Matematički fakultet

Univerzitet u Beogradu

Pristup lokalnom operativnom sistemu i njegova kontrola iz veb aplikacija

Mentor:

Vladimir Filipović

Student:

Nikola Živković

U Beogradu, 2015. godina

SADRŽAJ

PREDGOVOR

Dobro je poznato kojom brzinom se danas razvijaju tehnologije - računari su svuda i svet se više ne može zamisliti bez informacionih tehnologija, a moderan čovek retko provede dan bez interakcije sa nekim od pametnih uređaja.

Danas, većina modernih uređaja, od frižidera, šporeta i televizora, preko automobila i aviona, pa sve do robota, u sebi sadrži pametne elemente koji predstavljaju računare u malom i koji podižu nivo mogućnosti za kontrolu ovih uređaja na sasvim drugi nivo. Ovi pametni elementi se nazivaju **ugrađeni sistemi**.

Svaki ugrađeni sistem mora u sebi imati i softver koji će njime upravljati. Kako bi se taj softver što lakše i efikasnije razvijao, potrebna su specijalizovana razvojna okruženja. Ovakva razvojna okruženja se najčešće pokreću na lokalnim računarima, ali kako u poslednje vreme postoji tendencija da se sav softver izvršava na internetu, ovakva razvojna okruženja se mogu naći i u obliku veb aplikacija koje se pokreću u oblaku. Jedno takvo rešenje predstavlja razvojno okruženje pod imenom OBLAC.

Ovakav pristup osim što nudi određen broj prednosti, dolazi i sa nekim problemima koje je neophodno rešiti. Najveći od tih problema je da ovakva razvojna okruženja moraju imati neki način na koji bi napisani softver mogla da sačuvaju na ugrađeni sistem. To znači da je potrebno omogućiti da veb aplikacija koja se izvršava u internet pregledaču na datom računaru ima pristup ugrađenom sistemu koji je fizički priključen na taj računar.

Ovaj rad će se baviti jednim od mogućih rešenja za ovaj problem. Konkretno, razmatraće se arhitektura, dizajn i implementacija aplikacije koja omogućava korisnicima razvojne platforme OBLAC da brzo i jednostavno isprobaju napisani softver na konkretnim ugrađenim sistemima. Aplikacija čija se funkcionalnost i dizajn opisuje je dobila naziv „*SOMANETconnect"* i nju je u potpunosti razvio autor master rada. Aplikacija je namenjena radu na personalnim računarima, razvijena je u Java programskom jeziku i podržava većinu modernih operativnih sistema. Izvorni kod „*SOMANETconnect*" aplikacije se može pronaći na internet adresi *https://github.com/zivke/SOMANETconnect/tree/master-rad*.

Značaj primera koji će biti prikazan u ovom radu je u tome što je on veoma fleksibilan, tako da je moguće jednostavno modifikovati ga i iskoristiti kako bi svaki aspekat operativnog sistem bio dostupan iz specijalizovane veb aplikacije.

UVOD

Internet stvari

Internet stvari (engl. *Internet of things*) predstavlja ideju da se sve „stvari" opreme elektronikom, softverom, senzorima i načinom komunikacije, a zatim povežu u mrežu koja bi im omogućila da razmenjuju podatke sa proizvođačima, korisnicima ili drugim uređajima koji su takođe povezani na ovu mrežu. Na ovaj način se kontrola fizičkog sveta vrši automatski, uz pomoć računarskih sistema i preko već postojeće komunikacione infrastrukture, što znatno povećava sigurnost, efikasnost i preciznost svih uređaja. [1]

Internet stvari bi imao pregršt mogućih primena, sve u cilju poboljšanja kvaliteta života ljudi i za dobrobit čitave planete uopšte. Internet stvari bi se mogao koristiti za pomoć starijim licima i njihovo nadgledanje i praćenje, zatim u službi što sigurnijih autonomnih automobila, preciznog praćenja nivoa zagađenosti, automatske industrije i u mnogim drugim situacijama.

Osnovna komponenta ovakvog sistema su **ugrađeni sistemi**.

Ugrađeni sistemi

Ugrađeni sistemi predstavljaju računarske sisteme koji mogu imati razne uloge unutar većeg mehaničkog ili elektronskog sistema i najčešće se koriste za njihovu kontrolu [2].

Ugrađeni sistemi se nalaze u većini modernih uređaja. Oni su široko rasprostranjeni i često korišćeni zbog toga što su obično izuzetno mali, imaju malu potrošnju električne energije i troškovi njihove proizvodnje su niski pošto se potpuno mogu spakovati u jedno integrisano kolo.

Osnovni gradivni blok ugrađenih sistema je **mikrokontroler**. Mikrokontroler predstavlja „računar u malom", odnosno celokupan računar sadržan u integrisanom kolu, koje poseduje sopstveni mikroprocesor, memoriju i digitalne i analogne ulaze i izlaze koji se koriste za komunikaciju sa spoljnim svetom [2].

Funkcionisanje ugrađenih sistema u velikom delu zavisi od softvera koji je u njih učitan. Ovakav softver se vrti u beskonačnoj petlji i vrši izračunavanja na osnovu informacija koje dobije iz spoljašnjeg sveta putem senzora, a zatim u skladu sa dobijenim rezultatima šalje izlazne signale kojima i vrši upravljanje u realnom vremenu.

Razvojna okruženja za rad sa ugrađenim sistemima

S obzirom da softver predstavlja jedan od neizostavnih delova svakog ugrađenog sistema, javlja se potreba za razvojnim okruženjima koja mogu zadovoljiti sve potrebe razvoja ovakvog softvera.

Razvojna okruženja za ovu vrstu softvera moraju imati iste osnovne komponente kao i standardna razvojna okruženja - alate za pisanje, prevođenje i povezivanje koda. Ono što se ipak razlikuje kod razvojnih okruženja za ugrađene sisteme su specifični prevodioci koji zavise od vrste ugrađenog sistema za koji se piše kod [3], kao i njihova mogućnost da ovakav

softver snime i pokrenu na samom ugrađenom sistemu. Osim toga, neka razvojna okruženja čak nude i mogućnost traženja grešaka dok se softver izvršava na samom ugrađenom sistemu.

Većina razvojnih okruženja namenjenih razvoju softvera za ugrađene sisteme se pokreće na lokalnim računarima, ali ovakav način upotrebe sa sobom donosi i određene probleme. Ovakva razvojna okruženja su često veoma komplikovana za podešavanje, zavise od operativnog sistema, vezana su za jedan računar i imaju nizak stepen bezbednosti podataka u slučaju krađe ili kvara računara. Kako bi se ovi i slični problemi umanjili ili u potpunosti rešili, u poslednje vreme postoji tendencija prebacivanja razvojnih okruženja na internet, u vidu veb aplikacija koja se izvršavaju u oblaku. Razvojnih okruženja ovog tipa ima veoma malo jer je ideja još uvek mlada, a neka od njih su:

1. *mbed (http://www.mbed.com)*
2. *Arrow Cloud Connect (https://design.arrow.com/arrowcloudconnect)*
3. *Cloud Composer Studio (https://dev.ti.com)*
4. *OBLAC (https://www.synapticon.com/products/oblac-tools)*

Ova razvojna okruženja su veoma slična po načinu funkcionisanja (iako se koriste za razvoj različitih tipova ugrađenih sistema), tako da će u ovom radu ukratko biti predstavljen samo OBLAC, s obzirom da je *SOMANETconnect* aplikacija koja je direktno nastala iz potreba koje su postale očigledne tokom razvoja OBLAC-a.

OBLAC

OBLAC predstavlja objedinjeni sistem za modelovanje ugrađenih sistema, pisanje softvera za njih i njegovo snimanje u željene ugrađene sisteme koji se izvršava u oblaku [4].

Svaki korisnik koji se prijavi da koristi OBLAC odmah dobija vlastitu virtualnu mašinu koja se izvršava u oblaku i koja na sebi pokreće Linux operativni sistem sa Tomcat serverom i OBLAC aplikacijom. Kada korisnik ode na početnu stranu OBLAC-a, vrši se automatsko preusmeravanje saobraćaja ka već pomenutoj virtualnoj mašini i korisnik je spreman da počne sa radom.

OBLAC je po sistemu rada i izgledu veoma sličan drugim razvojnim okruženjima, osim činjenice da se izvršava u oblaku i da korisnicima nudi mogućnost da sami (vizuelno) modeluju hardverski deo ugrađenih sistema u skladu sa svojim potrebama. Korisnicima su na raspolaganju mikroprocesorski, komunikacioni, kontrolni i mnogi drugi moduli. Svi oni se povezuju u jednu ili više celina, a OBLAC u svakom trenutku automatski ili uz malu intervenciju korisnika, generiše osnovni programski kod koji je neophodan kako bi modelovani ugrađeni sistem mogao da funkcioniše. Kada je korisnik zadovoljan svojim sistemom, on pristupa izmeni koda po sopstvenoj želji direktno unutar OBLAC-a. Ovaj kod je zatim moguće prevesti takođe u OBLAC-u zahvaljujući činjenici da se svaki korisnik raspolaže ličnom Linux virtualnom mašinom u kojoj su unapred instalirani svi potrebni alati. Ovako prevedeni kod se odmah može snimiti ili pokrenuti na mikrokontroleru koji mora biti povezan preko USB porta na korisnikov računar. Za ovu funkcionalnost je presudan upravo SOMANETconnect koji povezuje OBLAC kao veb aplikaciju i sam računar na kome se ona izvršava, kako bi OBLAC-u omogućio direktan pristup svim resursima operativnog sistema.

1 PROBLEM

U trenutku kada je OBLAC dobio svoju prvu verziju koja je puštena u javnost, pojavio se jedan očigledan nedostatak. Korisnici su mogli da ga koriste kako bi dizajnirali svoj ugrađeni sistem i napisali softver za isti, ali nije bilo jednostavnog načina da taj softver i isprobaju pošto ga naprave. Softver za ugrađeni sistem bi u tom slučaju morao da bude isporučen i sačuvan na korisničkom računaru, posle čega bi korisnika očekivala dugačka i komplikovana procedura podešavanja specijalnog softverskog alata koji omogućava povezivanje ugrađenog sistema sa računarom, zatim korišćenje tog alata uz pomoć naredbi iz terminala i tako dalje. Ovo je veoma naporan i dugačak proces koji je često potrebno ponavljati veliki broj puta u toku samo jednog radnog dana, što je veoma nepraktično.

Dakle, bilo je potrebno pronaći način na koji bi se binarna datoteka već prevedenog softvera brzo i jednostavno dopremila sa interneta do ugrađenog sistema koji je prikačen na *USB* port korisničkog računara i na njemu pokrenula. Kako bi se ovaj problem razrešio, prvo je bilo potrebno rešiti problem povezivanja bilo koje internet aplikacije i računara na kome se izvršava internet pregledač koji izvršava datu aplikaciju. Ovo povezivanje je moralo da bude obavljeno tako da internet aplikacija ima potpuni pristup operativnom sistemu i svim njegovim funkcijama, i pre svega, priključenim uređajima. Na ovaj način bi se omogućilo da OBLAC pristupi svim specijalizovanim alatima za rad sa ugrađenim sistemima, zatim priključenim ugrađenim sistemima (preko *USB* portova) i da sa njima komunicira, kako bi korisnicima omogućio da iskoriste pun potencijal OBLAC-a kao razvojnog okruženja.

Najvažniji zahtevi koje je buduće rešenje moralo da ispuni su bili **univerzalnosti i fleksibilnost** - univerzalnost u smislu da je rešenje moralo da radi na svim operativnim sistemima i internet pregledačima, a fleksibilnost u smislu da je rešenje moralo da bude lako proširivo, kako bi moglo biti iskorišćeno za sve dodatne funkcionalnosti koje bi OBLAC zahtevao u budućnosti.

1.1 Pozadina problema

Poznato je da je uz pomoć klijentskog dela veb aplikacije moguće pristupiti njenom serverskom delu i zatim uz pomoć sistemskih naredbi pokrenuti neke procese na samom serveru. Ovako nešto bi moglo da se postigne, na primer, uz pomoć *CGI* skripte koja na serveru jednostavno izvršava konzolne naredbe koje joj pristižu sa internet stranice. Međutim, ovde je problem potpuno suprotan. Ovde je potrebno omogućiti da klijentski deo veb aplikacije koji se izvršava na internet pregledaču korisničkog računara ima potpuni pristup tom računaru (da podsetimo - serverski deo OBLAC aplikacije se izvršava u oblaku, tako da nema nikakve veze sa korisničkim računarom i korišćenje *CGI* skripte iz prethodnog primera ne bi bilo moguće). Ovo znači da je ustvari potrebno omogućiti samom internet pregledaču da neometano pokreće nove sistemske procese, a to zapravo znači da je potrebno ozbiljno ugroziti bezbednost korisničkog računara i otvoriti kanal za njegovu zloupotrebu kako bi se ovaj problem rešio. Upravo ovakve situacije, koje se tiču bezbednosti korisnika na internetu, svi poznatiji internet pregledači pokušavaju, i uspevaju, da zaustave.

Jedan od načina na koji se ovakvi sigurnosni problemi rešavaju je *Sandboxing*, ili u bukvalnom prevodu - zatvaranje u kutiju sa peskom. Ovo je konkretan pristup koji koristi

Google Chrome internet pregledač, dok ostali internet pregledači imaju slične načine za postizanje istih efekata. Osnovni princip *Sandbox* pristupa je da je svaki proces koji internet pregledač mora da pokrene na operativnom sistemu zapravo ograničen unutar vrlo restriktivne, veštački napravljene, sredine i da su jedini slobodno dostupni resursi procesor i radna memorija, dok su pristup fajl sistemu i mnogim drugim sistemskim funkcijama strogo zabranjeni [5].

1.2 Moguća rešenja

1.2.1 Programski dodaci

Programski dodaci ili priključci (engl. *plugin*) su softverske komponente koje proširuju funkcionalnosti već postojećih programa. U slučaju internet pregledača, programski dodaci im omogućavaju da, na primer, bez pokretanja posebnih programa reprodukuju zvučni ili video zapis koji se nalazi na nekoj internet stranici. Osim toga, programski dodaci napravljeni za internet pregledače mogu imati pristup samom sistemu na kome se pregledač i izvršava.

Ovo rešenje se praktično samo nametnulo upravo kroz proučavanje već opisanog ograničenja koje internet pregledači implementiraju. Naime, *Sandbox* koncept je dizajniran da spreči pretnje koje sa interneta mogu prodreti u korisnički računar, ali pošto je to relativno nov koncept, internet pregledači još uvek ne smeju da nameću ovakva ograničenja programskim dodacima (engl. *plugin*), jer je veliki broj njih dizajniran od strane nezavisnih programera pod pretpostavkom da im je dostupan pun pristup celom sistemu. [5]

Godinama je *NPAPI* arhitektura (*Netscape Plugin Application Programming Interface*) bila prvi izbor za razvoj programskih dodataka. Ova arhitektura se u te svrhe koristila još od 1995. godine i kasnije je postala veoma popularna i većina pregledača ju je podržavalo. [6]

Međutim, implementiranje mogućeg rešenja uz pomoć ove arhitekture je ubrzo izbačeno iz daljeg razmatranja jer bi se programski dodaci u tom slučaju pisali u *C* programskom jeziku i zavisili bi od platforme na kojoj se izvršavaju. Na taj način bi programski dodatak morao biti pisan za svaki od popularnijih operativnih sistema posebno. Osim toga, u periodu kada je ovo rešenje razmatrano, mnogi internet pregledači su već počeli da izbacuju NPAPI arhitekturu iz upotrebe kao veoma nebezbednu baš iz već spomenutih razloga neograničenog pristupa operativnom sistemu. [7] [8]

Druga popularna opcija za razvoj programskih dodataka je *PPAPI* (*Pepper Plugin Application Programming Interface*) koju je dizajnirao *Google* i koja funkcioniše na sličan način, ali nema sigurnosne propuste kao *NPAPI*. Međutim, i ovo rešenje je brzo odbačeno iz prostog razloga što *Mozilla* nije želela da podrži ovu arhitekturu u svom internet pregledaču *Firefoxu*, tako da rešenje zasnovano na ovoj arhitekturi ne bi bilo univerzalno. Slično se dogodilo i sa produžecima (engl. *extensions*) koje umesto programskih dodataka koristi samo *Firefox*.

1.2.2 *Google Chrome* platforma za aplikacije

Poslednja opcija koja je razmatrana kao moguće rešenje je bila *Google Chrome* platforma za aplikacije. Ova platforma je posvećena razvoju aplikacija uz pomoć postojećih veb tehnologija, ali uz istovremeno približavanje standardnim (*desktop*) aplikacijama. Aplikacije koje se prave uz pomoć ove platforme imaju odvojene prozore u kojima se izvršavaju (ne izvršavaju se unutar internet pregledača), uglavnom se izvršavaju nezavisno od interneta (ali mogu imati strogo kontrolisanu vezu sa njim) i obezbeđuju specijalan interfejs koji im omogućava pristup funkcijama operativnog sistema na kome se izvršavaju [9]. Na prvi pogled, ovaj pristup je obećavao, ali se kasnije i od njega odustalo iz dva razloga:

1. Interfejs koji je ova platforma nudila nije pružao dovoljno širok pristup operativnom sistemu
2. OBLAC je morao da ostane veb aplikacija koja se u potpunosti izvršava u oblaku.

1.3 Ideja rešenja

U toku potrage za pravim rešenjem za dati problem se često spominjao problem univerzalnosti. Odnosno, rešenje je moralo biti potpuno nezavisno od internet pregledača koji bi se koristio u radu sa OBLAC-om, pa je i potraga krenula u tom smeru. Takođe, kako se ne bi desilo da buduće rešenje bude zasnovano na nekoj nesigurnoj tehnologiji, ili tehnologiji koja je blizu izbacivanja iz upotrebe (poput već pominjane *NPAPI* arhitekture), još jedan važan uslov je bio da tehnologija koja će biti iskorišćena u finalnom rešenju bude novija i obećavajuća. Ovakav pristup potrazi za mogućim rešenjima je doveo do jednog, u tom trenutku logičnog rešenja u vidu *WebSocketa*.

WebSocket predstavlja tehnologiju koja je još uvek mlada, ali veoma obećavajuća, u prilog čemu ide i činjenica da je ova tehnologija već standardizovana od strane *IETF* (*Internet Engineering Task Force*) i *W3C* (*World Wide Web Consortium*). Ona omogućava da se internet aplikacija poveže brzim, dvosmernim komunikacionim kanalom sa bilo kakvom vrstom veb servera koji podržava *WebSocket* protokol.

Upravo iz ove osobine *WebSocketa* se rodila ideja o specijalnoj, nezavisnoj aplikaciji, koja bi se pokretala na računaru korisnika koji želi da koristi OBLAC i koja bi u sebi sadržala mali veb server koji bi bio zadužen za komunikaciju sa OBLAC-om. Ova aplikacija bi samim tim što se izvršava na korisničkom računaru imala potpuni pristup operativnom sistemu, a veb aplikacija (u ovom slučaju OBLAC) bi preko *WebSocketa* mogla slati zahteve za izvršenje potrebnih operacija nad pomenutim operativnim sistemom. Komunikacija bi, naravno, mogla da se vrši i u suprotnom smeru, kako bi se na primer podaci o raspoloživim uređajima prikazivali u OBLAC-u. Upravo ova specijalna lokalna aplikacija je kasnije dobila ime *SOMANETconnect.*

1.4 Izbor programskog jezika

Kada je osnova budućeg rešenja bila postavljena, pristupilo se izboru programskog jezika u kome će buduća aplikacija biti implementirana. Glavni i jedini kandidati su bili *C++* i *Java*. Inicijalno je izabran *C++* jer ne zahteva nikakav dodatni softver na korisničkom računaru, za razliku od *Jave* koja zahteva instaliranu *Java* virtualnu mašinu.

Prilikom izrade prototipa *SOMANETconnect* aplikacije, ubrzo je postalo jasno da *C++* zapravo nije najpogodniji izbor za ovaj tip aplikacije. Potreba za posebnim verzijama aplikacije za svaki od popularnih operativnih sistema, zavisnost od specijalno prevedenih biblioteka i potreba za intenzivnom obradom teksta je ubrzo dovela do napuštanja *C++* i prelaska na *Javu*, upravo zbog činjenice da se ovakva aplikacija može napisati jednom na Javi i potom izvršavati na bilo kom operativnom sistemu.

Osim što je dosta toga olakšao u razvoju i omogućio prenosivost, prelazak na *Javu* je sa sobom doneo i druge pogodnosti, kao što su na primer veliki izbor biblioteka koje su mogle jednostavno biti uvezene u projekat uz pomoć *Mavena*, zatim mnogo jednostavniju obradu i parsiranje teksta, pristup *Java* grafičkim alatima i tako dalje.

2 TEHNOLOGIJE

2.1 JAVA

Java pripada grupi objektno-orijentisanih programskih jezika i trenutno je najpopularniji programski jezik na svetu [10]. Jedna od najistaknutijih, a u ovom slučaju, i jedna od najpoželjnijih osobina ovog programskog jezika je mogućnost da se program napiše jednom i da se zatim izvršava na bilo kom uređaju koji može da pokrene *Java* virtualnu mašinu. Kod napisan u *Javi* se prevodi u *Java* bajt kod koji predstavlja set instrukcija za *Java* virtualnu mašinu i omogućava da se taj kod izvršava na svakoj platformi bez obzira na njenu arhitekturu, upravo zato što je *Java* virtualna mašina napisana za svaku arhitekturu posebno [11].

Osim za pisanje standardnih aplikacija, *Java* se može koristiti i za pisanje:

1. **Apleta** - aplikacija koje se mogu ugraditi u druge aplikacije (obično u veb aplikacije)
2. **Servleta** - aplikacija koja se izvršava na serverskoj strani i odgovara na *HTTP* zahteve koji stižu sa klijentske strane
3. **Swing aplikacija** - *Swing* je *Java* biblioteka koja se koristi za kreiranje korisničkih interfejsa i dostupna je za sve poznatije operativne sisteme

2.1.1 AWT (Abstract Window Toolkit) i SWING

Abstract Window Toolkit (AWT) je *Javin* originalni alat za prozore, grafiku i korisničke interfejse koji je prilagođen svim poznatijim platformama. *AWT* je specifičan po tome što zapravo predstavlja tanak sloj apstrakcije preko već postojećih funkcija korisničkog interfejsa nekog operativnog sistema. To znači da aplikacija koja koristi korisnički interfejs napisan koristeći *AWT* ustvari izgleda i ponaša se u skladu sa operativnim sistemom koji je trenutno izvršava.

AWT obezbeđuje dva sloja *API*-ja:

1. Osnovni interfejs između *Jave* i platforme na kojoj se ona izvršava koji omogućava direktnu vezu sa prozorima, događajima i menadžerima rasporeda. Ovaj *API* predstavlja osnovu svake interakcije *Jave* sa grafičkim elementima i takođe predstavlja osnovu *SWING*-a.
2. Osnovne grafičke elemente koji služe za pravljenje grafičkih korisničkih interfejsa (dugmići, meniji, tekstualna polja...).

Još jedna od značajnijih funkcija koje omogućava *AWT* je i pristup traci sa programima (engl. *system tray*), što omogućava kreiranje aplikacija koje se u suštini izvršavaju u pozadini, ali ipak daju korisniku mogućnost da ima interakciju sa njima.

Od *Java* verzije 1.2, *AWT* je zamenjen *SWING*-om, koji predstavlja sledeću generaciju alata za izgradnju korisničkih interfejsa. On pruža mnogo fleksibilnije komponente i pre svega, mnogo raznovrsniji izbor što se tiče izgleda istih. Za razliku od *AWT-a, SWING* nije napisan na istom jeziku kao i platforma na kojoj se izvršava, već je u potpunosti napisan u *Javi*, što ga čini potpuno nezavisnim od platforme.

SWING korisnicima omogućava da sami proširuju funkcionalnosti već postojećih komponenti zahvaljujući *Javinom* mehanizmu nasleđivanja, što ovaj alat čini još primamljivijim. Osim toga, *SWING* komponentama je takođe i veoma lako promeniti izgled.

2.1.2 JNI (Java Native Interface)

Java Native Interface (JNI) predstavlja programsku platformu koja omogućava *Java* programima da pozivaju i da budu pozivani od strane drugih programa i biblioteka koje su napisane u drugim programskim jezicima, najčešće *C, C++* i asembleru [12].

JNI se koristi u situacijama kada čist *Java* kod ne može da odgovori na specifične zahteve programera ili kada je potrebno koristiti specifične funkcije operativnog sistema. Takođe se može koristiti i kada je potrebno modifikovati ili koristiti već postojeću aplikaciju koja je napisana u drugom programskom jeziku. Osim u ovim slučajevima, *JNI* se koristi i kada su performanse izvršavanja neke funkcije od presudnog značaja, jer se *C* i asemblerski kod izvršavaju mnogo brže od čistog *Java* koda.

Ipak, *JNI* osim ovih prednosti sa sobom donosi i neke mane. Prvo, sav kod koji *Java* izvršava preko *JNI*-a mora biti unapred preveden upravo za operativni sistem na kome će se finalni program i izvršavati. To ustvari znači da ovakav *Java* program gubi svoju prenosivost, po čemu je *Java* i poznata. Ovaj problem se može delimično rešiti tako što će se sav kod koji se izvršava uz pomoć *JNI*-a unapred prevesti za sve poznatije operativne sisteme.

Drugi problem koji *JNI* donosi je gubitak sigurnosti koju *Java* pruža - niži programski jezici kao što su na primer *C* i *C++* nemaju automatsko oslobađanje memorije i proveru tipova, što znači da se prilikom pisanja ovakvog koda dodatno mora paziti na takve stvari.

Sam proces pisanja programa koji koriste mogućnosti *JNI* programske platforme se može prikazati kroz jednostavan *C* primer koji se sastoji od sledećih šest koraka:

1. Kreirati klasu *HelloWorld.java* koja deklariše nativnu funkciju (koristeći rezervisanu reč *native*), a zatim je i poziva. Ova klasa takođe mora i učitati nativnu biblioteku u kojoj je gore pomenuta funkcija i implementirana. Uz pomoć rezervisane reči *static* se osigurava da će se nativna biblioteka učitati pre izvršavanja bilo kakvog koda, odnosno u toku same inicijalizacije klase *HelloWorld*.

```
Class HelloWorld {
    private native void print();

    public static void main(String[] args) {
        new HelloWorld().print();
    }

    static {
        System.loadLibrary("HelloWorld");
    }
}
```

2. Iskoristiti *Java* alat pod imenom *javac* kako bi se prevela klasa *HelloWorld* i dobila *Java* datoteka *HelloWorld.class*.

```
javac HelloWorld.java
```

3. Izvršiti sledeću naredbu u direktorijumu koji sadrži sve prethodne datoteke:

```
javac -jni HelloWorld
```

Alat *javac* je takođe standardni *Java* alat. Na ovaj način se generiše *HelloWorld.h* datoteka koja sadrži prototip nativne funkcije napisan u *C* programskom jeziku. Ova datoteka sadrži sledeći kod:

```
JNIEXPORT void JNICALL
Java_HelloWorld_print (JNIEnv *, jobject);
```

JNIEXPORT i *JNICALL* predstavljaju standardne makroe koje prevodilac koristi kako bi identifikovao nativnu funkciju. Prvi argument nativne funkcije *JNIEnv* predstavlja pokazivač koji se nalazi u svim nativnim funkcijama i pokazuje na tabelu sa svim dostupnim nativnim funkcijama. Drugi argument, *jobject* predstavlja referencu na sam *HelloWorld* objekat. Nijedan od ova dva argumenta nije potreban za ovako jednostavan primer.

4. Napisati implementaciju nativnog metoda u novoj datoteci *HelloWorld.c*:

```
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"

JNIEXPORT void JNICALL
Java_HelloWorld_print(JNIEnv *env, jobject obj)
{
    printf("Hello World!\n");
    return;
}
```

Ova funkcija jednostavno ispisuje *"Hello World!"* i vraća se nazad. Kako bi ovaj kod ispravno funkcionisao, potrebno je uvesti nekoliko datoteka:

   a. *jni.h* - datoteka koja sadrži informacije koje su potrebne kako bi nativni kod mogao da pozove *JNI* funkcije
   b. *stdio.h* - sadrži definiciju *printf* funkcije
   c. *HelloWorld.h* - sadrži prototip nativne funkcije koja je ovde definisana

5. Prevesti *C* implementaciju nativnih funkcija koristeći ugrađeni prevodilac lokalnog operativnog sistema kako bi se dobila datoteka koja sadrži prevedenu biblioteku (*libHelloWorld.so* u *Linuxu* ili *HelloWorld.dll* u *Windowsu*)
6. Pokrenuti *Java* program koristeći naredbu:

```
java HelloWorld
```

Ovom prilikom se mora voditi računa da program ima pristup prethodno prevedenoj biblioteci. U *Linuxu* se na primer pre pokretanja programa mora sačuvati putanja do direktorijuma koji sadrži datoteku sa bibliotekom u sistemskoj promenljivoj *LD_LIBRARY_PATH*, dok se u *Windowsu* ova datoteka jednostavno može staviti u isti direktorijum gde se nalaze i ostale izvršne datoteke pokrenutog programa.

U *Linuxu* se putanja do direktorijuma sa datotekom biblioteke može navesti i prilikom pokretanja programa:

```
java -Djava.library.path=./lib HelloWorld
```

Mnoge standardne *Java* biblioteke direktno zavise od *JNI* platforme - kada je potrebno korisniku omogućiti pristup datotekama i grafičkim (*AWT)* ili zvučnim funkcijama operativnog sistema. Osim standardnih *Java* biblioteka, postoje i mnoge nestandardne koji dodaju još više funkcionalnosti i pritom se oslanjaju na *JNI*, na primer za pristup *USB* uređajima i slično.

Sve ostale informacije potrebne za rad sa *JNI* programskom platformom se mogu naći u zvaničnom uputstvu [12].

2.1.3 Apache Maven

Maven je alat za menadžment projekata koji obuhvata model projektnog objekta, skup standarada, životni ciklus projekta, sistem za upravljanje zavisnostima projekta i logiku za izvršavanje ciljeva dodatnih programa *Mavena* u definisanim fazama životnog ciklusa [13]. U suštini, *Maven* se može koristiti za veliki broj stvari u okviru menadžmenta projekata, ali se najčešće koristi za automatizaciju procesa prevođenja programa i upravljanje zavisnostima projekata. Pored ovoga, *Maven* se čak može koristiti i za generisanje veb strana o projektu ili za automatsko generisanje izveštaja.

Maven se rukovodi principom „konvencija pre podešavanja", koji propisuje da svaki aspekat njegovog ponašanja koji je moguće podesiti mora da ima neku podrazumevanu vrednost. To ukratko znači da sistem jednostavno mora da funkcioniše bez ikakvog nepotrebnog podešavanja. Ovaj princip rada se postiže tako što svaka opcija ima već unapred definisanu podrazumevanu vrednost. Tako na primer, *Maven* bez ikakvog dodatnog podešavanja pretpostavlja da se za Java projekat izvorni kod nalazi na putanji *${basedir}/src/main/java*, da se potrebni resursi nalaze na *${basedir}/src/main/resources*, da će se program prevesti u *JAR* datoteku i da prevedeni bajt kod treba da se sačuva na putanji *${basedir}/target/classes*. *Maven* takođe zna i koje svoje dodatne programe treba da iskoristi kako bi dobio potrebne rezultate nakon prevođenja *Java* programa. Ovo sve praktično znači da *Maven* ne zahteva ništa od programera osim da stavi izvorni kod na odgovarajuće mesto i da izvrši *Maven* naredbu za pokretanje prevođenja. [13]

Naravno, postoje i korisnici koji ne žele da budu toliko ograničeni konvencijom. Baš iz tog razloga *Maven* dopušta menjanje skoro svakog aspekta svog ponašanja.

Ono što čini *Maven* izuzetno popularnim jeste njegovo korišćenje zajedničkog interfejsa za prevođenje programa. Pre pojave *Mavena*, prevođenje kompleksnih programa, koji su se sastojali od većeg broja modula, je bilo veoma komplikovano, zato što je svaki modul imao svoj način prevođenja i zavisnosti od biblioteka koje su morale ručno biti ispunjene. Sada, sa *Mavenom* na raspolaganju, ovaj proces je veoma jednostavan zato što *Maven* propisuje zajednički interfejs za svaki od ovih modula, tako da se prevođenje za sve njih vrši na isti način. Ovo je glavni razlog popularnosti *Mavena* i zašto ga veliki broj programera koristi na svojim projektima. [13]

Još jedna *Mavenova* prednost su njegovi dodatni programi. Samo jezgro *Mavena* je veoma jednostavno - ono samo parsira *XML*, prati životni ciklus projekta i shodno tome upravlja programskim dodacima. Upravo programski dodaci su odgovorni za većinu posla koji *Maven* izvršava. Svaki programski dodatak ima definisan cilj koji mora da ispuni, bilo da

se radi o prevođenju programa, pakovanju bajt koda ili bilo kom drugom zadatku. Programski dodaci se nalaze na *Maven* repozitorijumu (koji se nalazi na internetu i konstantno se osvežava najnovijim verzijama dodatnih programa i drugih projekata) i odatle se svlače svaki put kada za tim ima potrebe, odnosno prilikom prvog pokretanja *Mavena* ili prilikom promene verzije nekog od programskih dodataka. Ovakvo ponašanje omogućava korisnicima da na veoma jednostavan način preuzmu najnoviju verziju nekog programskog dodatka i time dodaju još više mogućnosti za upravljanje projektom uz pomoć *Mavena*.

Kao što je već rečeno, *Maven* održava model projektnog objekta koji je zapisan u *XML* formatu (engl. *Project Object Model - POM*), koji opisuje sve atribute projekta i koji se čuva unutar osnovnog direktorijuma projekta u datoteci pod imenom *pom.xml*. U njemu su između ostalog definisani identitet projekta, njegova struktura, način na koji će biti preveden i njegove zavisnosti. Ova datoteka govori *Mavenu* o kojoj vrsti projekta se radi i na koji način je potrebno modifikovati podrazumevano ponašanje kako bi se program preveo na odgovarajući način [13].

Kao što se može videti u sledećem jednostavnom primeru *Java* projekta, svaka *pom.xml* datoteka na početku ima definiciju *XML* šeme i verziju modela koji će biti korišćeni. Odmah zatim se navode identifikacija grupe (obično predstavlja unazad navedenu adresu osnovne internet strane osobe ili kompanije koja razvija dati softver i imena samog softvera, na primer *net.java.dev.jna* za *Javin JNA* projekat), zatim skraćeno ime projekta, njegova trenutna verzija, način pakovanja i njegovo puno ime.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>tmp.project.example</groupId>
    <artifactId>example</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>

    <name>ExampleProject</name>
```

POM ima još jednu korisnu opciju, a to je definisanje neke vrste konstanti koje se kasnije mogu koristiti u njegovim okvirima. Na taj način je na primer moguće jednom definisati verziju programskog dodatka u obliku konstante i zatim koristiti tu konstantu gde god je potrebno u nastavku datoteke. Na ovaj način se svaka iduća promena verzije bilo kog od programskih dodataka vrši samo na jednom mestu, što olakšava kasnije održavanje projekta.

```
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

        <!-- JDK -->
        <target.jdk>1.7</target.jdk>

        <!-- Maven -->
        <maven-compiler-plugin.version>3.2</maven-compiler-plugin.version>

        <!-- testing -->
        <junit.version>4.11</junit.version>
```

```
        <!-- Logging -->
        <slf4j.version>1.7.10</slf4j.version>

        <!-- Apache Commons -->
        <commons-codec.version>1.10</commons-codec.version>
        <commons-lang3.version>3.4</commons-lang3.version>
    </properties>
```

Zatim, na red dolaze definicije faza životnog ciklusa projekta i programskih dodataka koji se koriste u njima. U navedenom jednostavnom primeru je definisana samo jedna, *build* faza, koja opisuje prevođenje projekta i samo jedan programski dodatak koji u stvari izvršava prevođenje - *maven-compiler-plugin*. Ovde se može videti i praktična primena već pomenutih konstanti koje određuju verziju programskog dodatka koji će biti iskorišćen, kao i *Java* verzija koju će on koristiti. Može se primetiti da je definicija programskog dodatka koji se koristi veoma slična definiciji projekta na početku *pom.xml* datoteke. Upravo to je zajednički interfejs koji je ranije spomenut.

```
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>${maven-compiler-plugin.version}</version>
                <configuration>
                    <compilerVersion>${target.jdk}</compilerVersion>
                    <source>${target.jdk}</source>
                    <target>${target.jdk}</target>
                    <encoding>${project.build.sourceEncoding}</encoding>
                    <showDeprecation>true</showDeprecation>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

Sledeći deo *pom.xml* datoteke definiše zavisnosti datog projekta od drugih projekata ili bibloteka. Koristeći već spomenuti zajednički interfejs, korisnik definiše sve projekte od kojih trenutni projekat zavisi navodeći informacije o njima koje *Maven* koristi da jednoznačno pronađe svaki od njih na *Maven* repozitorijumu kako bi ih svukao na lokalni računar i iskoristio u procesu prevođenja.

```
    <dependencies>
        <!-- Testing -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>

        <!-- Logging -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>${slf4j.version}</version>
        </dependency>

        <!-- Apache Commons -->
        <dependency>
            <groupId>commons-codec</groupId>
            <artifactId>commons-codec</artifactId>
            <version>${commons-codec.version}</version>
        </dependency>
        <dependency>
```

```
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>${commons-lang3.version}</version>
        </dependency>
    </dependencies>
</project>
```

Na osnovu ovako definisanog modela projektnog objekta, sve što je potrebno uraditi kako bi se *Java* projekat preveo je izvršiti naredbu „*mvn install*".

Zahvaljujući načinu na koji se faze životnog ciklusa projekta mogu definisati, *Maven* takođe korisnicima omogućava da sami naprave i opišu bilo kakav proces koji se mora izvršiti pre, za vreme ili posle prevođenja nekog programa. Na ovaj način se mogu, na primer, automatizovati veoma komplikovani procesi pripreme pre ili čišćenja posle prevođenja.

Osim za prevođenje programa u *Javi*, *Maven* se može koristiti i za prevođenje programa napisanih u *C#, Ruby, Scala* i drugim programskim jezicima.

2.2 WebSocket

Jedan od osnovnih zadataka interneta je da prenese informacije. U tu svrhu se od trenutka kada je nastao *World Wide Web* koristio *Hypertext Transfer Protocol (HTTP)* [14].

HTTP je protokol koji se koristi u komunikaciji preko interneta u kojoj postoje dve strane - klijent i server. Komunikacija se vrši tako što klijent (obično internet pregledač) šalje *HTTP* zahtev serveru, a server zatim vraća odgovor koji u sebi sadrži tražene podatke (na primer *HTML* stranicu). Kako se vremenom razvijala potreba za sve većom brzinom, interaktivnošću i raznolikošću interneta, tako je i *HTTP* morao uporedo da se razvija. Međutim, vrlo brzo je postalo jasno da će internet protokoli morati da podržavaju komunikaciju u realnom vremenu, komunikaciju koja omogućava serverima da sami pošalju podatke klijentu bez prethodnog zahteva, kao i druge načine komunikacije. Dugo su inženjeri pokušavali da zaobiđu ovaj problem i dalje koristeći *HTTP* kao osnovu (*HTTP polling, long polling, HTTP streaming, AJAX*), ali su njegove osobine (poludupleks komunikacija i sistem zahtev-odgovor) previše usporavale komunikaciju i povećavale količinu metapodataka koje je bilo potrebno preneti [15].

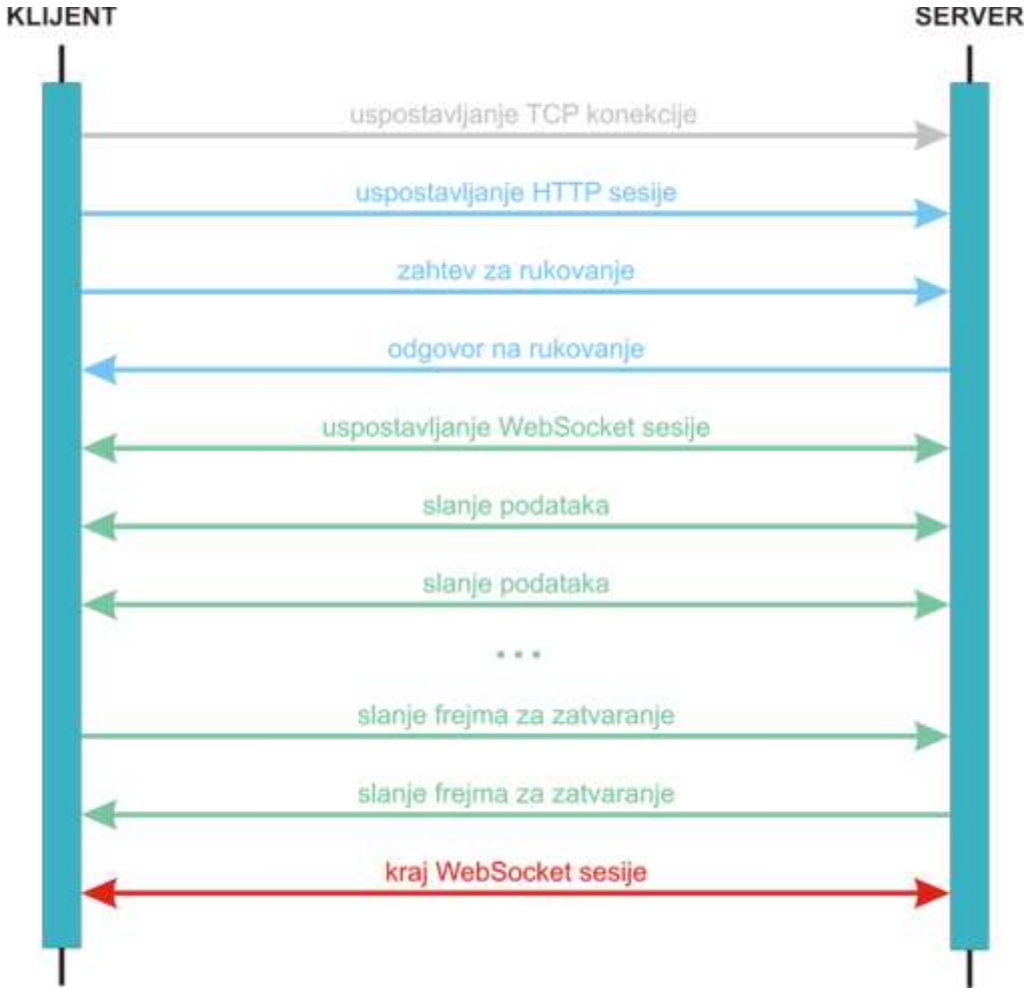
WebSocket je nastao kao direktan odgovor na prethodne probleme. On u osnovi podržava dvosmernu dupleks komunikaciju preko jednog internet soketa (jedna *TCP* konekcija) između klijenta i servera. Ovo u suštini znači da je uz pomoć *WebSocketa* moguće napraviti jedan komunikacioni kanal koji je konstantno otvoren, i koji, s obzirom da je dvosmeran, omogućava i serverskoj i klijentskoj strani da šalju podatke u bilo kom trenutku, a da pritom ne moraju da čekaju da stigne zahtev sa druge strane. Ovakvi kanali su savršeni za komunikaciju koju je potrebno izvršavati uživo. [15]

WebSocket se sastoji od protokola i *API*-ja, i trenutno je podržan od strane svih popularnijih internet pregledača.

2.2.1 WebSoket protokol

Svaka *WebSocket* konekcija počinje *HTTP* zahtevom koji je veoma sličan normalnom *HTTP* zahtevu, osim što ima specijalno *Upgrade* zaglavlje. Ovakvo zaglavlje signalizira serveru da klijent želi da unapredi konekciju u neki drugi protokol. Zahtev koji signalizira

serveru da klijent želi da unapredi komunikaciju u *WebSocket* se naziva zahtev za rukovanje (engl. *handshake request*). Nakon što server prihvati ovakav zahtev, komunikacija se prebacuje u format u kome se svi podaci šalju u obliku *WebSocket* frejmova.



Slika 2-1 WebSocket protokol

Frejmovi predstavljaju jedinice prenosa podataka preko *WebSocket* konekcije i oni zapravo predstavljaju delove poruka u binarnom obliku koji se kasnije mogu sastaviti u poruke. Svaki frejm sadrži zaglavlje koje signalizira njegov početak i u sebi nosi sledeće informacije:

1. **Kod** koji označava vrstu podataka koja se nalazi u frejmu (tekst, binarni podaci, zahtev za zatvaranjem konekcije...)
2. **Dužina** tih podataka (zapis ovog podatka može biti produžen)
3. **Maska** koja služi za obradu podataka kako bi oni bili kompatibilniji sa postojećim *HTTP* proksijima

Slika 2-2 WebSocket frejm

Celokupna komunikacija se uspostavlja preko istog porta kao i standardne veb usluge (*HTTP* port 80), što je specijalno pogodno pošto se na taj način zaobilaze standardne (*firewall)* zaštite koje ne dozvoljavaju ostale vrste komunikacije.

WebSocket protokol takođe propisuje i način na koji se svaka konekcija ova vrste mora zatvoriti u obliku zatvarajućeg rukovanja (engl. *closing handshake*). Na ovaj način se razlikuje namerno zatvaranje konekcije od nenamernog (zbog problema u vezi i slično). Kada se *WebSocket* konekcija namerno zatvori, šalje se frejm koji u sebi sadrži numerički kod (broj između 1000 i 1015) koji označava razlog zatvaranja. Neki od razloga za zatvaranje mogu biti: normalno zatvaranje - 1000, greška u protokolu - 1002, neispravni podaci - 1007, poruka prevelika - 1009 i tako dalje.

2.2.2 WebSocket API

WebSocket API je interfejs koji omogućava aplikacijama da koriste *WebSocket* protokol. Uz pomoć ovog jednostavnog *API*-ja, aplikacija može slati i primati poruke preko dvosmernog dupleks kanala za komunikaciju. [15]

Da bi se aplikacija povezala sa serverom potrebno je napraviti novu instancu *WebSocket* objekta uz pomoć konstruktora kome se prosleđuje adresa datog servera. Prilikom kreiranja ovog objekta, veza između klijenta i servera se ostvaruje preko WebSocket protokola na već opisani način i tada server i klijent mogu slobodno razmenjivati poruke uz pomoć metoda koje definiše *WebSocket API.*

```
// Otvaranje nove WebSocket konekcije
var ws = new WebSocket("ws://www.websocket.org");
```

WebSocket protokol definiše dve *URI* šeme - *ws* za neenkriptovani saobraćaj i *wss* za enkriptovani saobraćaj.

2.2.2.1 Događaji

Pošto je *WebSocket API* zasnovan na događajima, svaka akcija koja se dogodi sa druge strane konekcije će na već kreiranom *WebSocket* objektu rezultirati pozivom funkcije koja je zadužena za obradu tog događaja. Ovu funkciju je potrebno ručno definisati i pridružiti datom objektu. To upravo znači da aplikacija ne mora aktivno da čeka na poruku sa serverske strane, već da će automatski biti obaveštena kada ona stigne.

WebSocket objekat može reagovati na četiri različite vrste događaja. To su:

1. **Otvaranje** (engl. *open*) - događaj koji se dešava u trenutku uspešnog otvaranja konekcije, posle čega je dozvoljeno slati podatke.

```
// Funkcija koja obrađuje otvaranje konekcije
ws.onopen = function(e) {
    console.log("WebSocket je spreman");
};
```

2. **Poruka** (engl. *message*) - događaj koji se dešava u trenutku prijema poruke. Poruke mogu biti tekstualnog ili binarnog tipa i način njihove obrade zavisi od toga.

```
// Funkcija koja obrađuje primljenu poruku
ws.onmessage = function(e) {
```

```
    if(typeof e.data === "string"){
        console.log("Tekstualna poruka primljena", e, e.data);
    } else {
        console.log("Binarna poruka primljena", e, e.data);
    }
};
```

3. **Greška** (engl. *error*) - događaj koji se dešava kada dođe do greške. U tom slučaju se i konekcija automatski zatvara. Dobra praksa je imati posebnu funkciju koja obrađuje grešku koja se desila. Ovakva funkcija može, na primer, pokušati da ponovo uspostavi vezu ili da prikaže i zabeleži numerički kod i opis greške koji se nalaze u ovakvom događaju.

```
// Funkcija koja obrađuje grešku
ws.onerror = function(e) {
    console.log("WebSocket Error: " , e);
    // Specijalna funkcija koja obrađuje greške
    handleErrors(e);
};
```

4. **Zatvaranje** (engl. *close*) - događaj koji se dešava kada se *WebSocket* konekcija zatvori.

```
// Funkcija koja obrađuje zatvaranje konekcije
ws.onclose = function(e) {
    console.log("Konekcija zatvorena", e);
};
```

Od trenutka kada je konekcija zatvorena, klijent i server više ne mogu da šalju i primaju poruke. S obzirom da se konekcija može zatvoriti iz više razloga, ovaj događaj u sebi nosi još tri atributa čije se vrednosti ponekad mogu iskoristi za određivanje razloga zatvaranja:

a. *wasClean* - sadrži bulovsku vrednost koja označava da li je veza prekinuta namerno ili se dogodila neka greška
b. *code* - numerički kod razloga zatvaranja konekcije koji šalje server
c. *reason* - razlog zatvaranja konekcije koji šalje server

2.2.2.2 Funkcije

WebSocket API definiše dve funkcije:

1. ***send*** - od trenutka kada je konekcija otvorena, ova funkcija se može pozvati na klijentskoj strani kako bi se poslala poruka.

```
// Slanje tekstualne poruke
ws.send("Hello WebSocket!");
```

Osim tekstualnih poruka, ova funkcija se na isti način može koristiti i za slanje binarnih podataka (u obliku *Bloba* ili *ArrayBuffera*).

2. **close** - ova funkcija se koristi za zatvaranje konekcije, koja u tom trenutku može biti u stanju otvaranja ili u već otvorenom stanju. Ako je konekcija već zatvorena ili u procesu zatvaranja, ova funkcija ne radi ništa.

```
// Zatvaranje konekcije
ws.close();
```

Ova funkcija takođe može primiti i dva argumenta koji su već ranije spominjani - numerički kod i razlog zatvaranja. Ove informacije će zatim biti na raspolaganju serveru. Na primer:

```
ws.close(1000, "Closed normally");
```

2.3 JSON (JavaScript Object Notation)

JSON predstavlja veoma jednostavan format otvorenog standarda koji koristi ljudski čitljiv tekst za razmenu podataka. On je potpuno nezavistan od programskih jezika, ali koristi konvencije koje su slične onima koje se koriste u velikom broju programskih jezika. [16]

JSON definiše mali skup strukturnih pravila kako bi na što kompaktniji način predstavio strukturisane podatke. Celokupni *JSON* je izgrađen na dva strukture koje se na ovaj ili onaj način koriste u većini programskih jezika:

1. Kolekcija parova atribut-vrednost, koja se u programskim jezicima predstavlja kao objekat, mapa, struktura itd.
2. Uređena lista vrednosti, koja se u programskim jezicima predstavlja kao niz, lista, vektor itd.

Navedene strukture su univerzalne i poznaje ih većina modernih programskih jezika, što *JSON* čini posebno pogodnim za razmenu podataka. Jezik koji koristi strukture najsličnije *JSON*-u je upravo *JavaScript*, od koga ime *JSON* i vuče korene. Ova njegova osobina ga čini posebno pogodnim za upotrebu u *JavaScript* aplikacijama. Upravo zahvaljujući *JavaScriptu*, većina današnjih internet pregledača ima unapred ugrađenu sposobnost jednostavnog čitanja i pisanja *JSON* podataka, tako da se on uglavnom koristi kao alternativa *XML*-u u komunikaciji između klijenta i servera.

Primer:

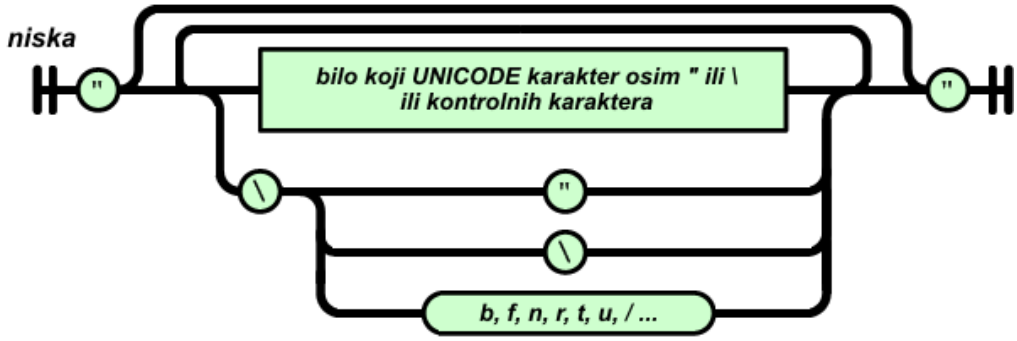
```
{ "Books":
  [
    { "ISBN":"ISBN-0-13-713526-2",
      "Price":85,
      "Edition":3,
      "Title":"A First Course in Database Systems",
      "Authors":[ {"First_Name":"Jeffrey", "Last_Name":"Ullman"},
                  {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }
    ,
    { "ISBN":"ISBN-0-13-815504-6",
      "Price":100,
      "Remark":"Buy this book bundled with 'A First Course' - a great deal!",
      "Title":"Database Systems:The Complete Book",
      "Authors":[ {"First_Name":"Hector", "Last_Name":"Garcia-Molina"},
                  {"First_Name":"Jeffrey", "Last_Name":"Ullman"},
                  {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }
  ]
}
```

2.3.1 Sintaksa

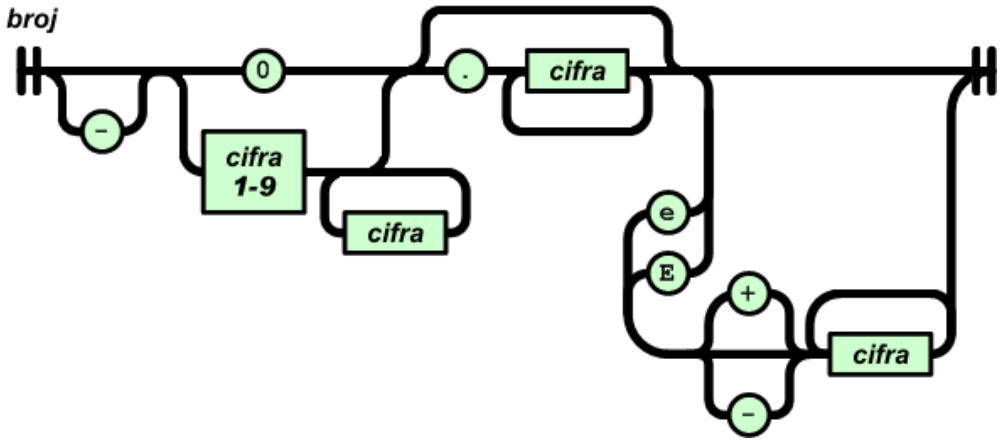
S obzirom da se *JSON* zasniva na veoma malom skupu pravila, cela njegova sintaksa se može opisati sa nekoliko grafova.

Osnovni koncepti u *JSON*-u su:

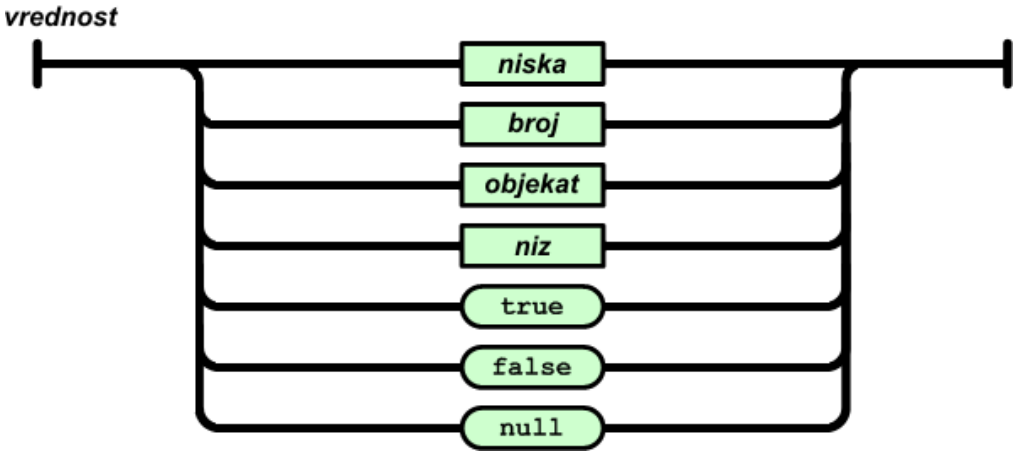
1. **Niske** (engl. *string*) - veoma slične niskama u *C* i *Java* programskim jezicima. Sastoje se od niza *Unicode* karaktera i oivičene su navodnicima. Svi karakteri osim kontrolnih (U+0000 i U+001F) se mogu naći unutar jedne niske, s tim da sami navodnici i karakter "\" moraju imati još jedan karakter "\" ispred sebe. Osim ovih karaktera, niske mogu sadržati i specijalne simbole koji se predstavljaju uz pomoć dva karaktera (na primer "\n" koji označava novi red i "\t" koji označava tabulator).



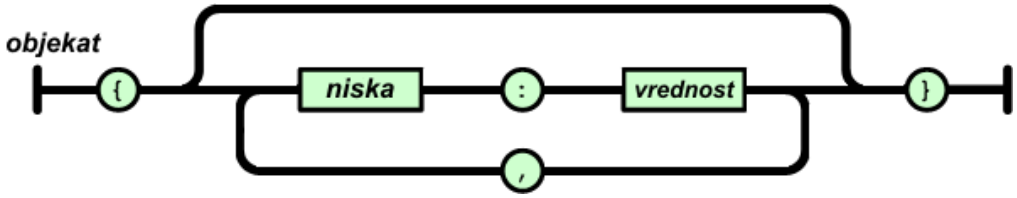
2. **Brojevi** - uvek su predstavljeni u dekadnom sistemu bez vodećih nula, mogu biti negativni i decimalni. Takođe mogu biti predstavljeni i sa eksponentom od deset.



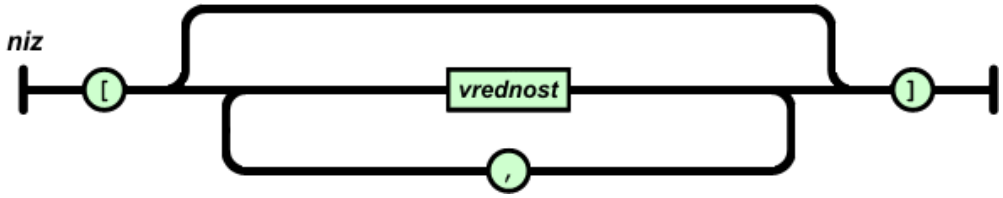
3. **Vrednosti** mogu biti niske, brojevi, objekti, nizovi, *true*, *false* i *null*.



4. **Objekti** - predstavljaju strukturu koja je oivičena vitičastim zagradama i koja u sebi sadrži nula ili više parova atribut-vrednost, gde je atribut predstavljen niskom. Ovi parovi su odvojeni zarezima, dok su imena atributa odvojena dvotačkama od njihovih vrednosti.



5. **Nizovi** - strukture koje su oivičene uglastim zagradama i koje su sadrže nula ili više vrednosti odvojenih zarezima.



2.3.2 JSON-RPC 2.0

JSON-RPC (JavaScript Object Notation - Remote Procedure Call) je vrlo jednostavan protokol za pozivanje procedura na serveru, koji je zapisan u *JSON* formatu. On se sastoji od vrlo striktnih pravila koja se moraju poštovati prilikom pisanja i slanja ovakvih zahteva i odgovora na iste. [17]

JSON-RPC 2.0 protokol definiše tri vrste poruka:

1. **Zahtev** - poziv nekog metoda koji klijent upućuje serveru. Svaki zahtev predstavlja *JSON* objekat koji sadrži sledeća četiri atributa:
   - ***jsonrpc*** - niska koja definiše verziju *JSON-RPC* protokola, koja mora biti „2.0".
   - ***method*** - niska koja sadrži ime metoda koji se poziva na serverskoj strani.
   - ***params*** - sadrži parametre koji će biti prosleđeni pozvanom metodu i može biti izostavljen. Parametri se mogu poslati u obliku parova atribut-vrednost ili u nizu.
   - ***id*** - ukoliko postoji, ovaj atribut mora sadržati neku vrednost koju određuje klijent i koja može biti niska, broj ili *null*. Odgovor koji server vraća na neki zahtev mora u sebi sadržati isti ***id*** kao i dati zahtev. Ako ovaj atribut ne postoji, pretpostavlja se da poruka nosi obaveštenje.
2. **Obaveštenje** - predstavlja zahtev bez ***id*** atributa, čime se serveru daje do znanja da klijent nije zainteresovan za odgovor. Server ne sme odgovoriti na obaveštenje.
3. **Odgovor** - kada klijent uputi neki zahtev, server mora vratiti odgovor. Odgovor je takođe u obliku *JSON* objekta i sadrži sledeća četiri atributa:
   - ***jsonrpc*** - niska koja definiše verziju *JSON-RPC* protokola, koja mora biti „2.0".
   - ***result*** - vrednost ovog atributa predstavlja rezultat izvršavanja metoda koji je bio pozvan na serverskoj strani. Ovaj atribut mora postojati ukoliko je poziv metoda bio uspešan. Ukoliko se prilikom poziva dogodila greška, ovaj atribut ne sme biti prisutan.
   - ***error*** - ovaj atribut u odgovoru sme biti prisutan samo ukoliko je došlo do greške i predstavljen je *JSON* objektom koji sadrži sledeće atribute:

- ***code*** - označava tip greške i mora biti ceo broj
- ***message*** - niska koja ukratko opisuje grešku
- ***data*** - objekat koji nosi više detalja o grešci (ne mora biti prisutan)

Primeri:

- Zahtev klijenta:

```
{"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
```

- Odgovor servera:

```
{"jsonrpc": "2.0", "result": 19, "id": 1}
```

- Obaveštenje:

```
{"jsonrpc": "2.0", "method": "update", "params": [1,2,3,4,5]}
```

- Greška:

```
{"jsonrpc": "2.0", "error": {"code": -32601, "message": "Method not found"},
"id": "1"}
```

3 RAZVOJ SOMANETconnect APLIKACIJE

3.1 Server

Kako bi *SOMANETconnect* mogao da komunicira sa OBLAC-om preko *WebSocket* kanala, potreban je mali i brz veb server koji će biti ugrađen u samu *SOMANETconnect* aplikaciju. Na sreću, jedan od najmanjih i najpoznatijih veb servera, *Jetty* je već obezbedio punu funkcionalnost veb servera u obliku *Java* biblioteke. Kako bi se ova biblioteka koristila unutar aplikacije, potrebno je samo uvesti je uz pomoć *Mavena*. Osim osnovne biblioteke *Jetty* servera, potrebno je uvesti i biblioteku koja mu pruža podršku za korišćenje *WebSocket* protokola.

```
<dependency>
    <groupId>org.eclipse.jetty.websocket</groupId>
    <artifactId>websocket-api</artifactId>
    <version>9.2.10.v20150310</version>
</dependency>

<dependency>
    <groupId>org.eclipse.jetty.websocket</groupId>
    <artifactId>websocket-server</artifactId>
    <version>9.2.10.v20150310</version>
</dependency>
```

Kada se *Jetty* biblioteka uveze uz pomoć *Mavena*, ona je već dostupna za korišćenje u programu koji se razvija. Najznačajnija i osnovna komponenta ove biblioteke je naravno *Jetty* serverska klasa, koju je pre korišćenja potrebno prvo podesiti.

Pod podešavanjem servera se podrazumeva postavljanje porta na kome će se server izvršavati, biranje vrste servera (u ovom slučaju će biti korišćen *WebSocket* server), kao i uspostavljanje sigurne konekcije (postavljanjem *SSL* ključa) ukoliko za to postoji potreba. Kreiranje nove instance servera i njeno podešavanje je najbolje učiniti odjednom koristeći namenski napravljenu pomoćnu *Java* klasu koja se naziva fabrika (engl. *factory*).

Server koji se nalazi u *SOMANETconnect* aplikaciji se izvršava na osnovnoj *IP* adresi operativnog sistema (127.0.0.1), što ga čini jednostavno dostupnim iz internet pregledača koji se izvršava na istom operativnom sistemu.

```
public class SomanetConnectServerFactory {
    public static Server createOblacServer(final WebSocketCreator creator) {
        Server server = new Server();

        // Normal web socket connection (ws)
        ServerConnector connector = new ServerConnector(server);
        connector.setPort(
            ApplicationConfiguration.getInstance().getInt("application.ws.port"));
        server.addConnector(connector);

        // Secure (SSL) web socket connection (wss)
        HttpConfiguration https = new HttpConfiguration();
        https.addCustomizer(new SecureRequestCustomizer());

        SslContextFactory sslContextFactory = new SslContextFactory();
        try {
            KeyStore keyStore = KeyStore.getInstance("jks");
            keyStore.load(SomanetConnectServerFactory.class.getResourceAsStream(
                ApplicationConfiguration.getInstance().getString(
                    "application.ws.ssl.key_store_path")),
                ApplicationConfiguration.getInstance().getString(
```

```
                    "application.ws.ssl.key_store_password").toCharArray());
            sslContextFactory.setKeyStore(keyStore);
            sslContextFactory.setKeyStorePassword(
                ApplicationConfiguration.getInstance().getString(
                    "application.ws.ssl.key_store_password"));
        } catch (KeyStoreException | IOException | NoSuchAlgorithmException |
                 CertificateException e) {
            logger.error(e.getMessage());
        }

        ServerConnector sslConnector = new ServerConnector(server,
                new SslConnectionFactory(sslContextFactory, "http/1.1"),
                new HttpConnectionFactory(https));
        sslConnector.setPort(
            ApplicationConfiguration.getInstance().getInt(
                "application.ws.ssl.port"));
        server.addConnector(sslConnector);

        // Setup the basic application "context" for this application at "/" and a
        // web socket handler
        ContextHandler context = new ContextHandler();
        context.setContextPath("/");
```

Osim podešavanja parametara novog servera, fabrika može i da odredi ponašanje tog servera u zavisnosti od nadolazećeg saobraćaja. *Jetty* serveri u tu svrhu koriste adapter klase koje u svakom pogledu određuju ponašanje nekog servera tako što u sebi sadrže logiku koja za svaki zahtev sprema određeni odgovor.

Adapteri se ubrizgavaju u *Jetty* servere uz pomoć kreatora. Oni predstavljaju vrlo jednostavnu pomoćnu klasu kojoj je jedini cilj da napravi novi adapter određenog tipa.

```
        context.setHandler(new WebSocketHandler() {
            @Override
            public void configure(WebSocketServletFactory webSocketServletFactory){
                webSocketServletFactory.setCreator(creator);
            }
        });
        server.setHandler(context);

        return server;
    }
}
```

Kao što je već rečeno, adapteri određuju ponašanje servera. Oni uz pomoć metoda koji su određeni u *WebSocket API*-ju određuju njegovo ponašanje kada se veza uspostavi, kada se veza prekine, kada dođe do greške ili kada na server stigne neki zahtev.

Prvi metod ovog tipa je metod *onWebSocketConnect*. Kako mu i samo ime kaže, ovaj metod se poziva u trenutku kada se uspešno uspostavila *WebSocket* konekcija. Ovaj metod daje pristup objektu tipa *Session* koji u sebi sadrži sve informacije o uspostavljenoj konekciji i pruža mogućnost promene parametara iste. U narednom delu koda možemo videti kako se ovaj objekat koristi kako bi se odredila maksimalna dozvoljena veličina tekstualne poruke koju ovaj server može primiti, kao i ukidanje vremena za koje će se veza automatski prekinuti ako na njoj ne postoji nikakva komunikacija. Zatim se promena statusa konekcije upisuje u dnevnik događaja, grafička komponenta aplikacije *SOMANETconnect* se ažurira trenutnim statusom konekcije i spisak trenutno raspoloživih uređaja se prosleđuje klijentskoj strani.

```
public class OblacWebSocketAdapter extends WebSocketAdapter implements Observer {
    ...
```

```
@Override
public void onWebSocketConnect(Session session) {
    super.onWebSocketConnect(session);
    getSession().getPolicy().setMaxTextMessageSize(10 * MB);
    getSession().setIdleTimeout(-1);
    logger.info("Socket connected to " + session.getRemoteAddress());
    SomanetConnectSystemTray.getInstance().oblacConnected(true);
    Util.sendWebSocketResultResponse(getRemote(),
        DeviceManager.getInstance().getDevices(), Constants.LIST);
}
```

Sledeći metod reaguje na prijem tekstualne poruke. Ovde se zapravo nalazi i samo srce ove aplikacije. Komunikacija između klijenta i servera se obavlja uz pomoć ranije opisanog *JSON-RPC* protokola, odnosno tekstualnih poruka koje su zapisane u tačno određenom formatu koristeći *JSON* specifikaciju. Kada takav zahtev stigne sa klijenta, server koristi adapter i njegov *onWebSocketText* metod kako bi isti obradio i odgovorio na njega.

Adapter najpre parsira zahtev i pritom proverava da li je format ispravan. Zatim, određuje metod koji treba izvršiti i, ukoliko postoje, parametre koje treba proslediti datom metodu. Izabrani metod se zatim izvršava, obrađuje dobijene podatke i vraća odgovor klijentu. *Važno je napomenuti da metod koji se tom prilikom izvršava može izvršiti bilo šta na korisničkom računaru (sistemsku naredbu, bilo kakav Java ili drugi kod).*

```
@Override
public void onWebSocketText(String message) {
    super.onWebSocketText(message);

    // Parse request string
    JSONRPC2Request request;

    try {
        request = JSONRPC2Request.parse(message);
    } catch (JSONRPC2ParseException e) {
        logger.error(e.getMessage());
        return;
    }

    try {
        switch (request.getMethod()) {
            case Constants.LIST:
                Util.sendWebSocketResultResponse(
                    getRemote(),
                    DeviceManager.getInstance().getDevices(),
                    request.getID());
                break;
            case Constants.FLASH:
                flash(request);
                break;
            case Constants.RUN:
                run(request);
                break;
            case Constants.INTERRUPT:
                String requestIdToInterrupt = String.valueOf(
                   request.getNamedParams().get(Constants.ID));
                Process process =
                    activeRequestRegister.get(requestIdToInterrupt);
                Util.killProcess(process);
                if (SystemUtils.IS_OS_LINUX) {
                    Util.linuxProcessCleanup(requestIdToInterrupt);
                }
                break;
            default:
                Util.sendWebSocketErrorResponse(
                    getRemote(),
                    JSONRPC2Error.METHOD_NOT_FOUND,
```

```
                        request.getID());
            }
        } catch (IOException e) {
            Util.sendWebSocketErrorResponse(
                getRemote(), JSONRPC2Error.INTERNAL_ERROR, request.getID());
        }
    }
```

Kao što se može videti iz priloženog koda, adapter koji koristi *Jetty* server omogućava *SOMANETconnectu* da se poveže sa klijentom, u ovom slučaju sa OBLAC-om, da mu prosledi spisak raspoloživih uređaja, da sačuva ili pokrene program napisan u OBLAC-u ili da zaustavi proces koji se *SOMANETconnect* trenutno izvršava. Svaki od ovih procesa vraća klijentu odgovarajući odgovor, koji može biti samo rezultat na kraju tog procesa ili ukoliko je to potrebno, *SOMANETconnect* može uživo vraćati izlaz datog procesa.

Na primer, ukoliko je OBLAC-u potreban spisak trenutno dostupnih uređaja, on će *SOMANETconnectu* poslati sledeći *JSON-RPC* zahtev:

```
{"jsonrpc": "2.0", "method": "list", "id": "4d9293ff-4b65-48ae-a020-e3f6019e2ec4"}
```

Zatim će objekat već opisane klase *OblacWebSocketAdapter* reagovati pozivom metoda *onWebSocketText* u kome će se ovaj zahtev parsirati, njegov metod odrediti i zahtev preusmeriti na granu koda koja je zadužena za obradu ovog metoda (*case Constants.LIST:*). Tada će se izvršiti metod *Util.sendWebSocketResultResponse()* koji će od objekta klase *DeviceManager* dohvatiti trenutni spisak uređaja, zapakovati ga u *JSON-RPC* odgovor i poslati nazad OBLAC-u u sledećem obliku:

```
{
   "id":"4d9293ff-4b65-48ae-a020-e3f6019e2ec4",
   "jsonrpc":"2.0",
   "result":[
      {
         "adapter_id":"O5xj4T",
         "devices":"None",
         "id":"1",
         "name":"XTAG3"
      },
      {
         "adapter_id":"P1zm9S",
         "devices":"L[0..1]",
         "id":"2",
         "name":"XTAG2"
      },
      {
         "adapter_id":"Q9ge4A",
         "devices":"L[0..3]",
         "id":"3",
         "name":"XTAG2"
      }
   ]
}
```

OBLAC zatim prima ovaj odgovor i koristi ga kako bi osvežio spisak dostupnih uređaja i prikazao ga korisniku. Može se primetiti da *JSON-RPC* zahtev i odgovor imaju identične identifikatore, kao što *JSON-RPC* protokol i propisuje.

Kada se konekcija na kraju zatvori, bilo da je do zatvaranja došlo namerno ili na neki drugi način, *onWebSocketClose* metod obrađuje ovaj događaj. On serveru pruža informacije o

načinu na koji je došlo do zatvaranja i, u ovom slučaju, osvežava status konekcije u grafičkom interfejsu aplikacije.

```
    @Override
    public void onWebSocketClose(int statusCode, String reason) {
        super.onWebSocketClose(statusCode, reason);
        logger.info("Socket Closed: [" + statusCode + "] " + reason);
        SomanetConnectSystemTray.getInstance().oblacConnected(false);
    }
```

Poslednji metod koji se nalazi u ovom adapteru obrađuje sve greške koje se mogu dogoditi u toku komunikacije. On dobija uzrok greške i zapisuje ga u dnevnik događaja.

```
    @Override
    public void onWebSocketError(Throwable cause) {
        super.onWebSocketError(cause);
        logger.error(cause);
    }

       ...
}
```

Na kraju, server se na osnovu prethodno opisanih podešavanja inicijalizuje i pokreće unutar glavne klase pod imenom *SomanetConnect*.

```
public class SomanetConnect {
    public static void main(String[] args) throws Exception {

        ...

        OblacWebSocketCreator oblacWebSocketCreator = new OblacWebSocketCreator();

        Server oblacServer =
            SomanetConnectServerFactory.createOblacServer(oblacWebSocketCreator);

        try {
            oblacServer.start();
            oblacServer.join();
        } catch (Throwable t) {
            logger.error(t.getMessage());
        }
    }
}
```

3.2 Procesi

Svaki od procesa koji *SOMANETconnect* pokreće se izvršava na korisničkom računaru, a ukoliko je to potrebno, i na priključenom ugrađenom sistemu. Kao što je već pomenuto, svaki od ovih procesa se izvršava kao rezultat zahteva koji je stigao sa OBLAC-a.

SOMANETconnect može da izvršava dve vrste procesa:

1. Procesi koji se sinhrono izvršavaju na korisničkom računaru u pozadini i isporučuju rezultat (klasa *SystemProcess*)
2. Procesi koji se asinhrono izvršavaju na korisničkom računaru u pozadini i isporučuju klijentu svaku liniju koja se pojavi na njihovom izlazu (klasa *SystemProcessLive*)

3.2.1 *SystemProcess*

Klasa *SystemProcess* se koristi kada je potrebno sinhrono izvršiti neki sistemski proces. Objekti ovog tipa pokreću proces na osnovu naredbe koja je prosleđena kroz konstruktor ove klase. Kada se proces izvrši, ova klasa omogućava dohvatanje njegovog rezultata, izlaza i eventualne greške.

Pokretanje procesa na korisničkom sistemu se vrši uz pomoć standardne *Java* klase *ProcessBuilder*. Objekti ove klase prihvataju samu naredbu i promenljive okruženja u kome će se proces izvršavati, nakon čega je pozivom metoda *start()* moguće pokrenuti takav (unapred pripremljen) proces u pozadini. Ovaj metod kao povratnu vrednost daje objekat tipa *Process*, koji u sebi sadrži mnoštvo informacija o pokrenutom procesu (rezultat, standardni izlaz, greške, i tako dalje).

```
public class SystemProcess {
    private static final Logger logger =
        Logger.getLogger(SystemProcess.class.getName());

    private int result;
    private String output;
    private String error;

    public SystemProcess(List<String> command) throws IOException {
        ProcessBuilder processBuilder = new ProcessBuilder().command(command);
        processBuilder.environment().putAll(Constants.environmentVariables);

        // Prevent more than one process from initializing at once
        SystemProcessLock.getInstance().lock();

        Process process;
        try {
            process = processBuilder.start();
        } catch (IOException e) {
            logger.error("An I/O error occurred during process executing.");
            throw e;
        }

        StreamReader outputReader = new StreamReader(process.getInputStream());
        outputReader.start();
        StreamReader errorReader = new StreamReader(process.getErrorStream());
        errorReader.start();

        try {
            result = process.waitFor();
        } catch (InterruptedException e) {
            // NO-OP
        }

        SystemProcessLock.getInstance().unlock();

        output = outputReader.getOutput();
        error = errorReader.getOutput();
    }
```

Klasa *SystemProcess* kao povratne vrednosti nudi rezultat procesa (nula ako je proces izvršen bez grešaka ili broj greške koja se desila), sam izlaz i greške ukoliko ih je bilo.

```
    public int getResult() {
        return result;
    }

    public String getOutput() {
        return output;
```

```
    }

    public String getError() {
        return error;
    }
}
```

Jedna specifičnost ovakvog načina pokretanja procesa je to što je obavezno čitati njegov izlaz koji se snima u privremenu memoriju ograničene veličine. Ukoliko se ta memorija napuni pre nego što se sam proces završi, *Java* virtualna mašina će pauzirati njegovo izvršavanje dok se taj izlaz ne isčita i privremena memorija oslobodi. Zato je potrebno pokrenuti još jedan proces u odvojenoj niti koji će čitati taj izlaz uporedo sa izvršavanjem već pomenutog procesa. U ovu svrhu se koristi interna klasa *StreamReader* koja sav izlaz procesa smešta u posebnu promenljivu iz koje se on kasnije može pročitati pozivom metoda *getOutput()*. Potpuno isti princip važi i za greške, koje se takođe moraju čitati na isti način kao i izlaz, da ne bi došlo do pauziranja izvršavanja procesa.

```
    class StreamReader extends Thread {
        InputStream is;
        String output;

        // Reads everything from the input stream until it is empty
        StreamReader(InputStream is) {
            this.is = is;
            this.output = "";
        }

        public void run() {
            try {
                InputStreamReader isr = new InputStreamReader(is);
                BufferedReader br = new BufferedReader(isr);
                String line;
                while ((line = readLineWithTerm(br)) != null) {
                    output += line;
                }
            } catch (IOException ioe) {
                ioe.printStackTrace();
            }
        }

        public String getOutput() {
            return output;
        }

        private String readLineWithTerm(BufferedReader reader) throws IOException {
            int code;
            StringBuilder line = new StringBuilder();

            while ((code = reader.read()) != -1) {
                char ch = (char) code;

                line.append(ch);

                if (ch == '\n') {
                    break;
                } else if (ch == '\r') {
                    reader.mark(1);
                    ch = (char) reader.read();

                    if (ch == '\n') {
                        line.append(ch);
                    } else {
                        reader.reset();
                    }
```

```
                    break;
                }
            }

            return (line.length() == 0 ? null : line.toString());
        }
    }
```

Ono što se može primetiti u priloženom kodu je i postojanje klase *SystemProcessLock*. Ova klasa se koristi kako bi se definisao jedinstveni objekat (deljen među svim nitima koje se trenutno izvršavaju - *singleton*) koji služi kao katanac koji zabranjuje da dve ili više niti izvršavaju kod koji sledi posle naredbe *SystemProcessLock.getInstance().lock()* u isto vreme. Ova naredba govori trenutnoj niti da počne sa izvršavanjem koda ako i samo ako neka druga nit već ne izvršava isti, a u suprotnom da sačeka da sve druge niti završe svoje izvršavanje i pozovu SystemProcessLock.*getInstance().unlock()*. Ova funkcionalnost je obavezna zbog prirode procesa koje objekti tipa *SystemProcess* izvršavaju u *SOMANETconnectu*.

```
public class SystemProcessLock extends ReentrantLock {
    private static SystemProcessLock systemProcessLock = new SystemProcessLock();

    private SystemProcessLock() {
        super();
    }

    public static SystemProcessLock getInstance() {
        return systemProcessLock;
    }
}
```

3.2.2 *SystemProcessLive*

Klasa *SystemProcessLive* je veoma slična klasi *SystemProcess*, s tim da se objekti ove klase koriste za asinhrono izvršavanje procesa i da se svaki njihov izlaz ne čuva u promenljivama, već se istog trenutka šalje do klijenta (uz pomoć *WebSocketa*) koji je inicirao izvršavanje tog procesa (u ovom slučaju OBLAC-a). Asinhrono izvršavanje omogućava da se objekti ovoga tipa izvršavaju u posebnim nitima koje su nezavisne od izvršavanja glavnog programa i samim tim ne utiču na njegovo izvršavanje.

```
public class SystemProcessLive implements Runnable {

    private static final Logger logger =
        Logger.getLogger(SystemProcessLive.class.getName());

    private List<String> command;
    private String requestId;
    private Map<String, Process> activeRequestRegister;
    private RemoteEndpoint remoteEndpoint;

    public SystemProcessLive(List<String> command,
        Map<String, Process> activeRequestRegister, String requestId,
        RemoteEndpoint remoteEndpoint) throws IOException {

        this.command = command;
        this.requestId = requestId;
        this.activeRequestRegister = activeRequestRegister;
        this.remoteEndpoint = remoteEndpoint;
    }
```

Još jedna velika razlika u odnosu na *SystemProcess* klasu je i to da klasa *SystemProcessLive* implementira standardnu *Java* klasu pod imenom *Runnable*. To znači da ova klasa mora imati implementiran metod *run()* koji omogućava *Javi* da objekat ove klase

pokrene u odvojenoj niti. Ovo je neophodno kako bi glavna nit koja pokreće samu *SOMANETconnect* aplikaciju mogla da neometano nastavi sa radom.

```
@Override
public void run() {
    ProcessBuilder processBuilder =
        new ProcessBuilder().command(command).redirectErrorStream(true);
    processBuilder.environment().putAll(Constants.environmentVariables);

    // Prevent more than one process from initializing at once
    SystemProcessLock.getInstance().lock();

    Process process;
    try {
        process = processBuilder.start();
        activeRequestRegister.put(requestId, process);
    } catch (IOException e) {
        String wholeCommand = "";
        for (String arg : command) {
            wholeCommand += arg + " ";
        }
        logger.error(e.getMessage() + " (Command: " + wholeCommand +
            +"; Request ID: " + requestId + ")");
        Util.sendWebSocketResultResponse(
            remoteEndpoint, e.getMessage(), requestId);
        Util.sendWebSocketResultResponse(
            remoteEndpoint, Constants.EXEC_DONE, requestId);
        activeRequestRegister.remove(requestId);
        return;
    }

    boolean unlocked = false;
    try {
        BufferedReader br =
           new BufferedReader(new InputStreamReader(process.getInputStream()));
        String segment;
        while ((segment = readLineWithTermAndLimit(br)) != null) {
            // Unlock the SystemProcessLock only once, when the process starts
            // its output, which means that it has
            // finished its initialization
            if (!unlocked) {
                unlocked = true;
                SystemProcessLock.getInstance().unlock();
            }
            Util.sendWebSocketResultResponse(remoteEndpoint, segment,
                requestId);
        }
    } catch (IOException e) {
        logger.error(e.getMessage());
    }

    try {
        process.waitFor();
    } catch (InterruptedException e) {
        // NO-OP
    }

    // In case that the started process didn't have any output, release the
    // lock once it finishes
    if (!unlocked) {
        SystemProcessLock.getInstance().unlock();
    }

    Util.sendWebSocketResultResponse(
        remoteEndpoint, Constants.EXEC_DONE, requestId);

    activeRequestRegister.remove(requestId);
}
```

S obzirom da postoji potreba da izlaz procesa koji se dobija na ovaj način što više liči i ponaša se poput standardnog izlaza u terminalu nekog operativnog sistema, potreban je poseban način na koji se on čita. Naime, standardni način čitanja redova teksta iz bafera u *Javi* se postiže uz pomoć metoda *readLine()* koji vraća tekst koji se nalazi između simbola koji označavaju kraj reda, ali ne i te same simbole. U ovom slučaju je potrebno vratiti i same simbole za kraj reda, bilo da su oni standardni kraj reda ili simbol za povratak na početak reda (engl. *carriage return*). U tu svrhu se koristi specijalni metod čitanja izlaza (*readLineWithTermAndLimit* metod). Na ovaj način se može postići da se aplikacija koja tumači ovaj izlaz (u ovom slučaju OBLAC) ponaša skoro identično kao i terminal, uključujući i situacije kada je potrebno pregaziti već postojeći tekst u nekom redu uz pomoć povratka na početak reda (npr. ispisivanje procenta završenog posla i njegovo konstantno izmenjivanje u istom redu ispisa na terminalu).

```
    private static String readLineWithTermAndLimit(BufferedReader reader) throws
        IOException {
        int code;
        StringBuilder segment = new StringBuilder();

        int counter = 0;
        while ((code = reader.read()) != -1) {
            char ch = (char) code;

            segment.append(ch);

            if (++counter > 25) {
                break;
            } else if (ch == '\n') {
                break;
            } else if (ch == '\r') {
                reader.mark(1);
                ch = (char) reader.read();

                if (ch == '\n') {
                    segment.append(ch);
                } else {
                    reader.reset();
                }

                break;
            }
        }

        return (segment.length() == 0 ? null : segment.toString());
    }
}
```

3.3 Upravljač uređajima

SOMANETconnect mora u svakom trenutku imati uvid u trenutno raspoložive uređaje (ugrađene sisteme) i u tu svrhu on koristi upravljač uređajima (klasa *DeviceManager*). Upravljač uređajima predstavlja klasu koja uz pomoć specijalizovane *usb4java Java* biblioteke detektuje i upravlja *USB* uređajima.

Java biblioteka *usb4java* pristupa *USB* uređajima uz pomoć *C* biblioteke (*libusb*) koja se u *Javi* izvršava preko *Java Native Interface*-a (*JNI*). S obzirom da je biblioteka napisana u *C*-u i da se izvršava preko *JNI*, ova biblioteka se može u punom ili delimičnom kapacitetu koristiti u svim poznatijim operativnim sistemima.

Upravljač uređajima predstavlja unikat klasu (engl. *singleton* - u toku izvršavanja programa, dozvoljeno je da postoji samo jedan objekat ovog tipa). Da bi ova klasa bila unikat klasa, mora se na neki način osigurati postojanje samo jednog objekta ove klase. To se najčešće postiže uz pomoć jedne privatne statičke promenljive unutar te klase koja čuva referencu na objekat koji se kreira prilikom inicijalizacije same klase. Osim ove promenljive, koristi se i statički metod *getInstance()* koji vraća referencu na tako kreirani (jedinstveni) objekat. Svaki put kada je potrebno pristupiti ovom objektu, potrebno je pozvati ovaj metod.

```
public class DeviceManager extends Observable {
    private static DeviceManager deviceManager = new DeviceManager();

    ...

    private DeviceManager() {
        this.deviceList = new DeviceList();

        // Create the libusb context
        this.usbContext = new Context();

        // Initialize the libusb context
        int result = LibUsb.init(usbContext);
        if (result == LibUsb.SUCCESS) {
            // Check if hotplug is available
            if (LibUsb.hasCapability(LibUsb.CAP_HAS_HOTPLUG)) {
                // Start the event handling thread
                usbEventHandlingThread = new UsbEventHandlingThread();
                usbEventHandlingThread.start();

                // Register the hotplug callback
                callbackHandle = new HotplugCallbackHandle();
                result = LibUsb.hotplugRegisterCallback(null,
                        LibUsb.HOTPLUG_EVENT_DEVICE_ARRIVED |
                        LibUsb.HOTPLUG_EVENT_DEVICE_LEFT,
                        LibUsb.HOTPLUG_ENUMERATE,
                        XMOS_VENDOR_ID,
                        LibUsb.HOTPLUG_MATCH_ANY,
                        LibUsb.HOTPLUG_MATCH_ANY,
                        new UsbHotplugCallback(), null, callbackHandle);
                if (result != LibUsb.SUCCESS) {
                    logger.error("Unable to register hotplug callback");
                    xmosDeviceCount = -1;
                }
            } else {
                logger.error("libusb doesn't support hotplug on this system");
                xmosDeviceCount = -1;
            }
        } else {
            logger.error("Unable to initialize libusb");
            xmosDeviceCount = -1;
        }

        if (xmosDeviceCount == -1) {
            devicePollingTimer.schedule(devicePollingTimerTask, 0, 2000);
        }
    }
    ...

    public static DeviceManager getInstance() {
        return deviceManager;
    }
}
```

U samom konstruktoru ove klase se inicira i sistem za obradu svih promena koje se dešavaju na *USB* uređajima. Svaku ovakvu promenu obrađuje podklasa pod imenom *UsbHotplugCallback*. Ova podklasa na svaku promenu koja se tiče *USB* uređaja (uključivanje

ili isključivanje iz računara) reaguje tako što pokreće odvojenu nit koja osvežava trenutni spisak raspoloživih uređaja koji je predstavljen klasom *DeviceList*.

```
class UsbHotplugCallback implements HotplugCallback {
        @Override
        public int processEvent(Context context, Device device, int event,
                                Object userData) {
            DeviceDescriptor descriptor = new DeviceDescriptor();
            int result = LibUsb.getDeviceDescriptor(device, descriptor);
            if (result != LibUsb.SUCCESS) {
                logger.error("Unable to read device descriptor");
                return result;
            }
            String status;
            if (event == LibUsb.HOTPLUG_EVENT_DEVICE_ARRIVED) {
                status = "Connected";
                if (++xmosDeviceCount == 1) {
                    devicePollingTimerTask = new TimerTask() {
                        @Override
                        public void run() {
                            timerTaskRunnable.run();
                        }
                    };
                    devicePollingTimer.schedule(devicePollingTimerTask, 0, 2000);
                }
            } else {
                status = "Disconnected";
                if (--xmosDeviceCount == 0) {
                    devicePollingTimerTask.cancel();
                    // Needs to run one more time to register that are no more
                    // devices connected
                    timerTaskRunnable.run();
                }
            }
            logger.info(status + ": " + Integer.toHexString(descriptor.idVendor())
                       + ":" + Integer.toHexString(descriptor.idProduct()));
            return 0;
        }
    }
```

U sledećem delu koda se može tačno videti kako je implementiran poziv ka nezavisnom „*xrun*” alatu, koji se nalazi u sklopu *SOMANETconnect* aplikacije, i koji na standardni izlaz ispisuje spisak dostupnih uređaja.

```
    private Timer devicePollingTimer = new Timer();
    private Runnable timerTaskRunnable = new Runnable() {
        @Override
        public void run() {
            ArrayList<String> command = new ArrayList<>();
            command.add(System.getProperty("user.dir") + "/bin/xrun");
            command.add("-l");
            SystemProcess listSystemProcess;
            try {
                listSystemProcess = new SystemProcess(command);
                if (listSystemProcess.getResult() != 0) {
                    throw new IOException(listSystemProcess.getError());
                }
            } catch (IOException e) {
                logger.error(e.getMessage());
                DeviceManager.this.setDeviceList(new DeviceList());
                return;
            }

            // Sometimes when a device is plugged/unplugged just at the right time,
            // the process output and error streams
            // are empty and the result is 0
```

```
            if (listSystemProcess.getOutput().isEmpty()
                    && listSystemProcess.getError().isEmpty()
                    && listSystemProcess.getResult() == 0) {
                logger.error("Error detecting devices");
                return;
            }

            DeviceList newDeviceList =
                new DeviceList(listSystemProcess.getOutput());
            if (!DeviceManager.this.deviceList.equals(newDeviceList)) {
                DeviceManager.this.setDeviceList(newDeviceList);
            }
        }
    };
    private TimerTask devicePollingTimerTask = new TimerTask() {
        @Override
        public void run() {
            timerTaskRunnable.run();
        }
    };
```

Klasa *DeviceList* predstavlja pomoćnu klasu koja prihvata tekstualni izlaz specijalizovanih konzolnih alata, zatim ga parsira i na osnovu toga pravi spisak raspoloživih uređaja koji su priključeni na korisnički računar.

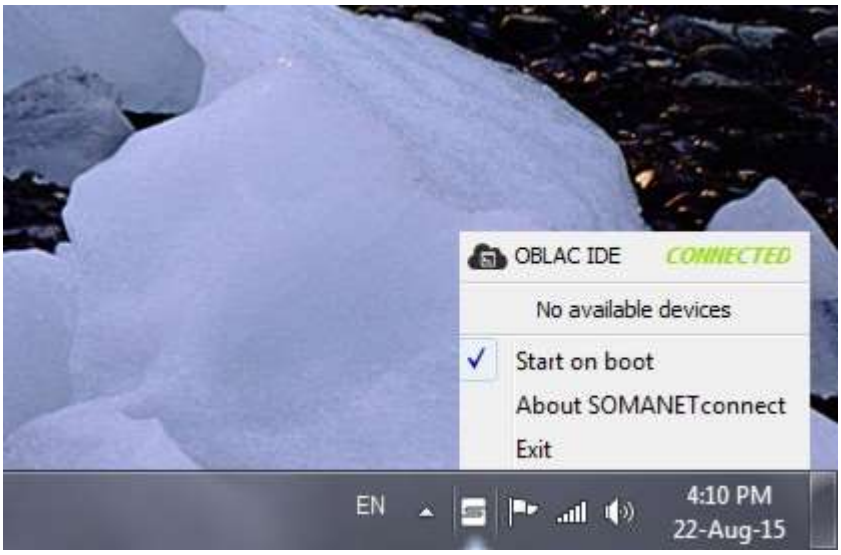
```
public class DeviceList {
    private List<Map<String, String>> devices;

    public DeviceList() {
        this.devices = new ArrayList<>();
    }

    public DeviceList(List<Map<String, String>> devices) {
        this.devices = devices;
    }

    public DeviceList(final String output) {
        devices = new ArrayList<>();
        if (!output.contains("No Available Devices Found")) {
            String tmpOutput = output;
            int marker = tmpOutput.indexOf("Available XMOS Devices");
            marker = tmpOutput.indexOf("  0 ", marker);
            if (marker == -1) {
                // There has been a problem with the XMOS tools output and it is
                // safe (?) to return an empty device list
                return;
            }
            tmpOutput = tmpOutput.substring(marker).trim();
            String[] lines = tmpOutput.split(System.getProperty("line.separator"));
            for (String line : lines) {
                String[] properties = line.trim().split("\\t");
                Map<String, String> device = new HashMap<>();
                device.put(Constants.ID, properties[0].trim());
                device.put(Constants.NAME, properties[1].trim());
                device.put(Constants.ADAPTER_ID, properties[2].trim());
                device.put(Constants.DEVICES, properties[3].trim());

                devices.add(device);
            }
        }
    }

    public List<Map<String, String>> getDevices() {
        return devices;
    }
    ...
}
```

Kako bi svi zainteresovani objekti stalno imali uvid u najsvežiju listu raspoloživih uređaja, klasa *DeviceManager* implementira *Java* klasu *Observable*, što znači da se objekat ovog tipa može dodati u spisak objekata koji posmatrač (engl. *Observer*) posmatra i reaguje na promene u njemu.

3.4 Grafički korisnički interfejs

Svaka aplikacija koja zahteva bilo kakvu interakciju sa korisnikom mora imati neku vrstu korisničkog interfejsa. Jedan od najjednostavnijih i najbržih načina na koji korisnik može da ostvari interakciju sa aplikacijom je uz pomoć grafičkog korisničkog interfejsa.

SOMANETconnect za sada nema veliki broj opcija koje korisnik direktno može koristiti ili promeniti, ali ipak ima potrebe za interakcijom sa korisnikom. Pošto je *SOMANETconnect* u suštini proces koji se izvršava u pozadini i nema mnogo opcija, sasvim je dovoljno za korisnika da postoji samo indikacija njegovog trenutnog izvršavanja zajedno sa malim menijem. Ovakva funkcionalnost se najlakše postiže kroz korišćenje trake sa pokrenutim programima (engl. *system tray*). Kao što je već spomenuto ranije, *Java AWT* omogućava dodavanje ikonice programa unutar ove trake i takođe omogućava pridruživanje jednog menija toj ikonici.



Kao što se može videti na slici, *SOMANETconnect* meni sadrži status povezanosti sa OBLAC-om, spisak raspoloživih uređaja, opciju za pokretanje *SOMANETconnecta* prilikom pokretanja operativnog sistema, opciju za otvaranje prozora sa informacijama o samom programu i opcije za izlaz.

Klasa koja se brine ovom meniju se zove *SomanetConnectSystemTray* i ona je ujedno i unikat klasa, što znači da u svakom trenutku izvršavanja ovog programa postoji tačno jedan objekat ove klase.

Prilikom samog kreiranja objekta ove klase, vrši se i provera da li je traka sa programima uopšte dostupna, zatim se u nju postavlja ikonica kojoj se pridružuje i meni. U slučaju da je traka sa programima iz nekog razloga nedostupna, program će prekinuti svoje izvršavanje i ispisati grešku.

```
public class SomanetConnectSystemTray implements Observer {
    private static SomanetConnectSystemTray somanetConnectSystemTray =
        new SomanetConnectSystemTray();

    private SomanetConnectSystemTray() {
        assertSystemTrayAvailable();

        // Set the default styling of the application to match that of the system's
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {
            logger.warn(e.getMessage());
        }

        initPopupMenu();
        initTrayIcon();

        // Set the OBLAC disconnected label as default
        setDefaultOblacStatusLabel();
    }

    public static SomanetConnectSystemTray getInstance() {
        return somanetConnectSystemTray;
    }

    ...
```

Osim toga se kreira i meni koji će biti pridružen ikonici *SOMANETconnecta* u programskoj traci. Tom prilikom se postavlja i osnovni izgled samog menija i dodaju se već spomenute opcije.

```
    private void initPopupMenu() {
        popupMenu = new JPopupMenuEx();

        popupMenu.add(setupOblacStatusPanel());
        popupMenu.addSeparator();
        JPanel loadingPanel = createLoadingPanel();
        popupMenu.add(loadingPanel);
        popupMenu.pack();
        currentDeviceMenuItems.add(loadingPanel);
        popupMenu.addSeparator();

        JCheckBoxMenuItem startOnBootItem = new JCheckBoxMenuItem("Start on boot");
        startOnBootItem.setState(Util.isStartOnBootEnabled());
        startOnBootItem.addItemListener(new ItemListener() {
            @Override
            public void itemStateChanged(ItemEvent event) {
                try {
                    Util.startOnBoot(event.getStateChange() == ItemEvent.SELECTED);
                } catch (IOException e) {
                    logger.error(e.getMessage());
                }
            }
        });
        popupMenu.add(startOnBootItem);

        JMenuItem aboutItem = new JMenuItem("About SOMANETconnect");
        aboutItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane jOptionPane = new JOptionPane();
                JDialog dialog = jOptionPane.createDialog("About");
                jOptionPane.setMessage("Synapticon SOMANETconnect v1.0");
                jOptionPane.setMessageType(JOptionPane.INFORMATION_MESSAGE);
                dialog.setSize(400, 150);
                dialog.setAlwaysOnTop(true);
                dialog.setModal(false);
                dialog.setVisible(true);
```

```
            }
        });
        popupMenu.add(aboutItem);

        JMenuItem exitItem = new JMenuItem("Exit");
        exitItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                SystemTray.getSystemTray().remove(trayIcon);
                System.exit(0);
            }
        });
        popupMenu.add(exitItem);

        popupMenu.setBorder(BorderFactory.createEmptyBorder());
    }
```

Sledeći deo koda predstavlja implementaciju već spomenutog metoda koji vrši kontrolu da li je traka sa programima dostupna. Ovaj metod se ne oslanja na jedan poziv metoda *SystemTray.isSupported()* koji zaista proverava dostupnost trake sa programima, već taj poziv upućuje nekoliko puta sa razmacima od po jednog sekunda. Razlog ovakvom ponašanju je činjenica da se prilikom podizanja nekih operativnih sistema prvo inicijalizuju programi koji se izvršavaju u pozadini, kao što je *SOMANETconnect*, a tek kasnije se inicijalizuje traka sa programima. Na ovaj način se dobija precizna informacija o dostupnosti trake, bez obzira na trenutak njene inicijalizacije.

U slučaju da trka sa programima zaista nije dostupna, informacija o tome će biti prikazana korisniku i aplikacija će prekinuti izvršavanje.

```
    private static void assertSystemTrayAvailable() {
        boolean systemTraySupported = SystemTray.isSupported();
        for (int i = 0; !systemTraySupported && i < 10; i++) {
            try {
                Thread.sleep(1000);
                systemTraySupported = SystemTray.isSupported();
            } catch (InterruptedException e) {
                // NO-OP
            }
        }

        if (!systemTraySupported) {
            JOptionPane.showMessageDialog(null, "SystemTray is not supported."
                + " SOMANETconnect will now exit.", "SOMANETconnect",
                JOptionPane.ERROR_MESSAGE);
            logger.fatal("SystemTray is not supported");
            System.exit(1);
        }
    }

    ...

}
```

Činjenica da se ovde radi o unikat klasi omogućava pristup jedinstvenom objektu ove klase iz svakog dela koda programa uz pomoć poziva metoda *SomanetConnectSystemTray.getInstance()*. Upravo se na taj način i održava indikacija povezanosti *SOMANETconnecta* sa OBLAC-om. Ukoliko se vratimo na deo koda koji obrađuje povezivanje *SOMANETconnecta* sa OBLAC-om (klasa *OblacWebSocketAdapter*), možemo videti da se upravo ovaj poziv vrši prilikom uspostavljanja i prekidanja *WebSocket* veze i da se zatim na jedinstvenom objektu tipa *SomanetConnectSystemTray* poziva metod za osvežavanje indikacije povezanosti.

Još jedan važan deo *SOMANETconnect* menija je i spisak trenutno raspoloživih uređaja. Klasa *SomanetConnectSystemTray* implementira standardnu *Java* klasu pod imenom *Observer* (srp. posmatrač). Implementacijom ove klase, klasa *SomanetConnectSystemTray* dobija mogućnost da registruje druge objekte (koji implementiraju klasu *Observable* - srp. posmatrani) i da reaguje kada na njima dođe do promene. U ovom konkretnom slučaju, *SomanetConnectSystemTray* reaguje kada se u *DeviceManager* objektu promeni trenutan spisak raspoloživih uređaja tako što odmah osvežava i spisak u samom meniju ikonice *SOMANETconnecta* u programskoj traci. U tom trenutku se pokreće odvojena nit koja dohvata spisak raspoloživih uređaja i na osnovu njega dodaje nove redove sa osnovnim informacijama tih uređaja u već pomenuti meni.

```
    private class Worker implements Runnable {
        @Override
        public void run() {
            showLoading();

            clearDeviceList();

            if (devices.isEmpty()) {
                JPanel devicePanel = new JPanel();
                JLabel noAvailableDevicesMenuItem =
                    new JLabel("No available devices");
                devicePanel.add(noAvailableDevicesMenuItem);
                popupMenu.insert(devicePanel, 2);
                currentDeviceMenuItems.add(devicePanel);
            } else {
                for (Map<String, String> device : devices) {
                    addDeviceToList(device);
                }
            }

            if (popupMenu.isVisible()) {
                // Resize the ancestor window of the popup menu to the required
                // size. The popupMenu.pack() is not used
                // because it causes the popup menu to flicker.
                Window window = SwingUtilities.getWindowAncestor(popupMenu);
                if (window != null) {
                    window.pack();
                    window.validate();
                }
                // Fix the popup menu location according to its new size
                popupMenu.setLocation(lastMouseClickPosition);
            }
            popupMenu.setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
        }
    }
```

Poslednja značajna opcija u meniju *SOMANETconnecta* je opcija automatsko za pokretanje aplikacije prilikom pokretanja operativnog sistema koja se na svakom od poznatijih operativnih sistema drugačije izvršava.

Ova opcija se na *Linux* operativnim sistemima omogućava tako što se kreira specijalna vrsta datoteke (*desktop* datoteka) unutar za to predviđenog direktorijuma. Ova datoteka sadrži putanju do aplikacije i sve ostale neophodne parametre kako bi se aplikacija uspešno pokrenula.

```
     public static void startOnBoot(boolean startOnBoot) throws IOException {
        if (SystemUtils.IS_OS_LINUX) {
            Path autoStartDir = Paths.get(System.getenv("HOME"), ".config",
                                          "autostart");
            if (!Files.exists(autoStartDir)) {
```

```
            Files.createDirectories(autoStartDir);
        }
        Path desktopFile = Paths.get(autoStartDir.toString(),
                                     "SOMANETconnect.desktop");
        if (startOnBoot) {
            String applicationJarPath = System.getProperty("user.dir") +
                                        "/SOMANETconnect.jar";
            String libPath = System.getProperty("user.dir") + "/lib";
            String command = "java -Djava.library.path=" + libPath + " -cp " +
                    + applicationJarPath
                    + " SOMANETconnect.SomanetConnect";
            String desktopFileContent = "[Desktop " +
                    + "Entry]\nType=Application\nName=SOMANETconnect\nPath="
                    + System.getProperty("user.dir") + "\nExec=" + command +
                    + "\n";
            Files.write(desktopFile, desktopFileContent.getBytes());
        } else {
            Files.deleteIfExists(desktopFile);
        }
```

Kako bi se *SOMANETconnect* automatski pokrenuo pri pokretanju *Windowsa*, ova opcija u pozadini mora da izmeni njegov registar ključeva (engl. *key registry*), uz pomoć za to specijalizovane biblioteke, i da doda ključ koji sadrži putanju do same aplikacije na unapred određenu lokaciju za automatsko pokretanje prilikom pokretanja operativnog sistema.

```
    } else if (SystemUtils.IS_OS_WINDOWS) {
        RegKeyManager rkm = new RegKeyManager();
        try {
            if (startOnBoot) {
               rkm.add("HKCU\\Software\\Microsoft\\Windows" +
                       + "\\CurrentVersion\\Run",
                       "SOMANETconnect", "REG_SZ",
                       System.getProperty("user.dir") + "\\start.vbs");
            } else {
                rkm.delete("HKCU\\Software\\Microsoft\\Windows" +
                           + "\\CurrentVersion\\Run", "SOMANETconnect");
            }
        } catch (Exception e) {
            logger.error(e.getMessage());
        }
```

Automatsko pokretanje aplikaciju u *OS X* operativnom sistemu se ostvaruje na sličan način kao i u *Linuxu*. Razlika je u tome što se u ovom slučaju datoteka naziva *launch agent* i zapisana je u *XML* formatu.

```
    } else if (SystemUtils.IS_OS_MAC_OSX) {
        Path launchAgentsDir = Paths.get(System.getenv("HOME"), "Library",
                                         "LaunchAgents");
        if (!Files.exists(launchAgentsDir)) {
            Files.createDirectories(launchAgentsDir);
        }
        Path plistFile = Paths.get(launchAgentsDir.toString(),
                                   "com.synapticon.somanetconnect.plist");
        if (startOnBoot) {
            String applicationPath = System.getProperty("user.dir") +
                                     + "/../MacOS/SOMANETconnect";
            String plistFileContent = "<?xml version=\"1.0\" +
                    + "encoding=\"UTF-8\"?>\n" +
                    "<!DOCTYPE plist PUBLIC \"-//Apple//DTD PLIST 1.0" +
                    + "//EN\" \"http://www.apple.com/DTDs/" +
                    + "PropertyList-1.0.dtd\">\n" +
                    "<plist version=\"1.0\">\n" +
                    "<dict>\n" +
                    "    <key>Label</key>\n" +
                    "    <string>com.synapticon.somanetconnect</string>\n" +
                    "    <key>ProgramArguments</key>\n" +
```

```
                        "    <array>\n" +
                        "        <string>" + applicationPath + "</string>\n" +
                        "    </array>\n" +
                        "    <key>ProcessType</key>\n" +
                        "    <string>Background</string>\n" +
                        "    <key>RunAtLoad</key>\n" +
                        "    <true/>\n" +
                        "    <key>KeepAlive</key>\n" +
                        "    <false/>\n" +
                        "</dict>\n" +
                        "</plist>";
                Files.write(plistFile, plistFileContent.getBytes());
            } else {
                Files.deleteIfExists(plistFile);
            }
        }
    }
```

4 ZAKLJUČAK

U radu se pokazuje kako je moguće povezati veb aplikacije i računar na kome se one izvršavaju na univerzalan način koji takođe dozvoljava maksimalnu fleksibilnost. Univerzalnost i fleksibilnost ove arhitekture potiču od kombinacije *WebSocketa* i *Java* aplikacije, koje omogućavaju da se ovakav pristup koristi na svakom računaru koji ima internet pregledač novije generacije i podržava *Java* virtualnu mašinu.

Opisana je arhitektura, dizajn i najvažniji delovi implementacije aplikacije *SOMANETconnect*, koja omogućava razvojnoj platformi OBLAC da koristi sve resurse nekog računara, kao i uređaje koji su na njega priključeni. Način na koji je ova aplikacija zamišljena i implementirana omogućava jednostavno dodavanje novih funkcionalnosti i potencijalno omogućava razvijenoj aplikaciji *SOMANETconnect* da postane univerzalni most između bilo kog broja veb aplikacija i računara koji izvršava te veb aplikacije.

Ono što je dobra strana celokupnog pristupa je da je na lokalnom računaru moguće izvršiti bilo koju naredbu koja je dostupna uz pomoć terminala, jer terminal predstavlja najmoćnije oruđe u jednom operativnom sistemu. Svaka akcija ili naredba koju veb aplikacija pošalje na serversku stranu (u ovom slučaju *SOMANETconnect*) će prvenstveno biti proverena, a zatim i izvršena ukoliko je provera bila uspešna. Ovakav sistem sprečava bilo kakve zloupotrebe, jer veb aplikacija može izvršiti samo one metode koje *SOMANETconnect* ima implementirane.

Što se budućnosti samog *SOMANETconnecta* tiče, već se radi na tome da ova aplikacija dobije neku vrstu programskih dodataka koji bi joj omogućili da komunicira i sa drugim veb aplikacijama. Za početak, plan je da se jednim od programskih dodataka omogući podešavanje i precizno nadgledanje performansi motora kontrolisanih od strane ugrađenih sistema, iz specijalizovane veb aplikacije.

5 LITERATURA

1. **International Telecommunication Union.** ITU-T Recommendations - Overview of the Internet of things. [Na mreži] 2012. http://www.itu.int/. Y.2060.

2. **Heath, Steve.** *Embedded Systems Design.* s.l. : Newnes, 2002. 0080477569.

3. **Barr, Michael / Massa, Anthony.** *Programming Embedded Systems: With C and GNU Development Tools.* s.l. : O'Reilly Media, Inc., 2006. 0596553285.

4. **Synapticon.** OBLAC Tools. *Synapticon – Cyber-Physical Systems.* [Na mreži] 2015. https://www.synapticon.com/products/oblac-tools/.

5. **Google Inc.** Sandbox. *The Chromium Projects.* [Na mreži] http://dev.chromium.org/developers/design-documents/sandbox.

6. NPAPI. *Wikipedia.* [Na mreži] https://en.wikipedia.org/wiki/NPAPI.

7. **Google Inc.** NPAPI Plugins. *Chrome Developer.* [Na mreži] https://developer.chrome.com/extensions/npapi.

8. **Mozilla.** Plugins. *Mozilla | MDN.* [Na mreži] https://developer.mozilla.org/en-US/Add-ons/Plugins.

9. **Google Inc.** What Are Chrome Apps. *Google Chrome.* [Na mreži] https://developer.chrome.com/apps/about_apps.

10. **TIOBE Software.** TIOBE Index for September 2015. *TIOBE Software: Tiobe Index.* [Na mreži] 9 2015. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

11. Java (programming language). *Wikipedia.* [Na mreži] https://en.wikipedia.org/wiki/Java_(programming_language).

12. **Liang, Sheng.** *The Java Native Interface.* s.l. : Addison-Wesley, 1999. 0-201-32577-2.

13. **O'Brien, Tim, i drugi.** *Maven: The Complete Reference.* s.l. : Sonatype Inc., 2008.

14. Hypertext Transfer Protocol. *Wikipedia.* [Na mreži] https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol.

15. **Wang, Vanessa, Salim, Frank / Moskovits, Peter.** *The Definitive Guide to HTML5 WebSocket.* s.l. : Apress, 2013. 1430247401.

16. Introducing JSON. *JSON.* [Na mreži] http://www.json.org/.

17. **Morley, Matt.** JSON-RPC 2.0 Specification. *JSON-RPC.* [Na mreži] http://www.jsonrpc.org/specification.