Introducere

Evoluția, inteligența și goana după confort au fost unele dintre cele mai importante motive pentru translația de la mersul pe jos, la călărit, la căruță și apoi mașini cu abur, urmate de cele pe ardere internă. De când omul a "descoperit" mașina, tot timpul s-au găsit lucruri de îmbunătățit. De la o bancă din lemn la un scaun ergonomic, de la o roată din lemn, la un pneu susținut de telescoape și un întreg chit de suspensie, de la un volan greu de rotit la sisteme servo, de la un felinar pe ulei la faruri led și laser. Evoluția acestor sisteme nu prea pare să ajungă la o limită cunoscută.

Industria auto, fiind o industrie foarte mare, înca de la început a fost domintată de anumiți giganți. Așa încât încă de la apariția motorului cu ardere internă, un număr mic de corporații au împărțit piața auto între ei. Și pentru că a existat din totdeauna această luptă pentru a câștiga piața, au investit în cercetare. Asa au început să apară tehnologii noi, scopurile principale fiind siguranța și confortul utilizatorilor.

Odată cu apariția calculatorului, au început să aleagă controlul electronic al componentelor, peste cel mecanic, fiind mult mai sigur și fiabil. Așa a luat naștere în industria automotiv utilizarea componentelor electronice. Odată cu aceste componente, a apărut nevoia da protocoale automotive, concepte și programatori specializați în domeniu.

În ultimii ani, prin conceptul de „internet of things" a început să se digitalizeze și mai mult această industrie. Lucrarea mea de licentă vine ca o continuare logică a evoluției. Mobile Control, prezintă o posibilă variantă de comunicare între om și mașină, via Wi-fi, în care constructorul de masină poate să ofere utilizatorului control și observație asupra anumitelor componente integrate în mașină.

Ideea este simplă. ECU-urile integrate pe mașină comunică printr-un protocol numit CAN, și pentru comunicare folosesc mesaje de tip CAN. Mobile Control se comportă ca un ECU (trimite și el mesaje CAN), doar că de pe o tabletă Android, și pentru a avea acces la rețeaua în care sunt conectate ECU-urile, se foloseste de un server montat pe un Raspberry Py.

Conceptul nu este nou, dar ca și contribuție personală, am venit cu ideea unei interfete generate dinamic, creată specific pe ce „suportă" componentele integrate în automobil.

Contribuții:

Pentru dezvoltarea conceptului Mobile Control a trebuit să învăț multe protocoale, standarde, tehnologii și tool-uri.

Vom prezenta pe scurt contribuțiile aduse pentru lucrarea prezentată:

Dezvoltarea concetului Mobile Control, diagnoză Wi-fi, unde aportul principal legat de concept a fost ideea de a avea o interfață generată dinamic, care la conectarea cu un dispozitiv cu capacitate de server, să primească informații despre automobilul în cauză, și să genereze interfața utilizabilă specific pentru el.

Pentru a demonstra funcționalitatea conceptului a fost dezvoltată o aplicație mobile, (contribuție integral personală), unde au fost implementate functionalitățile de bază pentru a susține conceptul Mobile Control cum ar fi, conexiune cu server-ul (websocket), creare/prelucrare/stocare/trimitere/primire de mesaj CAN, interfață statică, interfată dinamică, și doar pentru a prezenta concepte mobile au fost create și 2 meniuri mobile.

O contribuție majora este și configurarea mediului de lucru. Fiind o lucrare care este distribuită pe mia multe platforme (Windows, Andoid, Raspian) și având ca una dintre functionaliățti transmiterea de mesaj de tip CAN de pe aplicație Mobile până la un controler electronic din industria automotive (ECU), o provocare a fost să creezi o cale de mijloc între Andoid și ECU. Aici intervine un dispozititv cu capacitățile unui computer, unde au fost montate 2 routere wireless, și a fost configurat și comportamentul lor. Unul a fost configurat ca la pornirea Rasperry-ului să se conecteze cu un router conectat la PC-ul pe care lucram, pentru capabilități de cross-compiler (dezvoltare pe platforma Windows – compilare directă, prin wi-fi, pe Raspian), iar celălalt router era folosit pentru conexiunea cu tableta pe care rula aplicația Mobile Control.

La nivelul serverului pe langă partea de server din websocket-ul descris la nivel de aplicație a fost nevoie de utilizarea interfeței CanInterface, interfață capabilă să folosească capabilitățile modulului de CAN montat pe Raspberry Py.

De menționat (tehnologii și protocoale folosite) :

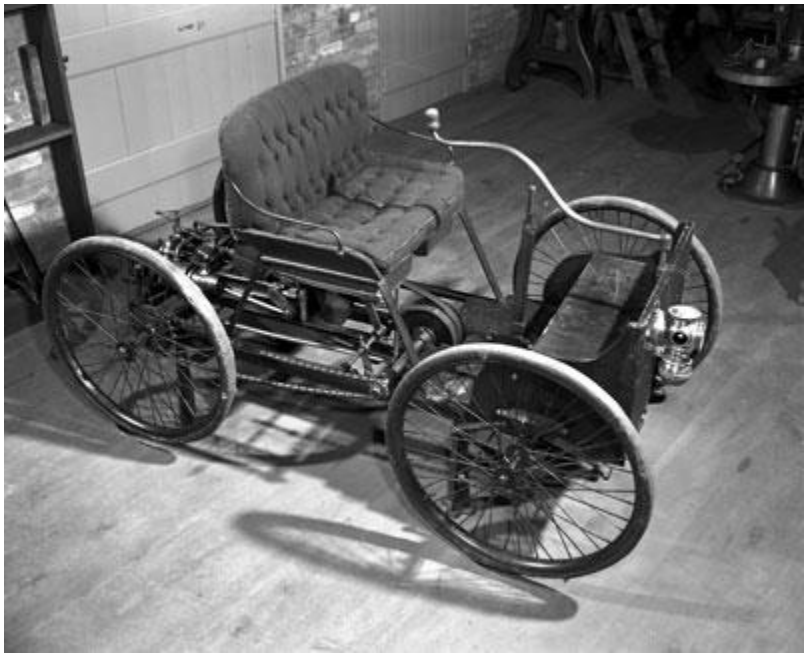
- tehnologie JSON
- CanInterface (interfața dezvoltata pentru functionalități ale protocolului CAN)
- Raspian (sistem de operare pentru Raspberry Py)
- Sincronizare de biblioteci (cygwin)
- Mesaje CAN
- Standard OBD
- Simulare CANoe (unde simulam comportamnetul Bus-ului de CAN)
- Hardware
    - Raspberry Py
    - Sursa de curent
    - ECU (Electronic Control Unit)

Informații ajutătoare:

Industria auto:

Industria automotive a avut debutul în anii 1890 cu sute de vehicule care erau devoltate având tracțiune mecanică.

1896 Henry Ford contruiește un “quadricycle” (foto).



Primul automobil construit de Henry Ford.(1)

Primul vehicul contruit de Henry Ford, contruit pe 4 roti de bicicletă, având un motor estimat la 4 cai putere. Cutia de viteze avea 2 viteze pentru mersul înainte și 0 pentru mersul cu spatele.

În anul 1929, înainte de “Great Depression”, erau aproximativ 32 de milioane de automobile, dintre care 90% erau produse de industria Americană.

În anul 1980 Japonia a preluat titlu de cel mai mare exportator auto din lume, titlu revendicat de China în 2009, un titlu greu de câștigat de altă națiune deoarece China produce anual aproximativ 28 de milioane de unități (2), echivalentul urmatoarelor 3 state din clasament (USA 12 mil, Japonia 9 mil, Germania 6 mil). Alte țări care s-au remarcat prin exportul de mașini : India 4,4 mil, Coreea de Sud 4,2 mil, Mexic 3,5 mil, Canada 2,3 mil. (statistică 2016)

Pentru a evidenția și mai mult dimensiunile acestei industrii, în 2007 erau înregistrate 806 milioane de mașini și camioane care cunsumau aproximativ 980 de miliarde de litri de combustibil anual (3).

Pe parcursul evoluției mașinilor, au aparut ECU-uri specifice automotive, Electronic Control Unit, este un termen aproape generic care se referă la partea de embedded system care controlează unul sau mai multe sisteme sau subsiteme ale vihiculului. Foarte curand au "acaparat" piața automotive, fiecare masină produsă în zilele noastre are un număr impresionant de ECU-uri.

LCU Light Control Unit:

Modul de control al luminilor montate pe un automobil. Controlat electronic, poate să controleze aprinderea și stingerea becurilor/led/laser, dar și controlul oglinzilor (integrate in far) pentru a regla fanta de lumină.

Pentru că industria IT a deschis un nou drum în industria automotive, au apărut noi companii, adevarate corporații care au început să se ocupe de dezvoltarea sistemelor de control electronic, precum Continental, Sony sau Bosch. Acest drum a produs schimbări masive în ce întelegem confort în automobil, siguranță și eficență. De la scaune electrice care se ajustează automat dupa nevoile pasagerului, la efincetizarea folosirii electricității între sistemele electrice montate pe automobil, la Glare Free Lighting care ajustează fanta de lumină produsă de sistemul de lumini pentru un confort maxim al pasagerului și al celorlalti participanți la trafic.

Clădire Contonental Automotive Romania, Iasi

Corporatia Continental, cunoscuta pentru amvelope, a preluat in 2007 Siemens AG, și în 2012 ajungea pe locul 3 în lume la vânzări în industria componentelor automotive(4).

Cu vânzări de peste 34,5 miliarde de euro în 2014, cu peste 200 000 de angajați în 53 de țări. Pe plan local, Continental este unul dintre cei mai mari angajatori, cu peste 20 000 de angajați în mai multe domenii de activitate, printre principale fiind și industria software unde chiar în Iasi sunt aproape 2000 de angajați in 5 departamente de R&D(Research and development)(5)(6).

Pentru a întelege mai bine dimensiunea activității corporației Continental în Romania, de notat este faptul că în anul 2017 s-a produs la fabrica din Timisoara amvelopa cu numărul 200 000 000. Un număr care pune țara noastră în topul țărilor cu export de amvelope notabil(7).

Embedded în automotive:

Dupa cum am spus, în industria auto, pe partea electronică, ECU, Electronic Control Unit, este un termen generic care se referă în general la partea de Embedded System, care controlează una sau mai multe sisteme electrice sau subsisteme întrun automobil.

Tipuri de systeme embedded :

- ECM (Electronic/engine Control Module)
- PCM (Powertrain Control Module)
- TCM (Transmission Control Module)
- BCM (Body Control Module)
- LCU (Lights Control Unit)

Aceste module comunică între ele printr-un protocol automotive, numit CAN Protocol,

folosind o retea numita CAN Bus.

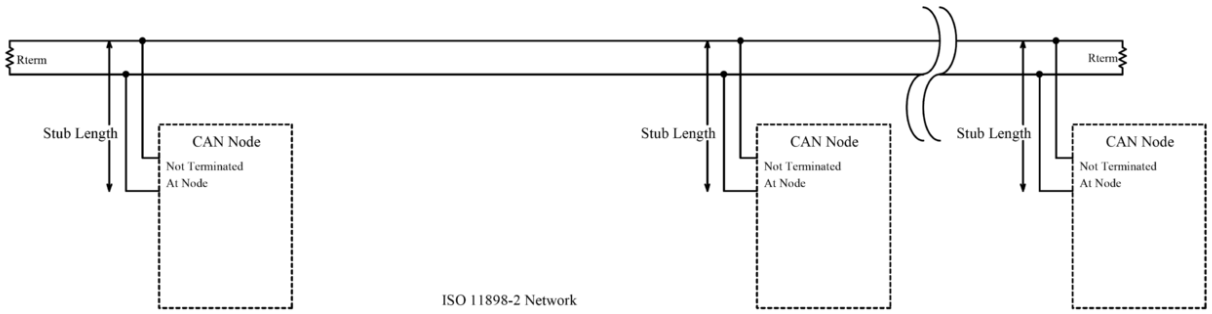
CAN, Controller Area Network, protocol dezvoltat de Robert Bosch în anul 1983. Lansat în anul 1986 la Society of Automotive Engineers(SAE) în Detroit Michigan. Primele cipuri de tip CAN controller au fost produse de Intel și Philips și au ieșit pe piață în anul 1987. În 1988 BMW seria 8 a fost prima masină cu un system bazat pe CAN.

Bosch a publicat mai multe variante de CAN, dar CAN 2.0 a fost publicat în 1991. Care are 2 variante, una cu identificator pe 11 biti (A), și varianta B cu identificator pe 29 de biti, usual numit CAN 2.0B.

CAN bus este un standard care ajută microcontrolerele montate pe masină să comunice între ele. Este un protocol bazat pe mesaje (CAN messages).

Un automobil modern are în medie 70 de componente electronice (ECU), pentru sistemele sale și subsisteme. Deobicei cel mai mare processor este montat pe unitatea care gestioneaza motorul, (Engine Control Unit).

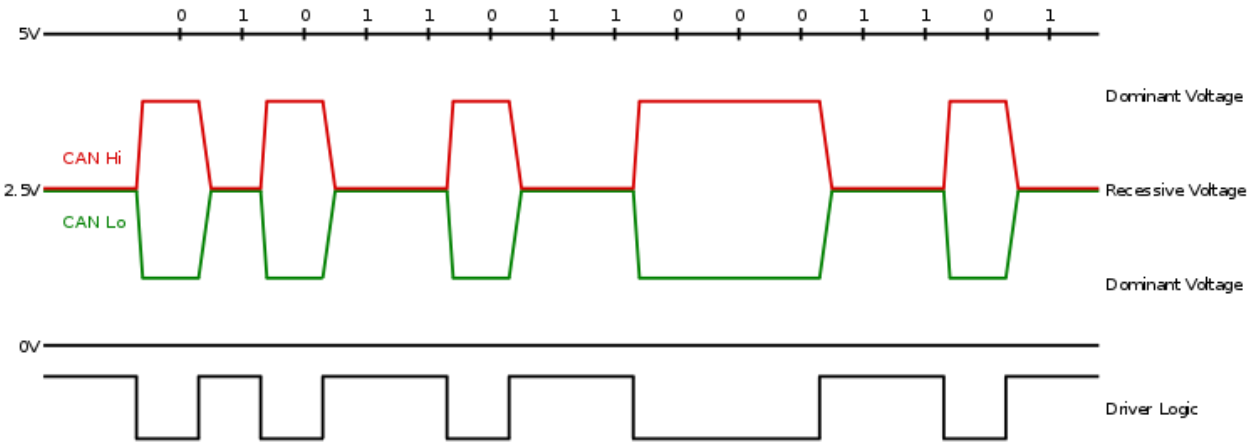
Arhitectura CAN este un multi-master serial bus standard pentru conexiunea componetelor electronice. Două sau mai multe ECU-uri au nevoie de o rețea CAN pentru a comunica. Toate nodurile sunt conectate între ele printr-un "bus" de 2 fire. ISO 11898-2, usual : High Speed CAN, utilizează un BUS linear care la fiecare capăt are un rezistor de 120 de ohmi.(1)



CAN – Control Area Network(8)

High Speed CAN Network. ISO 11898-2

În continuare vom discuta despre CAN high speed. Semnalul High Speed CAN este transmis pe 2 fire. Este foarte diferit de semnaul digital normal. În semnalul digital normal "1" semnifică tensiune pe fir, pe când "0" semnifică absența tensiunii. La High Speed CAN cele două fire sunt denumite "high" și"low". Când este transmis un semnal High Speed CAN, firul "high" trage tensiune de 5 V, și firul "low" coboară la 0 V (după ce inițial amandouă erau la 2,5) toate astea pentru a transmite un "0" dominant. Iar când amandouă sunt cu o tensine de 2.5 V pe bus este transmis un bit "1" dominant. Ca în imaginea ce urmează :



1 și 0 logic în protocolul CAN(9).

Bus-ul de CAN este unul dintre cele 5 protocoale utilizate în “on-board diagnostics (OBD)-II vehicle diagnostics standard”. Standardul OBD-II este obligatoriu pentru toate mașinile de uz personal cât și pentru camioanele ușoare vândute în Statele Unite înca din 1996, iar standardul EOBD este obligatoriu pentru toate mașinile pe benzina vândude în Uniunea Europeana înca din anul 2001, iar pentru mașinile pe diesel din 2004.

Pentru a întelege concepte și utilizări ale standardului ne vom folosii de o mică descriere preluata de pe site-ul : “http://www.e-automobile.ro/categorie-diagnoza/149-diagnoza-auto-obd-citire-parametrii-motor.html” .

“ Producătorii auto sunt obligați să transmită un număr minim de parametrii ai motorului către un dispozitiv de diagnosticare OBD 2. Acești parametrii sunt:

- Turația motorului
- Temperatura lichidului de racire
- Viteza automobilului
- Presiunea combustibilului din rampă (diesel)
- Sarcina calculată a motorului

Pe lângă acești parametrii fiecare constructor poate decide dacă mai adaugă informații adiționale sau nu.

Acești parametrii mai sunt utilizați pentru a memora informații despre starea motorului (freeze frame) în cazul aparitiei unui defect (DTC). În plus pot furniza informații obiective pentru a putea diagnostica o eventuala funcționare defectuoasă a motorului sau pentru a citi istoricul motorului.

De exemplu parametrul cu identificatorul (0x31) ne informează cați km au fost parcurși de la ultima ștergere a unui cod de defect (DTC). Informația poate fi utilă la achiziționarea unui automobil utilizat, pentru a verifica veridicitatea rulajului afișat în bordul automobilului.

Parametrii fizici ai motorului (presiune, debit de aer, presiune combustibil) pot fi inregistrați și apoi reprezentați grafic pentru o mai bună înțelegerea a funcționării motorului, atât în regim de funcționare normală dar mai ales în cazul funcționării defectuoase.”

Am explicat standardul OBD deoarece conceptul Mobile Control este un concept definit în mod special pentru a citii diagnoza .

De ce Android ?

După mii de ani de evoluție a umanității, în ultimul secol oamenii de știință au făcut un salt mare în tot ce înseamnă umanitate. Chiar dacă la început nu se așteptau ca o conglomerație de cabluri, electricitate și porți logice să atingă, să modeleze și chiar să controleze atât de mult din tot cea ce

ne reprezintă. De la agricultură mecanizată și controlata prin internet la operații effectuate pe oameni de roboți medicinali, până la "banalul" smartphone. Ultimul din urma fiind atât de răspândit pe Pământ, încât în zonele dezvoltate ale planetei, aproape fiecare ființă umană mai mare de 5 ani deține unul. Iar dreptul la internet a ajuns : "Potrivit unei rezoluţii a Consiliului pentru Drepturile Omului din cadrul ONU, accesul la internet este un drept uman de bază. (10)"

La bazele evoluției rapide ale informaticii, stă și un român care este considerat părintele Ciberneticii, ramură a informaticii. Meritele lui fiind recunoscute, ca în majoritatea cercetatorilor români, post-mortem. A ajuns atât de apreciat încat în 1990 în Elvetia a fost deschisa în onoarea lui "Academia de Cibernetică din Elveția", iar în România sunt 2 licee și un spital care îi poartă numele(11).

Dupa ce calculatoarele au renuntat să mai ocupe camere întregi și prin initiativele Dell și Microsoft în care liderii aveau în plan (inimaginabil la vremea lor, și fantazmagoric), ca și concept "personal computer", calculator personal. Concept în care toata lumea să aibă acces la un calculator propriu. Desi nu avea sorți de izbandă acest concept a ajuns realitate, și mult mai mult de atat. Acum intr-o masină sunt mai multe calculatoare decât numarul de calculatoare folosit de omenire să trimită primul om pe lună, Apolo 11. Și pe langă PC-uri, "personal computer", aproape fiecare om, cum am mentionat mai sus, deține și un smartphone, care fară doar și poate este un calculator, poate putem să îi spunem un PC miniaturizat.

Deși primul telefon "smart", este considerat de omaneii de stiintă ca fiind prototipul "Angel" dezvoltat de Frank Canova în 1992 (12), primul model a fost lansat de IBM (compania unde lucra Frank), "IBM Simon" pe numele lui.

(13)

Touchscreen, email, și PDA. Nașterea smartphon-ului modern, în perspectiva pieței și a publicului larg, a fost introducerea pe piata mobile a modelelor "iPhone". Încă de la primul model

compania Apple a dat tonul în inovație și design pentru multe din modelele aparute pe piata. Având baza software diferită de toate celelalte modele existente, dar la aceasta vom revenii mai tarziu. De mentionat este că smartphon-urile actuale ajung să aibă specificații exorbitante, cum sunt 12-14 Gb memorie Ram sau 256 Gb memorie Rom, baterii de 12000 mA, camere foto de peste 20 mP, si chiar cu vedere infra-rosu (model de telefon prezentat anul trecut de compania Caterpilar.

Telefon mobil dotat cu capacitati infra-rosu.(14).

Având toate funcțiile de bază, conexiune la internet, creare și editare de fisiere, capacități de conectare la monitoare, imprimante și tot felul de periferice, capabile de calcule numerice și doatate cu putere de procesare care în trecutul apropiat erau inimaginabile și pentru stațiile PC ultramoderne, putem trage concluzia clară că acest concept de "Personal computer" care dupa cum am menționat parea utopic, a fost depășit considerabil, și înca nu a ajuns la o bariera sau o limitare fizica în dezvoltare și inovație, deci, e foarte probabil ca în timpul vieții nostre să observăm modificări substanțiale ale smartphon-ului și modului în care acesta ne afectează viețile.

Având complexitate hardware, de la începuturile calculatorului a aparut necesitatea unui sistem de operare, "OS – Operatin Sistem", un sistem care să facă ușoară manipularea resurselor și a puterii de procesare oferită de hardware.

Primul OS a fost facut de General Motors, GM Operatin sistem, pentru IBM 701 în 1955.(15). Dar cel care a venit cu primul model de OS usor de folosit de publicul larg a fost cel care a ramas 2 decenii în topul celor mai bogați oameni din lume, mult timp fiind pe primul loc. Bill Gates, CEO Microsoft, a venit cu ideea unui Operatin Sistem "user frendly", Acesta idee și tot software-ul produs de Microsoft a propulsat compania pe primele locuri în IT înca de la înființare, până în zilele noastre.

Este timpul să împărțim OS-urile în 2 categorii. OS-uri pentru PC-uri (tradiționale) și OS-uri pentru aria "mobile, arie larg dezvoltată. Pentru PC-uri sunt 3 mari OS-uri folosite în ziua de azi, Linux, Windows, și macOs. Linux a fost gandit să fie gratis, open source, și este folosit mai

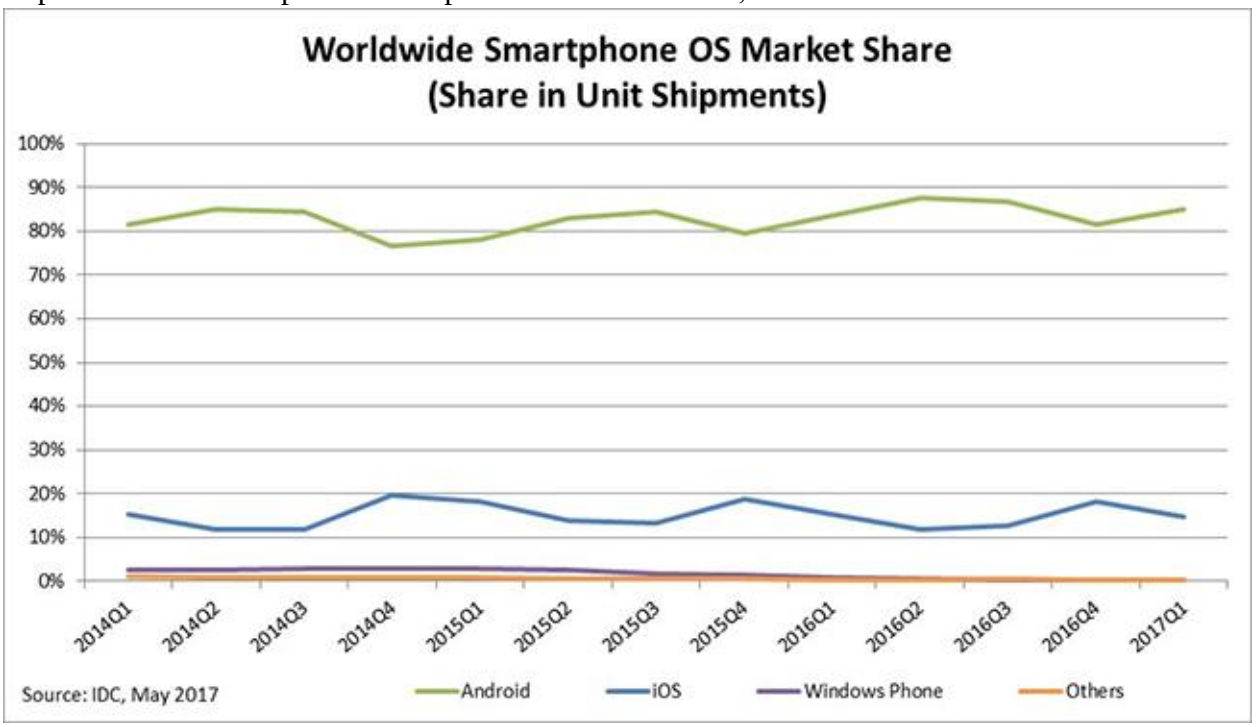
mult în dezvoltare. Windows, OS-ul dezvoltat de Microsoft, a rămas favoritul pieței, datorită suportului oferit de producator, și varietății tool-urilor dezvoltate pentru acest OS, este preferat și de companii. Iar macOs, sistem de operare oferit de Apple doar pentru PC-urile fabricate de ei, este un sistem de operare fiabil și gratuit, bazat pe Unix.

În aria mobile, au fost multe OS-uri la început, aproape ca fiecare producator încerca un OS propriu. OS-uri ca : Symbian Ltd, Nokia S40,Palm OS, Meamo OS, au fost printre primele sisteme de operare pentru mobile. BlackBerry OS a câștigat mult teren în trecut pe partea de securitate și era considerat clasa business. Dar în trecutul apropiat sau modelat, definitivat ca și concurenți principali pe piața OS-urilor mobile, două OS-uri, Android și iOS. Aproape că este un război taboo, ceva ce definește generația tânără, această alegere între Android și iOS. iOS este sistemul de operare bazat pe C dezvoltat și oferit gratis de Apple pentru toate device-urile produse de ei, în aria mobile. Fiind bazat pe C, un limbaj foarte puternic și rapid, OS-ul este unul fiabil, și foarte puternic. Neavând nevoie de prea multe resurse, Apple a întarziat dotarea smartphon-urilor produse de ei cu specificații echivalente pieței mobile. După cum am specificat, era doar o chestiune de necesitate, OS-ul fiind bazat pe un limbaj care nu necesită resurse multe, nu au adaugat resurse degeaba, ex: iPhone 4s are 512 mb memorie Ram cand majoritatea companiilor din anul în care a fost lasnat iPhone 4s, care au lansat smarphon-uri high-end, au folosit 1 Gb memorie Ram. Ce mai e de remarcat la acest OS este securitatea foarte buna oferita. Odata furat un smartphone dotat cu iOS, daca avea un cont de iCloud(cont de autentificare pe platforma Apple), acel telefon este inutil fara parola de autentificare. Pentru a putea fi folosit, trebuie, ori autentificat, și modificat din setări un nou proprietar, ori schimbată placa de bază complet.

Încă o mențiune, aplicațiile pentru acest OS, sunt găsite în Apple Store, un magazin foarte bine gestionat, iar aplicațiile stocate, sunt mai sigure decat pe alte magazine, datorită testelor prin care trebuie să treacă fiecare aplicație înainte să poată fi urcată în magazin.

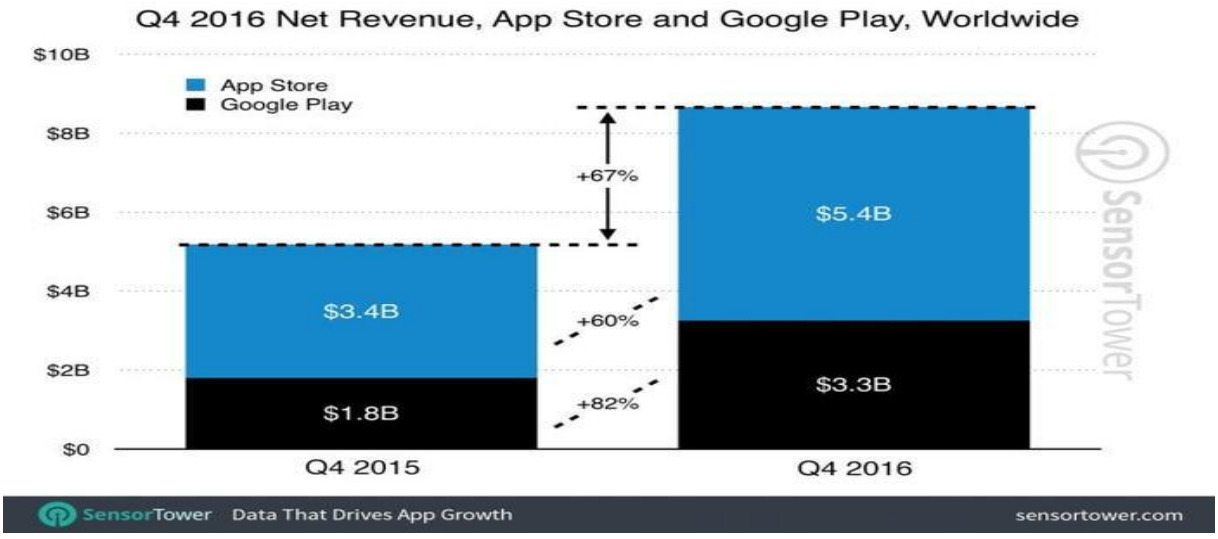
În 2003, în California, SUA, avea să se nască cel mai folosit OS din toate timpurile. Android. Produl de compania Android Inc. în Octombrie, Andy Rubin, Rich Miner, Nick Sears, și Chris White, lansau prima versiune. Doar 2 ani mai târziu, în 2005 Google, actualul cel mai mare gigant IT din lume, a cumparat Andorid pentru doar 50 de milioane de dolari. Ajungând sub marca Google să valoreze peste 2 miliarde de dolari. Bazat pe java, este un OS relativ greoi, care consumă resurse și are nevoie de hardware, cum am menționat mai sus. Principalul magazin pentru aplicații Android este Google Play, unde poti urca aplicații relativ usor, fiind necesar un cont de developer, care necesită la rândul lui o taxa de 20 de dolari. Lansând versiuni noi de Android aproape anual, de menționat este că numele versiunilor sunt luate din dulciuri celebre și urmează ordinea alfabetică pentru aliterație. Ex Marshmello, Nughat, Oreo.. etc.

Pentru a vedea o distribuție reală a OS-urilor în aria mobile am luat de pe : https://www.idc.com/promo/smartphone-market-share/os, urmatoarea statistică :



Distribuția sistemelor de operare mobile.(16)

Se vede clar că Android-ul acaparează o mare parte din piață, dar un lucru interesant și demn de monționat este că deși deține 80-90% din piața mobile, banii din aplicații mobile majoritatea sunt în App Store, de la Apple, pentru iOS.



Distribuția banilor din aplicatii pe sistemele de operare mobile.(17)

Concept mobile control.

Conceptul de Mobile Control la nivel de idee este foarte puternic: control și verificare de la distanță prin intermediul internetului. Având 2 componente principale: aplicație mobile și server c++. Aplicația este concepută cu o interfață generică, în sensul în care primește de la server un json care cuprinde informații legate de ce functionalități sunt implementate în ECU-ul care urmează să fie verificat/controlat, și genenrează o interfată care cuprinde tot ce poate fi afectat la nivel de ECU. Având interfață generică nu este nevoie de o aplicație specifică pentru fiecare ECU. Serverul este montat pe o component electronică care poate fi conectată cu mașina (protocol OBD), sau direct pe canalul de comunicație la care sunt conectate ECU-urile mașinii(protocol CAN).

- Mufa OBD
- Aplicație de diagnoză wi-fi

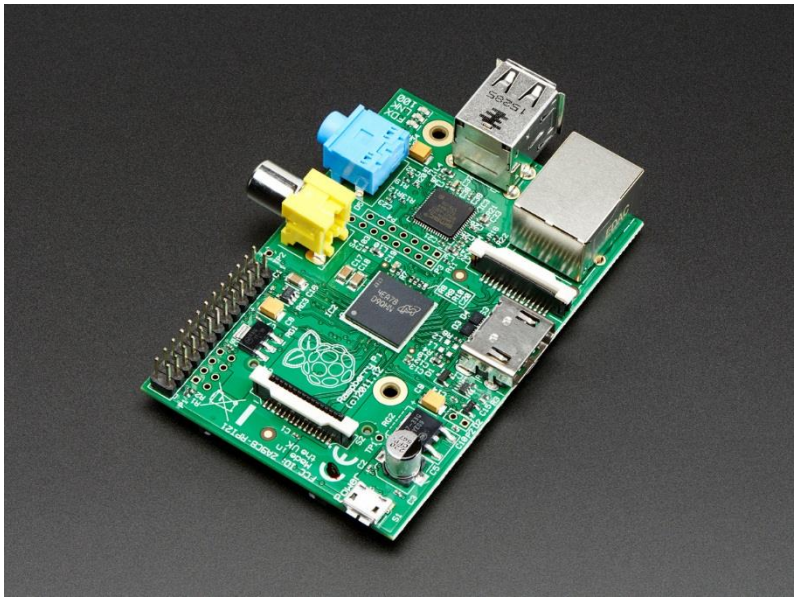
Având în minte conceptul prezentat mai sus, urmează să intrăm în detalii legate de implementarea conceptului, tool-uri folosite și echipament.

Tableta Samsung Tab 3.(18)

Am folosit pentru aplicația mobile o tabletă de la Samsung (Samsung Tab3), capabilă de conexiune wi-fi, și cu o versiune de android. Am folosit pentru dezvoltarea aplicatiei Andorid Studio. Pentru stocarea serverului de C++ am folosit un Raspberry py 1, iar pentru testare efectiva un ECU care a fost conceput pentru controlul unui far.

Seria Raspberry Pi reprezinta o serie de “small single-board computers” dezvoltate în Regatul Unit de catre Raspberry Pi Fundation, pentru a promova predarea informaticii de bază în școli și în țările în curs de dezvoltare. Modelul principal a devenit mult mai popular decat se anticipase, cu vanzari massive în afara pieței țintă, cum s-a întamplat în cazul roboticii. Nu include periferice (tastaturi, carcase ..etc).Potrivit Fundatiei Raspberry Pi, peste 5 milioane de plăcuțe Raspberry Pi au fost vândute până în Februarie 2015, devenind cel mai bine vândut

calculator britanic. Până în Noiembrie 2016 au vândut 11 milioane de unități, devenind al trei-lea best-selling "general purpose computer". În Martie 2018 vânzările au atins 19 milioane.(19)(20)



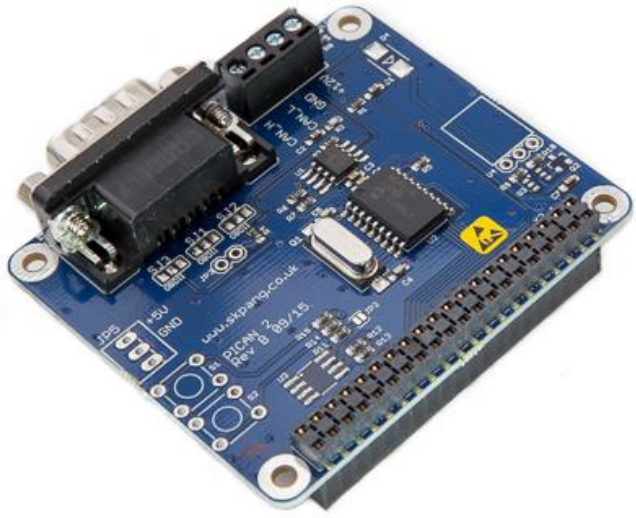
Raspberry Py 1. (21)

Modelul folosit în dezvoltarea conceptului Mobile Control face parte din seria 1. Modelul Raspberry Pi model B.Specificațiile acestui model fiind : 512 MB RAM, 2 porturi USB2, 40 de pini GPIO(Genereal Purpose Input/Output), și port de Eternet, slot pentru SD, și port HDMI.

Router Wi-fi. (22)

Router Wi-fi.(23)

Pentru comunicarea cu tableta am montat pe Raspberry Pi un modul extern de wi-fi. Am configurat OS-ul (Raspbian, versiune de Unix specială pentru Raspberry) ca la pornire să pornească un hotspot wi-fi la care te poți conecta cu orice dispozititv capabil wi-fi, în cazul nostru tabletă. Pentru aceasta am urmat pașii descrisi în mini-tutorialul : https://www.raspberrypi.org/documentation/configuration/wireless/access-point.md. Comenzi clasice de linux.



Modul PiCan (Modul CAN pentru Raspberry Pi).(24)

Pentru cominucarea cu Bus-ul de CAN, am folosit un modul de CAN, PiCAN – CAN Interface for Raspberry Pi. PiCAN oferă capabilități Controller Area Network(CAN) pentru

Raspberry Pi. Folosește microcipul MCP2515 CAN controller cu MCP2551 CAN transceiver. Conexiunea este făcută via DB9 sau 3-way screw terminal. Este usor de montat pe Rasberry Py și programarea lui poate fi făcută în Python sau C. Features : CAN v2.0 A/B la 1 MB/s, 120 ohm rezistor terminal + multe altele.

Cross – Compile:

Pentru a ușura procesul de dezvoltare a conceptului, am folosit o metoda de compilare mai complexa. Cross-compile între platformele Windows și Linux. Deși a durat foarte mult sincronizarea bibliotecilor de C, compilarea cross-compile a fost un succes. Am instalat pe stația doatată cu Windows un modul extern de wi-fi. Cum am notat mai sus, Raspberry Pi-ul cand pornește este configurat să genereze hotspot. Stația dotată cu Windows capabilă acum la conexiune Wi-fi, s-a conectat cu Raspberry Pi, am folosit tool-ul Cygwin pentru sincronizarea bibliotecilor, iar pe scurt, din Eclipse(Windows) puteam să fac direct debug pe Raspberry Pi, unde a fost instalat un simplu server C, dotat cu intervață : CanInterface.

Implementare concept.

Pana acum am prezentat concepul din exterior, componente și tehnologii.

Mobile Control Implementare:

Având în minte istoricul și detaliile prezentate mai sus urmează să intrăm în implementarea efectivă a conceptului.

Aplicațiile Android sunt compuse din 2 componente prnicipale : user interface și control.

User interface pentru o aplicație Android este împtartit la rândul lui în componenete denumite activități. Grafica putând fi ajustata direct în tool-ul folosit, în cazul nostru Android Studio, unde componentele integrate în interfață pot fi adaugate și mișcate în activitate folosind un simplu mouce. Dar pentru lucru de finețe, Andoid Studio ofera fișiere .xml, asemănătoare celor de CSS din programamrea web. Aceste fișiere .xml sunt generate din motorul care gestionează crearea obiectelor din interfață despre care am vorbit mai sus, dar sunt editabile și prin modificarea valorilor generate sau adăugarea de tag-uri și valori, poți gestiona interfața exact cum dorești. Pentru a înțelege descrierea, vom explica o bucata de cod xml dintr-un fisier de tip .xml care se ocupă de gestionarea interfeței legate activității de “static_menu”, activitate la care vom revenii in descriere ulterior.

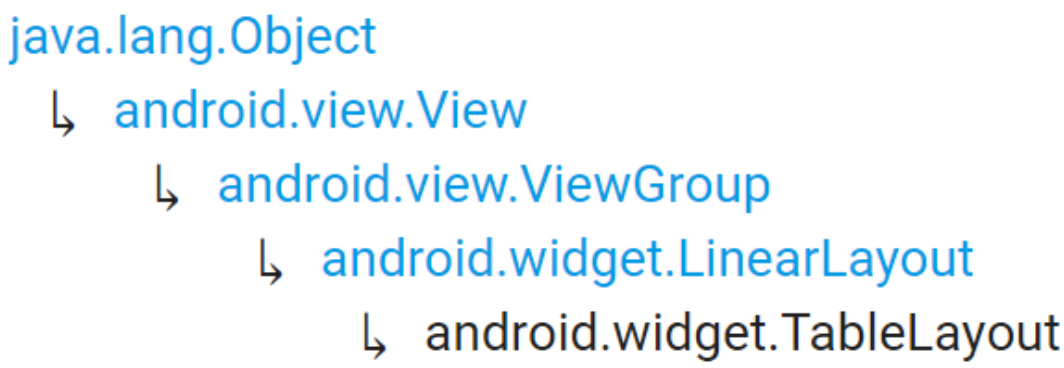
```
<TableLayout
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="3">

    <TableRow android:gravity="left"
        android:stretchColumns="2">
        <TextView android:text="Componenta"/>
        <ImageView
            app:srcCompat="@drawable/conti_logo"
            android:layout_width="45dp"
            android:layout_height="45dp"
            />
        <Button android:text="Vezi stare"/>
        <TextView android:text="Stare"
            android:gravity="center"/>
    </TableRow>
```

Static-menu.xml

Vom explica exemple de tag-uri, parametri și valori folosite în interfața "static_menu.xml".

Înainte să începem putem menționa un concept de java, fiecare obiect este moștenit la rândul lui din alt obiect recursiv, iar la rădăcină se află clasa Object. Pentru primul tag pe care îl vom explica ierarhia arată în felul urmator:



Android obiect ierarhie.(25)

Tagu-uri:

- TableLayout – layout care aranjează "copii"(cildrens) - descendenții în rânduri și coloane.
- TableRow – layout care asează descendenții la nivel orizontal.
- TextView – element din interfață care afisează text static, pentru text editabil se va folosii EditText.
- Button – element din interfață pe care utilizatorul îl poate apăsa ("click or tap") pentru a genera o acțiune.
- ImageView – afisează imagini stocate în directorul de resurse al aplicații

Parametri:

- layout_width – clasă care exprimă "înălțimea" tag-ului în care este setat.

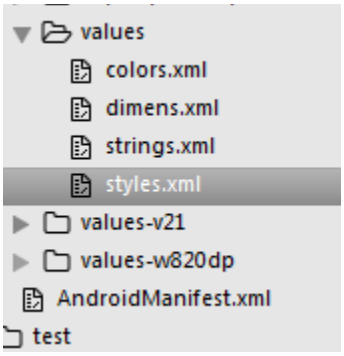
- layout_height – clasa care exprimă “lățimea” tag-uli în care este setat.
- Gravity – folosit pentru plasarea unui obiect într-un conteiner posibil mai mare, containerul fiind ajustabil, obiectul care are “gravitatea” setată va lua dinamic o poziție în funcție de valoarea specificată acestei clase.
- stretchColumns – index pentru numarul de coloane.
- srcCompat – metoda de a întegra imagini, sau resurse din categoria “drawables” în interfată.
- Text – folosit pentur specificarea textului stocat de obiect.

Valori:

- Pentru “layout_width” și “layout_height” sau orice alt layout moștenit din ViewGroup.LayoutParams sunt 2 metode principale prin care putem specifica poziția, una este mai relativă : “fill_parent” și valori asemănătoare care specifică valori relative tag-ului parinte, iar acesta specifică că obiectul va umple spațiul liber din tag-ul parinte. Iar o prezentare mai exacă a poziției sau a dimensiunilor se descrie în dp.
- Pentru “gravity”, valorile sunt valori care exprimă pozițitonare relativă în obiectul părinte: “left”, “right”, “top”, “botom”
- Pentru “srcCompat” se va folosi numele resursei de tip “drowable” din directorul de resurse, în exemplul nostru primește valoarea : “conti_logo”.

În dezvoltarea aplicației am acumulat 9 fisiere de tip .xml pentru gestioanrea interfeței aplciației Mobile Control, 7 pentru activitățile aplicației și 2 pentru test și pentru implementarea unui meniu.

Resurse de tip values:



Ierarhie fisiere ce stochează resurse de tip values:

```xml
<resources>
    <string name="app_name">MobileControl</string>
    <string name="action_settings">Settings</string>
    <string name="title_activity_second">SecondActivity</string>
    <string name="title_activity_login">Sign in</string>

    <!-- Strings related to login -->
    <string name="prompt_email">Email</string>
    <string name="prompt_password">Password (optional)</string>
    <string name="action_sign_in">Sign in or register</string>
    <string name="action_sign_in_short">Sign in</string>
    <string name="error_invalid_email">This email address is invalid</string>
    <string name="error_invalid_password">This password is too short</string>
    <string name="error_incorrect_password">This password is incorrect</string>
    <string name="error_field_required">This field is required</string>
    <string name="permission_rationale">"Contacts permissions are needed for providing email
        completions."
    </string>
</resources>
```

Cod .xml ce stochează resurse de tip values:

În aceste fișiere de tip .xml avem stocate resurse cu valori predefinite. Putem definii aici tot felul de resurse, fiecare resursă are un nume iar funcțional putem afla valoarea resursei/folosii valoarea resursei, doar prin numele dat. Mai sus sunt prezentate valorile predefinite în .xml-ul "strings.xml", unde pentru a explica funcționalitatea prezentăm prima resursă, numită :"app_name" căruia i-am atribuit valoarea "MobileControl", acum putem folosi "app_name" ca și resursă oridecateori avem nevoie de numele aplicației care este :"MobileControl".

Pentru a face tranziția de la interfață la partea de control al aplicației, vom definii conceptul de meniu, și vom explica implementarea lui în aplicația Mobile Control.

Meniu andoid

Pentru meniu, au fost create două fișiere de tip .xml separate care se ocupa de interfața meniurilor. Un fișier "static_menu", altul "menu_main". Iar imprementarea funcționalităților meniului "static_menu.xml" a fost facută în fișier-ul de tip java care este asignat fiecărui activități care utilizează meniul descris de "static_menu.xml". Iar fișerul java care gestionează activitatea principală, conține implementarea funcționalităților meniului descris de "menu_main.xml".

Pentru meniul definit în "menu_main.xml" au fost făcute urmatoarele setări :

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="com.example.uidk9044.mobilecontrol.Activitys.MainActivity">

    <item
        android:visible="false"
        android:id="@+id/action_settings"
        android:orderInCategory="100"
        android:title="@string/action_settings"
        app:showAsAction="never" >
        <menu >
            <item android:title="Font" />
            <item android:title="Dimensions" />
            <item android:title="Sizes" />
        </menu>
    </item>

    <item
        android:id="@+id/action_show_ir_list"
        android:icon="@drawable/conti_logo"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:title="Icon_Conti"
        app:showAsAction="always"
        android:orderInCategory="101" />
    <item
        android:id="@+id/about_app_id"
        android:title="About app"
        app:showAsAction="always"/>
    <item
        android:id="@+id/login_id"
        android:title="Login"
        app:showAsAction="always"/>

</menu>
```

Meniul este unul generic conceput mai mult pentru a prezenta acest concept tipic aplicațiilor mobile. Conține 3 mari obiecte, unul dintre ele, primul, fiind la rândul lui un sub-meniu care conține la rândul lui 3 obiecte.

Primul obiect prezent în meniu este un sub-meniu care a fost gândit ca și concept (neimplementat) pentru setările aplicației. Și prezintă doar scheletul pe care se paote construi funcționalități de tipul : selectare font, selectare mărime, selectare culori …etc.

Al doilea obiect este o imagine, preluată după cum am descris mai sus din directorul cu obiecte de tip “drowable”, este un obiect de tip PNG, și prezintă sigla companiei unde am dezvoltat conceptul Mobile Control.

Al treilea și al patrulea obiect din lista prezintă 2 butoane simple, text, care sunt gândite pentru a direcționa utilizatorul care apasă pe buton spre activitatea notată în textul afisat.

Sau folosit taguri de tip “item” și de tip “menu”. Cum este intuitiv un tag de tip “menu” este gândit să conțină unul sau mai multe tag-uri de tip “item”.

Parametri notabili din acest fișier ar fi:

- android:id, folosit pentru a identifica unic obiectul .
- icon, care specifică numele obiectului de tip”drowable”.
- showAsAction, specifică în context dacă obiectul să fie sau nu vizibil tot timpul.
- title, care reprezintă textul afișat de interfață, vizibil pentru utilizator.

Dupa cum am spus, controlul pentru meniu este implementat în fiecare activitate care afisează acel meniu, pentru examplificare vom folosii în contrinuare meniul definit în “menu_main.xml”, definit în ce ține de funcționalitate în fișierul “MainActivity.java”, fișier care intuitiv deține implementarea funcționalităților definite pentru activitatea principală (activitatea care se deschide la deschiderea aplicației în OS-ul Android).

În continuare vom prezenta cele mai improtante 2 funcții implementate pentru funcționalitățile meniului:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}
```

MenuInflater este clasa care este folosită pentru instanțierea meniurilor stocate în fișiere de tip .xml (in cazul nostru “menu_main”), în obiecte de tip Menu (în cazul nostru obiectul a fost denumit generic ”menu”, și se poate vedea în antetul funcției “onCreateOptionMenu(Menu menu)”).

Iar funcționalitatea principală a meniului este definită în funcția :

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    switch (item.getItemId()) {
        case R.id.login_id:
            Intent intent = new Intent(getApplicationContext(), LoginActivity.class);
            startActivity(intent);
            return true;
        case R.id.about_app_id:
            //showHelp();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Pe scurt codul spune că la selecția item-ului cu id-ul “login_id” se va folosi un obiect de tip Intent care va afișa user-ului activitatea LoginActivity, “pornind” activitatea prin intermediul funcției “startActivity(inteint)”.

În cazul în care userul apasă pe item-ul care are id-ul “about_app_id”, nu se va întampla nimic.

Controlul.

Înainte de a intra în detaliile legate de orice funcționalitate specifică Mobile Control vom explica câteva din funcțiile specifice activităților Android:

- void onCreate()
- void onDestroy()
- void onPause()
- void onStop()

Funcționalități specifice fiecărei activități. În care putem să definim acțiuni în contextul activității. Când e creată, când e distrusă, când e în pauză…etc. Pentru demonstrație vom folosii codul funcției onCreate() din fișierul StaticMenuActivity.java, în care la crearea activității se încearca o conectare la server, printr-un websocket (tehnologie la care vom revenii în continuarea acestei lucrări).

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.static_menu);
    //try to connect
    if (conn.IsConnected() == false){
        conn.connectWebSocket();
    }

    connect_text_id = (TextView) findViewById(R.id.connect_text_id);
    if (conn.getErr().equals("Conected")){
        //connect_text_id.setText("Connected");
        conn.Conecteaza();
        Log.i("cenect","conectat");
    }
    else {
        connectionError();
        Log.i("cenect","neconect");
    }
    TrimiteMesajCiclic();
}
```

Funcția onCreate – concept activitate Android

Controlul activităților și implementarea funcționalităților aplciației Mobile Control au fost dezvoltate în fișiere java, deobicei câte un java file pentru fiecare activitate. În plus au fost create alte 5 fișiere .java care stochează implementarea funcționalităților de manipulare a datelor (json & CAN message) și de conexiune cu serverul (websocket).

După cum am specificat mai sus, aplicația Mobile Control a fost dezvoltată pentru prezentarea unui concept de wi-fi diagnosis.

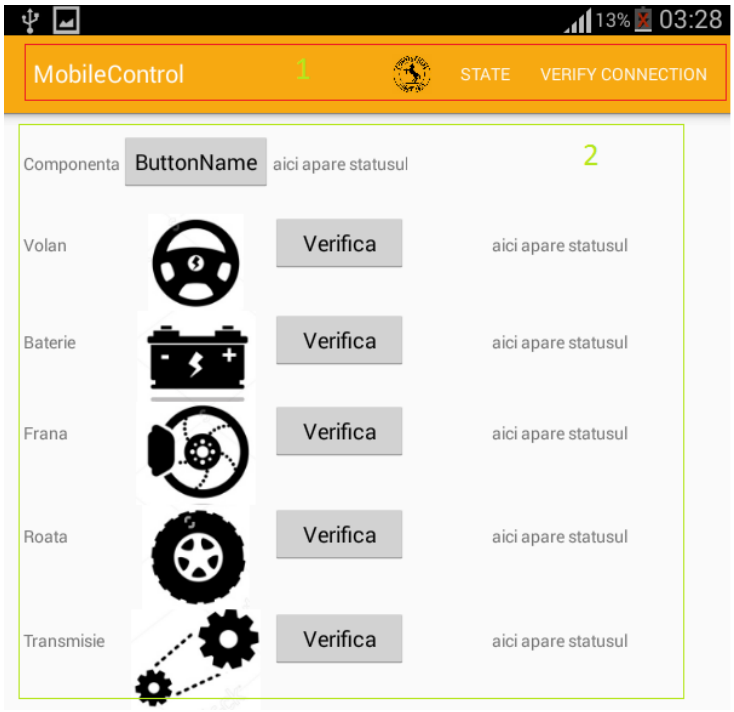
Aplicația este dezvoltată cu 2 “meniuri”, nu meniuri cum am explicat mai sus, meniuri native, ci meniuri care stochează funcționalități, sau schelet de funcționalitate pentru concept. Unul dintre meniuri este static, ceea ce înseamnă că fiecare imagine și fiecare buton, și tot ce ține de interfata grafică a fost adaugat din engine-ul care ajută developerul la creat interfață grafică, și funcționalitățile trebuie implementate separat în fișierul .java asignat activității în care au fost adăugate obiectele din interfață.

Dar meniul cu numarul 2, meniul dinamic prezintă conceptul de meniu generic. Având în fișierul de configurare a interfeței, fișier de tip .xml doar 2 tag-uri-obiect un “ScrollView” și un “TableLayout”, primește în cod informații despre ce obiecte vrea serverul (în concept, în implementare json-ul de configurare a fost stocat pe local în aplicația Android) să afișeze aplicația.

În continuare vom prezenta caracteristicile de implementare ale celor 2 “meniuri” de funcționalități.

Interfata statică.

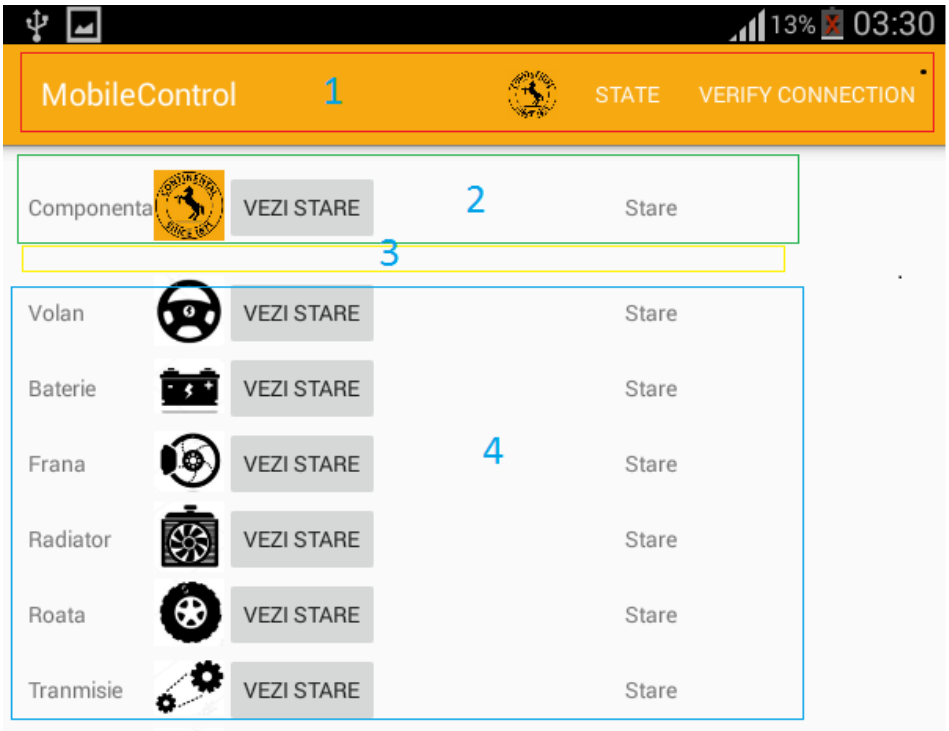
- are în fișierul de configurare a interfeței 137 de linii de cod xml.
    - 1 obiect de tip “ScrollView”
    - 1 obiect de tip “TableLayout”
    - 9 obiecte de tip “TableRow” fiecare avand cate
        - 2 obiecte de tip “TextView”
        - 1 obiect de tip “ImageView”
        - 1 obiect de tip “Button”
- Deci, estimativ 47 de obiecte, hardcodate în interfață.



Interfață statică.

Interfata dinamică.

- Are în fișierul de configurare doar 35 de linii de cod xml.
  - 1 obiect de tip “ScrollView”
  - 1 obiect de tip “TableLayout”
- Deci, 2 obiecte.



Interfață dinamică.

Putem observa că diferenţele nu sunt mari, şi conceptul de interfată generică, generată din cod funcţionează. Ideea care face puternic acest concept este aceea de a avea un dispozititv care se poate conecta la componentele electronice ale maşinii (mufa OBD), caută să vadă ce funcţionalităţi poate accesa, ce informaţii poate cere componentelor, adună toată această informaţie şi o trimite la client (aplicaţie Mobile Control), iar aplicaţia generează interfaţa în funcţie de ce resurse oferă, informaţii poate accesa. După cum am menţionat mai sus, nu am trimis JSON-ul cu interfaţa ce se vrea generată de la server, ci l-am stocat într-un fişier de tip .java între fişierele care gestionează funcţionalitatea aplicaţiei.

Pentru interfaţa generată dinamic, pe langă funcţiile primare din fiecare activitate, definite default pentru fiecare activitate, au fost definite funcţionalităţi care iau informaţii dintr-o structură şi încearcă definirea interfeţei.

Paranteză din prezentarea conceptului MobileControl pentru a prezenta concepte de Android, pentru întelegerea funcţionalităţilor implementate.

Activitati Android.

Pentru a întelege cum funcţionează o activitate Android am copiat o imagine de pe site-ul : “https://developer.android.com/reference/android/app/Activity “ unde este explicat tot ciclul activităţii.

Funcţionalităţile sunt deductibile, iar implementările lor reprezintă comportamentul aplicaţiei (mai ales al activităţii) în timpul şi funcţionalitatea respectivă. Pentru întelegere prezentăm doar funcţia onPause(). În această funcţie cel care dezvoltă aplicaţia poate scrie tot ce vrea ca aplicaţia (în mod special această activitate pentru care este definită funcţia) să facă în timpul în care activitatea prezentă este în starea de pauză. Funcţia vine predefinită de Android, dar cu un simplu "@Override" dezvoltatorul aplicaţiei işi poate pune amprenta lui în comportamentul activităţii, şi poate suprascrie comportamentul normal, sau dezvolta cel existent.

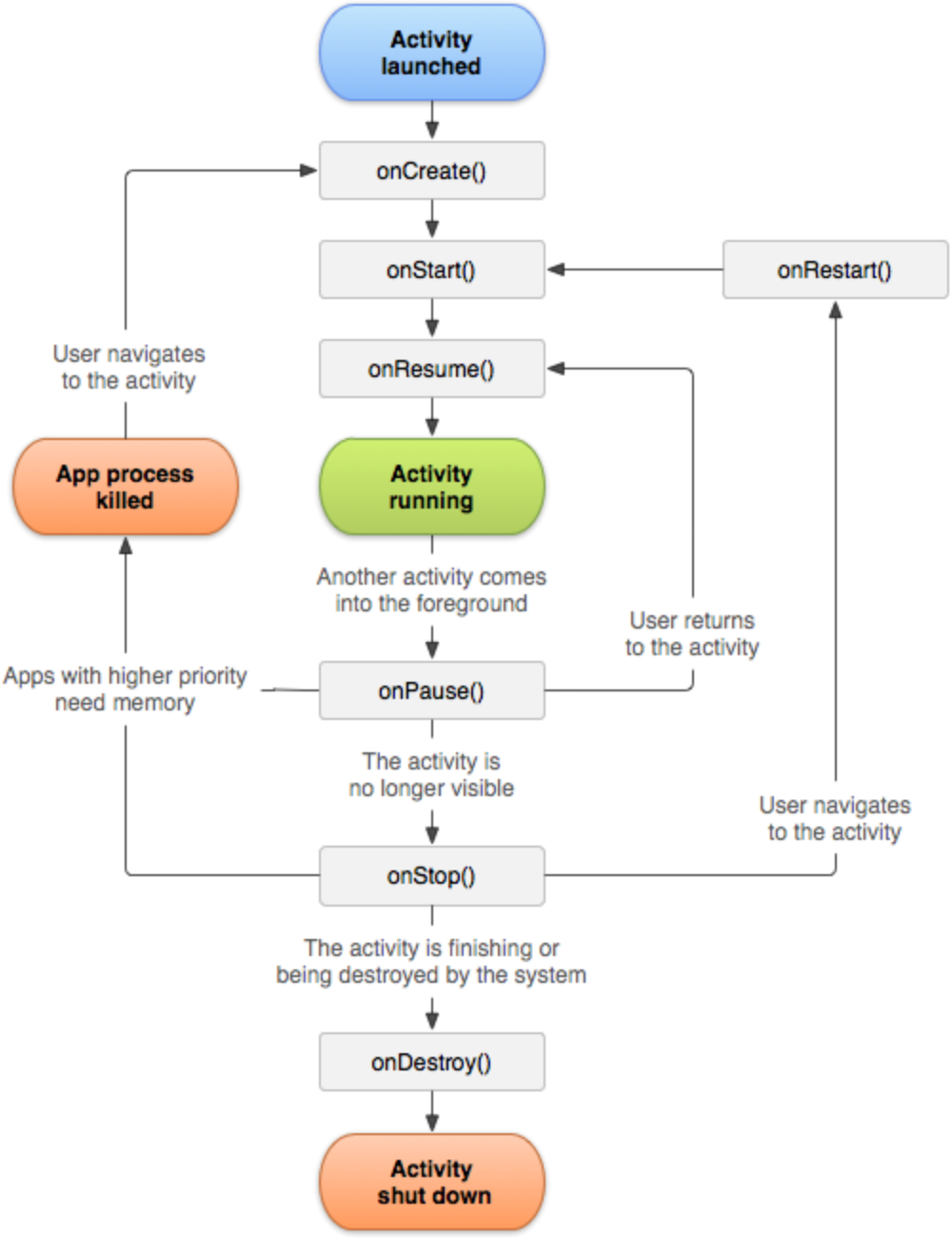


Diagrama activitate Android.

Functionalităţi folosite în generarea interfeţei dinamice:

```
public void addRowCompImageButtonAnsw(String componenta, String imageName, String buttonName){
    /* Find Tablelayout defined in main.xml */
    TableLayout tl = (TableLayout) findViewById(R.id.table_layout_id);
    /* Create a new row to be added. */
    TableRow tr = new TableRow(this);
    tr.setLayoutParams(new TableRow.LayoutParams(TableRow.LayoutParams.FILL_PARENT, TableRow.LayoutParams.WRAP_CONTENT));


    TextView textView = new TextView(this);
    textView.setText(componenta);
    int i = 2;
    tr.addView(textView);

    ImageView imageView = new ImageView(this);
    imageView.setImageResource(map.get(imageName));
    tr.addView(imageView);

    Button b = new Button(this);
    b.setText(buttonName);
    //b.callOnClick()
    //b.setLayoutParams(new TableRow.LayoutParams(TableRow.LayoutParams.FILL_PARENT, TableRow.LayoutParams.WRAP_CONTENT));
    /* Add Button to row. */
    tr.addView(b);
    /* Add row to TableLayout. */

    TextView statusText = new TextView(this);
    statusText.setText("aici apare statusul");
    statusText.setGravity(Gravity.CENTER);
    tr.addView(statusText);

    //tr.setBackgroundResource(R.drawable.sf_gradient_03);
    tl.addView(tr, new TableLayout.LayoutParams(TableLayout.LayoutParams.FILL_PARENT, TableLayout.LayoutParams.WRAP_CONTENT));
}
```

Funcţie adăugare linie în tabelul interfeţei dinamice.

Funcţia : “addRowCompImageButtonAnsw()” adaugă un rând în lista configurată pentru interfaţă.

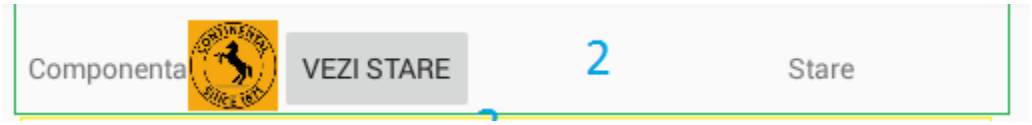
Pentru început se caută contextul pentru care trebuie adaugat un rând, iar pentru aceasta este creat local un obiect de tipul “TableLayout”, “tl”, unde se caută după id, obiectul din interfaţa (.xml), în cazul nostru id-ul = “table_layout_id”.

- Se instanţiază un obiect de tipul “TableRow”, “tr”, pentru care setăm parametri de layout, FILL_PARENT şi WRAP_CONTENT, parametri care spun că acest nou rând va lua dimensiunea obiectului părinte, ca lăţime, iar ca lungime va încorpora conţinutul.
- Urmează crearea unui obiect de tip “TextView”, obiect care va afişa numele componentei integrate în masina.
- Se adaugă acest TextView în obiectul de tip TableRow.
- Se crează un obiect imagine, “ImageView”, care primeşte o imagine şi este adaugat la rândul lui în TableRow-ul “tr”.
- Se adaugă în acelaşi mod un buton. Ce e de menţionat e că nu au funcţionalitate ataşată, este comentat codul care poate ataşa o funcţionalitate fiecărui buton,

"b.callOnClick()". Am lăsat comentat codul pentru a observa posibilitatea adăugării de funcţionalitate interfeţei generate.
- Se mai adaugă un "TextView"
- După care se adaugă acest "tr" în "tl"-ul părinte, cu parametri ce setează layout-ul descris iniţial.

Mai sunt definite 2 funcţii:

- addStaticFirstRow() – funcţie care adaugă un rând mai special, este rândul care conţine interfaţa:



- addEmptyRow()- funcţie care adaugă un rând gol, având aceleaşi specificaţii ca celelalte, dar cum conţinutul e gol, dimensiunea lui relativă la continut "WRAP_CONTENT", e mai mică

Nu vom mai descrie codul pentru aceste funcţionalităţi pentru că este foarte asemănător funcţiei "addRowCompImageButtonAnsw()".

Json

Pentru a întelege complet aceste funcţii generice, vom analiza textul care adaugă concret informaţia stocată în JSON în interfaţa dinamică:

```
//add a static first row and than will add a empty space
addStaticFirstRow();
addEmptyRow();
jsonForMenu = JSON.jsonForDynamicMenu();
for(int n = 0; n < jsonForMenu.length(); n++){
    try {
        jsonForRow =  jsonForMenu.getJSONObject(n);
        jsonForComponets = jsonForRow.getJSONObject("components");
        addRowCompImageButtonAnsw(jsonForComponets.getString("componenta"),
                jsonForComponets.getString("imageName"),
                jsonForComponets.getString("buttonName"));
    }
    catch (JSONException e) {
        Log.e("MYAPP", "unexpected JSON exception", e);
    }
}
```

(Aici este parsată structura care conţine informaţia necesară generării interfeţei.)

- Pentru început se adaugă primul rând care este mai special, şi definit în funcţie separată.
- Se adaugă un rând gol (estetic – separă rândul care arată modelul fiecărui rând urmator de rândurile efective)
- Pentru toată lungimea structurii se parsează JSON-ul şi se apelează funcţia : “addRowCompImageButtonAnsw()” cu valorile preluate din JSON.

În continuare vom trata această parte de JSON.

JSON tehnologie de stocat informaţie ierarhizat, dezvoltat şi folosit mai mult în dezvoltarea aplicaţiilor web. Este deobicei alegerea celor mai multi dezvoltatori în ceea ce se vrea comunicaţie între server şi client, câştigând mult terent în favoarea .xml-ului, tehnologie folosită iniţial.

În aplicaţia Mobile Ocntrol tehnologia JSON a fost folosită pentru a stoca două structuri interne, una folosită pentru a stoca informaţii legate de mesajul de CAN care urmează să fie trimis la server, pentru a fi trimis pe bus-ul de CAN, şi una folosită pentru a stoca informaţia necesară generării interfeţei generice. Pentru că până acum am explicat conceptul de interfaţă dinamică, vom explica pentru început JSON -ul în care stocăm informaţiile necesare interfeţei.

JSON pentru interfaţa dinamică.

Pentru a simplifica întelegerea conceptelor din spatele structurii de tip JSON, vom gândi toată structura stocată în obiectul de tip JSON ca o structură bine definită cu sub-structuri care stochează informaţia, usor de creat, usor de folosit, logic şi concis, care în final dupa ce a fost creat logic şi concret, este trimis ca un simplu String, şir de caractere, care de capătul celalalt al comunicării poate fi foarte usor transpus iară într-un obiect de tip JSON şi folosit ca structură logică de date.

```
public static JSONArray jsonForDynamicMenu(){
    try {
        JSONArray jsonObject = new JSONArray();

        JSONObject jsonRow1 = new JSONObject();
        JSONObject jsonComponents1 = new JSONObject();
        jsonRow1.put("rowNumber",1);
        jsonRow1.put("id",1);
        jsonComponents1.put("componenta", "Volan");
        jsonComponents1.put("imageName","volan");
        jsonComponents1.put("buttonName","Verifica");
        jsonRow1.put("components", jsonComponents1);
        jsonObject.put(jsonRow1);
```

După cum putem vedea în imaginea de mai sus a fost creat un obiect principal, părinte, “jsonObiect”, în care vom stoca toată informaţia. Este creat de tipul “JSONArray”, clasă concepută pentru a stoca obiecte asemanatoare de tip JSON. Au mai fost create două obiecte de tip “JSONObiect”, “jsonRow1” şi “jsonComponents1”.

Ierarhia structurii este simplă:

JSONArray - jsonObject:

JSONObject – jsonRow1:

text: “rowNumber”

text: “id”

JSONObject – jsonComponents1:

text: “componenta”

text: “imageName”

text: “buttonName”

JSONObject – jsonRow2:

text: “rowNumber”

text: “id”

JSONObject – jsonComponents2:

text: “componenta”

text: “imageName”

text: “buttonName”

…..

Structura conţine şase substructuri care descriu fiecare câte un rând din interfaţă ce urmează să fie generată. Interfaţa generată după cum am explicat anterior.

Despre Json-ul în care stocăm detaliile legate de CAN message vom explica cand vom aborda capitolul legat de mesajele de CAN.

Websocket.

Pentru conexiunea dintre tabletă şi Raspberry Py am folosit o tehnologie larg utilizată : websocket.

Codul de bază l-am luat de pe pagina https://www.varvet.com/blog/using-websockets-in-native-ios-and-android-apps/ unde prezintă secvenţele de cod minimale pentru partea de server şi pentru partea de client. În aplicaţia Mobile Control am folosit evindet partea de client, iar pe serverul montat pe Raspberry am folosit secvenţa pentru server.

Nu mai explic codul pentru ca sunt pur funcţii de conectare server – client, am folosit portul 8080 şi un IP static, şi clasa “WebSocketClient” din java, clasă care are funcţiile :

- void onOpen()
- void onMessage()
- void onClose()
- void onError()

Pentru exemplificare codul pentru onMessage() şi onClose() folosit este :

```
@Override
public void onMessage(String s) {
    final String message = s;

}

@Override
public void onClose(int i, String s, boolean b) {
    mWebSocketClient.close();
    Log.i("Websocket", "Closed " + s);
}
```

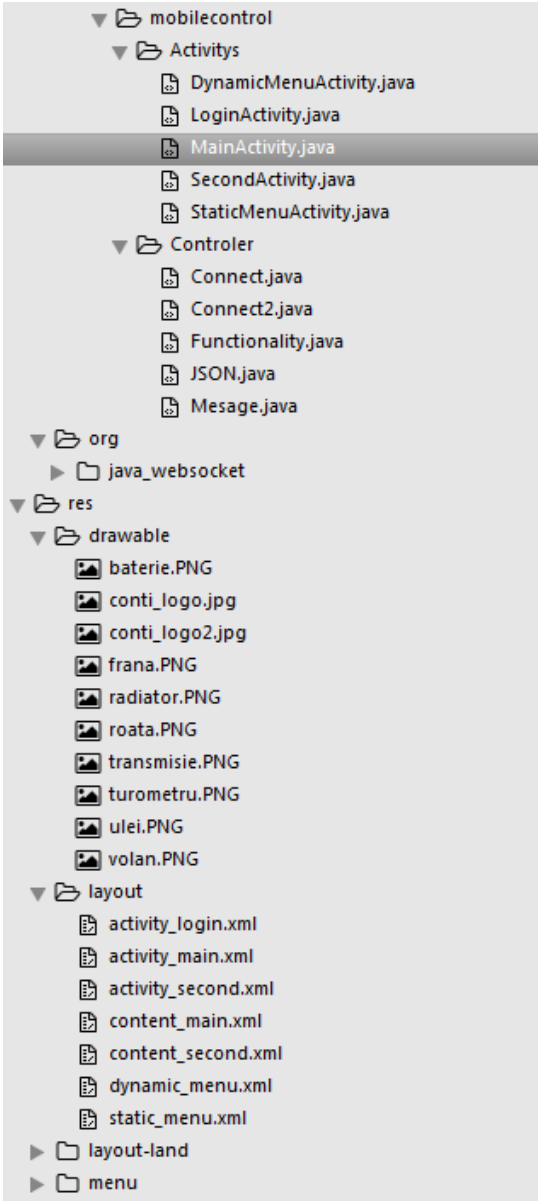
Iar o secvenţa de cod care apelează conectarea la server este prezentată în urmeatoare imagine. Unde se verifică dacă nu exista deja o conexiune la server, şi dacă nu este, se încearcă o conexiune. Iar în cazul prezentat se finalizează cu trimitere de mesaj ciclic, la care o să revenim cu prezentare când ajungem la capitolul legat de mesajele de CAN.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.static_menu);
    //try to connect
    if (conn.IsConnected() == false){
        conn.connectWebSocket();
    }

    connect_text_id = (TextView) findViewById(R.id.connect_text_id);
    if (conn.getErr().equals("Conected")){
        //connect_text_id.setText("Connected");
        conn.Conecteaza();
        Log.i("cenect","conectat");
    }
    else {
        connectionError();
        Log.i("cenect","neconect");
    }
    TrimiteMesajCiclic();
}
```

onCreate() din StaticMenuActivity.java

Per general o parte din ierarhia directoarelor şi fisierelor din cadrul workspece-ului dedicat aplicaţiei mobile arată în felul următor.



Mesaje CAN

Creare mesaj CAN

Una dintre componentele majore ale conceptului Mobile Control este trimiterea mesajelor de tip CAN de pe aplicaţia mobile pe Bus-ul de CAN. Această componentă a fost implementată pe 2 platforme, pe Android şi pe Raspbian. La nivel de aplicaţie a fost creată o structură de tipul mesajului CAN (Pentru informare amintim că mesajul de CAN are 8 bytes dedicaţi informaţiei transmise, un Id – identificator unic şi un DLC – dimensiunea datelor ce urmează a fi transmise). După care a fost implementată o funcţie care transformă structura locală într-un JSON care are să fie folosit ca structură pentru comunicarea cu serverul. Urmat fiind momentul în care serverul care ascultă, primeşte comanda "send", atunci se despachetează informaţia stocată în JSON-ul trimis de client (aplicatia Android) şi se adaugă într-un obiect de tip "CanInterface::Message" şi trimis pe Bus-ul de CAN ca şi mesaj, cu Id-ul şi DLC-urile setate.

Pentru început putem privi structura în care stocăm mesajele de CAN:

```
Mesage mes1 = new Mesage();
mes1.setDLC(8);
mes1.setId(0x193);
mes1.setByte0((byte)0);
mes1.setByte1((byte)0);
mes1.setByte2((byte)5);//blinker aprins
mes1.setByte3((byte)0);
mes1.setByte4((byte)0);
mes1.setByte5((byte)0);
mes1.setByte6((byte)0);
mes1.setByte7((byte)0);
```

Putem observa un obiect de tip Message (clasă definită în cadrul aplciaţiei, dar asupra detaliilor de implementare a clasei nu vom intra deoarece sunt simple concepte de POO, definire containere, definire funcţii de "set" şi "get" pentru informatia stocată în fiecare container.

Prezentăm detalii specifice acestui mesaj înainte sa continuăm cu explicarea implementării :

- DLC – dimensiunea mesajului care urmează să fie transmis e prezentată ca fiind 8(bytes)
- Id – e setat ca fiind 193 (în hexazecimal).
- Urmatorii bytes sunt 0 înafara de Byte2 care este setat pe 5(dupa cum se vede în comentariu, deoarece aveam acces la informaţii interne despre ECU-ul pe care testam ştiam că în implementarea codului stocat pe ECU, care controla un far. Si am stiut că pentru mesajul cu Id-ul 193 dacă setăm Byte-ul 2 cu valoare 5 aplicaţia montată pe ECU trebuie să permită sistemului de lumini aprinderea blinker-ului "semnalizare")

- Ca şi informaţie, deobicei o parte din aceşti bytes transmişi pe Bus-ul de CAN, în mesajele normale au şi funcţie de CRC – checksum, folosita pentru verificarea integrităţii datelor.

În continuare putem observa mutarea într-un JSON a informaţiei stocate în obiectul “mes1”, obiect de tipul clasei “Message”.

Mesaj CAN în JSON.

```
public static String toJSon(Mesage mesage){
    try {
        JSONObject json = new JSONObject();
        json.put("target","main_handler");

            JSONObject jsonMes = new JSONObject();
            jsonMes.put("cmd","send");

                JSONObject jsonData = new JSONObject();
                jsonData.put("ID", mesage.getId());
                jsonData.put("DLC", mesage.getDLC());
                jsonData.put("Byte_0", mesage.getByte0());
                jsonData.put("Byte_1", mesage.getByte1());
                jsonData.put("Byte_2", mesage.getByte2());
                jsonData.put("Byte_3", mesage.getByte3());
                jsonData.put("Byte_4", mesage.getByte4());
                jsonData.put("Byte_5", mesage.getByte5());
                jsonData.put("Byte_6", mesage.getByte6());
                jsonData.put("Byte_7", mesage.getByte7());

            jsonMes.put("data", jsonData);

        json.put("message",jsonMes);

        Log.i("Mesage1",json.toString());
        return json.toString();
    }
    catch (JSONException ex){
        ex.printStackTrace();
    }
    return null;
}
```

In această funcţie “toJson” se încearcă crearea unui string de tip Json în care să fie stocată informaţia stocată momentan în obiectul “mesage” obiect din clasa “Message”:

- Pentru început se crează un obiect de tip Json denumit generic “json”, se setează “targhet” cu valoarea “main_handler” (informaţie care ajută serverul sa direcţioneze mesajul către “main_handler”)
- Urmatoarea setare este serarea comenzii (“cmd”) ca fiind “send” (informaţie care ajută “main_handler”-ul să ştie pentru ce operaţie este informaţia transmisă de client), într-un nou obiect Json, “jsonMes”.

- Urmează crearea unui nou obiect de tip Json, “jsonData”, şi setarea valorilor pentru “Id”, “DLC” şi urmatorii 8 Bytes, valori preluate direct din obiectul mesage.
- Acum în obiectul “jsonMes” la tagul “data” se setează obiectul de tip Json, “jsonData”.
- Urmatorul pas este adăugarea în obiectul Json părinte, la tag-ul “message”, obiectul de tip Json, “jsonMes”.
- La final se returnează un string în care este cuprins toată informaţia descrisă mai sus.

Transmitere mesaj CAN.

Într-o activitate, în cazul nostru, “StaticMenuActivity”, în cadrul funcţiei “onCreate” am testat transmiterea informaţiei (mesajului de CAN) spre server, folosind tehnologii de tip websocket şi Json, după conectarea cu serverul.

```
connect_text_id = (TextView) findViewById(R.id.connect_text_id);
if (conn.getErr().equals("Conected")){
    //connect_text_id.setText("Connected");
    conn.Conecteaza();
    Log.i("cenect","conectat");
}
else {
    connectionError();
    Log.i("cenect","neconect");
}
TrimiteMesajCiclic();
```

Pentru a analiza funcţia “TrimiteMesajCiclic()” ne vom folosii de urmatoarea imagine:

```
public  void TrimiteMesajCiclic(){
    Functionality.setMesage();
    timer.scheduleAtFixedRate(functie, 100, 100);
}
```

- Unde în clasa java numită Functionality stocăm funcţionalitatea “setMesage” explicată mai sus.
- Timer – unde se setează ca la cu delay (al doilea parametru din antet) de 100 de milisecunde, şi ciclic la 100 de milisecunde (ciclicitate dată de al doilea parametru) să se execute task-ul – “funcţie” definit mai jos de clasa “TimerTask”:

```
public  TimerTask functie = new MyTimerTask();
class MyTimerTask extends TimerTask {
    public void run() {
        StaticMenuActivity.this.runOnUiThread(new Runnable() {
            public void run() {
                conn.sendMessage(Functionality.mesage1);
                conn.sendMessage(Functionality.mesage2);
            }
        });
    }
}
```

Ca un rezumat: codul de mai sus, trimite ciclic 2 mesaje (de tip CAN stocate în string-uri care conţin obiecte JSON) spre server.

Receptare mesaj CAN (JSON).

Recepţionarea la nivelul serverului se face în funcţia “main_handler” după cum am specificat la crearea JSON-ului. Unde se va face desfacerea JSON-ului în informaţia ce ne interesează să o transmitem pe Bus-ul de CAN. La nivelul serverului este prezentă o interfaţă numită “can_interface.c” care pe langă multele funcţii implementate implementează şi funcţia SendMessageHandler”, funcţie la care o să revenim în viitor. Pentru început să explicăm cum se desface informaţia primită de la client (aplicaţia MobileControl).

După cum am explicat până acum în contextul în care am povestit despre JSON, la server a ajuns un string, în care am stocat toată informaţia trimisă de client. La nivelul serverului desfacem exact după structura după care am alcătuit string-ul şi obţinem informaţia trimisă, ce urmează să fie trimisă iară, dar de data asta direct pe Bus-ul de CAN.

```
command = _message.get<std::string>("cmd");
data = _message.get_child("data");

DLOG(INFO) << "Command: " << command;

if(command == "send") {
    CanInterface::Message faceCanMessage;
    faceCanMessage.id = data.get<uint32_t>("ID");
    faceCanMessage.dlc = data.get<uint32_t>("DLC");
    faceCanMessage.data[0] = data.get<uint8_t>("Byte_0");
    faceCanMessage.data[1] = data.get<uint8_t>("Byte_1");
    faceCanMessage.data[2] = data.get<uint8_t>("Byte_2");
    faceCanMessage.data[3] = data.get<uint8_t>("Byte_3");
    faceCanMessage.data[4] = data.get<uint8_t>("Byte_4");
    faceCanMessage.data[5] = data.get<uint8_t>("Byte_5");
    faceCanMessage.data[6] = data.get<uint8_t>("Byte_6");
    faceCanMessage.data[7] = data.get<uint8_t>("Byte_7");
    interfaces->canInterface->SendMessageHandler(faceCanMessage);
```

(Implementarea preluării informaţiei de la client şi trimiterii pe BUS)

- În "_message" este tot JSON-ul contruit de noi la nivel de client stocat la nivel de client în "jsonMes" (pentru detalii putem să ne întoarcem la capitolul unde am descris acest lucru)
- În string-ul "command" este stocată informaţia setată de noi pe client în tag-ul "cmd" (care era "send").
- În "data" stocăm informaţia stocată în JSON la tag-ul "data" unde la nivel de client se numea "jsonData".
- If(command == "send") ne întreabă dacă comanda e "send", ceea ce lasă posibilitatea exitenţei multor altor comenzi.
- Si dacă comanda este "send"(cum am setat-o la nivel de client) atunci se crează un obiect ce moşteneşte interfaţa CanInterface, în care adăugăm toată informaţia stocată în "data" şi folosind funcţia "SendMessageHandler" se trimite direct pe BUS.

Implementarea funcţiei "SendMessageHandler":

```
void CanInterface::SendMessageHandler(const Message &message) {

    tx.can_id = (message.msgType << 31) |
                message.id;

    tx.can_dlc = message.dlc;

    for(int i = 0; i < message.dlc; i++) {
        tx.data[i] = message.data[i];
    }

    try {
        write(s, &tx, sizeof(struct can_frame));
        //boost::asio::write(descriptor, boost::asio::buffer(&tx, sizeof(tx)));

    }
    catch(boost::system::system_error &e) {
        LOG(ERROR) << "Error at send CAN message." << e.what();
        throw;
    }
}
```

("SendMessageHandler" implementare)

CanInterface este interfaţa care gestionează funcţionalităţile implemntate pentru modulul de CAN montat pe Raspberry Py, despre care am discutat în capitolul legat de componentele fizice ale conceptului.

În frame-ul "tx" se setează pe rând , id-ul, dlc-ul şi informaţia ce trebuie stocată în cei 8 bytes, după care se încearcă un "write" operaţie ce încearca scrierea directă pe Bus-ul de CAN a mesajul tx("tx" petnru că e transmis, daca era recepţionat era "rx")

Concluzii:

Observaţie :

Pentru partea de server nu am explicat toate detaliile deoarece codul este preluat din surse de tip opensurce. Dar ca rezumat este un server cu capacităţi de conectare cu clietul în tehnologie websocket, ce are în implemmentare şi CanInterface, interfaţa folosită după cum am spus pentru gestionarea funcţionalităţilor oferite de modulul de CAN de pe Raspbery Py, iar ce era esenţial pentru întelegerea conceptului şi demonstrarea funcţionalităţii a fost descris mai sus.

Deci, înafara de anumite funcţionalităţi şi modificări asupra codului existent, codul de pe partea de server nu este scris de mine, ci provine din usrse de tip OpenSurce sau documentaţii publice.

Demonstrarea conceptului Mobile Control a fost facută în totalitate la nivel de client. În totalitate ca și demonstrație a conceptelor, nu și în implementare completă a tuturor posibilităților și capabilităților conceptului.

Ce tine de interfața generată dinamic, implementarea a fost completă si funcțională. (Demonstrație usoara pe tableta folosită).

Dar pentru funcționalitatea transmiterii de mesaje CAN de pe tabletă pe ECU, sincronizarea esuată a bibliotecilor de C a blocat demonstrarea ei. Deși implementarea e completă, și în mare parte explicată in documentația aici de față, deoarece s-a dezvoltat pe 3 platforme (3 OS-uri diferite, Android – client, Raspian – server și Windows – cross compile (server)), undeva nu s-a reușit o sincronizare perfectă, ceea ce a dus la eșuarea demonstrarii funcționalității. Iară, acest eșec nu arată lipsa implementării, ci doar esecul configurării.

Ca și concluzie finală, lucrarea de licență, Mobile Control, concept diagnoza wi-fi, control de la distanță a componentelor integrate în automobil, a avut și parte de cercetare, parte din care am învățat multe despre industria si tehnologiile din ramura informaticii dedicate automotiv-ului.

Cuprins