



УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У
НОВОМ САДУ



Богдан Цветановски Пашалић

Једно решење бежичне IP комуникације између Raspberry Pi уређаја

ЗАВРШНИ РАД

Основне академске студије

Нови Сад, 2026.



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

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, РБР :	
Идентификациони број, ИБР :	
Тип документације, ТД :	Монографска документација
Тип записа, ТЗ :	Текстуални штампани материјал
Врста рада, ВР :	Завршни (Bachelor) рад
Аутор, АУ :	Богдан Цветановски Пашалић
Ментор, МН :	проф. др Илија Башичевић
Наслов рада, НР :	Једно решење бежичне IP комуникације између Raspberry Pi уређаја
Језик публикације, ЈП :	Српски / латиница
Језик извода, ЈИ :	Српски
Земља публикавања, ЗП :	Република Србија
Уже географско подручје, УГП :	Војводина
Година, ГО :	2026
Издавач, ИЗ :	Ауторски репринт
Место и адреса, МА :	Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, ФО : (поглавља/страна/ цитата/табела/слика/графика/прилога)	7/46/0/6/28/0/0
Научна област, НО :	Електротехника и рачунарство
Научна дисциплина, НД :	Рачунарска техника
Предметна одредница/Кључне речи, ПО :	Мрежни протоколи, Пајтон, Фрагментација пакета, Бежичне мреже, Мрежна архитектура, RF
УДК	
Чува се, ЧУ :	У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, ВН :	
Извод, ИЗ :	У овом раду су представљени дизајн и решење бежичне мреже између два рачунара користећи EFR32FG микроконтролере за бежичну комуникацију. Циљ рада је пројектовање и реализација слојевите мрежне архитектуре са алгоритмима за фрагментацију података и поновно слање пакета у случају неуспешног преноса.
Датум прихватања теме, ДП :	
Датум одбране, ДО :	
Чланови комисије, КО :	Председник: проф. др Мирослав Поповић
	Члан: проф. др Небојша Пјевалица
	Члан, ментор: проф. др Илија Башичевић
	Потпис ментора



KEY WORDS DOCUMENTATION

Accession number, ANO :			
Identification number, INO :			
Document type, DT :	Monographic publication		
Type of record, TR :	Textual printed material		
Contents code, CC :	Bachelor Thesis		
Author, AU :	Bogdan Cvetanovski Pašalić		
Mentor, MN :	Ilija Bašičević, PhD		
Title, TI :	A realization of wireless IP communication between Raspberry Pi devices		
Language of text, LT :	Serbian		
Language of abstract, LA :	Serbian		
Country of publication, CP :	Republic of Serbia		
Locality of publication, LP :	Vojvodina		
Publication year, PY :	2026		
Publisher, PB :	Author's reprint		
Publication place, PP :	Novi Sad, Dositeja Obradovica sq. 6		
Physical description, PD : (chapters/pages/ref./tables/pictures/graphs/appendixes)	7/46/0/6/28/0/0		
Scientific field, SF :	Electrical Engineering		
Scientific discipline, SD :	Computer Engineering, Engineering of Computer Based Systems		
Subject/Key words, S/KW :	Network protocols, Python, Packet fragmentation, Wireless networks, Network architecture, RF		
UC			
Holding data, HD :	The Library of Faculty of Technical Sciences, Novi Sad, Serbia		
Note, N :			
Abstract, AB :	This paper presents the design and implementation of a dedicated wireless network between two computers using EFR32FG microcontrollers for wireless communication. The goal of this paper is to design and develop a layered network stack with algorithms for packet fragmentation and packet retransmission in the case of transmission errors.		
Accepted by the Scientific Board on, ASB :			
Defended on, DE :			
Defended Board, DB :	President:	Miroslav Popović, PhD	Menthor's sign
	Member:	Nebojša Pjevalica, PhD	
	Member, Mentor:	Ilija Bašičević, PhD	



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

Број:

Датум:

ЗАДАТАК ЗА ЗАВРШНИ РАД

(Податке уноси предметни наставник - ментор)

Студијски програм:	Рачунарство и аутоматика		
Студент:	Богдан Цветановски Пашалић	Број индекса:	РА 26/2022
Степен и врста студија:	Основне академске студије		
Област:	Рачунарске мреже		
Ментор:	проф. др Илија Башичевић		
НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ЗАВРШНИ РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:			
- проблем – тема рада; - начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;			

НАСЛОВ ЗАВРШНОГ РАДА:

Једно решење бежичне IP комуникације између Raspberry Pi уређаја

ТЕКСТ ЗАДАТКА:

Циљ овог рада развој мрежне архитектуре и софтверског решења које омогућава мрежну комуникацију између два удаљена Raspberry Pi рачунара путем бежичне везе. Фокус је на дизајнирању слојевите мрежне архитектуре, реализацији алгоритама за фрагментацију података, управљање грешкама што укључује поновно слање порука. Рад обухвата мерење перформанси мреже у виду пропусности корисних података и брзине преноса.

Руководилац студијског програма:

Ментор рада:

Примерак за: ☐ - Студента; ☐ - Ментора



УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

21000 НОВИ САД, Трг Доситеја Обрадовића 6

ИЗЈАВА О НЕПОСТОЈАЊУ СУКОБА ИНТЕРЕСА

Изјављујем да нисам у сукобу интереса у односу ментор – кандидат и да нисам члан породице (супружник или ванбрачни партнер, родитељ или усвојитељ, дете или усвојеник), повезано лице (крвни сродник ментора/кандидата у правој линији, односно у побочној линији закључно са другим степеном сродства, као ни физичко лице које се према другим основама и околностима може оправдано сматрати интересно повезаним са ментором или кандидатом), односно да нисам зависан/на од ментора/кандидата, да не постоје околности које би могле да утичу на моју непристрасност, нити да стичем било какве користи или погодности за себе или друго лице било позитивним или негативним исходом, као и да немам приватни интерес који утиче, може да утиче или изгледа као да утиче на однос ментор-кандидат.

У Новом Саду, дана _____

Ментор

Кандидат

Захвалност

Захваљујем се проф. др Илији Башичевићу и техничким менторима Марату Каниарову и Милошу Пилиповићу на генералној сарадњи, подршци, помоћи, стрпљењу и разумевању током израде овог рада.

Такође бих да се завхалим својој породици, пријатељима и колегама који су непрекидно пружали подршку током студија.

САДРЖАЈ

1. Увод	1
1.1 Основна архитектура.....	2
2. Теоријске основе	4
2.1 UART комуникација.....	4
2.2 GPIO на Raspberry Pi системима.....	5
2.3 EFR32 системи на чипу.....	5
2.3.1 GPIO на EFR32 системима.....	5
2.3.2 Програмска подршка за EFR32.....	6
2.4 TUN/TAP руководиоци на Линукс оперативном систему.....	6
2.5 Једноканална бежична комуникација.....	7
2.5.1 CSMA/CA.....	7
3. Концепт решења	8
3.1 Мрежна архитектура система.....	8
3.1.1 Физички слој.....	9
3.1.2 EFR слој.....	9
3.1.2.1 Аутомат са коначним бројем стања.....	10
3.1.3 UART слој.....	12
3.1.3.1 Комуникација са EFR слојем.....	12
3.1.3.2 Контрола тока комуникације између два чвора.....	13
3.1.3.3 Анализа сродних техника.....	17
3.1.4 TUN слој.....	18
3.1.4.1 Структура фрагмената.....	18
3.1.4.2 Ретрансмисиона логика.....	19

4. Програмско решење	21
4.1 Програмско решење EFR слоја	21
4.1.1 Функција app_init	22
4.1.2 Модул packet_utils	23
4.1.2.1 Функција init_addr_filtering	23
4.1.2.2 Функција setup_payload	24
4.1.3 Модул app_process	24
4.1.3.1 Функција app_process_action	24
4.1.3.2 Функција sl_rail_util_on_event	24
4.1.3.3 Функција handle_packet_transmission	25
4.1.3.4 Функција start_receiving	25
4.1.3.5 Функција handle_received_packet	25
4.1.3.6 Функција handle_error_state	26
4.1.3.7 Функција try_read_cmd	26
4.2 Програмско решење UART и TUN слоја	26
4.2.1 Модул TRX	27
4.2.1.1 Методе send_req и send_ack	27
4.2.1.2 Метода send_single	28
4.2.1.3 Метода receive	28
4.2.1.4 Метода try_read	28
4.2.1.5 Метода init_mcu	29
4.2.1.6 Метода loop	29
4.2.1.7 Метода reset_state	29
4.2.2 Класа FragmentBuffer	29
4.2.2.1 Метода put_fragment	30
4.2.2.2 Метода is_valid	30
4.2.2.3 Класа FragmentGroup	30
4.2.2.4 Метода encapsulate	30
4.2.2.5 Метода do_fragmentation	30
4.2.3 Модул Network	31
4.2.3.1 Метода send_ack	31
4.2.3.2 Метода stop	31
4.2.3.3 Метода send_routine	31
4.2.3.4 Метода recv_routine	32

4.2.4 Модул main.....	33
5. Резултати	34
5.1 Тестирање уз помоћу ping команде.....	34
5.2 Тестирање функционалности користећи друге програме.....	40
5.3 Пропусност података.....	42
5.4 Брзина преноса података.....	43
6. Закључак	44
7. Литература	45

СПИСАК СЛИКА

Слика 1.1: Приказ Raspberry Pi рачунара повезани са EFR32 развојним плочама.....	2
Слика 1.2: Шематски приказ физичке архитектуре.....	3
Слика 2.1: Основно везивање пинова између два UART уређаја.....	4
Слика 3.1: Доњи слојеви мрежног модела система.....	8
Слика 3.2: Аутомат са коначним бројем стања за бежичну комуникацију.....	11
Слика 3.3: Аутомат са коначним бројем стања за UART комуникацију на EFR слоју...	11
Слика 3.4: Дијаграм слања поруке са UART слоја на EFR слој.....	12
Слика 3.5: Дијаграм слања поруке са EFR слоја на UART слој.....	13
Слика 3.6: Аутомат са коначним бројем стања координатор чвора на UART слоју.....	14
Слика 3.7: Аутомат са коначним бројем стања регуларног чвора на UART слоју.....	15
Слика 3.8: Дијаграм тока комуникације између координатора и регуларног чвора.....	16
Слика 3.9: Дијаграм тока комуникације између регуларног чвора и координатора.....	17
Слика 3.10: Дијаграм ретрансмисионе логике при успешној реконструкцији (лево) и неуспешној реконструкцији (десно).....	20
Слика 5.1: Приказ исправне конфигурације TUN уређаја.....	34
Слика 5.2: Резултати покретања пинг команде без додатних параметара.....	35
Слика 5.3: Логовани излази из координатора и регуларног чвора, без фрагментације...	35
Слика 5.4: Детаљан приказ ICMP пакета кроз Wireshark.....	36
Слика 5.5: Резултати ping команде са величином пакета од 528 бајтова.....	36
Слика 5.6: Логовани излази из мрежног стека, са фрагментацијом и потврдама.....	37
Слика 5.7: Резултати ping команде са величином пакета од 1500 бајтова.....	37

Слика 5.8: Логован излаз из мрежног стека при неуспешном слању података.....	38
Слика 5.9: Пример излаза из програма Wireshark, са ICMP пакетима од 1500 бајтова...	38
Слика 5.10: Приказ статистике на крају рада мрежног стека.....	39
Слика 5.11: Излаз из дијагностичке UART спреге EFR слоја.....	39
Слика 5.12: Успостављање везе између два чвора користећи SSH протокол.....	40
Слика 5.13: Пример учитавања странице www.trinitydesktop.org користећи реализован мрежни стек.....	41
Слика 5.14: Пример учитавања странице wiki.ubuntu.com користећи реализован мрежни стек.....	41
Слика 5.15: Граф пропусности корисних података за различите величине поруке у бајтовима, урачунајући фрагментацију података.....	42

СПИСАК ТАБЕЛА

Табела 3.1: Формат поруке на физичком слоју.....	9
Табела 3.2: Заглавље поруке на EFR слоју.....	10
Табела 3.3: Индикатори који се користе у оквиру UART преноса.....	13
Табела 3.4: Структура порука на UART слоју за контролу тока података.....	16
Табела 3.5: Заглавље фрагмената на TUN слоју.....	18
Табела 5.1: Резултати анализе брзине преноса података мрежног стека.....	43

СКРАЋЕНИЦЕ

API – *Application Programming Interface*, програмска спрега апликације
CSMA/CA – *Carrier Sense Multiple Access with Collision Avoidance*
CRC – *Cyclic Redundancy Check*
EFR – *EFR32FG28*, модел микроконтролера од компаније *Silicon Labs*
EUSART – *Extended Universal Synchronous/Asynchronous Receiver-Transmitter*
BFSK – *Binary Frequency Shift Keying*
FIFO – *First In, First Out*
GPIO – *General-purpose Input/Output*
HTTP – *Hypertext Transfer Protocol*
IoT – *Internet of Things*
IP – Интернет протокол
MTU – *Maximum Transmission Unit*, Максимална јединица преноса
OS – Оперативни систем
RAIL – *Radio Abstraction Interface Layer*
RPi – *Raspberry Pi*
SoC – *System on a Chip*, систем на чипу
SSH – *Secure Shell*
TAP – Виртуелни уређај за тунеловање Ethernet фрејмова
TCP – *Transmission Control Protocol*
TDMA – *Time Division Multiple Access*
TUN – Виртуелни уређај за тунеловање IP пакета
UART – *Universal Asynchronous Receiver-Transmitter*
USB – *Universal Serial Bus*

1. Увод

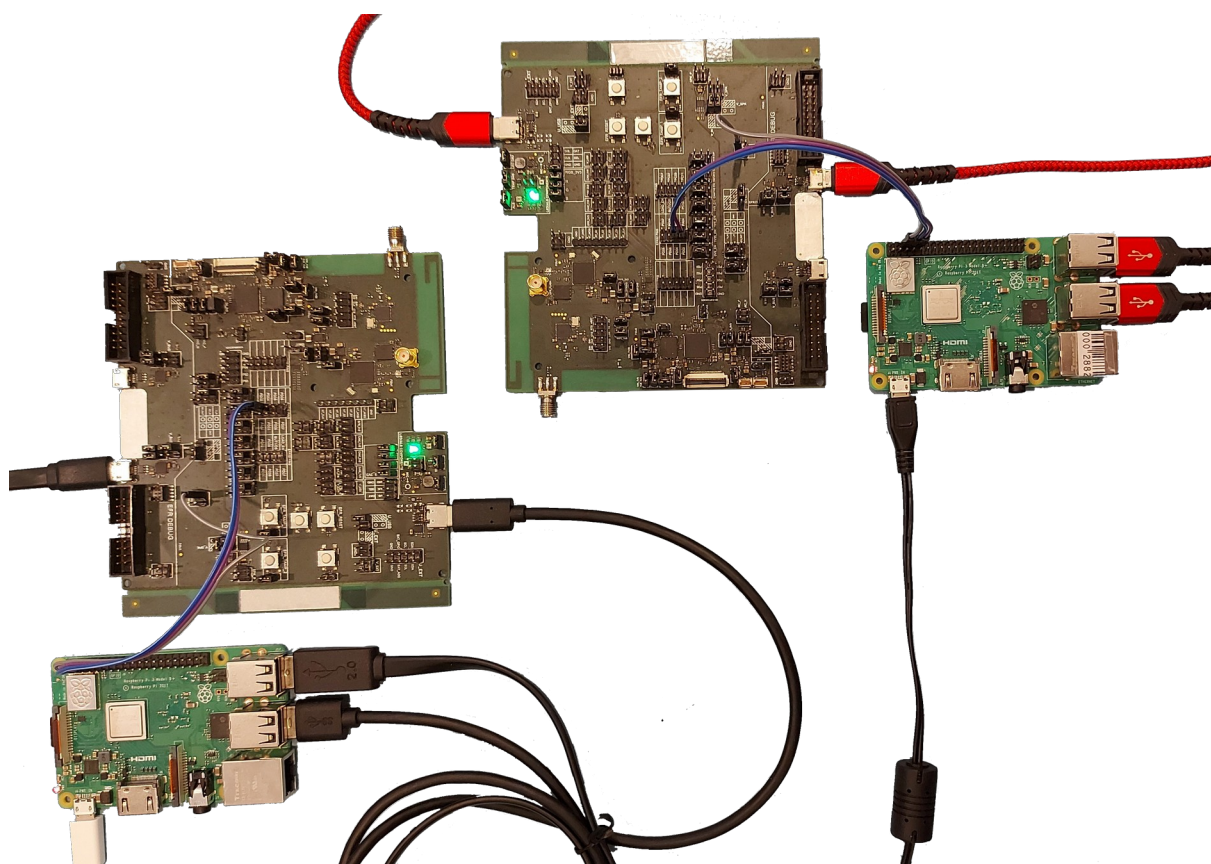
Честа потреба рачунарских мрежа јесте комуникација између уређаја који се значајно разликују по способностима и брзини преноса података. Као последица разноврсности уређаја који се могу појавити у рачунарској мрежи, такође ће се често разликовати максималан број бајтова који уређаји могу да пренесу и приме у једној трансакцији. У таквим случајевима се јавља потреба за фрагментацијом; поделом података на више делова како би уређај са већим MTU могао да пренесе податке уређају са мањим MTU.

У оваквим условима су нарочито осетљиви слабији и спорији уређаји који имају релативно мали MTU и који комуницирају преко бежичног канала који је склон сметњама. Због тога је потребан систем који може да врши фрагментацију пакета како би се подаци прилагодили овим ограничењима, као и накнадно спајање истих фрагмената са стране пријемника. Овакав систем такође треба бити отпоран на сметње при преносу фрагментованих података, што подразумева детекцију грешке при преносу, пријављивање успешног или неуспешног преноса целокупног пакета и понављање преноса у случају грешке.

Тема овог рада је представљање таквог система за два рачунара који ће комуницирати међусобно под претходно наведеним условима, као и процес основне реализације доњих слојева мрежног модела. Прво ће се представити физичка архитектура рачунарских система и теоријске основе које су неопходне за разумевање постављене архитектуре система као и концепта решења. Даље се представљају концепт, реализација, верификација и анализа програмског решења датог система. На крају се разматрају предности и мане целокупног система, као и могућности за унапређење и проширење представљеног решења.

1.1 Основна архитектура

У оквиру овог рада се презентује рачунарска мрежа која се састоји из два чвора који комуницирају преко бежичног канала. У овом случају то су два *Raspberry Pi 3B+* рачунара (скраћено RPi), засновани на ARM процесорској архитектури. Оба рачунара су опремљена са развојном плочом заснована на EFR32FG28 систему на чипу (ради концизности, даље у тексту ће се користити назив EFR32, или само EFR, да се односи на целу развојну плочу), на којој се налазе компоненте потребне за бежичну комуникацију.

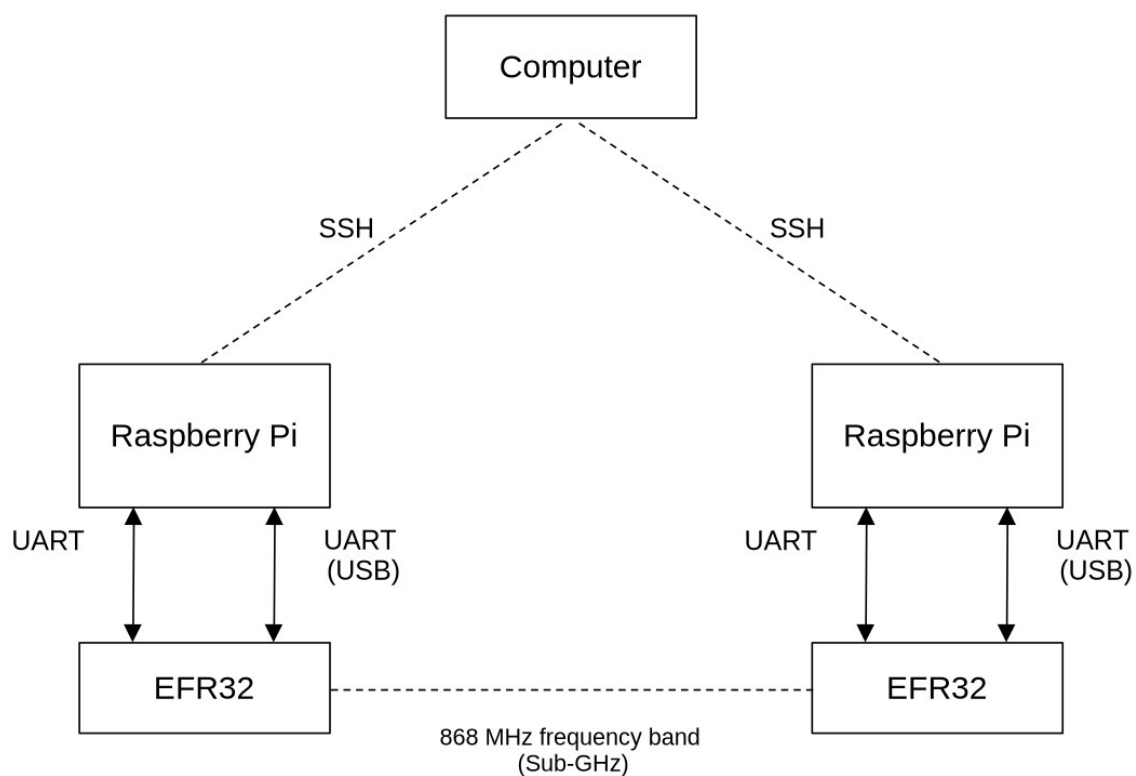


Слика 1.1: Приказ Raspberry Pi рачунара повезани са EFR32 развојним плочама

На EFR плочи се налази антена предвиђена за комуникацију на фреквенцијском каналу од 868.15 MHz (Sub-GHz), који ће представљати главни вид комуникације између два RPi рачунара. За комуникацију између EFR плоче и рачунара су предвиђене две UART комуникационе спреге, једна за несметан пренос мрежног саобраћаја, а друга за пренос дијагностичких информација од стране развојне плоче. Спрега за мрежни саобраћај користи директну везу уз помоћ GPIO пинова, док спрега за дијагностику се

спроводи преко USB пролаза који симулира UART серијску везу између развојне плоче и рачунара.

На крају, користи сте трећи рачунар који има успостављене везе између два RPi рачунара како би се мрежни саобраћај могао несметано пратити и бележити, користећи апликацију *Wireshark*, и за генерално управљање система, користећи SSH.

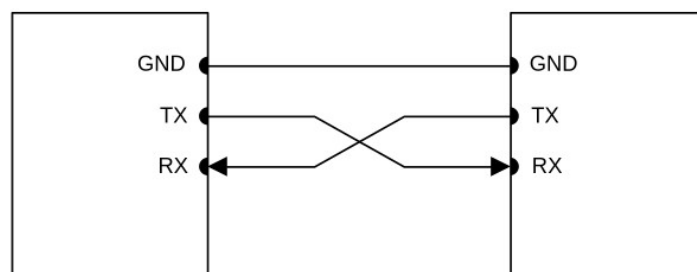


Слика 1.2: Шематски приказ физичке архитектуре

2. Теоријске основе

2.1 UART комуникација

UART^[1] (енгл. *Universal Asynchronous Receiver-Transmitter*) је стандардна серијска комуникациона спрега између два уређаја. У основном облику, оба уређаја имају RX и TX пинове који се унакрсно повезују, као и дељена маса која служи као референтна тачка за вредности сигнала који се преносе преко претходно наведених пинова.



Слика 2.1: Основно везивање пинова између два UART уређаја

Начин комуникације је подесив, и оба уређаја морају да имају иста подешавања како би пренос података био исправан, као што су формат пакета, напон сигнала (логички ниво), брзина преноса, смер података, итд. Како је комуникација асинхрона, не постоји заједнички тактни сигнал који диктира слање података, већ се користе почетни и крајњи бити при преносу података, као и усклађена брзина преноса (енгл. *Baud rate*). Брзина преноса се мери у симболима по секунди, што је у контексту UART комуникације еквивалентно битима по секунди. Већа брзина преноса подразумева строже временске захтеве при преносу података.

Физички, UART се најчешће реализује као наменски физички модул у оквиру рачунарског система, намењен да врши слање и пријем података независно од главног процесора, док се програмски најчешће представља као ток података, који се даље може користити у оквиру стандардог улаза и излаза.

2.2 GPIO на Raspberry Pi системима

GPIO на RPi и сличним рачунарским системима се састоји од пинова који се у основи могу користити као улазни пинови, који читавају неки улазни напон, или као излазни пинови, који могу поставити свој напон на логичку јединицу, или логичку нулу. Очекиван напон за логичку јединицу (ниво логике), на RPi је 3.3V. Поред општег карактера GPIO пинова, на RPi постоје пинови који могу преузети посебну функцију, подржана од стране физичке архитектуре система. Наиме, RPi има подршку за разне комуникационе протоколе, међу којих је и UART^[2]. Подршка за комуникационе протоколе кроз GPIO је реализована уз помоћ Линукс руковалаца који врше апстракцију над физичком архитектуром, пружајући стандардизовану спрегу који други програми могу користити.

2.3 EFR32 системи на чипу

EFR32^[3] системи на чипу су фамилија наменских рачунарских система засновани на ARM процесорској архитектури, коју развија компанија *Silicon Labs*. Користе се микропроцесори који припадају *ARM Cortex-M* групи, намењени за ниску енергетску потрошњу. EFR32 SoC додатно ставља фокус на IoT примене и тиме бежичну комуникацију, укључујући модуле који пружају способности као што су радио комуникација, регулација енергије и напајања, различите спреге за комуникацију са спољашњим физичким компонентама, као и комуникација са другим рачунарским системима, мерење времена, итд.

2.3.1 GPIO на EFR32 системима

GPIO пинови су груписани у портове, до 16 пинова за сваки порт. Сваки порт је именован пратећи форму Pxn , где x означава порт (A, B, C, ...), док n означава редни број пина (0, 1, ..., 15). Као и код RPi, EFR32 има подршку за разне комуникационе протоколе, укључујући и UART, разликујући се по томе што EFR32 има виши степен слободе избора пинова који се додељују за UART комуникацију, сходно ограничењима физичке архитектуре система.

Ниво логике за GPIO пинове није строго дефинисан, већ зависи од излазног напона на DCDC претварачу напона на микроконтролеру, који се користи за напајање читавог система. Стандардна конфигурација за EFR32 је употреба DCDC претварача како би се улазни напон у опсегу од 1.8V до 3.8V оборио на 1.8V на излазу^[4]. Као последица, GPIO пинови ће радити на логичком нивоу од 1.8V, што није у складу са логиком на страни RPi рачунара. Алтернативна конфигурација, која ће се користити у оквиру пројекта, је да се заобилази претварач напона активирањем унутрашњег прекидача, што омогућава да GPIO пинови функционишу на логичком нивоу од 3.3V. На овај начин омогућава се директно везивање између EFR32 развојне плоче и RPi рачунара.

2.3.2 Програмска подршка за EFR32

Препоручен начин развијања програмске подршке за EFR32 је кроз интегрисано развојно окружење *Simplicity Studio*, које омогућава једноставан приступ програмских алата, библиотека и примера потребни за развој програмских решења за EFR32 системе. Софтвер написан за уређај се програмира на микроконтролер при свакој промени изворног кода, користећи или USB комуникацију или *J-Link* комуникацију, у зависности од могућности развојне плоче на којој се налази EFR32 SoC.

Званичне библиотеке за EFR32 уређаје су написане у програмском језику C, као део *Simplicity SDK* пакета (претходно познат као *Gecko*). У оквиру пакета се налази основна програмска подршка за покретање система, стандардна библиотека, руководиоци за разне унутрашње модуле као и спреге за спољашње уређаје, итд.

За бежичну комуникацију се користи RAIL библиотека, која пружа апстраховану и подесиву спрегу за радио комуникацију, представљајући најнижи слој програмске подршке мрежног модела за EFR32 системе. Уз помоћ RAIL библиотеке је могуће реализовати мрежне протоколе засноване на стандардима, као и посебне комуникационе протоколе, независно од специфичности физичке архитектуре.

2.4 TUN/TAP руководиоци на Линукс оперативном систему

TUN и TAP^[5] представљају виртуелне мрежне спреге на Линукс оперативном систему који нису везане за физички мрежни уређај, већ се везују за програм који се извршава најчешће у оквиру корисничког простора. TUN и TAP спрега се користе најчешће у сврхе мрежног тунелирања (за виртуелне приватне мреже) и за повезивање виртуелних машина са остатком рачунарске мреже. TAP уређаји прикупљају етернет фрејмове, док TUN уређаји прикупљају IP пакете. Кориснички програм који користи TUN/TAP спрегу добија инстанцу уређаја при отварању датотеке `/dev/net/tun` која

припада TUN/TAP Линукс руковаоцу, избор између TUN или TAP спреге се спроводи преко одговарајућег *ioctl* позива, а затварањем датотеке се уклања та инстанца TUN/TAP уређаја. Док је активан уређај, читањем и писањем из датотеке је могуће вршити слање и пријем IP пакета или етернет фрејмова.

2.5 Једноканална бежична комуникација

Једноканална бежична комуникација подразумева употребу једне фреквенције као медијум комуникације. Како се фреквенцијски канал дели између два чвора и за слање и за пријем података у оба смера, следи да је облик комуникације полу-дуплекс (енгл. *half-duplex*)^[6]. Као последица, није могуће да оба чвора истовремено шаљу податке без појаве интерференције током преноса, тиме је потребно одредити начин комуникације где оба чвора могу несметано да шаљу и примају податке. Даље у тексту ће се представити конкретно решење овог проблема, као и упоређивање са сродним техникама које се користе у пракси као решење дате врсте проблема.

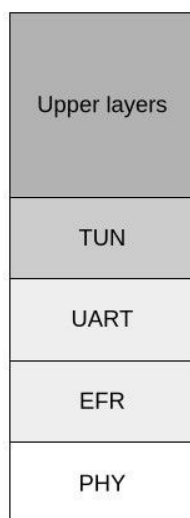
2.5.1 CSMA/CA

CSMA/CA^[7] (енгл. *Carrier Sense Multiple Access with Collision Avoidance*) је алгоритам за избегавање судара и интерференције при слању података у оквиру дељеног комуникационог канала. Пре него што се започне пренос података, чвор ослушкује комуникациони канал како би утврдио да ли се већ извршава пренос података, тј. канал је заузет. Ако је канал слободан, одмах се започиње пренос података. У супротном, уколико чвор утврди да је канал заузет, тада се не шаље порука, већ се чека одређен временски интервал, најчешће случајно одабран, пре него што се понови процес изнова. Чвор може да понови овај процес до одређеног броја пута уколико утврди да је комуникациони канал заузет, након чега се одустаје од слања и сматра се да је пакет изгубљен.

3. Концепт решења

3.1 Мрежна архитектура система

Као што је претходно наведено, топологија мреже је директна веза између два чвора (енгл. *Point-to-Point*). По узору на OSI модел, овај систем реализује физички слој, слој везе, и мрежни слој, док горњи слојеви (транспорт, сесија, презентација, апликација) остају неизмењени.



Слика 3.1: Доњи слојеви мрежног
модела система

Сваки слој има директну спрегу са горњим и доњим слојем респективно од тог слоја, тиме један слој може да шаље и прима податке само њему суседним слојевима. Ток података за предајника иде у смеру од горе ка доле, где се поруке спуштају наниже,

тако да сваки слој врши енкапсулацију података који пристиже директно од њему горњег слоја. За пријемника, подаци иду од доле ка горе, где се врши отпаковање заглавља за сваки слој.

3.1.1 Физички слој

Физички слој (PHY) представља бежични комуникациони канал, са дефинисаном стандардном^[8] фреквенцијом од 868.15 MHz, користећи BFSK модулатиону шему. Користи се само једна антена у оквиру развојне плоче због физичких ограничења пројекта, тиме се не користе технике разноврсности антена. У даљим радовима је могуће релаксирати ово ограничење, будући да EFR32 SoC може максимално две антене да користи и програмска подршка већ постоји у оквиру RAIL библиотеке.

Пакети који се преносе преко овог канала немају фиксну дужину, са доњом границом од једног бајта, а горњом границом од 255 бајтова. Формат пакета је дефинисан на следећи начин:

Преамбула	Синхронизациона реч	Заглавље	Садржај	CRC16
4B	4B	1B	≤ 255B	2B

Табела 3.1: Формат поруке на физичком слоју

Преамбула и синхронизациона реч^{[9][10]} се користе како би се најавио почетак пакета који треба да се прими, као и за синхронизацију између такта сигнала и такта пријемника. Синхронизациона реч има улогу граничника, раздвајајући преамбулу од остатка поруке који ће се заправо чувати након демодулације у меморијском баферу намењен за пријем података.

Заглавље се састоји од само дужине садржаја у бајтима као једино његово поље, Како је максимална дужина садржаја 255 бајтова, вредност дужине може стати у један бајт. Битна напомена је да RAIL библиотека не раздваја заглавље од главног садржаја током припреме и пријема пакета, већ се заглавље сматра као део садржаја. Као последица, ово поље ће се поново појавити на EFR слоју, али се односи на исти део поруке. Последња два бајта целокупне поруке садржи CRC код који се рачуна на основу заглавља и главног садржаја поруке, чија је улога провера интегритета садржаја поруке.

3.1.2 EFR слој

Овај слој, уз UART слој, заједно логички представљају слој везе у OSI моделу. Основне улоге EFR слоја су следеће: реализовање бежичне комуникације уз помоћу

аутомата са коначним бројем стања, обрада физичких прекида који представљају догађаје, реализација комуникације између EFR32 и RPi рачунара са стране микроконтролера користећи UART, читање пакета који пристижу кроз UART спрегу, слање примљених пакета кроз UART спрегу, слање статусних и осталих контролних сигнала кроз UART спрегу, енкапсулација ишчитаних пакета, адресно филтрирање ради спречавања сметњи у комуникацији, слање дијагностичких података кроз споредни UART канал.

Заглавље порука за овај слој је следећи:

Дужина пакета	Филтер адреса
1B	4B

Табела 3.2: Заглавље поруке на EFR слоју

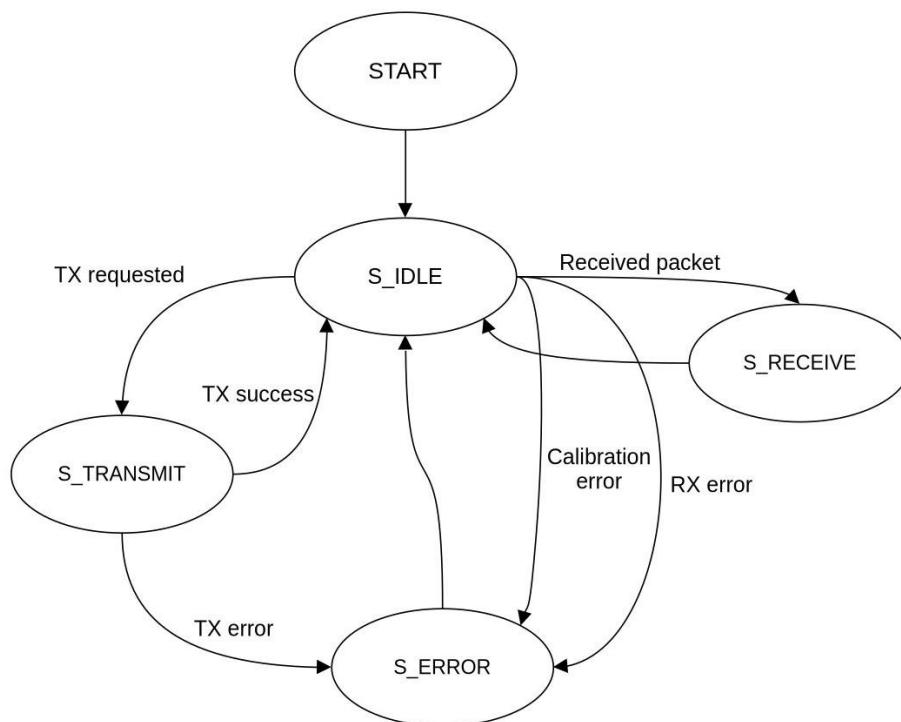
Филтер адреса је посебан низ бајтова који се користи за филтрирање порука на пријему. Уколико је низ бајтова почевши на одређеној позицији поруке једнака дефинисаном низу бајтова, тада је порука примљена успешно. Ако није, примљена порука се одбацује и пријављује догађај да је порука одбачена као резултат филтрирања. У оквиру овог решења, филтер адреса се дефинише као *0xDEADBEEF*, одмах после поља за дужину пакета.

3.1.2.1 Аутомат са коначним бројем стања

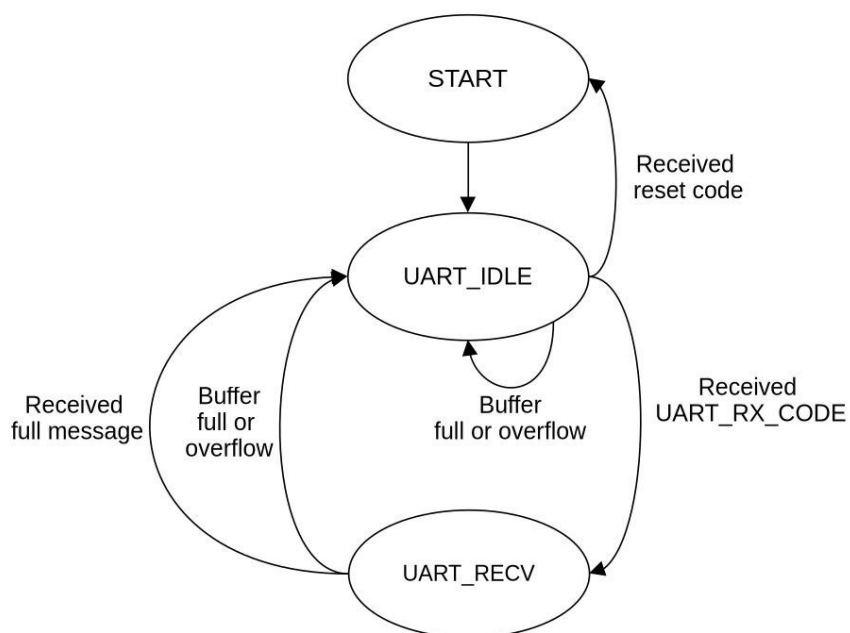
На овом слоју се реализују аутомати са коначним бројем стања за руковођење бежичне комуникације, како не би долазило до истовременог примања и слања порука, и комуникацију користећи UART.

Стање *START* представља почетак програма у оквиру микроконтролера, где се иницијализују физичке компоненте система. Стање *S_IDLE* има улогу да активира режим ослушкивања за нове поруке и читање улазних података са UART спреге. У овом стању је могуће да се деси физички прекид у ком може да се деси пријем поруке преко бежичног канала. Уколико се успешно деси пријем поруке у овом стању, прелази се у *S_RECEIVE* стање где се врши обрада примљене поруке. Након те обраде, прелази се назад у стање *S_IDLE*. Уколико се деси пријем читаве поруке преко UART спреге, тада се прелази у стање *S_TRANSMIT*, где се порука припрема и шаље преко бежичног канала користећи RAIL библиотеку. Користи се CSMA/CA техника током слања за избегавање судара и потенцијалних сметњи у каналу. Грешке се пријављују преко догађаја који се прослеђују током физичког прекида. Свака грешка пребацује аутомат у

стање S_ERROR , где се врши обрада грешке уколико је то потребно. На крају обраде, прелази се поново у стање S_IDLE .



Слика 3.2: Аутомат са коначним бројем стања за бежичну комуникацију



Слика 3.3: Аутомат са коначним бројем стања за UART комуникацију
на EFR слоју

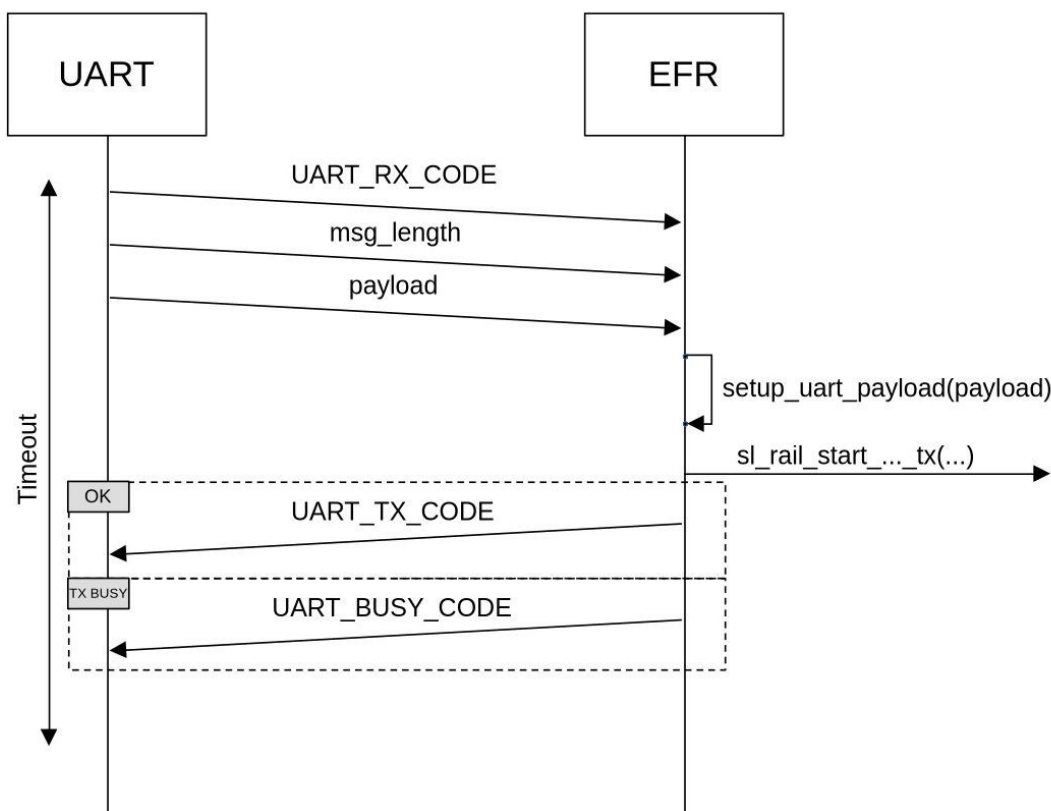
У стању *UART_IDLE* се очекују једно од два података: кôд за ресетовање EFR32 система, или кôд за пријем остатка поруке. Кôд за ресетовање развојне плоче је дефинисан као порука од четири бајта, са вредношћу *0x004E5E75*. У другом случају, прелази се у стање *UART_RECV* након пријема вредности *UART_RX_CODE*, после ког се очекује бајт који представља дужину остатка поруке, и након тога сам садржај поруке. Након што се ишчита целокупна порука, прелази се назад у стање *UART_IDLE*.

3.1.3 UART слој

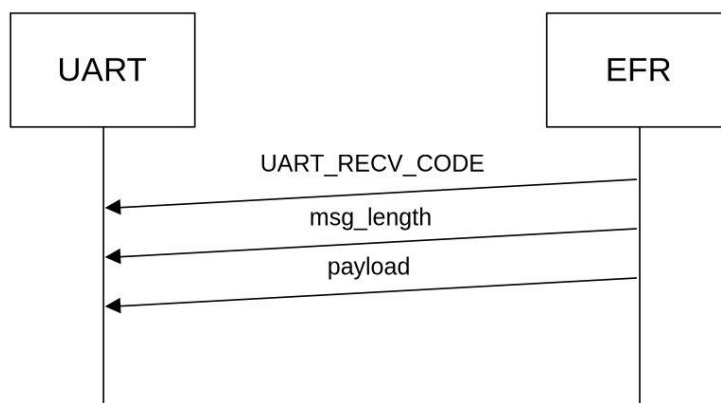
UART слој се реализује на страни RPi рачунара, имајући следеће улоге: комуникација између RPi и EFR32 система преко главног UART канала, примање порука са TUN слоја и регулисано слање истих кроз UART, контрола тока података између два чвора, и прикупљање података зарад статистике и анализе система.

3.1.3.1 Комуникација са EFR слојем

У оквиру задате UART комуникације није довољно само слати директан ток података, већ је потребно претходно обавестити доњи слој о контексту следећег тока бајтова. Такође је потребно обавестити UART слој око одређених догађаја и промена стања, где је довољно слати само један специфичан бајт који представља дати догађај.



Слика 3.4: Дијаграм слања поруке са UART слоја на EFR слој



Слика 3.5: Дијаграм слања поруке са EFR слоја на UART слој

Дефинисани су следећи индикатори који се користе за пренос података између доња два слоја:

Назив	Вредност	Функција
UART_RECV_CODE	0x11	Најављује пријем података на UART слоју
UART_BUSY_CODE	0x12	Неуспешно слање због заузетог канала
UART_RX_CODE	0x13	Индикатор почетка слања података на EFR
UART_TX_CODE	0x14	Потврда успешног слања пакета

Табела 3.3: Индикатори који се користе у оквиру UART преноса

У случају када се подаци преносе са UART слоја на EFR слој, прво се шаље UART_RX_CODE, а затим дужина остатка поруке која представља цео пакет, и на крају сам пакет. Успешно примљен пакет се обрађује на EFR слоју и шаље преко бежичног канала, након чега се враћа или потврда о успешном слању, или грешка због заузетог канала. Слични поступак се ради и за пренос података у супротном смеру, с тим што се не шаљу потврде назад ка EFR слоју.

3.1.3.2 Контрола тока комуникације између два чвора

У оквиру UART слоја се врши подела времена између два чвора како не би долазило до истовременог слања података кроз дељени комуникациони канал. Због тога се дефинишу два типа чвора, регуларан и координатор. Координатор има улогу да шаље контролне поруке, вршећи синхронизацију између себе и регуларног чвора. Регуларни чвор има улогу да ослушкује дате контролне поруке и да одговара на њих слањем порука потврде, и уколико је затражено, шаље своје поруке координатору. Регуларни чвор не шаље поруке уколико то није затражено од стране координатора. Синхронизација се одвија пре сваког преноса порука, у облику договора између

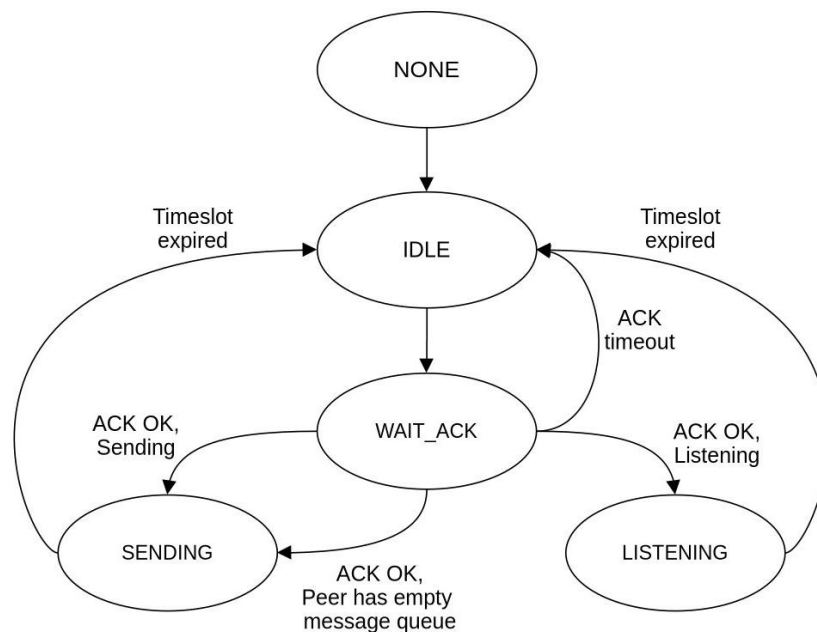
координатора и регуларног чвора, где се одређује смер комуникације и дужина временског слота током ког ће се одвијати слање порука.

Трајање временског слота није фиксне вредности, већ зависи од количине пакета који чвор жели да пошаље, и рачуна се на следећи начин:

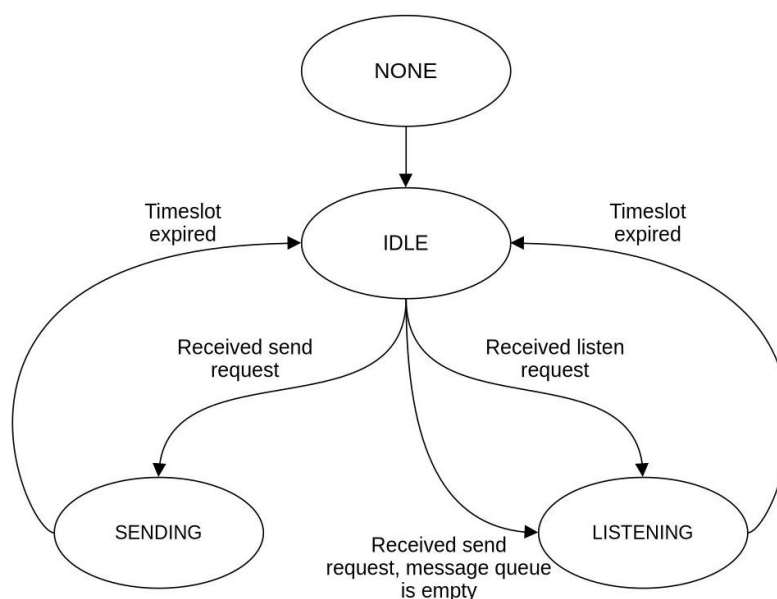
$$T_{send} = T_{single} \cdot \min(N, N_{max})$$

Где је T_{single} горња граница за време потребно да се пошаље један пакет, N је укупан број пакета на реду чекања за слање, а N_{max} максималан број пакета који се може послати у задатом временском интервалу. Избор параметара T_{single} и N_{max} је произвољан, и кроз мерења су одабране вредности 90ms и 20 респективно.

На овом слоју се реализује аутомат са коначним бројем стања, који се разликује у зависности од задате улоге чвора:



Слика 3.6: Аутомат са коначним бројем стања координатор чвора на UART слоју



Слика 3.7: Аутомат са коначним бројем стања регуларног чвора на UART слоју

Координатор чвор у *IDLE* стању шаље пакет за захтев који зависи од намењеног смера података и након тога прелази у стање *WAIT_ACK*. У случају да координатор не добије одговор на време, враћа се назад у стање *IDLE*. У случају да координатор добије потврду и уколико слање иде ка регуларном чвору, координатор прелази у стање *SENDING*, све до истека временског интервала, где се враћа у стање *IDLE*. У случају да је смер података ка координатору и регуларан чвор има поруке који чекају испоруку, координатор прелази у стање *LISTENING*, враћајући се у стање *IDLE* после истека времена. Уколико је смер података ка координатору, али регуларан чвор нема у том тренутку поруке за слање, како би се искористио тренутни временски интервал, координатор прелази у стање *SENDING* и шаље своје поруке, уколико их има, регуларном чвору до краја временског интервала.

Како се не врши строга временска синхронизација између два чвора током иницијалног договорања, реализација овог решења треба бити допустљива на примање регуларних порука у стањима *IDLE*, *LISTENING* и *WAIT_ACK*. Чвор ће бити у стању *LISTENING* током прослеђеног временског интервала од стране другог чвора, са додатним временским интервалом који представља горњу границу времена потребно да се пошаље пакет потврде од једног чвора ка другом.

У оквиру овог механизма се користе поруке које прате следећу структуру:

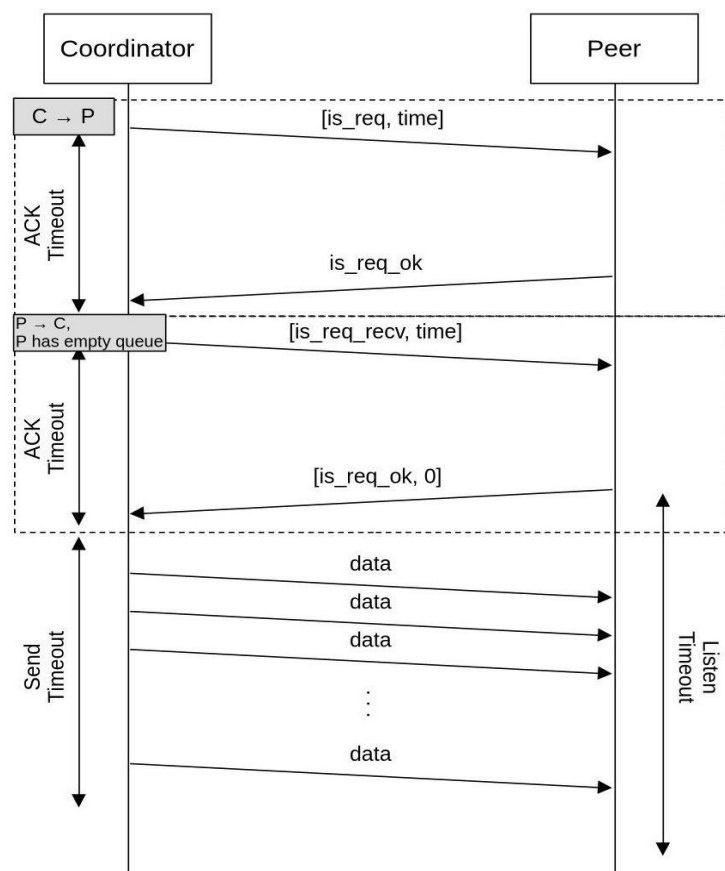
Тип пакета	Временски интервал
1B	2B

Табела 3.4: Структура порука на UART слоју за контролу тока података

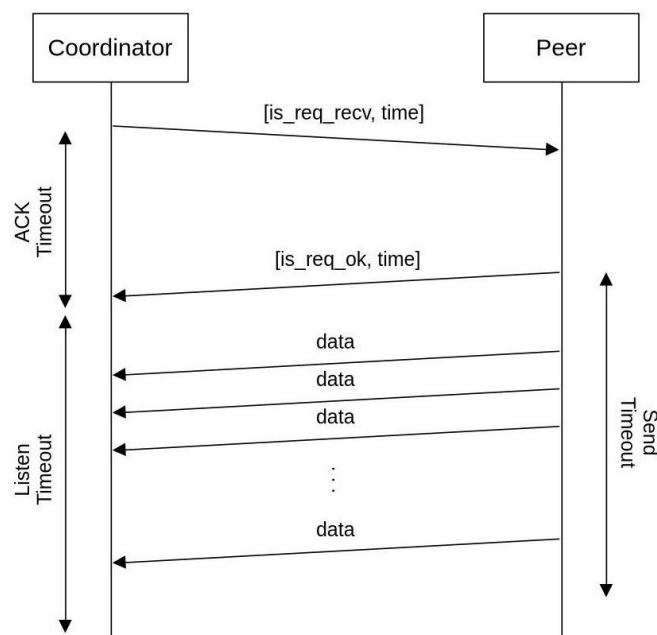
Тип пакета је дељено поље за пакете из UART и TUN слоја, за овај слој су од интереса следећи типови порука:

- *is_req* – Захтев за слање у смеру од координатора ка регуларном чвору
- *is_req_rcv* – Захтев за слање у смеру од регуларног чвора до координатора
- *is_req_ok* – Потврда захтева за слање

Пакети осталих типова се прослеђују нормално ка или од TUN слоја, што ће се детаљније обрадити у следећем поглављу. Временски интервал у милисекундама се прослеђује како би оба чвора знала трајање задатог временског слота.



Слика 3.8: Дијаграм тока комуникације између координатора и регуларног чвора



Слика 3.9: Дијаграм тока комуникације између регуларног чвора и координатора

3.1.3.3 Анализа сродних техника

Постоје више различитих техника које решавају или бар ублажавају проблем вишечланог приступа у бежичним комуникацијама. Технике које су по концепту инспирисале решење презентованог у овом раду су IEEE 802.11 RTS/CTS (енгл. *Request to send, clear to send*) и TDMA^[11] (енгл. *Time Division Multiple Access*).

RTS/CTS је техника која се опционо користи у склопу са CSMA/CA алгоритмом за избегавање судара у комуникацији између чворова у бежичној мрежи. Иако је фокус ове технике решавање проблема скривених чворова^[12], у комбинацији са CSMA/CA долази се до смањења судара у каналу. Главни концепт ове технике је да један чвор шаље захтев (RTS) другом чвору за слање података, након чега добија потврду за слање (CTS). Остали чворови у мрежи који су у довољној близини детектују ове поруке и престају са слањем порука током трајања преноса који се дефинише у оригиналној поруци за захтев.

За разлику од представљеног решења, RTS/CTS пакети нису намењени да шаљу већи број пакета, и користе се пакети потврде након слања главног пакета. Додатно, RTS/CTS пакети се могу користити у *ad-hoc* мрежама где не мора да постоји инхерентна разлика између чворова, тиме било који чвор може да их шаље, нити се користе за одређивање смера комуникације.

TDMA је техника вишечланог приступа комуникационог канала где се приступ комуникационом каналу дели на основу временских слотова, где само један чвор може да преноси податке у временским слотовима резервисани за њега. У зависности од реализације, чворовима се додељују фиксни временски интервали за слање, или је могуће динамички додељивати различите временске слотове на основу захтева сваког индивидуалног чвора. За разлику од осталих реализација (нпр. *Bluetooth*^[13]), у оквиру датог решења се не врши експлицитна временска синхронизација, већ координатор на почетку сваког временског слота шаље пакете за преговарање. Додатно, скала временских интервала је знатно већа у односу на већину реализација чији је циљ прављење ефекта дуплекс комуникације путем TDD (енгл. *Time Division Duplex*).

3.1.4 TUN слој

TUN слој логички представља мрежни слој у OSI моделу. Реализује се преко TUN виртуелне мрежне спреге како би се остварила веза са вишим мрежним слојевима, прикупљајући IP пакете који ће се обрадити и проследити даље на ниже слојеве. У склопу решења игнорисаће се IPv6 пакети, стављајући фокус на IPv4.

Главна улога TUN слоја је фрагментација прикупљених IP пакета, уколико су превелики да се пошаљу у целини кроз мрежу, као и реконструкција фрагмената на пријему. Додатна улога овог слоја је слање потврда пошаљиоцу након успешне реконструкције пакета на основу добијених фрагмената, или слање захтева да се фрагменти пошаљу поново, у случају грешке током преноса података.

3.1.4.1 Структура фрагмената

За IP пакете који су већи од задате MTU вредности мрежног система потребно је извршити фрагментацију. У супротном се IP пакети само директно прослеђују ка доњим слојевима, без икакве обраде. Разликују се три типа фрагмената; почетни фрагмент, регуларан фрагмент и последњи фрагмент. Заглавље ових пакета је следећи:

Тип пакета	Групни идентификатор	Величина/позиција	CRC32
1B	2B	2B	4B

Табела 3.5: Заглавље фрагмената на TUN слоју

Поље за тип пакета је бит маска која може да садржи више индикатора који описују конкретан тип поруке, односно сваки бит представља одређен тип поруке. За фрагментацију се користе следећи типови:

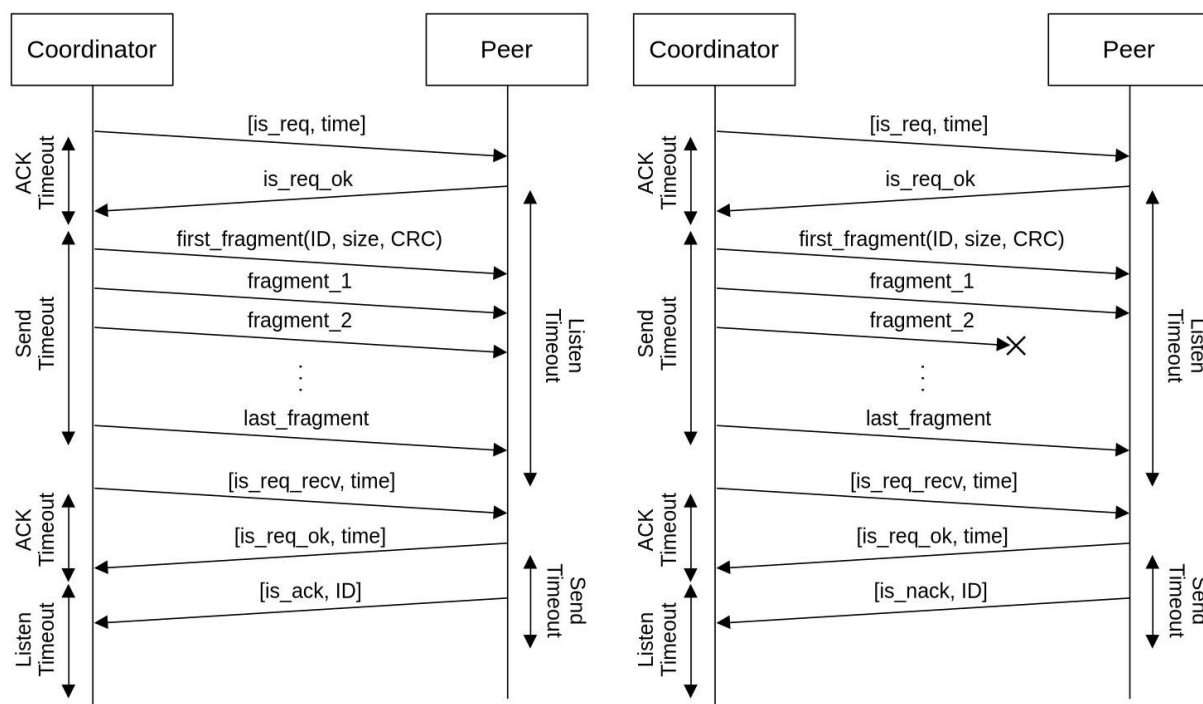
- *empty* – Поље има вредност 0, указује да пакет није никаквог посебног типа, и да не треба да се ради обрада над истим
- *is_fragment* – Указује да је пакет фрагмент, овај бит мора бити постављен и за прве и последње фрагменте
- *is_first_fragment* – Указује да је пакет први фрагмент
- *is_last_fragment* – Указује да је пакет последњи фрагмент

Сваком фрагменту се додељује број који представља групу фрагмената који у целини чине оригиналну поруку. Значење трећег поља зависи од типа фрагмента. Уколико је у питању први фрагмент, ово поље представља тоталну величину оригиналне поруке која је фрагментована. У супротном представља позицију у бајтовима где се садржај фрагмента налази у оригиналној поруци. На крају се налази поље за проверу интегритета поруке након реконструкције. Ово поље се једино појављује код првог фрагмента, и рачуна се на основу оригиналног садржаја IP пакета, користећи CRC32 алгоритам. Реконструкција пакета се дешава када пријемник прими последњи фрагмент у групи, након чега се шаље позитивна или негативна потврда у зависности од успеха у реконструкцији.

3.1.4.2 Ретрансмисиона логика

За потврду и поновно слање фрагмената се користе пакети који садрже поља за тип пакета и групни идентификатор. На првом пољу се поставља битски индикатор *is_ack*, а за негативне потврде се додатно поставља *is_nack* бит.

Друго поље представља групу фрагмената који су успешно послати у случају да је у питању потврда, или који требају поново да се пошаљу у случају грешке. Уколико дође до грешке током реконструкције или пријема фрагмената, сви фрагменти дате групе се одбацују са стране пријемника који шаље захтев за поновно слање фрагмената. Пакети ових типова, као и пакети који се поново шаљу као резултат грешке, имају виши приоритет у односу на остале поруке током слања.



Слика 3.10: Дијаграм ретрансмисионе логице при успешној реконструкцији (лево) и неуспешној реконструкцији (десно)

Пренос података са TUN слоја на нижи UART слој се ради преко два реда чекања. Први ред чекања се користи за поруке типа потврде, као и фрагменте који се поново шаљу као резултат негативне потврде. Други ред чекања се користи за све остале поруке, и има нижи приоритет у односу на претходни ред чекања, што се тиче редоследа у слању на UART слоју.

4. Програмско решење

4.1 Програмско решење EFR слоја

Реализација EFR слоја се налази у оквиру *Simplicity Studio* пројекта, заснован на стартном пројекту *RAIL – SoC Simple TRX*, који нуди костур са основном реализацијом бежичне комуникације између два EFR32 система. Пројекат садржи следеће програмске целине, где целину сматрамо као пар заглавља и пратеће имплементације:

- *main* – Полазна тачка програма, где се врши иницијализација програма, укључујући и основне системске функције, као и извршавање главне петље
- *app_init* – Функција за иницијализацију RAIL библиотеке и осталих компоненти
- *app_process* – Реализација аутомата са коначним бројем стања, укључујући и слање и пријем података преко радио канала
- *packet_utils* – Помоћне функције и дефиниције потребне за конфигурисање делова RAIL библиотеке и припремање података за слање

Simplicity Studio пројекти могу да укључују додатне софтверске компоненте који се могу подешавати по потреби, користећи графичку спрегу. Софтверске компоненте прате систем зависности, и многе софтверске компоненте се аутоматски укључују при стварању одређених пројекта. Одређене софтверске компоненте се могу представити преко више одређених инстанци у програму, које се могу посебно конфигурисати. Најважније софтверске компоненте које су укључене у склопу датог пројекта су следеће:

- *IO Stream* – Омогућава испис података користећи функције из стандардне библиотеке

- *IO Stream: Retarget STDIO* – Омогућава преусмеравање тока података преко задате спреге за стандардни У/И
- *IO Stream: EUSART* – Руководилац за комуникацију користећи стандардни У/И преко EUSART спреге, која се може користити као обична UART спрега
 - Инстанца *eusart0* – Спрега за директну комуникацију између EFR развојне плоче и RPi рачунара
 - Инстанца *eusart_debug* – Спрега за споредну комуникацију кроз USB периферијски уређај, намењен за логовање података
- RAIL библиотека – Главна библиотека за бежичну комуникацију
 - *RAIL Utility, Default Internal RF Path Configuration* – Конфигурација која омогућава избор активне антене на EFR32 систему
 - *Packet Assistant* – Помоћне функције за спремање пакета за слање преко RAIL позива
- *Simple LED* – Руководилац за контролисање светлећих диода преко GPIO пинова
 - Инстанца *inst0* – Повезана на светлећу диоду присутна на развојној плочи, користи се за индикацију укљученог стања развојне плоче
- *Energy Management Unit (EMU)* – Библиотека за иницијализацију и конфигурацију модула за енергетску потрошњу и напајање

Конфигурација EUSART инстанци подразумева подешавање начина комуникације, пре свега брзину комуникације, као и одређивање GPIO пинова за RX и TX смерове. У оквиру графичке конфигурације је могуће поставити и конфигурацију радио антене, где се детаљно могу поставити разни параметри, као што су основна фреквенција, број канала, модулациона шема, формат пакета, итд.

4.1.1 Функција `app_init`

```
void app_init(void);
```

У функцији `app_init` се прво позива функција `EMU_DCDCInit` која иницијализује DCDC претварач напона, где се прослеђује параметар који диктира конфигурацију претварача напона. У том параметру се поставља *Bypass* режим, у ком се не обара улазни напон, већ се директно повезује са излазним напоном за остатак микроконтролера. Даље се позива функција `sl_iostream_set_default` која одређује стандардни ток података који ће се користити у позивима функција стандардног улаза и

излаза. У овој функцији се прослеђује референца на EUSART инстанцу за дијагностику, због тога што су стандардне функције исписа прикладније за логовање података на форматиран начин, док се главни саобраћај прослеђује директним позивима за испис и читање, функцијама `sl_iostream_write` и `sl_iostream_read` респективно, где је потребно специфицирати улазно/излазну спрегу као параметар. Даље у тексту ће се стандардни излаз подразумевати као EUSART инстанца за дијагностику.

Након тога се врши иницијализација RAIL библиотеке, где се добија референца на инстанцу везана за библиотеку, врши опциона промена канала, постављају временски интервали за прелазе између стања, и на крају иницијализација адресних филтера.

4.1.2 Модул `packet_utils`

У модулу `packet_utils` се налазе дефиниције величина за меморијске бафере као и дефиниције вредности за UART комуникацију претходно описане у поглављу 3.1.3.1. Овде се дефинишу меморијски бафери који се користе за комуникацију са UART слојем, као и за бежично слање података преко RAIL библиотеке. Такође се дефинишу параметри који се користе за CSMA/CA алгоритам, и филтер адреса која представља део заглавља на овом слоју.

4.1.2.1 Функција `init_addr_filtering`

```
void init_addr_filtering(sl_rail_handle_t rail_handle);
```

Ова функција иницијализује филтер адресу која се користи за филтрирање долазних пакета. EFR32 подржава детекцију филтер адреса до два адресна поља, где обе адресе могу бити величине до 8 бајтова^[14]. Ова поља могу имати до 4 различите адресе које се могу разликовати и идентификовати, са генерално флексибилним понашањем детекције. У оквиру датог решења је потребна идентификација само једне адресе на једном пољу, тиме је реализација једноставнија. Конфигурација адресног филтрирања подразумева дефинисање структуре података `sl_rail_addr_config_t` где је могуће дефинисати понашање, позицију и величину адресних поља у оквиру садржаја пакета. Такође је потребно дефинисати табелу адреса као низ вредности који представљају адресе. Постављање конфигурације се ради уз помоћу функције `sl_rail_config_address_filter`, док се постављање филтер адресе ради позивајући `sl_rail_set_address_filter_address` за сваку адресу у низу. На крају се укључује функционалност позивајући функцију `sl_rail_enable_address_filter`.

4.1.2.2 Функција `setup_payload`

```
void setup_payload(sl_rail_handle_t rail_handle, const uint8_t *data, const
int size, const uint8_t addr);
```

Ова функција врши енкапсулацију података `data` величине `size`, користећи филтер адресу на индексу `addr`. Копирање података се врши позивом функције `memcpy`, где се подаци копирају у бафер `outpacket`. На крају се тај бафер прослеђује у позив функције `prepare_packet`, која припрема дате податке у FIFO бафер који се користи за слање порука преко RAIL библиотеке.

4.1.3 Модул `app_process`

Овде се налази функција која се позива у главној петљи и реализује главну логику овог слоја, што укључује читање и писање преко UART спреге, као и слање и пријем података преко бежичног комуникационог канала.

4.1.3.1 Функција `app_process_action`

```
void app_process_action(void);
```

У овој функцији се реализује аутомат са коначним бројем стања, извршавајући различите радње у односу на тренутно стање система. За стање `S_IDLE`, активира се режим примања порука уколико је то затражено. Ако је већ примљен пакет, тада се прелази у стање `S_RECEIVE`, где ће се даља обрада примљених података извршити, затражиће се активација режима примања поруке, и након тога се врши прелаз у `S_IDLE` стање. Слично понашање важи и за прелазак у стање `S_TRANSMIT`. У свим осталим случајевима, позива се функција `try_read_cmd` у којој се врши читање са UART спреге.

За стање `S_TRANSMIT`, ако је затражено, позива се функција `handle_packet_transmission` у којој се припремају подаци ишчитани са UART спреге. У тренутку када се добије повратна информација о успешном слању пакета, тада се обавештава горњи слој и исписује време потребно за слање целокупне поруке. На крају се прелази у стање `S_IDLE`.

Уколико је дошло до било какве грешке, прелази се у стање `S_ERROR`, где се позива функција `handle_error_state`. На крају извршавања те функције се прелази назад у стање `S_IDLE`.

4.1.3.2 Функција `sl_rail_util_on_event`

```
SL_CODE_RAM void sl_rail_util_on_event(sl_rail_handle_t rail_handle,
sl_rail_events_t events);
```

Ова функција се позива сваки пут када се деси један или више догађај у оквиру RAIL библиотеке. Позив се дешава у виду физичког прекида, тиме се цела функција извршава у контексту прекида, где су остали прекиди у општем случају онемогућени. Као последица, функција треба бити што краћа и бржа, и један начин убрзавања функције је да се сам машински код смести у радну меморију уместо програмске меморије, што се дефинише кроз макро `SL_CODE_RAM`^[15]. Поред тога, у овој функцији се проверавају стања догађаја у моменту позива функције, оглашавајући грешке уколико их има, успешно слање или пријем поруке, или потребу за калибрацијом радио хардвера. У случају успешног пријема, позива се функција `sl_rail_hold_rx_packet`, која чува садржај примљене поруке у FIFO бафер за даљу обраду.

4.1.3.3 Функција `handle_packet_transmission`

```
void handle_packet_transmission(sl_rail_handle_t rail_handle, int
prepare_payload);
```

У овој функцији се припрема поруке за слање позивом функције `setup_uart_payload`, која само позива функцију `setup_payload` са предодређеним аргументима. Након тога се шаљу подаци преко радио канала позивом функције `sl_rail_start_cca_csma_tx`, где се припремљен пакет шаље након ослушкивања комуникационог канала уз помоћ CSMA/CA. Уколико дође до грешке током позива дате функције, исписује се грешка на стандардном излазу.

4.1.3.4 Функција `start_receiving`

```
void start_receiving(sl_rail_handle_t rail_handle);
```

Ова функција једноставно позива функцију `sl_rail_start_rx`, где се активира режим ослушкивања канала за примање порука на одређен канал. Уколико дође до грешке током позива ове функције, исписује се дата грешка на стандардном излазу.

4.1.3.5 Функција `handle_received_packet`

```
void handle_received_packet(sl_rail_handle_t rail_handle);
```

Ова функција проверава да ли је претходни позив функције `sl_rail_hold_rx_packet` успешан, тј. да ли је референца на примљену поруку `rx_packet_handle` валидна након позива функције `sl_rail_get_rx_packet_info`, која учитава информације о поруци у променљиву `packet_info`. Уколико је валидна референца на пакет, учитавају се детаљније информације о пакету позивом функције `sl_rail_get_rx_packet_info`, где је могуће добити информације као што су статус интегритета поруке. Позивом функције `unpack_packet` добијамо величину садржаја

поруке, као и референцу на сам садржај, који се копира из FIFO бафера RAIL библиотеке у жељени бафер са стране апликације. Након учитавања садржаја потребно је ослободити FIFO бафер позивом `sl_rail_release_rx_packet`. На крају, врши се директно прослеђивање добијених података горњем слоју преко главне UART спреге.

4.1.3.6 Функција `handle_error_state`

```
void handle_error_state(void);
```

Ова функција обрађује последња активна стања након обраде прекида, и исписује грешке на стандардном излазу. Уколико је у питању грешка заузетог комуникационог канала, тада се додатно обавештава горњи слој о догађају. На крају извршавања, стање аутомата се пребацује на стање `S_IDLE`.

4.1.3.7 Функција `try_read_cmd`

```
void try_read_cmd(sl_rail_handle_t handle);
```

Ова функција чита податке са главне UART спреге у посебан меморијски бафер, очекивајући целокупну поруку. Након сваког читања се врши провера статуса бафера, у случају да долази до прекорачења капацитета бафера, након чега се пријављује грешка и празни бафер. Уколико је тренутно стање читања `UART_IDLE`, тада се након читања очекује код за слање поруке на почетној позицији бафера, након детекције кода прелази се у стање `UART_RECV`. Позитивна вредност после почетне позиције бафера представља дужину долазне поруке. Након читања са главне UART спреге, проверава се да ли је учитана цела порука на основу познате дужине долазне поруке. Уколико јесте, креће мерење времена потребно за слање поруке, врши се учитавање поруке у други меморијски бафер за припрему пакета. На крају се све везане променљиве и меморијски бафер за читање празне и постављају на своје првобитне вредности, враћајући аутомат стања назад у стање `UART_IDLE`.

4.2 Програмско решење UART и TUN слоја

Као што је претходно наведено, овај део програмског решења је написан користећи Python, уз додатну нестандартну библиотеку `python-pytun`, која нуди једноставну начин коришћења TUN/TAP спреге. Реализација се дели на следеће Python модуле:

- *enums* – Енумерације стања UART слоја и типови пакета који се користе у оквиру целог пројекта
- *log* – Помоћне функције за испис података на стандардни излаз

- *main* – Полазна тачка програма која иницијализује оба слоја у једној целини
- *values* – Разне бројне вредности које се референцирају у оквиру целог пројекта
- *TRX* – Реализација UART слоја
- *Network* – Реализација TUN слоја
- *FragmentBuffer* – структура за реконструкцију поруке на основу примљених фрагмената
- *FragmentGroup* – структура за фрагментовање дате поруке

4.2.1 Модул TRX

Овај модул реализује UART слој, користећи стандардну библиотеку `serial` за успостављање UART комуникације између рачунара и EFR32 развојне плоче. Као улазни параметар има путању ка датотеци која представља UART уређај, и референцу на објекат који представља горњи слој. Главна петља у којој се одвија логика UART слоја се извршава у посебној нити, извршавајући се независно од TUN слоја.

У конструктору се дефинише бафер за читање са UART спреге, као и три реда чекања за слање редовних порука, за слање порука везано за ретрансмисиону логику, и за примање порука са горњег слоја. За редове чекања се користи део стандардне библиотеке `Queue`, који осигурава безбедно читање и писање података између више нити. Осим тога се дефинишу променљиве и константе за контролу тока података, као и променљиве преко којих се води основна статистика током извршавања.

4.2.1.1 Методе `send_req` и `send_ack`

Методу `send_req` позива координатор, док методу `send_ack` позива регуларни уређај. где се конструише пакет за захтев комуникације у задатом смеру, прослеђивајући трајање временског интервала слања података. За прослеђивање података који се састоје од више бајтова, користе се методе `struct.pack` и `struct.unpack`, где је могуће дефинисати формат података и редослед бајтова, између *big-endian* и *little-endian* конвенције. Како је *big-endian* конвенција чешћа у оквиру мрежних протокола^[16], користиће се и у овој реализацији. Након што се пакет конструише, шаље се позивом методе `send_single`. Слична логика се користи и за реализацију методе `send_ack`, само са различитим типовима пакета, и разликује се по томе што враћа вредност израчунатог временског интервала на основу количине пакета у реду чекања.

4.2.1.2 Метода `send_single`

Ова метода шаље прослеђену поруку као аргумент ка доњем слоју користећи UART спрегу. На почетку поруке се налепљују индикатор почетка поруке, као и дужина поруке, након чега се поруке директно шаље. На крају се декрементира број пакета који могу да се шаљу у задатом временском слоту, и онемогућује се слање додатних порука све док се касније не добије потврда о успешном слању, пријаве грешке или истека одређеног временског интервала, чије се мерење започиње на почетку позива методе користећи стандардну библиотеку `time`.

4.2.1.3 Метода `receive`

У овој методи се обрађују пристигле поруке са нижег слоја. Уколико порука није намењена за UART слој, прослеђује се ка горњем слоју за даљу обраду. У супротном, координатор очекује пакете типа потврде захтева, и улази у стање *SENDING*. уколико је смер од координатора ка регуларном чвору. Ако је намењен смер супротан, тада се проверава да ли је временски интервал у поруци потврде једнак нули, и ако јесте, прелази се у стање *SENDING*, у супротном *LISTENING*.

Регуларан чвор очекује поруке типа захтева и враћа одговарајућу потврду у зависности од траженог смера комуникације. Ако регуларан добије захтев типа *is_req_rcv*, чвор нема поруке да шаље координатору, тада ће враћени временски слот бити једнак нули. У том случају, чита се временски интервал од координатора за додељен временски слот, прелазећи у стање *LISTENING*. У супротном, прелази у стање *SENDING*. У случају да добије захтев типа *is_req*, ту одмах прелази у стање *LISTENING*.

На крају обраде захтева и потврде, започиње се мерење времена за тренутни временски слот.

4.2.1.4 Метода `try_read`

Ова метода чита по један бајт са UART спреге ради детекције кодова који су претходно дефинисани у табели 3.3. Уколико се детектује најава заузетог комуникационог канала, тада се исписује грешка на стандардни излаз. У случају потврде успешног слања, тада се омогући слање следећег пакета и ажурира релевантна статистика. У случају најаве пристигле поруке, чита се следећи бајт и позива се метода `receive`. Уколико се прими јединствени идентификатор микроконтролера, тада аутомат са коначним бројем стања прелази у стање *IDLE*.

4.2.1.5 Метода `init_mcu`

Аутомат са коначним бројем стања има иницијално стање *NONE*, у ком UART слој није активан. На почетку програма позива се ова метода која шаље наредбу да се EFR32 ресетује на иницијално стање, како би читав систем био у познатом стању, и како би се потврдила функционалност развојне плоче. На сваком покретању система, EFR слој шаље поруку са својим јединственим идентификатором, који служи као индикатор да се развојна плоча ресетовала и поставила у иницијално стање.

4.2.1.6 Метода `loop`

У овој методи се реализује главна петља UART слоја, где се одвија аутомат са коначним бројем стања. На почетку сваке итерације, прво се врши провера истека временског слота, или провера истека времена за пријем потврде од стране регуларног чвора, у зависности од стања. У стању *IDLE*, координатор шаље захтев регуларном чвору да се започне пренос података у задатом смеру, прелазећи у стање *WAIT_ACK*, док регуларан чвор у стању *IDLE* само прима нове поруке. У стању *WAIT_ACK*, координатор чита поруке са UART спреге, очекивајући потврду захтева.

У стању *SENDING*, прво се шаљу поруке са реда чекања за ретрансмисиону логику, и тек након тога са реда чекања за све остале поруке. Порука ће се послати само уколико је то омогућено са условом `can_send`, који је тачан ако је добијена потврда о успешном слању претходне поруке, или ако је стигло до прекорачења времена чекања за ту потврду. Додатни услови за слање поруке су доступност порука у датим редовима чекања, и да је број порука послатих у тренутном временском слоту мањи од дефинисаног ограничења. Слање порука се врши позивом методе `send_single`. На крају итерације се врши провера истека времена чекања за потврду о успешном слању, где се омогућава слање порука уколико је прекорачено време чекања.

4.2.1.7 Метода `reset_state`

Ова метода пребацује UART слој у стање *IDLE* и ресетује релевантне променљиве на почетно стање. Ако је чвор координатор, тада се додатно врши обртање смера комуникације између координатора и регуларног чвора.

4.2.2 Класа `FragmentBuffer`

Ова класа је намењена да прима и реконструише фрагменте одређене групе. Објекат класе се инстанцира када се прими фрагмент који је први по редоследу целокупне поруке, и прима као улазне аргументе идентификатор, величину оригиналне

поруке, и CRC кôд. Садржај реконструисане поруке се учитава у меморијски бафер типа `bytearray`, који има унапред резервисану величину у бајтовима. Прати се колико је бајтова попуњено у баферу, да ли је примљен последњи фрагмент, и мери се време које је потребно да се реконструише оригинална порука, са дефинисаним временом истека од пет секунди.

4.2.2.1 Метода `put_fragment`

Ова метода прима као улазне параметре садржај фрагмента, позицију у бајтовима где је потребно сместити садржај, и индикатор да ли је у питању последњи фрагмент. Садржај фрагмента се учитава у меморијски бафер на задатој позицији и након тога се ажурира број бајтова учитан на основу дужине садржаја фрагмента. Уколико је индикатор последњег фрагмента постављен на вредност `True`, тада се ажурира и вредност унутар објекта ове класе.

4.2.2.2 Метода `is_valid`

Ова метода проверава валидност реконструисане поруке. Порука се сматра да је успешно реконструисана ако је број примљених и учитаних бајтова једнак оригиналној величини поруке, и ако је израчунат CRC кôд једнак оригиналној вредности из заглавља почетног фрагмента.

4.2.2.3 Класа `FragmentGroup`

Ова класа је намењена да преузме садржај оригиналне поруке и да изврши фрагментацију над њом, стварајући низ фрагмената над којима се додатно врши енкапсулација где се налепљују поља заглавља фрагмента. Овде се такође води рачуна о броју покушаја слања фрагмената, где након три неуспешна покушаја сви фрагменти се одбацују.

4.2.2.4 Метода `encapsulate`

Ова метода прима поруку, индикатор типа поруке, број који представља позицију или дужину поруке у бајтовима, и CRC вредност као улазне аргументе, где је CRC необавезни аргумент. Метода конструише заглавље поруке фрагмента и враћа поруку са додатим заглављем пре почетка садржаја.

4.2.2.5 Метода `do_fragmentation`

У овој методи се оригинална порука расцепљује у једнаке делове изузев почетног фрагмента, који има веће заглавље, и последњег фрагмента који не мора бити исте величине као и остали фрагменти. Фрагменти имају максималну величину, узимајући у

обзир дужине осталих заглавља након енкапсулације на сваком слоју. Након енкапсулације првог фрагмента, у петљи се врши енкапсулација осталих фрагмената, позивајући методу `encapsulate` за сваки фрагмент. У последњој итерацији се додаје тип `is_last_fragment` на индикатор типа поруке.

4.2.3 Модул Network

У овом модулу се дефинише класа TUN који представља TUN слој. Унутар класе се иницијализују инстанце `TunTapDevice` и `TRX` класе, успостављајући везу са доњим слојем. Као улазне параметре прима своју IP адресу, IP адресу другог чвора, путања датотеке UART уређаја и индикатор да ли је уређај координатор или регуларан чвор.

У конструктору објекта класе `TunTapDevice` се прослеђују конфигурационе заставице `IFF_TUN` и `IFF_NO_PI`, где прва заставица одређује тип уређаја, док друга заставица спречава додавање додатних информација о пакету као заглавље долазних пакета. Додатно се дефинишу параметри за подмрежу и MTU самог TUN уређаја, са вредностима `255.255.255.0` и `1500` респективно. Треба напоменути да MTU TUN уређаја и MTU TUN слоја се разликују, где у оквиру самог TUN слоја MTU зависи од максималне дужине пакета који EFR32 SoC може да преноси, што је 256 бајтова.

Иницијализује се ред чекања за пакете везани за ретрансмисиону логику `ack_buf`, два речника `frag_buf` и `send_buf`, који садрже инстанце класе `FragmentBuffer` и `FragmentGroup`, и променљиве везане за статистику примљених и послатих потврда. Класа има рутине за пријем и за слање порука, које се извршавају у две различите нити и тиме су независне операције.

4.2.3.1 Метода `send_ack`

Ова метода конструише пакет потврде која може бити негативна или позитивна, у зависности од улазног параметра ове методе. Инкрементирају се променљиве које прате статистику о послатим потврдама, и на крају се пакет убацује у ред чекања `ack_buf`.

4.2.3.2 Метода `stop`

Ова метода прекида се и затвара TUN уређај, прекида извршавање свих петљи, и на крају се исписују све измерене статистике током рада целокупног система.

4.2.3.3 Метода `send_routine`

Овде се покреће TUN уређај и започиње главна петља за обраду прикупљених порука са TUN слоја. Прво се врши провера верзије IP пакета на основу прва четири

бита заглавља^[17] и одбацује се ако је у питању IPv6 пакет. Даље се проверавају бајтови [16, 19], који означавају адресу пријемника, и одбацују се пакети где та адреса не одговара адреси прослеђена као улазни аргумент у конструктору.

Након филтрирања се проверава дужина учитаног пакета, и ако је дужина већа од дефинисане MTU вредности за TUN слој, тада се врши фрагментација поруке. Инкрементира се вредност идентификатора групе фрагмената, прелазећи назад на нулу ако се прекорачи горња граница вредности (како је величина поља за идентификатор групе 2 бајта, тада ће горња гранична вредност бити $2^{16} - 1$), и у речник `send_buf` се на кључ идентификатора конструише објекат класе `FragmentGroup`, у ком се врши фрагментација. Генерисани фрагменти се прослеђују ка доњем слоју позивом методе `TRX.add_msg`.

Пакети који не морају да пролазе кроз фрагментацију се директно прослеђују ка доњим слојевима. Енкапсулација за ове пакете је само додавање нултог бајта на почетку поруке, што представља *empty* тип пакета. На крају, поруке везане за ретрансмисиону логику (потврде и фрагменти који се поново шаљу) се прослеђују ка доњем слоју где се убацују у приоритетнији ред чекања.

4.2.3.4 Метода `recv_routine`

У овој методи се прикупљају поруке са UART слоја, и врши се декапсулација и даља обрада порука у зависности од типа пакета. Пакети са типом *empty* се декапсулирају и директно прослеђују ка горњим слојевима мрежног стека.

Пакете са основним типом *is_ack* парсирамо и читамо идентификацију групе фрагмената, гледајући да ли постоји објекат класе `FragmentGroup` са тим идентификатором у речнику `send_buf`. У случају да постоји, и ако је у питању позитивна потврда, тада се објекат брише из речника. Ако је у питању негативна потврда, тада се инкрементира број покушаја тог објекта, и фрагменти се поново прослеђују ка доњем слоју, на приоритетнији ред чекања. Ако је три пута неуспешно слање фрагмената, тада се брише објекат из речника и пријављује грешка.

Пакете са основним типом *is_fragment* парсирамо и читамо податке из заглавља. Ако је у питању фрагмент типа *is_first_fragment*, тада се конструише објекат класе `FragmentBuffer` који се убацује у речник `frag_buf`, и садржај декапсулираног пакета се уноси као први фрагмент. За остале фрагменте се налази одговарајући `FragmentBuffer` објекат где ће се уносити садржај фрагмента, све док се не реконструише у потпуности оригинална порука. Проверава се валидност реконструкције, и у успешном случају,

порука се прослеђује ка горњим слојевима, и шаље се позитивна потврда. У супротном, шаље се негативна потврда. У оба случаја брише се објекат из речника `frag_buf`. Ако `FragmentBuffer` са датим идентификатором не постоји у речнику `frag_buf`, тада се инстанцира објекат у ком ће се остатак фрагмената са истим идентификатором уносити, где ће се имплицитно послати негативна потврда након неуспешне реконструкције.

На крају се проверава да ли је дошло до истека времена за `FragmentBuffer` објекте, у случајевима где није успешно примљен последњи фрагмент који служи као индикатор завршене реконструкције. У тим случајевима се брише објекат из речника и шаље негативна потврда.

4.2.4 Модул `main`

Ово је полазна тачка програмског решења које се извршава на страни RPi рачунара. Са командне линије очекује следеће улазне аргументе:

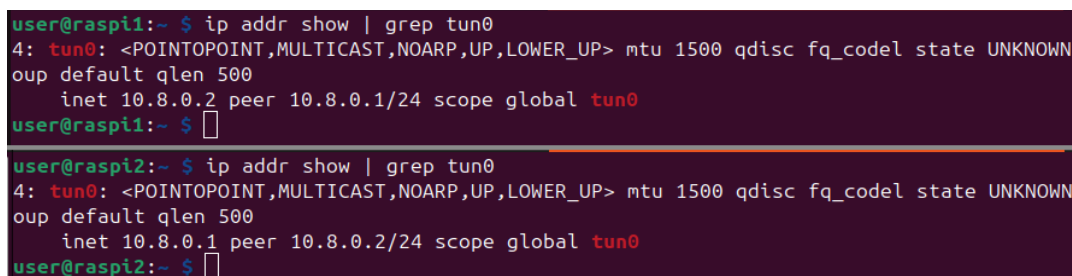
- Путања ка UART уређају
- IP адреса TUN уређаја
- IP адреса другог чвора
- MTU мрежног стека
- Индикатор да ли је чвор координатор или регуларан чвор

Функција `main` конфигурише MTU вредност која је се глобално може приступити из модула `values`, инстанцира TUN класу из модула `Network`, и на крају позива методе `init_mcu` и `send_routine`.

5. Резултати

У овом поглављу ће се извршити валидација и посматрати основне перформансе целокупног система. Очекиван резултат је омогућен пренос IP пакета са једног рачунара на други са оба смера, користећи реализоване слојеве мрежног стека.

Пре свега је потребно проверити да ли су инстанцирани TUN уређаји са оба рачунара, и да ли су конфигурисани параметри коректно примењени. Ако је TUN уређај коректно инстанциран, тада ће се појавити у листи мрежних уређаја са именом `tun0`.



```
user@raspi1:~ $ ip addr show | grep tun0
4: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN
    oup default qlen 500
    inet 10.8.0.2 peer 10.8.0.1/24 scope global tun0
user@raspi1:~ $

user@raspi2:~ $ ip addr show | grep tun0
4: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN
    oup default qlen 500
    inet 10.8.0.1 peer 10.8.0.2/24 scope global tun0
user@raspi2:~ $
```

Слика 5.1: Приказ исправне конфигурације TUN уређаја

5.1 Тестирање уз помоћу `ping` команде

Као основни вид тестирања исправности мреже, користиће се `ping` команда са различитим параметрима како би се проверила валидност конекције и њене перформансе у основном случају, узимајући да је MTU мреже 256 бајтова. Покретањем команде на оба чвора без додатних параметара, резултати су следећи:

параметара

дошло до фрагментације пакета, што се може видети на излазу од мрежног стека:

Слика 5.3: Логовани излази из координатора и регуларног чвора, без фрагментације

Чвору:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=1/256, ttl=64 (reply in 2)
2	0.200797	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=1/256, ttl=64 (request in 1)
3	1.000014	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=2/512, ttl=64 (reply in 4)
4	1.172583	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=2/512, ttl=64 (request in 3)
5	2.000748	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=3/768, ttl=64 (reply in 6)
6	2.144137	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=3/768, ttl=64 (request in 5)
7	3.001303	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=4/1024, ttl=64 (reply in 8)
8	3.108415	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=4/1024, ttl=64 (request in 7)
9	4.001654	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=5/1280, ttl=64 (reply in 10)
10	4.222854	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=5/1280, ttl=64 (request in 9)
11	5.001942	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=6/1536, ttl=64 (reply in 12)
12	5.143869	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=6/1536, ttl=64 (request in 11)
13	6.002942	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=7/1792, ttl=64 (reply in 14)
14	6.165741	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=7/1792, ttl=64 (request in 13)
15	7.002890	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=8/2048, ttl=64 (reply in 16)
16	7.125707	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=8/2048, ttl=64 (request in 15)
17	8.002899	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=9/2304, ttl=64 (reply in 18)
18	8.240412	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=9/2304, ttl=64 (request in 17)
19	9.003525	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=10/2560, ttl=64 (reply in 20)
20	9.161823	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=10/2560, ttl=64 (request in 19)
21	10.003981	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=11/2816, ttl=64 (reply in 22)
22	10.183752	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=11/2816, ttl=64 (request in 21)
23	11.003877	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=12/3072, ttl=64 (reply in 24)
24	11.145858	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=12/3072, ttl=64 (request in 23)
25	12.004027	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=13/3328, ttl=64 (reply in 26)
26	12.167670	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=13/3328, ttl=64 (request in 25)
27	13.004828	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=14/3584, ttl=64 (reply in 28)
28	13.188191	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=14/3584, ttl=64 (request in 27)
29	14.004684	10.8.0.2	10.8.0.1	ICMP	84	Echo (ping) request id=0xb143, seq=15/3840, ttl=64 (reply in 30)
30	14.159710	10.8.0.1	10.8.0.2	ICMP	84	Echo (ping) reply id=0xb143, seq=15/3840, ttl=64 (request in 29)
31	53.668695	fe80::2a89:62e2:3286:6d6a	ff02::2	ICMPv6	48	Router Solicitation

Header Checksum: 0x2ca0 [validation disabled]	0000 45 00 00 54 39 f7 00 00 40 01 2c a0 0a 08 00
[Header checksum status: Unverified]	0010 0a 08 00 02 00 00 7f e5 b1 43 00 01 3e 79 2
Source Address: 10.8.0.1	0020 00 00 00 00 9f 1f 08 00 00 00 00 10 11 1
Destination Address: 10.8.0.2	0030 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 2
Internet Control Message Protocol	0040 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 3
Type: 0 (Echo (ping) reply)	0050 34 35 36 37
Code: 0	
Checksum: 0x7fe5 [correct]	
[Checksum Status: Good]	
Identifier (BE): 45379 (0xb143)	
Identifier (LE): 17329 (0x43b1)	
Sequence Number (BE): 1 (0x0001)	
Sequence Number (LE): 256 (0x0100)	
[Request frame: 1]	
[Response time: 200.797 ms]	
Timestamp from icmp data: Jun 11, 2026 11:00:46.532383000 CEST	
[Timestamp from icmp data (relative): 0.200863000 seconds]	
Data (40 bytes)	
Data: 101112131415161718191a1b1c1d1e1f202122232425262728292a2b2c2d2e2f3031323334353637	
Length: 40	

Слика 5.4: Детаљан приказ ICMP пакета кроз Wireshark

Могуће је поставити различите параметре за *ping* који утичу на величину ICMP пакета, као и временски интервал између слања порука. Постављајући величину садржаја ICMP пакета на 500 бајтова, где се очекује фрагментација пакета, добијају се следећи резултати:

```
user@raspi1:~ $ ping -s 500 10.8.0.1
PING 10.8.0.1 (10.8.0.1) 500(528) bytes of data.
508 bytes from 10.8.0.1: icmp_seq=1 ttl=64 time=1024 ms
508 bytes from 10.8.0.1: icmp_seq=2 ttl=64 time=746 ms
508 bytes from 10.8.0.1: icmp_seq=3 ttl=64 time=676 ms
508 bytes from 10.8.0.1: icmp_seq=4 ttl=64 time=766 ms
508 bytes from 10.8.0.1: icmp_seq=5 ttl=64 time=699 ms
508 bytes from 10.8.0.1: icmp_seq=6 ttl=64 time=783 ms
508 bytes from 10.8.0.1: icmp_seq=7 ttl=64 time=718 ms
508 bytes from 10.8.0.1: icmp_seq=8 ttl=64 time=647 ms
508 bytes from 10.8.0.1: icmp_seq=9 ttl=64 time=732 ms
508 bytes from 10.8.0.1: icmp_seq=10 ttl=64 time=663 ms
508 bytes from 10.8.0.1: icmp_seq=11 ttl=64 time=747 ms
508 bytes from 10.8.0.1: icmp_seq=12 ttl=64 time=681 ms
508 bytes from 10.8.0.1: icmp_seq=13 ttl=64 time=763 ms
508 bytes from 10.8.0.1: icmp_seq=14 ttl=64 time=694 ms
508 bytes from 10.8.0.1: icmp_seq=15 ttl=64 time=780 ms
508 bytes from 10.8.0.1: icmp_seq=16 ttl=64 time=715 ms
^C
--- 10.8.0.1 ping statistics ---
17 packets transmitted, 16 received, 5.88235% packet loss, time 16003ms
rtt min/avg/max/mdev = 646.533/739.665/1023.762/83.833 ms, pipe 2
```

Слика 5.5: Резултати ping команде са величином пакета од 528 бајтова

```

[09:47:34.891380][TUN][INFO] Encountered a fragment: flags: 0b1 ID: 58 Offset: 236 is_last: False
[09:47:34.897459][TUN][INFO] Encountered a fragment: flags: 0b101 ID: 58 Offset: 476 is_last: True
[09:47:34.897875][TUN][INFO] Sending ACK for 58
[09:47:35.298743][UART][INFO] Timeout reached, resetting state
[09:47:35.298959][UART][INFO] Entered IDLE mode
[09:47:35.299183][UART][INFO] Sending request: Master -> Slave: True; Timeframe: 360ms
[09:47:35.313312][UART][INFO] Entering SENDING mode
[09:47:35.363725][UART][INFO] Sending ACK packet, QSize: 1
[09:47:35.370870][UART][INFO] Sending packet, QSize: 3
[09:47:35.371123][UART][DEBUG] Fragment flags: 0b11, ID = 59
[09:47:35.450691][UART][INFO] Sending packet, QSize: 2
[09:47:35.450953][UART][DEBUG] Fragment flags: 0b1, ID = 59
[09:47:35.529036][UART][INFO] Sending packet, QSize: 1
[09:47:35.529326][UART][DEBUG] Fragment flags: 0b101, ID = 59
[09:47:35.702634][UART][INFO] Timeout reached, resetting state
[09:47:35.702840][UART][INFO] Entered IDLE mode
[09:47:35.703066][UART][INFO] Sending request: Master -> Slave: False; Timeframe: 360ms
[09:47:35.719180][UART][INFO] Entering LISTENING mode
[09:47:35.727084][TUN][INFO] Received ACK for 58, safe to delete fragments
[09:47:35.826226][TUN][INFO] Encountered a fragment: flags: 0b11 ID: 59 Size: 528 CRC: 1352576959
[09:47:35.905081][TUN][INFO] Encountered a fragment: flags: 0b1 ID: 59 Offset: 236 is_last: False
[09:47:35.912200][TUN][INFO] Encountered a fragment: flags: 0b101 ID: 59 Offset: 476 is_last: True
[09:47:35.912659][TUN][INFO] Sending ACK for 59
[09:47:36.313527][UART][INFO] Timeout reached, resetting state
[09:47:36.313747][UART][INFO] Entered IDLE mode
[09:47:36.313973][UART][INFO] Sending request: Master -> Slave: True; Timeframe: 360ms
[09:47:36.330661][UART][INFO] Entering SENDING mode
[09:47:36.381116][UART][INFO] Sending ACK packet, QSize: 1
[09:47:36.389246][UART][INFO] Sending packet, QSize: 3
[09:47:36.389495][UART][DEBUG] Fragment flags: 0b11, ID = 60
[09:47:36.468680][UART][INFO] Sending packet, QSize: 2
[09:47:36.468950][UART][DEBUG] Fragment flags: 0b1, ID = 60
[09:47:36.546917][UART][INFO] Sending packet, QSize: 1
[09:47:36.547210][UART][DEBUG] Fragment flags: 0b101, ID = 60
[09:47:35.315050][UART][DEBUG] Received packet b'...'
[09:47:35.315271][UART][DEBUG] Received request
[09:47:35.315455][UART][INFO] Sending ACK with timeframe 0ms
[09:47:35.315738][UART][INFO] Entering LISTENING mode
[09:47:35.380960][UART][DEBUG] Received packet b'...'
[09:47:35.381427][TUN][INFO] Received ACK for 55, safe to delete fragments
[09:47:35.481999][UART][DEBUG] Received packet b'\xf9\xde\xad\xbe\xef\x03\x00;\x02\x10\xdb\xea\x19\x00\x02\x100H\x00\x01\xf4'...
[09:47:35.482477][TUN][INFO] Encountered a fragment: Flags: 0b11 ID: 59 Size: 528 CRC: 3609570585
[09:47:35.560142][UART][DEBUG] Received packet b'\xf9\xde\xad\xbe\xef\x01\x00;\x00\xec\x00\x12\x03\x04\x05\x06\x07\x08\x09\xda\xdb\xdc\xdd\xde'
[09:47:35.560993][TUN][INFO] Encountered a fragment: flags: 0b1 ID: 59 Offset: 236 is_last: False
[09:47:35.567125][UART][DEBUG] Received packet b'\x00\xad\xbe\xef\x05'...
[09:47:35.567957][TUN][INFO] Encountered a fragment: flags: 0b101 ID: 59 Offset: 476 is_last: True
[09:47:35.569131][TUN][INFO] Sending ACK for 59
[09:47:35.719228][UART][DEBUG] Received packet b'...'
[09:47:35.719454][UART][DEBUG] Received new request
[09:47:35.719650][UART][INFO] Sending ACK with timeframe 360ms
[09:47:35.719910][UART][INFO] Entering SENDING mode
[09:47:35.727648][UART][INFO] Sending ACK packet, QSize: 1
[09:47:35.735425][UART][INFO] Sending packet, QSize: 3
[09:47:35.735662][UART][DEBUG] Fragment flags: 0b11, ID = 59
[09:47:35.813304][UART][INFO] Sending packet, QSize: 2
[09:47:35.813573][UART][DEBUG] Fragment flags: 0b1, ID = 59
[09:47:35.892151][UART][INFO] Sending packet, QSize: 1
[09:47:35.892412][UART][DEBUG] Fragment flags: 0b101, ID = 59
[09:47:36.116167][UART][INFO] Timeout reached, resetting state
[09:47:36.116389][UART][INFO] Entered IDLE mode
[09:47:36.330782][UART][DEBUG] Received packet b'...'
[09:47:36.331005][UART][DEBUG] Received request
[09:47:36.331194][UART][INFO] Sending ACK with timeframe 0ms
[09:47:36.331482][UART][INFO] Entering LISTENING mode
[09:47:36.399295][UART][DEBUG] Received packet b'...'
[09:47:36.399762][TUN][INFO] Received ACK for 56, safe to delete fragments
[09:47:36.500051][UART][DEBUG] Received packet b'\xf9\xde\xad\xbe\xef\x03\x00;\x02\x10\x09\x02\x10\x0c\x00\x02\x100H\x00\x01\xf4'...
[09:47:36.500532][TUN][INFO] Encountered a fragment: Flags: 0b11 ID: 60 Size: 528 CRC: 3118603916
[09:47:36.578149][UART][DEBUG] Received packet b'\xf9\xde\xad\xbe\xef\x01\x00;\x00\xec\x00'

```

Слика 5.6: Логовани излази из мрежног стека, са фрагментацијом и потврдама

Може се приметити конзистентно повећање трајања преноса података као резултат потребе за фрагментацијом. Постављањем величине садржаја пакета на 1472 бајтова, урачунајући на енкапсулацију, може се проверити слање пакета са максималном величином од 1500 бајтова, која је стандардна MTU вредност за IP саобраћај у *Ethernet* мрежама:

```

user@raspi2:~$ ping -c 15 -s 1472 -i 2 10.8.0.2
PING 10.8.0.2 (10.8.0.2) 1472(1500) bytes of data.
1480 bytes from 10.8.0.2: icmp_seq=1 ttl=64 time=1958 ms
1480 bytes from 10.8.0.2: icmp_seq=2 ttl=64 time=1706 ms
1480 bytes from 10.8.0.2: icmp_seq=3 ttl=64 time=1964 ms
1480 bytes from 10.8.0.2: icmp_seq=4 ttl=64 time=1721 ms
1480 bytes from 10.8.0.2: icmp_seq=5 ttl=64 time=1983 ms
1480 bytes from 10.8.0.2: icmp_seq=6 ttl=64 time=1735 ms
1480 bytes from 10.8.0.2: icmp_seq=7 ttl=64 time=1994 ms
1480 bytes from 10.8.0.2: icmp_seq=8 ttl=64 time=1746 ms
1480 bytes from 10.8.0.2: icmp_seq=9 ttl=64 time=1941 ms
1480 bytes from 10.8.0.2: icmp_seq=10 ttl=64 time=1696 ms
1480 bytes from 10.8.0.2: icmp_seq=11 ttl=64 time=1955 ms
1480 bytes from 10.8.0.2: icmp_seq=12 ttl=64 time=1708 ms
1480 bytes from 10.8.0.2: icmp_seq=13 ttl=64 time=1966 ms
1480 bytes from 10.8.0.2: icmp_seq=14 ttl=64 time=1718 ms
1480 bytes from 10.8.0.2: icmp_seq=15 ttl=64 time=1974 ms

--- 10.8.0.2 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 27994ms
rtt min/avg/max/mdev = 1695.613/1850.969/1994.397/124.862 ms
user@raspi2:~$

user@raspi2:~$ ping -c 15 -s 1472 10.8.0.2
PING 10.8.0.2 (10.8.0.2) 1472(1500) bytes of data.
1480 bytes from 10.8.0.2: icmp_seq=2 ttl=64 time=2165 ms
1480 bytes from 10.8.0.2: icmp_seq=3 ttl=64 time=3334 ms
1480 bytes from 10.8.0.2: icmp_seq=4 ttl=64 time=2809 ms
1480 bytes from 10.8.0.2: icmp_seq=5 ttl=64 time=4732 ms
1480 bytes from 10.8.0.2: icmp_seq=6 ttl=64 time=4215 ms
1480 bytes from 10.8.0.2: icmp_seq=7 ttl=64 time=6236 ms
1480 bytes from 10.8.0.2: icmp_seq=8 ttl=64 time=5716 ms
1480 bytes from 10.8.0.2: icmp_seq=9 ttl=64 time=7735 ms
1480 bytes from 10.8.0.2: icmp_seq=10 ttl=64 time=7235 ms
1480 bytes from 10.8.0.2: icmp_seq=11 ttl=64 time=19400 ms
1480 bytes from 10.8.0.2: icmp_seq=12 ttl=64 time=9776 ms
1480 bytes from 10.8.0.2: icmp_seq=13 ttl=64 time=11795 ms
1480 bytes from 10.8.0.2: icmp_seq=14 ttl=64 time=11272 ms
1480 bytes from 10.8.0.2: icmp_seq=15 ttl=64 time=11440 ms
1480 bytes from 10.8.0.2: icmp_seq=15 ttl=64 time=10923 ms

--- 10.8.0.2 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 14173ms
rtt min/avg/max/mdev = 2164.989/7918.979/19400.020/4436.640 ms, pl
user@raspi2:~$

```

Слика 5.7: Резултати ping команде са величином пакета од 1500 бајтова

У првом случају, постављен је интервал слања између порука да буде две секунде, док у другом случају је интервал на стандардној вредности од једне секунде. У другом случају се може приметити неуспешно слање првог ICMP пакета, као и поновно слање и успешан пријем истог. Како је укупно време слања и пријема значајно дуже од

При искључењу мрежног стека, приказује се статистика различитих параметара на стандардном излазу:

```
[10:32:20.779074][TUN ][DEBUG] Statistics:
[10:32:20.780588][TUN ][DEBUG]   Packets sent (unconfirmed): 6673
[10:32:20.780870][TUN ][DEBUG]   Packets sent (confirmed):   6672
[10:32:20.781073][TUN ][DEBUG]   Packets sent success rate: 0.9999
[10:32:20.781196][TUN ][DEBUG]   Packets received:      6653
[10:32:20.781315][TUN ][DEBUG]   Bytes sent:           365802 bytes
[10:32:20.781432][TUN ][DEBUG]   Bytes received:      364862 bytes
[10:32:20.781571][TUN ][DEBUG]   Send time (min):      5.57ms
[10:32:20.781724][TUN ][DEBUG]   Send time (avg):      60.54ms
[10:32:20.781850][TUN ][DEBUG]   Send time (max):      79.68ms
[10:32:20.781969][TUN ][DEBUG]   Single send time:     90ms
[10:32:20.782092][TUN ][DEBUG]   ACKs sent:            249
[10:32:20.782206][TUN ][DEBUG]   NACKs sent:            1
[10:32:20.782321][TUN ][DEBUG]   ACKs received:        238
[10:32:20.782444][TUN ][DEBUG]   NACKs received:        2
```

Слика 5.10: Приказ статистике на крају рада мрежног стека

Мерење времена слања се рачуна у оквиру UART слоја, од тренутка како се позове функција за слање једног пакета, до тренутка када се прими порука успешног слања са стране EFR слоја. Посматрањем излаза са EFR32 спреге за дијагностику, може се закључити да време слања једног пакета зависи од његове величине, са горњом границом од око 80ms за највеће пакете. Мерење на EFR слоју се разматра од почетка припремања поруке за слање након преузетих података са UART спреге, до потврде успешног слања од стране RAIL догађаја. Најдуже време (50-60ms) је између позива RAIL функције за слање поруке и потврде успешног слања након обраде прекида.

```
Sending packet of 249 bytes
TX received: 1 13
Sent packet in 55 ms
Sending packet of 49 bytes
TX received: 101 13
Sent packet in 13 ms
Sent packet in 4 ms
RX received: 11 14
RX received: 1 14
RX received: 101 14
Sent packet in 4 ms
Sent packet in 4 ms
Sending packet of 249 bytes
TX received: 11 14
Sent packet in 56 ms
Sending packet of 249 bytes
TX received: 1 14
Sent packet in 55 ms
Sending packet of 49 bytes
TX received: 101 14
Sent packet in 14 ms
Sent packet in 5 ms
Sent packet in 4 ms
Sent packet in 4 ms
Sent packet in 6 ms
```

Слика 5.11: Излаз из дијагностичке
UART спреге EFR слоја

5.2 Тестирање функционалности користећи друге програме

Како је реализован мрежни стек представљен као регуларна мрежна спрега у Линукс оперативном систему, и како је могуће слати произвољне IP пакете кроз мрежни стек, може се тестирати функционалност и интегритет система користећи остале апликације. Примера ради, могуће је успоставити SSH везу између два чвора прослеђивајући IP адресу која је конфигурирана на TUN слоју, као што је приказано у слици 5.12.

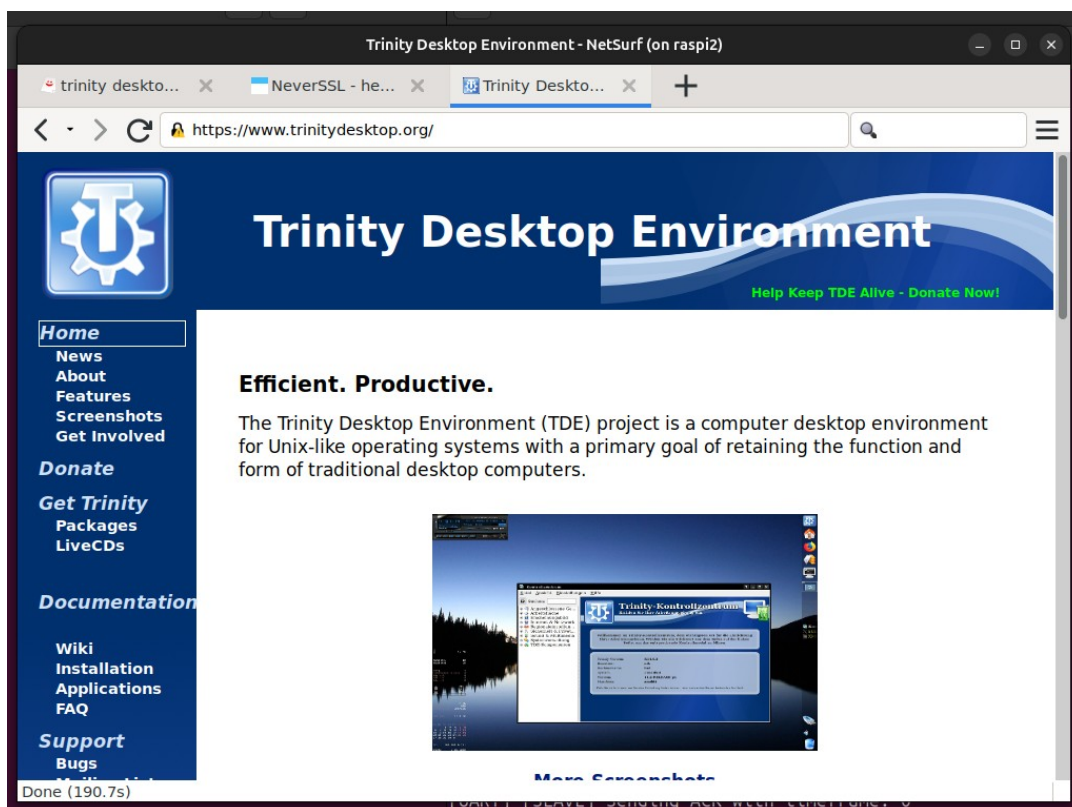
```
user@raspi1:~$ ssh user@10.8.0.1
The authenticity of host '10.8.0.1 (10.8.0.1)' can't be established.
ED25519 key fingerprint is SHA256:lbdZniEp/W9seKYJr+bMkJ1R6e+mbiezSXLQLQNNa0.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '10.8.0.1' (ED25519) to the list of known hosts.
user@10.8.0.1's password:
Linux raspi2 6.18.29+rpt-rpi-v8 #1 SMP PREEMPT Debian 1:6.18.29-1+rpt1 (2026-05-12) aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

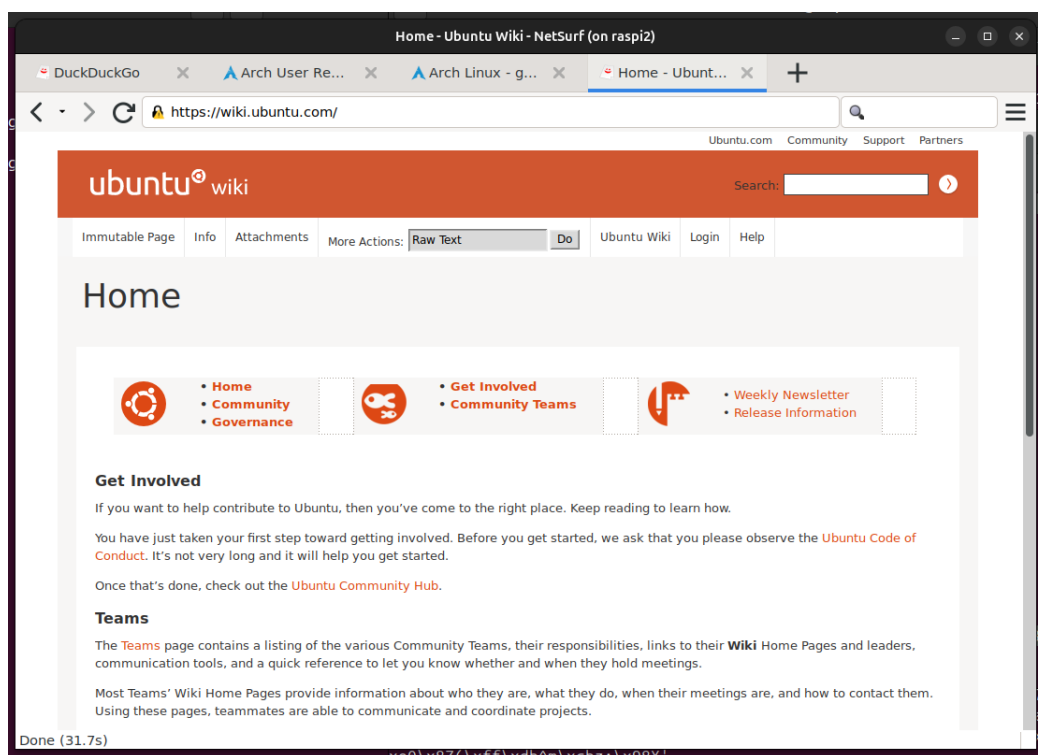
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Jun  3 08:54:12 2026 from 192.168.100.12
user@raspi2:~$ 
user@raspi2:~$ 
user@raspi2:~$ 
user@raspi2:~$ uname -r
6.18.29+rpt-rpi-v8
user@raspi2:~$ uname -a
Linux raspi2 6.18.29+rpt-rpi-v8 #1 SMP PREEMPT Debian 1:6.18.29-1+rpt1 (2026-05-12) aarch64 GNU/Linux
user@raspi2:~$
```

Слика 5.12: Успостављање везе између два чвора користећи SSH протокол

Као додатни пример, могуће је на једном чвору који има приступ интернету покренути HTTP прокси сервер (енгл. *proxy server*), пружајући другом чвору приступ интернету као посредник. Други чвор са одговарајућом конфигурацијом која показује на прокси сервер може преко претраживача да приступи веб страницама као што је приказано у сликама 5.13 и 5.14. У датим примерима се користи *NetSurf* претраживач, који се приказује на секундарном рачунару користећи функционалност SSH протокола да прослеђује X11 графичке апликације. У доњем ћошку се може видети потребно време учитавање страница, где је брзина учитавања очекивано спора због физичке архитектуре система, нарочито у случајевима када страница садржи слике.



Слика 5.13: Пример учитавања странице www.trinitydesktop.org користећи реализован мрежни стек



Слика 5.14: Пример учитавања странице wiki.ubuntu.com користећи реализован мрежни стек

5.3 Пропусност података

Могуће је израчунати колика је пропусност корисних података користећи „трошак“ (енгл. *overhead*) заглавља који се додавају кроз сваки слој. Изузимајући пакете намењени за контролу тока података и ретрансмисиону логику, посматрамо случај где се шаље пакет од 1500 корисних бајтова. За MTU од 256 бајтова, за регуларне фрагменте следи:

$$F = MTU - O_{total} = MTU - O_{EFR} - O_{UART} - O_{TUN} = 256 - 5 - 2 - 5 = 244$$

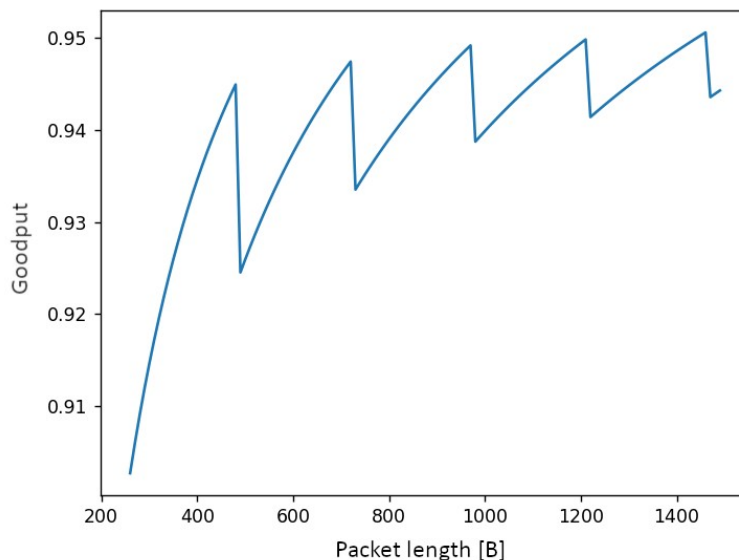
$$F_0 = MTU - O_{total0} = MTU - O_{EFR} - O_{UART} - O_{TUN0} = 256 - 5 - 2 - 9 = 240$$

Где је O_{total0} укупан трошак за прве фрагменте, док је O_{total} укупан трошак за остале фрагменте. O_{TUN0} представља величину заглавља за прве фрагменте, који је већи због додатог CRC поља. Број фрагмената који је потребан да се пренесе дата порука је:

$$N_{fragments} = 1 + \lceil \frac{D - F_0}{F} \rceil = 1 + \lceil \frac{1500 - 240}{244} \rceil = 7$$

На крају, пропусност корисних података за дату поруку је следећи:

$$D_{good} = \frac{D}{D + ((N_{fragments} - 1)O_{total} + O_{total0})} = \frac{1500}{1500 + 88} \approx 0.94$$



Слика 5.15: Граф пропусности корисних података за различите величине поруке у бајтовима, урачунајући фрагментацију података

Примењивајући дати поступак рачунања пропусности у општијем случају за поруке различитих величина, добија се граф приказан у слици 5.15. Може се приметити

конзистентан раст у пропусности како расте број бајтова у поруци, где је максимална пропусност вредности 0.95 за поруке са величином од 1460 бајтова. Додатно се примећује пад пропусности који се дешава сваки пут када је потребно вршити додатну фрагментацију поруке. У тим случајевима, последњи фрагмент садржи врло мало корисних података у односу на заглавље, и пропусност расте како корисни садржај последњег фрагмента расте.

5.4 Брзина преноса података

У овом поглављу ће се израчунати процена брзине преноса података на основу резултата *ping* команде. Разматраће се време потребно да се директно пошаље ICMP захтев од укупно 1500 бајтова, узимајући у обзир енкапсулацију заглавља, у једном смеру. Команда *ping* мери укупно време које је потребно да се пошаље ICMP захтев и да се прими одговор, који је исте величине као и захтев. Како су перформансе између оба чвора исте без обзира на улоге, узеће се претпоставка да је време преноса поруке у једном смеру једнака половини датог укупног времена од *ping* команде. Такође је битно напоменути да ће се занемарити време које је потребно *ping* команди да се изврши обрада слања захтева и пријема одговора, као и време потребно да се изврши синхронизација између два чвора пре слања порука, како је трајање слања самих пакета значајно дужи у односу на наведене операције. Додатно ће се занемарити садржај контролних пакета и садржаја преамбуле сваког пакета. Израчунате брзине су добијене на следећи начин:

$$S = \frac{1586 \cdot 8}{\frac{RTT}{2}}$$

Где је RTT укупно време слања ICMP захтева и примање одговора, у милисекундама. Мерена брзина је у килобитима по секунди (kbits/s).

Број узорака	Минимална брзина	Средња брзина	Максимална брзина	Стандардна девијација
22	11.721	14.179	20.044	1.985

Табела 5.1: Резултати анализе брзине преноса података мрежног стека

6. Закључак

У овом раду је реализовано основно решење мрежног система који омогућава пренос произвољног IP саобраћаја између два рачунара користећи Sub-GHz комуникациони медијум, уз реализацију алгоритама за фрагментацију података и поновно слање података при неуспешном слању, сходно ограничењима физичке архитектуре система који се често примењују за комуникацију оваквог типа. Иако комуникација користећи фреквенције у опсегу испод 1 GHz често налази примену код наменских уређаја и IoT система, у последњих неколико година медијум постаје атрактивна опција за независне децентрализоване мреже (нпр. *Meshtastic*) због ниске енергетске потрошње и дужих комуникационих раздаљина у односу на 2.4 GHz и 5 GHz фреквенцијских опсега.

Методологија реализације датог решења је била у фазама, где је прва фаза била упознавање са физичком архитектуром система и начином писања софтвера за EFR32 фамилију микроконтролера, као и успостављање основне бежичне комуникације између два EFR система. Друга фаза је била успостављање основне комуникације између RPi рачунара EFR микроконтролера, и тиме основне комуникације између два RPi рачунара користећи до тад реализовану инфраструктуру. Трећа фаза је била дизајнирање и реализација алгоритама и протокола за фрагментацију података и поновно слање порука, као и за контролу тока података.

Могућа унапређења система су дефинитивно генерализација мреже на више од два чвора, као и побољшање брзине преноса и енергетске потрошње целокупног система. Додатни вид унапређења би био увођење процедуре експлицитног успостављања везе између два уређаја, и енкриптовање података током преноса.

7. Литература

- [1] Frank Durda IV, *Serial and UART Tutorial*, 1996, Доступно: <https://docs.freebsd.org/en/articles/serial-uart/>, (приступљено: 28.06.2026)
- [2] Raspberry Pi, *Configuration, Configure UARTs*, 2026, Доступно: <https://www.raspberrypi.com/documentation/computers/configuration.html#uarts>, (приступљено: 24.06.2026)
- [3] Silicon Labs, *EFR32FG28 Wireless SoC Family Data Sheet, Revision 1.1*, 2023
- [4] Silicon Labs, *AN0948.2: EFM32 and EFR32 Series 2 DC-to-DC Converter, Revision 0.9*, 2025
- [5] Maxim Krasnyansky, Maksim Yevmenkin, *Universal TUN/TAP device driver*, 2001, Доступно: <https://docs.kernel.org/networking/tuntap.html>, (приступљено: 24.06.2026)
- [6] Andrea Goldsmith, *Wireless Communications, 14.1 Multiuser Channels: The Uplink and Downlink*, 2020
- [7] IEEE, *Std 802.11™-2020, Part 11: Wireless LAN Medium Access Control(MAC) and Physical Layer (PHY) Specifications, 10.3.1*, 2020
- [8] ETSI, *Short Range Devices (SRD) operating in the frequency range 25 MHz to 1 000 MHz;Part 2: Harmonised Standard for access to radio spectrum for non specific radio equipment*, 2018
- [9] IEEE, *Std 802.15.4™-2006, Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs), 5.5.3.2 Data Frame*, 2006
- [10] Silicon Labs, *RAIL: Proprietary Radio Configurator Guide, Radio Parameters*, 2026, Доступно: <https://docs.silabs.com/rail/latest/proprietary-radio-configurator-guide/03-radio-parameters#sync-word>, (приступљено: 25.06.2026)

-
- [11] D.D. Falconer, F. Adachi, B. Gudmundson, *Time division multiple access methods for wireless personal communications*, 1995
- [12] Andreana Goldsmith, *Wireless Communications 14.3.3 Carrier-Sense Multiple Access (CSMA)*, 2020
- [13] Bluetooth, *Bluetooth Core Specification, Architecture*, 2025, Доступно: <https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Core-61/out/en/architecture,-change-history,-and-conventions/architecture.html>,
(приступљено: 29.06.2026)
- [14] Silicon Labs, *Tx/Rx Options, Address Filtering*, 2026, Доступно: <https://docs.silabs.com/rail/latest/rail-training-trx-options/#address-filtering>,
(приступљено: 30.06.2026)
- [15] Silicon Labs, *Common Utilities, Code Placement*, 2026, Доступно: <https://docs.silabs.com/gecko-platform/latest/platform-common/code-placement>,
(приступљено: 01.07.2026)
- [16] J. Reynolds, J. Postel, *RFC 1700: Assigned Numbers, Data Notations*, 1994
- [17] J. Postel, *RFC 791, Internet Protocol, 3.1 Internet Header Format*, 1981