

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
ELEKTROTEHNIČKI FAKULTET

Sveučilišni studij

SERVER ZA PRIJENOS PODATAKA IZ
POSTROJENJA

Diplomski rad

Damir Jelić

Osijek, 2015.

Sadržaj

1	Uvod	1
2	Pregled korištenih programskih jezika i idioma	2
2.1	Haskell	3
2.2	Python	7
2.3	Arduino	9
2.4	Firmata	10
2.5	JSON-RPC	11
3	Implementacija	12
3.1	Server	12
3.2	Klijent	18
3.3	Postrojenje	22
4	Zaključak	30

Literatura

Sažetak

Životopis

Prilozi

1. UVOD

Cilj je diplomskog rada ispitati mogućnost korištenja jeftinih mikroupravljača za jednostavne poslove automatizacije te omogućiti udaljeno upravljanje postrojenjem preko *socket servera*.

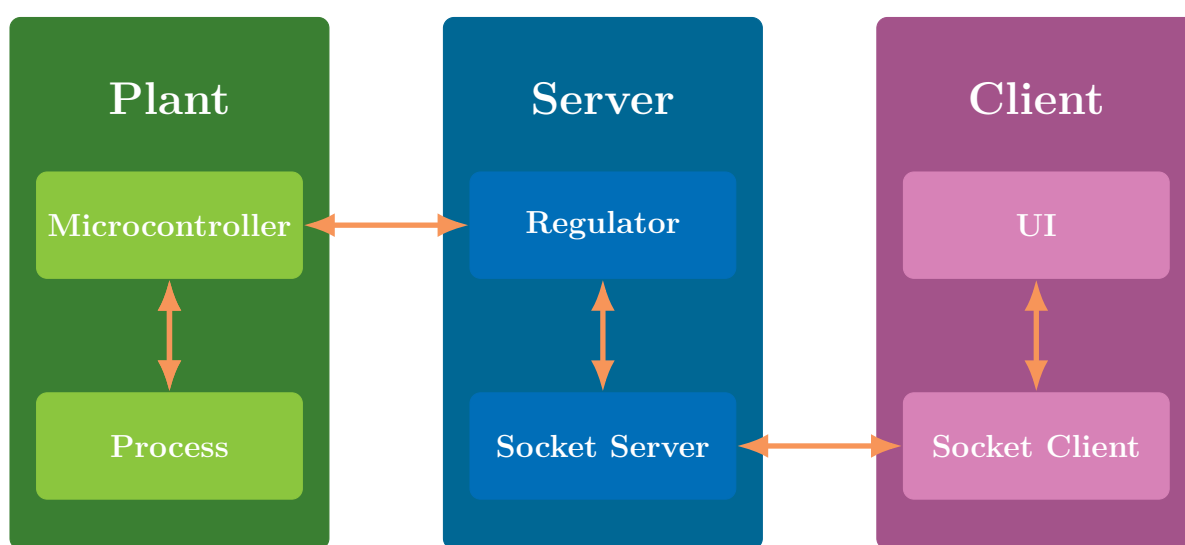
Kako bi se utvrdio cilj diplomskog rada, pomoću mikroupravljača je simulirano postrojenje. Komunikacija prema računalu je ostvarena putem *USB* sučelja. Računalo šalje mikroupravljaču naredbe pomoću komunikacijskog protokola. *Socket server* izrađen je tako da s jedne strane koristi navedeni protokol da bi komunicirao s mikroupravljačem, a s druge strane omogućuje klijentima upit u stanje postrojenja i postavljanje referentne varijable postrojenja.

Za mikroupravljač odabran je *Arduino Uno* koji preko USB-serijskog sučelja i preko *Firmata* protokola komunicira s računalom. Na *Arduino* spojena je vodena pumpa i senzor koji mjeri razinu vode u boci. Sustav od dvije boce čini postrojenje, u jednoj boci održava se razina vode. Na računalu se nalazi *socket server* pisan u *Haskellu* koji upravlja mikroupravljačem. Testni klijent pisan u *Pythonu* spaja se na *server* te dohvaća mjernu veličinu (razinu vode u boci) i prikazuje trenutno stanje sustava.

Rad je podijeljen u dva dijela. U prvom dijelu razrađene su teorijske pretpostavke vezane uz temu. Važno je napomenuti kako je u ovom dijelu detaljno opisan proces razvoj *socket servera*. Drugi dio bavi se implementacijom postrojenja, regulatora, *socket servera* i popratnog klijenta.

2. PREGLED KORIŠTENIH PROGRAMSKIH JEZIKA I IDIOMA

U ovom su poglavlju detaljno opisani programski jezici, tehnike i komunikacijski protokoli koji su korištene pri izradi diplomskoga rada.



Slika 2.1.: Shematski prikaz rada

Na slici 2.1. je prikazan sinopsis diplomskoga rada koji je podijeljen u 3 dijela:

- Postrojenje
- Server
- Klijent

Kao što je već spomenuto u poglavlju 1, postrojenje se sastoji od mikroupravljača i dviju boca koje služe kao model za održavanje razine vode.

Server, koji je pisan u Haskellu, također se sastoji od dva dijela, regulatora i socket servera. Regulacijski dio s postrojenjem komunicira pomoću Firmata protokola, a socket server s klijentima komunicira pomoću JSON-RPC protokola.

Klijent koji je pisan u Pythonu sastoji se od korisničkog sučelja i socket klijenta koji periodički dohvaća trenutno stanje postrojenja od servera.

2.1 Haskell

Haskell je moderan, standardan, nije striktan, čisto funkcionalan programski jezik. Dizajniran je za širok spektar primjena, od numeričkih do simboličkih [1].

Haskell je nastao 1990. godine te nakon nekoliko godina razvoja glavne osobine bile su mu [2]:

- Statički pisan
- Sigurnost tipova
- Lijenost
- Klase tipova.

Statički pisan jezik zahtijeva da pri kompilaciji koda svi tipovi podataka budu poznati, što donosi niz prednosti kao primjećivanje grešaka prije nego što se program izvrši, efikasniji kod jer compiler zna unaprijed veličinu podataka, nije potrebno provjeravati tip podatka dok se program izvršava.

Sigurnost tipova proizlazi iz strogo statičkog pisanja. Ona osigurava funkcije od primanja podatak pogrešnog tipa. Takve će greške prepoznati compiler pri kompilaciji.

Lijenost (engl. *Lazy evaluation*) je svojstvo jezika pri kojem se izrazi evaluiraju tek kada je to potrebno. Rezultat komputacije računa se tek kada je on zatražen negdje dalje. Lazy evaluation omogućuje elegantan rad s beskonačnim poljima. Moguće je definirati polje s beskonačnim brojem članova, a da program ne zauzme svu memoriju računala. Pojedini će se elementi polja evaluirati ako, i samo ako, mu probamo pristupiti.

Klase tipova (engl. *Type classes*) nastale kako bi omogućile implementiranje preopterećenih aritmetičkih operatora i operatora jednakosti.

2.1.1 Mrežno programiranje

Za mrežno programiranje Haskell nudi modul **Network.Socket** [3], koji korisniku pruža sve standardne C funkcije za stvaranje UNIX socketa.



Slika 2.2.: Programski tok izrade socket servera

Na slici 2.2. je prikazan programski slijed izrade socket servera. Prvo se pomoću `socket` funkcije stvori socket određenih karakteristika. U **Network.Socket** modulu je funkcija definirana kao:

```
socket :: Family -> SocketType -> ProtocolNumber -> IO Socket
```

Iz definicije funkcije vidljivo je kako je ona zapravo ista kao i standardna C funkcija za stvaranje socketa [4, str.132]. Nakon što je stvoren socket njemu se treba dodijeliti lokalna adresa odabranog protokola, dodjeljivanje adrese postiže se s `bind` funkcijom čija definicija glasi:

```
bind :: Socket -> SockAddr -> IO ()
```

Slijedi pretvaranje stvorenog socketa u pasivni socket koji će čekati na ulazne konekcije. To se ostvaruje pomoću funkcije `listen` koja je definirana kao:

```
listen :: Socket -> Int -> IO ()
```

Na kraju je još potrebno pričekati konekciju klijenta što se ostvaruje s funkcijom `accept` koja blokira daljni tok programa dok se klijent ne spoji. Definicija funkcije glasi:

```
accept :: Socket -> IO (Socket, SockAddr)
```

Funkcija nam vraća novi socket s pomoću kojim možemo razmjenjivati podatke s klijentom i adresu klijenta. Razmjena se podataka može vršiti s `sendTo` i `recvFrom` no radi jednostavnosti možemo socket pretvoriti u handle te pisati i čitati podatke kao da se radi o datoteci. Za to postoji `socketToHandle` koja jednostavno primi socket i mode u kojem treba biti handle (read, write ili oboje) i vrati handle. Definicija joj glasi:

```
socketToHandle :: Socket -> IOMode -> IO Handle
```

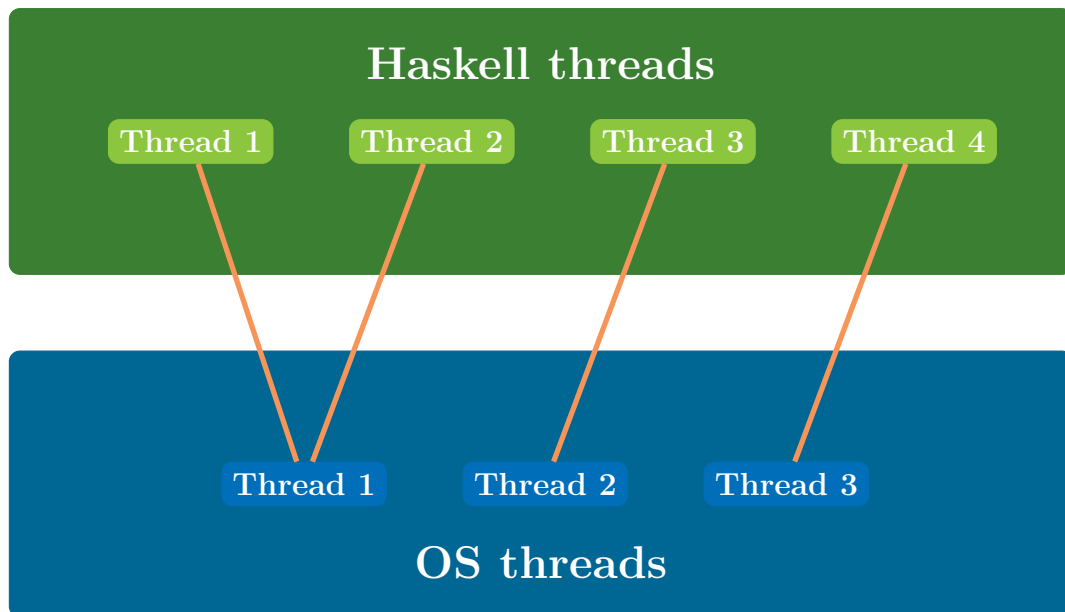
2.1.2 Istodobnost

Istodobnost (engl. *Concurrency*) je svojstvo sustava kojim se istodobno može izvršavati više računskih operacija ili komputacija.

Ekstenzija za istodobnost u Haskellu pojavila se 1996. godine koja uvodi dva nova idioma u Haskell [5]:

- Procese i mehanizme za stvaranje procesa
- Atomarno promjenjivo stanje

Istodobnost u Haskellu uglavnom je izvedena pomoću "Zelenih niti" (engl. *Green threads*). Zelene niti ne izvršavaju se u jezgri operativnog sustava već u *Haskell runtime-u*. Haskell ima hibridni višenitni model kojim se N Haskell niti mogu vezivati na M niti operativnog sustava.



Slika 2.3.: N:M višenitni model Haskell-a

Na slici 2.3. je prikazano kako se Haskell niti mapiraju na niti operativnog sustava. Interno se u Haskellu stvaranje nove niti pretvara u alokaciju strukture koja sprema trenutno stanje niti te se niti pretvaraju u jednu petlju.

Za stvaranje nove niti Haskell nudi `forkIO` funkciju koja je dio `Control.Concurrent` modula [6], a definirana je kao:

```
forkIO :: IO () -> IO ThreadId
```

Što znači da funkcija kao prvi argument prima komputaciju (engl. *computation*) i vraća novu komputaciju koja kao rezultat proizvodi `ThreadId`, što nam služi kao referenca na novu nit koju će funkcija stvoriti.

Osim niti za istodobno izvršavanje programa, bitna je i komunikacija između niti. Za komunikaciju između niti Haskell nudi mutabilne dijeljene varijable zvane `MVar` (Mutable Variables) koje se nalaze unutar `Control.Concurrent.MVar` modula [7]. One se mogu koristiti na razne načine:

- Sinkronizirane mutabilne varijable.
- Komunikacijski kanali između niti.
- Binarni semafori.

Nova mutabilna varijabla koja sadrži podatak proizvoljnog tipa može se napraviti s `newMVar`, definirana je kao:

```
newMVar :: a -> IO (MVar a)
```

Nakon što je varijabla stvorena, njom se može na razne načine manipulirati, između ostalog može se zapisati novi podatak u nju, pročitati podatak ili zamijeniti s nekim drugim podatkom.

Čitanje podatka bez njegove izmjene omogućuje nam `readMVar`:

```
readMVar :: MVar a -> IO a
```

Za zamjenu podatka s nekim drugim postoji `swapMVar` funkcija:

```
swapMVar :: MVar a -> a -> IO a
```

Za izmjenu više dijeljenih varijabli postoji `Control.Concurrent.Chan` [8] modul, koji sadržava implementaciju neograničenog komunikacijskog kanala.

Novi komunikacijski kanal stvara se s `newChan` funkcijom:

```
newChan :: IO (Chan a)
```

Funkcija stvara novi prazni komunikacijski kanal određenog tipa te ga vraća kao rezultat. Komunikacijski sprema vrijednosti i čuva ih sve dok ih se ne izvadi iz njega. Služi kao FIFO spremnik vrijednosti određenog tipa.

Novi podatak stavlja se u kanal `writeChan` funkcijom:

```
writeChan :: Chan a -> a -> IO ()
```

Funkcija kao argument prima komunikacijski kanal i podatak. Tip podatka mora odgovarati tipu podatka određenog pri stvaranju komunikacijskog kanala. Vrijednost se sprema u kanal te tamo ostaje sve dok se ona ne pročita `readChan` funkcijom:

```
readChan :: Chan a -> IO a
```

`readChan` funkcija kao argument prima komunikacijski kanal i vraća najstariji podatak koji se nalazi u kanalu.

2.2 Python

Python je interpretiran, objektno orijentiran, programski jezik visoke razine s dinamičkom semantikom. Podatkovne strukture visoke razine i dinamičko pisanje čine ga atraktivnim za vrlo brzo razvijanje aplikacija i za skriptni jezik za spajanje više postojećih komponenti. Python posjeduje jednostavnu i laku sintaksu koja povećava čitljivost i olakšava održavanje softvera [9].

2.2.1 Mrežno programiranje

Slično kao i Haskell, Python pruža sve standardne C funkcije za mrežno programiranje. Sve se bitne funkcije nalaze u `Socket` modulu [10].



Slika 2.4.: Programski tok izrade socket klijenta

Na slici 2.4. je prikazan programski tok izrade socket klijenta. Slično kao i sa serverom, prvo se stvori socket:

```
socket.socket()
```

Socket mora biti iste vrste kao i socket servera na koji se želimo spojiti. Nakon stvaranja socketa potrebno ga je spojiti na serverski kraj socketa. Spajanje na server vrši se pomoću `connect()` funkcije kojoj se predaje adresa i port socket servera:

```
socket.connect()
```

Nakon spajanja može započeti komunikacija između klijenta i servera. Komunikacija se može izvršiti pomoću standardnih C funkcija `send` i `recv` ili, kao i u Haskellu, socket se može pretvoriti u handle te se koristiti kao da je datoteka:

```
socket.makefile()
```

Izrada socket-klijenta daleko je lakša od servera, pogotovo jer se nerijetko ne moramo brinuti o više istovremenih konekcija.

2.2.2 Urwid

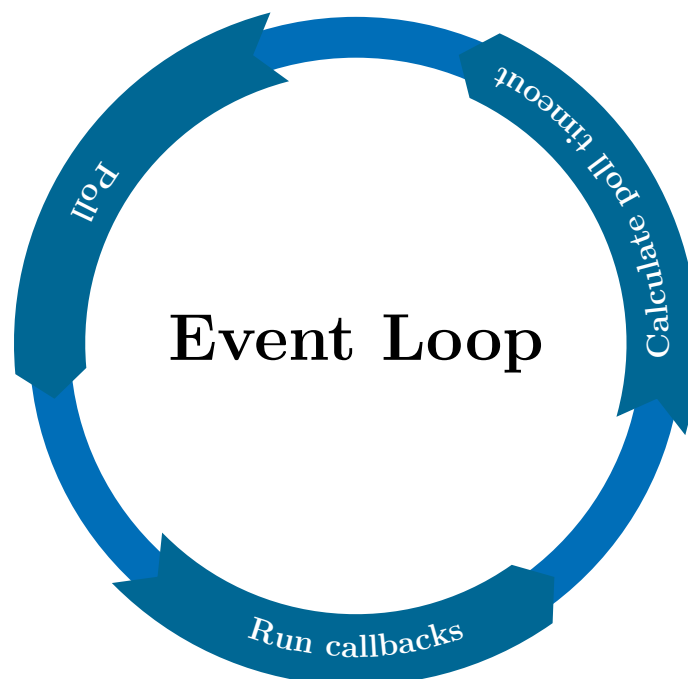
Urwid je Python biblioteka za izradu tekstualnih korisničkih sučelja. Urwid je alternativa za standardnu Curses biblioteku [11]. Interno koristi Curses biblioteku, no Urwid olakšava neke teže poslove pri izradi tekstualnih sučelja [12].

Programi s Curses tekstualnim sučeljem često sliče programima s grafičkim sučeljem koji posjeduju tekstualne kutije, razne forme za ispunjavanje, liste s navigacijskim trakama i gumbе. Takvi grafički elementi olakšavaju korištenje programa, naspram standardnih tekstualnih programa, a ujedno omogućavaju rad na čisto tekstualnim uređajima. Programi s Curses tekstualnim sučeljem također zahtijevaju manje resursa naspram grafičkih programa.

Urwid koristi događajnu petlju (engl. *Event loop*) koja olakšava rukovanje programskim ulazima i osvježavanje sučelja.

2.2.3 Event loop

Event loop je programski konstrukt koji čeka i otprema događaje ili poruke unutar programa. Event loop šalje zahtjev pružatelju događaja (koji uglavnom blokira sve dok neki događaj ne bude dostupan) te zatim poziva određenog rukovatelja. Event loop je jedan od načina asinkronog programiranja.



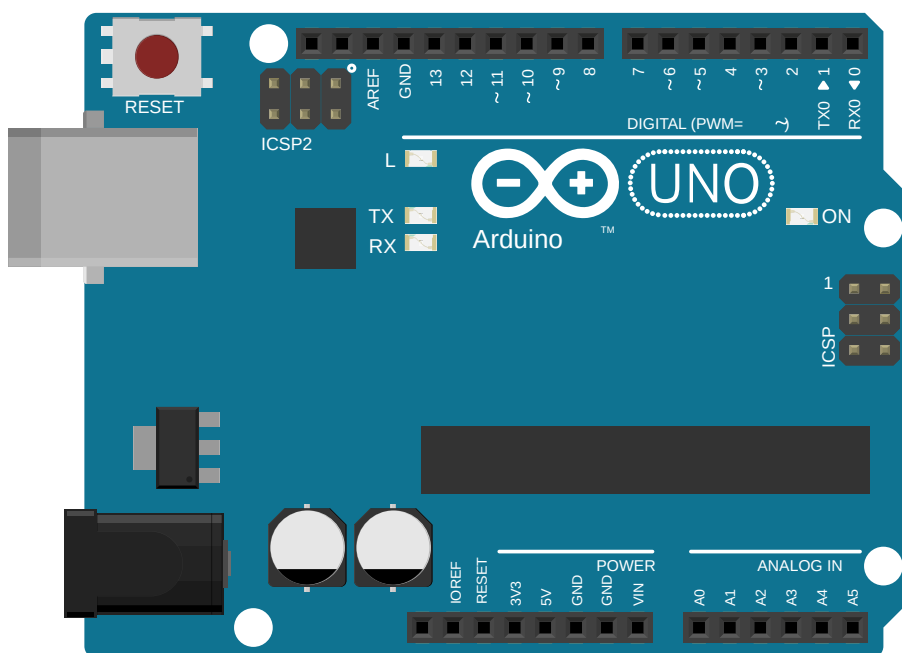
Slika 2.5.: Programski tok *Event loop*a

Na slici 2.5. je prikazan programski tok Event loop-a. Prvo se izračuna koliko će se čekati na događaj, pa se pomoću Unix sistemskog poziva *poll* čeka na događaj te se na kraju poziva funkcija koja je namijenjena za obradu određenog događaja.

2.3 Arduino

Arduino je mali uređaj za izradu računalnih sustava koji mogu mjeriti fizikalne veličine i upravljati fizičkim svijetom. Arduino je open-source računalni sustav koji se temelji na jednostavnoj mikroupravljačkoj pločici i razvojnom okruženju za pisanje softvera.

Arduino se može koristiti za razvoj interaktivnih fizičkih objekata, može primati ulazne fizikalne veličine od raznih senzora i upravljati sa svjetlima, motorima ili drugim fizičkim izlazima. Arduino projekti mogu biti samostalni ili mogu komunicirati sa softverom koji se nalazi na računalu. [13].



Slika 2.6.: Arduino UNO mikroupravljačka ploča.¹

Na slici 2.6. se vidi Arduino UNO mikroupravljačka ploča. Kao mikroupravljač koristi se ATmega328 [14]. Mikroupravljačka pločica posjeduje 14 digitalnih ulazno/izlaznih pinova, od kojih se 6 mogu koristiti kao pulsno širinski modulirani izlazi, 6 analognih ulaza, 16 MHz keramički oscilator, USB sučelje, ulaz za eksterno napajanje i reset tipku. Pločica se može napajati preko USB sučelja ili preko ulaza za eksterno napajanje. Komunikacija s računalom odvija se preko USB-serijskog sučelja. Za programski dio mikroupravljač sadrži 32 kB flash memorije koje se mogu programirati.

¹Izvorna slika, licencirana pod Creative Commons Attribution-ShareALike 3.0 preuzeta je i prilagođena od Fritzing projekta (https://raw.githubusercontent.com/fritzing/fritzing-parts/master/svg/core/breadboard/arduino_Uno_Rev3_breadboard.svg)

2.4 Firmata

Firmata je jednostavan protokol baziran na MIDI-ju i koji je namijenjen za komunikaciju računala s mikroupravljačem. Cilj mu je omogućiti direktno upravljanje što većim dijelom mikroupravljača od strane računala.

Za podatkovni format protokola odabrane su MIDI [15] poruke. Nije u potpunosti kompatibilan s MIDI standardom, budući da se koristi brža serijska konekcija i poruke se ne preklapaju u svim slučajevima. No za parsiranje poruka moguće je koristiti postojeće MIDI parsere.

Protokol se može implementirati kao firmware za bilo koji mikroupravljač ili kao softver za računalo. Firmata je prvi puta implementirana za mikroupravljače za Arduino obitelj mikroupravljača. Kao računalni softver, postoji mnogo Firmata implementacija za razne jezike, između ostalog i za Python [16] i Haskell [17]

Naredba	Numerička vrijednost
analog I/O message	0xE0
digital I/O message	0x90
report analog pin	0xC0
report digital port	0xD0
start sysex	0xF0
set pin mode(I/O)	0xF4
set digital pin value	0xF5
sysex end	0xF7
protocol version	0xF9
system reset	0xFF

Tablica 2.1.: Firmata naredbe

Na tablici 2.1. se vide neke od Firmata naredbi i njihove odgovarajuće numeričke vrijednosti. Da bi se naredio reset mikroupravljača, jednostavno se preko serijske veze pošalje vrijednost 0xFF te, nakon što mikroupravljač primi poruku, izvršit će reset rutinu.

2.5 JSON-RPC

JSON-RPC je jednostavan protokol za udaljene proceduralne pozive, ne sadržava stanje i nije definiran kao transportni protokol [18]. Koristi JSON [19] kao podatkovni format.

Pozivna poruka sastoji se od:

- `jsonrpc` - verzija protokola
- `method` - ime udaljene metode koja se poziva
- `params` - ulazni parametri za metodu
- `id` - identifikacijska oznaka poruke

Ulazni parametri moraju biti strukturiran podatak. Mogu biti u obliku niza ili JSON objekta, s imenovanim varijablama koje odgovaraju parametrima pozvane metode.

Odgovor se sastoji od:

- `jsonrpc` - verzija protokola ("2.0")
- `result` - rezultat koji vraća pozvana metoda
- `error` - greška ako ona postoji
- `id` - identifikacijska oznaka poruke

Ako dolazi do greške odgovor neće sadržavati rezultat, nego će rezultat će zamijeniti greška, a svi ostali dijelovi odgovora ostaju isti. Identifikacijska oznaka u odgovoru mora biti ista kao i u pozivu.

Primjer poziva:

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

Tu se vidi poziv koji koristi verziju 2.0 JSON-RPC protokola. Ona poziva metodu *subtract* kojoj predaje parametre u obliku niza (brojčane vrijednosti 42 i 23) te koristi identifikacijsku oznaku s brojem 1.

Primjer odgovora za prethodnu pozivnu poruku:

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

U odgovoru je vidljivo kako server koristi isti protokol kao i klijent, vraća rezultat pozvane metode te koristi istu identifikacijsku oznaku kao što je dobio u pozivnoj poruci od klijenta.

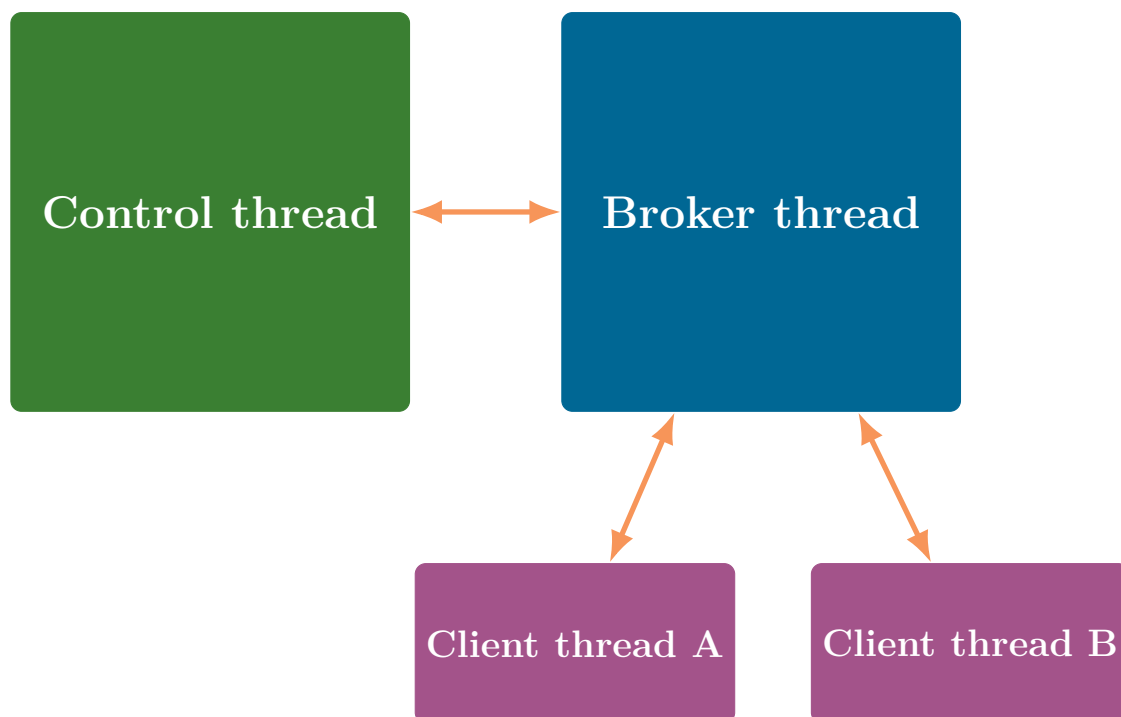
3. IMPLEMENTACIJA

U ovom poglavlju objašnjena je implementacija socket servera, odgovarajućeg klijenta i na kraju samog postrojenja. Detaljno je opisana arhitektura servera, klijenta te izrada makete postrojenja.

3.1 Server

Server je centralni dio rada. Na slici 2.1. u poglavlju 2 se vidi njegova uloga. Komunicira s postrojenjem s jedne strane i s klijentima s druge strane. Dizajn servera se oslanja na istodobnost i modularnost.

Glavni poslovi servera su jasno razdvojeni u različite niti. Server sadržava tri glavne niti od koje se jedna brine oko regulacije postrojenja, jedna prihvatanju konekcija klijenta i jedna služi kao posrednička nit između klijenata i regulacijske niti. Nit koja se brine oko prihvatanju konekcija od strane klijenta za svakog klijenta stvara novu nit te se prihvatanje novih konekcija može nastaviti nesmetano.



Slika 3.1.: Arhitektura servera

Na slici 3.1. je prikazana komunikacija između pojedinih niti unutar servera. Regula-
cijska nit, označena zelenom bojom, čita referentnu vrijednost iz dijeljene varijable, koju
postavlja posrednička nit, označena plavom bojom, a postavlja mjerenu procesnu veličinu u
drugu dijeljenu varijablu koje klijentske niti mogu direktno očitati. Zbog zahtjeva za ato-
marnosti, operaciju čitanja i pisanja dijeljenih varijabli nije moguće odobriti više od jednoj
niti zapisivanje u pojedinu dijeljenu varijablu. Zbog toga postoji posrednička nit koja od
raznih klijentskih niti preuzima referentnu vrijednost i atomarno prosljeđuje regulacijskoj
niti.

```
startDaemon :: Integer -> FilePath -> Bool -> IO ()
startDaemon port arduinoPort simulate = do
    sock <- socket AF_INET Stream 0

    bindSocket sock $ SockAddrInet (fromInteger port) INADDR_ANY
    listen sock 50

    pv <- newMVar 0
    referenceChan <- newChan
    let com = ProcCom pv referenceChan

    _ <- forkIO $ serverLoop sock com

    controllerBroker arduinoPort simulate com
```

Ispis koda 3.1: Početna funkcija servera

Na ispisu koda 3.1 prikazana je glavna funkcija, zvana `startDaemon`, koja pokreće server. Funkcija prolazi skoro kroz sve korake koji su prikazani na slici 2.2. u poglavlju 2.1.1. Prvo stvara socket, te ga povezuje s lokalnom adresom i portom. Nakon toga se socket pretvara u pasivni socket koji prihvća maksimalno 50 konekcija. Nakon stvaranja socketa i njegove pripreme, stvara se jedna dijeljena varijabla te jedan komunikacijski kanal.

Dijeljena varijabla sadržavati će mjerenu procesnu veličinu, to jest razinu vode u spremniku, dok će se komunikacijski kanal puniti sa željenim referentnim veličinama od strane klijenata. Kako pojedina referentna veličina stigne od strane klijenta, ona se stavlja u komunikacijski kanal te će ih kasnije posrednička nit vaditi iz komunikacijskog kanala i proslijediti regulacijskoj niti.

Na kraju se logika za socket server odvađa u novu nit sa `forkIO` funkcijom koja kao argument prima `serverLoop` funkciju i njene argumente. Glavna nit poziva `controllerBroker` funkciju koja će nastaviti s pripremom regulacijske niti.

```

serverLoop :: Socket -> ProcCom PVMType -> IO ()
serverLoop sock com = do
    (s, host) <- accept sock
    let hostinfo = show host
    noticeM rootLoggerName $ "Connected: " ++ hostinfo
    hdl <- socketToHandle s ReadWriteMode
    _ <- forkIO $ runConn hdl com hostinfo

    serverLoop sock com

```

Ispis koda 3.2: Glavna serveska petlja

Na ispisu koda 3.2 vidi se `serverLoop` funkcija. Ona skupa s funkcijom `accept` čeka na konekciju klijenta, a funkcija blokira sve dok se klijent ne spoji.

Za svakog klijenta koji se spoji na server ispisuje se informacija o klijentu, socket se pretvara u handle pomoću `socketToHandle` funkcije, te se stvara nova nit koja će izvršavati `runConn` funkciju. Ona je zadužena za komunikaciju s klijentom te će obraditi sve zahtjeve koje klijent pošalje. Nakon pokretanja nove niti, funkcija rekurzivno poziva samu sebe. Rekurzivni poziv služi kao beskonačna petlja.

Socket je pretvoren u handle u `ReadWrite` modu i zato s klijentima možemo komunicirati nizom standardnih IO funkcija:

- `hGetChar :: Handle -> IO Char`
- `hGetLine :: Handle -> IO String`
- `hPutChar :: Handle -> Char -> IO ()`
- `hPutStr :: Handle -> String -> IO ()`

Gore navedene funkcije ekvivalentne su standardnim C funkcijama: `getchar`, `gets`, `putchar` i `puts`.


```

runConn :: Handle -> ProcCom PVMType -> String -> IO ()
runConn hdl com hostinfo = do
    isEof <- hIsEOF hdl

    if isEof then do
        hClose hdl
        noticeM rootLoggerName $ "Disconnected: " ++ hostinfo
    else do
        contents <- hGetLine hdl
        debugM rootLoggerName $ "RPC request : " ++ contents

        response <- handleMessage com $ C.pack contents
        debugM rootLoggerName $ "RPC response: " ++ C.unpack response

        C.hPutStrLn hdl response
        runConn hdl com hostinfo

```

Ispis koda 3.3: Funkcija za klijentsku komunikaciju

Na ispisu koda 3.3 nalazi se `runConn` funkcija koja se brine o komunikaciji s klijentom. Komunikacija s klijentom vrši se pomoću `hGetLine` i `hPutStr` funkcija. Ukoliko funkcija pronađe *end of file* znak, ona zatvara handle, ispisuje informaciju o odpajanju i gasi nit. Ako pročita čitavu liniju, string sprema u `contents` varijablu.

Nakon što je string spremljen, ukoliko je to poželjno, ispisuje se njegov sadržaj te se poziva `handleMsg` funkcija, koja obrađuje poruku i, ukoliko je ona valjana, izvršava određenu radnju koja je zatražena u poruci te odgovor sprema u `response` varijablu. Odgovor se također ispisuje te se šalje spojenom klijentu. Na kraju funkcija rekurzivno poziva samu sebe kako bi bila spremna obraditi sljedeći RPC poziv.

```

controllerBroker :: FilePath -> Bool -> ProcCom PVType -> IO ()
controllerBroker arduinoPort simulate (ProcCom pvMVar refChan) = do
    refMVar <- newMVar 0

    _ <- forkIO $ forever $ do
        ref <- readChan refChan
        swapMVar refMVar ref

    if simulate then
        simulatorLoop refMVar pvMVar
    else do
        controlLoop arduinoPort refMVar pvMVar
        shutDownArduino arduinoPort

    noticeM rootLoggerName "Shutting daemon down."

```

Ispis koda 3.4: Posredniča funkcija

Ispis koda 3.4 sadržava `controllerBroker` funkciju. Ona je zadužena za pokretanje regulacijske logike te za komunikaciju između klijenata i regulatora.

Prvo se stvara nova dijeljena varijabla. Ona će sadržavati referentnu veličinu procesne varijable te će iz nje će regulator čitati referentnu veličinu. Stvara se nova nit, koja čita redom referentne vrijednosti koje su postavili klijenti u komunikacijski kanal te je postavlja u dijeljenu varijablu. Funkcija te niti je zapakirana u `forever` funkciju, koja nam služi kao beskonačna petlja.

Nakon što je posrednička i komunikacijska nit stvorena, funkcija je spremna pokrenuti regulator. Moguće je pokrenuti simulaciju postrojenja, ukoliko Arduino i postrojenje nisu spojeni s `simulatorLoop`.

Ako se ne pokrene simulacija postrojenja, pokreće se `controlLoop` funkcija. U nju je implementiran jednostavni PI regulator [20, str. 494]. Nakon što `controlLoop` funkcija završi, zbog gašenja servera ili greške pri komunikaciji s regulatorom, poziva se `shutDownArduino` funkcija koja šalje mikroupravljaču reset signal, kako bi on bio u poznatom i ugašenom stanju.

Jedine dvije funkcije koje ovise o mikroupravljaču koji se koristi su `controlLoop` i `shutDownArduino` te se socket server može prilagoditi bilo kojem mikroupravljaču ili procesu.

```
# ardaemon --help
Arduino control daemon 0.1

options [OPTIONS]

Common flags:
-p --port=INT           Listnening port
-a --arduinoport=ITEM   Path to the arduino serial port
-s --simulate           Simulate controller
-d --debugregulator     Enable debugging info for the regulator
-v --verbose            Output more
-? --help              Display help message
-V --version            Print version information
--numeric-version      Print just the version number
```

Ispis koda 3.5: Pomoćni ispis servera

Ispis koda 3.5 prikazuje moguće opcije za pokretanje servera. *port* opcija podešava TCP port na kojem će server slušati, *arduinoport* opcija prima serijski port na kojem se nalazi arduino mikroupravljač. Najzanimljivija opcija je *simulate* opcija. Ona naređuje serveru da pokrene simulaciju postrojenja te više ne zahtjeva arduino za rad.

Mogućnost simulacije postrojenja olakšala je izradu servera, usmjerila server ka modularnom dizajnu i ubrzala i olakšala izradu klijenta. Simulacijska funkcija izvršava sve bitne operacije normalne regulacijske funkcije (čitanje referentne vrijednosti i osvježavanje trenutne vrijednosti sustava) no to čini bez senzora i aktuatora, već je postrojenje matematički simulirano.

Slijedeće dvije opcije *debugregulator* i *verbose* uključuju ispis za debugiranje regulatora ili servera. Pomoću *debugregulator* regulacijska nit ispisuje informacije o stanju sustava i regulatora za svaku iteraciju regulacijske petlje, dok *verbose* opcija ispisuje informacije o primljenim RPC-pozivima te odgovarajućim odgovorima.

Ostale tri opcije *help*, *version* i *numeric-version* ispisuju samo informacije o serveru te ne pokreću server. *help* ispisuje informacije o serveru videne na ispisu koda 3.5, *version* ispisuje ime i trenutnu verziju servera dok *numeric-version* ispisuje samo trenutnu verziju.

3.2 Klijent

Klijent je najatraktivniji dio rada za korisnike. On korisniku preko grafičkog prikaza omogućuje uvid u stanje postrojenja u stvarnom vremenu i lagano upravljanje postrojenjem. Klijent je pisan u Pythonu i pomoću asinkronog programskog modela omogućen je grafički prikaz, koji je reaktivan te ne blokira izvođenje programa u niti jednom trenutku.

```
def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((args.host, args.port))

    tank = TankWidget()
    command_line = CommandLine(cmd_list, remote_cmds, sock)
    top = urwid.Frame(tank, None, command_line, 'footer')

    evl = urwid.AsyncioEventLoop(loop=asyncio.get_event_loop())
    loop = urwid.MainLoop(top, palette, event_loop=evl)

    loop.watch_file(sock.fileno(), read_cb)
    loop.set_alarm_in(0.1, periodic_tasks)

    loop.run()
```

Ispis koda 3.6: Početna funkcija klijenta

Ispis koda 3.6 sadržava main funkciju klijenta. Funkcija započinje stvaranjem socketa te se odmah spaja na server. Nakon spajanja slijedi inicijalizacija grafičkih elemenata. Grafičko sučelje sastoji se od dva grafička elementa: `TankWidget` i `CommandLine`.

`TankWidget` zadužen je za iscrtavanje shematskog prikaza postrojenja. Iscrtava shemu spremnika vode i njegovo trenutno stanje. Za iscrtavanje koristi Unicode-Brailleove znakove, koji su se pojavili u 3.0 verziji Unicode standarda [21].

`Commandline` stvara komandnu liniju koja je zadužena za primanje naredbi od strane korisnika te ih izvršava sam ukoliko su lokalne naredbe ili ih prosljeđuje serveru preko socketa. Komandna linija podržava standardne *Emacs* kratice i upotpunjavanje naredbi pomoću *tab* tipke.

Nakon inicijalizacije grafičkog sučelja stvara se event loop te se dodaje ranije stvoreni socket na listu praćenih događaja pomoću `read_cb()` funkcije, koja se poziva ukoliko podatci stignu na socket. Također se dodaje vremenski događaj koji će pozvati `periodic_tasks()` funkciju nakon 0.1 sekunde nakon što se event loop počne izvršavati. Ona je zadužena za

periodičko osvježavanje stanja postrojenja.

Na kraju se samo još pokreće event loop koji će se izvršavati sve dok korisnik, pomoću komandne linije, ne preda naredbu za gašenje programa.

```
def periodic_tasks(loop, data):
    request , _ = lvl_cmd('update-tank', [])
    try:
        sock.send(bytes(request, 'utf-8'))
    except BrokenPipeError as e:
        pass

    tank.update()

    loop.set_alarm_in(args.interval, periodic_tasks)
```

Ispis koda 3.7: Funkcija periodičkih poslova

Na ispisu koda 3.7 vidljiva je `periodic_tasks()` funkcija. Ona stvara RPC poziv pomoću *update-tank* metode te ga šalje serveru preko socketa.

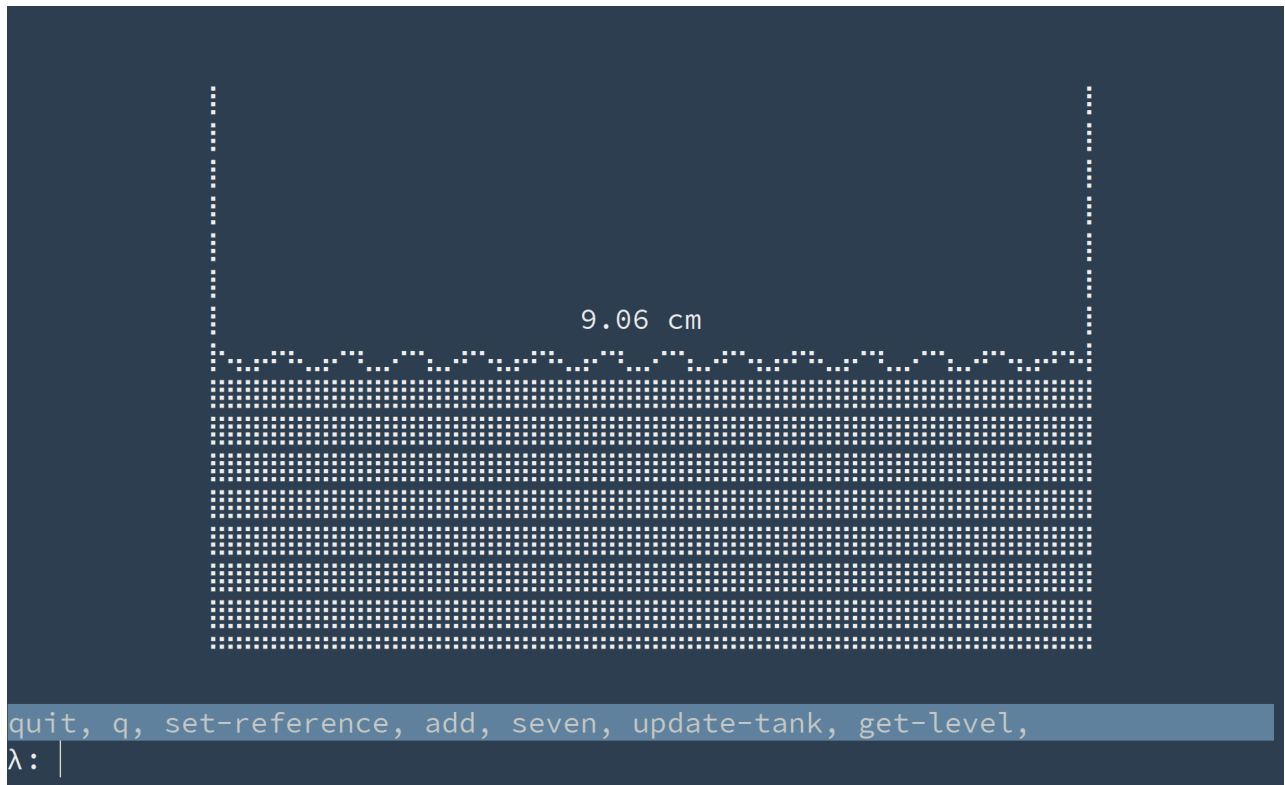
Nakon slanja RPC poziva poziva `update()` metodu *TankWidget* grafičkog elementa. Ukoliko je došlo do promjene u stanju postrojenja, ona osvježava iscrtavanje njegovog shematskog prikaza.

Na kraju se još dodaje novi vremenski događaj koji će ponovno pozvati `periodic_tasks()` funkciju. Interval tih periodičkih poslova korisnik može namjestiti na komandnoj liniji pri pokretanju klijenta.

```
# arclient --help
optional arguments:
  --help                show this help message and exit
  --port PORT           port to use (Default 4040)
  --host HOST           host to connect to (Default localhost)
  --interval INTERVAL  update interval
```

Ispis koda 3.8: Pomoćni ispis klijenta

Na ispisu koda 3.8 prikazane su moguće opcije pri pokretanju klijenta. *help* opcija ispisuje informacije o klijentu videne na ispisu koda 3.8, *port* opcija postavlja korišteni TCP port pri spajanju na server, *host* opcija postavlja adresu koja će se koristiti pri spajanju i *interval* opcija postavlja interval za osvježavanje stanja sustava i grafičkog sučelja.



Slika 3.2.: Prikaz klijenta

Na slici 3.2. je prikazano grafičko sučelje klijenta. Gornji dio grafičkog sučelja prikazuje shematski prikaz spremnika vode i razinu vode u stvarnom vremenu. Iznad razine vode ispisana je zadnje izmjerena razina vode.

Ispod shematskog prikaza spremnika vode nalazi se informacijski dio komandne linije, to jest statusna linija gdje su trenutno ispisane sve podržane naredbe:

- quit
- q
- set-reference
- add
- seven
- update-tank
- get-level

Naredbe su odvojene u lokalne i udaljene naredbe. Lokalne naredbe su: *quit* i *q*. Naredba *quit* služi za gašenje klijenta, a naredba *q* samo je alias na naredbu *quit*.

Udaljene naredbe su: *set-reference*, *add*, *seven*, *update-tank*, *get-level*. Sve udaljene naredbe generiraju JSON-RPC poziv te ga šalju serveru i registriraju funkcije koje će obraditi odgovor kad on stigne.

Naredba *set-reference* je najvažnija naredba klijenta. Ona nam omogućuje promjenu referentne vrijednosti u stvarnom vremenu. Kao argument prima razinu vode u boci te ga spremi u JSON-RPC poziv. Ukoliko dolazi do greške, greška se ispisuje u statusnoj liniji.

Naredba *add* je testna naredba koja prima dva brojana argumenta te ih šalje serveru u JSON-RPC pozivu. Server će zbrojiti primljene brojčane argumente i rezultat zbrajanja poslati natrag u JSON-RPC odgovoru. Nakon što odgovor stigne, njegov rezultat bit će prikazan u statusnoj liniji.

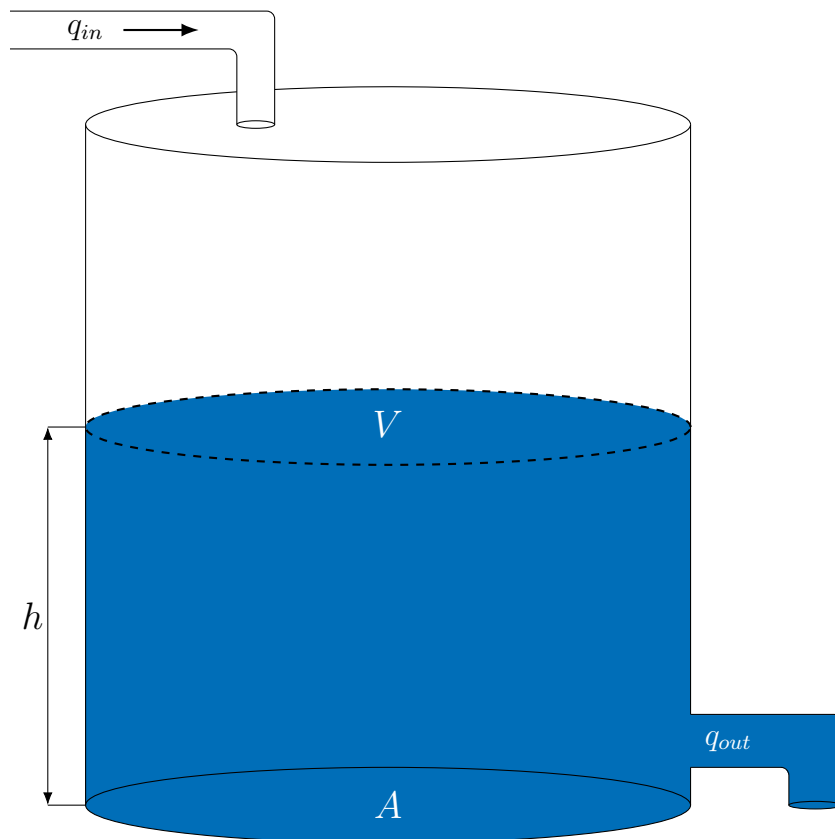
Naredba *seven* također je testna naredba. Ona serveru šalje, unutar JSON-RPC poziva, broj sedam. Server u JSON-RPC odgovoru šalje istu brojčanu vrijednost natrag. Naredba služi kao jednostavni *echo* test. Rezultat odgovora prikazuje se u statusnoj liniji.

update-tank naredba je naredba koja se izvršava automatski kao dio periodičkih zadataka koje event loop obavlja. Naredba dohvaća trenutnu razinu postrojenja te ga prosljeđuje **TankWidget** grafičkom elementu koji osvježava iscrtavanje shematskog prikaza, ukoliko je došlo do promjene. *get-level* naredba šalje isti JSON-RPC poziv kao i *update-tank* naredba, no umjesto da trenutnu razinu proslijedi grafičkom elementu za prikaz, ona razinu jednostavno prikaže u statusnoj liniji.

Ispod statusne linije nalazi se sama komandna linija u koju je moguće prije navedene naredbe upisati.

3.3 Postrojenje

Postrojenje je napravljeno u obliku makete spremnika u kojem se održava razina vode. Ulazni tok u spremnik ostvaren je pomoću vodene pumpe čijom se brzinom može upravljati pomoću pulsno širinske modulacije. Izlazni tok spremnika ostvaren je pomoću otvora pri dnu spremnika te voda slobodnim padom teče iz spremnika. Razina u spremniku direktno se mjeri pomoću senzora.



Slika 3.3.: Shematski model postrojenja

Na slici 3.3. vidi se postrojenje. Iznad glavnog spremnika vidi se ulazna cijev s ulaznim tokom q_{in} , dok se u desnom dijelu glavnog spremnika nalazi otvor za izlazni tok q_{out} .

Razina vode u spremniku označena je s h , dok je volumen koji voda zauzima u spremniku označen s V . Površina poprečnog presjeka spremnika označena je s A .

3.3.1 Matematički model postrojenja

Volumen vode u spremniku ovisi o ulaznom toku i izlaznom toku. Volumen će ostati konstantan, ukoliko su ulazni i izlazni tok isti, dok će se volumen vode mijenjati ovisno o razlici ulaznog toka i izlaznog toka te stoga možemo reći:

$$\frac{dV}{dt} = q_{in} - q_{out} \quad (3-1)$$

Gdje je ulazni tok definiran kao:

$$q_{in} = k_p U \quad (3-2)$$

gdje je:

k_p - konstanta pumpe

U - napon pumpe

Izlazni tok je određen pomoću Torricellijevog zakona [22, str. 75]:

$$q_{out} = a\sqrt{2gh} \quad (3-3)$$

gdje je:

a - poprečni presjek izlazne cijevi

g - ubrzanje zemljine sile teže (9.80665 m/s²)

h - visinska razina vode u spremniku

Budući da je spremnik cilindričnog oblika, za volumen spremnika možemo uzeti jednakost $V = Ah$. Uvrštavanjem ulaznog i izlaznog toka te jednakosti za volumen u jednadžbu 3-1 dobiva se:

$$A\frac{dh}{dt} = k_p U - a\sqrt{2gh} \quad (3-4)$$

Iz jednadžbe 3-4 je vidljivo da ona sadržava nelinearni element u obliku $a\sqrt{2gh}$. Kako bi uspješno mogli analizirati sustav i primijeniti Laplaceovu transformaciju, jednadžba koja opisuje sustav mora biti linearizirana [20, str. 88-97]:

$$\frac{d\delta h}{dt} = \frac{k_p}{A}\delta U - \frac{a\sqrt{2g}}{2A\sqrt{h_0}}\delta h \quad (3-5)$$

gdje je:

h_0 - radna točka oko koje je sustav lineariziran

Konstantne vrijednosti, uz δU i δh , mogu se jednostavnosti radi staviti u poseban izraz:

$$\frac{d\delta h}{dt} = k_1\delta U - k_2\delta h \quad (3-6)$$

Sada se može primijeniti Laplaceova transformacija [20, str. 35-44] pri čemu se dobiva:

$$sH(s) = k_1U(s) - k_2H(s) \quad (3-7)$$

Nakon grupiranja $H(s)$ elemenata na lijevoj strani jednadžbe, dobiva se:

$$H(s)(s + k_2) = k_1U(s) \quad (3-8)$$

Te konačno se dobiva prijenosna funkcija sustava [20, str. 45]:

$$G(s) = \frac{H(s)}{U(s)} = \frac{k_1}{s + k_2} \quad (3-9)$$

Iz jednadžbe 3-9 je vidljivo da jednadžba predstavlja sustav prvog reda [20, str. 166]. Osim što je sustav poprilično jednostavan, on je i samo stabilizirajući [23], tj. za svaku ulaznu veličinu sustav će se nakon određenog vremena stabilizirati.

Zbog jednostavnosti sustava možemo odabrati jednostavni PI regulator koji će omogućiti brz dolazak u stacionarno stanje te smanjiti regulacijsku grešku stacionarnog stanja.

3.3.2 Senzor i aktuator

Da bi regulator mogao obavljati svoj posao, mora imati uvid u stanje sustava te mora imati način na koji promijeniti stanje. Uvid u stanje regulatoru pruža senzor razine tekućine, a promjena stanja moguća je pomoću pumpe tekućine.

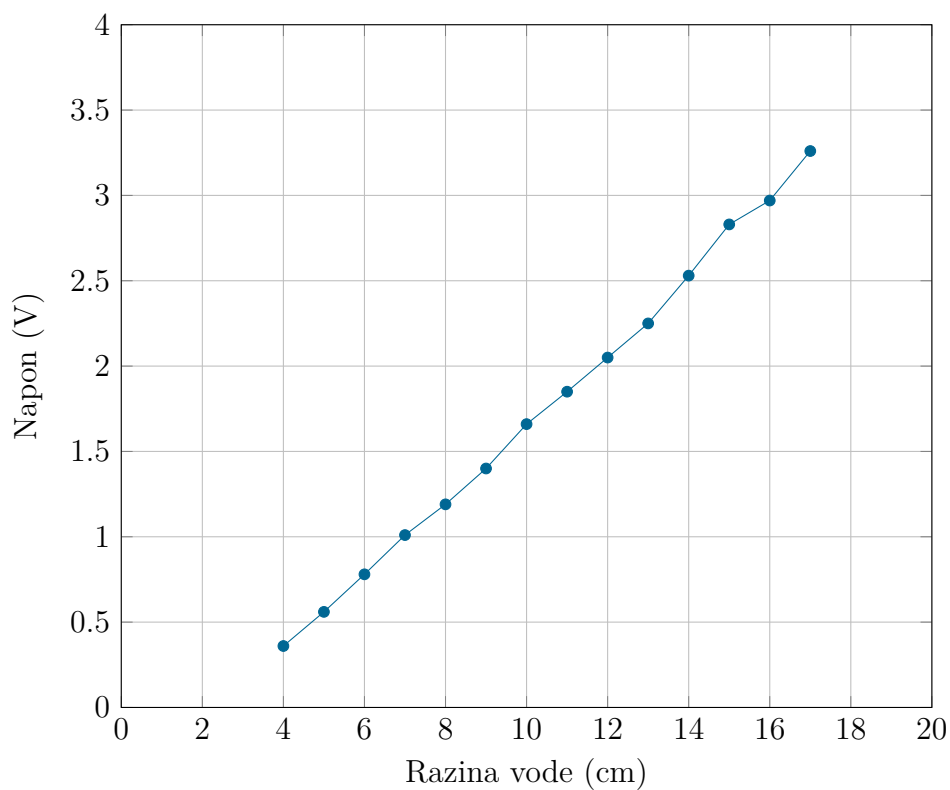
Za senzor je korišten *eTape* senzor razine tekućine [24]. Senzor je realiziran kao promjenjivi otpornik, čiji se otpor mijenja ovisno na kojem dijelu senzora hidrostatski tlak tekućine pritišće senzora. Izlazni otpor senzora obrnuto je proporcionalan razini tekućine.

Radi veće preciznosti mjerenja, izlazni otpor senzora pretvara se u napon raspona od 0 – 5 V. Pretvorba izlaznog otpora u napon ostvarena je pomoću 0 – 5 V *Linear Resistance to Voltage Module* istog proizvođača kao i senzora [25].

Napon (V)	0.36	0.56	0.78	1.01	1.19	1.40	1.66
Razina vode (cm)	4	5	6	7	8	9	10
Napon (V)	1.85	2.05	2.25	2.53	2.83	2.97	3.26
Razina vode (cm)	11	12	13	14	15	16	17

Tablica 3.1.: Kalibracijske vrijednosti senzora

U tablici 3.1. se nalaze mjerene naponske vrijednosti senzora uz određene razine vode. Senzor ima mrtvi pojas od 0 – 2.54 cm, u kojem se otpor te u konačnici napon ne mijenjaju.



Slika 3.4.: Graf odziva senzora

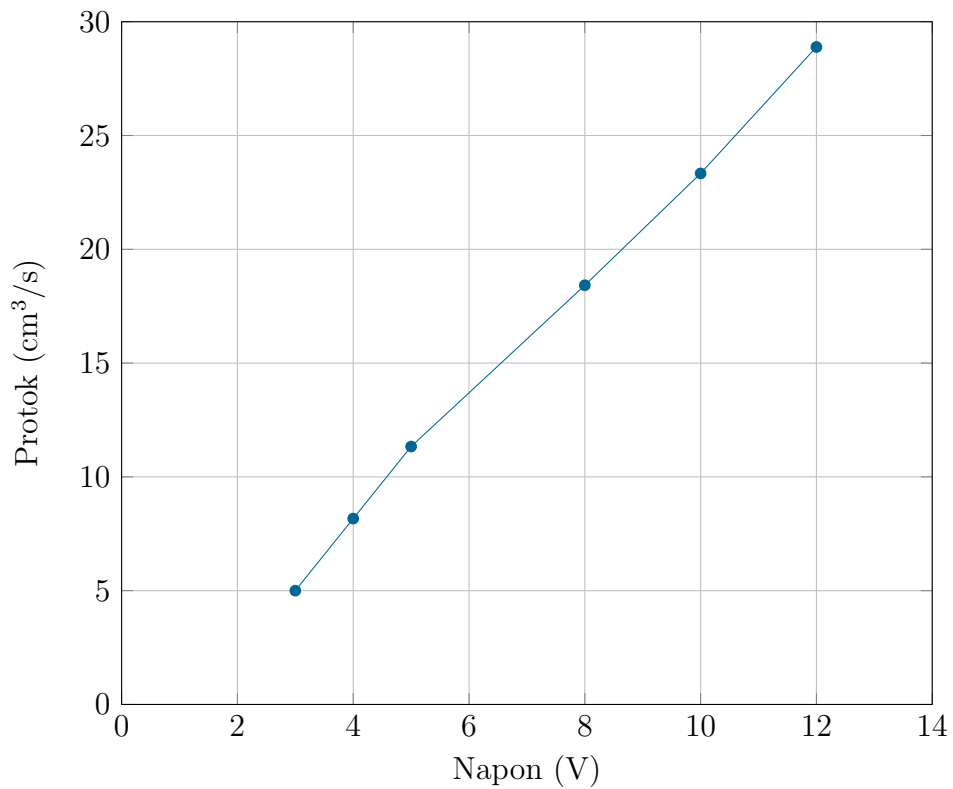
Na grafu 3.4. se vidi promjena napona ovisno o razini vode. Jasno je vidljiv linearni odziv senzora.

Pumpa koja je odabrana je mala pumpa za modelske svrhe maksimalnog protoka od 1.84 l/min. Protok pumpe se može regulirati naponski. Minimalni napon pumpe je 3 V, a maksimalni 12 V.

Napon (V)	12	10	8	5	4	3
Protok (cm ³ /s)	28.8889	23.3332	18.4193	11.3328	8.1663	5.0008

Tablica 3.2.: Kalibracijske vrijednosti pumpe

U tablici 3.1. se nalazi mjereni protok pumpe senzora uz određene naponske razine.

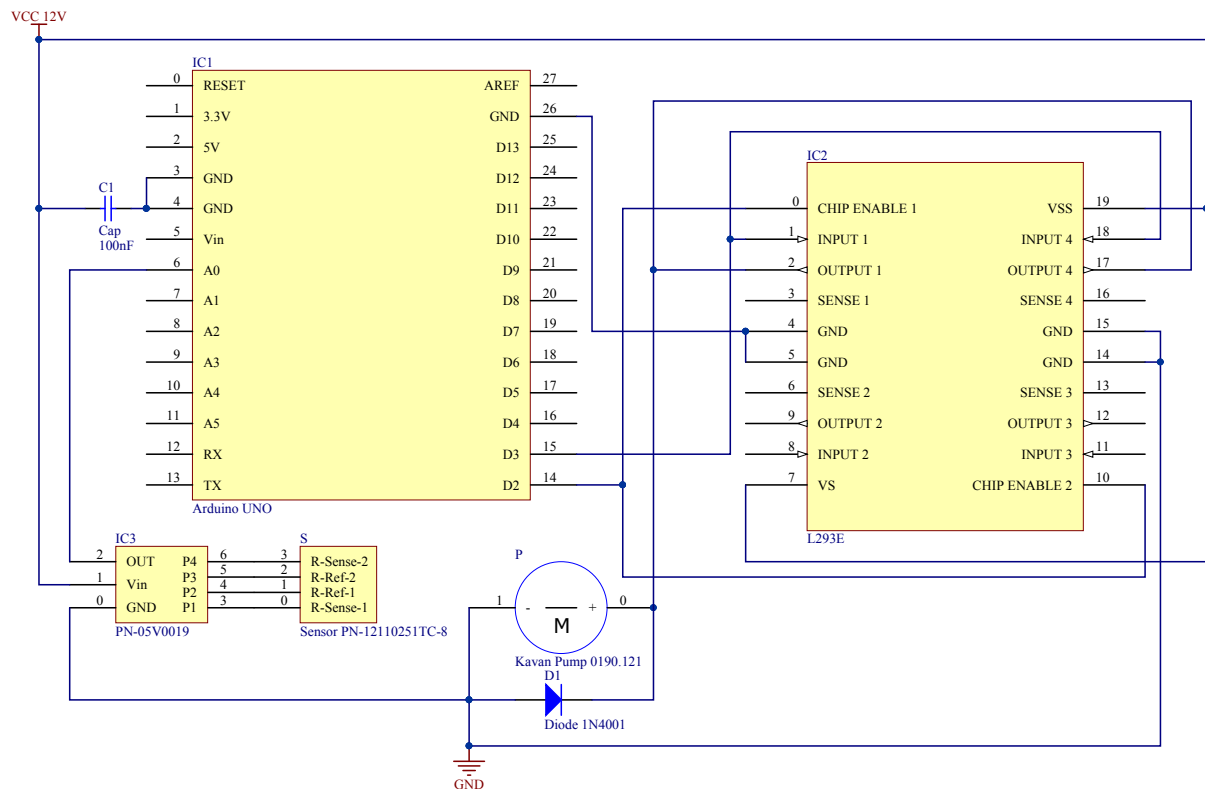


Slika 3.5.: Graf odziva pumpe

Na grafu 3.5. se vidi promjena protoka pumpe ovisno o naponskoj razini. Zbog ograničenog maksimalnog protoka pumpe, brzina odziva sustava ovisi o veličini spremnika. Za relativno brz odziv sustava odabrana je boca volumena od 1 l. Boca je cilindričnog oblika, radijus joj je 4 cm, a visina nešto ispod 20 cm. Zbog mrtvog pojasa senzora regulacija ispod 3 cm nije moguća.

3.3.3 Arduino shield

Operativni napon izlazno/ulaznih pinova Arduino mikroupravljačke pločice je u rasponu od 0 – 5 V te nije u stanju iskoristiti čitav operativni raspon pumpe. Da bi se mogao iskoristiti čitav operativni raspon pumpe koristi se *L293E* integrirani krug za pogon istosmjernih motora [26].



Slika 3.6.: Električna shema postrojenja

Slika 3.6. prikazuje potpunu električnu shemu postrojenja gdje je:

IC1 - Arduino mikroupravljačka pločica

IC2 - L293E IC za pogon istosmjernih motora

IC3 - Modul koji pretvara otpor senzora u napon

S - *eTape* senzor razine tekućine

P - 12 V pumpa

D1 - zaštitna dioda

C1 - stabilizacijski kondenzator

Schema također prikazuje spojeve između svih komponenti postrojenja. Analogni ulaz Arduina A0 spojen je na izlaz IC3 modula. Arduino na analognom ulazu A0 prima 0 – 5 V signal koji je proporcionalan razini vode. Na *L293E* integriranom krugu su pinovi za pogon prvog i četvrtog kanala kratko spojeni te se opterećenje tijekom rada raspodjeljuje na oba kanala.

Digitalni pin D2, koji je konfiguriran kao izlaz, spojen je na *Chip Enable* pin *L293E* integriranog kruga. Njime se upravlja aktivacija pogonskih kanala integriranog kruga. Napon na izlazu *L293E* integriranog kruga ovisi o naponu na ulazu. Ulaznim naponom upravljamo pomoću izlaznog pina D3 na Arduino.

D3 pin Arduina konfiguriran je kao pulsno širinski izlazni pin. Izlazni pin *L293E* integriranog kruga spojen je na pumpu. Paralelno s pumpom u spoju je dioda D1. Ona štiti strujni krug od induktivnog napona koji se pojavljuje pri gašenju pumpe.

3.3.4 Regulator

Regulator, koji je dio servera, komunicira s Arduinoom te pomoću Arduina upravlja postrojenjem. Osnovna mu je zadaća primiti željenu referentnu vrijednost te sustav dovesti u novo, stacionarno stanje i zadržati ga ondje.

Radi jednostavnosti i niskog reda sustava, kao regulator odabran je PI regulacijski algoritam.

```
forever $ do
  reference <- liftIO $ readMVar refMVar
  integral <- liftIO $ takeMVar integralMVar

  sensorValue <- analogRead sensor
  let fillHeight = sensorValueFunc sensorValue
  _ <- liftIO $ swapMVar pvMVar fillHeight
  let error = reference - fillHeight

  let proportional_term = kp * error
  let integral_term      = integral + ki * error / sampleTime
  liftIO $ putMVar integralMVar integral_term

  let output = clampAndScaleOutput $ proportional_term + integral_term

  analogWrite pwm_pin output
  delay sampleTime
```

Ispis koda 3.9: Regulacijska petlja

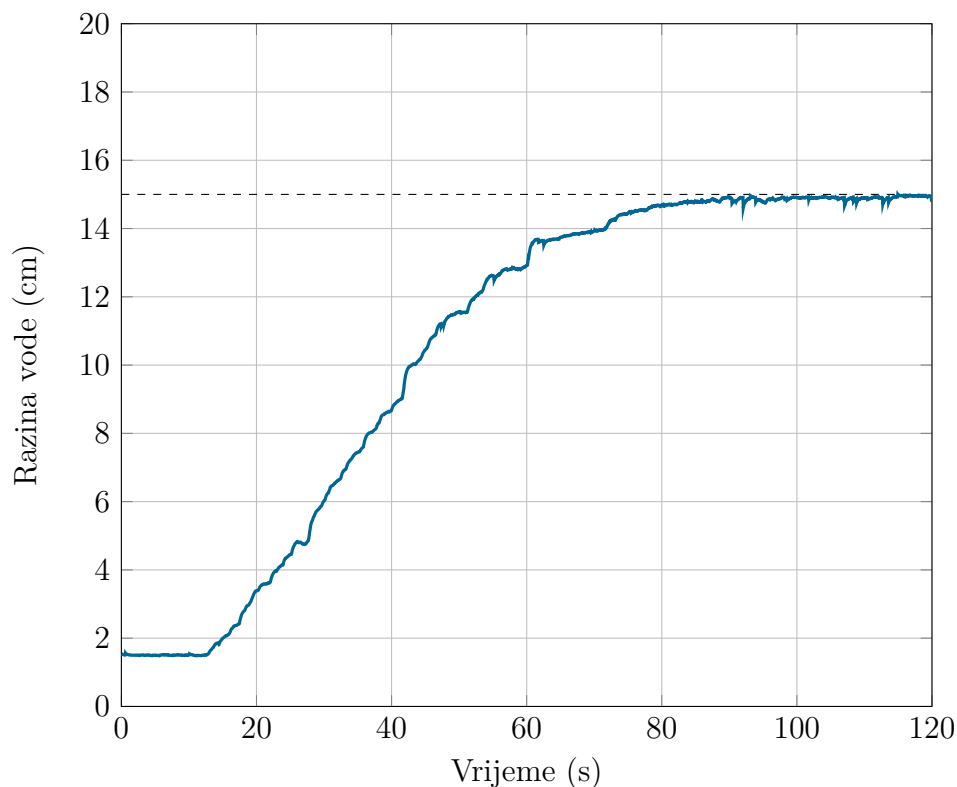
Na ispisu koda 3.9 prikazana je regulacijska funkcija koja se nalazi unutar servera. Prvo se uzima referenca i vrijednost integratora iz dijeljenih varijabli. `integralMVar` je interna varijabla regulatora gdje se sprema akumulirana vrijednost integratora, dok je `refMVar` dijeljena varijabla u koju posrednička funkcija stavlja referentnu vrijednost.

Odmah nakon čitanja dijeljenih varijabli, od Arduina se traži trenutna razina vode pomoću `analogRead` funkcije. Razina se pretvara iz naponske vrijednosti u visinu pomoću `sensorValueFunc` te se sprema u `fillHeight` varijablu i prosljeđuje ostatku servera, spremajući je u za to namijenjenu dijeljenu varijablu.

Nakon što je mjerenje izvršeno i trenutna referentna vrijednost preuzeta, pomoću istih se računa odstupanje od referentne vrijednosti te se sprema u `error` varijablu. Integralni dio regulatora sprema se u prije spomenutu varijablu `integralMVar`, koja će ju sačuvati za sljedeću iteraciju regulacijske petlje.

Sada se mogu proporcionalni i integralni članovi regulatora izračunati i zbrojiti. Funkcija `clampAndScaleOutput` ograničava izlaz regulatora na operativni raspon pumpe i brine se o tome da pumpa ne pokušava raditi na naponu ispod 3 V.

Preostaje samo poslati novu naponsku razinu na Arduino, a on će ju primijeniti na pulsno širinskom pinu i tako postaviti protok pumpe. Nakon toga petlja čeka određeno vrijeme te ponovno izvršava sve navedene korake.



Slika 3.7.: Graf odziva regulatora

Na slici 3.7. je prikazan izmjereni odziv regulatora uz $k_p = 4$ i $k_i = 1.5$.

4. ZAKLJUČAK

U radu je razvijen i predstavljen socket server za komunikaciju s postrojenjem. Server komunicira s Arduino mikroupravljačkom pločicom i upravlja postrojenjem koje je spojeno na pločicu. Demonstrirano je uspješno automatsko upravljanje razine vode u spremniku. Razvijen je i klijent koji u stvarnom vremenu daje uvid u stanje postrojenja te omogućuje udaljenu promjenu regulirane veličine. Tijekom izrade, glavni problemi bili su omogućavanje istovremenog obavljanja poslova regulacije i posluživanja podataka udaljenim klijentima, izrada makete postrojenja te povezivanje svih dijelova rada u cjelinu.

Predstavljeno rješenje koje koristi regulator unutar socket servera ima svoje prednosti i nedostatke. Nedostatci su vezani uz brzinu regulacije, zbog serijskog protokola koji je korišten za komunikaciju postrojenja i socket servera nije moguće regulirati procese koji imaju jako ograničene vremenske zahtjeve. Zahtjev za dodatnim računalom pored mikroupravljača za obavljanje posla regulacije također je jedan nedostatak. Prednosti su mogućnost rapidnog razvoja regulatora, mogućnost izmjene regulatora bez ponovnog programiranja mikroupravljača, mogućnost udaljenog upravljanja postrojenjem te fleksibilnost.

Široka dostupnost jeftinih mikroupravljača te malih računalnih platformi omogućuje korištenje predstavljenog rješenja u razne svrhe kućne automatizacije. Moguća je izrada mreže automatiziranih procesa kojima je omogućeno daljinsko upravljanje pomoću socket servera.

Rješenje također ima visoki potencijal za poboljšanja. Moguće je poboljšati parametre regulatora te ostvariti bolje upravljanje, prebaciti regulator na mikroupravljač te tako smanjiti latenciju. Na serveru se lako mogu implementirati razne vrste regulatora te bi se one mogle dinamički izmjenjivati. Komunikacija klijenta i servera izvršava se nekrptiran. Na serveru se također može implementirati zahtjev za autentikacijom prije nego se dozvoli uvid u stanje postrojenja.

LITERATURA

- [1] *Haskell Wiki Introduction*. URL: <https://wiki.haskell.org/Introduction>.
- [2] Paul Hudak i dr. „A History of Haskell: Being Lazy with Class”. *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, 2007, str. 1–55. ISBN: 978-1-59593-766-7. DOI: [10.1145/1238844.1238856](https://doi.org/10.1145/1238844.1238856). URL: <http://doi.acm.org/10.1145/1238844.1238856>.
- [3] *Network.Socket Documentation*. URL: <https://hackage.haskell.org/package/network-2.6.0.2/docs/Network-Socket.html>.
- [4] Andrew M. Rudoff W. Richard Stevens Bill Fenner. *Unix Network Programming Volume 1, Third Edition: The Sockets Networking API*. Addison-Wesley Professional, 2003. ISBN: 0131411551.
- [5] Simon Peyton Jones, Andrew Gordon i Sigbjorn Finne. „Concurrent Haskell”. *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '96. St. Petersburg Beach, Florida, USA: ACM, 1996, str. 295–308. ISBN: 0-89791-769-3. DOI: [10.1145/237721.237794](https://doi.org/10.1145/237721.237794). URL: <http://doi.acm.org/10.1145/237721.237794>.
- [6] *Control.Concurrent Documentation*. URL: <https://hackage.haskell.org/package/base-4.7.0.2/docs/Control-Concurrent.html>.
- [7] *Control.Concurrent.MVar Documentation*. URL: <https://hackage.haskell.org/package/base-4.7.0.2/docs/Control-Concurrent-MVar.html>.
- [8] *Control.Concurrent.Chan Documentation*. URL: <https://hackage.haskell.org/package/base-4.7.0.2/docs/Control-Concurrent-Chan.html>.
- [9] Guido Van Rossum i Fred L. Jr. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011. ISBN: 1906966141.
- [10] *Python Documentation - Low-level networking interface*. URL: <https://docs.python.org/3/library/socket.html>.
- [11] *NCURSES FAQ*. URL: <http://invisible-island.net/ncurses/ncurses.faq.html>.
- [12] *Urwid Manual*. URL: <http://urwid.org/manual/overview.html>.
- [13] *Arduino Introduction*. URL: <http://www.arduino.cc/en/guide/introduction>.

- [14] *ATMEL 8-Bit Microcontroller with 4/8/16/32KB Flash Datasheet*. URL: http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf.
- [15] MIDI Manufacturers Association Incorporated. *Table of MIDI Messages*. Teh. izv. URL: <http://www.midi.org/techspecs/midimessages.php>.
- [16] *pyFirmata*. URL: <https://github.com/tino/pyFirmata>.
- [17] *The hArduino package*. URL: <https://hackage.haskell.org/package/hArduino>.
- [18] JSON-RPC Working Group. *JSON-RPC 2.0 Specification*. Teh. izv. 2010. URL: <http://www.jsonrpc.org/specification>.
- [19] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. Teh. izv. 4627. 2006. URL: <http://www.ietf.org/rfc/rfc4627.txt>.
- [20] Norman S. Nise. *Control Systems Engineering*. Wiley, 2010. ISBN: 0470917695.
- [21] *The Unicode Standard 8.0*. URL: <http://www.unicode.org/charts/PDF/U2800.pdf>.
- [22] Michel Rieutord. *Fluid Dynamics: An Introduction*. Springer, 2015. ISBN: 3319093509.
- [23] *Practical Process Control: Proven Methods and Best Practices for Automatic PID Control*. URL: <http://www.controlguru.com/wp/p90.html>.
- [24] *Continuous Fluid Level Sensor PN-12110215TC-X Datasheet*. URL: http://www.milonetech.com/uploads/Standard_eTape_Datasheet.pdf.
- [25] *0 – 5 V Linear Resistance to Voltage Module*. URL: http://www.milonetech.com/uploads/0-5V_Module_Datasheet.pdf.
- [26] *L293E Push-Pull Four Channel Drivers Datasheet*. URL: <http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/CD00000058.pdf>.

SAŽETAK

Pojavom jeftinih mikroupravljača i mini računala mogućnosti povezivanja fizikalnog svijeta s digitalnim postaju sve veće. Svrha diplomskog rada je ispitivanje mogućnost korištenja jeftinih mikroupravljača za jednostavne poslove automatizacije kao i udaljenog upravljanja postrojenja preko *socket servera*.

Za mikroupravljač *Arduino Uno* preko USB-serijskog sučelja i preko *Firmata* protokola komunicira s računalom. Na njega je je vodena pumpa i senzor koji mjeri razinu vode u boci te sustav od dvije boce. Na računalu se nalazi *socket server* pisan u *Haskellu* koji upravlja mikroupravljačem. Testni klijent pisan u *Pythonu* spaja se na *server* te dohvaća mjernu veličinu i prikazuje trenutno stanje sustava.

Rad je podijeljen u dva dijela: teorijski dio koji detaljno opisuje proces razvoja socket servera te praktični dio koji se bavi implementacijom postrojenja, regulatora, *emphsocket* servera i popratnog klijenta.

Ključne riječi: Haskell, Arduino, Automatizacija, Python, Socket server, Firmata

ABSTRACT

The rise of cheap microcontrollers and mini computers have enabled new possibilities of connecting the physical world with the digital. The purpose of this paper is to explore the possibilities of cheap microcontrollers for simple automation as well as to enable remote control using a socket server.

The microcontroller Arduino Uno was chosen which communicates through the USB interface and the Firmata protocol with the computer. A water pump and a sensor, which measures the water level, were connected as well as two bottles. The computer is equipped with a socket server written in Haskell which controls the microcontroller. The test client written in Python is being connected to the server and pumps the requested amount of water and shows the current system status.

The paper is divided into two parts. The first part shall elaborate the theoretical pre-suppositions regarding the subject matter. The second shall address the implementation of the system, the regulators, the socket server and the accompanying client.

Keywords Haskell, Arduino, Automation, Python, Socket server, Firmata

ŽIVOTOPIS

Damir Jelić rođen je 8. ožujka 1989. godine u Osijeku. Osnovno obrazovanje stekao je u osnovnoj školi Šećerana u Šećerani. Nakon toga, pohađao je Prvu srednju školu u Belom Manastiru, smjer Tehničar računarstva, 2007. godine polaže maturu i iste godine upisuje Preddiplomski studij Računarstva na Elektrotehničkom fakultetu u Osijeku. 2010. godine završava Preddiplomski studij računarstva te upisuje Diplomski studij Procesnog računarstva. Dva puta je sudjelovao u programu Google Summer of Code: oba puta u sklopu PulseAudio organizacije, gdje je radio na dinamičkom podešavanju latencije i *resampling* poboljšanjima za PulseAudio softverski projekt.

PRILOZI

Prikaz spremnika sa senzorom



Prikaz Arduina i popratnog *shilda*

