УНИВЕРЗИТЕТ У НОВОМ САДУ ●**ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

УНИВЕРЗИТЕТ У НОВОМ САДУ

ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Милан Пандуров

Програмска подршка за апстракцију и управљање системским ресурсима наменских система заснованих на Linux-у

МАСТЕР РАД

Нови Сад, 2016

УНИВЕРЗИТЕТ У НОВОМ САДУ ●**ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска документација
Тип записа, **ТЗ**:		Текстуални штампани материјал
Врста рада, **ВР**:		Дипломски – мастер рад
Аутор, **АУ**:		**Милан Пандуров**
Ментор, **МН**:		**Доц. др Иштван Пап**
Наслов рада, **НР**:		**Програмска подршка за апстракцију и управљање системским ресурсима наменских система заснованих на Linux-у**
Језик публикације, **ЈП**:		Српски / латиница
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		**2016**
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		**7/48 7/19/0/0**
Научна област, **НО**:		Електротехника и рачунарство
Научна дисциплина, **НД**:		Рачунарска техника
Предметна одредница/Кqучне речи, **ПО**:		**Управљање системским ресурсима, кућна аутоматизација, апстракција**
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		**Рад описује организацију програмске пордшке којом се олакшава преносивост на различите уграђене уређаје.**
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	**Доц. др Иван Каштелан**
	Члан:	**Др Милан Лукић** — Потпис ментора
	Члан, ментор:	**Доц. др Иштван Пап**

UNIVERSITY OF NOVI SAD●**FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic publication	
Type of record, **TR**:		Textual printed material	
Contents code, **CC**:		Master Thesis	
Author, **AU**:		**Milan Pandurov**	
Mentor, **MN**:		**Istvan Papp PhD**	
Title, **TI**:		**Software for abstraction and management of system resources on Linux embedded devices**	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		**2016**	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		**7/48 7/19/0/0**	
Scientific field, **SF**:		Electrical Engineering	
Scientific discipline, **SD**:		Computer Engineering, Engineering of Computer Based Systems	
Subject/Key words, **S/KW**:		**System resources management, home automation, abstraction**	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		**Paper describes software architecture that allows easier porting to new new devices.**	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:			
Defended Board, **DB**:	President:	**Ivan Kastelan PhD**	
	Member:	**Milan Lukic PhD**	Menthor's sign
	Member, Mentor:	**Istvan Papp PhD**	

Zahvalnost

Zahvaljujem se Ištvanu Papu i Romanu Pavloviću na stručnoj pomoći i savetima koji su pomogli u izradi ovog rada.

Posebno se zahvaljujem porodici na pruženoj podršci tokom celokupnog školovanja.

SADRŽAJ

SPISAK SLIKA

SPISAK TABELA

SKRAĆENICE

FPGA - *Field Programming Gate Array*, Programabilne sekvencijalne mreže
CPU - *Central Processor Unit*, Centralni procesoor
GND - Oznaka za signal na nultom potencijalu
GW - *Gateway*, Centralni kontroler
SOC - *System on chip*, Sistem na čipu
IoT - *Internet of things*, internet stvari
GW - *Gateway*, centralni kontroler
PC - *Personal computer*, Personalni računar
LE - *Light-emitting*, Emituje boju
LAN - *Local area network*, Lokalna mreža
HAL - *Hardware abstraction layer*, sloj za apstrakciju hardvera
SM - *System manager*, softver za rukovanje sistemskim resursima
JSON - *JavaScript Object Notation*, JavaScript objektna notacija
NFC - *Near field communication*, komunikacija kratkog polja
CLI - *Command-line interface*, interfejs komandne linije

1. Uvod

Primetno je da se na tržištu potrošačke elektronike svakodnevno pojavljuje velik broj novih uređaja, niska cena i široka dostupnost hardverskih komponenti postala je "katalizator" u kreiranje raznovrsnih i inovativnih softverskih rešenja koja se oslanjaju na funkcionalnosti tog hardvera. Jedan od dobrih primera koji oslikava ovo stanje jeste ekspanzija IoT (engl. *internet of things - IoT*) ekosistema, pojavljivanje najraznovrsnijih vrsta senzora, aktuatora i jeftinih opštenamenskih računara koji imaju sposobnost da međusobno razmenjuju podatke preko nekog fizičkog medijuma, predstavlja upravo taj skup hardverskih komponentina kojima se dalje grade najrazličitija softverska rešenja. Softver u ovom smislu daje semantiku hardveru koji leži ispod, kombinuje dostupne podatke sa senzora u informacije i prezentuje ih korisniku kroz konkretno dostupne medijume (ekran, zvuk, svetlosna indikacija, akcije aktuatora itd). Skup senzora i aktuatora međusobno komuniciraju sa računarom opšte namene gradeći tako izvesnu vrstu računarske mreže u kojoj se senzori i aktuatori mogu nazivati čvorovima (engl. *nodes*) dok opštenamenski računar nosi naziv centralni kontroler (engl. *Gateway - GW*). Očigledno je da svaki od čvorova mora posedovati određeni softver koji kontroliše pretvaranje analognih vrednosti u digitalne informacije (i obrnuto) i rukovanje komunikacijom sa ostalim čvorovima u mreži. Međutim suštinska obrada podataka se dešava tek na centralnom kontroleru, a organizacija njegovog softvera i sprega sa različitim vrstama hardvera jeste tema ovog rada. Jedna od standardnih funkcionalnosti koje omogućava centralni kontroler je povezivanje sa udaljenim serverima (engl. *cloud*), mobilnim i PC aplikacijama korisnika.

1.1 Struktura centralnog kontrolera

Centralni kontroler, kao što je već pomenuto, jeste računar opšte namene, međutim njegove performanse i resursi kojima raspolaže su višestruko manji od konvencijalnih PC-računara i stoga se on više uklapa u kategoriju ugrađenih uređaja (eng. *embedded*). Hardver

centralnog kontrolera se najčešće zasniva na nekom SOC-u (engl. *system on chip*) koji se različitim fizičkim interfejsima (SPI, UART, I2C itd) povezuje sa opcionim periferijama na ploči. Periferije koje se najčešće mogu naći na centralnom kontroleru jesu moduli za komunikaciju sa čvorovima (Wi-Fi, Ethernet, ZigBee, Z-Wave itd.), međutim pored toga centralni kontroler skoro uvek poseduje i neke periferije koje omogućavaju neposrednu interakciju sa korisnikom. Obzirom da je većina interakcijasakorisnikom najčešće delegirana na mobilne i PC aplikacije, centralni kontroler je opremljen određenim brojem tastera i LE dioda za tu namenu.

Obzirom da je platforma na kojoj je zasnovan centralni kontroler uglavnom namenska, proizvođač platforme obezbeđuje skup sistemskog softvera (engl. *board support package - BSP*) koji zavisi od konkretne platforme. Najčešći operativni sistem koji se koristi u ovim okruženjima je neka distribucija Linux operativnog sistema za ugrađene uređaje. Raznovrsnost BSP-a se oslikava u različitim verzijama i tipovima prevodioca koje podržava, različitim skupom biblioteka i različitim sistemskim aplikacijama koje obezbeđuje. Dodavanje funkcionalnosti centralnom kontroleru se zasniva na razvoju korisničkih (engl. *user space*) aplikacija, a taj razvoj direktno zavisi od mogućnosti koje pruža BSP.

1.2 Problematika i zadatak

Kao što je pomenuto, različite kombinacije platforme i periferija centralnog kontrolera čine izuzetno heterogenu sredinu iz perspektive razvoja softvera. Ovo konkretno znači da promene na hardveru imaju direktan uticaj na funkcionisanje i rad softvera. Ovo ne bi predstavljalo problem kad bi softver bio namenjen za isključivo jednu platformu sa fiksnim periferijama, međutim tržište kao izuzetno dinamična sredina, zahteva česte promene na hardveru i softveru centralnog kontrolera. Kako bi se ispratila ova dinamika, potrebno je napraviti takvu arhitekturu softvera koja omogućava brz razvoj i prilagođenje za različite platforme. Izvorni kod softvera potrebno je struktuirati na način koji jasno pravi razliku između delova koji su zavisni od konkretne platforme i onih koji su zajednički za sve.

1.3 Zadaci centralnog kontrolera

IoT koncept je pronašao primenuu mnogim oblastima, među kojima i u oblasti automatizacije domova i kreiranju tzv.pametnih kuća. Osnovna stvar koja se očekuje od centralnog kontrolera u ovom scenariju jeste da u svakom trenutku omogući korisnicima povezivanje, kontrolu čvorova kao I dobavljanje informacija sa njih. Udaljeni pristup čvorovima i nezavisno izvršavanje automatizovanih akcija (scene [1] ili aplikacije [2]) iako jeste opciona stvar, u većini slučajeva se podrazumeva da je prisutna. Pored ovoga centralni kontroler izvršava

mnoštvo drugih aktivnosti, koje neretko nisu direktno prezentovane korisniku, izvršavaju se u pozadini, a neophodne su za nesmetano funkcionisanje uređaja. Neke od tih funkcionalnosti su:

- Pokretanje aplikacija i potrebnih sistemskih servisa prilikom podizanja sistema.
- Rukovanje podešavanjima lokalne mreže (LAN).
- Dobavljanje sistemskih informacija i informacija o hardveru.
- Rukovanje periferijama, pre svega LE diodama i tasterima pomoću kojih se korisnik može obavestiti o promenama stanja uređaja.

2. Teorijske osnove – analiza problema

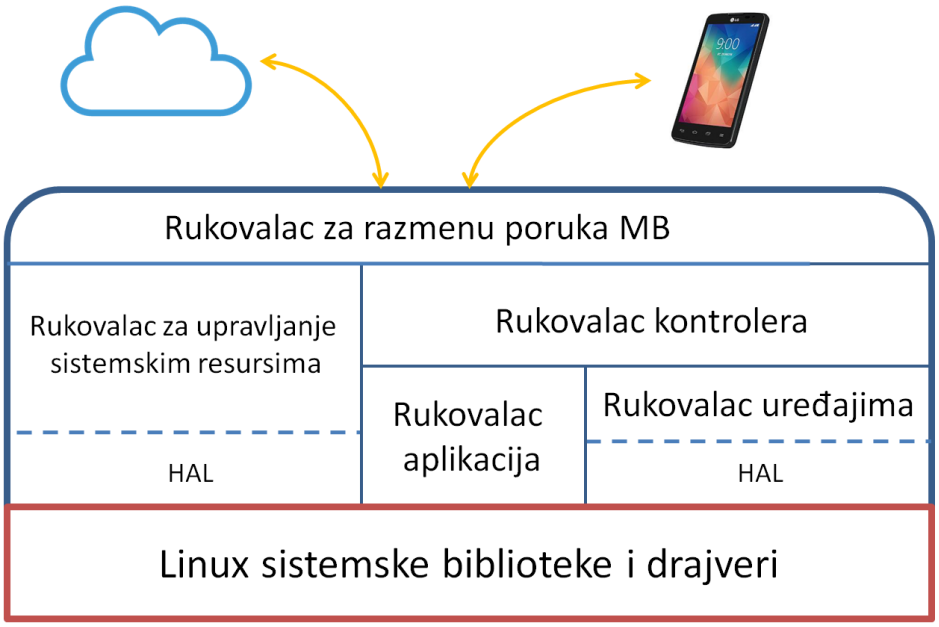
2.1 Globalni pregled softvera centralnog kontrolera

Funkcionalno gledano, softver centralnog kontrolera se može podeliti na nekolikokomponenti po zadacima koje obavljaju. Komponente su podeljene tako da svaka od njih upravlja jednom specifičnom grupom entiteta.

- **Rukovalac uređaja (Device manager - DM)**: ova komponenta je osnovni deo softvera koji je zadužen za komunikaciju sa čvorovima u mrežu i interpretaciju njihovih podataka. On razmenjuje podatke sauređajima u mreži implementirajući konkretne protokole za komunikaciju (ZigBee, Z-Wave itd.) i preslikava dobijene podatke na apstraktne reprezentacije tih uređaja predstavljenih kao skup servisa [3].
- **Rukovalac kontrolera (Controller manager - CM)**:komponenta sistema koja je zadužena za upravljanje uređajima. Ona definiše elementarne oblike kontrole uređaja, programsku spregu (API) za tu kontrolu i izvršavanje automatizovanih akcija kontrole nad uređajima (scene).
- **Rukovalac aplikacija (Application manager - AM)**: komponenta zadužena za izvršavanje modula za proširenje (eng. plug in modules) i upravljanje uređajima.
- **Rukovalacza razmenu poruka (Message broker - MB)**:zadužen da obezbedi uniformnu komunikaciju između GW, udaljenog servera i klijenata. Drugim rečima, MB definiše standardni protokol za komunikaciju koja se koristi u razmeni poruka sa serverom i mobilnim aplikacijama.
- **Rukovalac za upravljanje sistemskim resursima (system manager - SM)**: komponenta zadužena za upravljanje sistemskim i hardverskim resursima spcifičneplatforme centralnog kontrolera.

Da bi se ispunile zahtevane funkcionalnosti GW neke ili sve od navedenih komponenti moraju biti prisutne, zavisno od zahteva konkretnog tržišta.

Jedan način za dizajn datog softvera jeste kreiranje monolitne aplikacije koja sadrži sve komponente kao svoje module i od kojih, zavisno do potreba, se pojedine mogu isključiti u fazi prevođenja. Ovakva aplikacija mora sadržati i izvesni sloj za apstrakciju hardvera - HAL (engl. *Hardware abstraction layer*) u kom bi se izdvojile sve specifičnosti konkretnog hardvera. Na slici 1 blokovski su prikazane komponente ovakve aplikacije.



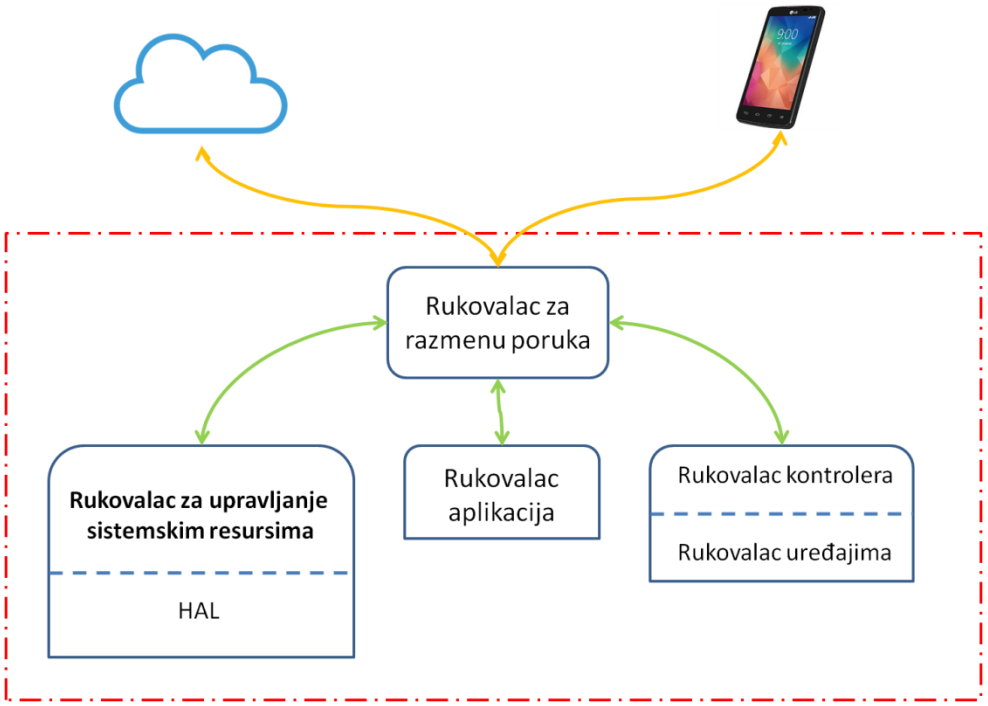
Slika 1 Izgled monolitne aplikacije centralnog kontrolera

Prednosti ovakvog rešenja se ogledaju u brzini obrade podataka, svi moduli dele isti adresni prostor pa su samim tim razmene poruka među njima izuzetno brze. Međutim ovakvo rešenje karakteriše potencijalni problem sa stabilnošću. Manifestacija greške u nekom od modula može dovesti do otkaza celokupne aplikacije, pa samim tim u situacijama promena platforme i hardvera nastaje problem sa robusnošću softvera.

Drugo rešenje bi bilo da se komponente razdvoje u posebne aplikacije od kojih bi svaka bila zadužena za izvršavanje određenog, specijalizovanog, dela funkcionalnosti. Ovim bi se postigla bolja modularnost i fleksibilnost obzirom da se željene komponente ne moraju isključivati u vreme prevođenja, već se jednostavno nepotrebne aplikacije neće instalirati na željenojplatformi. Kako bi ovo funkcionisalo potrebno je obezbediti mehanizam kojim će ove zasebne aplikacije moći međusobno da razmenjuju poruke. Obzirom da se više ne radi o jedinstvenom adresnom prostoru već međuprocesnoj komunikaciji, razmena poruka je sporija od prethodnog rešenja, ali analizom očekivane učestalosti razmene poruka između modula može se dizajnirati sistem koji u realnim situacijama ne pokazuje vidno lošije rezultate. Međutim pozitivna stvar ovog rešenja ogleda se u tome što greška u jednoj aplikaciji ne može ugroziti

stabilnost celokupnog sistema, već samo jednog njegovog dela. Takođe razvoj aplikacija je dosta olakšan jer svaka aplikacija može biti struktuirana na način koji je najprihvatljiviji za rešavanje problema za koji je odgovorna. Kako bi se omogućila ova komunikacija između aplikacija, neophodno je postojanje jedinstvenog komunikacionog protokola preko kog bi aplikacije međusobno razmenjivale poruke.

Kako je predviđena količina podataka za obradu u ovim sistemima relativno mala, a robusnost celokupnog sistema od velikog značaja softver je organizovan kao skup pojedinačnih aplikacija, slično kao što je malopre navedeno, uz sledeće izmene.



Slika 2 Organizacija aplikacija GW

Zbog velike količine poruka koje međusobno razmenjuju rukovalac uređaja i rukovalac kontrolera, oni su se našli kao celina u jedinstvenoj aplikaciji. Rukovalac za razmenu poruka je proširen funkcionalnošću prenošenja poruka između lokalnih aplikacija, dok su sve lokalne aplikacije proširene modulom za razmenu poruka.Da bi se obezbedila što lakša prenosivost softvera svi platformski zavisni delovi funkcionalnosti izmešteni su u rukovalac za upravljanje sistemskim resursima (SM).

3. Koncept rešenja

Kao što je već navedeno u prethodnom poglavlju, rukovalac za upravljanje sistemskim resursima je zasebna aplikacija u sistemu, zadužena za rukovanje svim platformski zavisnim resursima, pokretanjem, praćenjem njihovog rada i zaustavljanjem ostalih aplikacija. Kako bi se napravio robustan sistem koji je jednostavno proširiti i prilagoditi različitim uslovima korišćenja potrebno je definisati takvu arhitekturu softvera i definisati skup pravila za proširenje funkcionalnosti. SM je zamišljen kao aplikacija koja može jednostavno da se proširuje na dva načina:

- dodavanjem novih funkcionalnosti
- prilagođenjempostojećih funkcionalnosti za nove platforme

SM je dizajniran kao softver koji prati komponentno-orjentisanu arhitekturu, odnosno fokusira se na dekompoziciji dizajna u funkcionalne i logičke komponente koje su predstavljene kroz dobro definisani komunikacioni interfejs ostatku sistema. Ovaj komunikacioni interfejs sadrži listu metoda sa tačnim tipovima ulaznih i izlaznih parametara. Ovakav dizajn obezbeđuje viši nivo apstrakcije i rasparčava problem u više podproblema, od kojih je svaki asociranna određenu komponentu.

3.1 Modularnost

SM upravlja različitim delovima sistema GW kao što je rukovanje indikacije LE dioda, detekcija korisničkih akcija sa tastera, podešavanje mreže i slično. Ideja je da se ove različite grupe funkcionalnosti grupišu u obliku slabo povezanih, ponovo upotrebljivih komponenti sistema od kojih svaka komponenta enkapsulira svoju specifičnu implementaciju i izlaže je kroz apstraktni interfejs. Ove gradivne komponente SM nazivaju se *kontrolerima* i svaki kontroler se,

zavisno od potreba, može isključiti iz SM ili ponovno koristiti u kombinaciji sa drugim kontrolerima.

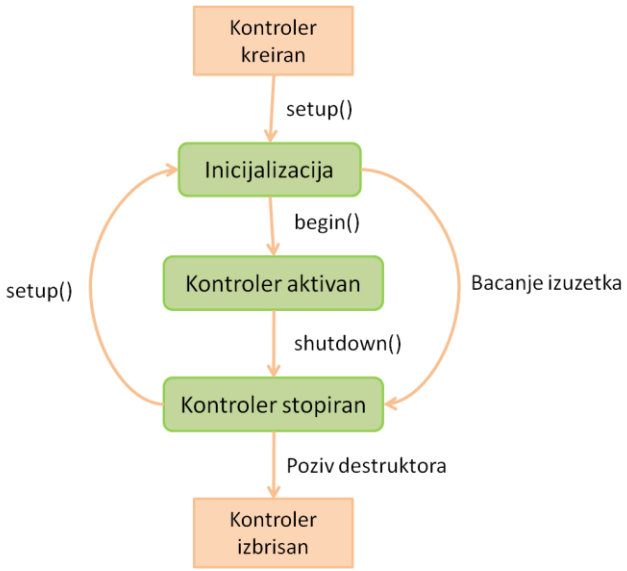
3.1.1 Apstraktni interfejs kontrolera

Kontroler je zadužen da rukuje određenom grupom funkcionalnosti sistema, on sadrži konkretne algoritme i strukture podataka za obavljanje tih akcija, međutim detalji konkretne implementacije ne trebaju biti vidljivi ostatku sistema. Kako bi se obezbedila raspodela odgovornosti, svaki kontroler definiše listu svojih metoda i konkretnih tipova podataka koje te metode koriste kao svoje parametre. Tipovi podataka se definišu posebno za ulazne i izlazne parametre metoda. Ove metode su sprega kontrolera sa ostatkom sistema i preko njih se kontroleru vrši promena stanja i iniciraju akcije od strane ostalih kontrolera sistema. Lista prototipova ovih metoda se naziva apstraktni opis kontrolera i on faktički predstavlja minimalniskup informacija koje je potrebno imati o kontroleru kako bi se izvršila akcija nad njim, nalazi se u posebnom fajlu kako bi mogao biti uključen u ostalim kontrolerima.

Metode i tipovi podataka koje definiše apstraktni interfejs kontrolera definišu akcije na visokom nivou, bez ulaska u detalje implementacije i bez redundantnih informacija. Ovakva organizacija obezbeđuje da se promene u implementaciji konkretnog kontrolera ne reflektuje kao promena u njegovom prototipu, a samim tim i promene u ostatku sistema koji ga koristi.

3.1.2 Životni ciklus kontrolera

Kako bi kontroler mogao biti uključen u SM on mora ispoštovati određeni skup pravila u pogledu strukture i funkcionalnosti koje propisuje SM., jedan od tih zahteva je definisanje apstraktnog interfejsa, opisanog u prethodnom poglavlju.



Slika 3 Životni ciklus kontrolera

Iz perspektive implementacije konkretnog kontrolera, sprega sa sistemom se obavlja kroz metode povratnog poziva životnog ciklusa (engl. *lifecycle callbacks*) koje obaveštavaju kontroler o promenama stanja sistema. Tokom životnog veka kontrolera, SM poziva skup metoda povratnog poziva u redosledu koji je predstavljen u slici 3 (nazivi pored strelica) i tako definiše stanja u kojima se kontroler može naći (nazivi u zelenim blokovima).

U svakoj od navedenih metoda povratnog poziva, kontroler je potrebno da izvrši određeni skup akcija zavisno od semantike konkretnog stanja u kom se nalazi, tačan opis stavki koje treba izvršiti nalazi se u Tabela 1, ni jedna od navedenih akcija nije obavezna već zavisi od konkretnih potreba kontrolera. Način na koji će kontroler izvršiti te akcije nije precizno definisan, ali je bitno da se njihovo izvršavanje obavlja u tačno definisanoj metodi. SM obezbeđuje skup pomoćnih funkcija koje mogu biti iskorišćene za izvršenje pojedinih akcija.

Naziv metode	**Akcije koje trebaju biti izvršene**
setup()	• Registracija metoda rukovalaca poruka, čiji prototipovi su definisani u apstraktnom interfejsu kontrolera. • Inicijalizacija stanja kontrolera na osnovu podataka sačuvanih u toku svog prethodnog rada. (obradom perzistovanih podataka)
begin()	• Dobavljanje referenci na kontrolere SM koji su potrebni za rad konkretnog kontrolera. • Alociranje proizvoljnih objekata koji su potrebni za rad kontrolera. • Pokretanje dodatnih niti za obradu kontrolera • Otvaranje konekcije sa udaljenim serverom. • Zauzimanje ostalih resursa.
shutdown()	• Zaustavljanje kreiranih niti kontrolera • Zatvaranje svih otvorenih konekcija. • Dealociranje zauzetih resursa
Poziv destruktora	• Dealociranje zauzetih resursa

Tabela 1 Skupovi akcija koji se izvršavaju pozivima metoda životnog ciklusa kontrolera

Ideja je da se u procesu inicijalizacije tačan oblik kontrolera, tj. da se prototipovi iz apstraktnog interfejsa povežu sa konkretnim implementacijama, koje će u aktivnom stanju biti pozivane radi obrade podataka.

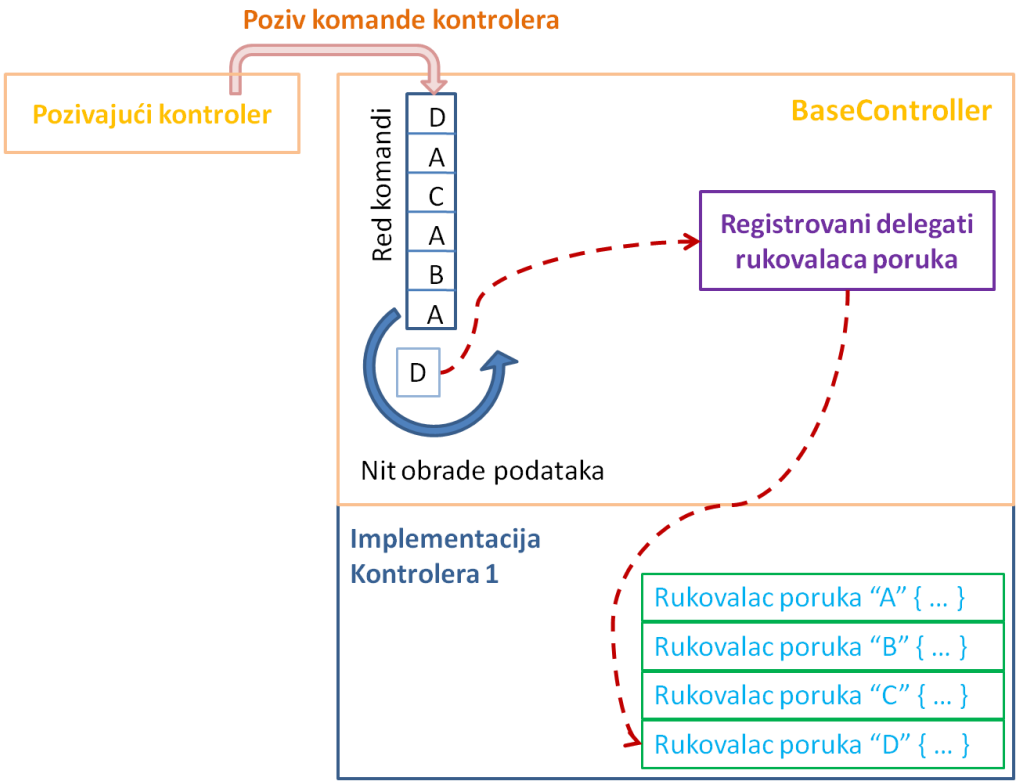
3.1.3 Struktura kontrolera

Svi kontroleri u SM nasleđuju apstraktnu klasu osnovnog kontrolera (*BaseController*), a SM pristupa svim kontrolerima kroz metode te apstraktne klase, ne ulazeći u detalje njihove implementacije i pojedine specifičnosti. Implementacija klase *BaseController* pokušava da enkapsulira što veći broj akcija koje su zajedničke za kontrolere i sakrije specifičnosti implementacije samog SM od implementacije kontrolera. *BaseController* je zadužen da obezbedi radno okruženje za jednostavnu i bezbednu implementaciju konkretnih kontrolera. Kako je osnovni način za izlaganje funkcionalnosti kontrolera pomoću rukovalaca poruka, jedna od osnovnih zadataka *BaseController*-a je da obezbedi jednostavan mehanizam za njihovu registraciju i učini njihovo pozivanje jednostavnim za ostatak sistema.

Kako je od izuzetne važnosti da SM bude robusna aplikacija, potrebno je obezbediti takvo radno okruženje koje će smanjiti mogućnost za nastajanje grešaka uz što manje narušavanje performansi. Kontroler može da sadrži više rukovalaca poruka i svaki od njih u nekom trenutku može biti pozvan iz više različitih niti, što je iz aspekta programiranja standardna pojava, međutim pisanje konkurentnih programa je kompleksnija aktivnost od pisanja sekvencijalnih programa, a samim tim i podložnija greškama. Da bi se implementacija kontrolera rasteretila od, što je moguće više, stvari koje mogu ugroziti stabilnost odlučeno je da se pozivi svih rukovalaca poruka unutar jednog kontrolera serijalizuju na nivou *BaseController*-a i tako rasterete implementaciju konkretnog kontrolera od kreiranja mehanizama za zaštitu od trke podataka i nastajanja nekonzistentnog stanja unutar kontrolera. Svaki kontroler, prilikom svoje inicijalizacije, registruje listu rukovalaca poruka sa sledećim podacima:

- Delegatom metode koja će biti pozvana za obradu podataka
- Tipovima ulaznih i izlaznih parametara metode
- Jedinstvenim simboličkim imenom rukovaoca poruka (tekstualna vrednost)
- Jedinstvenim identifikatorom rukovaoca poruka (numerička vrednost)

Ovi podaci su dovoljni da se konkretne metode povežu sa metodama apstraktnog interfejsa i da ostatak sistema može koristiti usluge novog kontrolera. Prilikom poziva funkcije kontrolera, kreira se nova komanda koja se ubacuje u red sa komandama. Komandna nit sekvencijalno vadi iz reda pristigle komande i na osnovu jedinstvenog identifikatora funkcije apstraktnog interfejsa, pronalazi registrovani delegat unutar implementacije kontrolera i poziva ga. Crvena isprekidana linija na slici 4 prikazuje tok obrade komande.



Slika 4 Organizacija obrade podataka u kontroleru

U slučaju da ne postoji registrovani rukovalac poruka sa datim identifikatorom komande ili su ulazni parametri pogrešnog tipa, *BaseController* će „baciti“ izuzetak pozivaocu sa opisom greške.

3.1.4 Komunikacija između kontrolera

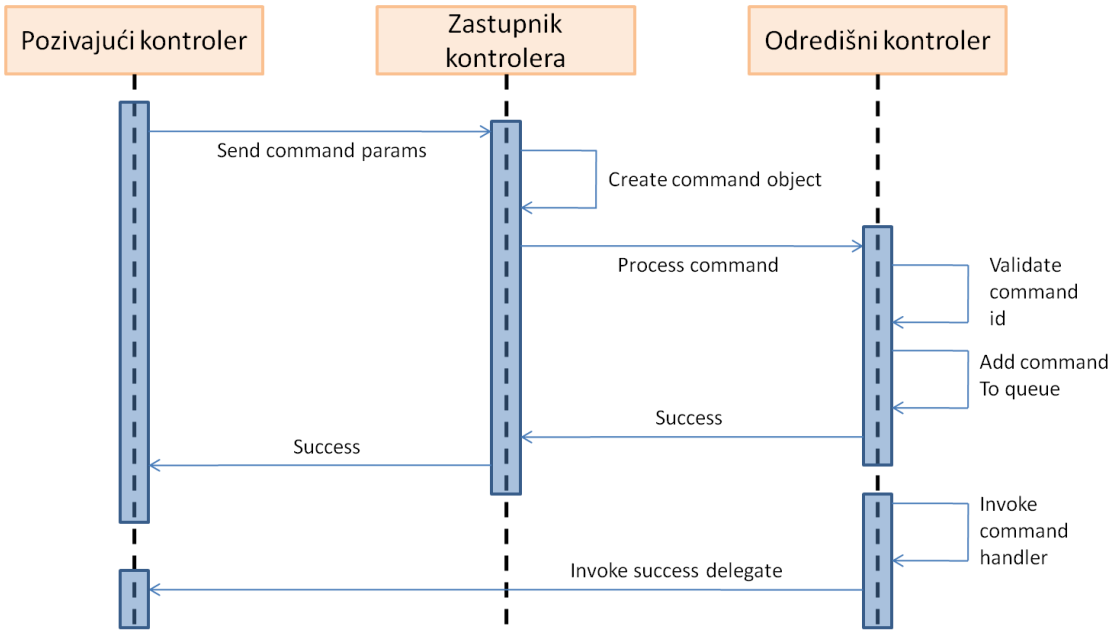
Već je pomenuto kako se komunikacija između dva kontrolera vrši posredstvom komandnih poruka. Da bi se razumeli detalji u komunikaciji između kontrolera, potrebno je pre svega definisati strukturu komande poruke, kao nosioca informacije i strukturu njenih polja, a zatim način na koji kontroleri vrše međusobnu interakciju. Komandna poruka sadrži sledeća polja:

- **Jedinstveni identifikator rukovaoca poruke**: Obavezan parametar, služi da se pomoću njega adresira tačan rukovalac poruka unutar kontrolera koji treba da obradi komandu.
- **Ulazne parametre za poziv metode:** Obavezan parametar predstavlja ulazni argument metode kojimora biti objekat definisanog tipa.
- **Delegat povratnog poziva za dobijanje rezultata obrade / informacije o grešci**: Opcioni parametar, služi da se asinhrono obavesti pozivajući kontroler o rezultatu obrade. Definiše dva delegata od kojih će jedan biti pozvan u slučaju da se uspešno obradi poruka sa argumentima koji odgovaraju izlaznim parametrima rukovaoca

poruke. U slučaju da se desi greška (kontroler prilikom obrade poruke „baci“ izuzetak) biće pozvan delegat greške sa argumentom nastalog izuzetka.

- **Referenca na kontekst:**Opcioni parametar,služi da pozivalac postavi referencu na proizvoljnu memorijsku strukturu i prilikom asinhronog poziva odgovora, biće mu vraćena referenca na istu tu strukturu. Služi kako bi se lakše povezali zahtevi i odgovori u kontroleru pozivaoca.

Pored komandne poruke, u komunikaciji bitnu ulogu ima i posrednik kontrolera (eng. controller proxy) koji služi kao sprega za slanje poruka ciljanom kontroleru. Prilikom poziva metode *begin()* kontroler može dobaviti posrednike kontrolera sa kojima će vršiti komunikaciju tokom svog rada. Dijagram koji prikazuje osnovni proces komunikacije predstavljen je na slici 5 i podrazumeva da kontroler pozivaoc zna tačan identifikator rukovaoca poruka, da sam kreira ulazne argumente (sam zauzimaodgovarajući objekat) i očekuje asinhroni odgovor pozivom metode *onResult()*. Pozivajući kontroler prosleđuje posredniku kontrolera: identifikator rukovaoca poruka, kreirane ulazne parametre i delegat za asinhroni odgovor. Posrednik kontrolera na osnovu primljenih parametara kreira novu komandu i prosleđuje je odredišnom kontroleru. Odredišni kontroler proverava validnost identifikatora rukovaoca poruke u komandi i u slučaju da je postojeći, ubacuje komandu u red za obradu. U niti za obradu komandi, odredišni kontroler vadi iz reda pristiglu komandu, na osnovu njenog identifikatora pronalazi odgovarajući delegat za obradu i proverava validnost tipa ulaznih parametara. U slučaju da je tip validan, poziva se odgovarajuća metoda, a njena povratna vrednost se vraća pozivajućem kontroleru pozivom metode delegata.



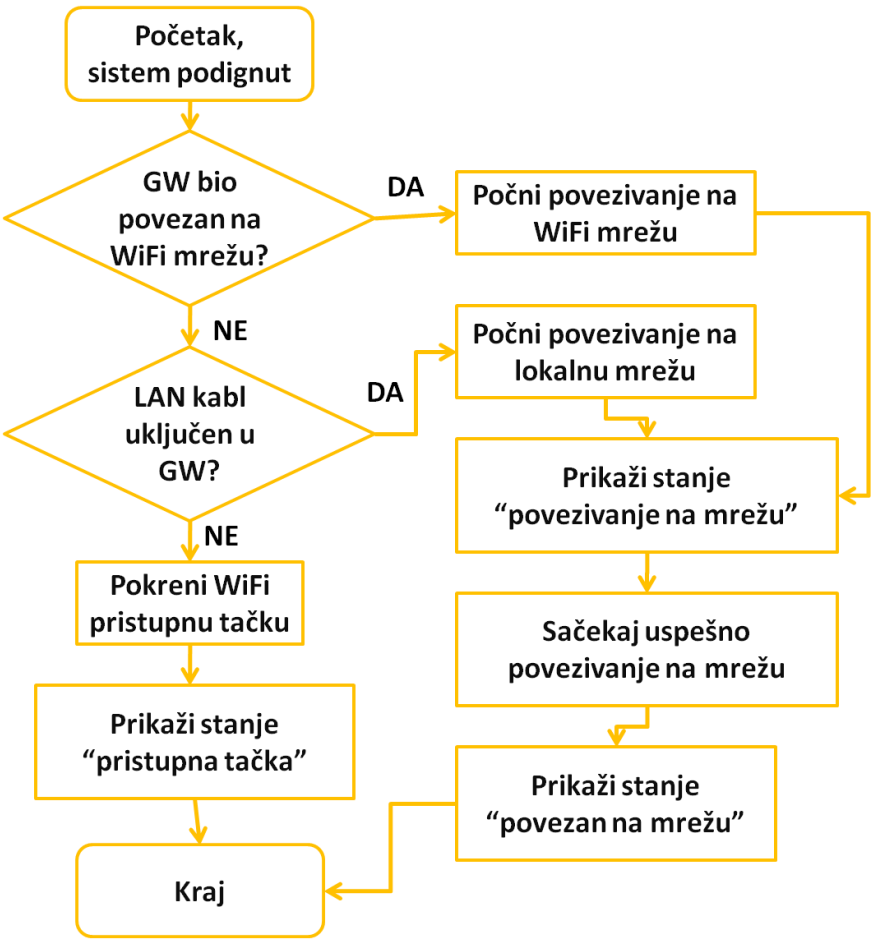
Slika 5 Redosled poziva kontrolera u međusobnoj komunikaciji

Posrednik kontrolera sadrži i dodatne metode za slanje komandi koje omogućavaju da se umesto identifikatora rukovaoca poruke definiše njegov simbolički naziv i da se umesto konkretnog objekta ulaznih parametara prosledi serijalizovana vrednostna osnovu koje će biti kreiran objekat, o čemu će biti reči u narednim poglavljima.

3.2 Prenosivost

Pored toga što treba da obezbedi jednostavan mehanizam za implementaciju različitih funkcionalnosti, SM treba da obezbedi da se ta funkcionalnost lako i brzo može prilagoditi novoj ciljanoj platformi. U SM-u ovaj problem je rešen uvođenjem posebnog sloja za apstrakciju platforme nazvanog PAL (engl. *platform abastraction layer*) koji se zasniva na ideji da svi platformski zavisni delovi implementacije softvera trebaju biti izdvojeni, a njihova funkcionalnost jasno izložena kroz uniforman interfejs višim slojevima. Ovim bi se arhitektura softvera SM takođe mogla svrstati u slojevitu, a njeni slojevi podeliti na:

- Platformski nezavisan (gornji sloj)
- Sloj za prilagođenje platforme PAL (donji sloj)



Slika 6 Algoritam željenog ponašanja GW prilikom startovanja sistema

Ideja je da se u gornjem slojuimplementirazajedničko ponašanje svakog GW, a koje je definisano nezavisno oddetalja konkretne platforme. Ovo ponašanje se precizno definiše korišćenjem apstraktnih akcija GW u određenim situacijama. Jedan primer je rukovanje mrežnim podešavanjima GW u trenutku podizanja sistema što se može predstaviti algoritmom na slici 6.

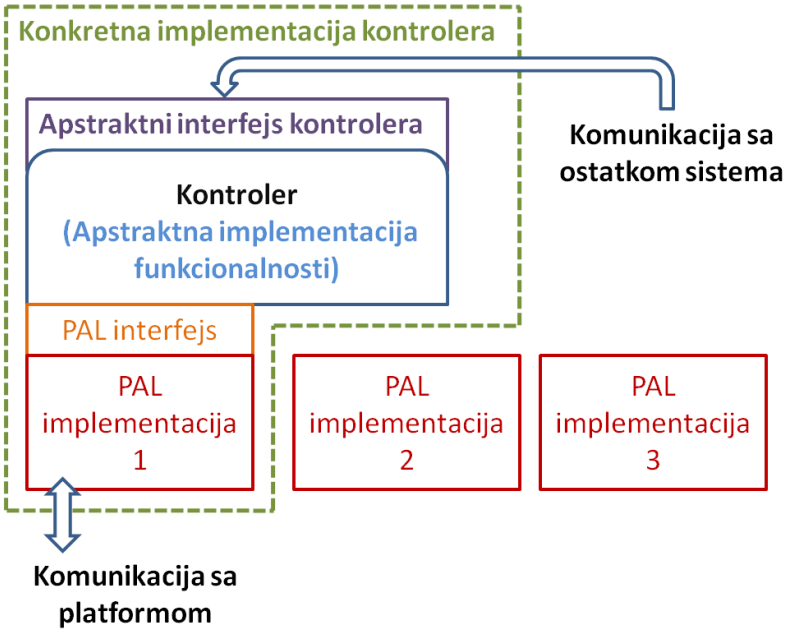
Algoritam sa slike definiše opšte zahtevano ponašanje GW, ali ne ulazi u detalje implementacije karakteristične za određenu platformu. Gornji sloj služi da implementira ove zahteve, a konkretnu realizaciju akcija delegira PAL sloju. Gornji sloj stoga definiše tačan interfejs koji PAL mora implementirati, a on se svodi na definisanju apstraktnih akcija koje se moraju naći na svakoj platformi i na osnovu kojih je definisano ponašanje GW.

Ovakvim dizajnom je omogućeno da se prenos softvera na različite platforme svede na implementaciju konkretnih akcija iz PAL interfejsa specifičnih za datu platformu.

3.2.1 Prenosivost kontrolera

Kako je funkcionalnost SM podeljena u zasebne komponente – kontrolere, prenosivost sistema se zasniva na prenosivosti tih kontrolera. Svaki kontroler stoga definiše PAL interfejs na čijim apstraktnim akcijama se zasniva njegova funkcionalnost. Prilagođenje kontrolera za platformu zasniva se implementaciji konkretnog PAL sloja.

Kontroler se stoga može posmatrati kao komponenta koja definiše poslovnu logiku jednog dela sistemana platformski nezavisnom nivou, a koja može biti prilagođena za različite platforme jednostavnom implementacijom PAL sloja, što je prikazano na slici 7.



Slika 7 Princip prilagođenja kontrolera za različite platforme

Kontroler je u ovakvoj implementaciji “ograđen” sa dva interfejsa:

- sa gornje strane apstraktni interfejs kontrolera izlaže njegovu funkcionalnost prema ostatku sistema,
- dok sa donje strane kontroler koristi usluge PAL interfejsa za izvršenja akcija na konkretnoj platformi.

U slučaju da se ova dva interfejsa ne menjaju, moguće je kreiranje više različitih ponašanja kontrolera jednostavnom zamenom njegove apstraktne implementacije, bez ikakvog uticaja na ostatak sistema.

3.3 Jezgro softvera za upravljanje sistemskim resursima

Kako bi kontroleri funkcionisali na način na koji su definisani, neophodno je postojanje određenog dela softvera koji će se baviti njihovim upravljanjem. Pod upravljanjem ovde se pre svega misli na njihovo instanciranje, rukovanjem životnim ciklusom, pomoći u međusobnoj razmeni poruka i obezbeđivanjem pomoćne funkcinalnosti (engl. *utils*).

3.3.1 Podsistem za evidentiranje događaja

Kako u procesu razvoja, tako i u produkcionom okruženju faktički je neophodno postojanje određenih ispisa evidencije događaja (engl. *logs*). Jezgro SM obezbeđuje centralizovan sistem za evidentiranje događaja koji se može na jednostavan način konfigurisati da zapise o događajima aplikacije ispisuje kako u fajlove tako i na standardni konzolni ispis. Podržan je i ispis događaja sa različitim nivoima važnosti.

Svaka komponenta sistema može da koristi usluge mehanizma za evidenciju događaja tako što će definisati pod kojim imenom želi da se njeni zapisi prikazuju i da prilikom ispisa definiše nivo važnosti konkretnog događaja. Događaje je moguće filtrirati po nazivu komponente i po važnosti ispisa.

3.3.2 Čuvanje podataka

Očekivani način korišćenja GW podrazumeva iznenadne nestanke napajanja i ponovna pokretanja sistema, npr. korisnik može u svakom trenutku izvaditi GW iz struje. U trenutku ponovnog pokretanja, potrebno je da se GW vrati u stanje u kom je prethodno bio i da se povrate sva prethodno sačuvana podešavanja.

Jezgro SM za ove potrebe obezbeđuje centralizovani sistem za čuvanje podataka na masovnoj memoriji i kroz jednostavnu spregu, omogućava kontrolerima da ažuriraju i u trenutku podizanja sistema povrate svoja podešavanja. Podešavanja su organizovana u obliku parova ključ – vrednost (engl. *key-value pairs*), odnosno podacima se pristupa po vrednosti njihovog ključa. Ovim pristupom kontroleri ne ulaze u detalje implementacije vezane zakonkretno čuvanje podataka, već samo naznačenju datih podataka.

Konkretno čuvanje podataka se obavlja korišćenjem SQLite baze podataka koju kreira SM i popunjava je na zahtev kontrolera. Ova baza podataka sadrži jednu relaciju sa 3 obeležja:

- Naziv kontrolera (tekstualna vrednost)
- Naziv ključa (tekstualna vrednost)
- Vrednost (tekstualna vrednost)

Ključ šeme relacije predstavlja naziv kontrolera i naziv ključa, što omogućava da svaki kontroler ima svoj nezavisni skup podešavanja koja neće dolaziti u konflikt sa podešavanjima ostalih kontrolera. Upis podešavanja koja već postoje u bazi tretiraju se kao ažuriranje, odnosno stari podaci se zamenjuju novim.

Kontroleri u toku svog rada mogu u bilo kom trenutku da sačuvaju i dobave pojedinačna podešavanja od SM posredstvom ugrađenih metoda *BaseController*-a. Međutim prilikom startovanja sistema potrebno je obezbediti mehanizam kojim će kontroler moći da se na uniforman način inicijalizuje pri prvom pokretanju (kada je baza podataka prazna) i u slučaju kada baza ima sačuvane podatke. Kako bi se izbeglo proveravanje praznih vrednostiprilikom inicijalizacije, uveden je dodatni korak u životnom ciklusu kontrolera koji omogućava kontroleru da definiše svoja podrazumevana podešavanja koja će biti korišćena u slučaju da nema snimljenih podešavanja u bazi podataka (najčešće pri prvom pokretanju).

Ovim je obezbeđeno da se podrazumevana podešavanja kontrolera nalaze unutar jedne metode, a da se inicijalizacija kontrolera obavlja uvek na identičan način, nezavisno od stanja baze podataka.

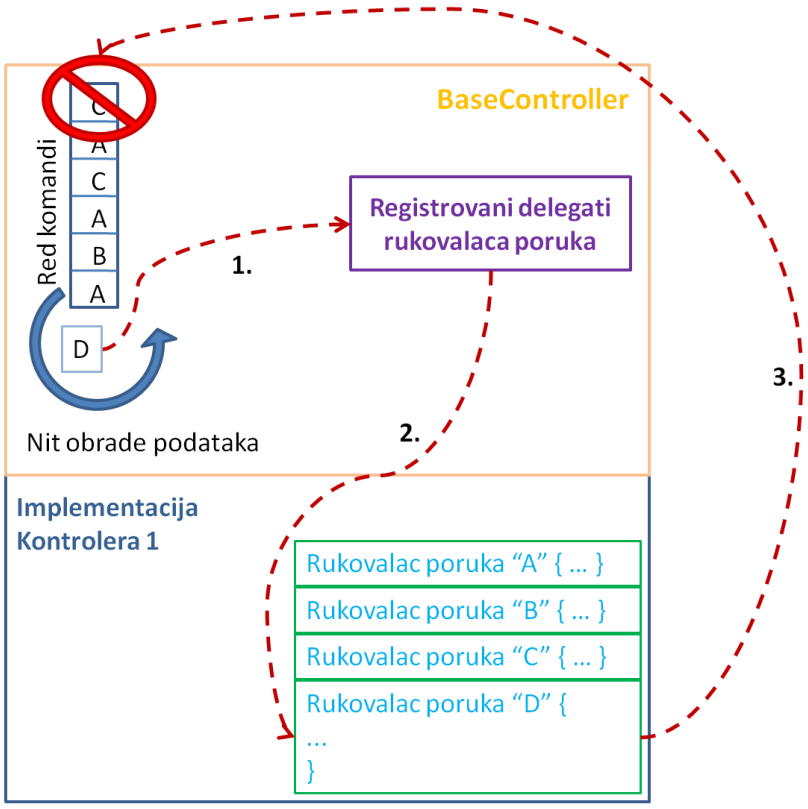
3.3.3 Problem zaključavanja kontrolera

Obrade komandi izvršavaju se sekvencijalno, pozivom odgovarajućih rukovalaca poruka kontrolera iz niti obrade podataka. Kontroler koji obrađuje poruku, rezultat te obrade vraća kao povratnu vrednost funkcije, a rezultat se tada asinhrono dostavlja pozivajućem kontroleru, pozivom delegate odgovora (Slika 5). Međutim u nekim slučajevima izvršenje komande u rukovaocu poruka jednog kontrolera zavisi od rezultata obrade drugih kontrolera, drugim rečima izvršenje rukovaoca poruke mora biti blokirano dok se ne dobije odgovor koji asinhrono stiže od drugog kontrolera. Efekat ovoga je da se kompletno izvršavanje komandne niti jednog kontrolera blokira, čekajući da se komanda koju je poslao obradi u drugom kontroleru.

U trenutnoj arhitekturi sistema, čekanjem na odgovor poslate komandeunutar rukovaoca poruke kontrolera može se blokirati kompletan kontroler, a samim tim i ceo SM na dva načina. Treba napomenuti da se ovi problem isključivo odnose na slučaj u kom se blokirajući pozivi izvršavaju iz komandne niti kontrolera, odnosno pozivom iz rukovaoca poruka ili njegovih podpoziva.

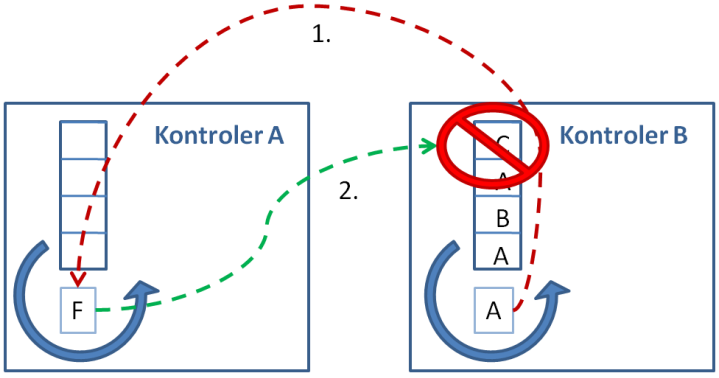
3.3.3.1 Problem rekurzivnog poziva

Jedan problem nastaje u slučaju kad kontroler šalje poruku samom sebi i blokira se dok ne dobije odgovor. Ovaj slučaj je predstavljen na slici 8, crvena isprekidana linija predstavlja tok poziva funkcija niti obrade podataka koja se blokira u trenutku ubacivanja komande u red. Da bi se obradila komanda koju je kontroler poslao samom sebi, potrebno je da se pre toga završi trenutno obrađivana komanda, ali obzirom da on čeka na odgovor, kontroler postaje blokiran.



Slika 8 Situacija zaključavanja kontrolera u trenutku rekurzivnog poziva

Na prvi pogled ovaj problem se može rešiti boljom implementacijom konkretnog kontrolera koji neće pozivati samog sebe, već podatke dobiti na drugačiji način (direktnim pozivom svoje metode). Međutim problem bi se i dalje mogao pojaviti u slučaju da drugi kontroler u svojoj obradi blokirajući poziva prvi kontroler, a prvi kontroler u svojoj realizaciji blokirajuće poziva drugi, slici 9. Rešenje bi se svelo na to da se svi kontroleri drugačije implementiraju kako ne bi došlo do ovoga, ali ovakvim pristupom bi se izgubio osnovni princip modularnosti sistema, a to je da komponente ne trebaju da zavise od implementacije drugih komponenti. Očigledno je potrebno bolje rešenje za ove potrebe.

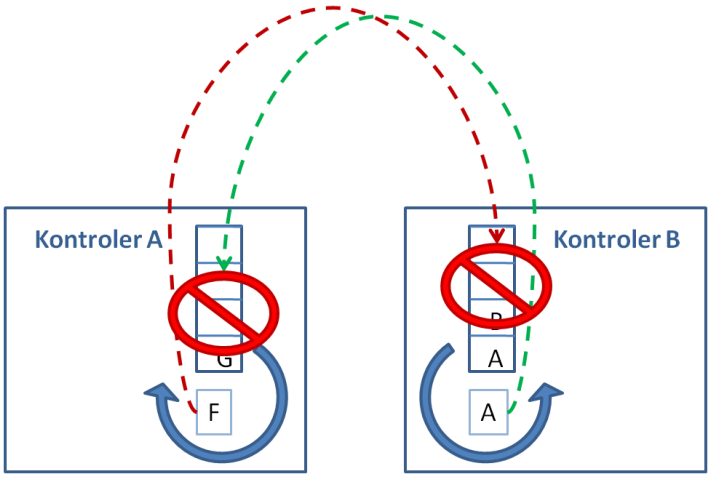


Slika 9 Blokiranje više kontrolera u rekurzivnom pozivu

3.3.3.2 Problem međusobne zavisnosti izvršavanja rukovalaca poruka

Za razliku od prethodnog problema koji je ponovljiv (manifestuje prilikom svakog poziva) slanje blokirajućih komandi može dovesti i do blokiranja koje zavisi od vremena preključivanja niti operativnog sistema i redosleda na koji se funkcije unutar kontrolera izvršavaju.

Situacija u kojoj problem nastaje ilustrovana je na slici slici 10 i predstavlja dva kontrolera koja u isto vreme šalju komande jedan drugom dok blokirajući čekaju na odgovor. Komande završavaju u redu komandi ova dva kontrolera, ali ne stignu nikad da budu izvršene obzirom da je komandna nit blokirana čekajući odgovor od drugog kontrolera.



Slika 10 Blokiranje kontrolera u međusobin pozivima

3.3.3.3 Rešenje problema

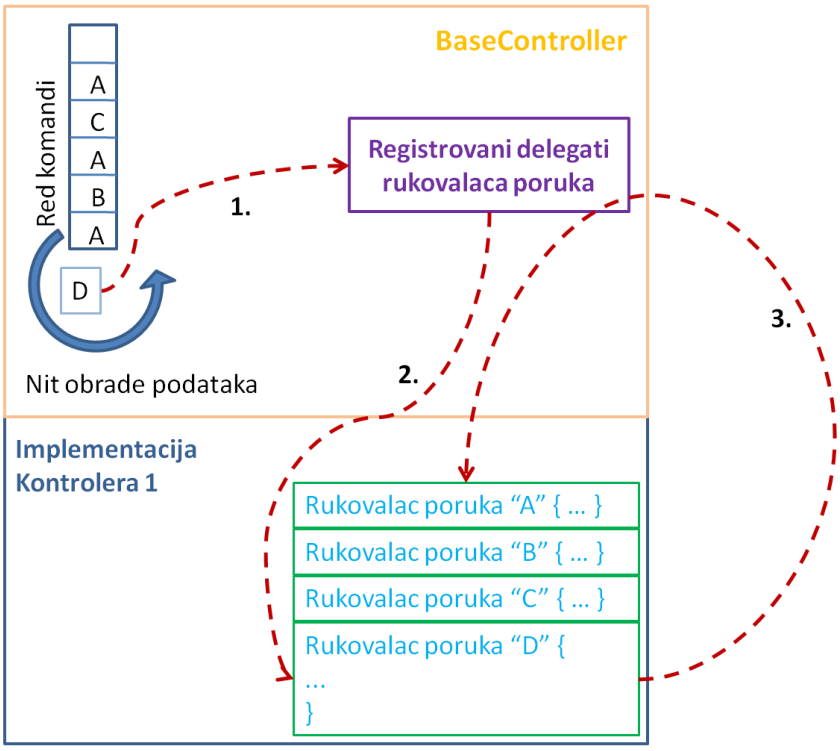
Postojećim mehanizmima koje obezbeđuje SM nije moguće rešiti navedene probleme, sekvencijalni način izvršavanja poruka koje se uzimaju iz reda će uvek dovesti do blokiranja u slučaju rekurzivnih poziva. Uzrok drugog problema, pored sekvencijalnog načina izvršavanja komandi, je u tome što ne postoji međusobna koordinacija između pozivanja kontrolera pa se u pojedinim slučajevima može dogoditi greška. Jedno rešenje bi bilo da se svi odgovori obrađuju

asinhrono i u tom slučaju bi se problem izbegao, međutim ovakvim pristupom bi se implementacija funkcionalnosti kontrolera znatno zakomplikovala, što nije opcija.

Drugo rešenje se zasniva na uvođenju posebnog mehanizma za slanje komandi kontrolera na sinhron načinkoji bi sprečio nastajanje navedenih grešaka.

3.3.3.4Sinhroni pozivi komandi kontrolera

Kao rešenje na predstavljeni problem uvedena je posebna metoda zastupnika kontrolera *sendSyncCommand* koja se razlikuje od slanja obične komande kontroleru. Komande koje se šalju kontroleru na ovaj način zaobilaze red komandi odredišnog kontrolera i direktno pozivaju delegate rukovalaca poruka, što znači da se tako pozvan rukovalac unutar kontrolera ne izvršava u svojoj komandnoj niti, već u niti pozivaoca (Slika 11).



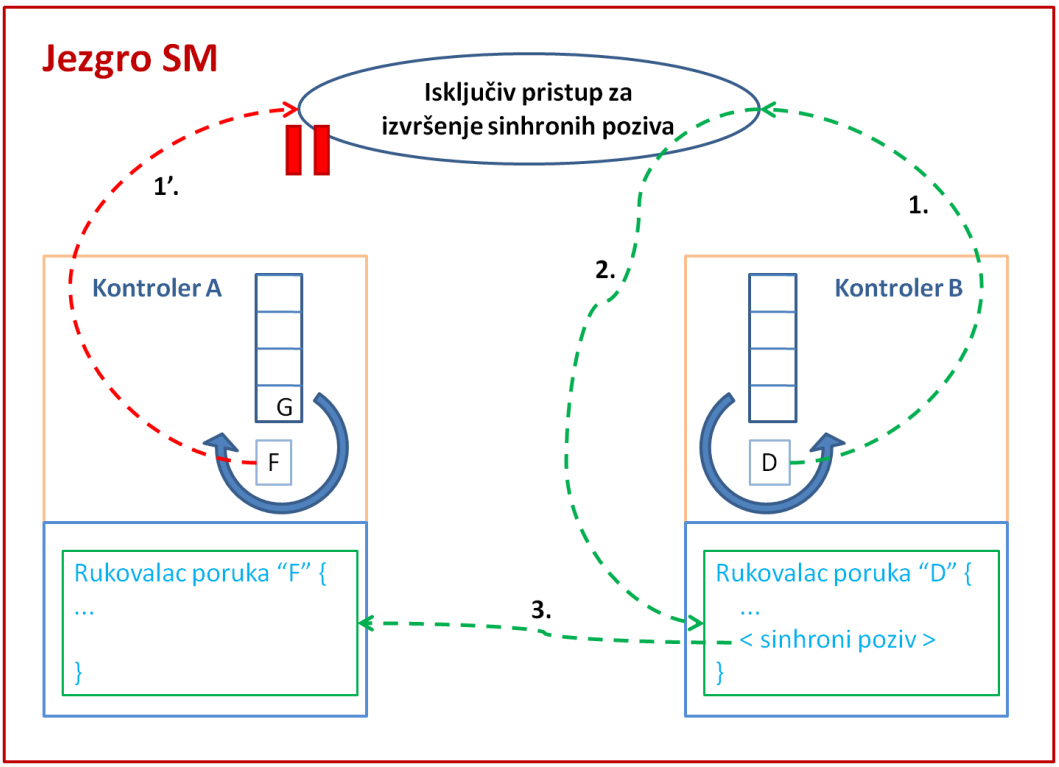
Slika 11 Sinhroni pozivi bez blokiranja

Da bi se održao princip sekvencijalnog izvršavanja komandi kontrolera, potrebno zaključavati pozive rukovaoca porukarekurzivnim mutex-om koji bi obezbedio da se rekurzivni pozivi kontrolera ne blokiraju. Komanda nit kontrolera bi u tom slučaju "čekala" da se izvrše sinhroni pozivi i tek tad nastavila sa obradom komandi iz reda. Stvar na koju treba obratiti pažnju u ovom rešenju je to što se sinhrone komande izvršavaju preko reda.

Ovo rešenje sprečava blokiranje kontrolera u rekurzivnim pozivima, međutim drugi problem u kojem se dva kontrolera međusobno pozivaju u istom trenutku i blokiraju,ostaje nerešen. Osnovi uzrok ovog problema predstavljapostojanje jedne niti za obradu komandi i činjenica da se rukovaoci poruka kontrolera međusobno isključivo izvršavaju. Detaljnom

analizom problema došlo se do zaključka da se ovaj problem može rešiti sprečavanjem istovremene obrade više rukovaoca poruka kontrolera koji u svojoj implementaciji imaju sinhrone pozive drugih kontrolera. Drugim rečima ako rukovalac poruke koristi sinhroni poziv ka nekom drugom kontroleru, on mora biti jedini izvršavani rukovalac poruka u sistemu u tom trenutku. Ovim se sprečava da dva ili više kontrolera istovremeno međusobno pošalju poruke i tako se blokiraju.

Kako bi se implementiralo ovo rešenje pre svega potrebna je informacija o tome koji rukovalac poruka koristi sinhrone pozive u svojoj realizaciji i definiše se u implementaciji konkretnog kontrolera, prilikom registracije delegata rukovaoca poruka. Ovom informacijom se rukovaoci poruka kontrolera mogu podeliti na one koji koriste i one koji ne koriste sinhrone pozive u svojoj implementaciji. Ovu informaciju koristi *BaseController* i pre poziva delegata rukovaoca poruka koji koristi sinhrone pozive, zahteva isključiv pristup od jezgra SM. Tek kada kontroler dobije isključiv pristup od jezgra SM, on započinje izvršenje delegata. Ovim pristupom se obezbeđuje da rukovaoci poruka jednog kontrolera uvek bududostupniza sinhrone pozive iz rukovaoca drugih kontrolera.



Slika 12 Rešenje sinhronih poziva u međusobnim pozivima kontrolera

Na slici 12 prikazana je situacija u kojoj kontroleri A i B istovremeno počinju sa obradom komandi koje pozivaju rukovaoce poruka sa sinhronim pozivima. Oba kontrolera pre poziva delegata zahtevaju od jezgra SM isključiv pristup, kontroler B uspeva da dobije pristup dok se izvršavanje kontrolera A pauzira (trenutci 1 i 1'). Kontroler B počinje sa obradom poruke,

pozivom delegata rukovaoca poruka D (trenutak 2), u toku izvršenja metode sinhrono se poziva rukovalac poruka “F” iz kontrolera A. Ovaj poziv će se izvršiti bez blokiranja i nakon završetka obrade poruke “D”, kontroler A će dobiti isključiv pristup i moći će da nastavi sa svojom obradom.

Treba naglasiti da se ovim pristupom svi rukovaoci poruka u sistemu koji imaju sinhrone pozive sekvencijalno izvršavaju. Ovo usporava izvršenje komandi u sistemu, čak i kada je sekvencijalnost izvršenja nepotrebna, kao što su slučajevi u kojima se kontroleri ne pozivaju međusobno. Međutim analizom potreba sistema došlo se do zaključka da ovo neće predstavljati problem jer se sinhroni pozivi neće često koristiti, a robusnost je prevashodni prioritet sistema kao što je SM.

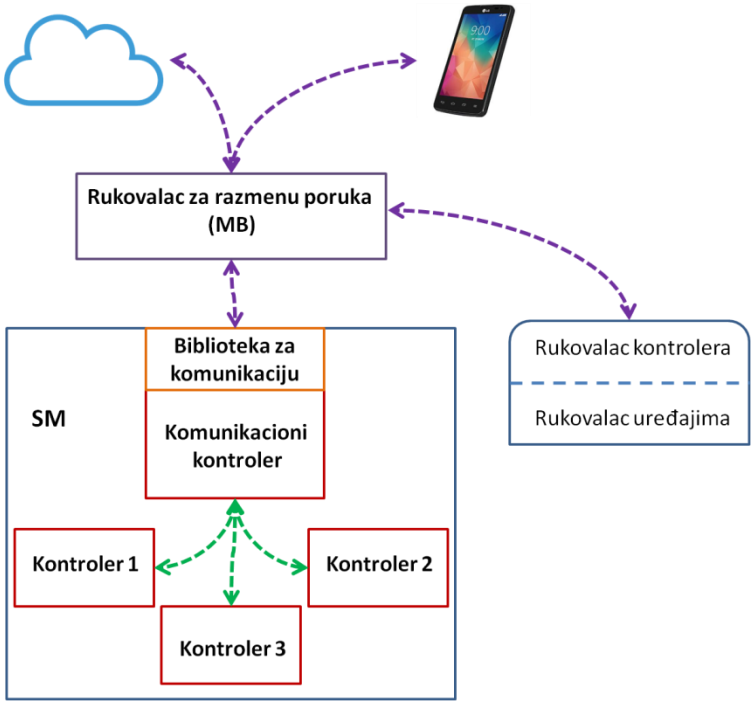
3.4 Kontroleri SM

Kontroleri SM podeljeni su na takav način da svaki od njih rukuje jednim delom funkcionalnosti sistema i ne zavisi direktno od postojanja ostalih kontrolera. Analizom potreba sistema došlo se do sledećeg skupa kontrolera.

3.4.1 Komunikacioni kontroler

Pored komunikacije između kontrolera unutar SM (3.1.4), takođe je potrebno obezbediti mehanizam kojim će se vršiti komunikacija SM i ostatka aplikacija iz sistema, kao i komunikacija sa udaljenim servisima. Ovu funkcionalnost obezbeđuje rukovalac za razmenu poruka (MB) koji se nalazi na sistemu kao posebna aplikacija, a povezivanje sa njim se obavlja posredstvom postojeće biblioteke. MB definiše uniforman način za razmenu poruka između svih aplikacija sistema.Podaci koji se razmenjuju definiše protokol i predstavljaju JSON tekstualne strukture, a njihov tačan sadržaj i semantiku definišu aplikacije koje odgovaraju na te poruke. Kako je JSON tekstualni format korišćen za prenos serijalizovanih podataka, a SM obezbeđuje internu komunikaciju isključivo razmenom lokalnih memorijskih struktura (objekata) između kontrolera, potrebno je napraviti odgovarajuću spregu između ova dva dela.Sprega zahteva postojanje mehanizma za serijalizaciju / deserijalizaciju JSON podataka u specifične poruke kao memorijske strukture, ali takođe i adresiranje tačnih kontrolera i obrađivača poruka u njima, zarad izvršenja komandi.

Za ovu namenu uvodi se kontroler koji je zadužen da ostvari vezu sa MB i datu međuprocesnu komunikaciju preslika na komunikaciju između kontrolera, ovaj kontroler se naziva komunikacioni kontroler i može se posmatrati kao sprega SM sa ostatkom sistema.Komunikacioni kontroler treba da obezbedi razmenu poruka u oba smera, tj.da obezbedi kontrolerima SM slanje poruka drugim aplikacijama, a takođe i drugim aplikacijama da pozovu rukovaoce poruka kontrolera SM, Slika 13.



Slika 13 Organizacija interprocesne komunikacije unutar SM

Komunikacioni kontroler treba da obavi sledeće zadatke.

3.4.1.1 Izlaganje funkcionalnosti SM ostatku sistema

Kako bi se obezbedio mehanizam drugim aplikacijama da pozivaju SM, komunikacioni kontroler definiše strukturu poruke i algoritam kojim će se date poruke preslikati na interne pozive kontrolera. Poruka je JSON sa sledećim poljima:

Naziv polja	Opis
serviceName	Simboličko ime kontrolera unutar SM. Ime koje definiše jezgro SM prilikom instanciranja kontrolera
commandName	Simboličko ime rukovaoca poruke kontrolera. Definiše se prilikom registracije delegata u fazi inicijalizacije kontrolera (3.1.3).
params	Neobavezni parameter. Predstavlja JSON objekat sa proizvoljnom strukturom, na osnovu kog se treba kreirati ulazni parameter rukovaoca poruke kontrolera.

Tabela 2 Specifikacija poruke za komunikaciju sa SM

U trenutku primanja poruke, komunikacioni kontroler koristi vrednost polja "*serviceName*" i na osnovu njega, posredstvom jezgra SM, dobavlja zastupnik kontroler datog naziva. U slučaju da kontroler sa datim nazivom postoji šalje mu se komanda kroz dobijeni zastupnik kontrolera, ali za razliku od primera u poglavlju 3.1.4, umesto jedinstvenog identifikatora komande šalje se simbolički naziv komande (dobijen iz polja "*commandName*"), akao ulazniargument metode se

šalje JSON objekat dobijen iz polja "*params*", za razliku od kreiranog objekta datog tipa. Da bi se ovakav poziv mogao izvršiti, potrebno je uvesti određena proširenja unutar zastupnika kontrolera, objekata poruka i samih kontrolera. Pre svega zastupnik kontrolera mora na osnovu simboličkog imena rukovaoca poruke da dobije njen jedinstveni identifikator i to radi proverom liste registrovanih rukovaoca unutar kontrolera. Sledeći korak je da se kreira objekat ulaznih parametara na osnovu JSON podataka. Svaki kontroler prilikom registracije svojih rukovalaca poruka definiše i tačantipulaznih parametara, ta informacija se koristi u zastupniku kontrolera da se kreira objekat definisanog tipa i da se oninicijalizuje na osnovu pristiglih JSON podataka. Metoda za inicijalizaciju bi trebala da se nađe u implementaciji svakog objekta ulaznih parametara i da iz dobijenog JSON objekta izvadi potrebne informacije i postavi vrednosti svojih polja na date vrednosti. Kada se ovi koraci izvrše posrednik kontrolera ima sve podatke na osnovu kojih može kreirati komandu i ubaciti je u red komandi kontrolera.

Dobra posledica ovakve organizacije je u tome što se implementacija parsiranja (deserijalizacije) podataka određene poruke nalazi u objektu same poruke i komunikacioni kontroler tome pristupa samo kroz interfejs zastupnika kontrolera, na generički način. Komunikacioni kontroler ne trpi nikakve izmene prilikom dodavanje novih kontrolera i definisanjem novih rukovalaca poruka, što omogućava jednostavno skaliranje sistema.

3.4.1.2 Slanje poruka ostatku sistema

Komunikacioni kontroler definiše dve vrste metoda rukovalaca poruka koje izlažu funkcionalnost slanja poruka ostatku aplikacija sistema:

- Direktne poruke (eng. request – response messages). Ove poruke se adresiraju tačno određenoj aplikaciji, šalju zahtev i očekuju odgovor.
- Poruke o događajima (eng. event messages). Ove poruke služe da se celokupni sistem obavesti o nekom događaju unutar SM. Svaki događaj ima svoj naziv (tekstualnu vrednost) i sadržaj.

Izlaganjem ovih metoda, komunikacioni kontroler je obezbedio svim kontrolerima SM da vrše interprocesnu komunikaciju, što omogućava da se na jednostavan način funkcionalnost koja je specifična za SM, poveže sa ostatkom sistema.

3.4.1.3 Bezbednosni mehanizam

Komunikacioni kontroler obezbeđuje različitim aplikacijama sistema da pozivaju funkcije koje izlažu kontroleri SM, međutim nije poželjno dozvoliti svim aplikacijama kompletan pristup funkcionalnostima SM. Na primer, dobavljanje log fajlova GW treba dozvoliti samo serveru, a ne i mobilnoj aplikaciji.

Ovaj problem je rešen uvođenjem prava pristupa za svaki rukovalac poruka koji izlaže kontroler. Uvedene su 4 vrste prava pristupa koje prilikom registracije rukovaoca poruka, kontroler može da definiše a ona mogu biti:

- INTERNAL pozivi: rukovalac poruka mogu pozvati kontroleri unutar SM.
- GW pozivi: rukovalac poruka mogu pozvati aplikacije koje se nalaze na GW zajedno sa SM.
- CLOUD pozivi: rukovalac poruka mogu pozvati udaljeni servisi.
- CLIENT pozivi: rukovalac poruka mogu pozvati klijentske aplikacije (npr mobilna aplikacija)

Takođe komunikacioni kontroler je proširen mehanizmomza identifikovanje porekla pristiglih poruka sa MB i na osnovu tih informacija se poziva kontroler sa određenim pravima. U slučaju da rukovalac poruka ne prihvata ta prava, pozivaocu će biti vraćena greška.

3.4.2 Kontroler za upravljanje periferijama

Za potrebe rukovanja periferijama GW, gde se pre svega misli na rukovanje LE diodama i tasterima, kreiran je poseban kontroler. On ima zadatak da detektuje korisničke akcije sa tastera GW i obavesti ostatak sistema o tim akcijama. Sa druge strane potrebno je da korisnika obavesti o stanju sistema adekvatnim indikacijama LE dioda.

3.4.3 Kontroler za upravljanje mrežnim podešavanjima

Kako je za normalno funkcionisanje GW potrebna mrežna konekcija, potrebno je da softver GW omogući korisniku jednostavno povezivanje nalokalnu mrežu i internet. Kontroler za upravljanje mrežnim podešavanjima, konfiguriše mrežna podešavanja operativnog sistema koristeći dostupne servise i aplikacije zavisno od stanja u kom se nalazi GW.

3.4.4 Kontroler za upravljanje sistemskim resursima

Ovaj kontroler upravlja najrazličitijim sistemskim podešavanjima, oslanja se na podešavanja operativnog sistema i izlaže ih ostatku sistema kroz uniforman interfejs. Podešavanja vremenskih zona, ponovno pokretanje GW i resetovanje sistema na fabrička podešavanja su samo neki od funkcionalnosti koje obavlja ovaj kontroler.

3.4.5 Kontroler za upravljanje radom aplikacija

Softver GW se sastoji od nekoliko nezavisnih aplikacija, kontroler za upravljanje aplikacijama pokreće aplikacije prilikom pokretanja sistema i gasi ih u trenutcima promene mrežnih podešavanja i pre početka reseta ploče.

3.4.6 Web poslužilac

U trenutku kada GW nije povezan ni na jednu mrežu, kontroler mrežnih podešavanja na GW-u "podiže" WiFi pristupnu tačku i očekuje od korisnika da se poveže na nju kako bi konfigurisao mrežna podešavanja. Korisnik ova podešavanja vrši kroz svoj web pretraživač, a komunikacija sa SM se obavlja posredstvom web servera koji je implementiran u posebnom kontroleru. Ovaj kontroler služi da prevede HTTP pozive u pozive ka kontrolerima SM (na sličan način kao komunikacioni kontroler).

4. Programsko rešenje

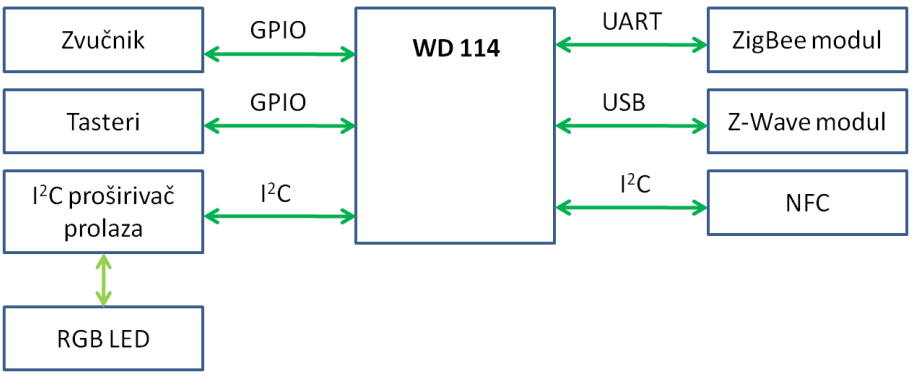
Kako je osnovna uloga SM da apstrahuje sve specifičnosti platforme, implementacija sloja za prilagođenje u velikoj meri zavisi od mogućnosti ciljne platforme.

4.1 Referentna platforma

Kao referentni primer implementacije SM koristila se platforma zasnovana na MIPS arhitekturi i distribuciji operativnog sistema Linux, zvanoj *OpenWRT* [4]. Centralni modul platforme je WD114koji sadrži QCA4531 [5] procesor, 64MB RAM memorije, 128MB flash memorije i ugrađeni WiFi modul. Pored ovog modula platforma sadrži sledeće periferije:

- ZigBee modul
- Z-Wave modul
- 2 tastera
- RGB LE diodu
- NFC
- Zvučnik

Modul je povezan sa periferijama različitim magistralama što je predstavljeno na slici 14.



Slika 14 Organizacija periferija referentne platforme

4.2 Registracija rukovalaca poruka

Kao što je opisano u prethodnom poglavlju, osnovni mehanizam interakcije sa kontrolerima je preko njihovih rukovalaca poruka. Obzirom da komunikacija između kontrolera predstavlja jezgro SM,detaljnije će se opisati proces registracije rukovalaca poruka kontrolera.

Registracija rukovalaca poruka se obavlja u procesu inicijalizacije kontrolera u metodi *setup()*. Pre registracije rukovaoca poruke, potrebno je definisati tipove njegovih ulaznih i izlaznih parametara. Ulazni parametri su klase koje nasleđuju klasu *CommandParam* i definišu proizvoljnu strukturu svojih polja i metoda koja će biti korišćena u implementaciji konkretnih kontrolera. U slučaju da je potrebno obezbediti deserijalizaciju poruke iz JSON podataka koji dolaze od drugih aplikacija, potrebno je implementirati metodu *init()*. Primer deklaracije klase ulaznih parametara za podešavanja vremenske zone prikazan je u primeru:

```
1.  class SetTimezone : public CommandParam{
2.  public:
3.      void init(Poco::Dynamic::Var& paramethers);
4.      string getLocation() const;
5.  private:
6.      string m_Location;
7.  };
```

Implementacija deserijalizacije se svodi na korišćenje metoda i klasa za parsiranje JSON-a i konverziju tipova. U slučaju da dobijena struktura nije odgovarajuća potrebno je baciti izuzetak sa opisom greške, a data greška će biti vraćena aplikaciji pozivaoca.

```
8.  void SetTimezone::init(Dynamic::Var& paramethers){
9.      JSON::Object::Ptr obj=paramethers.extract<JSON::Object::Ptr>();
10.     m_Location=obj->get("location").toString();
11. }
```

Na sličan način je potrebno kreirati i objekat izlaznih parametara.Izlazni parametri su klase koje nasleđuju klasu*ResponseParam*i imaju konstruktor bez parametara.Ostatak strukture može biti proizvoljan. U slučaju da se izlazni parametri žele serijalizovati u JSON strukturu kao odgovordrugoj aplikaciji, potrebno je implementirati metodu *toDynamicStruct*. Izgled deklaracije klase izlaznih parametara prikazana je u sledećem primeru:

```
1.  class TimezoneInformation:public ResponseParam{
2.  public:
3.      void setTimeZone(const string& location, const string& tzString);

4.      Poco::DynamicStruct::Ptr toDynamicStruct();
5.  private:
6.      string m_Location;
7.      string m_TZString;
8.  };
```

Implementacija serijalizacije podataka u JSON svode se na kreiranje novog objekta *DynamicStruct* i popunjavanjem njegovih polja koja odgovaraju stanju objekta.

```
1. Poco::DynamicStruct::Ptr TimezoneInformation::toDynamicStruct(){
2.     Poco::DynamicStruct::Ptr response=new Poco::DynamicStruct();
3.     response->insert("location", m_Location);
4.     response->insert("TZ_string", m_TZString);
5.     return response;
6. }
```

Kada su se definisali tipovi ulaznih i izlaznih parametara potrebno je kreirati metodu objekta koja će biti pozvana kao rukovalac poruka, tj u kojoj će se vršiti obrada podataka. Prototip metode ima dva parametra koji odgovaraju tipovima ulaznih i izlaznih parametara.

```
1. void setTimezone(const SetTimezone& params, TimezoneInformation& response)
```

Registracija samog rukovaoca poruka se sastoji od dva koraka.

- Kreiranja delegata pomenute metode
- Definisanje simboličkog imena i jedinstvenog identifikatora rukovaoca poruke i povezivanje sa kreiranim delegatom.

Sledeći primer prikazuje kako izgleda ovaj proces. Tipovi ulaznih i izlaznih parametara se definišu kao parametri šablon klase *CommandCallback* (linija 2), kreiranje delegata zahteva informacije o nazivu metode i konkretnom objektu nad kojim metodu treba pozvati (linija 3).

```
1. void SystemSettingsController::setup(map<std::string, core::Setting> &settings){
2.     CommandCallback<SetTimezone, TimezoneInformation, SystemSettingsController>
3.     setTimezoneCB(*this, &SystemSettingsController::setTimezone);
4.
5.     registerMessageHandler("setTimezone", SET_TIMEZONE, setTimezoneCB);
6. }
```

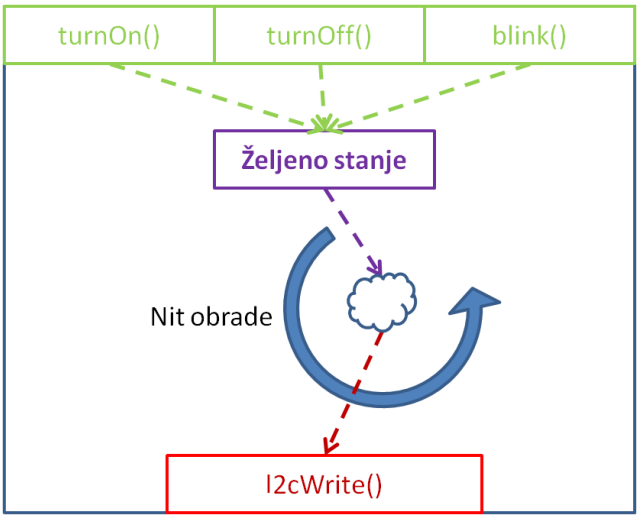
4.3 Upravljanje LED indikacijom

Referentna platforma ima jednu LE diodu preko koje je potrebno obavestiti korisnika o stanjima GW različitim svetlosnim indikacijama. LE dioda ima 3 osnovne boje i moguće ih je posebno kontrolisati, a njihovom međusobnom kombinacijom moguće je prikazati 7 različitih boja na diodi. LE dioda je povezana na modul preko I^2C proširivača prolaza i sa softverske strane upravljanje diodom se obavlja komunikacijom sa proširivačem prolaza korišćenjem I^2C protokola.

Softverska implementacija upravljanja LE diode enkapsulira implementaciju zavisnu od specifičnosti referentne platforme kao što je I^2C protokol i izlaže jednostavan interfejs zasnovan na akcijama upravljanja koji sadrži tri metode:

```
1. void turnOn(uint8_t color);
2. void turnOff();
3. void blink(uint8_t color, int period);
```

Upravljanje LE diodom zasniva se na postojanju posebne niti obrade koja će na osnovu podataka o željenom stanju diode komunicirati sa proširivačem periferija i tako uključivati i isključivati diodu. Nit obrade u pravilnim vremenskim intervalima proverava vrednosti želejnog stanja i u slučaju da je došlo do promene, šalje podatke na I^2C magistralu. U slučaju da je željeno stanje diode u režimu treptanja, ova nit će zadatom periodu uključivati i isključivati diodu. Metode koje definiše interfejs služe samo da konfigurišu željeno stanje diode na osnovu kog će se u niti obrade izvršavati potrebne akcije.



Slika 15 Mehanizam rukovanja LE diodom

Podaci kojima se definiše željeno stanje diode predstavljeni su u Tabela 3 Opis vrednost kojima se definiše željeno stanje diode.

Naziv polja	**Opis**
Stanje	Stanje u kom treba da se nalazi dioda. Može biti jedno od sledećih: • UKLJUČENO • ISKLJUČENO • TREPTANJE
Boja	Definiše boju kojom dioda treba da svetli. Definiše vrednost jednog bajta u kom se tri najniža bita odnose na stanja 3 osnovne boje.
Period	U slučaju da je stanje podešeno na "TREPTANJE" period koji se definiše ovim poljem predstavlja vreme na koje će dioda menjati stanje upaljena – ugašena.

Tabela 3 Opis vrednost kojima se definiše željeno stanje diode

4.4 Rukovanje tasterima

Kako stanje tastera može uzeti jednu od dve vrednosti: pritisnut i otpušten, a broj akcija koje korisnik treba preneti GW je dosta veći, potrebno je uvesti određeni mehanizam kojim će se na osnovu načina na koji korisnik pritiska taster detektovati željena akcija. Definisana su dva načina pritiska tastera:

1. Kratak pritisak (eng. *Short press*): definiše se kao akcija pritiskanja i otpuštanja tastera u kojoj je taster bio pritisnut ne više od 500ms.
2. Dug pritisak (eng. *Long press*): definiše se kao akcija pritiskanja iotpuštanja tastera u kojoj je taster bio pritisnut više od 500ms.

Da bi uopšte bilo moguće detektovati promene stanja tastera unutar aplikacije, potrebna je podrška od strane operativnog sistema i drajvera koji će omogućiti korisničkim aplikacijama da dobiju informaciju o trenutnom stanju tastera. Generalno postoje 2 načina na koji aplikacija može detektovati promene stanja taster:

- Prozivanjem (eng. *Polling*): aplikacija u vremenskim intervalima proverava trenutno stanje tastera i u slučaju da se ono promenilo u odnosu na prethodni period prozivanja, tretira se kao akcija pritiska ili otpuštanja tastera. Ovaj mehanizam se smatra lošim jer se procesorsko vreme troši za detekciju stanja tastera, međutim u nekim slučajevima je to jedini način kojim se može detektovati promena.
- Obaveštenjem o promeni stanja: U nekim slučajevima operativni sistem (drajver) dobija obaveštenja o promeni stanja periferija mehanizmom prekida i te promene stanja proizvoljnim mehanizmom prosleđuje do korisničkih aplikacija u formi obaveštenja. Treba napomenuti da je očekivano ponašanje drajvera da odradi *debouncing* dobijenog signala i tek tada obavesti korisničke aplikacije o promeni, mada to ne mora biti uvek slučaj.

Operativni sistem referentne platforme *OpenWRT* poseduje podsistem kernela nazvan "*Hotplug*". On omogućava da se pokrenu proizvoljne skripte zavisno od događaja kao što su dodavanje ili izbacivanje hardvera, pritisci tastera i slično [6]. *Hotplug* takođe obaveštava korisničke aplikacije o nastalim događajima preko interfejsa za interprocesnu komunikaciju *Netlink* [7]. Poruke koje se dobijaju na ovakav način tekstualnog su formata, primer poruke koja nosi informaciju o pritisnutom tasteru:

```
1. ACTION="pressed"
2. BUTTON="RESET_BUTTON"
```

Parsiranjem podataka koji se ovim putem dobijaju detektuju se pritisci tastera bez potrebe da se vrši prozivanje uređaja i time se štedi procesorsko vreme.

Softverska implementacija mehanizma za detekciju akcija tastera organizovana je na način koji omogućava njeno jednostavno prilagođenje različitim platformama, broju fizičkih tastera i akcijama koje oni detektuju. Sastoji se od sledećih komponenti:

4.4.1 Apstraktna implementacija tastera

U apstraktnoj klasi *Button* nalazi se implementacija algoritma za detekciju dugih i kratkih pritisaka tastera na osnovu vremena koja su prošla od pritiska i puštanja tastera. U ovoj klasi se ne definiše konkretna akcija na određeni broj klikova, već se ta implementacija delegira klasama naslednicama. Klasa *Button* definiše sledeći interfejs:

```
1. void pressed();
2. void released();
3. void ping();
```

Metode *pressed()* i *released()* se pozivaju kada se određeni taster pritisne ili otpusti, što se može zaključiti iz naziva. Metoda *ping()* sa druge strane nema direktne veze sa stanjima tastera, ona služi da bi se tasteru dao "osećaj" prolaska vremena, kako bi on u određenim trenucima vremena mogao da učini akcije. Metoda *ping()* se poziva periodično i trebalo bi da omogući klasi *Button* da proveri u kom stanju se nalazi i da li je potrebno izvršiti neku akciju. Zahtev da posle 2 sekunde držanja tastera treba izvršiti određenu akciju može se smatrati dobrim primerom.

Klasa *Button* na osnovu poziva ove tri metode određuje da li se radi o kratkom ili dugom pritisku tastera. Određivanje stanja u kom se taster nalazi vrši se merenjem proteklog vremena od pritiska ili puštanja tastera i brojanja ukupnog broja pritisaka ili dužine držanja tastera. Taster se može nalaziti u jednom od 3 stanja: IDLE, SHORT_PRESS i LONG_PRESS. Prelasci između stanja tastera i akcijama u tim trenucima mogu se predstaviti sledećom tabelom.

Stanje / **Akcija**	**IDLE**	**SHORT_PRESS**	**LONG_PRESS**
`pressed()`	*SHORT_PRESS*, Broj klikova +1	*SHORT_PRESS*, Broj klikova +1	*LONG_PRESS*, Broj klikova = 0
`released()`	*IDLE*, Broj klikova = 0	*SHORT_PRESS*	*IDLE*, Detektovan dug pritisak
`ping()` **(proteklo vreme > 500ms)**	*IDLE*	Taster pritisnut: *LONG_PRESS* Taster otpušten: Detekt. kratak pritisak *IDLE*	*LONG_PRESS*

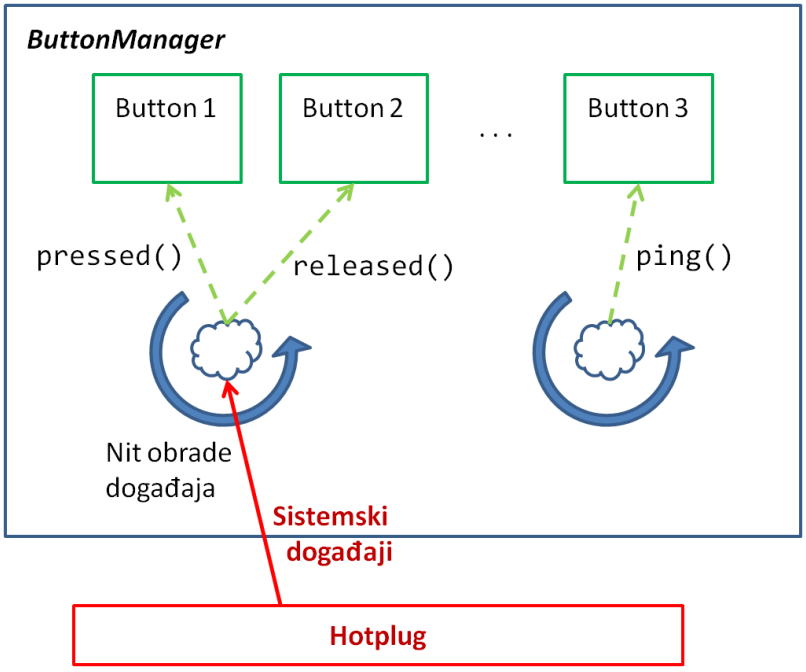
Tabela 4 Prelasci između stanja kod rukovanja tasterima

U trenutku detektovanja dugog ili kratkog pritiska poziva se metoda klase naslednice *Button* klase koja treba da na osnovu broja klikova ili dužine dugog pritiska odluči koju akciju treba izvršiti. Komunikacija između *Button* klase i konkretnih naslednika vrši se pozivom apstraktnih metoda u klasi *Button* koje se moraju implementirati u nasleđenim klasama. Apstraktne metode imaju sledeći prototip:

```
1. void longPressFinished(unsigned holdInterval);
2. void longPressUpdate(unsigned holdInterval);
3. void shortPressFinished(int count);
```

4.4.2 Rukovalac tastera

Kako bi klasa *Button* mogla na pravilan način funkcionisati potrebno je pozivati njene metode interfejsa u pravim trenucima. Klasa *ButtonManager* ima tu ulogu i zamišljena je kao jedinstvena klasa koja obrađuje sve tastere na sistemu. Na slici 1 prikazan je šematski izgled gradivnih blokova *ButtonManager* klase. Ona ima dve niti za obradu podataka, prva služi da se u njoj parsiraju pristigli podaci preko *Netlink* interfejsa i na osnovu parsiranih podataka pozivaju metode odgovarajućeg tastera. Druga nit u pravilnim vremenskim intervalima poziva metodu *ping* svakog tastera.



Slika 16 Princip rada mehanizma za detekciju pritiska tastera

Rukovalac tastera može imati proizvoljan broj *Button* objekata kojima rukuje, jedino je potrebno da se prilikom registracije tastera navede tačan identifikator kojim je taster registrovan u operativnom sistemu, jer se preko tog naziva vrši povezivanje sa konkretnim objektima.

4.4.3 Konkretna implementacija tastera

Kao što je napomenuto implementacija logike konkretnog tastera se svodi na implementaciju apstraktnih funkcija *Button* klase. Sledeći primer prikazuje kako izgleda implementacija odlučivanja ponašanja GW na akcije kratkog klika. U metodi je potrebno samo odlučiti kako odreagovati na detektovani broj klikova (izvršiti akciju).

```
1. void ResetButton::shortPressFinished(int count){
2.     switch(count){
3.     case 3:
4.         m_EventHandler->resetNetworkParamsEvent();
5.         break;
6.     case 6:
7.         m_EventHandler->clearUserDataEvent();
8.         break;
9.     case 9:
10.        m_EventHandler->factoryResetEvent();
11.        break;
12.    default:
13.        break;
14.    }
15.}
```

4.5 Upravljanje sistemskim i mrežnim podešavanjima

Kao što je ranije već pomenuto, SM ima zadatak da izvrši konfigurisanja operativnog sistema na kom se izvršava. Ovo zahteva modifikaciju sistemskih podešavanja ili pokretanje različitih sistemskih pozadinskih procesa.

Operativni sistem referentne platforme obezbeđuje centralizovani uniforman mehanizam za pristup konfiguracije celokupnog sistema. Ovaj mehanizam je nazvan *UCI* (eng. *Unified Configuration Interface*) – objedinjeni interfejs za konfiguraciju [8]. UCI obezbeđuje pristup sistemskim podešavanjima na nekoliko načina:

- Modifikacijom konfiguracionih fajlova
- Korišćenjem CLI aplikacije
- Korišćenjem *libuci* C sistemske biblioteke
- Korišćenjem *libuci-lua* proširenja za Lua programski jezik

Za potrebe SM iskorišćena je *libuci*C biblioteka preko koje se vrše sva potrebna podešavanja sistema. Radi lakše upotrebe, kreirana je klasa omotač preko koje se pristupa *libuci* biblioteci i ona ima interfejs koji omogućava pristup podešavanjima na nivou ključ – vrednost. Ključ je putanja – opis konkretnog podešavanja koje definiše UCI i lako se može pronaći u dokumentaciji koju obezbeđuje *OpenWRT*. Primer rukovaoca poruka koji dobavlja podešavanja trenutne WiFi konekcije korišćenjem UCI mehanizma prikazan je u sledećem isečku:

```
1. void getWifiSettings(const messages::EmptyParam &params,
2.                              WifiSettings &response){
3.     osm::utils::UciWrapper uci;
4.     string ssid=uci.getValue("wireless.@wifi-iface[0].ssid");
5.     string encryption=uci.getValue("wireless.@wifi-iface[0].encryption");
6.     string mode=uci.getValue("wireless.@wifi-iface[0].mode");
7.     response.setWifiSettings(ssid,encryption,mode);
8. }
```

Na sličan način se može pristupiti i ostalim podešavanjima sistema kao što su podešavanja sistemskog vremena, mrežnih podešavanja i slično.

5. Rezultati i testiranje

Konstantne izmene na softveru su sastavni deo njegovog razvoja, kako se zahtevi menjaju implementacija softverskog rešenja mora da se prilagođava novonastalim situacijama. Iako je cilj ovih modifikacija zapravo stabilnije softversko rešenje, bolje prilagođeno zahtevima tržišta, proces izmena i dodavanja novih funkcionalnosti je čest uzrok nastanka grešaka i otkaza u softveru. Da bi se sprečilo da softver koji ima grešku dospe na uređaj korisnika, potrebno je definisati detaljan plan i alate testiranja koji će sistematično proveriti ispravnost celokupnog sistema i sprečiti nenamerne greške da dospeju u produkciju.

5.1 Testovi softvera za rukovanje sistemskim resursima

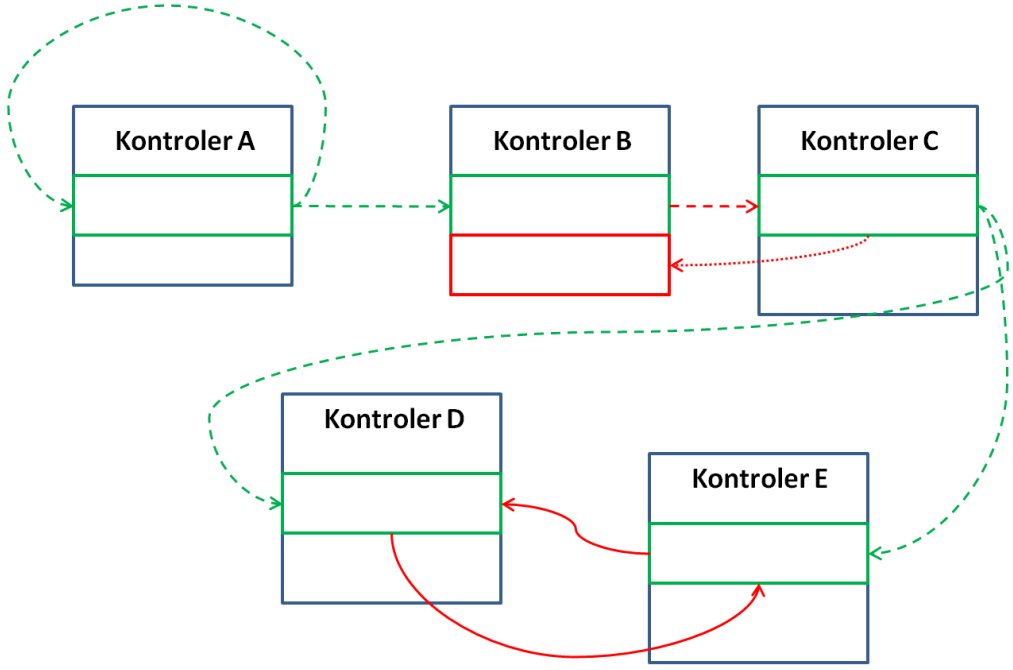
Da bi se ispitala ispravnost SM napravljene su dve grupe testova koje bi trebalo da daju kompletnu sliku o radu sistema i trebali bi se pokretati i ažurirati nakon svake nove dodate funkcionalnosti.

5.1.1 Testovi opterećenja jezgra SM

Ovaj skup testova je namenjen da proveri ispravno funkcionisanje mehanizama jezgra koje obezbeđuje SM. Ovo podrazumeva ispravno instanciranje kontrolera, njihovu međusobnu komunikaciju u graničnim slučajevima i ispravno čuvanje podataka na masovnoj memoriji. Ovi testovi se izvršavaju na način da maksimalno opterete SM kako bi se manifestovale eventualne greške sinhronizacije. Ova grupa testova ne uključuje funkcionalnosti specifičnih platformi već isključivo platformski nezavisnog jezgra sistema, stoga se pokreću na radnoj stanici i proverava se nepravilno oslobađanje memorije tokom njenog rada.

Za potrebe ovih testova SM se prevodi sa posebnim skupom kontrolera koji služe isključivo za testiranje. U trenutnoj postavci nalazi se 5 različitih vrsta kontrolera koji međusobnom komunikacijom pokušavaju da izazovu greške u graničnim slučajevima pri

međusobnoj komunikaciji. Slika 17 prikazuje kontrolere koji učestvuju u testiranju i njihov međusobni odnos u razmeni porukama.



Slika 17 Razmena poruka u kontrolerima za testiranje opterećenosti jezgra SM

Zeleni pravougaonici unutar kontrolera predstavljaju jednu metodu rukovalaca porukama, dok usmerene linije predstavljaju putanju poruke. Zelene isprekidane linije predstavljaju asinhrone pozive metoda rukovalaca porukama koji ne očekuju odgovor (nemaju registrovane metode za obradu odgovora). Crvene linije predstavljaju pozive rukovalaca porukama koji očekuju odgovor, u slučaju da je crvena linija isprekidana odgovor se dobija asinhrono, dok puna crvena linija između kontrolera predstavlja poziv sinhronih metoda. Test počinje tako što kontroler A proizvoljan broj puta šalje poruke samom sebi i kontroleru B. Kontroler B na pristizanje svake poruke šaljenovu poruku kontroleru C i proverava dobijeni odgovor asinhrono. Kontroler C prilikom obrade svake komande šalje poruke kontrolerima D i E koji se prilikom obrade sinhrono međusobno pozivaju.

Ovim testom se proverava da li u mehanizmu komunikacije koji je osnova funkcionisanja SM postoji greška. Ovim testovima se pokrivaju standardni načini komunikacije kao i granični slučajevi pomenuti u poglavlju 3.3.3.

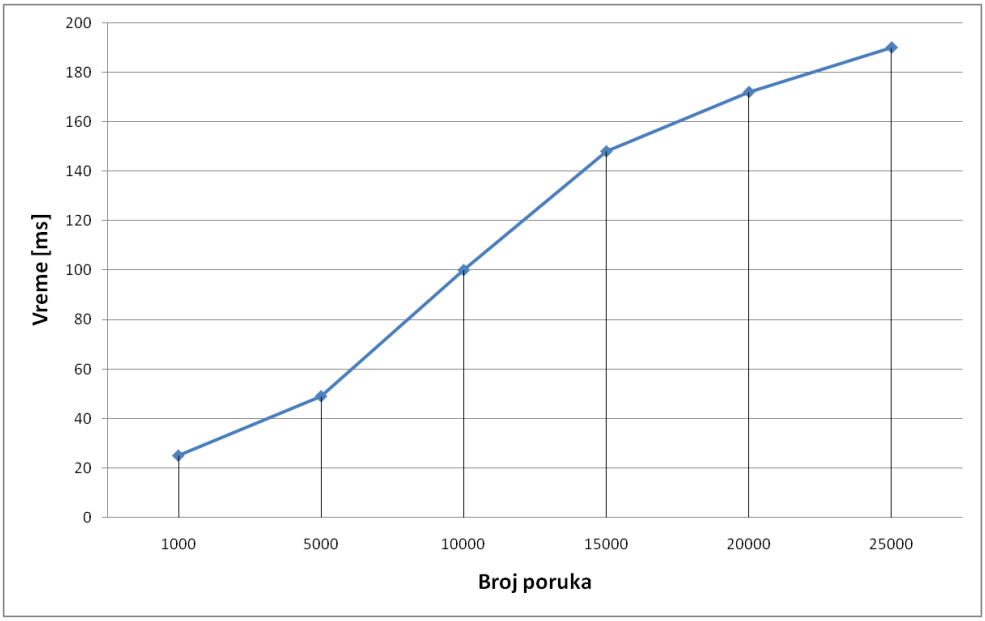
Za proveru nepravilnog oslobađanja memorije koristi se alat *Valgrind* [9] koji bi trebalo da na kraju izvršenih testova pokaže da je sva memorija uspešno oslobođena. Ovi testovi bi trebali da se pokreću posle svake modifikacije jezgra SM.

5.1.2 Rezultati testova opterećenja SM

Potvrdu da jezgro SM nema nepravilnog oslobađanja memorije dobija se kao rezultat pokretanja aplikacije *Valgrind*. U njoj se može videti da aplikacija nema direktno i indikretno izgubljenih blokova. Vrednosti za *possibly lost*i *still reachable* prikazuju određene vrednosti, međutim analizom je ustanovljeno da ti blokovi potiču iz biblioteka koje se koriste i da su oni konstantni, tj ne zavise od broja poruka koje se razmenjuju i vremena izvršenja aplikacije tako da se oni mogu zanemariti.

```
...
==11709== LEAK SUMMARY:
==11709==    definitely lost: 0 bytes in 0 blocks
==11709==    indirectly lost: 0 bytes in 0 blocks
==11709==      possibly lost: 288 bytes in 1 blocks
==11709==    still reachable: 72,704 bytes in 1 blocks
==11709==         suppressed: 0 bytes in 0 blocks
==11709==
==11709== For counts of detected and suppressed errors, rerun with: -v
==11709== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Takođe uzelo se u obzir i vreme potrebno za obradu poruka, u zavisnosti od broja poruka. Sledeći grafik prikazuje tu zavisnost i na njemu se može primetiti da vreme obrade poruka linearno zavisi od broja poruka, što je bila i očekivana kompleksnost.

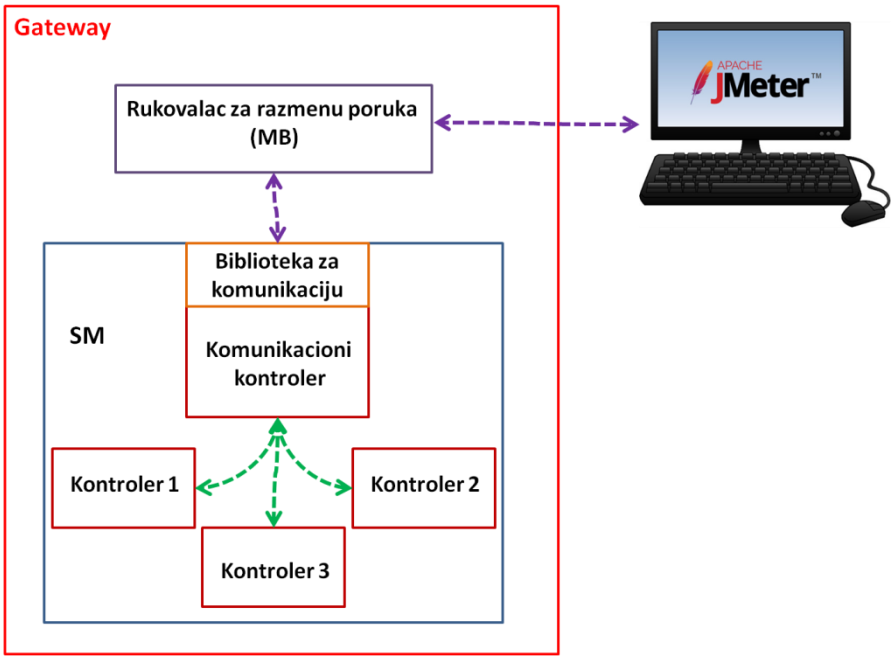


Slika 18 Dužina obrade u zavisnosti od broja poruka

5.1.3 Funkcionalni testovi

Ovom grupom testova se želi proveriti ispravnost funkcionalnosti koju SM treba da obezbedi na ciljnoj platformi, ovi testovi se koriste kako bi se proverilo da li je implementirana

funkcionalnosti dobro napisana. Za izvršenje ovih testova koristi se *JMeter* [10] aplikacija namenjena za funkcionalno testiranje, ona se izvršava na host mašini i komunicira sa platformom preko protokola za razmenu poruka (slika 19). Testiranje se odvija tako što *JMeter* poziva rukovaoce poruka SM i na osnovu sadržaja odgovora i događajima koje SM šalje u određenim vremenskim trenucima zaključuje da li je ponašanje adekvatno.



Slika 19 Izgled sistema za funkcionalno testiranje SM

Testiranje različitih funkcionalnosti grupisano je u posebne testne slučajeve u kojima se definiše tačna sekvenca poruka koje se trebaju poslati SM i sekvenca odgovora koji se trebaju dobiti, sledeće tabele opisuju neke od testnih slučajeva.

Tip poruke	**Šalje**	**Telo poruke**

Zahtev	JMeter	{
		"service":"systemController",
		"command":"setTimezone",
		"params":{
		"location":"Europe/Belgrade"
		}
		}

Odgovor	SM	{
		"status":true
		}

Zahtev	JMeter	{
		"service":"systemController",
		"command":"getTimezone",
		"params":{}
		}

Tip poruke	Šalje	Telo poruke

Odgovor	SM	{
		"location":" Europe/Belgrade ",
		"TZ_string":"CET-1CEST,M3.5.0",
		}

Tabela 5 Testni slučaj: provera ispravnosti podešavanja vremenske zone

Tip poruke	Šalje	Telo poruke

Zahtev	JMeter	{
		"service":"systemController",
		"command":"reboot",
		"params":{}
		}

Odgovor	SM	{
		"status":true
		}

Obaveštenje	SM	{
		"systemStatus":"online"
		}

Tabela 6 Testni slučaj: provera akcije ponovnog pokretanja uređaja

Treba napomenuti da se ovim testovima ne mogu pokriti akcije promene podešavanja mreže jer se u tim slučajevima gubi konekcija sa *JMeter* aplikacijom. Takođe nemoguće je na ovaj način proveriti da li su se adekvatne fizičke promene desile na GW (kao što je paljenje LE diode).

5.1.4 Rezultati funkcionalnih testova

Na sličan način kao u prethodnom poglavlju definisan je određeni skup testova kojima se trebalo potvrditi ispravno funkcionisanje platforme. Sledeća tabela prikazuje rezultate funkcionalnih testova:

Broj testa	Zadatak testa	Rezultat testa
1	Proveriti da li se vremenske zone uspešno menjaju na platformi	Zadovoljava
2	Proveriti da li se SM uspešno resetuje platformu na fabrička podešavanja	Zadovoljava
3	Proveriti da li SM uspešno pokreće i isključuje HTTP web server	Zadovoljava

Broj testa	Zadatak testa	Rezultat testa
4	Proveriti da li SM uspešno dobavlja vrednosti serijskog broja sa platforme	Zadovoljava
5	Proveriti da li SM uspešno dobavlja informacije o trenutnim mrežnim podešavanjima	Zadovoljava

Tabela 7 Skup testova za proveru ispravnosti rada SM

Ovim skupom testova potvrdilase ispravnost jednog dela funkcionalnosti. Za automatizovanu proveru komplente funkcionalnosti SM, potrebno je postojanje alata za testiranje koji komunikaciju sa SM neće izvršavati preko lokalne mreže. Testovi se ne mogu u potpunosti automatizovati jer će uvek postojati određeni elementi koji su vezani za fizičke karakteristike platforme (npr. tasteri i LE diode) čiji ispravan rad se ne može proveriti isključivo softverskom implementacijom testa.

6. Zaključak

Dizajnom i implementacijom programske podrške za apstrakciju i upravljanje sistemskim resursima rešio se problem čvrste povezanosti softverske implementacije centralnog kontrolera i njegovih hardverskih karakteristika.Razvijena aplikacija definiše jedinstveno ponašanje GW koje se oslanja na pozive apstraktnih metoda PAL interfejsa. Prenošenje softvera na nove platforme svodi se na implementaciji PAL sloja za konkretnu platformu sa specifičnostima koje ona donosi.

Radno okruženje koje obezbeđuje SM, kroz skup apstraktnih klasa, enkapsulira mehanizme razmene poruka između svojih modula i omogućava jednostavnu interprocesnu komunikaciju sa ostalim aplikacijama sistema. Prilikom dizajna mehanizama za razmenu poruka vodilo se računa kako o jednostavnosti njihovog korišćenja, tako i o sprečavanju nastanka grešaka u specifičnim slučajevima korišćenja. Obezbeđen je i skup pomoćnih funkcionalnosti od strane jezgra SM kao što je mehanizam za čuvanje podataka na masovnoj memoriji.

Ispravno funkcionisanje razvijenog softvera je potvrđena implementacijom prilagođenja za referentnu platformu. Kreirani su i testovi kojima se potvrđuje ispravnost rada platforme. Vreme implementacije i količina programskog koda potrebno za implementaciju nove platforme je višestruko smanjeno, što je bio i osnovni zadatak koji je ova platforma trebala da zadovolji.

Dalji pravac razvoja ovakve platforme trebao bi biti usmeren u kreiranju boljih automatizovanih alata za funkcionalno testiranje. Alat kao što je *JMeter* ima bitno ograničenje u tome što je potrebna mrežna konekcija sa uređajem koji se testira, a funkcionalni testovi SM uključuju promene mreže i ponovno pokretanje uređaja u kojima se mreža menja, što se alatom kao što je *JMeter* ne može ispratiti. Automatizovani testovi bi takođe trebali da omoguće mehanizam kojim bi se proverila ispravna komunikacija sa periferijama kao što su LE diode i tasteri.

7. Literatura

[1] I. Lazarević, M. Pandurov, B. Radin, I. Papp, *Advanced scene mechanism in home automation system,* IcETRAN konferencija 2016

[2] M. Pandurov, B. Petelj, R. Pavlović, N. Teslić, *Platform for extending home automation gateway's functionality with plugin mechanism,* Consumer Electronics - Berlin (ICCE-Berlin), 6-9 septembar 2015 IEEE 5th International Conference, str. 354-357

[3] M. Pandurov, I. Lazarević, R. Pavlović, N. Smiljković, *Unified device access in home automation environment,* Telecommunications Forum Telfor (TELFOR), 25-27 novembar 2014, str. 971-974

[4] OpenWrt, embedded operating system, https://openwrt.org/ jul 2016

[5] QCA4531 - A low-power Linux connectivity hub for the Internet of Everything, https://www.qualcomm.com/documents/low-power-wi-fi-qca4531, jul 2016

[6] Hotplug, https://wiki.openwrt.org/doc/techref/hotplug jul 2016

[7] Netlink, http://man7.org/linux/man-pages/man7/netlink.7.html jul 2016

[8] OpenWrt Unified Configuration Interface (UCI), https://wiki.openwrt.org/doc/uci jul 2016

[9] Valgrind - instrumentation framework for building dynamic analysis tools, http://valgrind.org/ jul 2016

[10] Jmeter - application designed to load test functional behavior and measure performance, http://jmeter.apache.org/ jul 2016