Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Andolšek

Razširitev platforme interneta stvari OM2M za potrebe hranjenja velikih količin podatkov

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: prof. dr. Marko Bajec

Ljubljana, 2017

Avtorske pravice. Rezultati magistrskega dela so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov magistrskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

©2017 Matej Andolšek

Zahvala

Rad bi se zahvalil prof. dr. Marku Bajcu, ki mi je pomagal s strokovnimi nasveti pri izdelavi magistrske naloge. Posebna zahvala gre tudi staršem, ki so mi omogočili študij na fakulteti za računalništvo in informatiko in me pri tem ves čas podpirali. Zahvalil bi se tudi Neži za pomoč pri izdelavi magistrske naloge.

Matej Andolšek, 2017

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
ADN	Application Dedicated Node	aplikacijsko namensko vozlišče
AE	Application Entity	aplikacijska entiteta
ASN	Application Service Node	aplikacijsko storitveno vozlišče
BLOB	Binary Large OBject	zbirka binarnih podatkov
CSE	Common Service Entity	entiteta skupnih storitev
CSF	Common Service Function	funkcije skupnih storitev
HTTP	Hyper Text Transport Protocol	protokol za izmenjavo hiperteksta
IN	Infrastructure Node	infrastrukturno vozlišče
IoT	Internet of Things	internet stvari
JDBC	Java DataBase Connectivity	standard za povezovanje s pod. bazo
JSON	JavaScript Object Notation	JavaScript objektna notacija
M2M	Machine to Machine	komunikacija med napravami
MN	Middle Node	vmesno vozlišče
NFC	Near Field Communication	komunikacija kratkega dosega
NoSQL	Not Only SQL database	nerelacijska podatkovna baza
NSE	Network Services Entity	entiteta mrežnih storitev
POM	Project Object Model	konfiguracijska datoteka za Maven
RFID	Radio-Frequency IDentification	radiofrekvenčno prepoznavanje
SCL	Service Capability Layer	storitveni vmesnik
SQL	Structured Query Language	jezik za delo s pod. bazami
XML	Extensible Markup Language	razširljivi označevalni jezik

Povzetek

Naslov: Razširitev platforme interneta stvari OM2M za potrebe hranjenja velikih količin podatkov

V magistrski nalogi smo se osredotočili na internet stvari ter komunikacijo med napravami. Eden od izzivov interneta stvari, kjer je lahko med seboj povezanih neomejeno število naprav, je obvladovanje velikih količin podatkov, ki jih te naprave generirajo. V ta namen so bile razvite številne IoT platforme, ki delujejo kot povezovalnik med napravami. Obenem lahko podatke prikazujejo, hranijo in obdelujejo.

V magistrski nalogi smo za shranjevanje podatkov uporabili odprtokodno platformo OM2M, ki je razvita v programskem jeziku Java in temelji na razširljivi platformi OSGi. Rešitev OM2M za svoje delovanje in shranjevanje podatkov uporablja relacijsko podatkovno bazo H2, ki pa zaradi pomanjkljivosti ni primerna za našo uporabo. Zato smo v sklopu magistrske naloge razvili OSGi vtičnik, ki omogoča uporabo poljubne nerelacijske podatkovne baze, in pri tem implementirali podporo nerelacijski podatkovni bazi MongoDB.

Skozi uporabo platforme smo opazili, da ima spletni vmesnik platforme probleme pri prikazu velikih količin podatkov. V ta namen smo na spletni vmesnik dodali paginacijo. Preverili smo delovanje paginacije in odzivnost spletnega vmesnika. Izkazalo se je, da je rešitev pripomogla k hitrosti prikaza podatkov. Izvedli smo testiranje ustreznosti novo razvitega vtičnika za pisanje v nerelacijsko podatkovno bazo in pokazali, da je rešitev pripomogla k hitrejšem zapisovanju podatkov.

Ključne besede

internet stvari, komunikacija med napravami, OM2M, hranjenje velikih količin podatkov, nerelacijske baze

Abstract

Title: Extending the internet of things platform OM2M for the purpose of storing large amounts of data

In the master's thesis, we focused on the Internet of Things and communication between devices. One of the challenges of the Internet of Things, where unlimited number of devices can be interconnected, is the control of large amounts of data generated by these devices. For this purpose, a number of IoT platforms have been developed. IoT platforms act as a link between multiple devices. They can also display, store and process data.

For storing data we used open source platform OM2M, which is developed in Java programming language. OM2M is based on extendable OSGi platform and uses a relational H2 database for its operation and storage, which, however, is not suitable for our use due to deficiencies. For that reason, we developed OSGi plugin which allow us to use any non-relational database. We implemented writing and reading data from/to non-relational database MongoDB.

Through the use of the platform, we noticed that the platform's web interface has problems in displaying large amounts of data so we added a pagination to the web interface. After implementation we tested our developed solution. It turned out that the solution helped with the speed of displaying data. We also performed the testing of the suitability of a newly developed plug-in for writing data to non-relational database MongoDB. In the end we have proven that the solution has contributed to faster data writing.

Keywords

Internet of things, machine to machine, OM2M, storing large amount of data, non-relational database

Poglavje 1

Uvod

Da razumemo pomembnost interneta stvari, je treba poznati razliko med internetom in svetovnim spletom. Internet je fizični sloj oz. omrežje, ki ga sestavljajo stikala, usmerjevalniki in druga oprema. Njegova glavna naloga je, da je transport podatkov iz ene točke v drugo hiter, zanesljiv in varen. Splet pa predstavlja plast aplikacij, ki delujejo na vrhu interneta (spletni brskalniki itd.).

V današnjem času je na svetu prisotnih veliko število najrazličnejših naprav, od tehnoloških naprav do gospodinjskih aparatov, ki so povezane v internet. Vse te naprave, ki imajo dostop do internetne povezave in imajo programsko opremo, se povezujejo v mrežo fizičnih objektov, imenovano internet stvari (*ang. Internet of things* - IoT) [1]. Internet stvari je danes široko razširjena tema, trendi pa nakazujejo, da bo v prihodnjih letih doživela še večji porast [2].

Glede na Cisco Internet Business Solutions Group (IBGS) [3] je bilo pred letom 2003 zabeleženih manj kot 0,08 naprav na prebivalca. Vzpon števila naprav se je začel po letu 2003, ko je število naprav preseglo število prebivalcev. V letu 2010 je bilo tako zabeleženih že 1,84 naprav/prebivalca, kar je posledica povečanja števila pametnih telefonov in tablic. Internet stvari naj bi bil tako "rojen" nekje med letoma 2008 in 2009. Cisco IBGS napoveduje, da bo v letu 2020 število naprav narastlo že na 6,58 naprav/prebivalca.

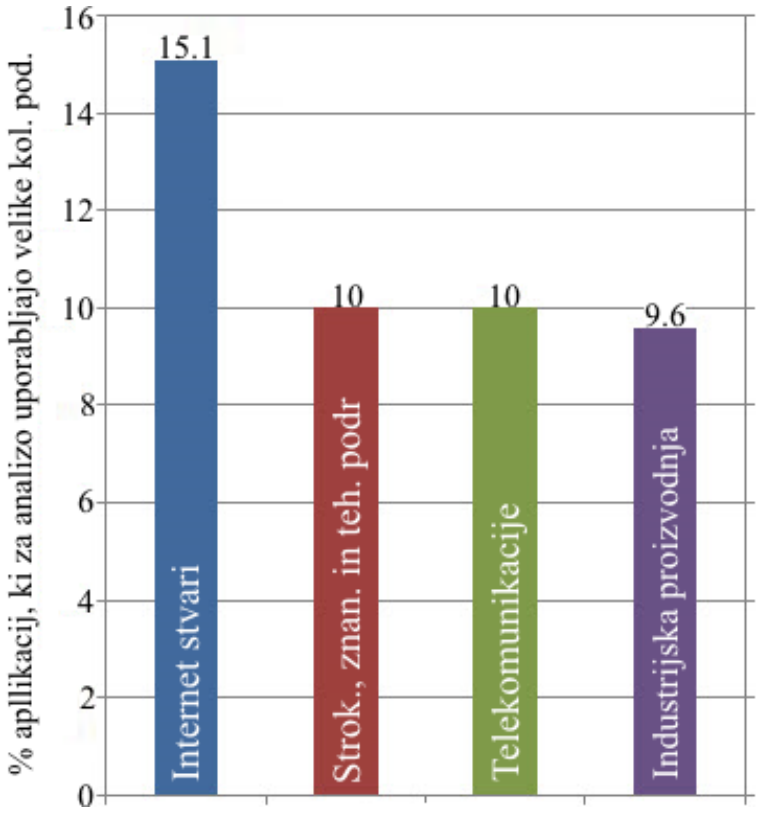
Število povezanih naprav na prebivalca se mogoče zdi nizko, vendar je to posledica tega, da so v statistiko vključili celotno prebivalstvo (od tega jih velik del še ni povezanih v internet).

Za vse naprave je značilno, da imajo svoje unikatne identifikatorje ter sposobnost pošiljanja podatkov preko omrežja. Naprave morajo biti sposobne prenosa oz. pošiljanja podatkov preko omrežja brez človeške ali računalniške interakcije. Vse naprave, ki so zmožne pošiljati podatke preko omrežja, morajo imeti svoj lasten IP naslov. IP naslovi so neke vrste imena, ki so sestavljena iz številk oz. črk in pripadajo vsaki napravi, ki se pojavlja v internetnem omrežju. Pomembno je, da je vsak naslov unikaten, saj lahko le tako enolično določimo, za katero napravo gre. Ravno IP naslovi lahko upočasnijo rast interneta stvari. Poznamo dve vrsti IP protokola, in sicer IPv4 [4] in IPv6 [5]. Protokol IPv6 je nadgradnja obstoječega protokola IPv4 predvsem zaradi pomanjkanja IP naslovov v svetu internetnih omrežij. Največja razlika med IPv4 in IPv6 je ta, da so IPv6 naslovi 128 bitni, medtem ko so IPv4 32 bitni. Ravno ta sprememba je glavni razlog za hitro razvijanje interneta stvari, saj lahko sedaj praktično vsaka naprava dobi svoj IP naslov. Pri nadaljnji uporabi IPv4 bi bilo to nemogoče, saj bi jih prehitro zmanjkalo. V primeru, da bi vsakemu atomu na zemlji dodelili svoj IPv6 naslov, nam bi jih še vedno ostalo za več kot 100 zemelj. IPv4 tako omogoča 2^{32} (4.294.967.296) naslovov, medtem ko IPv6 omogoča 2^{128} (pribl. $3.4 * 10^{38}$) naslovov.

Trenutno je internet stvari sestavljen iz zbirke namenskih mrež. Na primer, današnji avtomobili imajo več omrežij za nadzor delovanja motorja, varnostnih funkcij, komunikacijskih sistemov itd. Različne nadzorne sisteme za ogrevanje, prezračevanje in klimatizacijo uporabljajo tudi poslovne in stanovanjske stavbe. Poleg tega najdemo naprave, ki uporabljajo internet stvari tudi v zdravstvu, energiji (razsvetljava), transportu, gradbeništvu, telekomunikacijah in drugje.

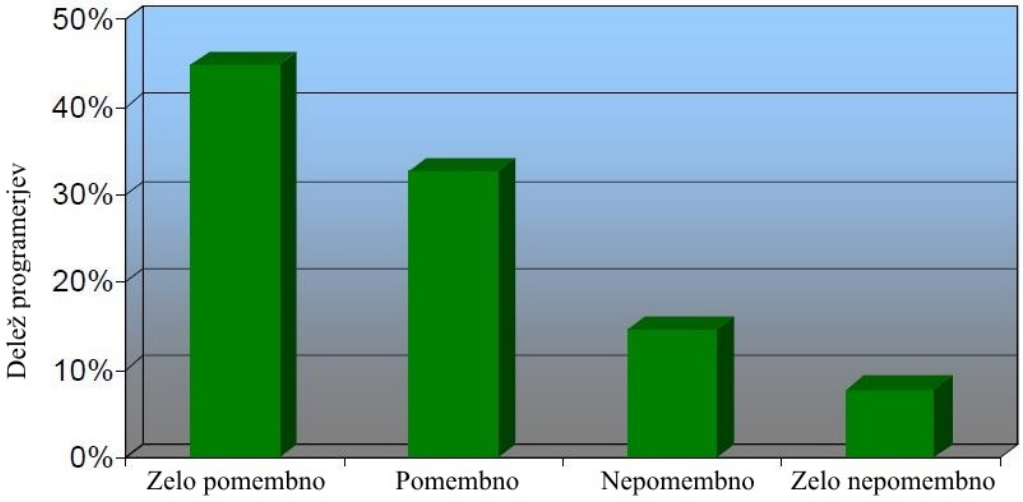
Zanimivi so tudi izsledki spletne ankete o razvijanju aplikacij, ki jo je podjetje Evans Data Corporation (EDC) poslalo 1441 razvijalcem [6]. Anketa je temeljila na razgovorih z razvijalci, ki aktivno sodelujejo pri razvijanju no-

vih aplikacij z najnovejšimi tehnologijami. Eno izmed vprašanj se je glasilo, na katero industrijo bodo ciljale aplikacije, ki za analizo uporabljajo velike količine podatkov.



Slika 1.1: Na katero industrijo bodo ciljale aplikacije, ki za analizo uporabljajo velike količine podatkov

Kot lahko vidimo na sliki 1.1, s 15,3% prepričljivo vodi internet stvari. Sledijo mu strokovna, znanstvena in tehnična področja (10%) ter telekomunikacije (10%). Na četrtem mestu je industrijska proizvodnja, ki ni povezana z računalniškimi storitvami. Eno izmed vprašanj je bilo tudi, kako razvijalci vidijo pomembnost interneta stvari. Iz slike 1.2 lahko razberemo, da skoraj polovica vseh vprašanih razvijalcev vidi internet stvari kot pomembno digitalno strategijo.



Slika 1.2: Kako razvijalci vidijo pomembnost interneta stvari

1.1 Opis problema in rešitve

Pri internetu stvari je ena izmed ovir zagotavljanje interoperabilnosti v okviru IoT scenarijev, ki povezujejo veliko število senzorjev, naprav in oblačnih storitev. Omenjeni scenariji zahtevajo veliko stopnjo interoperabilnosti, ki je zaradi pomanjkanja standardizacije oziroma predvsem neupoštevanja standardov ni možno zagotavljati. Ena redkih odprtokodnih IoT platform, ki sledi enemu izmed najpopularnejših IoT standardov (oneM2M), je Eclipse OM2M. Žal ima omenjena platforma tudi določene omejitve, saj pri svojem delovanju uporablja interno bazo H2. H2 je relacijska podatkovna baza, ki ne zmore obdelave velikih količin podatkov. Ravno zaradi težave pri obdelavi velikih količin podatkov in hitrosti smo se odločili, da izdelamo nov vtičnik, ki bo omogočal komunikacijo z NoSQL podatkovnimi bazami. Pri implementaciji smo se osredotočili na podatkovno bazo MongoDB, na podoben način pa bi bilo moč podpreti tudi drugo poljubno podatkovno bazo.

1.2 Pristop k delu

Najprej smo se seznanili s teoretičnim delom M2M, IoT in platforme OM2M ter preučili standarda ETSI-M2M in oneM2M. Na testni sistem smo namestili osnovno verzijo platforme ter se seznanili z njenim delovanjem. Testni sistem je sestavljala naslednja strojna in programska oprema: procesorska enota Intel i5, 256 GB velik podatkovni disk, 4 GB sistemskega pomnilnika ter operacijski sistem Windows 7. Pregledali smo obstoječe IoT rešitve in možnosti za nadgradnjo platforme OM2M tako, da bi pri svojem delovanju uporabljala podatkovno bazo MongoDB. Sklope, ki so zadolženi za pisanje in branje v oz. iz podatkovne baze, smo podrobneje preučili in implementirali nov vtičnik. Med samim razvojem smo odkrili določene pomanjkljivosti platforme, ki smo jih kasneje odpravili oz. izboljšali. Poiskali smo orodja za testiranje in preverili njihovo ustreznost. Odločili smo se, da bomo razvili svoje orodje, ki smo ga uporabili pri testiranju ustreznosti novo razvitih funkcionalnosti.

1.3 Pregled strukture dela

Na začetku dela smo v poglavju 2 predstavili standarde in arhitekturo, ki jih uporablja platforma OM2M. V poglavju 3 nadaljujemo z natančnim opisom uporabljene platforme OM2M in modularno platformo OSGi, ki se uporablja pri delovanju platforme in njenih razširitev. Nadaljujemo s prikazom uporabe platforme, nato pa v poglavju 4 opišemo vse novo razvite razširitve. Poglavje 5 obsega testiranje novih funkcionalnosti. Delo zaključimo s sklepnimi ugotovitvami.

Poglavje 2

Standardi in arhitektura

V današnjem času je velika množica računalnikov med seboj povezanih v omrežje. Zato, da lahko računalniki med seboj komunicirajo brez težav, so se uvedli standardi. Računalniki pri komuniciranju uporabljajo računalniški jezik, ki je strukturiran v obliki dogovorjenih pravil in postopkov, ki jim pravimo protokoli. V primeru, da računalnika uporabljata različne protokole, je potreben posrednik, ki protokole ustrezno pretvori. Standardi so dokumenti, ki vsebujejo tehnične specifikacije o načrtovanju omrežja oz. uradno sprejeti protokoli, ki jih razumejo vsi uporabniki. Omrežja postanejo z upoštevanjem standardov zanesljiva in učinkovita.

Poznamo industrijske (*de facto*) in pravne (*de iure*) standarde [31]. Industrijske standarde razvijajo različni proizvajalci brez formalnega načrtovanja. Za njih je značilno, da se zaradi široke in enostavne uporabe hitro uveljavijo oz. v določenih primerih tudi propadejo. Pravni standardi pa nastanejo v okviru mednarodnih organizacij za standardizacijo v skladu z zakonom ali predpisom. Ti standardi so razviti z ustreznimi raziskavami. Vodilne organizacije za razvoj komunikacijskih protokolov in standardov so ISO (International Organization for Standards), IEEE (Institute of Electrical and Electronics Engineers), ANSI (American National Standards Institute), ETSI (European Telecommunications Standards Institute), EIA (Electronic Industries Association) itd.

2.1 Komunikacija med napravami (M2M)

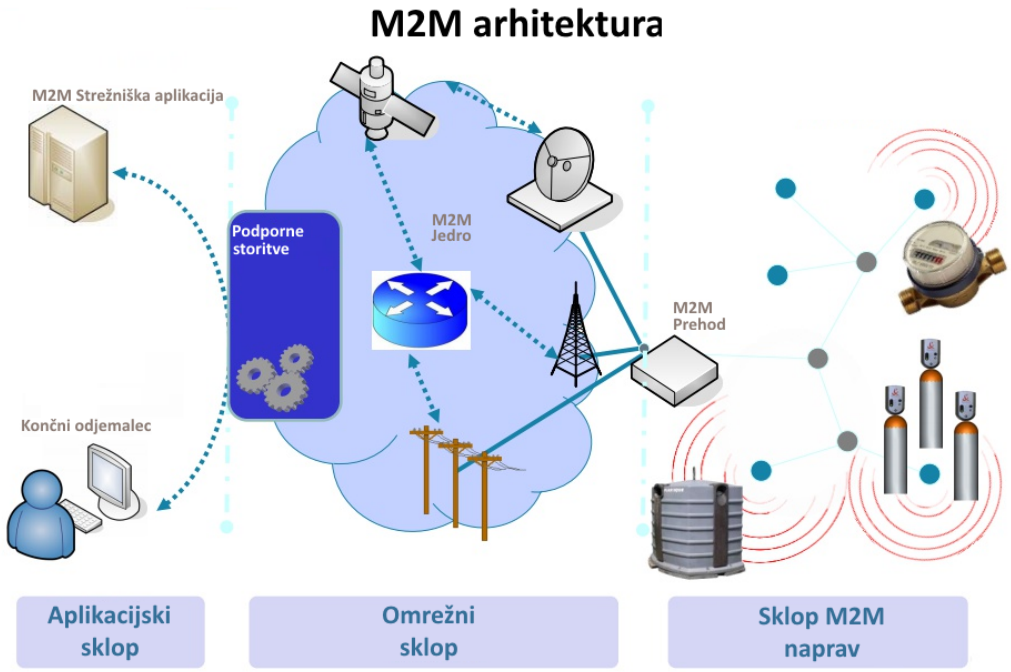
Kadar govorimo o pojmu komunikacije med napravami (*ang. Machine to machine* – M2M) [7], mislimo na vsaj dve napravi (lahko tudi več), ki sta zmožni medsebojno komunicirati brez človeškega poseganja. Komunikacija poteka tako po fizičnih kot tudi brezžičnih omrežjih, kar omogoča bolj vsestransko in preprosto uporabo. M2M omogoča povezavo več milijonov naprav v eno omrežje. V omrežju so lahko priključene različne naprave, in sicer od tipal, inteligentnih sistemov, RFID/NFC čipov, gospodinjskih aparatov do različnih vozil. Možnosti so tako rekoč neskončne.

V grobem so M2M omrežja zelo podobna prostranim in lokalnim računalniškim omrežjem le, da so namenjena samo za komunikacijo med napravami in senzorji. Naprave sprejemajo in oddajajo podatke, do katerih lahko dostopamo s pomočjo pametnih kontrolnih naprav. Pri tem je zaželeno upoštevati standarde, kot je npr. ETSI [8]. Slika 2.1 prikazuje M2M arhitekturo.

M2M arhitektura je sestavljena iz treh glavnih sklopov [9] [10]:

- **Sklop M2M naprav** združuje pametna tipala ter naprave, ki omogočajo zbiranje podatkov (temperaturni senzor, senzor srčnega utripa, senzor tlaka itd.). Poznamo dva tipa naprav. Naprave, ki so sposobne neposredne povezave z omrežjem in naprave, ki za povezavo potrebujejo M2M prehod. M2M prehod skrbi za komunikacijo med napravami in omrežnim sklopom.

- **Omrežni sklop** skrbi za povezavo med aplikacijskim sklopom in sklopom M2M naprav. Sestavljen je iz M2M jedra in podpornih storitev. M2M jedro zagotavlja povezljivost preko različnih tehnologij, kot so WiFi, mobilno omrežje (2G, 3G, LTE), satelitsko omrežje itd. Podporne storitve zagotavljajo funkcionalnosti, ki so pogosto uporabljene, in sicer aktivacija naprave, nadzor nad komunikacijo, upravljanje z varnostno politiko, nadzor nad podatkovnimi shrambami, upravljanje z napravo itd. Pri tem je celotna programska logika skrita uporabniku, kar omogoča preprosto uporabo pri implementaciji in razvoju.



Slika 2.1: M2M arhitektura

- **Aplikacijski sklop** vsebuje strežniške aplikacije in aplikacije, ki so namenjene končnemu odjemalcu. Aplikacije pri svojem delovanju uporabljajo podporne storitve omrežnega sklopa. Strežniške aplikacije so namenjene interakciji z napravami, medtem ko so aplikacije namenjene končnemu odjemalcu in mu omogočajo razne analize podatkov, izdelavo poročil itd.

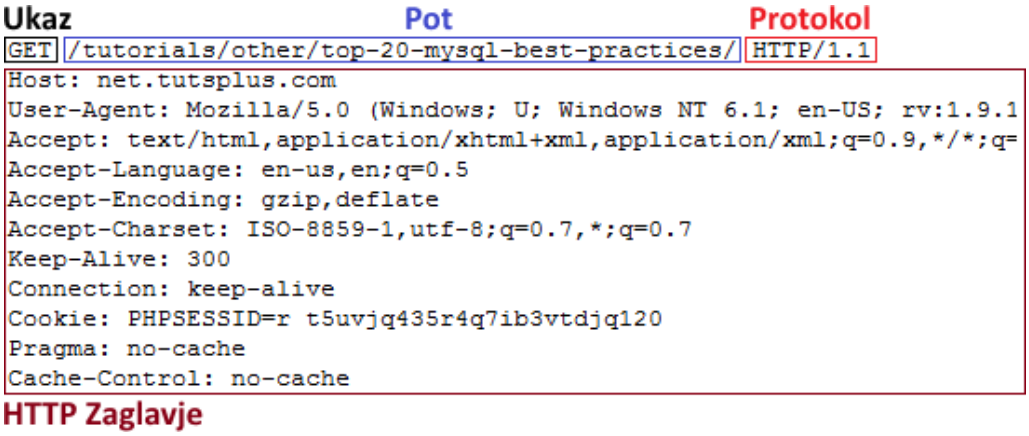
2.1.1 M2M komunikacijski protokoli

M2M komunikacijski protokoli imajo eno izmed ključnih vlog pri zagotavljanju učinkovite komunikacije. Izbran protokol mora biti čim bolj optimalen (zaglavje paketa, število kontrolnih paketov, zanesljivost itd.), saj to vpliva na porabo pasovne širine in sistemskih virov.

Leta 2014 so na 10. plenarnem zasedanju v Berlinu [11] projektni par-

tnerji oneOM2M prišli do dogovora o uporabi komunikacijskih protokolov HTTP, CoAP in MQTT kot *de facto*.

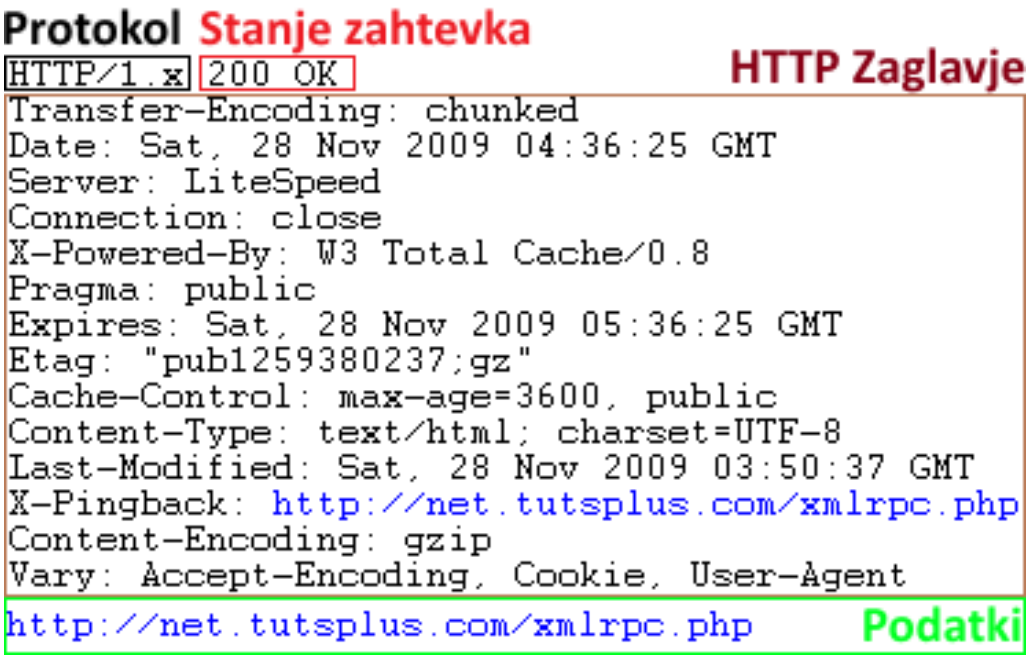
HTTP [12] je protokol, ki je temelj za komuniciranje preko svetovnega spleta. Deluje po principu zahteva-odgovor. Klient pošlje zahtevo na oddaljen strežnik, ki zahtevo izpolni, nato pa strežnik pošlje odgovor na to zahtevo. Tipičen primer je dostop do spletnih strani. Klient s pomočjo spletnega brskalnika pošlje zahtevo na strežnik, ki gostuje spletno stran.



Slika 2.2: Struktura HTTP zahtevka

Vsak HTTP zahtevek (slika 2.2) je sestavljen iz štirih pomembnih sklopov. Sklop *"ukaz"* predstavlja tip zahtevka, *"pot"* predstavlja del za imenom gostitelja, *"protokol"* označuje vrsto/verzijo protokola in *"http zaglavje"*, ki predstavlja dodatne (lahko opcijske) informacije o zahtevku npr.: User-Agent (informacije o spletnem brskalniku ter operacijskem sistemu), Accept-Encoding (podatek o tem, ali je klient zmožen sprejemati stisnjene podatke) itd. HTTP zahtevek mora vsebovati vsaj prvo vrstico (ukaz, pot, protokol) in informacijo *"Host"*. Strežnik sprejme HTTP zahtevek, izvede določene ukaze, nato pa klientu posreduje HTTP odgovor (slika 2.3).

Struktura HTTP odgovora je sestavljena podobno kot HTTP zahtevek. V prvi vrsti je namesto kombinacije ukaza, poti in protokola samo podatek o protokolu ter stanju zahtevka. Tako kot pri zahtevku tudi tukaj zaglavje



Slika 2.3: Struktura HTTP odgovora

vsebuje dodatne informacije npr.: vrsta programske opreme na strežniku, čas zadnje spremembe, vrsto vsebine, nabor znakov itd. Glavna razlika med strukturo odgovora in zahtevka je ta, da odgovor vsebuje tudi zahtevane podatke, ki jih je vrnil strežnik.

Protokol pri svojem delovanju ne beleži nobenega stanja. Predstavljajmo si, da dostopamo do podatkov, ki so zaščiteni z geslom. Glede na to, da protokol ne beleži stanja, je treba ob vsaki zahtevi za podatke, posredovati tudi podatke o dostopu (uporabniško ime in geslo). Ob uporabi beleženja stanja to ne bi bilo treba. Poznamo tudi različico HTTPS, ki pri pošiljanju podatkov uporablja šifriranje podatkov s pomočjo TLS/SSL [13].

CoAP [14] je protokol, ki je tako kot HTTP namenjen prenosu podatkov. Za razliko od protokola HTTP je CoAP zasnovan za uporabo v okoljih, kjer je na voljo zelo malo sistemskih virov. Velikosti CoAP paketov so bistveno manjše kot pri HTTP protokolu. Preprosta zasnova in oblika paketov omogoča preprosto razčlenitev in generiranje paketov. Predvideno je, da

brez težav deluje tudi na mikrokrmilnikih z 10 kB sistemskega pomnilnika in 100 kB prostora za programsko kodo. CoAP pri svojem delovanju uporablja protokol UDP [15] za razliko od protokola HTTP, ki uporablja TCP [16]. Bistvene razlike med protokoloma so prikazane v tabeli 2.1.

Tabela 2.1: Primerjava protokolov TCP in UDP

	TCP	UDP
Glavna lastnost	Zanesljivost	Hitrost
Uporaba	HTTP/S, S/FTP, SMTP, Telnet	DNS, DHCP, RIP, SNMP
Vrstni red paketov	Določen	Nedoločen
Hitrost	Počasen	Hiter
Potrjevanje paketov	Da	Ne
Kontrola pretoka	Da	Ne
Velikost zaglavja	20 bajtov	8 bajtov
Število polj v zaglavju	12	4

UDP protokol deluje hitreje od protokola TCP zaradi svojih lastnosti. Uporablja se v primerih, kjer zanesljiva komunikacija ni tako pomembna (DNS, DHCP, spletne igre, VoIP telefonija itd.), medtem ko se TCP protokol uporablja pri aplikacijah, ki zahtevajo visoko zanesljivost (prenos podatkov, spletno bančništvo, spletna pošta itd.). TCP to funkcionalnost rešuje s sistemom potrjevanja paketov (ponovno pošiljanje paketov, ki niso bili potrjeni s strani prejemnika) za razliko od protokola UDP, ki tega ne počne. Omenjeno funkcionalnost je treba reševati na aplikacijskem nivoju. UDP ne omogoča kontrole pretoka, kar pomeni, da je količina poslanih podatkov med komunikacijo stalno enaka. TCP je zmožen količino poslanih podatkov dinamično prilagajati glede na zmožnosti prejemnika. UDP protokol, za razliko od TCP protokola, vsebuje bistveno manjše zaglavje. Njegovo zaglavje je veliko 4 bajte. Zaradi vseh omenjenih lastnosti je protokol UDP hitrejši in bolj primeren za CoAP uporabo.

Tako kot HTTP tudi CoAP temelji na REST modelu. REST pri svojem delovanju uporablja veliko število različnih ukazov. Najbolj pogosto uporabljeni ukazi so GET (pridobivanje podatkov), POST (manipulacija nad

podatki/potrjevanje spletnega obrazca) in DELETE (izbris podatkov). Po izvedbi ukaza uporabnik prejme stanje zahtevka, ki je prikazan s številčno oznako. 1XX (informativne narave), 2XX (akcija je bila sprejeta, razumljiva ter uspešno izvedena), 3XX (klient mora za uspešno izvedbo ukaza izvesti dodatne akcije (preusmeritev itd.)), 4XX (na klientu se je zgodila napaka, izvedba ukaza je bila neuspešna) in 5XX (napaka se je zgodila pri izvedbi ukaza na strežniku). Poznamo tudi neuradne oznake, ki niso točno definirane (440 – prijava je potekla, 495 – napaka pri SSL certifikatu itd.).

MQTT protokol definira ISO/IEC 20922 [17] standard. Namenjen je za komunikacijo med napravami, pri čemer imajo naprave omejeno količino sistemskega pomnilnika in majhno pasovno širino. Za razliko od zgoraj omenjenih protokolov, MQTT temelji na načinu "objava-naročnina". Medtem ko pri HTTP oz. CoAP protokolu naprave komunicirajo neposredno, pri MQTT potrebujemo distributerja. Distributer skrbi za posredovanje podatkov od pošiljatelja do prejemnika. Pošiljatelj pošlje podatke distributerju, ki poskrbi za posredovanje podatkov končnemu cilju glede na to, ali je na njih naročen ali ne.

Glede na zanimanje in uporabo tehnologije, bo število medsebojnih M2M povezav močno narastlo. Analitiki napovedujejo, da bo v prihodnosti med seboj povezanih tudi do/več milijard naprav.

2.2 Standard ETSI-M2M

Do leta 2020 naj bi obstajalo že 50 milijard naprav, ki bodo uporabljale tehnologijo M2M. Trenutne M2M rešitve so močno prilagojene posameznim zahtevam, kar predstavlja problem pri razvoju aplikacij večjih obsežnosti. Zaradi omenjenega problema in potrebe po standardizaciji je Inštitut za Evropske Telekomunikacijske Standarde (*ang. European Telecommunications Standards Institute* - ETSI) izdal specifikacije za standardizacijo [18]. Glede na podatke globalnega standardizacijskega združenja (*ang. Global Standards Collaboration*) je k standardizaciji M2M pripomoglo več kot 140 organizacij

po svetu, vendar je precejšen del standardizacije opravil ETSI.

Tri glavne komponente, ki sestavljajo ETSI M2M sistem, so [19] [20]:

- **M2M naprava** je opremljena s sistemi za zaznavanje in oddajanje podatkov. Naprave so med seboj zmožne komunikacije in so vidne ostalim komponentam v sistemu. Primer naprave: senzor v pnevmatiki, hladilniku itd.

- **M2M prehod** omogoča združevanje več naprav. To pomeni, da lahko do združenih naprav in njihovih podatkov dostopamo tudi zunaj njihovega lokalnega omrežja. Primer prehoda: usmerjevalnik.

- **M2M omrežje** omogoča dostop do M2M naprav in M2M prehodov zunaj ETSI M2M omrežja.

S pomočjo standardizacije so želeli doseči generično funkcionalno arhitekturo, ki bi jo lahko uporabili na vseh treh komponentah. Arhitekturo so poimenovali storitveni vmesnik (*ang. Service Capability Layer* – SCL). Vsaka izmed treh glavnih komponent je izpeljanka storitvenega vmesnika, ki vsebuje določene funkcionalnosti. M2M naprava je tako poimenovana DSCL, M2M prehod kot GSCL in M2M omrežje kot NSCL. M2M naprava z uporabo SCL izvaja aplikacijo, ki se lahko povezuje na M2M prehod. Tudi M2M prehod za izvajanje aplikacije uporablja SCL, ki mu omogoča, da se vede kot posredniški strežnik med M2M napravo in M2M omrežjem. SCL, ki se izvaja na M2M omrežju, pa omogoča komunikacijo z zunanjim ETSI M2M omrežjem. Kasneje je bil izdan nov standard oneM2M, ki je nadomestil ETSI-M2M.

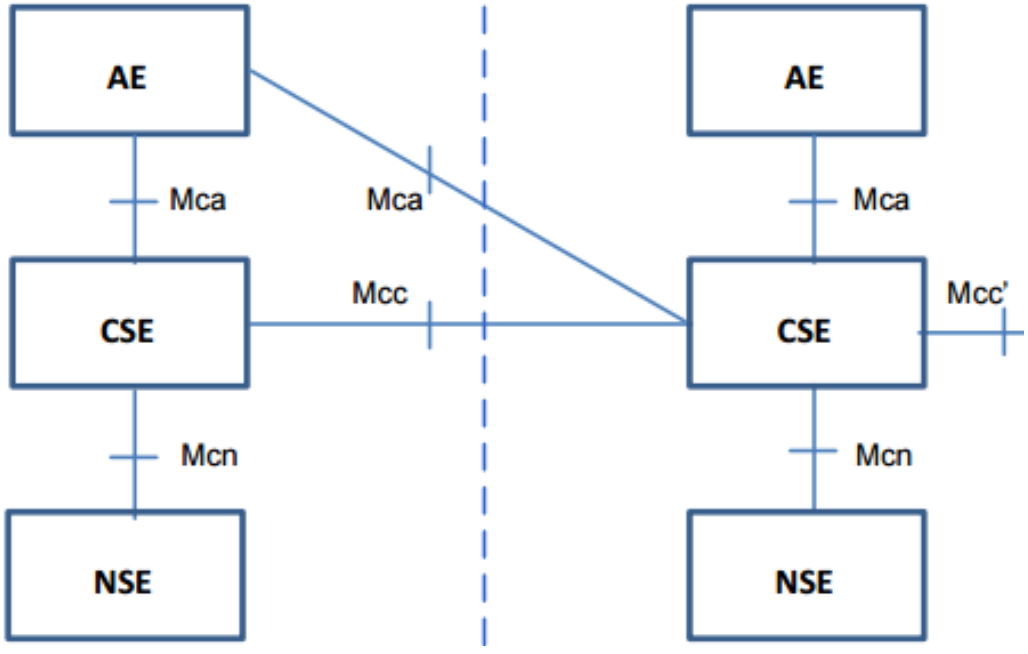
2.3 Standard oneM2M

Zadnja verzija platforme OM2M implementira standard oneM2M [21], ki je naslednik standarda ETSI-M2M. OneM2M je partnerski projekt, ustanovljen leta 2012, ki ga je začelo sedem najpomembnejših organizacij za telekomunikacijsko standardizacijo na svetu: Association of Radio Industries

and Businesses (ARIB) in Telecommunication Technology Committee (TTC) iz Japonske, Alliance for Telecommunications Industry Solutions (ATIS) in Telecommunications Industry Association (TIA) iz ZDA, China Communications Standards Association (CCSA) iz Kitajske, European Telecommunications Standards Institute (ETSI) iz Evrope in Telecommunications Technology Association (TTA) iz Koreje. Glavni cilj je določiti globalno sprejeto M2M storitveno platformo, ki bo podpirala različne IoT aplikacijske storitve. Standard oneM2M je organiziran v pet tehničnih delovnih skupin, ki se osredotočajo na M2M zahteve, arhitekturo, protokole, varnost in upravljanje ter na abstrakcijo in semantiko.

2.3.1 OM2M referenčna arhitektura in referenčne točke

Standard OneM2M določa tri nivoje [22] [23] in sicer aplikacijskega, skupnih storitev ter mrežnega. Nivoji si sledijo od zgoraj navzdol.



Slika 2.4: oneM2M arhitektura

Za vsakega izmed treh nivojev standard določa entiteto (slika 2.4):

- **Aplikacijska entiteta (AE)** predstavlja instanco aplikacijske logike M2M rešitve. Vsaka aplikacijska entiteta je predstavljena z unikatnim AE-ID identifikatorjem. Primer aplikacijske entitete: aplikacija za merjenje sladkorja v krvi, aplikacija za merjenje porabe elektrike, aplikacija za nadzor zračnega prostora itd.

- **Entiteta skupnih storitev (CSE)** predstavlja instanco množice skupnih storitev, ki so na voljo v M2M okoljih. Storitve so dosegljive tudi drugim entitetam, in sicer preko referenčnih točk Mca in Mcc. Entiteta CSE lahko dostopa tudi do entitete omrežnih storitev, in sicer preko referenčne točke Mcn. Storitve, ki jih nudi entiteta skupnih storitev, imenujemo funkcije skupnih storitev (CSF). Omenjene funkcije omogočajo podatkovno upravljanje, upravljanje z napravami, upravljanje z M2M naročninami itd.

- **Entiteta mrežnih storitev (NSE)** zagotavlja entiteti skupnih storitev dostop do funkcionalnosti, ki so na mrežnem nivoju. Primer mrežnih storitev: lokacijske storitve, nadzor nad napravami, proženje naprav itd.

Specifikacije standarda oneM2M definirajo referenčne točke (slika 2.4). Referenčne točke določajo oz. opisujejo tip povezave med entiteto CSE ter eno izmed preostalih dveh entitet. V določenih primerih med seboj povezujejo tudi dve entiteti CSE.

- **Mca** predstavlja komunikacijo med entiteto CSE in entiteto AE. Povezava omogoča aplikacijski entiteti dostop do funkcij oz. storitev, ki jih ponuja entiteta CSE in zmožnost komunikacije entitete CSE ter entitete AE.

- **Mcc** predstavlja komunikacijo med dvema CSE entitetama. Povezava omogoča dostop do medsebojnih storitev.

- **Mcn** predstavlja komunikacijo med entitetama CSE in NSE. Povezava

omogoča entiteti CSE uporabo vseh storitev, ki jih omogoča entiteta NSE.

- **Mcc'** predstavlja komunikacijo med dvema CSE entitetama. Mcc' se od Mcc razlikuje po tem, da povezuje dve CSE entiteti, ki sta v različnih domenah. To pomeni, da lahko entiteta CSE dostopa tudi do storitev zunanjih ponudnikov.

2.3.2 Vozlišča

Pri vozliščih govorimo o vrsti funkcionalnih objektov. Poznamo dva tipa vozlišč. Prvi tip vsebuje natanko eno entiteto CSE ter nič oz. več entitet AE. Pri takem tipu govorimo o "CSE-Capable" vozlišču. Primer vozlišča sta ASN in MN. Drug tip vozlišča vsebuje eno oz. več entitet AE in nobene CSE entitete. Takim vozliščem rečemo "Non-CSE-Capable" vozlišča.

- **Aplikacijsko storitveno vozlišče (ASN)** vsebuje natanko eno entiteto CSE in vsaj eno AE entiteto. S pomočjo referenčne točke *Mcc* ASN komunicira z natanko enim vmesnim vozliščem oz. z natanko enim infrastrukturnim vozliščem. V celotnem sistemu je lahko nič oz. več takšnih vozlišč.

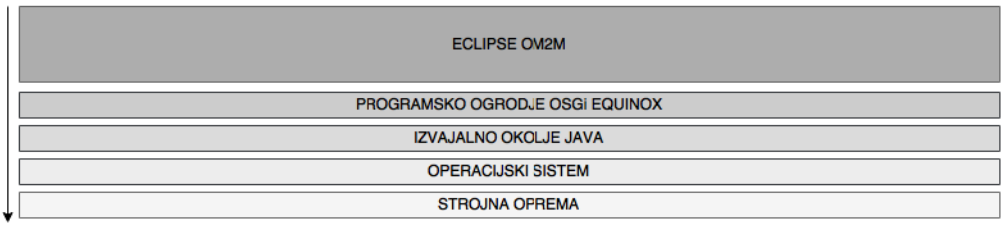
- **Vmesno vozlišče (MN)** vsebuje natanko eno entiteto CSE in nič ali več AE entitet. Vozlišče se preko *Mcc* referenčne točke povezuje z enim izmed vozlišč MN oz. IN. Poleg tega je obvezna povezava tudi na IN/MN oz. ASN preko *Mcc* referenčne točke.

- **Infrastrukturno vozlišče (IN)** vsebuje eno entiteto CSE ter nič ali več AE entitet. IN vozlišče se preko referenčne točke *Mcc* povezuje z MN vozlišči in/ali z drugimi IN vozlišči.

Poglavje 3

Platforma Eclipse OM2M

Projekt Eclipse OM2M [20] je odprtokodna platforma za povezavo med dvema napravama (*ang. Machine to Machine* – M2M). Razvoj platforme je začel francoski Laboratorij za analize in arhitekturo sistemov, pod okriljem Nacionalnega centra za znanstvene raziskave. Sedaj je platforma dostopna pod odprtokodno licenco EPL. Platforma je dostopna v dveh različnih verzijah. Starejša verzija (0.8.0 – izdana 8.4.2015) implementira star ETSI-M2M standard, medtem ko nova verzija (1.0.0 – izdana 22.6.2016) implementira nov standard oneM2M. V nadaljevanju magistrske naloge se bomo osredotočili na uporabo verzije 1.0.0.



Slika 3.1: Arhitekturni nivo platforme Eclipse OM2M

Slika 3.1 prikazuje arhitekturno lokacijo platforme OM2M. Na najnižjem nivoju imamo strojno opremo, nad katero se izvaja operacijski sistem. Ker je OM2M rešitev razvita v programskem jeziku Java, potrebujemo tudi izva-

jalno okolje Java. OM2M je razvita na modularni način, in sicer s pomočjo programskega ogrodja OSGi Equinox. Zaradi zagotavljanja lažje nadgradnje, večjega nadzora pri razvoju in lažjega vzdrževanja je celotna platforma zgrajena modularno. Vsak izmed modulov je zadolžen za določeno funkcionalnost npr. core, http, coap, mqtte, datamapping itd. V primeru, da uporabnik želi dodatno funkcionalnost, mu OM2M omogoča, da jo implementira in vključi v platformo.

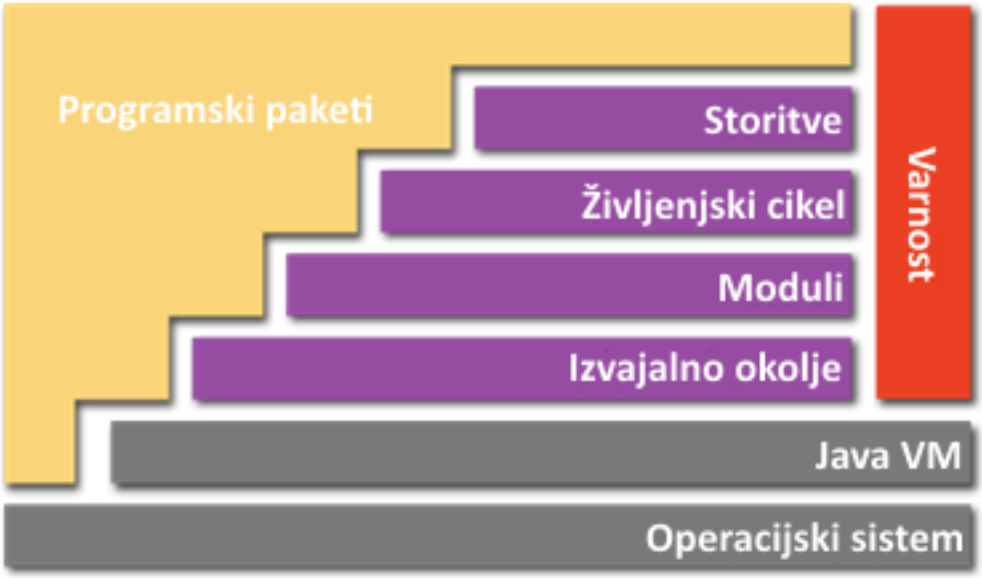
Ko platformo prenesemo z uradne spletne strani, jo je treba "zgraditi". To storimo s pomočjo razvojnega orodja Eclipse in orodja Apache Maven. Po uspešnem "buildu" se v mapi projekta kreirajo ASN-CSE, IN-CSE ter MN-CSE. Nastavitve je mogoče spreminjati preko nastavitvene datoteke *config.ini*. Ta vsebuje nastavitve, kot so IP naslov, naslov vrat, lokacijo podatkovne baze itd.

3.1 OSGi Equinox

OSGi [24] [25] je standard, ki opisuje dinamično modularno platformo za razvoj aplikacij v programskem jeziku Java. Celoten standard je definiralo združenje približno štiridesetih različnih podjetij. Obstaja več implementacij OSGi standarda. Equinox je eno izmed najbolj razširjenih odprtokodnih OSGi ogrodij, ki se uporablja tudi v jedru razvojnega okolja Eclipse. Uporablja se kot osnova v številnih aplikacijah kot tudi v aplikacijskih strežnikih. Omenjeno ogrodje uporablja tudi platforma OM2M, ki je osrednji del magistrske naloge. Knopflerfish je popularna implementacija OSGi specifikacij s strani Švedskega podjetja MakeWave AB. Za razliko od Equinox-a je Knoplerfish dostopen tudi v komercialni različici Knoplerfish Pro. Apache Felix je najmanjša odprtokodna OSGi rešitev (velikost *jar* datoteke), ki je razvita pod licenco Apache. Namenjena je predvsem za vgrajene naprave, kot so senzorji v avtomobilu, dvižnih vratih itd. Concierge je kompaktna in visoko optimizirana OSGi implementacija. Uporablja se predvsem pri napravah, ki imajo omejeno število virov (mobilni telefoni, tablice itd.). Vse omenjene

rešitve zasedejo zelo majhno količino prostora (do 1 MB). Vidimo torej, da je OSGi splošno uporabljen pri razvoju, in sicer od razvojnih orodij (Eclipse), aplikacijskih strežnikov (GlassFish, Oracle Weblogic, JBoss), aplikacijskih ogrodij (Spring) do industrijskih avtomatizacij.

Cilj OSGi-ja je, da razvijalcu omogoči preprost razvoj aplikacije v obliki programskih paketov. Vsak programski paket je implementacija določene logike. Med seboj so lahko odvisni ali pa delujejo sami zase. Platforma s tem omogoča skrivanje celotne implementacije komponent, medtem ko lahko komponente komunicirajo preko storitev.



Slika 3.2: OSGi arhitektura

S stališča arhitekturnega pogleda je OSGi sestavljen iz več nivojev [26], kar lahko vidimo na sliki 3.2. Posamezni arhitekturni nivoji so:

- **Programski paketi** so komponente, razvite s strani razvijalca. Komponenta, ki skrbi za prijavo uporabnika v sistem, komponenta za beleženje zahtevkov itd.

- **Storitve** predstavljajo arhitekturni nivo, ki omogoča dinamično povezovanje med različnimi programskimi paketi.
- **Življenjski cikel** predstavlja API, ki omogoča namestitev, odstranjevanje, zagon, ustavitev in posodobitev programskega paketa.
- **Moduli** predstavljajo arhitekturni nivo, ki določa lastnosti glede souporabe storitev med posameznimi programskimi paketi. V Javi je programski paket predstavljen kot datoteka *JAR*, pri čemer lahko celotno njegovo funkcionalnost uporabljajo vsi ostali *JAR-i*. Pri OSGi je to drugače. Programski paket privzeto skrije celotno funkcionalnost, razen, če določimo, da je vidna navzven.
- **Varnost** je arhitekturni nivo, ki skrbi za varnostni vidik.
- **Izvajalno okolje** določa, katere metode in razredi so dostopni določeni platformi.

3.1.1 Programski paketi in življenjski cikel

OSGi predpostavlja, da so rešitve oz. aplikacije razvite v več različnih programskih paketih (v nadaljevanju paketi). Celoten opis, specifikacije in zahteve so v manifestni datoteki *"META-INF/MANIFEST.MF"*. V manifestni datoteki podatka *"Bundle-SymbolicName"* in *"Bundle-Version"* predstavljata unikatno ime in verzijo modula. Oba podatka sta obvezna. V splošnem so vsi paketi med seboj neodvisni, razen, če pri razvoju izrecno zahtevamo odvisnost. Tako lahko paket, ki skrbi za pisanje v podatkovno bazo, za svoje delovanje zahteva predhodno delovanje paketa, ki skrbi za izvajanje podatkovne baze. Vsak paket se vede kot ločena rešitev. V primeru, da želimo funkcionalnost izbranega paketa uporabiti v drugem modulu, OSGi to rešuje z arhitekturnim nivojem modulov.

Struktura paketov je zasnovana tako, da jih lahko zaganjamo, ustavljamo, posodabljamo, nameščamo in brišemo. Pakete lahko urejamo preko temu namenjene OSGi konzole. Dostopnost OSGi konzole je odvisna od implemen-

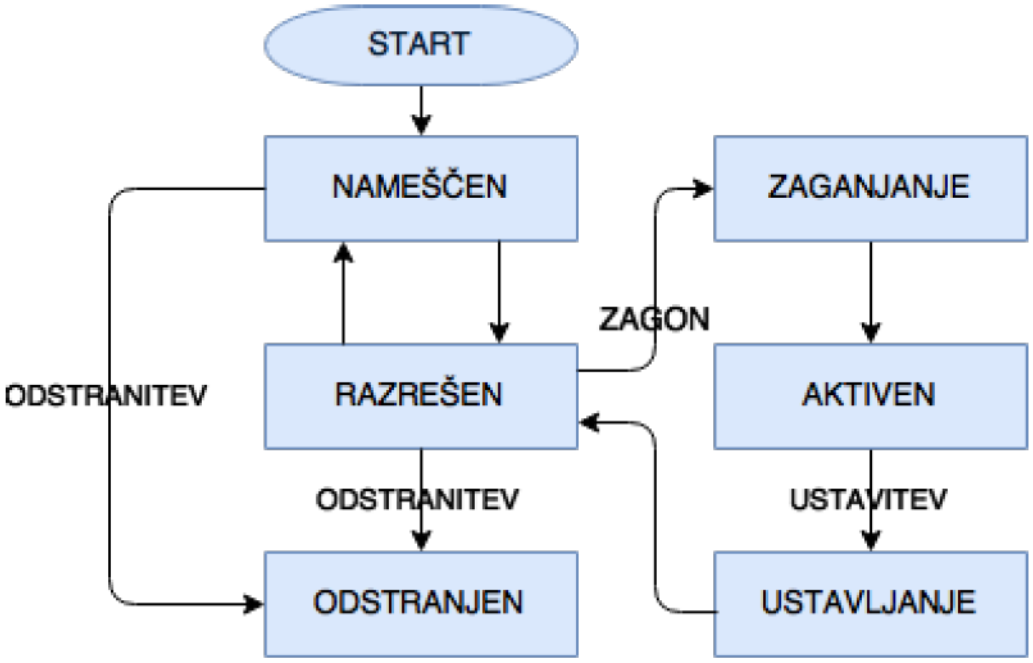
tacije. Ob zagonu OSGi Equinox-a se avtomatsko odpre nadzorna konzola. Preko konzole imamo nadzor in pregled nad vsemi paketi. Konzola omogoča funkcionalnosti, kot so nameščanje, brisanje, zagon in zaustavitev posameznega paketa. Ukaz *ss* nam vrne seznam vseh trenutnih paketov, ki so v okolju. Vsak paket ima naslednje lastnosti:

- **id** - unikatna številka, ki enolično določa modul
- **status** - določa, v katerem statusu je modul
- **paket** - naziv paketa (*"Bundle-SymbolicName"*)

Če želimo namestiti nov paket, to storimo tako, da izvedemo ukaz *install file:datoteka.jar*, pri čemer za datoteko podamo ime datoteke in njeno polno pot. Po uspešni namestitvi dobi paket svoj unikatni ID, ki ga lahko preverimo s pomočjo ukaza *ss*. Če želimo paket zagnati, to storimo s pomočjo ukaza *start id_modula*. Zaustavitev oz. odstranitev paketa izvedemo na podoben način, in sicer s pomočjo ukazov *stop* in *uninstall*. Vsem ukazom podamo ID paketa. Velikokrat se zgodi, da paketu, ki ga želimo namestiti, manjkajo določene odvisnosti. Za ta problem ima OSGi Equinox ukaz *diag id_modula*, ki pove, katere odvisnosti manjkajo za zagon paketa. Poleg teh ukazov OSGi Equinox omogoča še dosti drugih.

Za boljše razumevanje delovanja OSGi paketov je treba poznati njihov življenjski cikel (slika 3.3).

Predpostavimo, da smo implementirali nov paket, ki ga želimo dodati v OSGi platformo. Ob izvedbi funkcije *install* se status nameščenega paketa spremeni v nameščen. Od tod lahko nad paketom izvedemo več ukazov. Če ga odstranimo (ukaz *uninstall*), se mu status začasno spremeni v odstranjen, nato pa izgine s seznama paketov v platformi. S pomočjo funkcije *update* lahko izvedemo posodobitev paketa, pri čemer se mu status ne spremeni. V primeru, da izvedemo funkcijo *start*, se izvede zagon procesa. Ob zagonu se paket iz statusa nameščen prestavi v status razrešen. V tem stanju se preverijo vse povezave in odvisnosti med ostalimi paketi. Če so zagnani vsi



Slika 3.3: Življenjski cikel OSGi paketa

ostali paketi, ki jih potrebuje trenutni, se status spremeni v zaganjanje in nato v status aktiven. Omenjenega prehoda v večini primerov ne opazimo, saj se zgodi zelo hitro. V zadnjem statusu ostane toliko časa, dokler se nad njim ne izvede funkcija *stop*. Ob zaustavitvi paketa se status najprej spremeni v ustavljanje in nato v status razrešen, v katerem ostane, dokler ne izvedemo metode za zagon, posodobitev oz. brisanje paketa. V primeru, da je modul ob zagonu OSGi že nameščen, se status nameščen preskoči. Če ni nobenih napak, modul avtomatsko dobi status razrešen. Nad njim lahko izvedemo vse zgoraj omenjene funkcije. Prav tako so vsi prehodi statusov identični kot zgoraj.

Paketi omogočajo tudi dinamično spremembo statusa oz. obveščanje drugih. Posamezne pakete lahko po potrebi tudi ugasnemo, kar pomeni, da lahko izklopimo samo del aplikacije. S tem zagotovimo lažji nadzor in preprostejše nadgradnje sistema.

3.2 Registracija nove naprave

Pred uporabo platforme OM2M je treba registrirati novo napravo. Napravo si najlažje predstavljamo kot neke vrste tabelo, ki vsebuje določene parametre oz. stolpce. Vsaka naprava je sestavljena iz sklopov/podsklopov. Napravo in sklope kreiramo s pomočjo HTTP POST zahtevkov. Pri vsakem POST zahtevku je treba v zaglavju podati podatke o avtorizaciji v obliki Base64. V nadaljevanju bomo napravo registrirali na vozlišču MN-CSE.

3.2.1 Kreiranje glavnega sklopa naprave

Glavni sklop, ki predstavlja napravo, je "root" objekt, ki vsebuje vse naslednje sklope. Za kreiranje glavnega sklopa pošljemo zahtevek POST (izsek 3.1).

```
Naslov: http://127.0.0.1:8080/~/mn-cse
Vsebina:
<m2m:ae xmlns:m2m="http://www.onem2m.org/xml/protocols" rn="
    MY_SENSOR">
    <api>app-sensor</api>
    <lbl>Type/sensor Category/temperature Location/home</lbl>
    <rr>false</rr>
</m2m:ae>
Zaglavje:
Content-Type: application/xml;ty=2
X-M2M-Origin: admin:admin
```

Izsek 3.1: Kreiranje glavnega sklopa naprave

3.2.2 Kreiranje sklopa Descriptor

Sklop "Descriptor" skrbi za shranjevanje vseh instanc, ki so namenjene shranjevanju metapodatkov. Za kreiranje glavnega "Descriptor" pošljemo zahtevek POST (izsek 3.2).

```
Naslov: http://127.0.0.1:8080/~/mn-cse/mn-name/MY_SENSOR
Vsebina:
```

```
<m2m:cnt xmlns:m2m=http://www.onem2m.org/xml/protocols
rn="DESCRIPTOR">
</m2m:cnt>
Zaglavje:
Content-Type: application/xml;ty=3
X-M2M-Origin: admin:admin
```

Izsek 3.2: Kreiranje sklopa Descriptor

3.2.3 Kreiranje instance sklopa Descriptor

Instanca sklopa "Descriptor" vsebuje vse dodatne metapodatke (opis polj, metode za prikaz zadnjega podatka itd.). Za kreiranje nove instance sklopa "Descriptor" pošljemo zahtevek POST (izsek 3.3). Posamezna instanca sklopa "Descriptor" se v specifikacijah OM2M imenuje "contentInstance".

```
Naslov: http://127.0.0.1:8080/~/mn-cse/mn-name/MY_SENSOR/
    DESCRIPTOR
Vsebina:
<m2m:cin xmlns:m2m="http://www.onem2m.org/xml/protocols">
<cnf>application/xml</cnf>
<con>
 &lt;obj&gt;
  &lt;strname=&quot;type&quot; val=&quot;Temperature_Sensor&quot
    ;/&gt;
  &lt;str name=&quot;location&quot; val=&quot;Home&quot;/&gt;
  &lt;str name=&quot;appId&quot; val=&quot;MY_SENSOR&quot;/&gt;
  &lt;op name=&quot;getValue&quot; href=&quot;/in-cse/in-name/
    MY_SENSOR/DATA/la&quot;
   in=&quot;obix:Nil&quot; out=&quot;obix:Nil&quot; is=&quot;
     retrieve&quot;/&gt;
 &lt;/obj&gt;
</con>
</m2m:cin>
Zaglavje:
Content-Type: application/xml;ty=4
X-M2M-Origin: admin:admin
```

Izsek 3.3: Kreiranje instance sklopa Descriptor

3.2.4 Kreiranje sklopa Data

Sklop "Data" skrbi za shranjevanje vseh podatkovnih instanc, ki jih posamezen senzor oz. zunanja naprava pošilja v platformo OM2M. Postopek kreiranja omenjenega sklopa je identičen kot pri sklopu "Descriptor", le da v vsebini telesa (izsek 3.4) spremenimo vrednost parametra *rn* iz "DESCRIPTOR" v "DATA". Zahtevek POST pošljemo na isti naslov kot pri kreiranju sklopa "Descriptor".

```
Naslov: http://127.0.0.1:8080/~/mn-cse/mn-name/MY_SENSOR
Vsebina:
<m2m:cnt xmlns:m2m=http://www.onem2m.org/xml/protocols
rn="DATA">
</m2m:cnt>
Zaglavje:
Content-Type: application/xml;ty=3
X-M2M-Origin: admin:admin
```

Izsek 3.4: Kreiranje sklopa Data

3.2.5 Kreiranje instance sklopa Data

Sklop "Data" skrbi za shranjevanje fizičnih podatkov, ki jih pošiljajo naprave. Za shranjevanje podatkov pošljemo zahtevek POST (izsek 3.5).

```
Naslov: http://127.0.0.1:8080/~/mn-cse/mn-name/MY_SENSOR/DATA
Vsebina:
<m2m:cin xmlns:m2m="http://www.onem2m.org/xml/protocols">
    <cnf>application/xml</cnf>
    <con>
        &lt;obj&gt;
            &lt;str name=&quot;type&quot; val=&quot;
    Temperature_Sensor&quot;/&gt;
            &lt;str name=&quot;location&quot; val=&quot;Home&
    quot;/&gt;
            &lt;str name=&quot;appId&quot; val=&quot;MY_SENSOR&
    quot;/&gt;
```

```
10              &lt;op name=&quot;getValue&quot; href=&quot;/in-cse/
    in-name/MY_SENSOR/DATA/la&quot;
11            in=&quot;obix:Nil&quot; out=&quot;obix:Nil&quot; is=&
    quot;retrieve&quot;/&gt;
12          &lt;/obj&gt;
13      </con>
14 </m2m:cin>
15 Zaglavje:
16 Content-Type: application/xml;ty=4
17 X-M2M-Origin: admin:admin
```

Izsek 3.5: Kreiranje sklopa Data

Z zgoraj omenjenimi postopki je naprava uspešno kreirana in pripravljena za uporabo.

3.2.6 Funkcionalnost naročnin

Platforma OM2M omogoča funkcionalnost naročnin. To pomeni, da lahko OM2M pošlje obvestilo zunanjemu sistemu ob določenem dogodku, kot so izbris podatkov, posodobitev podatkov, vnos novega podatka itd. Za naročnino na sklop "Data" pošljemo zahtevek POST (izsek 3.6). Platforma OM2M bo v primeru novega podatka v aplikaciji MY_SENSOR poslala obvestilo na naslov, ki je naveden v sklopu *nu*.

V našem primeru je to naslov http://127.0.0.1:1400/monitor.

```
1 Naslov: http://127.0.0.1:8080/~/mn-cse/mn-name/MY_SENSOR/DATA
2 Vsebina:
3 <m2m:sub xmlns:m2m="http://www.onem2m.org/xml/protocols" rn="
    SUB_MY_SENSOR">
4     <nu>http://localhost:1400/monitor</nu>
5     <nct>2</nct>
6 </m2m:sub>
7 Zaglavje:
8 Content-Type: application/xml;ty=23
9 X-M2M-Origin: admin:admin
```

Izsek 3.6: Uporaba naročnin - sklop Data

Poglavje 4

Razširitev platforme Eclipse OM2M

4.1 Vtičnik za pisanje v podatkovno bazo MongoDB

Platforma OM2M za svoje delovanje uporablja H2 [27] podatkovno bazo. H2 je relacijska podatkovna baza, ki je razvita v programskem jeziku Java in se izvaja v pomnilniku. Za dostop do podatkov se uporablja določena podmnožica SQL ukazov. Dva glavna programska vmesnika sta SQL in JDBC. Podatkovna baza je hitra, saj je v hitrem pomnilniku, vendar ne omogoča shranjevanja velikih količin podatkov. Večja količina podatkov pomeni večji pomnilnik, ki je tudi dražji. Zaradi omenjenih problemov smo za shranjevanje podatkov izbrali NoSQL podatkovno bazo. NoSQL podatkovne baze so bolj primerne za shranjevanje velikih količin podatkov, saj so bolj skalabilne. Prednost NoSQL podatkovnih baz je tudi, da se lahko struktura tabele spremeni na preprostejši način. Z NoSQL podatkovno bazo smo želeli doseči hitrejši vnos podatkov in podporo shranjevanja velikih količin podatkov. Za NoSQL bazo smo vzeli MongoDB [28].

MongoDB je večplatformna odprtokodna podatkovna baza. Nerelacijske podatkovne baze svoje podatke shranjujejo na več različnih načinov. Eden

izmed osnovnih načinov je *"ključ-vrednost"*, ki velja za temelj nerelacijskih podatkovnih baz. Vsak unikatni ključ se preslika v pripadajočo vrednost. Shranjujemo lahko poljubne dokumentne tipe, saj jih podatkovna baza vedno shranjuje kot objekt tipa BLOB. MongoDB je dokumentna podatkovna baza, ki podatke shranjuje v obliki dokumentov. Dokument si lahko predstavljamo kot nabor parov oblike *"ključ-vrednost"*. Vsi pari so združeni in strukturirani v dokument. Večina podatkov v MongoDB prihaja v obliki formatov XML in JSON, ki vnaprej nimata točno določene strukture podatkov. V praksi to pomeni, da lahko MongoDB shrani podatek, ki vsebuje dve vrednosti, v naslednji iteraciji pa podatek, ki vsebuje štiri vrednosti. Za razliko od nerelacijskih podatkovnih baz se struktura MongoDB tabele avtomatsko prilagodi.

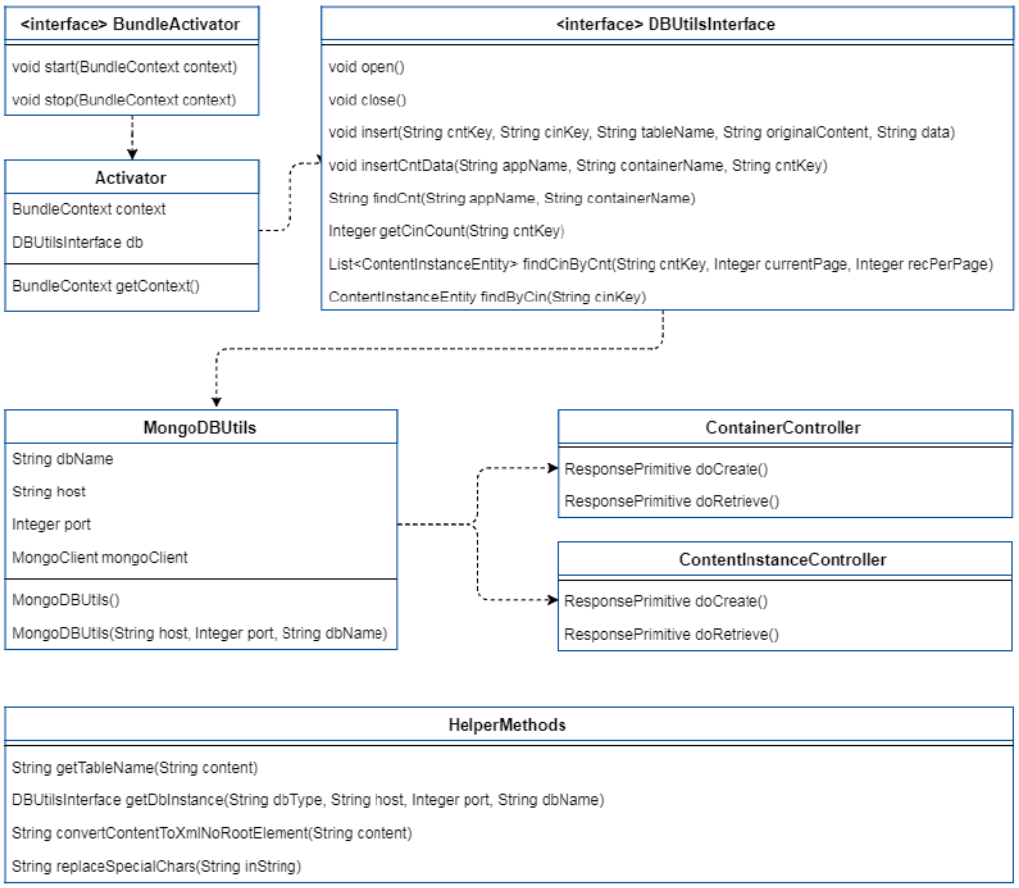
Pred razvojem vtičnika smo razmišljali o dveh različnih načinih implementacije. Eden izmed načinov je bil uporaba OM2M naročnin. V OM2M verziji 0.8 se je logika naročnin izvajala pred zapisovanjem v H2. Obstajala je možnost, da se pisanje v NoSQL izvaja hitrejše od pisanja v H2. V OM2M verziji 1.0 so razvijalci platforme izvajanje naročnin prestavili za izvajanje sklopa pisanja v H2. To pomeni, da s funkcionalnostjo naročnin ni mogoče doseči pohitritve pri zapisovanju v podatkovno bazo NoSQL. Zaradi omenjenih problemov smo se odločili, da naročnin ne bomo uporabljali, temveč bomo spremenili logiko, ki skrbi za pisanje v podatkovno bazo.

S tem namenom smo s pomočjo OSGi razvili nov vtičnik, ki omogoča pisanje v MongoDB. Način implementacije ter pripravljeni programski vmesniki zagotavljajo preprosto uporabo in možnost morebitne nadgradnje. V primeru, da želi uporabnik dodati podporo za drugo NoSQL bazo, ima glavno ogrodje že pripravljeno.

4.1.1 Opis programskih razredov

Vtičnik se imenuje *"org.eclipse.om2m.nosql"* in je sestavljen iz več programskih razredov (v nadaljevanju razred) (slika 4.1).

Razred **Activator** je izpeljan iz OSGi razreda *BundleActivator* in vsebuje



Slika 4.1: UML diagram razredov vtičnika *"org.eclipse.om2m.nosql"*

programski metodi (v nadaljevanju metodi) *start* in *stop*. Metoda *start* skrbi za zagon modula in inicializacijo celotne programske logike. Ob klicu metode *start* se izvede branje parametrov iz nastavitvene datoteke. Parametri, ki jih potrebujemo za inicializacijo, so:

- **Naziv podatkovne baze** *(org.eclipse.om2m.nosql.dbName)*. Privzeta vrednost je "*<om2m_vozlišče>*-om2m". "*<om2m_vozlišče>*" se zamenja z nazivom vozlišča, na katerem se vtičnik izvaja (mn-om2m, asn-om2m oz. in-om2m).
- **Vrsta podatkovne baze** *(org.eclipse.om2m.nosql.dbName)*. Privzeta vrednost je *"MongoDB"*.

- **Lokacija podatkovne baze** *(org.eclipse.om2m.nosql.host)*. Privzeta vrednost je *"localhost"*.

- **Naslov vrat** *(org.eclipse.om2m.nosql.port)*. V primeru, da je nastavitev prazna, se uporabi privzeta vrednost, ki je definirana v izbrani podatkovni bazi.

Zaradi lažjega in bolj intuitivnega dostopa do podatkov smo se odločili, da imena tabele, v katero se shranjujejo podatki, ni mogoče spreminjati. Tabela oz. podatkovna zbirka je avtomatsko poimenovana tako kot ime ustvarjene aplikacije znotraj platforme OM2M, v katero zapisujemo podatke. V primeru, da zapisujemo podatke v OM2M aplikacijo "MY_SENSOR", bodo vsi podatki shranjeni v istoimenski tabeli. Po branju nastavitvenih parametrov smo na podlagi njihovih vrednosti s pomočjo metode *getDbInstance* pridobili instanco objekta *DBUtilsInterface*. Objekt vsebuje vse funkcionalnosti, ki se navezujejo na upravljanje z izbrano podatkovno bazo. Ker je razred izpeljan iz abstraktnega razreda, se kasneje pri uporabi ni treba ukvarjati, za kateri tip podatkovne baze gre, temveč je lahko implementacija enotna za vse tipe podatkovne baze. OSGi omogoča možnost uporabe storitev. V start metodi smo izvedli registracijo storitve, pri čemer smo v konstruktor podali ime razreda ter instanco objekta *DBUtilsInterface* (izsek 4.1). To nam omogoča, da lahko po zagonu programskega paketa do njegove funkcionalnosti dostopamo tudi v drugih programskih paketih, ki so "naročeni" na njegovo storitev.

```
1 this.db = HelperMethods.getDbInstance(dbType, host, intPort,
    dbName);
2 context.registerService(DBUtilsInterface.class.getName(),this.db
    ,null);
```

Izsek 4.1: Registracija storitve DbUtilsInterface (izvorna koda)

Funkcija *stop* se izvede ob zaustavitvi paketa in vsebuje klic metode za zapiranje povezave do podatkovne baze in zaustavitev izvajanja vtičnika.

Razred **MongoDBUtils** vsebuje implementacijo vseh abstraktnih funkcij, ki so definirane v razredu *DBUtilsInterface*:

- Funkcija ***open*** izvaja vzpostavitev povezave s podatkovno bazo MongoDB in kreiranje instance objekta *MongoClient*, ki jo pri svojem izvajanju uporabljajo druge funkcije. Funkcija ne sprejme nobenega parametra. Uporaba v razredu *Activator (org.eclipse.om2m.core)*.

- Funkcija ***close*** izvaja prekinitev povezave s podatkovno bazo MongoDB. Funkcija ne sprejme nobenega parametra. Uporaba v razredu *Activator (org.eclipse.om2m.core)*.

- Funkcija ***insert*** izvaja vnos podatkov v tabelo *CNT_CIN_MONGO* in končno tabelo, ki hrani vse *CIN* vrednosti. Naziv tabele je odvisen od imena aplikacije, v katero zapisujemo podatke. Funkcija sprejme parametre *cntKey* (id očeta), *cinKey* (id zapisa), *tableName* (ime končne tabele), *originalContent* (vsebina zahtevka iz vozlišča con), *data* (podatki za zapis).
  Uporaba v razredu *ContentInstanceController (org.eclipse.om2m.core)*.

- Funkcija ***insertCntData*** izvaja vnos podatkov v tabelo *CNT_MONGO*. Funkcija sprejme parametre *appName* (naziv končne tabele), *containerName* (naziv sklopa), *cntKey* (id očeta). Uporaba v razredu *ContentContainer (org.eclipse.om2m.core)*.

- Funkcija ***findCnt*** izvaja pridobivanje podatkov iz tabele *CNT_MONGO*. Funkcija sprejme parametre *appName* (naziv končne tabele), *containerName* (naziv sklopa). Uporaba v razredu *ContentInstanceController (org.eclipse.om2m.core)*.

- Funkcija ***findCinByCnt*** na podlagi vrednosti očeta pridobi vse podrejene zapise iz končne tabele. Funkcija sprejme parametre *cntKey* (id očeta), *currentPage* (trenutna številka strani), *recPerPage* (maksimalno število zapisov na stran). Uporaba v razredu *ContainerController (org.eclipse.om2m.core)*.

- Funkcija ***findByCin*** izvaja pridobivanje podrobnosti končnega podatka (*cin*). Funkcija sprejme parameter *cinKey* (id *cin* podatka).

Uporaba v razredu *ContentInstanceController (org.eclipse.om2m.core)*.

- Funkcija ***getCinCount*** vrača število vseh podrejenih zapisov posameznega očeta. Funkcija sprejme parameter *cntKey* (id očeta). Uporaba v razredu *ContainerController (org.eclipse.om2m.core)*.

Razred **HelperMethods** vsebuje implementacijo vseh pomožnih funkcij, ki se uporabljajo v vtičniku:

- Funkcija ***getTableName*** iz naslova, na katerega pošiljamo podatke, pridobi naziv tabele, v katero kasneje zapisujemo podatke. Funkcija sprejme parameter *content*. Uporaba v razredu *ContentInstanceController (org.eclipse.om2m.core)*.

- Funkcija ***getDbInstance*** na podlagi določenih pogojev vrne instanco objekta *DBUtilsInterface*. Na vrnjeni instanci izvajamo metode *insert*, *open* itd. Funkcija sprejme parametre *dbType* (vrsta podatkovne baze), *host* (lokacija izvajanja), *port* (vrata izvajanja), *dbName* (naziv podatkovne baze). Uporaba v razredu *Activator (org.eclipse.om2m.nosql.rest)*.

- Funkcija ***convertContentToXmlNoRootElement*** pretvori niz podatkov v XML obliko, ki jo kasneje spremenimo v JSON format s pomočjo metode *toJSONObject*. Funkcija sprejme parameter *content* (niz podatkov). Uporaba v razredu *MongoDBUtils (org.eclipse.om2m.nosql.utils)*.

- Funkcija ***replaceSpecialChars*** v nizu zamenja HTML znake s pravimi simboli ("<" z "<" itd). Funkcija sprejme parameter *inString* (niz podatkov za pretvorbo). Uporaba v razredu *ContentInstanceController (org.eclipse.om2m.nosql.utils)*.

4.1.2 Implementacija

Glavna logika pisanja v podatkovno bazo H2 se izvaja v jedru platforme, in sicer v programskem paketu *"org.eclipse.om2m.core"*. Za uporabo novo

razvitega programskega paketa NoSQL je treba v razred *Activator* dodati logiko za "zaznavanje" storitve NoSQL (izsek 4.2).

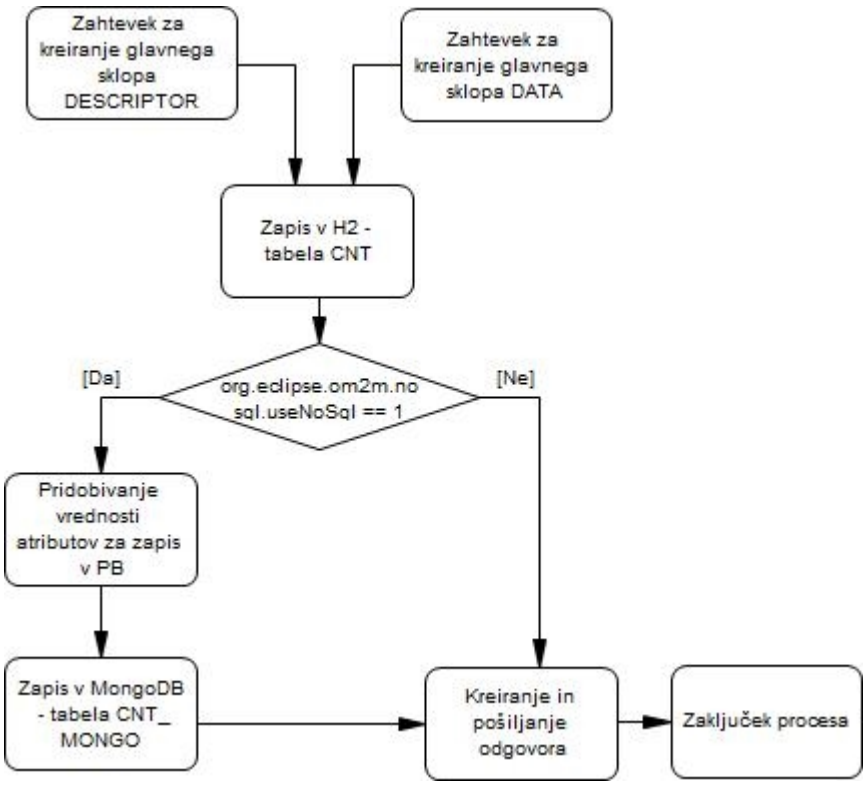
```
noSqldatabaseServiceTracker = new ServiceTracker<Object,Object>
(bundleContext,DBUtilsInterface.class.getName(), null){
  public Object addingService(org.osgi.framework.
   ServiceReference
                              <Object> reference) {
    DBUtilsInterface service = (DBUtilsInterface) this.context.
   getService(reference);
    NosqlDBService.setDbUtilsInterface(service);
    NosqlDBService.getDbUtilsInterface().open();
    LOGGER.info("NoSQL (DBUtilsInterface) service discovered");
    return service;
  }

  @Override
  public void removedService(ServiceReference<Object> reference,
    Object service) {
    LOGGER.info("NoSQL (DBUtilsInterface) service removed");
    NosqlDBService.setDbUtilsInterface(null);
  }
};
noSqldatabaseServiceTracker.open();
```

Izsek 4.2: Zaznavanje storitve NoSQL (izvorna koda)

Pri implementaciji smo se osredotočili na razreda *ContainerController*, kjer je logika, ki zapisuje zapise o metapodatkih (*DESCRIPTOR*) in *ContentInstanceController*, ki skrbi za zapisovanje zapisov o podatkih (*DATA*). Oba razreda vsebujeta metodo *doCreate*, ki izvaja omenjeni logiki. Zapise o metapodatkih shranjuje v tabelo *CNT*, zapise o podatkih pa v tabelo *CIN*. Relacija med tabelama *CNT* in *CIN* je tipa "oče-sin". V primeru, da v OM2M pošljemo zahtevek za shranjevanje podatka, ki ima 5 stolpcev, se celoten sklop o podatkih shrani v tabelo *CIN*, in sicer v en stolpec. To pomeni, da je oteženo iskanje po podatkih, izvajanje povezav med več tabelami, težavna uporaba podatkov iz zunanjih aplikacij itd. Zaradi omenjenih težav,

želje po čim manjših posegih v jedro OM2M in težav pri shranjevanju velikih količin podatkov, smo sklop pisanja v tabelo *CIN* prestavili iz H2 v MongoDB.

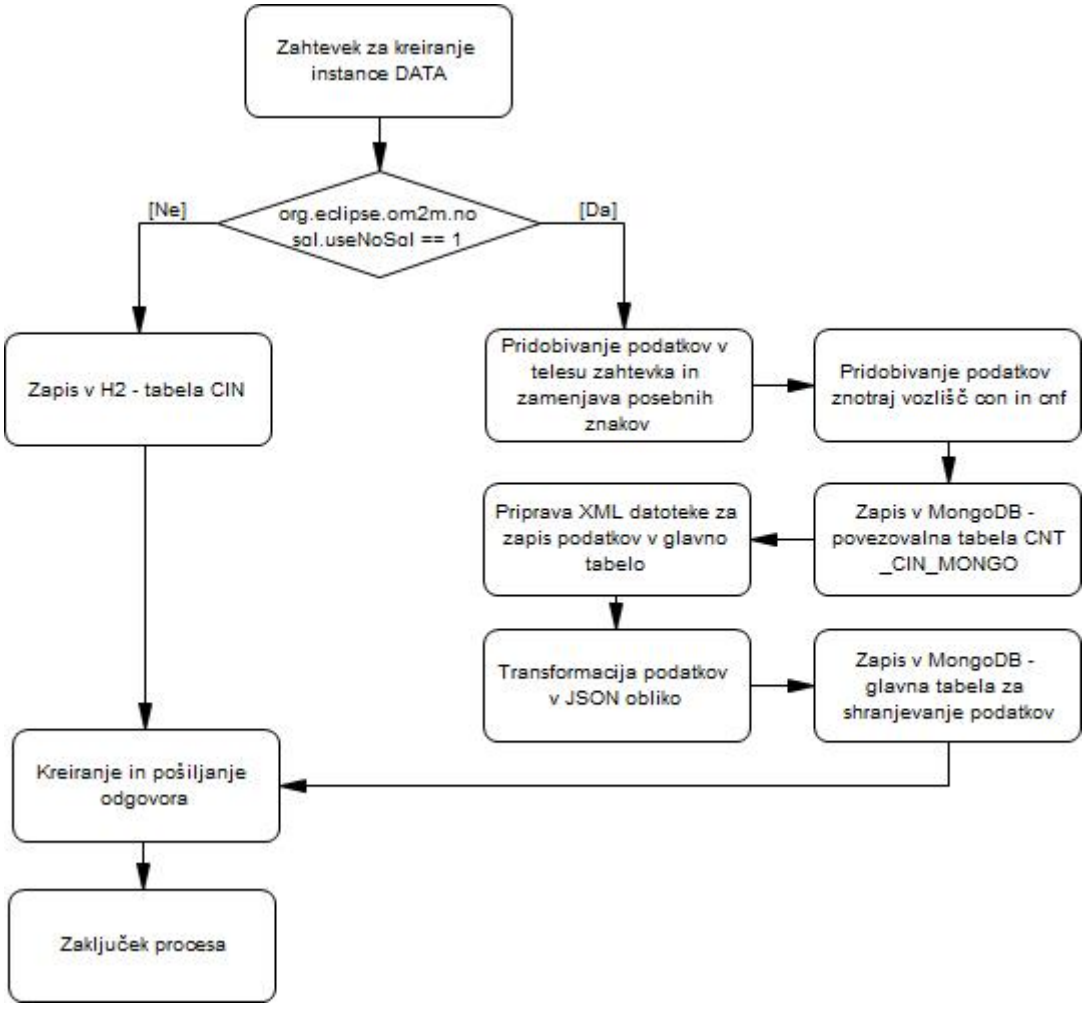


Slika 4.2: Procesni diagram kreiranja glavnega sklopa DESCRIPTOR oz. DATA

Ob kreiranju glavnega sklopa *DESCRIPTOR* oz. *DATA* (slika 4.2) se preveri vrednost atributa *org.eclipse.om2m.nosql.useNoSql*. V primeru, da je vrednost nastavljena na "1", se podatek poleg v H2 zapiše tudi v MongoDB. V MongoDB tabelo *CNT_MONGO* shranjujemo tri vrednosti: *appName*, *containerName* in *cntKey*. Tabela *CNT_MONGO* ima pomen pri prikazovanju podatkov iz MongoDB, zato bo njena vloga opisana v naslednjih poglavjih. Opisana programska logika se izvaja v programskem razredu *ContainerController*. Slika 4.3 prikazuje primer vsebine tabele *CNT_MONGO* v podatkovni bazi MongoDB.

```
> db.CNT_MONGO.find()
{ "_id" : ObjectId("59a16f11ed67261e60d6459c"), "appName" : "MY_SENSOR", "contai
nerName" : "DESCRIPTOR", "cntKey" : "/mn-cse/cnt-349747846" }
{ "_id" : ObjectId("59a16f12ed67261e60d6459d"), "appName" : "MY_SENSOR", "contai
nerName" : "DATA", "cntKey" : "/mn-cse/cnt-66517914" }
>
```

Slika 4.3: Primer vsebine tabele CNT_MONGO - MongoDB



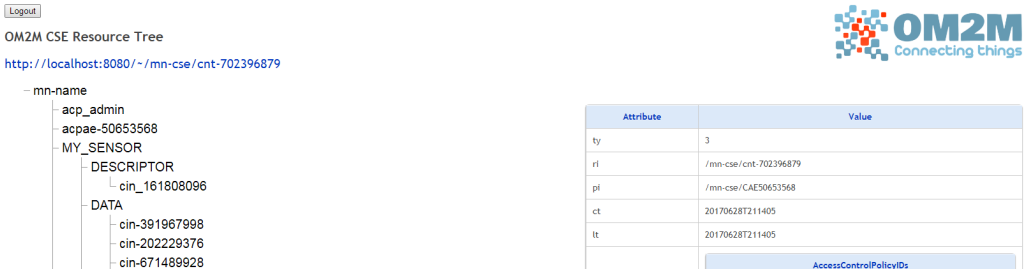
Slika 4.4: Procesni diagram kreiranje instance DATA

Zapisovanje podatkov, ki jih pošiljajo razne naprave, se izvaja s pomočjo kreiranja instance sklopa "Data" (slika 4.4). Tako kot pri kreiranju glavnega sklopa *DESCRIPTOR* oz. *DATA* tudi tukaj preverimo vrednost atributa *org.eclipse.om2m.nosql.useNoSql*. Atribut omogoča spreminjanje izvajanja programske logike. V primeru, da vrednost atributa nastavimo na "1", se podatki shranjujejo v MongoDB, v nasprotnem primeru pa v H2. Vrednost atributa lahko v datoteki *config.ini* spreminjamo za vsako vozlišče posebej. Kot lahko vidimo v poglavju 'Kreiranje instance sklopa "Data"', je zahtevek za vnos novega podatka sestavljen iz vozlišč *cnf* in *con*. Znotraj vozlišča *con* je vozlišče *obj*, ki vsebuje podatke, primerne za zapis. Vsak zapis znotraj vozlišča *con* predstavlja svoj stolpec v MongoDB tabeli. Zapis se začne z oznako, za katero vrsto podatka gre. Podatek je lahko v obliki niza (str) ali številke (int). Nato sledita atribut *name*, ki predstavlja naziv stolpca v tabeli in *val*, ki predstavlja vrednost podatka. Pred zapisom podatkov v MongoDB smo iz vozlišča *obj* izluščili podatke in jih pretvorili v JSON obliko, ki je primerna za zapis. Poleg podatkov, izluščenih iz vozlišča *obj*, smo v tabelo zapisali tudi nekaj preostalih podatkov, ki se prikazujejo na spletnem vmesniku (datum kreiranja, datum spremembe itd.). Ker so vse instance sklopa *DATA* vezane na glavni sklop *DATA*, v tabelo zapisujemo tudi ID vrednost glavnega sklopa *DATA*. Poleg zapisovanja podatkov v glavno tabelo, določene podatke zapisujemo tudi v tabelo *CNT_CIN_MONGO*, in sicer vrednosti *cinKey* in *cntKey*. Tako kot tabela *CNT_MONGO* se tudi *CNT_CIN_MONGO* uporablja pri prikazovanju podatkov iz MongoDB, zato bo njena vloga opisana v naslednjih poglavjih. Opisana programska logika se izvaja v programskem razredu *ContentInstanceController*. Slika 4.5 prikazuje primer vsebine tabele *MY_SENSOR* v podatkovni bazi MongoDB.

Slika 4.5: Primer vsebine tabele MY_SENSOR - MongoDB

4.2 Vtičnik za prikaz podatkov iz podatkovne baze MongoDB

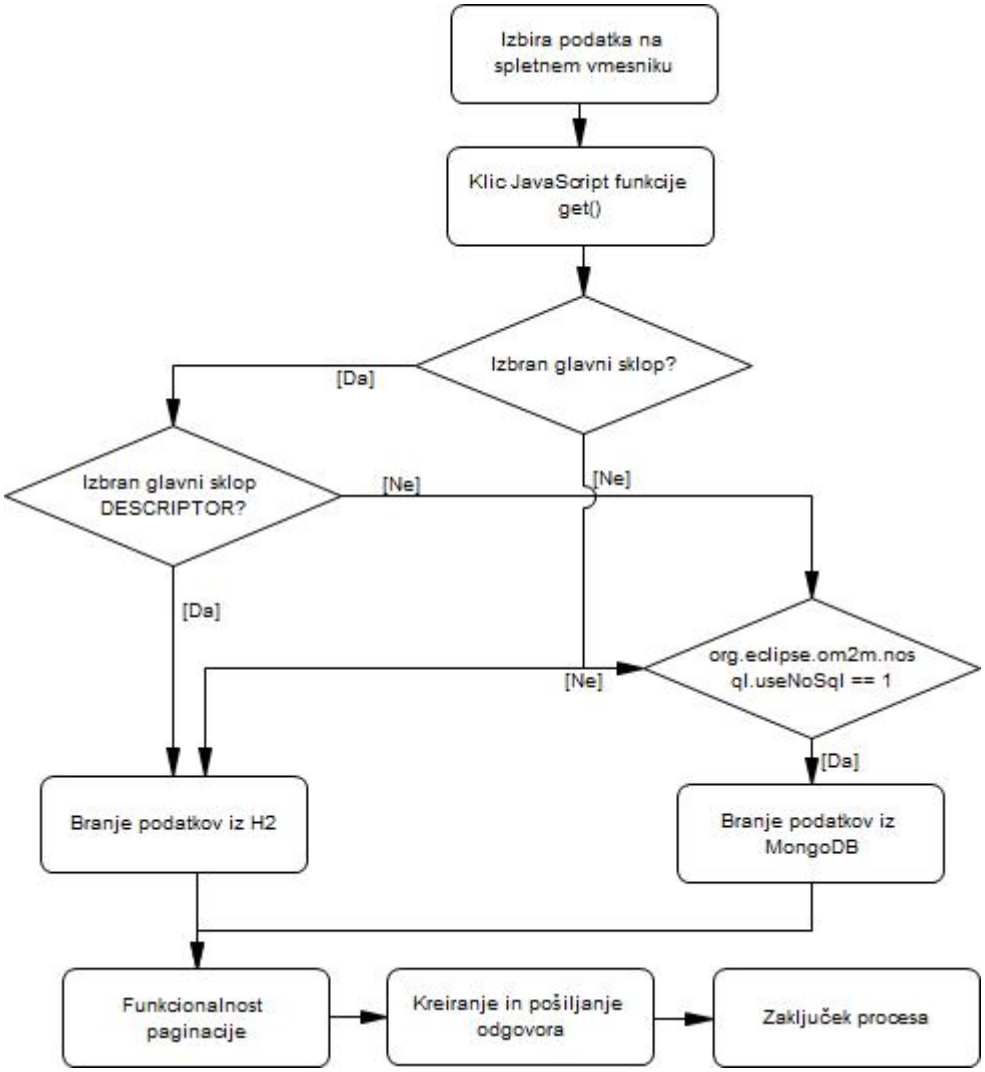
Platforma OM2M omogoča pregled podatkov preko spletnega vmesnika. Zaradi spremembe pisanja podatkov iz H2 v MongoDB smo spremenili tudi sklop, ki omogoča branje podatkov.



Slika 4.6: Primer spetnega vmesnika OM2M

Slika 4.6 prikazuje spletni vmesnik vozlišča MN platforme OM2M. Na vozlišču je registrirana aplikacija *MY_SENSOR*, ki vsebuje sklopa *DESCRIPTOR* in *DATA*. Sklop *DESCRIPTOR* vsebuje eno instanco, sklop *DATA* pa vsebuje štiri podatke (instance). Programska logika, ki se izvaja ob kliku na sklop *DESCRIPTOR*, ostaja nespremenjena. Ob kliku na sklop *DATA* se izvede JavaScript funkcija *get*, ki sprejme parameter *id*. Tako kot pri pisanju v MongoDB se tudi tukaj (slika 4.7) branje izvaja na podlagi atributa *org.eclipse.om2m.nosql.useNoSql*. Vrednost parametra se ob kliku na glavni sklop *DATA* vedno začne z nizom "cnt-", ki mu sledi unikatna številka

(cnt-63876721, cnt-330206243 itd.).



Slika 4.7: Primer spetnega vmesnika OM2M

V primeru, da uporabljamo H2 bazo, se na podlagi tega niza pridobijo vsi podatki iz *CNT* tabele. Ker sta *CNT* in *CIN* med seboj odvisni, posledično pridobimo tudi vse podrejene zapise *CIN*, ki jih prikažemo na spletnem vmesniku. Vtičnik za pisanje v MongoDB zapisuje vse *CIN* podatke v svojo tabelo, ki je poimenovana enako kot OM2M aplikacija, v katero zapisujemo podatke. To pomeni, da tabele, ki vsebuje *CIN* podatke, ni mogoče določiti

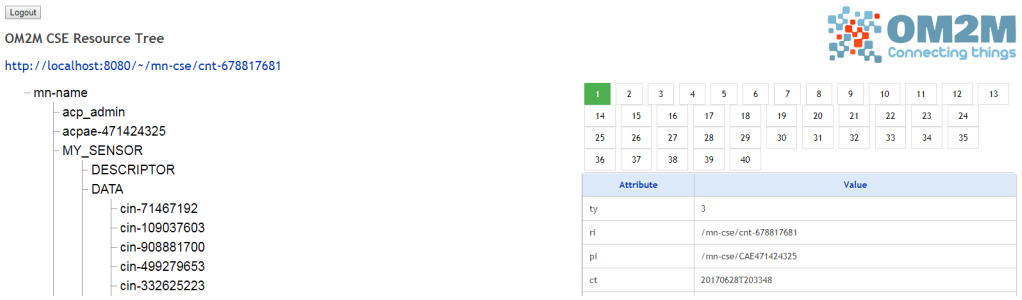
samo na podlagi *id* vrednosti. Za ta namen smo v sklopu pisanja v MongoDB zapisovali podatke v tabelo *CNT_MONGO*. Podatki so v obliki ime aplikacije (appName), naziv sklopa (descriptorName) in primarni ključ očeta (cntKey). Na podlagi *id* vrednosti smo s pomočjo tabele *CNT_MONGO* (omejimo se po stolpcu *cntKey*) pridobili ime tabele, v kateri so shranjeni *CIN* podatki (stolpec *appName*). Omenjene podatke smo nato pretvorili v OM2M obliko. Za vsak zapis v MongoDB tabeli smo kreirali novo instanco objekta *ContentInstanceEntity* in mu nastavili vse potrebne vrednosti, ki se prikazujejo na spletnem vmesniku. Vse instance smo združili v Java objekt "List", shranili v očetovo lastnost *childContentInstances* in prikazali na spletnem vmesniku. Omenjena logika se izvaja v razredu *ContainerController*.

Ob kliku na posamezen podatek sklopa *DATA* se, tako kot ob kliku na glavni sklop *DATA*, izvede JavaScript funkcija *get*. Tokrat se vrednost parametra *id* začne z nizom "cin-", ki mu sledi unikatna številka (cin-474469020, cin-590898564 itd.). V primeru uporabe H2 se s pomočjo parametra iz tabele *CIN* pridobijo podrobnosti o izbranem podatku. Zaradi naše podatkovne strukture v MongoDB to ni možno. S pomočjo tabele *CNT_CIN_MONGO*, na podlagi vrednosti parametra *id* (omejimo se po stolpcu *dataKey*), pridobimo id očeta (stolpec *descriptorKey* - vrednost "cnt"), ki mu pripada iskan *CIN* podatek. Na podlagi vrednosti id očeta nato v tabeli *CNT_MONGO* poiščemo ime tabele v kateri je *CIN* podatek. Pridobimo podatek iz tabele, kreiramo novo instanco objekta *ContentInstanceEntity*, nastavimo vse potrebne vrednosti in podatek prikažemo na spletnem vmesniku. Omenjena logika se izvaja v razredu *ContainerInstanceController*.

4.3 Funkcionalnost paginacije

Spletna platforma OM2M omogoča pregled shranjenih podatkov. Ob pritisku na glavni sklop "DATA" oz. "DESCRIPTOR" se izvede prikaz podrejenih zapisov. Pri veliki količini podatkov (10.000 in več) postane spletni vmesnik neodziven. Težava je v tem, da obstoječa funkcionalnost s pomočjo Java-

Scripta na eno stran izpisuje vse podrejene zapise, kar pa je zamudno in zelo zahtevno. Zaradi omenjene težave smo implementirali funkcionalnost paginacije.



Slika 4.8: Funkcionalnost paginacije glavnega sklopa DATA

Implementirali smo jo na nivoju glavnih sklopov za obe podatkovni bazi. JavaScript funkcija *get* je v datoteki *om2m.js*, in sicer v projektu *org.eclipse.om2m.webapp.resourcesbrowser.xml*. Uporabnik lahko s pomočjo atributa *org.eclipse.om2m.numOfRecPerPage* (datoteka *config.ini*) na vsakem vozlišču posebej nastavlja število zapisov na strani. Poleg obstoječega parametra *targetId* funkcija sprejme nov parameter *pagenum*, ki določa zaporedno številko strani, ki jo želimo prikazati. Parameter upoštevamo pri klicu logike za pridobivanje podatkov, kjer s pomočjo id vrednosti očeta pridobimo število njegovih podrejenih zapisov. Na podlagi števila podrejenih zapisov in izbranega števila zapisov, prikazanih na eni strani, izračunamo končno število strani. Na podlagi omenjenega števila s pomočjo JavaScripta in CSS-ja izrišemo primerno število gumbov za premikanje po straneh. Ob kliku na glavni sklop se vedno izvede prikaz prve strani. MongoDB paginacijo izvajamo s pomočjo funkcij *sort*, *skip* in *limit*. Funkcija *sort* omogoča, da so pridobljeni podatki vedno prikazani v istem vrstnem redu. S pomočjo funkcije *skip* preskočimo število podatkov, ki ga izračunamo na podlagi parametra *pagenum* in želenega števila zapisov, prikazanih na eni strani. Funkcija *limit* omogoča prikaz omejene količine podatkov. Podatkovna baza H2 ne omogoča prej omenjenih funkcij, zato smo paginacijo razvili s pomočjo Java funkcije *su-*

bList. Slika 4.8 prikazuje uporabo paginacije za 20.000 podatkov, pri čemer je prikazanih 500 na posamezno stran.

Poglavje 5

Testiranje in evalvacija

V zgornjih poglavjih smo prikazali in opisali novo razvite funkcionalnosti za platformo OM2M. V trenutnem poglavju se bomo usmerili na sam postopek testiranja in končne rezultate.

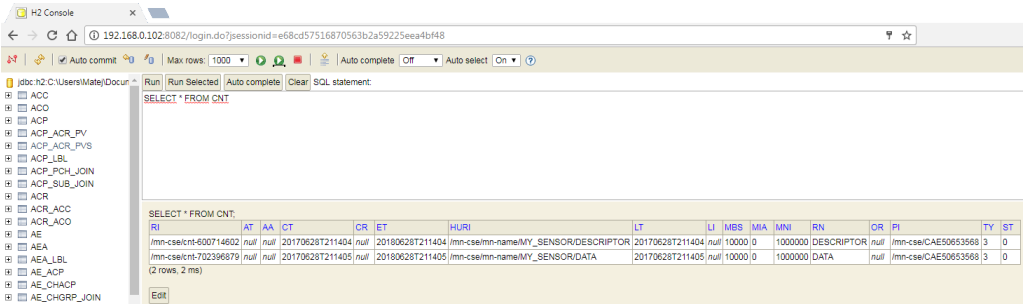
5.1 Orodja za testiranje

Ustreznost implementiranih funkcionalnosti smo preverili s pomočjo različnih orodij, ki bodo opisana v nadaljevanju magistrske naloge.

Platforma OM2M shranjuje podatke v podatkovno bazo H2. Med samo uporabo in testiranjem smo ugotovili, da spletni vmesnik, ki je namenjen pregledu podatkov (pregled aplikacij, vsebine, naročnin itd.), ob velikih količinah ne deluje tako, kot bi želeli. Vmesnik je počasen in neodziven, zato smo preverjanje vsebine izvajali s pomočjo spletnega vmesnika, ki je vključen v namestitev podatkovne baze H2.

Orodje **H2 Console** [27] je spletni vmesnik (slika 5.1), s katerim lahko pregledujemo vsebino podatkovne baze H2. Vmesnik je na naslovu *imeRacunalnika:8082/login.jsp*. OM2M platforma zapisuje podatke v obliki 'MVStore – končnica ".mv.db"', kar je značilno za H2 verzije 1.4.x. Pri odpiranju podatkovne baze moramo biti pozorni na JDBC URL, ki mora na koncu vsebovati parameter "MV_STORE=TRUE". V nasprotnem primeru orodje kreira

prazno podatkovno bazo oblike "*.h2.db".



Slika 5.1: Spletni vmesnik orodja H2 Console

Vrednosti, ki jih uporabimo za odpiranje podatkovne baze OM2M, so naslednje:

- Saved Settings: Generic H2 (Embedded)
- Driver Class: org.h2.Driver
- JDBC URL: jdbc:h2:\<potDoH2>
  Pot do H2 baze podamo brez končnice. V primeru, da je podatkovna baza poimenovana indb.mv.db in je na "C:\OM2M\", je JDBC URL enak "jdbc:h2:C:\OM2M\indb;MV_STORE=TRUE"
- User Name in Password: om2m

Za potrebe testiranja vtičnika za pisanje v podatkovno bazo MongoDB smo potrebovali orodje za simulacijo naprav (senzorjev - HTTP zahtevkov). Obstaja veliko obstoječih rešitev, kot so npr.: Apache JMeter, AutoSIM itd. Zaradi kompleksnosti omenjenih orodij smo za potrebe testiranja razvili svoje testno orodje. Orodje je sestavljeno iz dveh Javanskih razredov *main* in *tempSensor*. Razred *main* vsebuje spremenljivki *dataPerSec* in *numOfSensors*. *DataPerSec* se uporablja za nastavitev števila podatkov, ki jih simulator pošlje v eni sekundi, medtem ko *numOfSensors* predstavlja število

senzorjev, ki jih želimo simulirati. Simulator pri svojem delovanju uporablja knjižnico *async-http-client* [29], ki omogoča izvajanje asinhronih HTTP zahtevkov. Za zagotavljanje izvajanja programske logike vsako sekundo uporabljamo funkcijo *scheduleAtFixedRate* razreda *ScheduledExecutorService*, ki mu kot parameter podamo instanco razreda *tempSensor*, časovni zamik, periodo izvajanja in časovno enoto (izsek 5.1).

```
1 ScheduledExecutorService executor = Executors.
    newScheduledThreadPool(4);
2 TempSensor ts = new TempSensor(dataPerSec, numOfSensors,
    asyncHttpClient);
3 t = executor.scheduleAtFixedRate(ts, 0, 1, TimeUnit.SECONDS);
```

Izsek 5.1: Primer uporabe razreda ScheduledExecutorService (izvorna koda)

Razred *tempSensor* vsebuje konstruktor, ki izvaja klic funkcije *createBoundRequestBuilder*. Funkcija s pomočjo razreda *RequestBuilder* skrbi za kreiranje instance objekta *Request*. Kot parameter pri kreiranju instance podamo vrsto zahtevka, ki je v našem primeru "POST". Na novo kreirani instanci objekta izvedemo funkcije *setUrl* (naslov, kamor pošiljamo zahtevek), *setBody* (vsebina zahtevka) in *addHeader* (dodatni parametri zaglavja – Content-Type in X-M2M-Origin). Izvedemo klic funkcije *build*, ki vrača novo instanco objekta *BoundRequestBuilder*. Objekt *BoundRequestBuilder* omogoča uporabo metode *execute*, ki skrbi za pošiljanje podatkov. Razred *tempSensor* implementira razred *Runnable*, kar pomeni, da je treba prepisati (ang. override) funkcijo *run*. Funkcija *run* vsebuje implementacijo logike pošiljanja HTTP zahtevkov. Po vsakem poslanem HTTP zahtevku v programsko konzolo izpišemo trenutni čas. Ker se pošiljanje zahtevkov izvaja asinhrono, funkcija *onCompleted* skrbi za izpis časa, ko platforma OM2M obdela zahtevek in simulatorju pošlje odgovor.

Orodje **MongoDB shell** [30] (slika 5.2) omogoča izvajanje poizvedb, manipulacijo nad podatki in izvajanje administrativnih operacij na podatkovni bazi MongoDB. Orodje se namesti skupaj s podatkovno bazo MongoDB za okolje Windows. Primer ukazov: *db* (vrne seznam podatkovnih baz), *use*

<*imeBaze*> (izbere podatkovno bazo), *show collections* (vrne seznam vseh tabel v izbrani podatkovni bazi), *db.imetabele.find()* (vrne podatke tabele "imetabele") itd.

Slika 5.2: Orodje MongoDB shell

5.2 Priprava testnega okolja

Pri testiranju smo želeli dobiti čim bolj realno sliko delovanja novo razvitega vtičnika pisanja v podatkovno bazo MongoDB, zato smo testiranje izvajali s pomočjo treh računalnikov, na katerih smo izvajali orodje za simulacijo naprav.

Vsi računalniki so se med seboj razlikovali po vgrajeni strojni opremi, medtem ko je en računalnik deloval na drugem operacijskem sistemu kot preostala dva. Zaradi bolj jasne in preproste predstavitve smo računalnike poimenovali A, B in C. Specifikacije strojne in programske opreme so prikazane v tabeli 5.1.

Tabela 5.1: Primerjava protokolov TCP in UDP

Računalnik	Procesor	Pomnilnik	Operacijski sistem
A	AMD FX 8320 (3.5Ghz) – 8 jeder	16 GB	Windows 10 (64bit)
B	Intel i5 M540 (2.53GHz) – 2 jedri	4 GB	Windows 10 (64bit)
C	Intel i5 (2,4GHz) – 2 jedri	8 GB	macOS 10.12 Sierra

Računalnik A je po specifikacijah najbolj zmogljiv, zato smo na računalniku

izvajali platformo OM2M ter vse potrebne instance podatkovnih baz, ki smo jih uporabljali pri testiranju. Prav tako smo računalnik A uporabili tudi za pošiljanje podatkov v platformo OM2M. Pri testiranju smo zanemarili vpliv internetne povezave. Računalnika B in C sta bila v skupno omrežje povezana preko brezžične povezave, računalnik A pa preko žične povezave. Usmerjevalnik, ki je skrbel za notranje omrežje, je omogočal hitrosti do 1 Gbit/s.

Zaradi sprememb programske kode oz. dopolnitev funkcionalnosti platforme OM2M moramo ponovno prevesti in zgraditi aplikacijo. Ker vtičniki platforme OM2M vsebujejo XML datoteke POM, lahko to storimo s pomočjo orodja Maven. Prednost orodja Maven je ta, da lahko v datoteko POM zapišemo vse zahteve po paketih, ki so nujno potrebni za prevajanje aplikacije. Več informacij o sistemu Maven in njegovi uporabi je dosegljivih na uradni spletni strani. Končno obliko aplikacije smo prenesli in zagnali na računalniku A. Na isti računalnik smo namestili podatkovno bazo MongoDB (verzija 3.4.3) in jo zagnali.

Pred začetkom testiranja je bilo treba določiti strukturo podatkov, ki jih bodo pošiljali senzorji. Odločili smo se, da bomo v sklopu testiranja simulirali podatke, ki vsebujejo naslednje atribute: *sensorId* (identifikator senzorja), *category* (tip temperaturnega senzorja), *tempData* (temperatura okolja), *tempUnit* (enota temperature), *humidity* (vlažnost okolja), *city* (mesto zajema podatka) ter *country* (država zajema).

Pred začetkom testiranja smo v platformi OM2M izdelali strukturo aplikacije. Strukturo smo izdelali s pomočjo REST zahtevkov, ki smo jih pošiljali s pomočjo vtičnika Restlet Client – REST API Testing. Vtičnik smo namestili v spletni brskalnik Google Chrome. V nadaljevanju A predstavlja ime računalnika, na katerem se izvaja platforma OM2M.

Prvi korak pri izdelavi strukture aplikacije je kreiranje glavnega sklopa naprave. Na spletni naslov http://A:8080/~/mn-cse smo poslali POST zahtevek z vsebino:

```
1 <m2m:ae xmlns:m2m="http://www.onem2m.org/xml/protocols" rn="
```

```
    MY_SENSOR" >
2     <api>app-sensor</api>
3     <lbl>Type/sensor Category/temperature Location/home</lbl>
4     <rr>false</rr>
5 </m2m:ae>
```

Naslednji korak je kreiranje sklopa "Descriptor". Na spletni naslov http://A:8080/~/mn-cse/mn-name/MY_SENSOR smo poslali POST zahtevek z vsebino:

```
1 <m2m:cnt xmlns:m2m="http://www.onem2m.org/xml/protocols" rn="
    DESCRIPTOR">
2 </m2m:cnt>
```

Sledi kreiranje instance sklopa "Descriptor", ki hrani metapodatke vseh atributov. Na spletni naslov http://A:8080/~/mn-cse/mn-name/MY_SENSOR/DESCRIPTOR smo poslali POST zahtevek z vsebino:

```
1 <m2m:cin xmlns:m2m="http://www.onem2m.org/xml/protocols">
2     <cnf>application/xml</cnf>
3     <con>
4         &lt;obj&gt;
5           &lt;str name=&quot;sensorId&quot; val=&quot;Sensor ID&
    quot;/&gt;
6             &lt;str name=&quot;category&quot; val=&quot;
    Temperature Sensor&quot;/&gt;
7             &lt;int name=&quot;tempData&quot; val=&quot;Temp
    data&quot;/&gt;
8             &lt;str name=&quot;tempUnit&quot; val=&quot;Temp
    unit&quot;/&gt;
9           &lt;int name=&quot;humidity&quot; val=&quot;Humidity
    data&quot;/&gt;
10        &lt;str name=&quot;city&quot; val=&quot;City&quot;/&gt;
11          &lt;str name=&quot;country&quot; val=&quot;Country&
    quot;/&gt;
12            &lt;op name=&quot;getValue&quot; href=&quot;/mn-cse/
    mn-name/MY_SENSOR/DATA/la&quot;
13          in=&quot;obix:Nil&quot; out=&quot;obix:Nil&quot; is=&
    quot;retrieve&quot;/&gt;
```

```
14          &lt;/obj&gt;
15      </con>
16 </m2m:cin>
```

Naslednji korak skrbi za izdelavo sklopa "Data", ki hrani sprejete podatke zunanjih senzorjev. Na naslov http://A:8080/~/mn-cse/mn-name/MY_SENSOR smo poslali POST zahtevek z vsebino:

```
1 <m2m:cnt xmlns:m2m="http://www.onem2m.org/xml/protocols" rn="
    DATA">
2 </m2m:cnt>
```

Po izvedbi vseh zgornjih korakov je platforma OM2M pripravljena za uporabo.

5.3 Testiranje

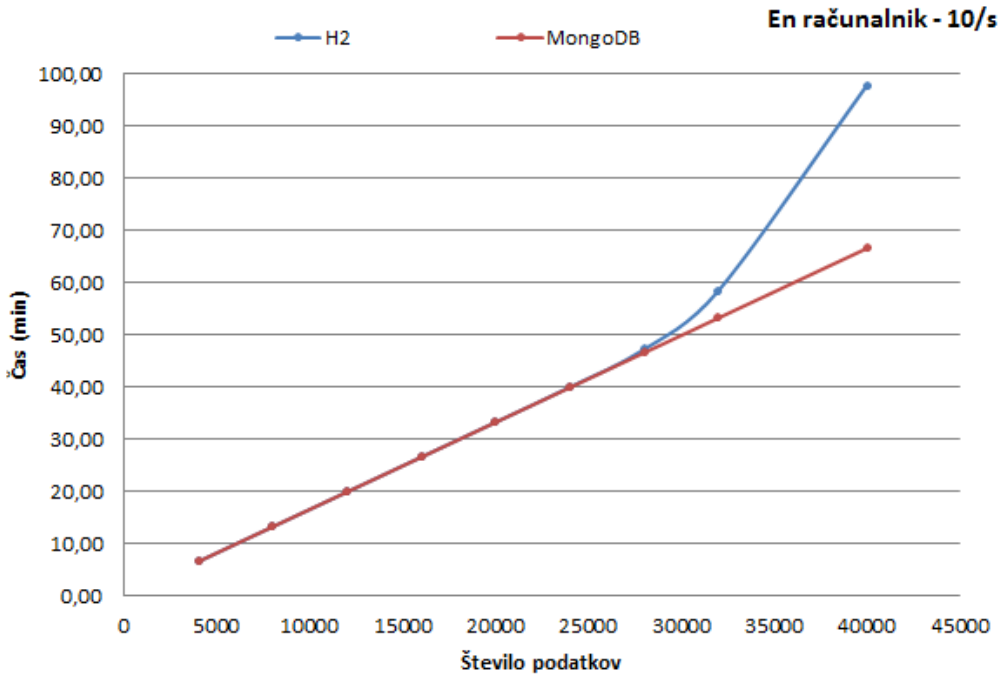
Že v prejšnjih poglavjih smo omenili, da je podatkovna baza MongoDB že s samega strukturnega sistema bolj primerna za shranjevanje velikih količin podatkov kot podatkovna baza H2. S pomočjo simulatorja smo v platformo OM2M pošiljali različne količine podatkov pri hitrosti 10 in 20 podatkov/s. Testiranje smo izvajali s pomočjo enega oz. kasneje s pomočjo več računalnikov hkrati. Na začetku smo podatke pošiljali iz enega računalnika, nato iz dveh in nazadnje iz vseh treh hkrati. Celotno testiranje smo izvedli na obeh podatkovnih bazah posebej.

Pričakovani čas pošiljanja podatkov je čas, ki bi ga morala podatkovna baza potrebovati za obdelavo vseh prejetih podatkov. Izračunamo ga tako, da število poslanih podatkov delimo s hitrostjo pošiljanja podatkov (tabela 5.2). Ob vsakem novem testiranju smo pobrisali vsebino H2 in MongoDB. Izvedli smo tudi ponoven zagon podatkovne baze in platforme zato, da pri testiranju ni prihajalo do razlik.

Tabela 5.2: Pričakovani čas obdelave podatkov v minutah

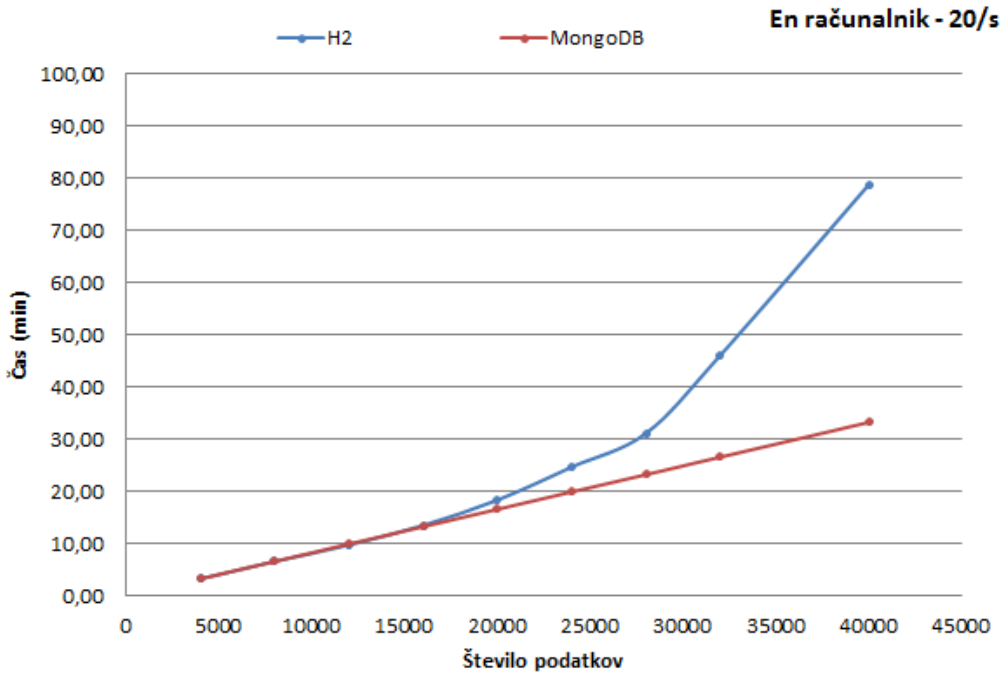
Število podatkov	Čas (10/s)	Čas (20/s)
4000	6,65	3,32
8000	13,32	6,65
1200	019,98	9,98
1600	026,65	13,32
2000	033,32	16,65
2400	039,98	19,98
2800	046,63	23,30
3200	053,32	26,65
4000	066,67	33,32

Testiranje s pomočjo enega računalnika (A) smo izvajali pri hitrosti pošiljanja 10 in 20 podatkov/s. Testiranje smo začeli s podatkovno bazo H2 pri hitrosti pošiljanja 10 podatkov/s in štiri tisoč podatkih (slika 5.3). Pri vsakem naslednjem testu smo količino povečali za štiri tisoč. To smo počeli vse do 32 tisoč podatkov, ko smo opazili, da čas obravnave začenja odstopati od pričakovanega časa. Pri količini 32 tisoč podatkov je bil pričakovan čas obravnave 53 min, medtem ko je podatkovna baza H2 potrebovala 58 min. Glede na to, da je bilo odstopanje pribl. 5 min in da se je testiranje izvajalo 58 min, smo se odločili, da testiranje izvedemo še za 40 tisoč podatkov. Iz slike 5.3 lahko razberemo, da se je čas zelo povečal, in sicer na 97 min. Pričakovani čas za 40 tisoč podatkov je 66 min, kar pomeni, da čas obravnave pri podatkovni bazi H2 odstopa za 32 %. Po testiranju podatkovne baze H2 smo izvedli testiranje MongoDB. Vsi podatki so bili obravnavani v pričakovanem času.



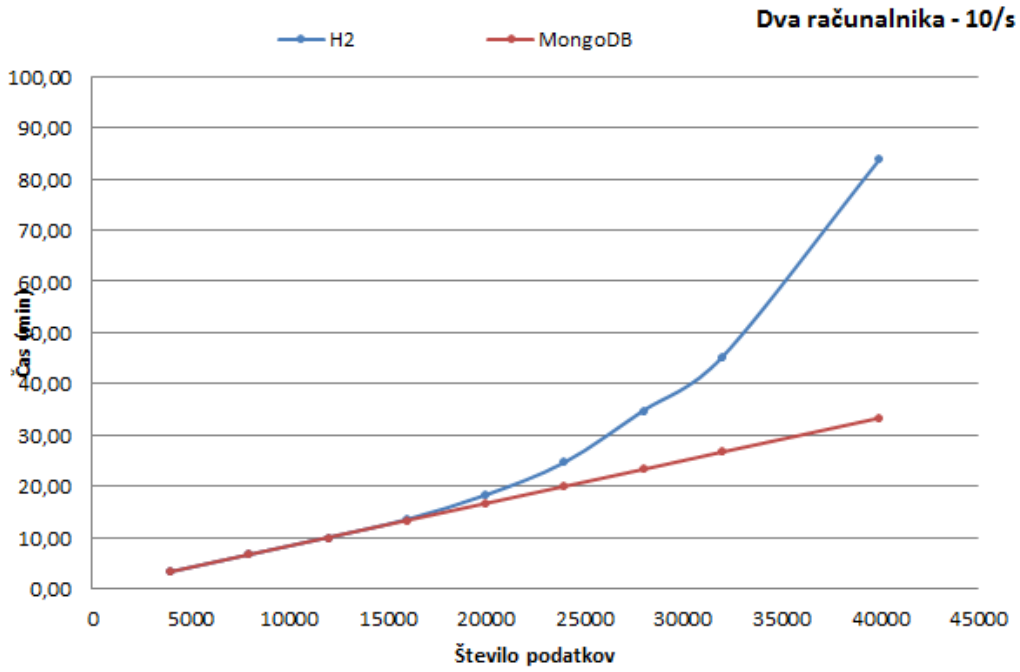
Slika 5.3: Graf časa obravnave podatkov (10 podatkov/s) - A)

Testiranje smo nadaljevali pri hitrosti pošiljanja 20 podatkov/s. Na sliki 5.4 lahko vidimo, da v primeru hitrejšega pošiljanja podatkov pride do večjega odstopanja že pri manjši količini poslanih podatkov (od 15 tisoč naprej). Med 20 tisoč in 28 tisoč poslanih podatkov odstopanje počasi narašča. Bistvena razlika nastane med 28 tisoč in 40 tisoč poslanih podatkih. Pričakovani čas za 40 tisoč podatkov je 78 min, kar pomeni, da čas obravnave pri podatkovni bazi H2 odstopa za 58 %. Po testiranju podatkovne baze H2 smo izvedli testiranje MongoDB. Vsi podatki so bili obravnavani v pričakovanem času.



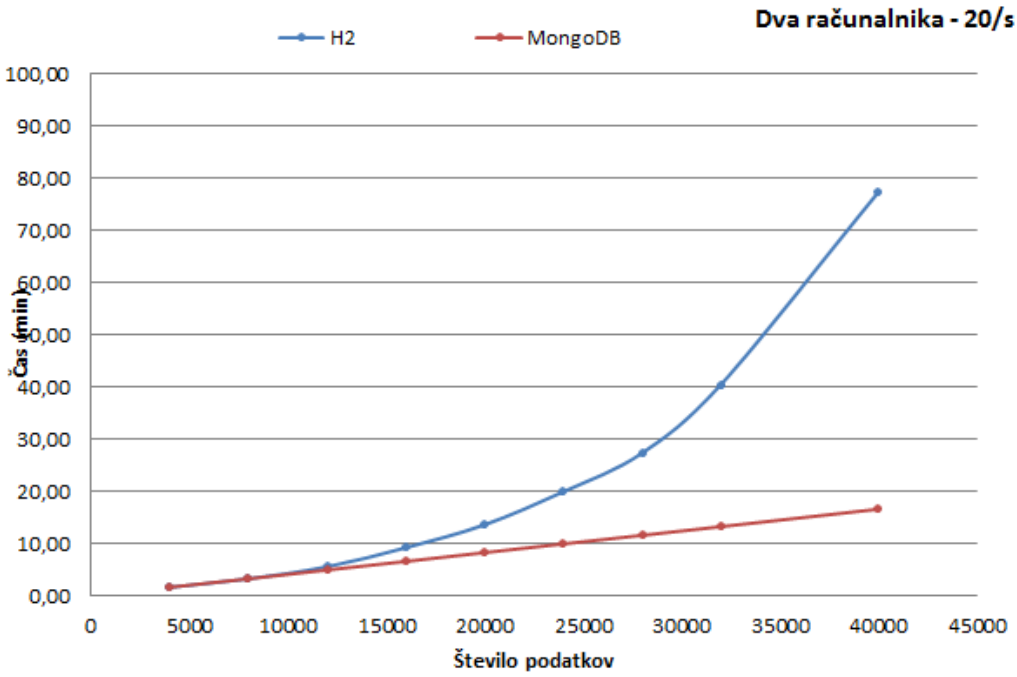
Slika 5.4: Graf časa obravnave podatkov (20 podatkov/s) - A)

Testiranje s pomočjo dveh računalnikov (A, B) smo začeli pri 10 podatkih/s. Količino poslanih podatkov smo razdelili med dva računalnika. To pomeni, da je pri štiri tisoč podatkih vsak izmed računalnikov poslal dva tisoč podatkov. Testiranje smo začeli s podatkovno bazo H2 in nadaljevali z MongoDB. Iz slike 5.5 je razvidno, da razlika v času nastane pri pribl. 18 tisoč podatkih in se strmo povečuje. Izkaže se, da ima H2 probleme pri obravnavi podatkov, medtem ko podatkovna baza MongoDB sledi pričakovanemu času. Pričakovani čas za 40 tisoč podatkov je 33 min, kar pomeni, da čas obravnave pri podatkovni bazi H2 odstopa za 61 %. Odstopanje med enim računalnikom (20 podatkov/s) in dvema računalnikoma (10 podatkov/s) je praktično zanemarljivo.



Slika 5.5: Graf časa obravnave podatkov (10 podatkov/s) - A, B

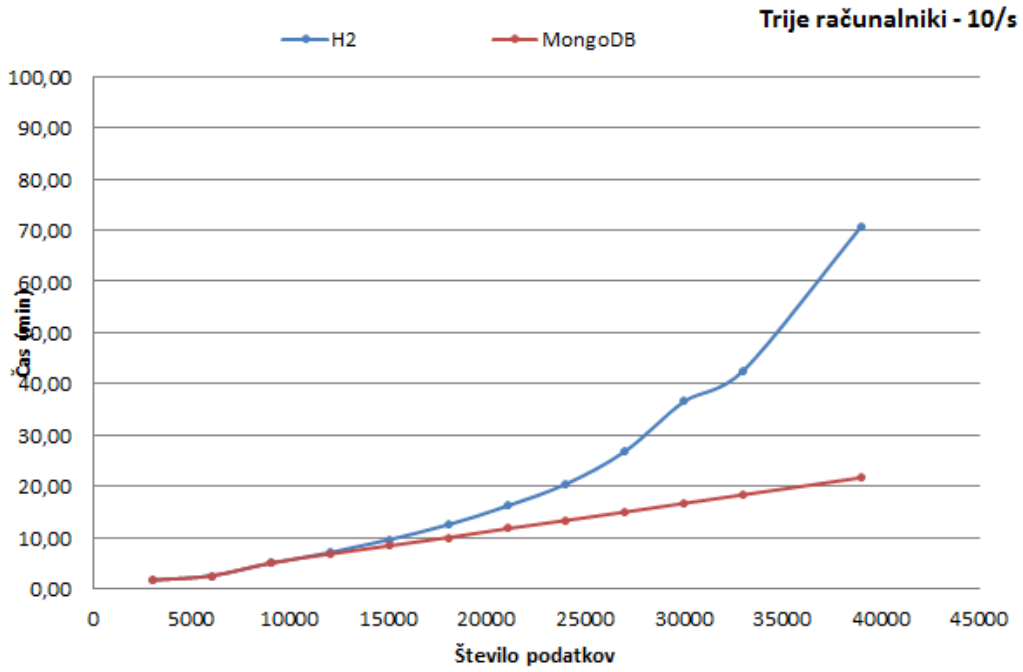
Pri hitrosti 20 podatkov/s se razlike pojavijo že pri količini pribl. 12 tisoč podatkov (slika 5.6). Razlika med pričakovano in dejansko hitrostjo pri 40 tisoč podatkih je približno 80 %. Podatkovna baza H2 potrebuje približno 77 min za obdelavo vseh podatkov, kar je za 60 min počasneje, kot za isto količino potrebuje MongoDB. Tudi tukaj lahko vidimo, da je podatkovna baza MognoDB hitrejša kot H2, saj je vse podatke obdelala v pričakovanem času.



Slika 5.6: Graf časa obravnave podatkov (20 podatkov/s) - A, B

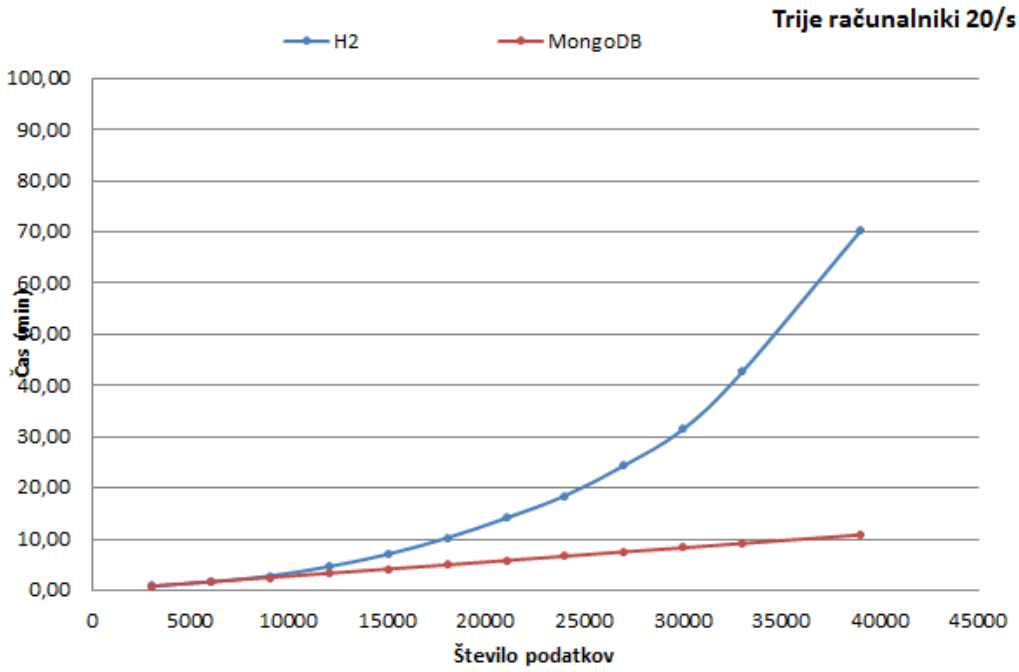
Testiranje s pomočjo treh računalnikov (A, B, C) smo začeli pri 10 podatkih/s. Zaradi lažje porazdelitve podatkov med tri računalnike smo test začeli pri tri tisoč podatkih. To pomeni, da smo na začetku iz vsakega računalnika poslali tisoč podatkov. Skozi test smo korak poslanih podatkov povečevali po tri tisoč, in sicer vse do 33 tisoč podatkov. Zaradi časovne kompleksnosti smo test zaključili pri 39 tisoč podatkih.

Iz slike 5.7 vidimo, da sta obe podatkovni bazi do 12 tisoč podatkov le-te obdelali v pričakovanem času. Pri vseh nadaljnjih testih je čas obdelave H2 začel odstopati od pričakovanega časa. Največja razlika med podatkovnima bazama se je pojavila pri 39 tisoč podatkih, ko je odstopanje narastlo na 70 %.



Slika 5.7: Graf časa obravnave podatkov (10 podatkov/s) - A, B, C

Zadnji test smo izvedli pri hitrosti 20 podatkov na sekundo. Slika 5.8 prikazuje odstopanje že pri količini nad pribl. devet tisoč podatki. Gre za najhitrejše odstopanje izmed vseh izvedenih testov. Z vsakim novim korakom čas obravnave podatkov hitro narašča. Omembe vredna razlika se začne pri približno 12 tisoč poslanih podatkih (28 %) in se nadaljuje vse do 39 tisoč poslanih podatkov (84 %).



Slika 5.8: Graf časa obravnave podatkov (20 podatkov/s) - A, B, C

Poglavje 6

Sklepne ugotovitve

Namen magistrske naloge je bil razširiti odprtokodno IoT platformo OM2M na tak način, da bo omogočala hranjenje velikih količin podatkov. Zaradi večje skalabilnosti smo pri razvoju namesto relacijske podatkovne baze H2 uporabili nerelacijsko podatkovno bazo MongoDB.

Prva težava, na katero smo naleteli pri razvoju in testiranju, je bila pisanje v podatkovno bazo H2. Logika, ki se izvede ob vsakem novem zahtevku, zapiše podatke v tabelo *CIN*. Poleg tega izvede posodobitev atributa *st* v tabeli *CNT*. Atribut *st* predstavlja število vseh *CIN* zapisov, ki pripadajo posameznemu zapisu v tabeli *CNT*. Ob vsakem novem zahtevku se vrednost atributa poveča za 1. Ker nove zahtevke pošiljamo asinhrono, lahko pride do težav pri zaklepanju vrstice v tabeli *CNT*. Primer: prvi oz. n-ti zahtevek zaklene vrstico v tabeli *CNT*, medtem pa platforma OM2M sprejme še n zahtevkov, ki želijo zakleniti isto vrstico. Ker se prejšnji zahtevek še ni izvedel, je vrstica še vedno zaklenjena. Nov zahtevek počaka nekaj časa in v primeru, da mu ne uspe zakleniti vrstice v določenem časovnem obdobju, vrne napako pri delovanju (timeout). Časovno obdobje lahko poljubno nastavimo, vendar s tem samo prestavljamo trenutek, ko bo prišlo do napake. Mislimo, da gre v tem primeru za arhitekturno težavo. Namesto vsakokratnega povečevanja atributa *st* bi lahko izvedli poizvedbo nad tabelo, ki bi vrnila število vseh *CIN* zapisov. Glede na to, da bi šlo za preprosto poizvedbo, bi bilo takšno

delovanje hitrejše. Omenjena težava je onemogočala testiranje delovanja H2, saj so se pri količini 18 tisoč prejetih podatkov začele pojavljati težave in potreben je bil ponovni zagon platforme OM2M. Delovanje smo odstranili, saj ni bilo bistveno za naše testiranje.

Med testiranjem H2 smo opazili, da čas obdelave v določenih primerih odstopa od pričakovanega časa. Več, kot je podatkov, večje je odstopanje, kar lahko vidimo na grafih v poglavju o testiranju. Opazimo, da čas obdelave podatkov v podatkovni bazi H2 ni odvisen samo od hitrosti pošiljanja, ampak tudi od števila računalnikov, iz katerih smo podatke pošiljali. To je logična posledica, saj, če povečamo število računalnikov, s tem povečamo tudi hitrost pošiljanja, kljub temu da število poslanih podatkov ostaja enako. Primer: če iz treh računalnikov pošiljamo podatke s hitrostjo 10/s, je to enako, kot če bi enako količino podatkov pošiljali iz enega računalnika pri hitrosti 30/s. Ker je imela podatkovna baza H2 težave pri obdelavi podatkov, smo se odločili, da uporabimo eno izmed nerelacijskih podatkovnih baz. Odločili smo se za MongoDB, saj je bolj skalabilna kot H2. Da MongoDB deluje hitreje kot H2, smo to preverili tako, da smo izvedli enake teste kot pri H2. Iz testov je razvidno, da MongoDB brez težav obdela vse podatke v pričakovanem času. To velja tudi za večje količine. Na podlagi vseh rezultatov lahko trdimo, da bi MongoDB brez težav obdelala tudi veliko večje količine podatkov.

Pri uporabi platforme smo opazili, da je spletni vmesnik pri veliki količini podatkov neodziven. Težavo smo rešili tako, da smo implementirali funkcionalnost paginacije.

Platforma OM2M je ena izmed edinih odprtokodnih platform IoT, ki implementira standard oneM2M. Funkcionalnost vtičnikov nam omogoča, da jo lahko poljubno nadgradimo v skladu s svojimi potrebami. Kljub temu ima platforma določene slabosti. Razvijalcem, ki želijo posegati v jedro platforme, je na voljo zelo malo dokumentacije, prav tako pa ima tudi uradni forum relativno malo objav. Možnosti za nadgradnjo obstoječe funkcionalnosti so praktično neomejene: zamenjava celotne podatkovne baze H2 s poljubno nerelacijsko bazo, implementacija vtičnika za podporo kompleksnih

SQL operacij, implementacija vtičnika za analizo shranjenih podatkov itd.

Literatura

[1] Internet of Things (IoT). Dostopno na: http://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT Dostopno 1. 7. 2017

[2] Gartner, Gartner's 2015 hype cycle for emerging technologies identifies the computing innovations that organizations should monitor (Avgust 2015). Dostopno na: http://www.gartner.com/newsroom/id/3114217 Dostopno 1. 7. 2017

[3] D. Evans. The Internet of Things - How the Next Evolution of the Internet Is Changing Everything. Cisco IBSG, 2011.

[4] Internet protocol (IPv4), darpa internet program protocol specification. Dostopno na: https://tools.ietf.org/html/rfc791 Dostopno 1. 7. 2017

[5] IPv6 (Internet Protocol Version 6). Dostopno na: http://searchenterprisewan.techtarget.com/definition/IPv6 Dostopno 1. 7. 2017

[6] Global Development Survey 2016 V1. Dostopno na: http://www.evansdata.com/reports/viewRelease.php?reportID=40 Dostopno 1. 7. 2017

[7] C. Pereira, A. Aguiar. Towards efficience mobile M2M communications: Survey and open challenges. *Sensors*, 2014, str. 19582-19608, Doi:10.3390/s141019582.

[8] The European Telecommunications Standards Institute. Dostopno na: http://www.etsi.org/WebSite/homepage.aspx Dostopno 1. 7. 2017

[9] V. Galetić, I. Bojić, M. Kušek, G. Ježić, S. Dešić, D. Huljenić. Basic principles of Machine-to-Machine communication and its impact on telecommunications industry. Mipro, Maj 23-27, Opatija, Hrvaška, 2014

[10] D. Lucić, A. Carić, I. Lovrek. Standardisation and Regulatory Context of Machineto- Machine Communication, 2015

[11] oneM2M Prepares for August 2014 Release. Dostopno na: http://www.onem2m.org/press/2014-0411%20TP10_Release_final2.pdf Dostopno 1. 7. 2017

[12] HTTP/2.0. Dostopno na: https://www.rfc-editor.org/rfc/pdfrfc/rfc7540.txt.pdf Dostopno 1. 7. 2017

[13] R. Oppliger. SSL and TLS: Theory and Practice. *eSECURITY Technologies*, 2009, str. 19582-19608, ISBN-13 978-1-59693-447-4.

[14] The Constrained Application Protocol (CoAP). Dostopno na: https://tools.ietf.org/html/rfc7252 Dostopno 1. 7. 2017

[15] User Datagram Protocol. Dostopno na: https://tools.ietf.org/pdf/rfc768.pdf Dostopno 1. 7. 2017

[16] Transmission Control Protocol. Dostopno na: https://tools.ietf.org/pdf/rfc793.pdf Dostopno 1. 7. 2017

[17] MQTT Version 3.1.1. Dostopno na: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf Dostopno 1. 7. 2017

[18] Machine-to-Machine communications (M2M); Functional architecture. Dostopno na: http://www.etsi.org/deliver/etsi_ts/102600_102699/102690/01.01.01_60/ts_102690v010101p.pdf Dostopno 1. 7. 2017

[19] L.A. Grieco, M. Ben Alaya, T. Monteil, K. Drira. Architecting information centric ETSI-M2M system. *Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 2014, Doi: 10.1109/PerComW.2014.6815203.

[20] M. Ben Alaya, Y. Banouar, T. Monteil, C. Chassot, K. Drira. OM2M: Extensible ETSI-compliant M2M service platform with self-configuration capability, *Procedia Computer Science*, 2014, vol. 23., str. 1079-1086. Doi: 10.1016/j.procs.2014.05.536.

[21] OM2M Connecting things. Dostopno na: `https://www.eclipse.org/om2m/` Dostopno 1. 7. 2017

[22] J. Swetina, G. Lu, P. Jacobs, F. Ennesser, J. Song. Toward a standardized common M2M service layer platform: Introduction to oneM2M. *IEEE Wireless Communications*, 2014, str. 20-26.

[23] oneM2M technical specification. Dostopno na: `http://www.onem2m.org/images/files/deliverables/TS-0001-oneM2M-Functional-Architecture-V-2014-08.pdf` Dostopno 1. 7. 2017

[24] J. McAffer, P. VanderLei, S. Archer. *OSGi and Equinox: Creating Highly Modular Java Systems.*, Addison-Wesley, 2010 ISBN: 978-0-321-58571-4.

[25] OSGi (Open Service Gateway Initiative). Dostopno na: `http://searchnetworking.techtarget.com/definition/OSGi` Dostopno 1. 7. 2017

[26] OSGI Architecture. Dostopno na: `http://searchnetworking.techtarget.com/definition/OSGi` Dostopno 1. 7. 2017

[27] H2 Database Engine. Dostopno na: `http://www.h2database.com/html/main.html` Dostopno 1. 7. 2017

[28] MongoDB. Dostopno na: https://www.mongodb.com/ Dostopno 1. 7. 2017

[29] AsyncHttpClient. Dostopno na: https://github.com/AsyncHttpClient/async-http-client/tree/master/client/src Dostopno 1. 7. 2017

[30] MongoDB shell. Dostopno na: https://docs.mongodb.com/getting-started/shell/client/ Dostopno 1. 7. 2017

[31] Teach-ich - standards. Dostopno na: http://www.teach-ict.com/as_a2_ict_new/ocr/A2_G063/333_networks_coms/standards/miniweb/pg2.htm Dostopno 1. 7. 2017