul interacționeaza cu activitatea prin intermediul interfeței dispozitivului (tastatura, touchscreen, display);
- *Paused* - Activitatea pierde prim-planul (onPause()), deoarece o altă activitate este

executată, cum ar fi o fereastră de dialog; de asemenea, în cazul în care aparatul intră în modul sleep, activitatea este oprită temporar; activitatea își poate relua execuția (onResume()) și este plasată înapoi în prim-plan;

- *Stopped* – Activitatea nu mai este în uz și pentru că este oprită, (onStop()) nu este vizibilă; pentru a fi reactivată (ea deja există), activitatea trebuie să fie repornită (onRestart() și onStart()) și reluată (onResume());
- *Destroyed* – Activitatea este distrusă (onDestroy()) și memoria s-a eliberat, deoarece nu mai este necesară sau sistemul are nevoie de memorie suplimentară pentru rutinele proprii sau pentru alte activități; deoarece managementul memoriei este un aspect important pentru sistemul de operare Linux al dispozitivului mobil, procesul care găzduiește o activitate întreruptă, oprită sau distrusă, poate fi terminat pentru a elibera memorie pentru noi activități; doar procesele ce gestionează activități ce ruleaza sunt protejate;[6]

În Figura 2.2, poate fi observat ciclul de viață al unei activități.

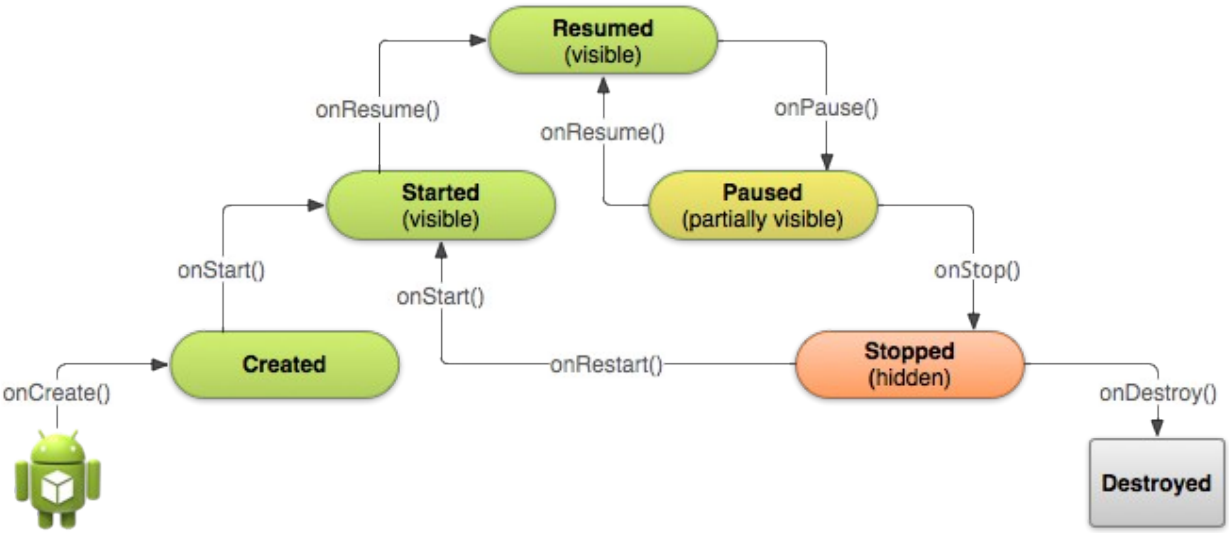


Figura 2.2: Ciclul de viață a unei activități

- onCreate(Bundle) – apelată când activitatea este creată; folosind argumentul metodei de tip Bundle există posibilitatea să restabiliți starea activității, care a fost salvată într-o sesiune anterioară; după ce activitatea a fost creată, va fi pornită (onStart());
- onStart() – apelată in cazul in care activitatea urmează sa fie afișată; din acest punct, activitatea poate veni in prim-plan (onResume()) sau ramane ascunsă în fundal (onStop());
- onRestoreInstanceState(Bundle) – apelată în cazul în care activitatea este inițializată cu datele dintr-o stare anterioară, ce a fost salvată; in mod implicit, sistemul restaurează starea interfeței cu utilizatorul (starea controalelor vizuale, poziția cursorului, etc.)
- onResume() – apelată când activitatea este vizibilă iar utilizatorul poate interacționa cu aceasta; din această stare, activitatea poate fi plasată în fundal, devenind întreruptă (onPause());
- onRestart() – apelată în cazul în care activitatea revine în prim-plan dintr-o stare oprită (stopped); după aceasta, activitatea este pornită (onStart()) din nou
- onPause() – apelată atunci când sistemul aduce în prim-plan o altă activitate; activitatea curentă este mutată în fundal și mai târziu poate fi oprită (onStop()) sau repornită și afișată (onResume()); acesta este un moment bun pentru a salva datele aplicației într-un

mediu de stocare persistent (fişiere, baze de date) deoarece după această fază activitatea poate fi terminata şi distrusă fără a se anunța acest lucru.

- onSaveInstanceState(Bundle) – apelată pentru a salva starea curentă a activității; în mod implicit, sistemul salvează starea interfeței cu utilizatorul;
- onStop() – apelată în cazul în care activitatea nu mai este utilizată şi nu mai este vizibilă deoarece o altă activitate interacționează cu utilizatorul; din acest punct, activitatea poate fi repornită (onRestart()) sau distrusă (onDestroy());
- onDestroy() – apelată în cazul în care activitatea este distrusă, iar memoria sa eliberată; acest lucru se poate întâmpla în cazul în care sistemul necesită mai multă memorie sau dacă programatorul termină explicit activitatea apelând metoda finish() din clasa Activity.[7]

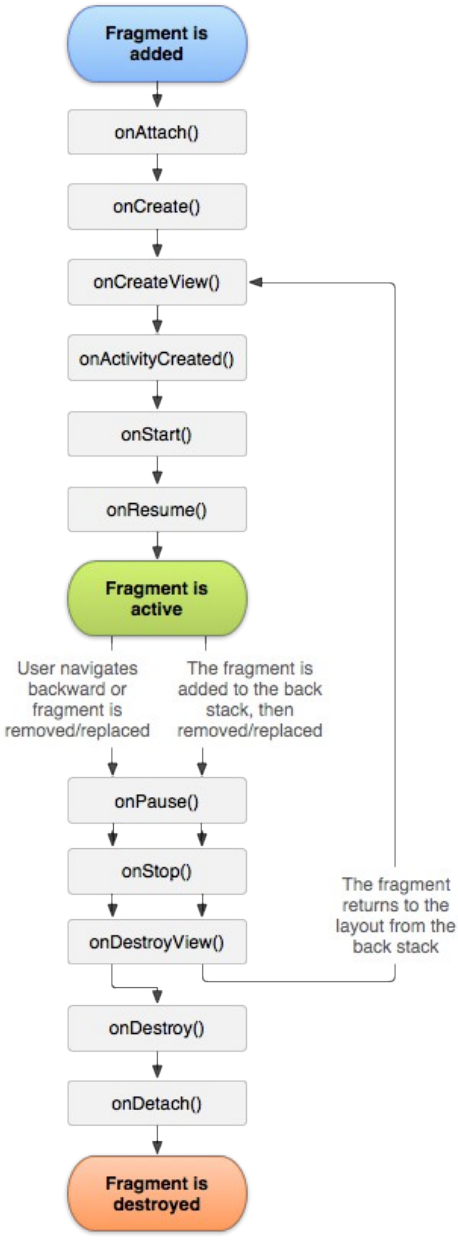


Figura 2.3: Ciclul de viață al unui fragment

Un fragment(Fragment) reprezintă un comportament sau o porțiune a unei activități. Se pot grupa mai multe fragmente într-o singură activitate pentru a construi o interfață cu mai multe panouri vizuale. Fragmentul are propriul lui ciclu de viață şi se poate adăuga şi îndepărta cât timp activitatea rulează. Se poate spune că este un fel de activitate care se poate reutiliza. În Figura 2.3, se poate observa ciclul de viață al unui fragment.[8]

2.4. Bluetooth

Bluetooth este un set de specificații (un standard) pentru o rețea personală (engleză: personal area network, PAN) fără fir (wireless), bazată pe unde radio.Tehnologia Bluetooth a fost creată în 1994.

Printr-o rețea Bluetooth se poate face schimb de informații între diverse aparate precum telefoane mobile, laptop-uri, calculatoare personale, imprimante, camere foto şi video digitale sau console video prin unde radio criptate (sigure) şi de rază mică, desigur numai dacă aparatele respective sunt înzestrate şi cu Bluetooth.

Aparatele care dispun de Bluetooth comunică între ele atunci când se află în aceeaşi rază de acțiune. Ele folosesc un sistem de comunicații radio, aşa că nu este nevoie să fie poziționate față în față pentru a transmite; dacă transmisia este suficient de puternică, ele pot fi chiar şi în camere diferite.

O caracteristică cheie a Bluetooth este aceea de a permite dispozitivelor realizate de diverşi producători să lucreze împreună. Pentru acest scop, Bluetooth nu defineşte doar un sistem Radio, ci şi o stivă de protocoale pentru ca aplicațiile respective să poată sesiza prezența altor dispozitive Bluetooth, să descopere ce servicii pot acestea oferi şi să utilizeze aceste servicii.

Stiva de protocoale este definită ca o serie de straturi, deşi unele caracteristici nu pot fi delimitate ca

aparţinând unui anumit strat. În Figura 2.3 este evidenţiat acest aspect.

Profilurile Bluetooth ghidează aplicaţiile în utilizarea stivei de protocoale Bluetooth.

- TCS (Telephony Control Protocol Specification) oferă servicii telefonice.
- SDP (Service Discovery Protocol) lasă dispozitivele Bluetooth să descopere ce servicii suportă celelalte dispozitive.
- RFCOMM oferă o interfaţă serială asemănătoare cu RS232.
- L2CAP multiplexează date de la straturile superioare şi converteşte dimensiunile pachetelor informaţionale, după necesităţi.
- HCI manipulează comunicaţiile între modulul Bluetooth şi aplicaţia gazdă.
- LM controlează şi configurează legăturile cu alte dispozitive.
- BB/LC controlează legăturile fizice prin radio, asamblează pachetele şi controlează salturile în frecvenţă.
- Stratul Radio modulează şi demodulează datele pentru transmisia şi recepţia aeriană. [9]

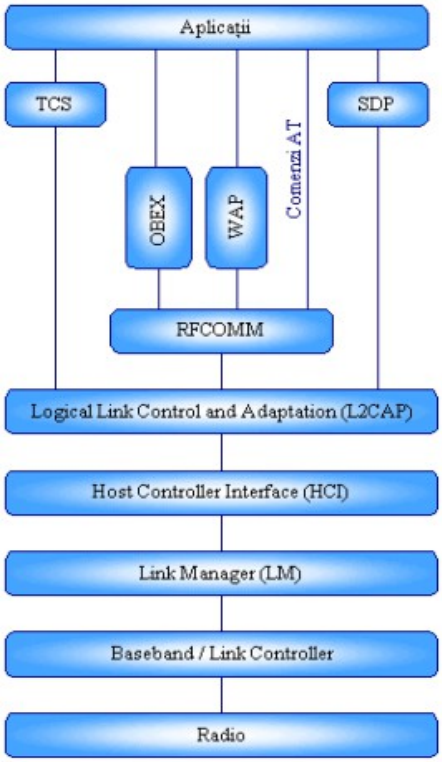


Figura 2.4: Stiva de protocoale care compun standardul Bluetooth

Capitolul 3. Proiectarea aplicației

Platforma Android este disponibilă pe o gamă variată de dispozitive, numărul sistemelor care o utilizează crescând de la an la an. Dintre dispozitivele care utilizează Android, majoritatea sunt portabile, oferă o interacțiune rapidă și ușoară pentru utilizator, și suportă conectivitatea (Wi-Fi, Bluetooth, NFC). Toate acestea fac dintr-un dispozitiv cu sistem de operare Android, un mediu prielnic pentru a primi și a trimite date, pe care sistemul să le gestioneze, să le filtreze, interprezeteze și pe care apoi să le pună la dispoziția utilizatorului într-o manieră eficientă și prietenoasă.

Aplicația *Thermal Monitor* este dezvoltată pentru dispozitive mobile ce au ca sistem de operare Android, cu versiunea minimă 4.0.3. De asemenea, dispozitivul trebuie să aibă suport pentru comunicații prin protocolul Bluetooth pentru transmiterea mesajelor și a datelor. Acest lucru nu este, însă, o problemă majoră, deoarece majoritatea dispozitivelor mobile cu sistem de operare Android, au suport pentru Bluetooth. Totuși, acest aspect nu trebuie neglijat, deoarece aplicația este dependentă de acest tip de comunicație.

3.1. Arhitectura sistemului

Aplicația dezvoltată are în vedere câteva funcții principale ce sunt puse la dispoziția utilizatorului, scopul final fiind monitorizarea comportamentului centralei termice. Aceste funcții sunt schițate cu ajutorul unei diagrame a cazurilor de utilizare ce poate fi observată în Figura 3.1 de mai jos.

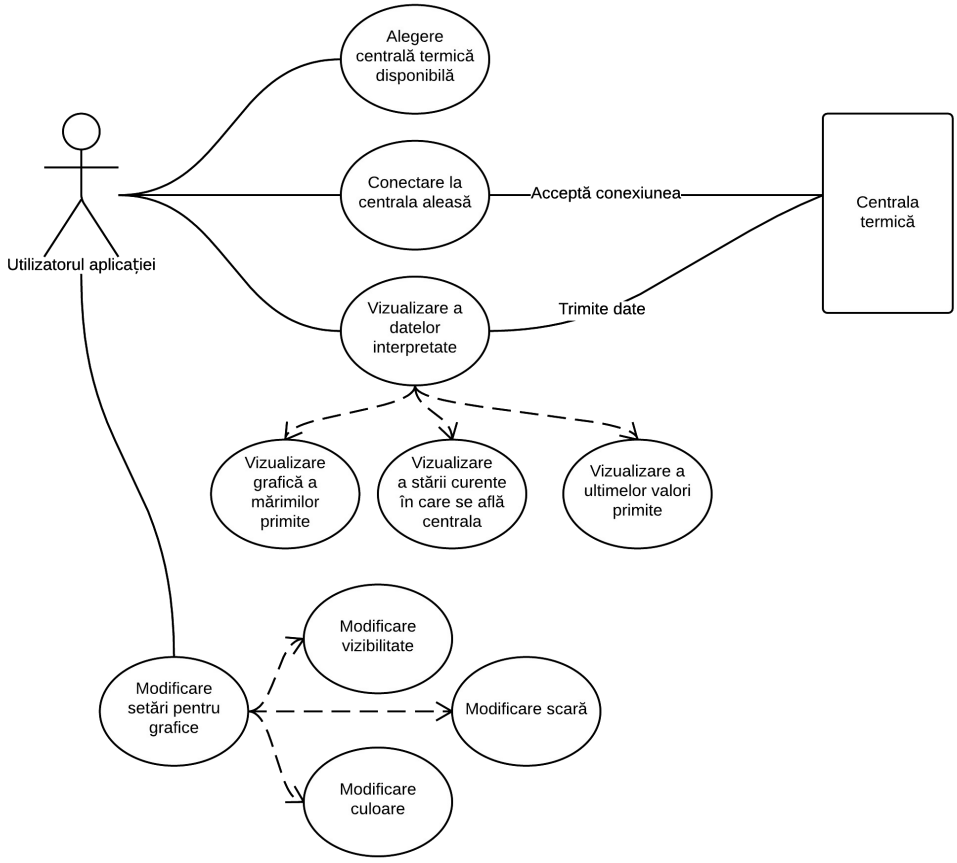


Figura 3.1: Diagrama cazurilor de utilizare pentru principalele activități

Prin urmare, utilizatorul are posibilitatea de a se conecta la o centrala termică din apropiere, iar în cazul în care există mai multe centrale disponibile, acesta o poate alege pe cea pe care va dori să o monitorizeze.

După conectare aplicația va începe să primească date de la centrală, date ce reprezintă stările în care se află aceasta în momentele respective de timp. Utilizatorul are posibilitatea de a vizualiza reprezentări grafice ale evoluției mărimilor măsurate în cadrul centralei din momentul începerii monitorizării. De asemenea, poate vedea valorile numerice curente ale acestor mărimi, cât și starea în care se află centrala sau diferite componente ale acesteia.

Sistemul este reprezentat de aplicația mobilă care monitorizează stările centralei termice. Acesta este dependent de centrala la care se conectează, în sensul că trebuie să existe o comunicație bidirecțională între cele două mari componente, așa cum se poate vedea și în Figura 3.2 de mai jos:

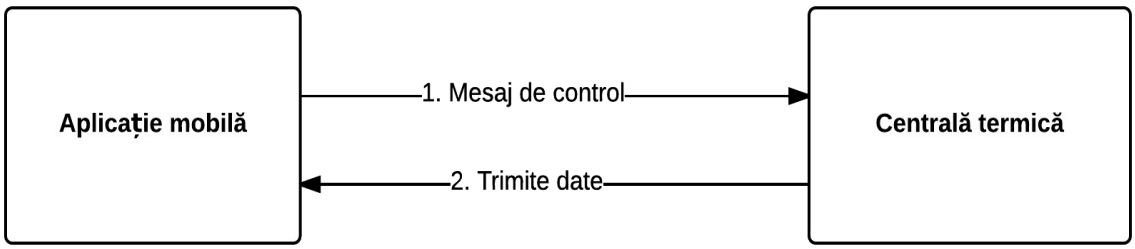


Figura 3.2: Comunicația între aplicație și centrala termică

1. *Mesaj de control* – aplicația trimite un mesaj de control către centrala termică, ce are ca scop notificarea centralei că există o conexiune care așteaptă să primeasca date despre starea acesteia. Acest mesaj trebuie transmis la un inteval de maxim 8 secunde pentru ca centrala să nu dezactiveze transmisia mesajelor către aplicația conectată.
2. *Trimitere date* – centrala termică trimite date odată la fiecare 0.5 secunde (frecvența de transmisie = 2 Hz) sub forma unor mesaje de stare împachetate cu ajutorul unui protocol de comunicație serială

3.2. Componente software

Aplicația software necesită realizarea mai multor operații și prelucrări asupra unor seturi de date primite de la un alt sistem prin intermediul unei conexiuni Bluetooth. Datorită acestor aspecte, trebuie avută în vedere realizarea următoarelor obiective:

- Comunicația prin Bluetooth și recepția datelor
- Parsarea datelor și extragerea informațiilor
- Interpretarea informațiilor
- Reprezentarea informațiilor sub formă de valori, grafice sau stări

Prin urmare, pentru a acoperi obiectivele prezentate mai sus, aplicația necesită existența următoarelor mari module (prezentate și în Figura 3.3, mai jos, sub formă unei diagrame bloc, împreună cu interacțiunea cu centrala termică):

- Modulul de comunicație Bluetooth bidirecțională, care:
  - administrează conexiunea dintre aplicație și centrală

- trimite pachete, ce reprezintă mesaje de control, către centrală
- primește date, în mod serial, pe care le trimite mai departe către modulul de parsare.

- Modulul de parsare al datelor, care respectă un anumit protocol de împachetare utilizat de centrala termică.
- Modulul interfață grafică, care se ocupă cu reprezentarea și gestionarea datelor procesate de modulul anterior.

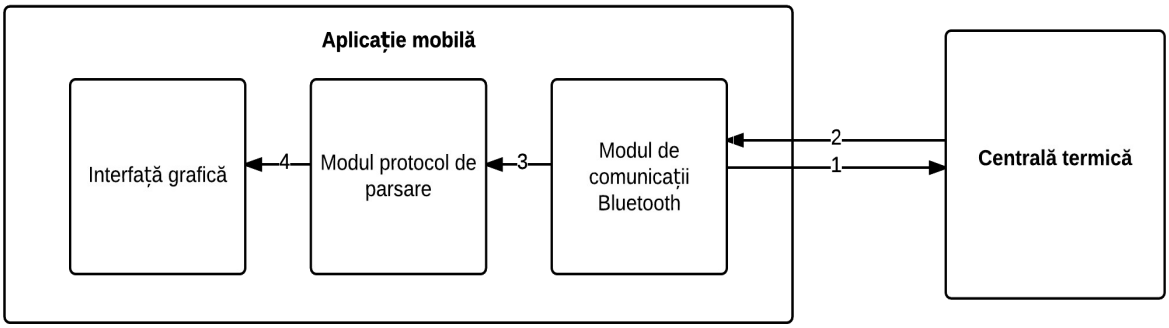


Figura 3.3: Principalele module ale aplicației - diagramă bloc

Între modulele prezentate mai sus, în Figura 3.3, există o serie de interacțiuni, după cum urmează:

1. Comunicație dinspre modulul Bluetooth al aplicației către centrala termică, alcatuită din două operații unitare:
   - Conectare la centrala termică.
   - Trimitere mesaj de stare către centrala, odată la 5 secunde.
2. Trimitere date către aplicație, în mod serial, din 500 în 500 de milisecunde.
3. Trimitere date recepționate de către modulul Bluetooth, spre modulul de parsare al datelor.
4. Trimiterea datelor procesate dinspre modulul de parsare către modul de reprezentare pe interfața grafică.

3.2.1. Fire de execuție

După cum se poate observa, există operații care depind unele de rezultatul celorlalte, însă nu ar fi eficient ca modulele care au trimis datele mai departe să le aștepte pe cele care tocmai au primit datele spre procesare, pentru a-și continua activitatea. De exemplu, modulul Bluetooth, după ce recepționează date, le trimite mai departe, iar apoi ar trebui să continue să primească datele care s-au trimis către acesta. Prin urmare, conexiunea Bluetooth nu trebuie blocată cu așteptarea altor procese. Ea trebuie să se ocupe strict de fluxul de date dintre aplicație și centrală.

De asemenea, interfața grafica nu trebuie blocată niciodată cu procesele de transmisie a datelor sau cu procesarea și parsarea acestora, de aceea fiecare modul se va executa pe fire de execuții separate.

În Figura 3.4 de mai jos este schițată activitatea fiecărui modul pe câte un fir propriu de execuție și comunicația cu celelalte module care se execută în paralel.

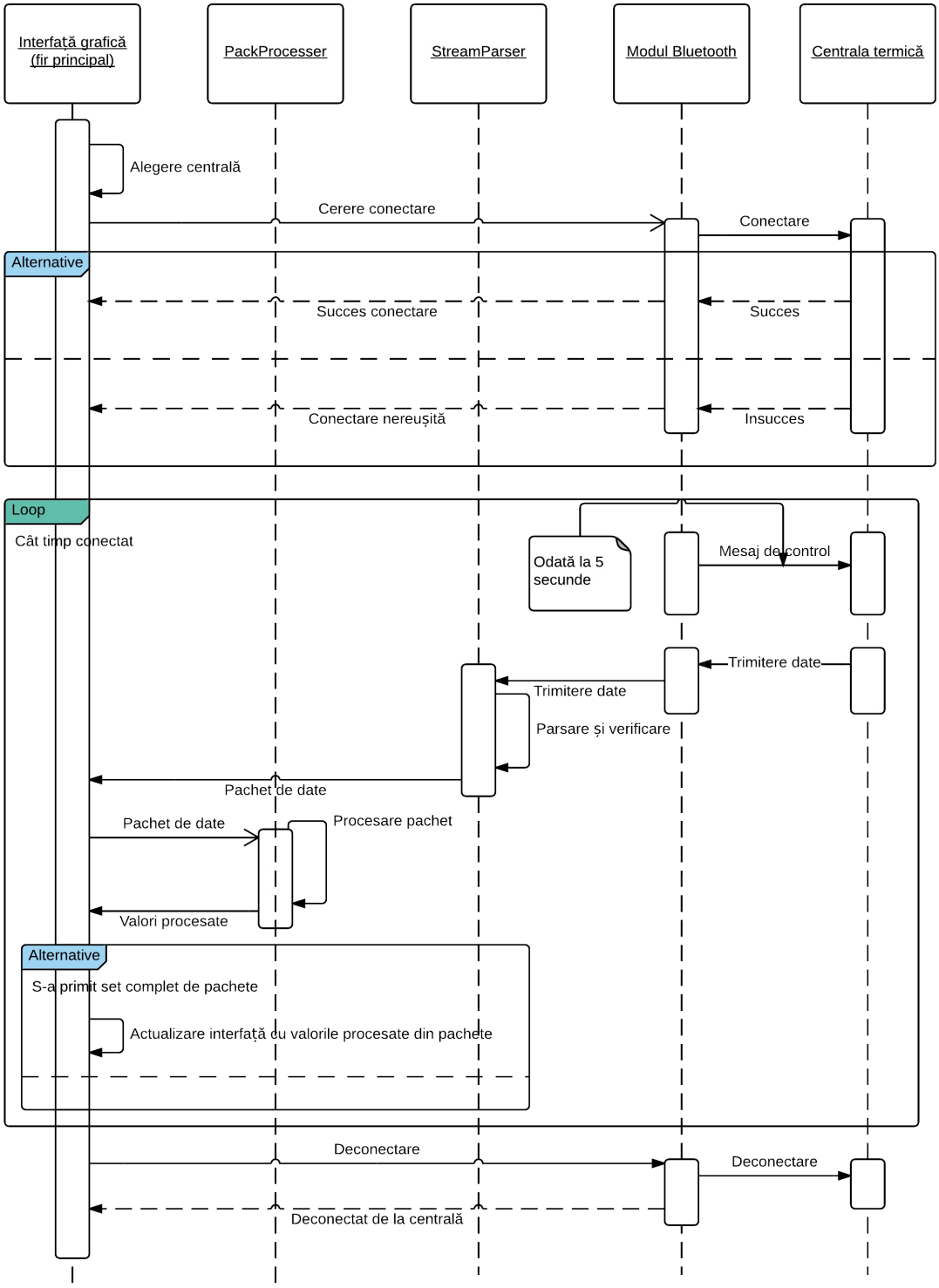


Figura 3.4: Diagrama de secvențe pentru principalele module ale aplicației

Cu toate că operațiile care implică fluxul de date sau parsarea acestora sunt gestionate de module care se execută pe alte fire execuție decât cel principal, acesta este punctul de legătură pentru aproape toate procesele care se execută în paralel, pachetele parsate ajungând tot la firul principal, de unde sunt trimise mai departe către modulul ce procesează pachetele. În urma

finalizării procesării unui set de pachete rezultă valori și date interpretabile în legătură cu centrala termică de la care sunt primite. Astfel, firul principal este notificat pentru a-și actualiza interfața cu aceste date procesate.

În Figura 3.4 de mai sus, se pot observa interacțiunile dintre firele de execuție, dar și comunicațiile, mesajele și fluxul de date ce se desfășoară între aceastea.

Conform acestei diagrame, principalele operații care se desfășoară în sistem, într-o succesiune firească de evenimente sunt:

1. De pe interfața grafică (UI[1]), care se desfășoară pe firul principal de execuție, se alege centrala termică la care se dorește conectarea.
2. De pe firul principal este inițiat modulul Bluetooth pe un fir de execuție separat de unde se încearcă a se conecta aplicația mobilă la centrala termică:
    - În caz de succes, este anunțat firul principal de faptul că s-a reușit conexiunea. Această conexiune va fi gestionată în continuare pe același fir de execuție de pe care a fost inițiată.
    - În caz de eșec, se anunță firul principal de acest fapt și, eventual se reîncearcă a se conecta. Insuccesul se poate datora faptului că centrala nu este disponibilă sau nu se află în limita razei de acțiune a protocolului Bluetooth.
3. Cât timp aplicația este conectată la centrala termică se execută următoarele acțiuni:
    - modulul conexiunii Bluetooth trimite un mesaj de control la fiecare 5 secunde, necesar pentru a înștiința centrala că această conexiune este încă activă. Dacă nu se trimite un mesaj de control timp de mai mult de 8 secunde, centrala nu va mai trimite date către această conexiune.
    - Atunci când se primesc date de la centrală, acestea sunt trimise de la modulul Bluetooth către alt fir de execuție(modulul StreamParser) care se va ocupa de parsarea acestor date, detecția fiecărui pachet în parte, verficarea pachetului prin CRC.
    - în caz de succes, are loc trimiterea pachetului mai departe către firul principal de execuție printr-o coadă comună.
    - Firul principal trimite apoi acest pachet către modulul de procesare al pachetelor(*PackProcesser*), care extrage, în mod asincron, valorile și informațiile propriu-zise.
    - Valorile procesate sunt făcute disponibile firului principal.
    - În caz că s-au primit valorile de la un set complet de pachete, firul principal este anunțat de către modulul *PackProcesser* pentru a-și actualiza interfața cu noile valori.
4. Utilizatorul poate alege să întrerupă conexiunea.

3.2.2. Modulele aplicației

Se poate observa, în diagrama de secvențe de mai sus (Figura 3.4), că modulul protocol de parsare nu mai apare. Acest lucru se datorează faptului că el a fost, de fapt, înlocuit cu principalele sale submodule, pentru a vedea că fiecare în parte are de executat o acțiune separată, pe un fir de execuție separat. Astfel, schema bloc din Figura 3.3, devine cea din Figura 3.5 de mai jos, cu următoarele interacțiuni între module:

1. Conectare, la fel ca în schema precedentă.

1 Engl. User Interface

2. Trimitere date, la fel ca în schema precedentă.
3. Trimitere date către modulul StreamParser.
4. StreamParser trimite datele împachetate și verificate către interfața grafică.
5. Modulul interfață grafică trimite pachetele primite către procesare
6. PackProcesser trimite valorile și informațiile procesate înapoi către interfața grafică.

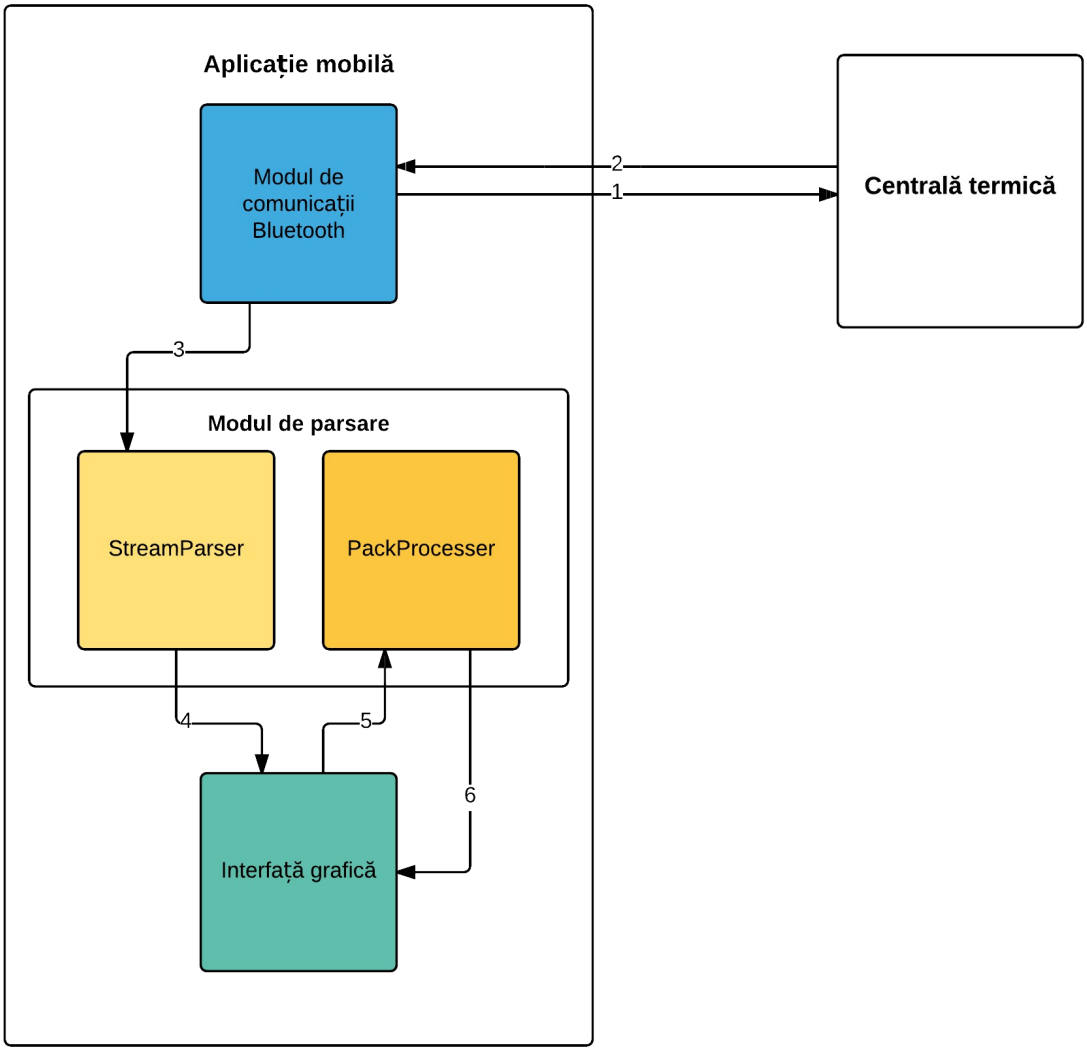


Figura 3.5: Principalele module ale aplicației - Diagramă bloc

Pentru a detalia principalele module amintite până acum, mai jos, în Figura 3.6, este reprezentată diagrama de clase, cu relațiile între ele. Fiecare modul este alcătuit din propriile clase și submodule, după cum urmează:

- Interfața grafică - este reprezentată de clasa MainActivity, împreună cu următoarele clase secundare:
  - SlidingTabsFragment, care se ocupă de gestiunea fragmentelor de tip:
    - LedsFragment – afișează stările centralei.
    - ValuesFragment – afișează valorile curente ale mărimilor monitorizate.

        - GraphFragment - afișează grafic evoluția mărimilor din momentul începerii monitorizării până la momentul actual.
    - SettingsFragment – modifică preferințe în legătură cu graficele
    - Graph – se ocupă de construirea și gestionarea unui grafic.
    - GraphView - afișează pe ecran graficele.
- Modulul de parsare – cu clasele:
    - MC1XPackProcesser – extrage informațiile din pachete.
    - MC1XDataH cu extensia MC1XData – stochează informațiile extrase din pachete, aferente centralei termice.
    - StreamParser – construiește pachetele de date din octeții primiți și le verifică dacă au fost transmise corect.
- Modulul Bluetooth – reprezentat de clasa BluetoothThread.

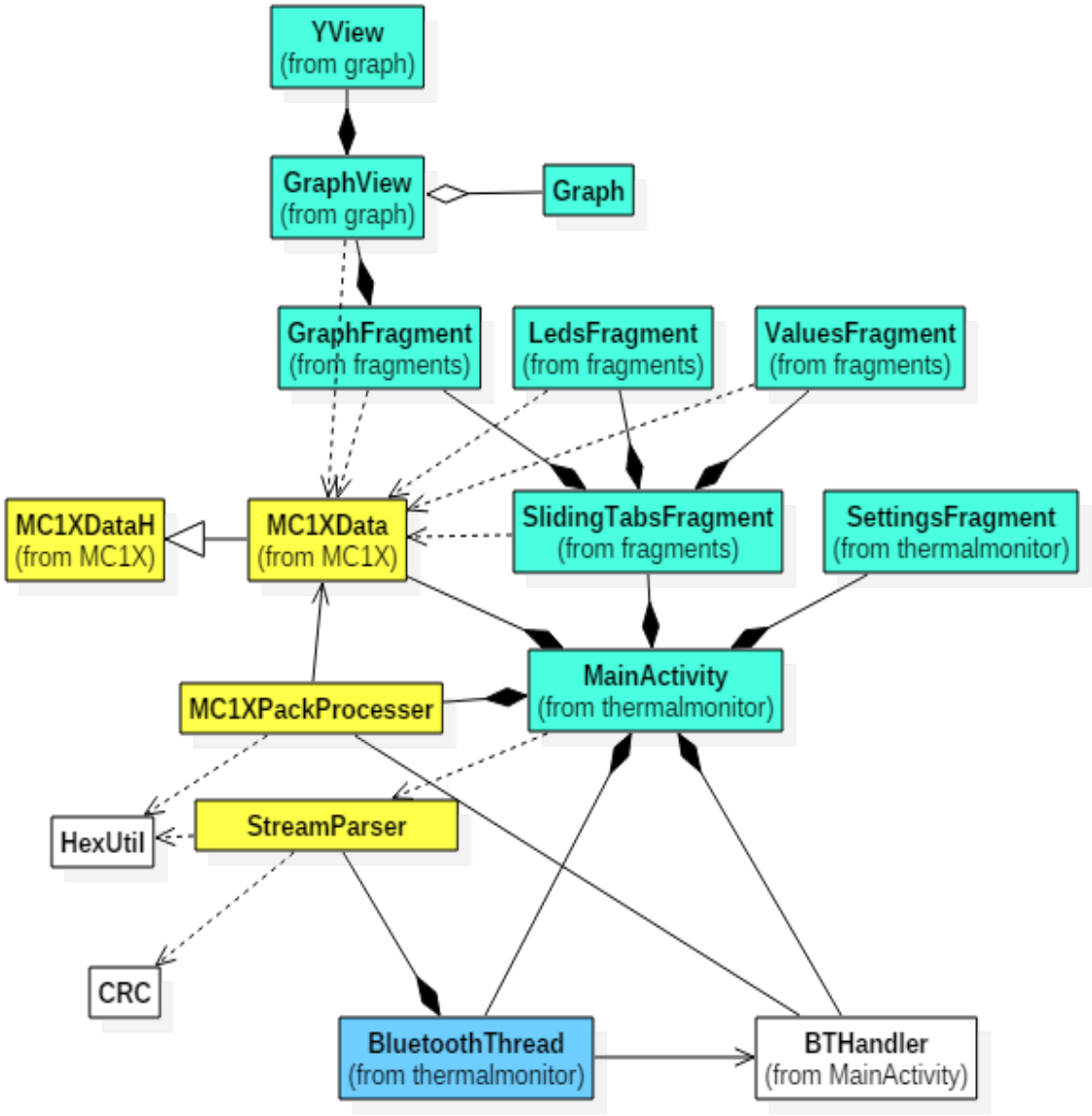


Figura 3.6: Diagrama de clase simplificată

Diagrama completă a fiecărei clase în parte se poate consulta în Anexa 1.

3.2.3. Modulul de comunicații Bluetooth

După cum reiese și din denumirea acestuia, modulul de comunicații Bluetooth se ocupă cu gestiunea conexiunii și a fluxului de date dintre aplicație și centrala termică. Acest modul este reprezentat de clasa *BluetoothThread*, care extinde clasa *Thread* din *java.lang*.

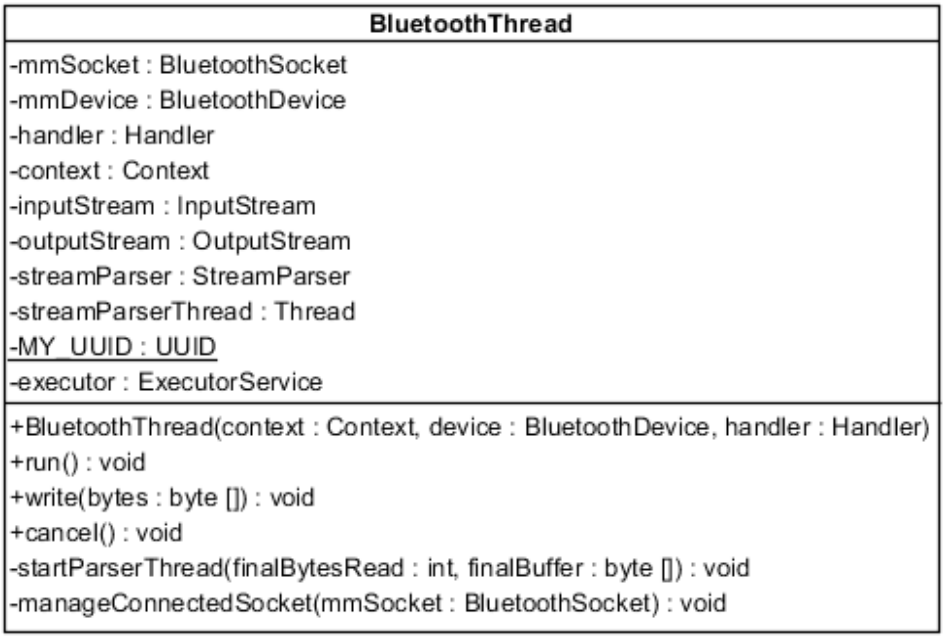


Figura 3.7: Diagrama clasei BluetoothThread

Clasa este instanțiată din *MainActivity*, de pe firul principal de execuție, unde îi sunt asociate detaliile dispozitivului la care se va conecta (în acest caz, centrala termică).

Deoarece extinde clasa *Thread*, modulul se va executa pe un fir diferit față de cel principal. La pornirea acestui fir de execuție se vor rula instrucțiunile din metoda *run()*, unde este inițiată conexiunea printr-un soclu(*BluetoothSocket*) către dispozitivul ales. După acest pas, se ascultă pe soclul respectiv primirea biților de date, cât timp conexiunea este activă.

Când s-au primit date, acestea sunt trimise către parsare la un obiect de tipul StreamParser, execuția sa fiind lansată pe un fir separat.

Prin apelul metodei publice *write()*, se pot scrie biți pe fluxul de ieșire. În cazul de față, metoda este folosită pentru trimiterea mesajelor de control către centrala termică.

În Figura 3.7 de mai sus, se poate observa diagrama clasei *BluetoothThread*, unde:

- *mmSocket* este soclul pe care se transmit datele între modulul curent și dispozitivul la care este conectat.
- *mmDevice* reprezintă datele încapsulate ale dispozitivului la care se conectează.
- *handler* este obiectul prin care se trimit notificări către firul principal.
- *inputStream* și *outputStream* sunt canalele de intrare și de ieșire ale datelor, aferente soclului la care s-a stabilit conexiunea.
- *MY_UUID* este un identificator unic care trebuie să apară în cele două sisteme între care se realizează conexiunea.
- *executor* este folosit pentru a lansa execuția modulului *StreamParser* pe un fir prealocat de sistem, pentru a reduce costurile și timpul de alocare al firelor de execuție. Acest component este necesar întrucât se va apela foarte des metoda de parsare a modulului *StreamParser*, ce va necesita fi executată pe un fir separat.
- *streamParser* este obiectul la care se trimit datele primite pentru a fi parsate.

3.2.4. Modulul protocol de parsare

Datele primite de la centrala termică respectă un anumit protocol de împachetare orientat pe octet.

3.2.4.1. Împachetarea datelor

Mai mulţi octeţi împachetaţi într-unul din formatele de mai jos formează un mesaj(pachet).

Un mesaj este format din mai multe câmpuri. Un câmp poate conţine unul sau mai mulţi octeţi. Structura câmpurilor se poate observa în Tabelul 3.1 de mai jos.

Tabelul 3.1: Structura câmpurilor pachetului de date

Nr. Octeţi	1	2	4	1	4	1
Tip	SOM	ID	DATA	DEN	CRC	EOM
Descriere	>	AA-ZZ	0000-FFFF	#	0000-FFFF	<
TOTAL	13 octeţi					

Semnificaţiile tipurilor de câmpuri prezente în pachet sunt după cum urmează:

- SOM[2]
  - semnificaţie : delimitează începutul unui mesaj
  - lungime : 1 octet
  - valoare : ASCII (>) = 0x3E = 62
- ID
  - semnificaţie : ID mesaj: pentru interpretare date (vezi secţiunea următoare)
  - lungime : 2 octeţi
  - valoare : AA÷ZZ sau aa÷zz
- DATA
  - semnificaţie : datele componente ale mesajului
  - lungime : 4 octeţi
  - valoare : 0÷FFFFFFFF
  - atenţie datele sunt codificate ASCII (vezi secţiunea codificare ASCII)
- DEN[3]
  - semnificaţie : delimitează sfârşitul octeţilor de date
  - lungime : 1 octet
  - valoare : ASCII (#) = 0x23 = 35
- CRC[4]
  - semnificaţie : sumă de control pentru mesaj CRC16 - CCITT
  - lungime : 4 octeţi
  - valoare : calculată cu polinomul : (0x1021)
  - include toţi cei 8(12) octeţi care formează câmpurile precedente
- EOM[5]
  - semnificaţie : delimitează sfârşitul unui mesaj

2 Eng. Start of Message
3 Eng. Data End
4 Eng. Cyclic Redundancy Check
5 Engl. End Of Message

- lungime : 1 octet
- valoare : ASCII (<) = 0x3C = 60
- orice caracter trimis după EOM va fi ignorat

Exemple ale pachetelor de date pot fi următoarele:

- >RQ0201#7CDC<
- >SS0123#DC23<
- >QV1028#FFDE<
- >MF0156#FA30<

CRC-ul se calculează de la SOM până la DEN inclusiv, utilizând polinomul 0x1021. CRC-ul se aplică datelor din mesajul final codificate HEX ASCII. CRC-ul se transmite codat HEX ASCII pe 2 octeţi.

Orice mesaj începe cu SOM şi se sfârşeşte cu EOM. Orice caracter trimis între un EOM şi următorul SOM este ignorat.

Timpul de transmisie pentru un mesaj de la aplicație la dispozitiv nu trebuie să depăşească 2 secunde.

Orice mesaj are lungimea maximă de 32 octeţi. Dacă un mesaj, sau un şir de date (între un SOM şi un EOM) depăşeşte 32 de octeţi datele sunt suprascrise într-un buffer circular.

Mesajele vehiculate între unitatea de control a centralei și aplicație sunt de 2 tipuri:

- de la aplicație către centrală:
  - comenzi
- de la centrală către aplicație:
  - mesaje de stare
  - răspunsuri la comenzi

Singurul mesaj de comandă de la aplicație către centrală va fi reprezentat de:

- comanda CC[6]:
  - ID : CC
  - DATA : 4 octeţi codați DEC ASCII cu rol de cheie (parolă)
  - cheia are valoarea 7137
  - control comunicație: activează transmisia de mesaje de stare către PC; dacă acest mesaj nu este primit sau dacă nu conţine cheia corectă pentru un anumit interval de timp (cel mult 8 secunde), atunci transmisia mesajelor de stare către PC este dezactivată.
  - Exemplu :
    - >CC7137#6DFC<
  - Răspuns :
    - >ME0011#0733<
  - Efect :
    - Pornește/menține activă transmisia de mesaje de stare de la centrală către aplicație.

Unitatea de control a centralei trimite la fiecare 500ms mai multe mesaje de stare către aplicație dacă este activată periodic transmisia prin mesajul CC (Communication Control). Mesajul CC trebuie trimis de aplicatia PC la fiecare cel mult 8 secunde.

Structura mesajelor este cea prezentată mai sus cu precizarea că toate mesajele au un câmp de date de 4 octeţi. Datele sunt aliniate în cadrul câmpului la dreapta.

Toate mesajele au datele codificate ASCII HEX dar pot avea grupări/reprezentări diferite ale octeţilor de date.

6 Engl. Communication Control

Tabelul 3.2: Tipuri de identificatori ale datelor

Nr.	ID	Data	Grupare
1	ID[7]	identificator boiler	2 + 2 octeţi
2	NM[8]	număr mesaje procesate	4 octeţi
3	RQ[9]	cerere date	2 + 2 octeţi
4	SS[10]	stare sistem	2 + 2 octeţi
5	EE[11]	elemente de execuție	2 + 2 octeţi
6	QV[12]	valorile cantitative ale senzorilor	2 + 2 octeţi
7	CE[13]	intrări de activare încălzire centrală	2 + 2 octeţi
8	MC[14]	comenzi de modulare	1 + 3 octeţi
9	MF[15]	răspuns de modulare	1 + 3 octeţi
10	US[16]	setări utilizator	1 + 3 octeţi

Cele mai multe mesaje sunt indexate, adică un mesaj cu un acelasi ID va fi trimis de mai multe ori succesiv, până când toate elementele din vectorul asociat mesajului sunt trimise. Elementul care este conținut in cadrul mesajului este identificat pe baza unui index, ce este inclus in mesaj si este incrementat după fiecare transmisie a mesajului indexat. Deoarece aceste mesaje au asociate un index și un vector, ele sunt numite mesaje indexate sau mesaje vectorizate.

- Mesajul de stare ID
  - ID : ID
  - DATA : informație identificare tip boiler (model si revizie firmware), 4 octeţi = 2 model boiler + 2 revizie firmware
  - LIMITE : 11 ÷ 17 (model) şi 00 ÷ 99 (revizie firmware), 0x1100 ÷ 0x1799
  - EXEMPLU : >ID1401#D4C5< - boiler model C14, revizie firmware 1.

Tabelul 3.3: Mesajul de stare ID

	DATA (codat ASCII)			
Nume	ID1	ID0	REV1	REV0
Octet	3	2	1	0

Byte	Simbol	Functie
3 ÷ 2	ID1 ÷ ID0	ID boiler model
1 ÷ 0	REV1 ÷ REV0	Revizie firmware

- Mesajul de stare NM
  - ID : NM (Number of Messages)
  - DATA : număr mesaje recepționate cu succes : 4 octeți
  - LIMITE : 0x0000 ÷ 0xFFFF (0 - 65535)
  - EXEMPLU : >NM1A73#98A1< - mesaje recepționate cu succes în număr de

7 Engl. boiler IDentification
8 Engl. Number of processed Messages
9 Engl. ReQuest data
10 Engl. System Major Status
11 Engl. Execution Elements
12 Engl. sensors Quantities Values
13 Engl. Central-heating Enable inputs
14 Engl. Modulation Commands
15 Engl. Modulation Feedbacks
16 Engl. User Settings

6771

Tabelul 3.4: Mesajul de stare NM

	DATA (codat ASCII)			
Nume	NM3	NM2	NM1	NM0
Octet	3	2	1	0

Octet	Simbol	Functie
3 ÷ 0	NM3 ÷ NM0	Numar de mesaje receptionate cu succes

- Mesajul de stare SS
  - ID : SS (System major Status)
  - DATA : 4 octeţi de forma IIDD, unde II=index in cadrul vectorului de stări majore ale sistemului, iar DD=conţinutul vectorului corespunzător indexului
  - LIMITE : II : 0x00 ÷ 0x04 (dimensiunea vectorului fiind 5)
    DD: 0x00 ÷ 0xFF
  - EXEMPLU : >SS0120#ED42< - sistemul este in starea PREPURGE (din modul Utilizator)

Tabelul 3.5: Mesajul de stare SS

	DATA (codat ASCII)			
Nume	I1	I0	D1	D0
Octet	3	2	1	0

Octet	Simbol	Functie
3 ÷ 2	I1 ÷ I0	Indexul in cadrul vectorului de stare
1 ÷ 0	D1 ÷ D0	Conţinutul vectorului corespunzător indexului

	Vector System major Status				
Nume	IERROR	NERROR	CERROR	STATE	OPMODE
Index	4	3	2	1	0

Index	Simbol	Functie
4	IERROR (I)	Eroare informativa (nu opreste ciclul de ardere, este doar afisata)
3	NERROR (N)	Eroare normala (opreste ciclul de ardere, dar se face postventilare si postcirculatie)
2	CERROR (F)	Eroare critica (blocheaza imediat centrala)
1	STATE	Stare curenta, relativa la modul de operare
0	OPMODE	Mod de operare curent

Tabelul 3.6: Stare

Valoare	Semnificatie
0x00	Initial
0x11	Wait
0x20	Prepurge
0x21	Interpurge
0x22	Ignition

0x23	Burning
0x24	Extinction
0x25	Postpurge
0x30	Circulation
0x40	Error

EXEMPLU: >SS0123#DC23< //(Burning state)

Tabelul 3.7: Mod de operare (OPMODE)

Valoare	Semnificatie
0x00	Initial – mod de operare in care se afla sistemul in primele secunde dupa un reset
0x01	Standby – mod de operare in care se afla sistemul daca centrala a fost oprita (prin apasarea butonului POWER sau trimiterea unui mesajul US ce modifica starea centralei)
0x02	Failure – mod de operare in care se intra imediat ce a fost detectata o eroare critica; toate elementele de executie sunt oprite imediat
0x03	User – mod de operare in care se afla sistemul daca nu sunt erori critice, daca nu se afla in primele secunde dupa un reset si daca centrala a fost pornita (prin apasarea butonului POWER sau trimiterea unui mesajul US ce modifica starea -status- centralei in ON)

EXEMPLU: >SS0003#C4F7< //(Mod utilizator)

- Mesajul de stare QV
  - ID : QV (măsurători senzori – valori cantitative)
  - DATA : valorile mărimilor măsurate de senzori . Mesaj de forma IDDD unde : I este indexul in cadrul vectorului *QV* iar DDD reprezintă data x 10.
  - LIMITE : 0x0000 ÷ 0x8FFFF
  - EXEMPLU : >QV324E#9A90<

Tabelul 3.8: Mesajul de stare QV

	DATA (codat ASCII)			
Nume	I0	D2	D1	D0
Octet	3	2	1	0

Octet	Simbol	Functie
3	I0	Indexul in cadrul vectorului
2 ÷ 0	D2 ÷ D0	Data x 10

Vectorul cu valorile marimilor masurate

Nume	T_T2	T_T1	T_EXT	T_EXH	T_CHR	T_CHT	T_DHW	F_DHW	P_CH
Index	8	7	6	5	4	3	2	1	0

Index	Simbol	Functie
8	T_T2	Temperatura masurata in tancul de apa – senzorul 2
7	T_T1	Temperatura masurata in tancul de apa – senzorul 1
6	T_EXT	Temperatura masurata in exterior – atentie poate avea si valoare negativa
5	T_EXH	Temperatura masurata a gazelor de evacuare

4	T_CHR	Temperatura masurata pe circuitul de termoficare - retur
3	T_CHT	Temperatura masurata pe circuitul de termoficare - tur
2	T_DHW	Temperatura masurata pe circuitul DHW
1	F_DHW	Debitul masurat pe circuitul DHW
0	P_CH	Presiunea masurata in circuitul de incalzire (doar octetul low) presiune, (17 = 1.7 bar).

EXEMPLU: >QV324E#9A90< // T_CHT = (590/10) = 59º C

- Mesajul de stare MC
  - ID : MC
  - DATA : Comandă de modulare (PWM) pentru ventilator ori vană de gaz. Mesaj de forma IDDD unde I reprezintă indexul în cadrul vectorului : cu elemente de modulat iar DDD comanda de modulare.
  - LIMITE : 0x0000 ÷ 0x13E8
  - EXEMPLU : >MC013A#9A5F<

Tabelul 3.9: Mesajul de stare MC

	DATA (codat ASCII)			
Nume	I0	D2	D1	D0
Octet	3	2	1	0

Octet	Simbol	Functie
3	I0	Index in cadrul vectorului cu elemente de modulat
2 ÷ 0	D2 ÷ D0	Valoare modulare (PWM)

Vector comenzi modulare

Nume	GVV	FRPM
Index	1	0

Index	Simbol	Functie
1	GVV	Comandă modulare ventilator // C13/C14/C15/C17: ventilator PWM: 0-1000
0	FRPM	Comandă modulare vana de gaz // C11/C12/C15/C17: vana gaz PWM: 0-1000

EXEMPLU: >MC013A#9A5F< // modulare ventilator, PWM 314

- Mesajul de stare MF
  - ID : MF
  - DATA : Feedback-ul corespunzător elementelor comandate. Mesaj de forma : IDDD unde I reprezintă indexul în cadrul vectorului cu valori măsurate.
  - LIMITE : 0x0000 ÷ 0x2FFF
  - EXEMPLU : >MF0156#FA30<

Tabelul 3.10: Mesajul de stare MF

	DATA (codat ASCII)			
Nume	I0	D2	D1	D0
Octet	3	2	1	0

Octet	Simbol	Functie
3	I0	Index in cadrul vectorului cu elemente de modulat
2 ÷ 0	D2 ÷ D0	Valoare masurată

Vectorul cu valorile marimilor masurate

Nume	IONV	GVV	FRPM
Index	2	1	0

Index	Simbol	Functie
2	IONV	All models: burner ionisation voltage:scale x100 (unit=10mV)
1	GVV	C11/C12: gas-valve regulator voltage: scale x100 (unit=10mV)
0	FRPM	C13/C14/C15/C17: fan rotation speed: scale x0.1 (unit=10RPM)

EXEMPLU: >MF0156#FA30< //FAN RPM = 342

3.2.4.2. StreamParser

Octeții primiți de modulul Bluetooth ajung mai departe, către parsare, la un obiect de tipul *StreamParser*. Acesta construiește, din șirul de biți primit, caractere ASCII care fac parte din mesajul trimis de centrală. Apoi încearcă să urmărească formarea pachetelor de tipul celor descrise mai sus, în capitolul 3.2.4.1 (ex.: >QV1028#FFDE<).

Pentru acest lucru am construit un automat care să identifice un astfel de pachet în șirul parsat. Acesta este reprezentat în Figura 3.8.

Simbolurile de tranziție sunt, de fapt, delimitatorii de pachet descrişi în capitolul anterior:

- SOM – începutul mesajului – caracterul “>”.
- DEN – delimitator pentru sfârşit de date şi început de cod CRC – caracterul “#”.
- EOM - sfârşit de mesaj - caracterul “<”.

Stările au următoarele semnificații:

- Starea 1: se ignoră orice caracter până la SOM, după care se începe construcția șirului de date cu SOM și se trece la starea următoare.
- Starea 2: se adaugă orice caracter(alfa-numeric) primit la șirul de date. La apariția caracterul DEN, se introduce în șir, apoi se trece la starea următoare.

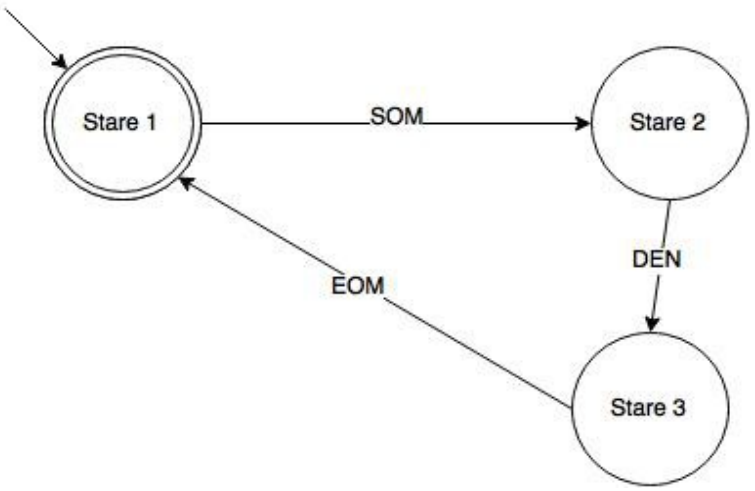


Figura 3.8: Automat pentru parsarea mesajului

- Starea 3: Se construiește șirul CRC din caracterele primite. La apariția caracterului EOM, se calculează CRC-ul șirului de date curent convertit la întreg hexazecimal. Dacă CRC-ul calculat, corespunde cu CRC-ul primit în mesaj, atunci datele au fost transmise corect. În acest caz, șirul de date este inserat într-o coadă partajată cu activitatea care se rulează pe firul principal, acesta fiind apoi notificat. Se trece înapoi în starea 1.

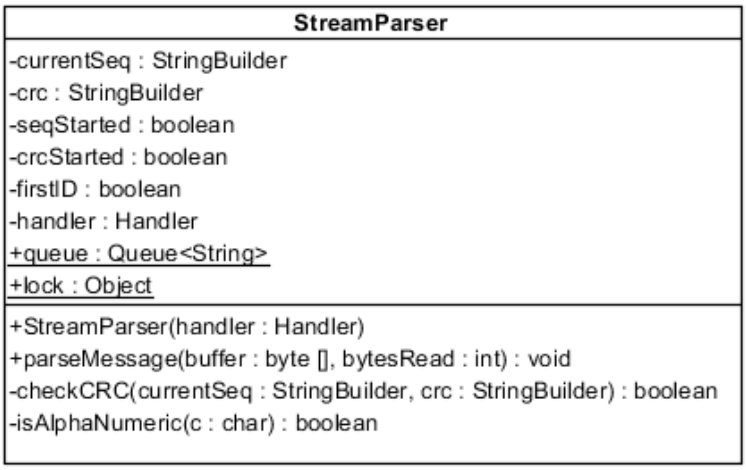


Figura 3.9: Diagrama clasei *StreamParser*

3.2.4.3. MC1XPackProcesser

După primirea unui șir de date de la obiectul *StreamParser*, activitatea principală îl trimite mai departe la procesare către *MC1XPackProcesser*, care se ocupă de acest pachet de date primit pe un fir separat de execuție.

Șirul de date ajuns aici are forma >IDHHHH# (ex.: >QV1028#), unde ID este identificatorul tipului de mesaj, iar HHHH este câmpul de date codate HEX ASCII.

Acest modul extrage câmpul ID al fiecarui pachet și în funcție de valoarea acestuia este apelată o metodă corespunzătoare pentru extragerea informațiilor din câmpul de date.

Câmpul de date este trimis la procesare, însă, doar dacă a fost recepționat până în acel moment cel puțin un pachet cu valoarea câmpului ID egală cu șirul „ID”. De asemenea, interfața este notificată atunci când este gata un set întreg de date procesate, care sunt diponibile pentru afișare, atunci când s-au primit două pachete care au câmpul ID cu valoarea „ID”.

Așadar, fiecare câmp de date este procesat diferit în funcție de valoarea câmpului ID aferent pachetului său. Procesarea datelor se realizează cu metode ale unui obiect *MC1XData*, informațiile extrase fiind stocate tot în acest obiect.

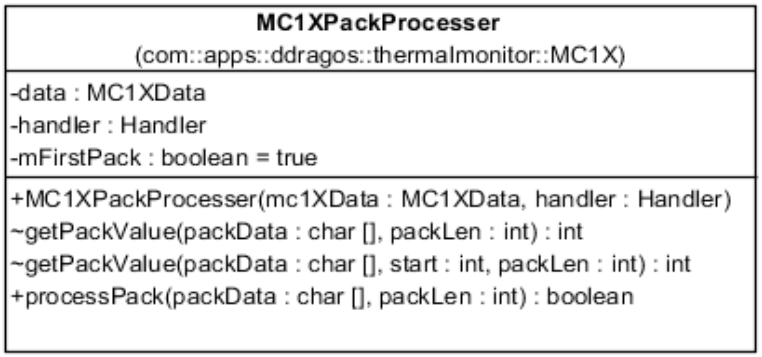


Figura 3.10: Diagrama clasei *PackProcesser*

3.2.4.4. MC1XData

Această clasă a modului de parsare se ocupă cu stocarea informațiilor primite de la centrala termică și reprezintă toate valorile și stările în care se află centrala la un moment dat.

MC1XData are propriile metode de extragere a informațiilor din șirul de date primit în format hexazecimal. Modulul *MC1XPackProcesser* indică ce metodă se va folosi pentru extragerea informațiilor în funcție de tipul de pachet identificat.

Acest modul este constituit de fapt dintr-o ierarhie de două clase: *MC1XDataH*, format din câmpurile, ce reține informațiile centralei la un moment dat, și metode pentru accesul acestora, și *MC1XData* care extinde *MC1XDataH*, la care adaugă metode pentru extragerea informațiilor în câmpurile respective.

3.2.5. Modulul interfață grafică

Acest modul se ocupă cu preluarea comenzilor de la utilizator, recepția mesajelor de la modulele care se execută pe alte fire de execuție, recepția datelor procesate și afișarea acestora sub diferite forme.

După cum se poate observa și în Figura 3.11, punctul central al modulului este activitatea *MainActivity*. Aceasta conține fragmentele care se ocupă de afișare sau setări, însă are și rol de releu, întrucât aici vin toate mesajele și datele procesate de către celelalte fire de execuție, care sunt trimise apoi mai departe către fragmente pentru a fi afișate.

MainActivity este numită activitate, deoarece extinde clasa *Activity*[17], care reprezintă un ecran al aplicației. Aceasta se execută pe firul principal, unde este redată și interfața cu utilizatorul, de aceea este bine ca alte procese blocante să se execute pe alte fire de execuție. Această clasă are, așadar, următoarele roluri:

- gestionează comenzile primite de pe intefață, cum ar fi actualizarea listei cu dispozitive disponibile, selectarea și conectarea la unul din ele.
- Lansarea modulului Bluetooth pentru gestionarea fluxului de date.
- Primirea informațiilor din conexiunea curentă, dar și a datelor preprocesate.
- Trimiterea datelor procesate mai departe, către fragmentele ce se ocupă cu afișarea sugestivă.

SlidingTabsFragment este un fragment folosit pentru comutarea între celelalte fragmente ce se ocupă cu afișarea informațiilor pe interfață. Un fragment este un concept foarte asemănător cu activitatea, singura diferență majoră fiind că o activitate sau un fragment poate conține mai multe fragmente, ce pot apărea în prim-plan simultan sau pe rând. *SlidingTabsFragment* comută într-o manieră fluentă fragmentele din prim-plan în funcție de alegerea utilizatorului. Existența altor trei fragmente este necesară pentru a putea împărți informațiile afișate, întrucât volumul interpretabil de date este mare și diferit ca semnificație.

GraphFragment este fragmentul care primește datele aferente mărimilor ce pot fi reprezentate grafic. Acesta conține un obiect de tip *GraphView* pe care se desenează graficele construite cu aceste mărimi. Fiecare grafic este gestionat printr-un obiect de tip *Graph,* care primește valorile și le transformă în puncte pe ecran. Fiecare două puncte consecutive sunt apoi unite, formând graficul aferent. *GraphView* este actualizat la fiecare set nou de valori, astfel graficele avansează, reprezentând tot mai detaliat comportamentul mărimilor monitorizate.

LedsFragment este fragmentul care primește informațiile legate de stările în care se află centrala. În funcție de starea curentă, acest fragment aprinde un led care indică faptul că centrala se află în starea corespunzătoare ledului repsectiv.

17 Trad.(eng.) - Activitate

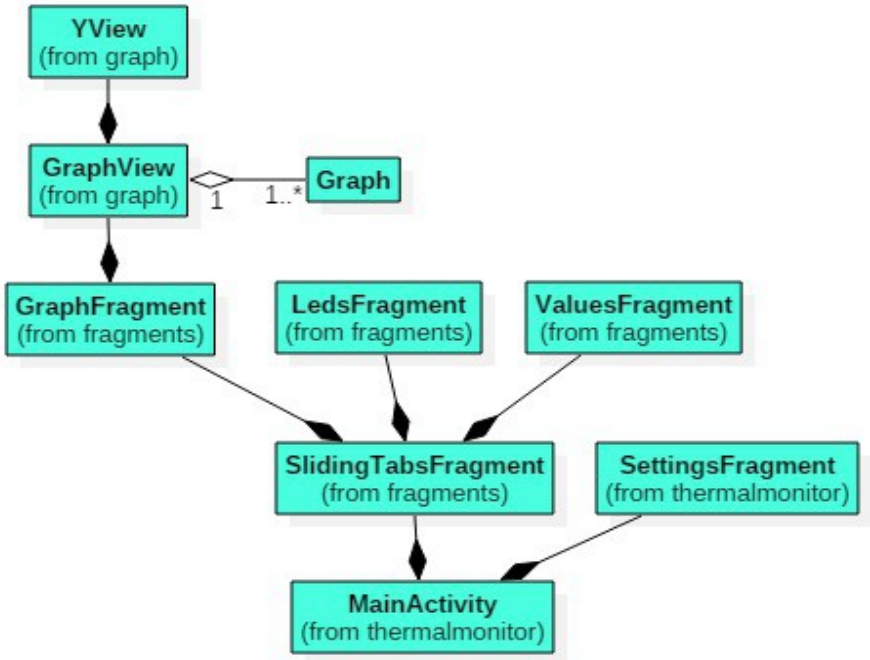


Figura 3.11: Diagrama de clase pentru modulul interfață grafică

ValuesFragment afișează valoarea curentă sau ultima valoare primită pentru fiecare mărime monitorizată de către aplicație.

SettingsFragment se ocupă cu gestiunea preferințelor pentru grafice. Din acest fragment se pot schimba vizibilitatea, culoarea și scara graficelor. Preferințele sunt salvate automat de sistem.

Capitolul 4. Implementarea aplicației

Aplicația mobilă *Thermal Monitor* a fost realizată pe și pentru platforma Android, folosind pachetul de dezvoltare Android versiunea 21, corespunzătoare platformei cu versiunea 5.0 (*Android Lollipop*). Sistemul de operare minim necesar pentru a putea rula aplicația este Android 4.0.3.

4.1. Activitatea principală și firele de execuție

Prima activitate care se execută la lansarea aplicației este *MainActivity*, ce este de fapt o extensie a clasei *FragmentActivity,* întrucât *MainActivity* conține fragmente, pentru afișarea mai dinamică a elementelor de interfață. În ciuda numelui, *SettingsFragment,* este, de asemenea, tot o activitate. Ambele clase sunt specificate în fișierul *AndroidManifest.xml. MainActivity* este specificată ca fiind activitatea incipientă a aplicației prin completarea câmpurilor din *<intent-filter>* cu următorii parametri:

```
<intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category
           android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

MainActivity conține următoarele controale pentru interfața grafică:

- obiecte de tip *Button:*
  - conectare
  - deconectare
  - actualizare dispozitive disponibile
- *TextView,* pentru afișarea stării conectat/deconectat.

De asemenea, pentru a gestiona cât mai bine interfața și pentru a oferi posibilitatea utilizatorului de a schimba foarte repede între ferestrele cu diferite elemente de importanță, *MainActivity* conține un fragment numit *SlidingTabsFragment,* care se ocupă de alte trei fragmente propriu-zise pentru afișarea datelor.

Deoarece activitatea principală așteaptă mesaje și date de la alte module (cum ar fi *BluetoothThread* sau *StreamParser*), care se execută în paralel, aceasta are nevoie de un mijloc pentru a primi notificări de la alte fire de execuție. Pentru asta, este folosit un obiect care implementează interfața *android.os.Handler.* Clasa *BTHandler* este imbricată în *MainActivity*, iar metoda sa *handleMessage(Message msg)* este apelată de fiecare dată când se primește un mesaj de pe un alt fir de execuție. În funcție de tipul mesajului transmis(câmpul *msg.what*), este identificată acțiunea ce trebuie executată. De exemplu, atunci când conectarea a reușit, *BluetoothThread* trimite către activitatea de unde a fost instanțiat handler-ul(*MainActivity*), un mesaj în felul următor:

```
handler.obtainMessage(CONNECTED,mmDevice).sendToTarget();
```

Astfel, este construit mai întâi mesajul, format din tipul său (constanta *CONNECTED*) și obiectul transmis mai departe (*mmDevice*, reprezentând datele dispozitivului la care s-a efectuat conectarea, pe care activitatea le va utiliza pentru afișare). Metoda *sendToTarget()* trimite acest mesaj construit, către activitatea principală.

Un alt astfel de mesaj este primit de la un obiect de tipul *StreamParser*, care identifică

pachetele de date în octeții primiți prin Bluetooth. Acesta pune pachetul găsit într-o coadă partajată și notifică firul principal prin același handler, cu tipul mesajului *QUEUE_DATA_RDY.* În acest caz, pachetul este extras în mod sincron în felul următor:

```
synchronized (StreamParser.lock) {
      try {
            data = queue.remove();
          } catch (NoSuchElementException ex) {
            Log.i("Queue", "Empty Queue");
          }
 }
```

Lock este un obiect folosit de ambele fire pentru a sincroniza accesul la coadă.

Următoarea operație este de extragere a informațiilor din pachet, lucru realizat, de asemenea, asincron.

Deoarece fiecare fir care își termină execuția trebuie alocat din nou dacă are o nouă acțiune de îndeplinit, ar fi foarte costisitor pentru aplicația de față dacă s-ar întampla asta. De exemplu, de fiecare dată când este extras un pachet nou din coadă ar trebui lansat un nou fir de execuție pentru a se ocupa de procesarea pachetului. Pentru a evita acest lucru, aplicația folosește fire de execuție prealocate de sistem, cu ajutorul clasei *Executors* din *java.util.concurrent*. Inițial, trebuie specificat de câte fire este nevoie:

```
executor = Executors.newFixedThreadPool(1);
```

Atunci când este lansat în execuție un fir prealocat, se verifică disponibilitatea acestuia în funcție de numărul de fire cerute specificate la început. Dacă, în acest caz, singurul fir prealocat nu și-a terminat execuția, acțiunea de rulat în paralel va aștepta ca aceasta să se termine. În exemplul următorul este lansată în execuție procesarea pachetului, folosind acest fir de execuție prealocat:

```
executor.execute(new Runnable() {
       @Override
       public void run() {
          packProcesser.processPack(fData.toCharArray(),fData.length());
       }
});
```

4.2. Conexiunea Bluetooth

Pentru a stabili o conexiune Bluetooth, în clasa *BluetoothThread,* este creat un soclu de comunicație cu dispozitivul ales, iar apoi, în metoda *run(),* suprascrisă peste cea a clasei *Thread,* este apelată *connect()* pentru acest soclu:

```
tmp = device.createRfcommSocketToServiceRecord(MY_UUID);
…
 @Override
    public void run() {
        try {
            mmSocket.connect();
        }
```

```
catch (IOException connectException) { … }
    …
}
```

MY_UUID este un identificator unic folosit pentru a se conecta la serviciul potrivit (care are același *UUID)* din dispozitivul remote[18].

Datele primite de modulul Bluetooth mai apoi la *StreamParser,* unde sunt procesate în paralel, folosind tot fire de execuție prealocate.

Pentru a avea acces la serviciul Bluetooth și la lista cu dispozitive disponibile este necesară specificarea în fișierul *AndroidManifest.xml* a următoarelor a următoarelor permisiuni:

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
```

Acest lucru este necesar, deoarece utilizatorul trebuie să-și dea acordul pentru ca aplicația să poată folosi acest serviciu.

4.3. Parsarea și reprezentarea datelor

StreamParser implementează, de fapt automatul prezentat în capitolul anterior, folosind însă următoarele flag-uri pentru a codifica stările mai sugestiv: *seqStarted, crcStarted.* După identificarea octeților de date și a CRC-ului, este calculat codul CRC al datelor pentru a fi verificată corectitudinea lor:

```
int computedCRC = new CRC().crcFromString(currentSeq.toString());
int receivedCRC = Integer.parseInt(crc.toString(),16);
if ((computedCRC ^ receivedCRC) == 0)
        return true;
```

Pentru calculul CRC-ului este folosită metoda clasei *CRC,* care calculează CRC-ul pentru un șir de caractere ASCII dintr-un tip de dată *String.*

După acest pas, pachetul este trimis la *MainActivity* prin intermediul handler-ului, de unde este procesat, pe firul de execuție prealocat descris mai sus, de un obiect *PackProcesser.* Acesta identifică tipul de pachet și acțiunea potrivită pentru acesta. Mai jois este un exemplu pentru un pachet de tip *MF*:

```
int packType = buildPackType(packData[1], packData[2]);
if (packType == PACKTYPE_MOTANC15ID) { … }
…
else if (packType == PACKTYPE_MODULATION_FEEDBACK) {
            uIntValue = getPackValue(packData, packLen);
            data.setModulationFeedback(uIntValue);
            return true;
       }
```

Se construiește ID-ul pachetului din al doilea și al treilea caracter al pachetului, apoi se alege acțiunea corespunzătoare pentru acesta. În acest caz se extrage valoarea corespunzătoare în variabila *uIntValue,* de tip *int,* valoare ce corespunde caracterelor de pe pozițiile 4-7 ale pachetului. Apoi se apelează metoda obiectului de tip *MC1XData* corespunzătoare pentru setarea

18 (engl.) trad. - îndepărtat, de la distanță.

valorilor de tip *Modulation Feedback,* ale aceluiași obiect.

MC1XData toate valorile și stările centralei la un moment dat, adică cele provenite din pachetele cuprinse între două ID-uri cu valoarea *„ID”.*

În metoda *setModulationFeedback* se extrag valorile pentru viteza ventilatorului, tensiunea vanei de gaz sau tensiunea de ionizare a flăcării boilerului:

```
index = ( value >> 12 ) & 0x0000000F;
val =  ( value & 0x00000FFF );

switch( index )
{
     case 0:
           m_mfFanSpeed = val;
           m_mfFanSpeedAvaible = true;
           break;
     case 1:
           m_mfGValveVoltage = val / 100.0;
           m_mfGValveVoltageAvailable = true;
           break;
     case 2:
           m_mfFlameIonisationVoltage = val / 100.0;
           m_mfFlameIonisationVoltageAvailable = true;
           break;
}
```

Index identifică pentru care din cele trei mărimi se va seta valoarea. El este extras luând primii 4 biți din cei 2 octeți de date. Deoarece 2 octeți înseamnă 16 biți, se deplasează *value* cu 12 biți la dreapta pentru extrage primii 4 biți. Pentru *val*, care este valoarea ce se va atribui mărimii selectate de index, se iau restul de 12 biți. Tensiunile sunt împărțite la 100 pentru a putea fi exprimate în volți, cu două zecimale.

Atunci când obiectul *MC1XData* a fost actualizat cu un nou set de valori, activitatea principală este notificată prin intermediul handler-ului pentru a trimite noile date către afișare. Mai departe, activitatea va notifica fragmentul *SlidingTabsFragment*, care se ocupă de cele trei tipuri de fragment folosite pentru afișare, iar acesta va apela metodele de actualizare pentru fiecare fragment:

```
((GraphFragment)adapter.getFragment(0)).updateView(mc1XData);
((LedsFragment)adapter.getFragment(1)).updateView(mc1XData);
((ValuesFragment)adapter.getFragment(2)).updateView(mc1XData);
```

LedsFragment va aprinde pe interfață ledurile corespunzătoare stărilor în care se află centrala. *ValuesFragment* va actualiza valorile curente ale mărimilor monitorizate cu ultimele valori primite prin *mc1XData.*

GraphFragment conține un obiect de tip *GraphView,* care va afișa grafic evoluția mărimilor prin valorile primite. De fiecare dată când se primește un nou set de valori, *GraphView* le adaugă ca puncte la graficele aferente mărimilor corespunzătoare. Pentru ca un grafic să fie reprezentat bine relativ la ecranul dispozitivului, trebuiesc efectuate transformări de vizualizare 2D asupra punctelor reale ale graficului. Următoarea secvență de cod efectuează această transformare:

```
point.x = offsetX + screenPoints.get(i-1).x + unitSize;
point.y = -offsetY + screenHeight – points.get(i).y*unitSize*yScale;
```

Pentru coordonata *x* se folosește punctul ecran calculat anterior, deci, inițial, primul punct este calculat separat:

```
point.x = offsetX + points.get(0).x*unitSize +
          Math.abs(points.get(0).x)*unitSize;
```

Semnificația parametrilor folosiți este următoarea:

- *point* – punctul de afișat pe ecran.
- *screenPoints* – mulțimea punctelor de afișat pe ecran.
- *points* – mulțimea punctelor reale.
- *offsetX, offsetY* – deplasamentul folosit pe ecran pentru ambele axe de coordonate.
- *screenHeight* – înălțimea ecranului(sau a porții de vizualizare).

unitSize – dimensiunea unei unități pe poarta de vizualizare (aceeași pentru ambele axe de coordonate). Este calculată cu formula (1):

$$unitSize = \frac{dimensiunea\ porții}{numărul\ maxim\ de\ puncte\ de\ afișat} \quad (1)$$

- *yScale* – scara punctelor pe axa Y, folosită pentru scalarea graficelor.

Deoarece fiecare eșantion de date este trimis de către unitatea de control a centralei odată la 500 milisecunde, frecvența de reprezentare a punctelor pe grafic va fi de 2 Hz. Pe axa X, poarta de vizualizare este împărțită astfel încât să acopere 300 secunde. Formula (2) calculează numărul maxim de puncte de afișat pentru acest caz.

$$număr\ maxim\ puncte = dT \cdot F = 300\,s \cdot 2\,Hz = 600 \quad (2)$$

Fiecare obiect de tip punct este adăugat la obiectul corespunzător de tip *Graph,* este transformat în punct pentru ecran și adăugat la un obiect *Path.* Acesta îl unește cu punctul anterior inserat, formând astfel un drum care va ușura afișarea graficului, fiind necesară doar apelarea metodei *drawPath(Path p)* a obiectului de tip *Canvas,* în metoda *onDraw(Canvas c)* a *View*-ului *GraphView.*

4.4. Descrierea funcționării aplicației

Pentru instalarea aplicației este necesară rularea pachetului *ThermalMonitor.apk* salvat în dispozitiv sau în mod *Debug,* cu dispozitivul conectat printr-un cablu USB.

După deschiderea aplicației, se va putea observa un meniu în partea stângă, iar în partea dreaptă, un cadru inițial cu gridul pe care vor fi desenate graficele. Acest cadru poate fi schimbat cu ajutorul tab-urilor din partea de sus a acestuia.

Ecranul inițial poate fi observat în Figura 4.1. În meniul din dreapta se pot observa următoarele butoane:

- *Connect* – conectarea la dispozitivul ales.
- *Disconnect* – deconectare.
- Get Devices – actualizează lista cu dispozitivele împerecheate.

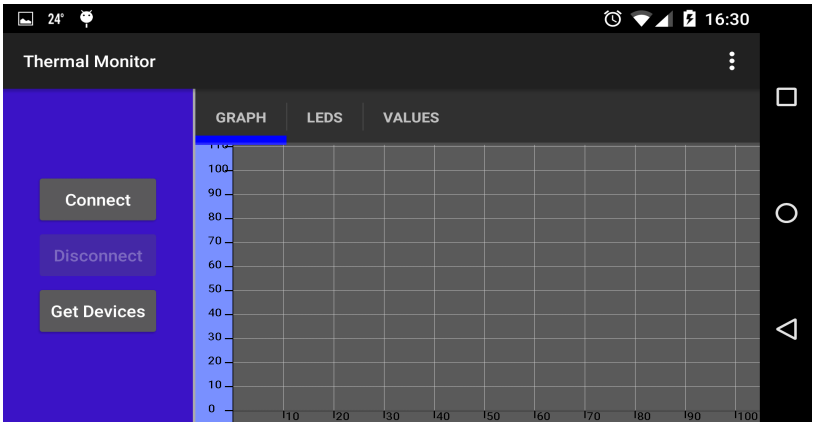


Figura 4.1: Ecranul inițial

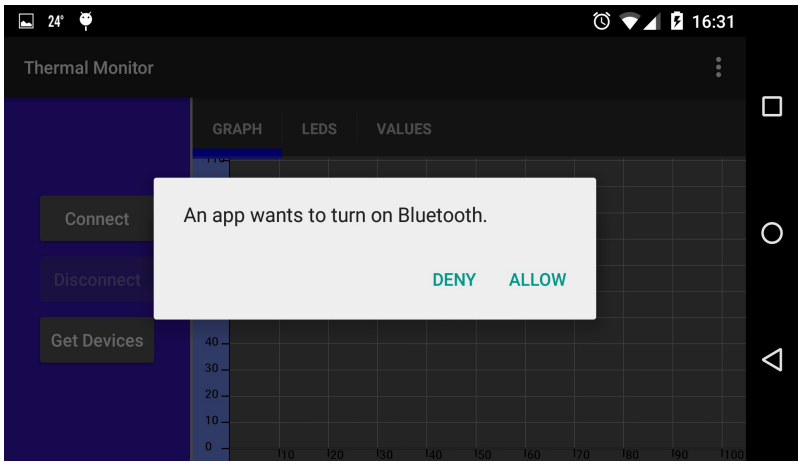


Figura 4.2: Activarea serviciului Bluetooth

La apăsarea butonului *Get Devices*, în cazul în care dispozitivul nu are serviciul Bluetooth activat, aplicația va cere permisiunea utilizatorului de a activa această funcție(Figura 4.2), iar apoi va afișa o listă cu dispozitivele împerecheate(Figura 4.3).

În Figura 4.3, utilizatorul a ales dispozitivul *Dropp* din lista cu dispozitive, iar apoi s-a conectat apăsând butonul *Connect*(după cum se poate vedea, dispozitivul are acum serviciul Bluetooth activat). După conectare, apare un mesaj pe mijlocul ecranului timp de câteva secunde, care anunță faptul că s-a reușit conexiunea. De asemenea, acest mesaj este afișat în meniul din stânga pe toată durata conexiunii.

Mai departe, unitatea de control a centralei termice va primi un mesaj de control și va începe să trimită date.

În Figura 4.4, sunt reprezentate grafic primele date primite. Se observă, în legenda graficului, că sunt selectate 7 mărimi pentru a fi reprezentate. Acest lucru poate fi schimbat din setările aplicației. Graficele continuă să se deseneze în timp real, odată cu primirea datelor de la centrală.

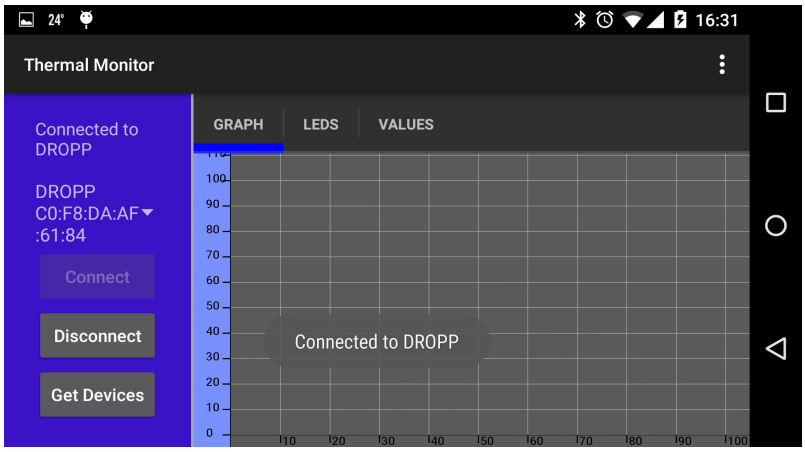


Figura 4.3: Conectare la dispozitivul ales

Figura 4.4: Reprezentarea primelor date primite

În Figura 4.5, este prezentat un stadiu mai avansat al execuției aplicației, unde s-au primit eșantioane pe o mai lungă perioadă de timp. Aici se poate observa mai clar comportamentul și evoluția mărimilor monitorizate.

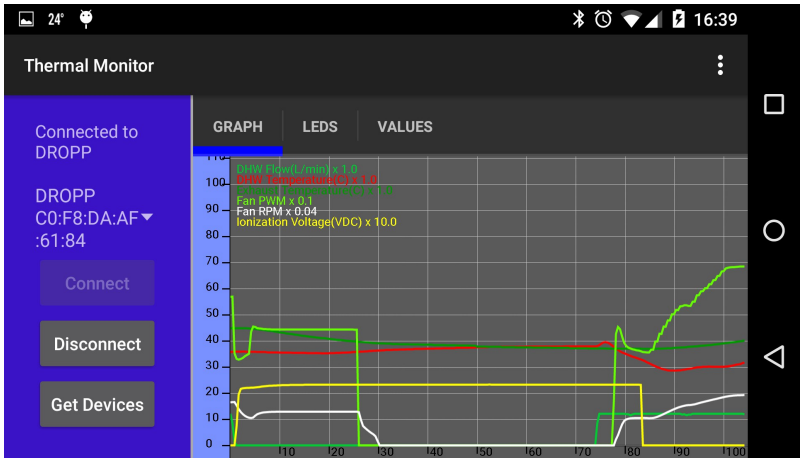


Figura 4.5: Comportamentul și evoluția mărimilor monitorizate

Pentru o vizualizare mai specifică, utilizatorul poate alege care dintre grafice să fie afișate. De asemenea, se mai pot schimba scara și culoarea. Aceste lucruri pot fi modificate din setările aplicației, ce se accesează apăsând cele 3 puncte verticale din colțul dreapta-sus al ecranului. În Figura 4.6, este un exemplu de cadru cu setări (preferințe utilizator).

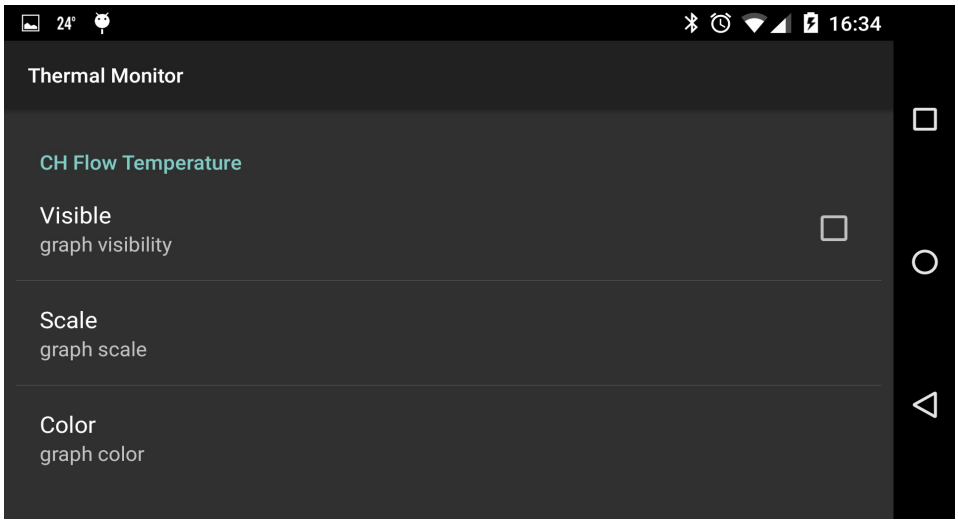


Figura 4.6: Meniul cu preferințele utilizatorului

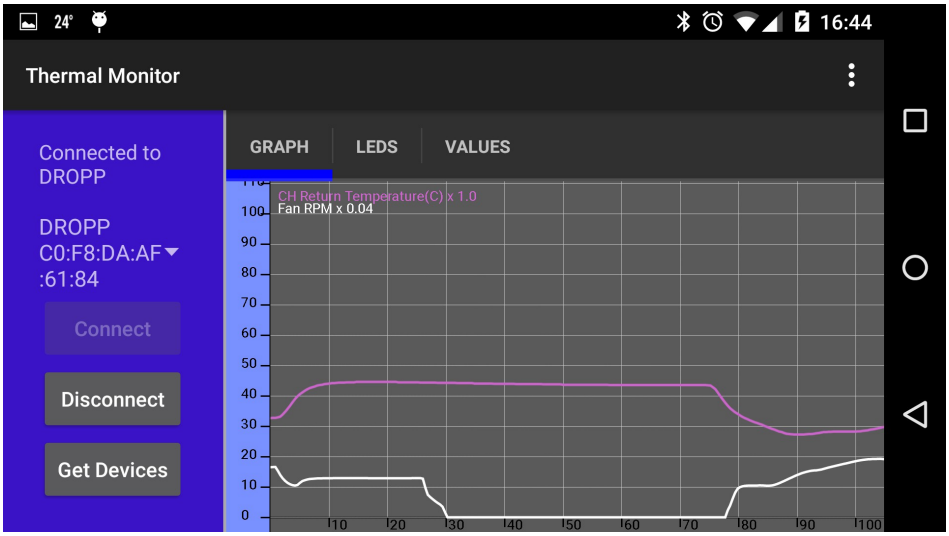


Figura 4.7: Reprezentare temperatură circuit de termoficare-retur și viteză ventilator

Pentru o vizualizare mai clară, după selectarea *checkbox*-urilor aferente din setări, în Figura 4.7, se pot observa reprezentările grafice numai a temperaturii măsurate pe circuitul de termoficare-retur(mov) și a vitezei ventilatorului(alb).

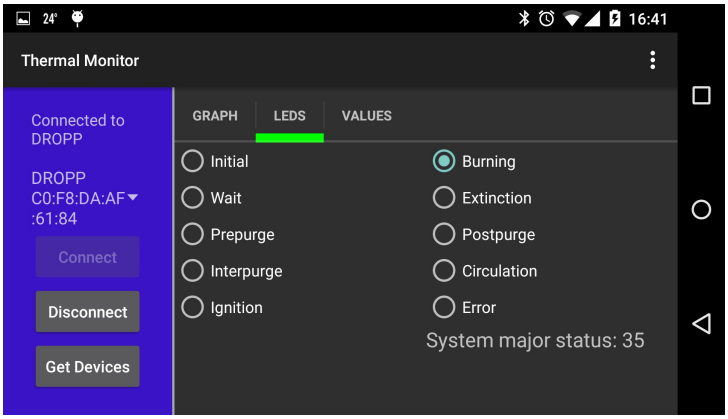


Figura 4.8: Starea curentă a centralei

La selectarea tabului *Leds,* va apărea cadrul cu stările centralei(Figura 4.8).

În Figura 4.8, centrala este în stadiul de ardere(*Burning*).

La selectarea tabului *Values*, va fi afișat cadrul cu valorile curente ale tuturor mărimilor monitorizate(Figura 4.9). Acestea se actualizează, de asemenea, în timp real, corespunzând cu punctele reprezentate pe grafic în același timp. Semnificația mărimilor monitorizate este următoarea (între paranteze poate fi specificată unitatea de măsură):

- DHW Flow(l/min) – debitul măsurat pe circuitul DHW.
- External Temperature(C) – temperatura la exterior.
- CH Flow Temperature(C) – temperatura pe circuitul de termoficare – tur.
- CH Return Temperature(C) – temperatura pe circuitul de termoficare – retur.
- DHW Temperature(C) – temperatura pe circuitul de termoficare DHW.
- Tank S1 Temperature(C) – temperatura măsurată în tancul de apă – senzorul 1.
- Tank S2 Temperature(C) – temperatura măsurată în tancul de apă – senzorul 2.
- Exhaust Temperature(C) – temperatura măsurată a gazelor de evacuare.
- Fan RPM – viteza de rotație a ventilatorului.
- CH Pressure(Bar) – presiunea măsurată în circuitul de încălzire.
- Ionization Voltage(V) – tensiunea de ionizare a gazelor de evacuare.

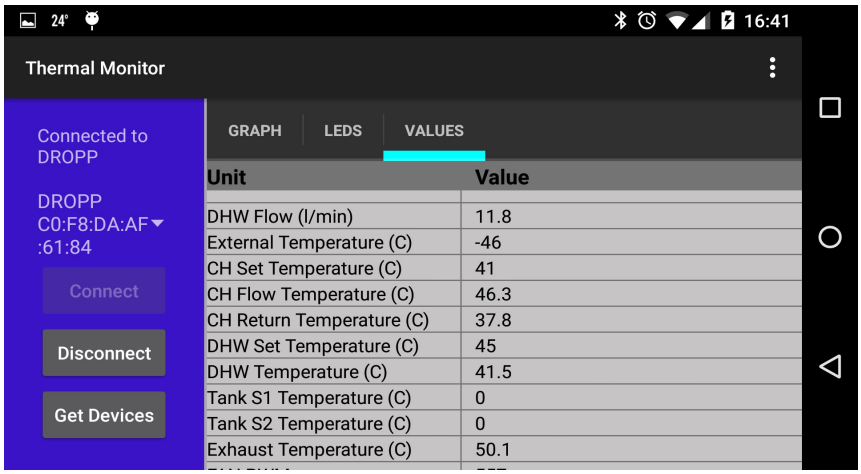


Figura 4.9: Valorile curente ale mărimilor monitorizate

Capitolul 5. Testarea aplicației

Aplicația a fost dezvoltată în mediul de programare *Android Studio,* care oferă toate uneltele necesare pentru dezvoltare și depanare, fiind pus la dispoziție și susținut de *Google.*

Pentru testarea aplicației Android *ThermalMonitor*, am creat o altă aplicație desktop de simulare a unității de control, care trimite datele prin serviciul Bluetooth al unui laptop. Acesta este un mod destul de eficient de testare, întrucât nu ar trebui să conteze în ce manieră se trimit datele sau cât de frecvent sunt trimise pachetele, atât timp cât aplicația le gestionează cu succes.

Aplicația desktop este implementată pentru sistemul de operare Windows, cu ajutorul platformei .NET, în limbajul C#. Pentru gestionarea conexiunii Bluetooth a fost folosită biblioteca open-source 32feet.NET[10].

Datele sunt citite dintr-un fișier stocat pe hard disk. Acest fișier este format din log-urile generate la un moment dat de către altă aplicație, ce a primit datele de la unitatea de control a centralei termice, prin transmisie serială. Log-urile sunt de fapt pachetele primite de la centrală, în format text, așa că aplicația de simulare poate fi considerat un intermediar între cetrala termică și aplicația mobilă. Câteva linii din fișierul cu log-uri se pot observa în Figura 5.1.

```
>ID0D02#69B2<
>NM0014#2D06<
>RQ0000#A285<
>RQ0180#7D90<
>RQ0201#7CDC<
>RQ032D#93A4<
>SS0003#C4F7<
>SS0123#DC23<
```

Figura 5.1: Pachetele salvate în format text

Fișierul folosit pentru testare conține pachete pentru aproximativ 400 de seturi de valori și stări ale centralei.

Interfața aplicației este prezentată în Figura 5.2. În prima fereastră de text vor apărea

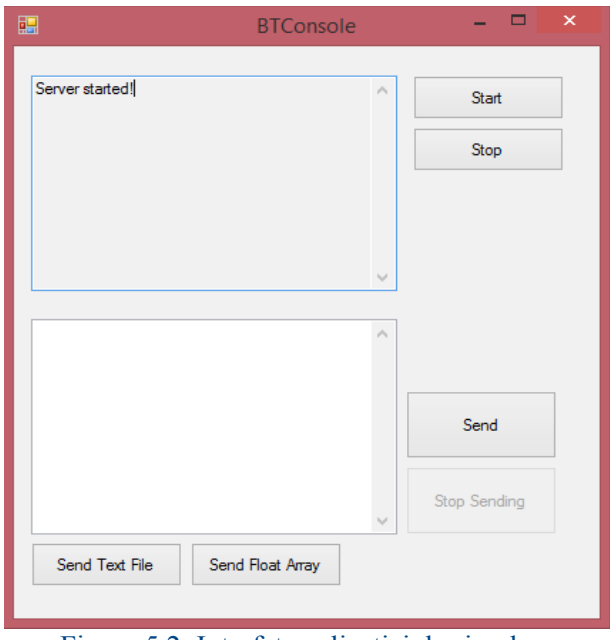


Figura 5.2: Interfața aplicației de simulare

mesaje de la aplicația curentă, precum notificări de conectare și erori, dar și mesaje de la dispozitivele conectate, în caz că acestea vor trimite.

Butoanele oferă următoarele funcționalități:

- *Start* – începe așteptarea conexiunilor. Se poate observa că în cazul de față(Figura 5.2) a fost pornită deja ascultarea.
- *Stop* – întreruperea conexiunilor și a ascultării.
- *Send* – trimite mesajul introdus în fereastra de text de jos, la toate dispozitivele conectate.
- *Send Text File* – utilizat pentru selectarea fișierului text care va fi trimis, și expedierea acestuia către toate dispozitivele conectate.
- SendFloatArray – expedierea unui set de valori reale.

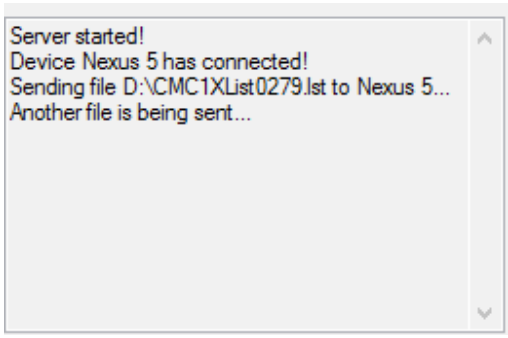


Figura 5.3: Trimiterea fișierului cu pachete

În Figura 5.3, se poate observa că un dispozitiv cu numele „Nexus 5” s-a conectat cu succes, iar apoi s-a inițiat trimiterea fișierului cu pachete. De asemenea, în timpul transferului, s-a încercat trimiterea altui fișier, de unde și mesajul de atenționare cum că un fișier este deja transferat în acel moment.

```
thermalmonitor I/word:  >EE0E1E#
thermalmonitor D/PPROCESSING:  Processer thread
thermalmonitor I/myCRC:  8576 : 8576
thermalmonitor I/word:  >QV0010#
thermalmonitor D/PPROCESSING:  Processer thread
thermalmonitor I/myCRC:  1817 : 1817
thermalmonitor I/word:  >QV1000#
thermalmonitor D/PPROCESSING:  Processer thread
thermalmonitor I/myCRC:  8c81 : 8c81
thermalmonitor I/word:  >QV21B9#
thermalmonitor D/PPROCESSING:  Processer thread
thermalmonitor I/myCRC:  d3c4 : d3c4
thermalmonitor I/word:  >QV323A#
thermalmonitor D/PPROCESSING:  Processer thread
```

Figura 5.4: Procesarea pachetelor - depanare

În Figura 5.4, este reprezentată procesarea pachetelor primite în timpul depanării aplicației mobile în *Android Studio.* Se pot observa următoarele elemente de interes:

- În dreptul etichetei *myCRC,* este verificată corespondența dintre CRC-ul calculat pentru secvența de mai jos și CRC-ul primit pentru aceeași secvență.
- În dreptul etichetei word, este secvența de date primită, încadrată de delimitatori.
- În dreptul etichetei *Processing*, este anunțat faptul că pachetul a ajuns pe firul de execuție pentru procesare.

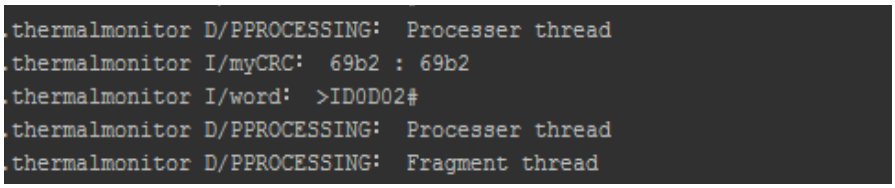


Figura 5.5: Procesarea pachetelor - depanare

- În dreptul etichetei *FragmentThread*(Figura 5.5), este indicat faptul că informațiile procesate au ajuns la fragmentele, ce le vor afișa corespunzător. Acest pas are loc atunci când s-a procesat un set complet de pachete, adică atunci când se primește al doilea pachet consecutiv cu ID-ul egal cu valoarea „ID”. Se poate observa, deasupra etichetei *ProcesserThread*, că ultimul pachet primit îndeplinește această condiție.

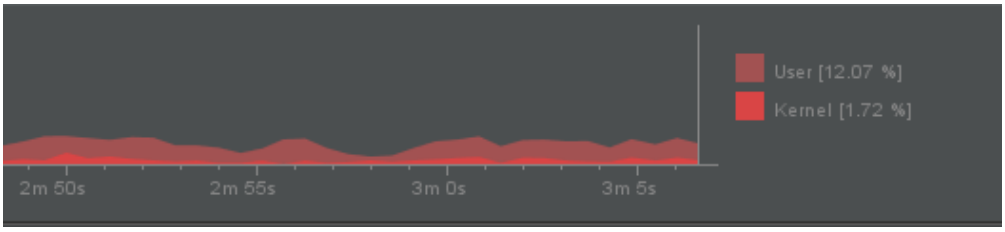


Figura 5.6: Utilizarea procesorului

În Figura 5.6, se pot observa procentele de utilizare a procesorului și evoluția acestora, în timpul unei perioade uzuale de monitorizare (primire, procesare pachete, desenare grafice pe interfață). Acest test a fost realizat pe un dispozitiv „Nexus 5”. În mod normal, procentul indicat în imagine este obținut dacă se utilizează și funcția de scroll în cadrul cu graficele, întrucât implică operații suplimentare de redesenare efectuate de sistem. În rest, utilizarea procesorului, nu depășește 10%, pe acest dispozitiv. De asemenea, trebuie luat în considerare că sunt utilizate resurse și pentru depanarea prin cablul USB.

Pentru fișierul de date utilizat, se poate observa în Figura 5.7, că graficele au ajuns aproximativ la valoarea 261 pe axa X. În acest moment, fișierul de date a fost trimis în totalitate, iar graficele s-au oprit din evoluție. Acest lucru indică faptul că fișierul conține valori de stare pentru aproximativ 261 x 2 = 522 de puncte, deoarece frecvența de reprezentare a punctelor pe grafic este de 2 Hz.

Pentru determinarea corectitudinii reprezentării grafice, se pot compara valorile din cadrul *Values* al aplicației cu valoarea pe axa Y a graficului în momentul respectiv.

În funcție de dispozitivele pe care se rulează aplicația și de clasa de putere a protocolului Bluetooth, poate varia distanța maximă de la care se pot primi date. Este indicat, însă, să se încerce inițial de la o distanță de maxim 3 metri.

După cum este descris și în capitolul anterior, pentru instalarea aplicației este necesară rularea pachetului *ThermalMonitor.apk* salvat în dispozitiv sau în mod *Debug,* cu dispozitivul conectat printr-un cablu USB.

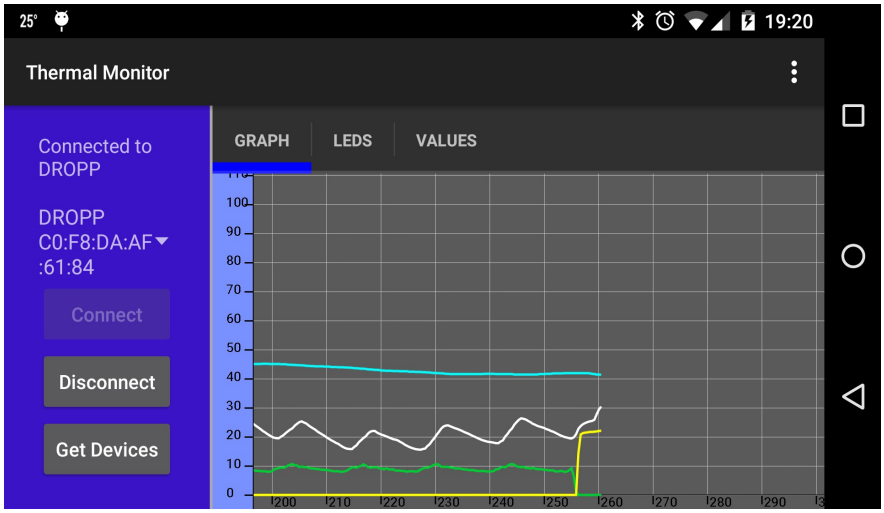


Figura 5.7: Oprirea reprezentării grafice în momentul în care s-a primit întreg fișierul

Concluzii

În cadrul proiectului, a fost realizată aplicația pentru dispozitive mobile cu sistem de operare Android, care să monitorizeze comportamentul, stările și mărimile implicate în procesele unei centrale termice. Aplicația a reușit să reprezinte atât valorile numerice aferente mărimilor, cât și evoluțiile acestora, prin intermediul unor reprezentări grafice, construite pe măsură ce au fost primite date de la unitatea de control a centralei termice.

S-a reușit atât extragerea și reprezentarea informațiilor, cât și afișarea grafică, în timp real, pentru următoarele mărimi ale unei centrale termice: temperatura pe circuitul de termoficare – tur și retur, temperatura măsurată în tancul de apă – pentru doi senzori diferiți, temperatura măsurată a gazelor de evacuare, viteza de rotație a ventilatorului, temperatura la exterior, presiunea măsurată în circuitul de încălzire, tensiunea de ionizare a gazelor de evacuare.

Proiectul a fost implementat de-a lungul a trei mari module: modulul de comunicații Bluetooth, modulul de parsare al datelor respectând un anumit protocol de împachetere, Comunicațiile care au fost utilizate pentru transmiterea datelor sunt reprezentate de standardul Bluetooth. După primirea unui set complet de pachete, informațiile extrase din acestea sunt reprezentate corespunzător. Stările sunt afișate sugestiv prin aprinderea unor led-uri grafice aferente stărilor curente, valorile mărimilor sunt reprezentate tabelar și, de asemenea, sunt adăugate la graficele corespunzătoare acestora. Pentru ca experiența utilizatorului să fie plăcută, acesta va putea alege ce grafice vor fi afișate sau va putea schimba culoarea acestora.

Pentru partea de testare a fost implementat un program pentru PC, care trimite datele către aplicația de monitorizare. Datele sunt pachete primite de la centrală într-o sesiune anterioară. Astfel s-a putut urmări recepția, parsarea și reprezentarea corectă a mesajelor primite prin Bluetooth.

Aplicația este una foarte ușor de utilizat, cu o interfață „prietenoasă”, accesibilă oricărui utilizator care cunoaște noțiuni de bază despre o centrală termică. Graficele sunt desenate pe interfață într-un mod foarte sugestiv și pot fi customizate după preferințele utilizatorului. Pentru o observare mai clară se pot alege pentru afișare, la orice moment din timp, numai graficele care prezintă interes pentru utilizator. De asemenea, pentru a îmbunătăți experiența, acesta poate schimba oricând culorile fiecărui grafic în parte.

Cu toate că au fost realizate, în mare, obiectivele acestui proiect, există și potențiale direcții de dezvoltare a acestuia. Deoarece comunicația este bidirecțională, se pot trimite mesaje care să reprezinte comenzi pentru centrala termică. Se poate comanda recepția numai a anumitor date de interes, sau se pot chiar trimite comenzi pentru a modifica anumite stări și setări ale centralei termice.

Numărul de dispozitive ce rulează platforma Android e în creștere de la an la an, deci orice aplicație nou dezvoltată va fi o resursă pentru multitudinea de utilizatori a acestor dispozitive. Aplicația dezvoltată poate fi, de asemenea un beneficiu, pentru specialiștii în centrale termice, dar și pentru persoanele care doresc să țină evidența tuturor lucrurilor din jurul lor.

Bibliografie

[1] tutorialspoint, Java - Overview [Online], Disponibil la adresa: http://bit.ly/1LSD1xc, Accesat: 2014.

[2] Herbert Schildt, „Java - A Beginner's Guide - Sixth Edition”, Oracle Press, 2013.

[3] tutorialspoint, Android Architecture [Online], Disponibil la adresa: http://bit.ly/1FNpmCt, Accesat: 2015.

[4] Marko Gargenta, „Learning Android”, O'Reilly, 2011.

[5] Google Inc., Application Fundamentals [Online], Disponibil la adresa: http://bit.ly/1j9o2Rx, Accesat: 2014.

[6] itcsolutions, Concepte, Activități și Resurse ale unei aplicații Android [Online], Disponibil la adresa: http://bit.ly/1HvDQOG, Accesat: 2014.

[7] Google Inc., Activities [Online], Disponibil la adresa: http://bit.ly/1hDWTcL, Accesat: 2014.

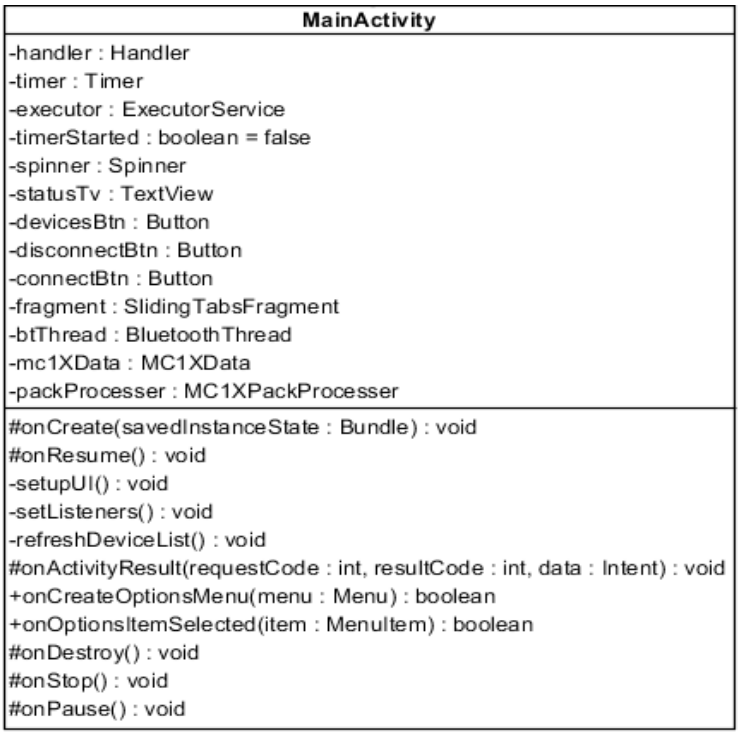
[8] Google Inc., Fragments [Online], Disponibil la adresa: http://bit.ly/1kZPuVr, Accesat: 2014.

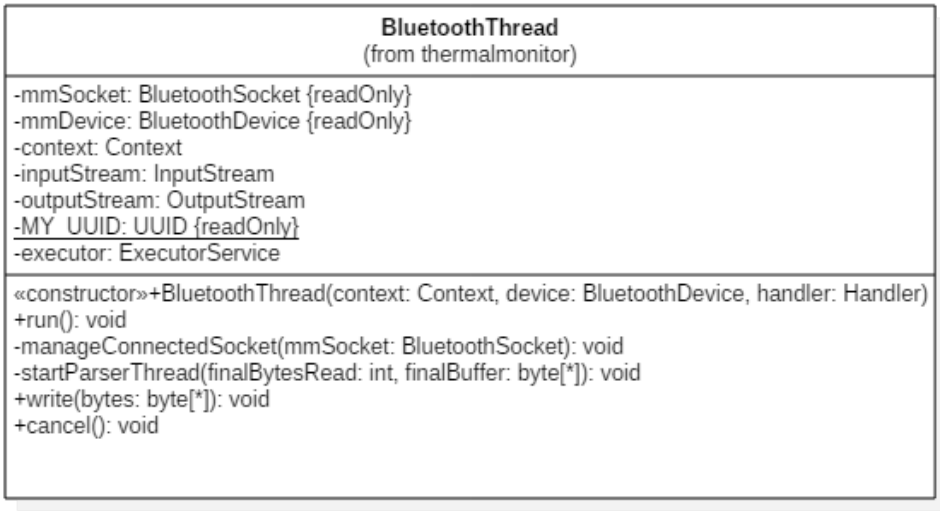
[9] Ciprian Comșa, „Introducere în Bluetooth”, Universitatea Tehnică „Gh. Asachi ” Iași Facultatea de Electronică şi Telecomunicaţii , 2001.

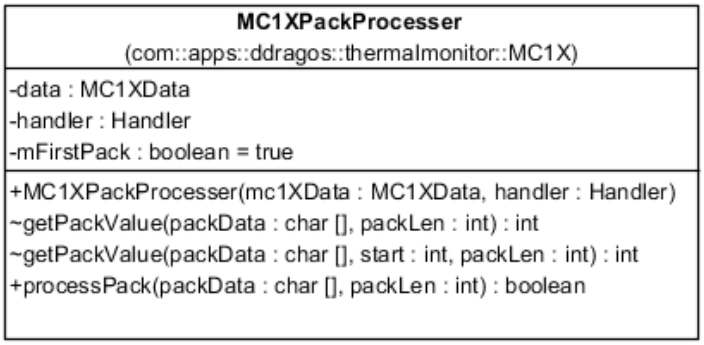
[10] Codeplex, 32feet.NET Documentation [Online], Disponibil la adresa: http://bit.ly/1NswfPS, Accesat: 2014.

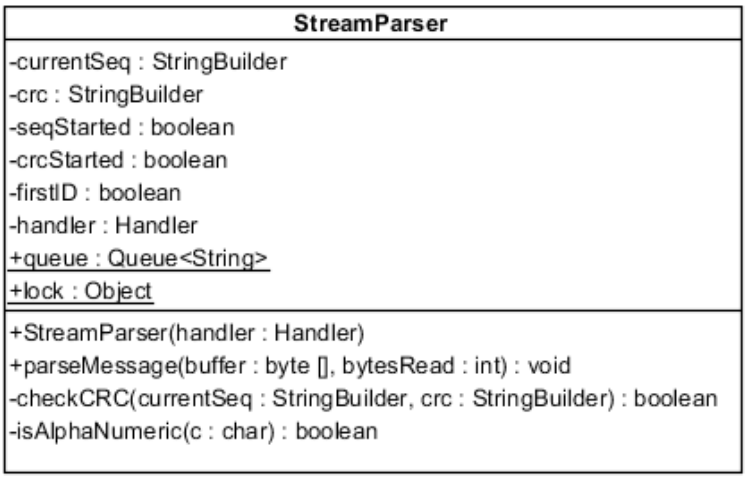
Anexe

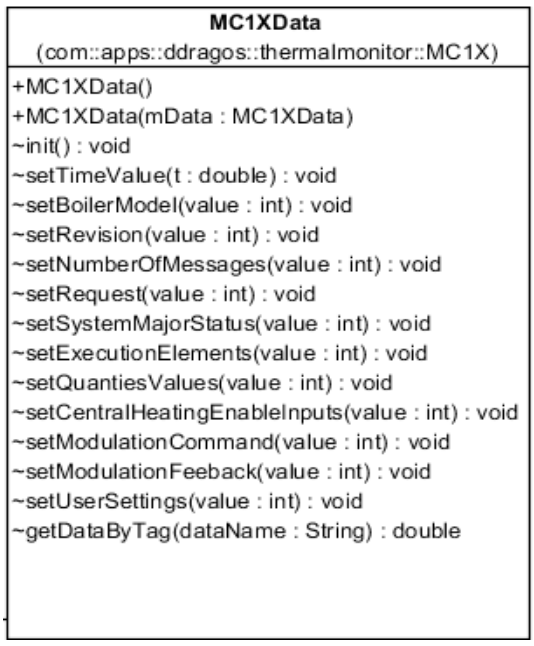
Anexa 1. Diagrame UML ale claselor

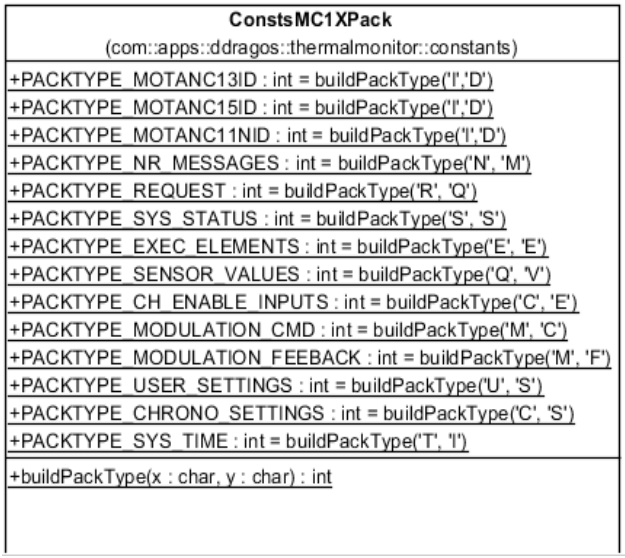


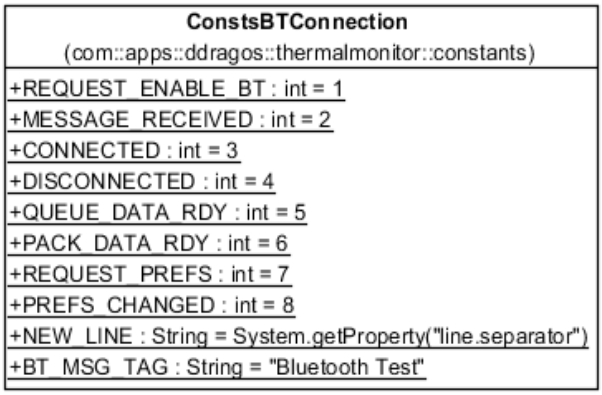


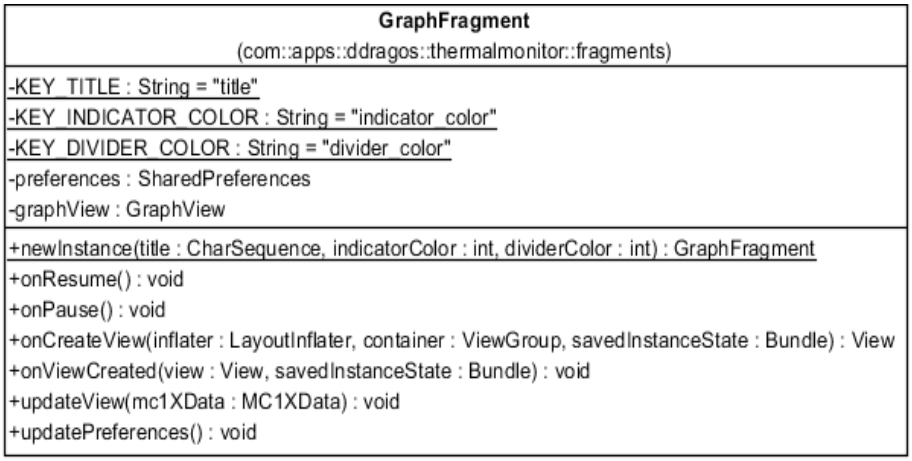

MC1XPackProcesser
(com::apps::ddragos::thermalmonitor::MC1X)
-data : MC1XData
-handler : Handler
-mFirstPack : boolean = true
+MC1XPackProcesser(mc1XData : MC1XData, handler : Handler)
~getPackValue(packData : char [], packLen : int) : int
~getPackValue(packData : char [], start : int, packLen : int) : int
+processPack(packData : char [], packLen : int) : boolean


StreamParser
-currentSeq : StringBuilder
-crc : StringBuilder
-seqStarted : boolean
-crcStarted : boolean
-firstID : boolean
-handler : Handler
+queue : Queue<String>
+lock : Object
+StreamParser(handler : Handler)
+parseMessage(buffer : byte [], bytesRead : int) : void
-checkCRC(currentSeq : StringBuilder, crc : StringBuilder) : boolean
-isAlphaNumeric(c : char) : boolean


MC1XData
(com::apps::ddragos::thermalmonitor::MC1X)
+MC1XData()
+MC1XData(mData : MC1XData)
~init() : void
~setTimeValue(t : double) : void
~setBoilerModel(value : int) : void
~setRevision(value : int) : void
~setNumberOfMessages(value : int) : void
~setRequest(value : int) : void
~setSystemMajorStatus(value : int) : void
~setExecutionElements(value : int) : void
~setQuantiesValues(value : int) : void
~setCentralHeatingEnableInputs(value : int) : void
~setModulationCommand(value : int) : void
~setModulationFeeback(value : int) : void
~setUserSettings(value : int) : void
~getDataByTag(dataName : String) : double


ConstsMC1XPack
(com::apps::ddragos::thermalmonitor::constants)
+PACKTYPE_MOTANC13ID : int = buildPackType('I','D')
+PACKTYPE_MOTANC15ID : int = buildPackType('I','D')
+PACKTYPE_MOTANC11NID : int = buildPackType('I','D')
+PACKTYPE_NR_MESSAGES : int = buildPackType('N', 'M')
+PACKTYPE_REQUEST : int = buildPackType('R', 'Q')
+PACKTYPE_SYS_STATUS : int = buildPackType('S', 'S')
+PACKTYPE_EXEC_ELEMENTS : int = buildPackType('E', 'E')
+PACKTYPE_SENSOR_VALUES : int = buildPackType('Q', 'V')
+PACKTYPE_CH_ENABLE_INPUTS : int = buildPackType('C', 'E')
+PACKTYPE_MODULATION_CMD : int = buildPackType('M', 'C')
+PACKTYPE_MODULATION_FEEBACK : int = buildPackType('M', 'F')
+PACKTYPE_USER_SETTINGS : int = buildPackType('U', 'S')
+PACKTYPE_CHRONO_SETTINGS : int = buildPackType('C', 'S')
+PACKTYPE_SYS_TIME : int = buildPackType('T', 'I')
+buildPackType(x : char, y : char) : int


ConstsBTConnection
(com::apps::ddragos::thermalmonitor::constants)
+REQUEST_ENABLE_BT : int = 1
+MESSAGE_RECEIVED : int = 2
+CONNECTED : int = 3
+DISCONNECTED : int = 4
+QUEUE_DATA_RDY : int = 5
+PACK_DATA_RDY : int = 6
+REQUEST_PREFS : int = 7
+PREFS_CHANGED : int = 8
+NEW_LINE : String = System.getProperty("line.separator")
+BT_MSG_TAG : String = "Bluetooth Test"


GraphFragment
(com::apps::ddragos::thermalmonitor::fragments)
-KEY_TITLE : String = "title"
-KEY_INDICATOR_COLOR : String = "indicator_color"
-KEY_DIVIDER_COLOR : String = "divider_color"
-preferences : SharedPreferences
-graphView : GraphView
+newInstance(title : CharSequence, indicatorColor : int, dividerColor : int) : GraphFragment
+onResume() : void
+onPause() : void
+onCreateView(inflater : LayoutInflater, container : ViewGroup, savedInstanceState : Bundle) : View
+onViewCreated(view : View, savedInstanceState : Bundle) : void
+updateView(mc1XData : MC1XData) : void
+updatePreferences() : void

Graph (com::apps::ddragos::thermalmonitor::graph)
-screenHeight : int -screenPoints : ArrayList<PointF> -maxSize : int -size : int -unitSize : float -offsetX : float -offsetY : float <<Property>> -points : ArrayList<PointF> <<Property>> -path : Path <<Property>> -visible : boolean = true <<Property>> -color : int = 0x000000 <<Property>> -yScale : float = 1f <<Property>> -paint : Paint
+Graph(unitSize : float) +Graph(color : int, yScale : float, unitSize : float, screenHeight : int, offsetX : float, offsetY : float) +Graph(points : ArrayList<PointF>, unitSize : float) +addPoint(point : PointF) : void +removePoint(position : int) : void +generateScreenPoints(offsetX : float, offsetY : float) : ArrayList<PointF> +generateSin(min : float, max : float, unitSize : float) : ArrayList<PointF> -addScreenPoint() : void -refreshPath() : void -addToPath() : void

GraphView (com::apps::ddragos::thermalmonitor::graph)
~axisPaint : Paint ~graphPaint : Paint ~textPaint : Paint ~gridPaint : Paint ~frequency : int = 2 ~scale : float = 1f ~yOffset : int = 30 ~startTime : int = 0 ~endTime : int = 300 ~currentTime : float = 0 ~metrics : DisplayMetrics ~graphMap : HashMap<String, Graph> ~graph : Graph
+GraphView(context : Context) +GraphView(context : Context, attrs : AttributeSet) +GraphView(context : Context, attrs : AttributeSet, defStyleAttr : int) -initDummyGraph() : void -init() : void -initGraphs() : void -drawAxis(canvas : Canvas) : void -initPaint() : void +onDraw(canvas : Canvas) : void #onMeasure(widthMeasureSpec : int, heightMeasureSpec : int) : void +updateGraphs(mc1XData : MC1XData) : void +setGraphsPreferences(preferences : SharedPreferences) : void +setColors() : void -setGraphsColors(preferences : SharedPreferences) : void -setGraphsScale(preferences : SharedPreferences) : void -setGraphsVisibility(preferences : SharedPreferences) : void

Anexa 2. Codul sursă

Aici este listat codul sursă pentru câteva din principalele clase dezvoltate. Codul complet se găsește pe CD-ul atașat lucrării.

MainActivity.java

```
public class MainActivity extends FragmentActivity {


    private BluetoothThread btThread;
    private Handler handler;
    private Timer timer;
    private ExecutorService executor;
    private MC1XData mc1XData;
    private MC1XPackProcesser packProcesser;
    private boolean timerStarted = false;

    //UI
    private Spinner spinner;
    private TextView statusTv;
    private Button devicesBtn;
    private Button disconnectBtn;
    private Button connectBtn;

    private SlidingTabsFragment fragment;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        if (savedInstanceState == null) {

            fragment = new SlidingTabsFragment();
            getSupportFragmentManager()
                    .beginTransaction()
                    .add(R.id.tabs_fragment, fragment)
                    .commit();
        }

        handler = new BTHandler(Looper.myLooper());
        setupUI();

        PreferenceManager.setDefaultValues(this, R.xml.preferences, false);
    }

    @Override
    protected void onResume() {
        super.onResume();
        if (fragment == null)
        {
            fragment = new SlidingTabsFragment();
            getSupportFragmentManager()
```

```
                    .beginTransaction()
                    .replace(R.id.tabs_fragment, fragment)
                    .commit();
        }

    }

    private void setupUI() {
        spinner = (Spinner) findViewById(R.id.spinner);
        statusTv = (TextView) findViewById(R.id.statusTv);
        devicesBtn = (Button) findViewById(R.id.devicesBtn);
        disconnectBtn = (Button) findViewById(R.id.disconnectBtn);
        connectBtn = (Button) findViewById(R.id.connectBtn);

        setListeners();
    }

    private void setListeners() {
        devicesBtn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.d("", "CONNECT");
                refreshDeviceList();
            }
        });

        disconnectBtn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                if (btThread != null && btThread.isAlive()) {
                    btThread.cancel();
                    statusTv.setVisibility(View.GONE);
                    disconnectBtn.setEnabled(false);
                    connectBtn.setEnabled(true);
                }
            }
        });
    }

    private void refreshDeviceList() {
                ArrayAdapter<String>  mArrayAdapter  =  new  ArrayAdapter<>(this,
R.layout.device_item);
                  final        BluetoothAdapter        mBluetoothAdapter        =
BluetoothAdapter.getDefaultAdapter();

        if (mBluetoothAdapter == null) {
              Toast.makeText(this, "Bluetooth is not supported on this device.",
Toast.LENGTH_LONG).show();
        } else {

            try {
                if (!mBluetoothAdapter.isEnabled()) {
                                                        Intent   intent   =   new
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
                    startActivityForResult(intent, REQUEST_ENABLE_BT);
                }
```

```
                            final List<BluetoothDevice> pairedDevices = new
ArrayList<>(mBluetoothAdapter.getBondedDevices());

                if (pairedDevices.size() > 0) {

                    for (BluetoothDevice device : pairedDevices) {
                                mArrayAdapter.add(device.getName() + "\n" +
device.getAddress());
                    }

                    spinner.setAdapter(mArrayAdapter);
                    spinner.setVisibility(View.VISIBLE);

                    connectBtn.setOnClickListener(new View.OnClickListener() {
                        @Override
                        public void onClick(View v) {
                            mBluetoothAdapter.cancelDiscovery();
                            if (btThread == null || !btThread.isAlive()) {
                               int position = spinner.getSelectedItemPosition();
                          btThread = new BluetoothThread(getApplicationContext(),
pairedDevices.get(position), handler);
                                btThread.start();
                            } else
                                Toast.makeText(getApplicationContext(), "Already
connected", Toast.LENGTH_SHORT).show();
                            }
                    });
                } else
                    spinner.setVisibility(View.GONE);

            } catch (Exception e) {
                e.printStackTrace();
                Toast.makeText(this, e.getMessage(), Toast.LENGTH_SHORT).show();
            }
        }
    }

      protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
            switch (requestCode) {
                case REQUEST_ENABLE_BT:
                    if (resultCode == RESULT_OK) {
                        Log.i(BT_MSG_TAG, "Bluetooth enabled!");
                        refreshDeviceList();
                    }
                    break;
                case REQUEST_PREFS:
                    if (resultCode == PREFS_CHANGED) {
                        if (fragment != null) {
                            fragment.updatePreferences();
                        }
                    }
                    break;
            }
    }
```

```
    private class BTHandler extends Handler {

        public BTHandler(Looper looper) {
            super(looper);
        }

        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case CONNECTED:
                    BluetoothDevice device = (BluetoothDevice) msg.obj;
                    Toast.makeText(getApplicationContext(),  "Connected  to  "  +
device.getName(), Toast.LENGTH_SHORT).show();
                    statusTv.setText("Connected to " + device.getName());
                    statusTv.setVisibility(View.VISIBLE);

                    if (mc1XData == null)
                        mc1XData = new MC1XData();
                    if (packProcesser == null)
                        packProcesser = new MC1XPackProcesser(mc1XData,handler);

                    disconnectBtn.setEnabled(true);
                    connectBtn.setEnabled(false);
                    executor = Executors.newFixedThreadPool(1);
                    break;
                case MESSAGE_RECEIVED:
                    break;
                case DISCONNECTED:
                    disconnectBtn.setEnabled(false);
                    connectBtn.setEnabled(true);
                    Toast.makeText(getApplicationContext(), "Disconnected from "
+ ((BluetoothDevice) msg.obj).getName(), Toast.LENGTH_SHORT).show();
                    statusTv.setVisibility(View.GONE);

                    /*if (timerStarted) {
                        timer.cancel();
                        timerStarted = false;
                        timerBtn.setEnabled(false);
                        timerBtn.setText("Timer");
                    }*/
                    break;
                case QUEUE_DATA_RDY:

                    Queue<String> queue = StreamParser.queue;
                    String data = null;

                    synchronized (StreamParser.lock) {
                        try {
                            data = queue.remove();
                        } catch (NoSuchElementException ex) {
                            Log.i("Queue", "Empty Queue");
                        }
                    }

                    final String fData = data;
```

```
                    if (data != null) {
                        executor.execute(new Runnable() {
                            @Override
                            public void run() {
                                Log.d("PPROCESSING","Processer thread");
                                   packProcesser.processPack(fData.toCharArray(),
fData.length());
                            }
                        });
                       /* new Thread(new Runnable() {
                            @Override
                            public void run() {

                            }
                        }).start();*/
                    }

                    break;
                case PACK_DATA_RDY:
                    if (fragment != null) {
                        fragment.updateFragments(mc1XData);
                    }
                    break;
                default:
                    break;
            }
        }

    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        if (btThread != null)
            btThread.cancel();
    }

}
```

StreamParser.java

```
public class StreamParser {

    private StringBuilder currentSeq = null;
    private StringBuilder crc = null;
    private boolean seqStarted = false;
    private boolean crcStarted = false;
    private boolean firstID = false;

    private Handler handler = null;

    public static Queue<String> queue = null;
    public static final Object lock = new Object();
```

```
    public StreamParser(Handler handler) {
        this.handler = handler;
    }


    public synchronized void parseMessage(byte[] buffer, int bytesRead) {
        String sequence = new String(buffer,0,bytesRead);

        for (int i = 0; i < sequence.length(); i++) {
            char c = sequence.charAt(i);

            switch (c) {
                case '>':
                    seqStarted = true;
                    currentSeq = new StringBuilder();
                    currentSeq.append('>');
                    break;
                case '<':
                    if (currentSeq != null && currentSeq.length() >= 3) {
                        crcStarted = false;

                        if (currentSeq.substring(1,3).equals("ID")) {
                            if (!firstID) {
                                firstID = true;
                                if (queue == null)
                                    queue = new LinkedList<>();
                            }
                        }

                        //crc check + insertion in queue
                        if (firstID) {
                            if (checkCRC(currentSeq,crc)) {
                                synchronized (lock) {
                                    queue.add(currentSeq.toString());
                                     handler.obtainMessage(ConstsBTConnection.QUE
UE_DATA_RDY, null).sendToTarget();
                                    Log.i("word", currentSeq.toString());
                                }
                            }
                        }

                    }
                    break;
                case '#':
                    if (seqStarted) {
                        seqStarted = false;
                        crcStarted =  true;
                        currentSeq.append('#');
                        crc = new StringBuilder();
                    }

                    break;
                default: break;
            }

            if (isAlphaNumeric(c))
```

```
            {
                if (seqStarted)
                    currentSeq.append(c);
                else if (crcStarted)
                    crc.append(c);
            }
        }

    }

    private boolean checkCRC(StringBuilder currentSeq, StringBuilder crc) {
        try {
     int computedCRC = new CRC().crcFromString(currentSeq.toString());
    int receivedCRC = Integer.parseInt(crc.toString(),16);
     Log.i("myCRC",Integer.toHexString(computedCRC)     +     "     :     "     +
Integer.toHexString(receivedCRC));


            if ((computedCRC ^ receivedCRC) == 0)
                return true;
        } catch (NumberFormatException e) {
            e.printStackTrace();
            Log.d("CRC INT", crc.toString());
        }

        return false;
    }

    private boolean isAlphaNumeric(char c) {
        char ch = Character.toLowerCase(c);
        return (ch >= 'a' && ch <= 'z') || Character.isDigit(ch);
    }

}
```

Graph.java

```
public class Graph {

    private int screenHeight;
    private ArrayList<PointF> points;
    private ArrayList<PointF> screenPoints;
    private Path path;
    private int size;
    private int maxSize;
    private float unitSize;
    private float yScale = 1f;
    private boolean visible = true;
    private int color = 0x000000;

    private float offsetX;
    private float offsetY;
    private Paint paint;
```

```
    public Graph(float unitSize) {
        points = null;
        screenPoints = null;
        path = null;

        points = new ArrayList<>();
        screenPoints = new ArrayList<>();

        this.unitSize = unitSize;
        this.offsetX = 0;
        this.offsetY = 0;
    }

    public Graph(int color, float yScale, float unitSize, int screenHeight,float
offsetX, float offsetY) {
        points = null;
        screenPoints = null;
        path = null;

        this.yScale = yScale;
        this.color = color;
        this.unitSize = unitSize;
        this.screenHeight = screenHeight;
        this.offsetX = offsetX;
        this.offsetY = offsetY;

        this.paint = new Paint();
        this.paint.setColor(color);
        this.paint.setStyle(Paint.Style.STROKE);
        this.paint.setStrokeWidth(5);
        this.paint.setAntiAlias(true);

    }

    public Graph(ArrayList<PointF> points,float unitSize) {

        this.points = points;
        size = points.size();

        this.screenPoints = null;
        this.path = null;
        this.unitSize = unitSize;
    }

    public void addPoint(PointF point) {
        if (points == null)
            this.points = new ArrayList<>();
        points.add(point);
        addScreenPoint();
        addToPath();
//        refreshPath();
    }
    public void removePoint(int position)
    {
        points.remove(position);
    }
```

```
    private void addScreenPoint() {
        size = points.size();
        PointF point = new PointF();
        if (screenPoints == null) {
            screenPoints = new ArrayList<>();

           point.x      =      offsetX      +      points.get(0).x*unitSize      +
Math.abs( points.get(0).x ) * unitSize;
            point.y = -offsetY + screenHeight - points.get(0).y*unitSize*yScale;
            screenPoints.add(point);
            Log.d("Point0", point.x + " " + point.y);
        }
        else if (size > 1)
        {
            int i = size - 1;

            point.x = offsetX + screenPoints.get(i-1).x + unitSize;
            point.y = -offsetY + screenHeight - points.get(i).y*unitSize*yScale;

            screenPoints.add(point);

            Log.d("Point",point.x + " " + point.y);
        }


    }

     public ArrayList<PointF> generateScreenPoints(float offsetX, float offsetY)
{

        size = points.size();
        screenPoints = new ArrayList<>();

//        float difX = points.get(size - 1).x - points.get(0).x;
//        float xUnitSize = ((screenWidth*scale) / difX);

        screenPoints.set(0, new PointF());
          screenPoints.get(0).x    =    offsetX    +    points.get(0).x*unitSize    +
Math.abs(points.get(0).x)*unitSize;
           screenPoints.get(0).y     =     -offsetY     +     screenHeight     -
points.get(0).y*unitSize*yScale ;

        for (int i = 1; i < size; i++ ) {
            screenPoints.set(i, new PointF());
           screenPoints.get(i).x = offsetX + screenPoints.get(i-1).x + unitSize;
          screenPoints.get(i).y     =     -offsetY     +     screenHeight     -
points.get(i).y*unitSize*yScale ;

        }

        return screenPoints;
    }

    public ArrayList<PointF> generateSin(float min, float max, float unitSize) {
```

```
        size = (int)((max-min)/unitSize);
        points = new ArrayList<>();

        PointF point = new PointF(min,(float)Math.sin(min));
        points.add(point);

        for (int i=1; i<size; i++) {
            point = new PointF();
            point.x = points.get(i - 1).x + unitSize;
            point.y = (float) Math.sin(points.get(i).x);
            points.add(point);

            Log.i("Sine", "X: " + points.get(i).x + "   Y: " + points.get(i).y);
        }

        size = points.size();

        return points;
    }

    private void refreshPath() {
        path = new Path();
        path.moveTo(screenPoints.get(0).x,screenPoints.get(0).y);

        for (int i=1; i<size; i++) {
            path.lineTo(screenPoints.get(i).x, screenPoints.get(i).y);
        }
    }

    private void addToPath() {
        if (path==null) {
            path = new Path();
            path.moveTo(screenPoints.get(0).x,screenPoints.get(0).y);
        }
        else
            path.lineTo(screenPoints.get(size-1).x,screenPoints.get(size-1).y);
    }

    public Path getPath() {
        return path;
    }


    public float getYScale() {
        return yScale;
    }

    public void setYScale(float yScale) {
        this.yScale = yScale;
    }
    public boolean isVisible() {
        return visible;
    }

    public void setVisible(boolean visible) {
        this.visible = visible;
```

```
    }

    public int getColor() {
        return color;
    }
    public void setColor(int color) {
        this.color = color;
        this.paint.setColor(color);
    }
    public ArrayList getPoints() {
        return points;
    }
    public void setPoints(ArrayList points) {
        this.points = points;
    }
    public void setPath(Path path) {
        this.path = path;
    }
    public Paint getPaint() {
        return paint;
    }
}
```

Alte clase importante precum *GraphView, MC1XData, BluetoothThread, LedsFragment, ValuesFragment,* n-au fost incluse în anexă din motive de lizibilitate a liniilor abundente de cod, dar pot fi consultate pe CD-ul atașat.