УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Невена Мијаиловић

ЕВОЛУЦИЈА КОМУНИКАЦИЈЕ У РЕАЛНОМ ВРЕМЕНУ У ВЕБ АПЛИКАЦИЈАМА: WEBSOCKET ПРОТОКОЛ И УПОРЕДНА АНАЛИЗА ПЕРФОРМАНСИ РАЗЛИЧИТИХ ПРИСТУПА

мастер рад

Београд, 2026.

Ментор:

проф. др Јелена Граовац, ванредни професор
Универзитет у Београду, Математички факултет

Чланови комисије:

проф. др Владимир Филиповић, редовни професор
Универзитет у Београду, Математички факултет

др Милан Банковић, доцент
Универзитет у Београду, Математички факултет

Датум одбране: _____

Наслов мастер рада: Еволуција комуникације у реалном времену у веб апликацијама: *WebSocket* протокол и упоредна анализа перформанси различитих приступа

Резиме: Комуникација у реалном времену представља важан захтев савремених веб апликација у којима се промене података морају проследити корисницима са малим кашњењем. Предмет овог рада су развој приступа комуникацији у реалном времену и упоредна анализа *WebSocket* протокола, *Server-Sent Events* и *long polling* приступа. За потребе поређења реализовано је контролисано окружење за мерење перформанси које се састоји од *ASP.NET Core* сервера, генератора оптерећења и анализатора резултата. Сва три приступа користе исти генератор, садржај и редослед порука, чиме су обезбеђени упоредиви услови мерења. Испитиван је утицај броја истовремених клијената, брзине генерисања порука и величине корисног садржаја. Анализирани су кашњење, проток, удео успешне испоруке, губитак порука, време успостављања комуникације, остварење циљног броја генерисаних порука, процењени протоколски додатак и употреба серверских ресурса, као и додатни показатељи поузданости испоруке и стабилности извршавања. Резултати показују да су при нижем и средњем оптерећењу разлике између посматраних приступа мале. У испитаном једносмерном току *WebSocket* и *Server-Sent Events* показују слично понашање, док се код приступа *long polling* са повећањем учесталости порука раније јављају раст кашњења и губитак испорука. Добијени резултати показују да избор комуникационог приступа зависи од очекиваног оптерећења и захтева за кашњењем и поузданошћу, а не само од могућности испоруке података у реалном времену.

Кључне речи: веб, комуникација у реалном времену, *WebSocket*, *Server-Sent Events*, *long polling*, *ASP.NET Core*, анализа перформанси

Садржај

Глава 1

Увод

Савремене веб апликације све чешће захтевају да промене података буду доступне корисницима непосредно након њиховог настанка. Апликације за размену порука, приказ спортских резултата, праћење финансијских података, заједничко уређивање докумената и онлајн игре представљају примере система у којима традиционални модел, заснован на томе да клијент пошаље захтев и сачека одговор сервера, није увек довољан. У таквим системима сервер мора да буде у могућности да нове податке проследи клијенту без непотребног чекања на наредни класичан захтев.

Приступи комуникацији у реалном времену развијали су се упоредо са развојем веба. Рана решења била су заснована на периодичном слању *HTTP* захтева и дуготрајном задржавању одговора, док су касније стандардизовани механизми који омогућавају трајнији ток података или двосмерну комуникацију. Међу најзначајнијим приступима налазе се *long polling*, *Server-Sent Events* и протокол *WebSocket*. Иако сва три могу да се користе за правовремено достављање података, разликују се по моделу комуникације, начину коришћења везе, протоколском додатку и понашању при већем оптерећењу.

Протокол *WebSocket* омогућава дуготрајну и двосмерну комуникацију између клијента и сервера. Механизам *Server-Sent Events* обезбеђује једносмерни ток података од сервера ка клијенту преко дуготрајног *HTTP* одговора. Код приступа *long polling* сервер задржава клијентов захтев док нови подаци не постану доступни или док не истекне временско ограничење, након чега клијент шаље нови захтев. Избор између ових приступа зато зависи од смера комуникације, броја клијената, учесталости порука и захтева у погледу кашњења и поузданости испоруке.

Циљ овог рада је да прикаже еволуцију комуникације у реалном времену у веб апликацијама, детаљније објасни протокол *WebSocket* и експериментално упореди његове перформансе са *Server-Sent Events* и *long polling* приступима. Поређење је усмерено на питање како се посматрани приступи понашају са повећањем броја истовремених клијената, брзине генерисања порука и величине корисног садржаја.

За потребе експерименталног дела рада реализовано је посебно окружење за мерење перформанси. Оно обухвата сервер развијен помоћу технологије *ASP.NET Core*, генератор оптерећења и анализатор резултата. Сервер реализује сва три комуникациона приступа, док генератор оптерећења симулира већи број клијената и бележи њихове резултате. Анализатор се покреће након завршетка мерења, групише поновљене експерименте и припрема агрегиране податке за анализу. Иако протокол *WebSocket* подржава двосмерну комуникацију, у имплементацији мерног окружења реализован је само смер од сервера ка клијентима, како би сва три приступа могла непосредно да се пореде.

Упоредивост осталих услова обезбеђена је коришћењем истог генератора и истог садржаја порука за сва три приступа. Сценарији се разликују само по комуникационом механизму и контролисаним параметрима. Мерења су више пута поновљена, а поред просечних вредности анализирана је и варијабилност резултата. Посматране метрике обухватају, између осталог, кашњење испоруке, проток, удео успешне испоруке, губитак порука, време успостављања комуникације, процењени протоколски додатак, употребу серверских ресурса.

Допринос рада огледа се у повезивању теоријске анализе посматраних приступа са резултатима добијеним у контролисаном и поновљивом експерименталном окружењу. Циљ мерења није утврђивање универзалног победника, већ уочавање услова под којима разлике између приступа постају значајне и објашњавање узрока таквог понашања. Резултати се зато тумаче заједно са ограничењима конкретне имплементације и хардвера на којем су експерименти извршени.

Рад је организован на следећи начин. Најпре је приказана еволуција комуникације у реалном времену у веб апликацијама. Затим је детаљније приказан WebSocket протокол. Након тога следи упоредна техничка анализа посматраних приступа. У наредном поглављу описани су методологија, архитектура и имплементација мерног окружења, док су потом приказани и размотрени

резултати експеримената. На крају су издвојена ограничења истраживања и изведени завршни закључци.

Глава 2

Еволуција комуникације у реалном времену у веб апликацијама

2.1 Појам комуникације у реалном времену

Развој веба пратио је постепен прелазак од статичких страница, чији се садржај мењао искључиво поновним учитавањем целог документа, ка интерактивним апликацијама које могу готово тренутно да прикажу промене настале на серверу.

Израз „комуникација у реалном времену" у контексту веб апликација углавном не означава строго реално време, у којем систем гарантује да ће одређена операција бити извршена у унапред дефинисаном року. У овом контексту реч је о комуникацији чије је кашњење довољно мало да корисник има утисак да се промена приказује непосредно након њеног настанка.

На пример, када један корисник пошаље поруку у апликацији за дописивање, очекује се да је други корисник види готово тренутно. Слично томе, у систему за приказ берзанских цена промена вредности треба да буде приказана без ручног освежавања странице. У апликацији за заједничко уређивање текста, измена коју направи један корисник треба да буде прослеђена осталим учесницима са минималним кашњењем.

Основни изазов у развоју оваквих система произилази из чињенице да је веб првобитно био заснован на моделу комуникације у којем клијент иницира сваки захтев. Сервер је могао да одговори на примљени захтев, али није

постојао стандардизован начин да самостално пошаље нову информацију веб прегледачу (клијенту). Због тога је развој комуникације у реалном времену подразумевао проналажење начина да се ограничења традиционалног модела протокола *HTTP* превазиђу или заобиђу.

Током развоја веб технологија појавио се већи број приступа. Најпре су коришћени ручно освежавање странице и периодично слање захтева серверу. Након тога развијени су асинхрони *HTTP* захтеви, периодични полинг, *long polling*, *HTTP* стриминг, *Comet* приступи и *Server-Sent Events (SSE)*. На крају је стандардизован протокол *WebSocket*, који је омогућио трајну и двосмерну комуникацију између клијента и сервера.

2.2 Традиционални модел комуникације протокола *HTTP*

Прве веб апликације биле су засноване на једноставном моделу захтева и одговора. Корисник би отворио одређену адресу, након чега би веб прегледач послао *HTTP* захтев серверу. Сервер би обрадио захтев и вратио *HTML* документ, који би прегледач приказао кориснику.

Комуникација се у основи одвијала на следећи начин:

1. корисник покреће одређену акцију;
2. веб прегледач шаље захтев серверу;
3. сервер обрађује примљени захтев;
4. сервер враћа одговор;
5. веб прегледач обрађује и приказује добијени садржај.

Протокол *HTTP* је на апликационом нивоу заснован на размени захтева и одговора. Клијент шаље захтев, а сервер на основу њега формира одговарајући одговор. Иако једна транспортна веза може бити поново употребљена за више размена, свака апликациона интеракција задржава семантику захтева и одговора [3].

Такав модел био је погодан за приказ статичких докумената и страница које се ретко мењају. Међутим, био је непрактичан за апликације у којима се

подаци често ажурирају. Сервер није могао самостално да обавести клијента да се нешто променило. Клијент је морао да пошаље нови захтев како би сазнао да ли постоји нова информација.

На пример, ако би корисник отворио страницу са спортским резултатима, резултат приказан на страници остао би непромењен све док корисник не би поново учитао страницу. Чак и када би се податак на серверу променио, веб прегледач о томе не би имао никакву информацију.

Поред кашњења у приказу података, поновно учитавање целе странице стварало је и додатно мрежно оптерећење. Сервер је често поново слао комплетан *HTML* документ, делове корисничког интерфејса и друге податке, иако се променила само једна вредност.

Традиционални модел протокола *HTTP* није био погрешан, већ је био прилагођен првобитној намени веба, односно преузимању и приказивању докумената. Проблем је настао када је веб почео да се користи као платформа за сложене и интерактивне апликације. Ово ограничење представљало је један од главних разлога за развој техника које су омогућиле асинхроно преузимање података.

2.3 *AJAX* и асинхрона комуникација

Значајан корак у развоју интерактивних веб апликација представљала је примена асинхроних захтева из програмског језика *JavaScript*. Овај приступ постао је познат под називом *AJAX*, односно *Asynchronous JavaScript and XML*.

Термин *AJAX* популаризован је 2005. године у тексту Џесија Џејмса Герета „*Ajax: A New Approach to Web Applications*". У том тексту описан је модел у којем веб страница може да комуницира са сервером у позадини, без поновног учитавања целог документа [4].

Најважнија промена није била у самом мрежном протоколу, јер је комуникација и даље користила *HTTP*, већ у начину на који је клијентска апликација обрађивала одговор. Кôд написан у језику *JavaScript* могао је да пошаље захтев, преузме нове податке и измени само одговарајући део странице.

На пример, у апликацији за електронску пошту клијент више није морао поново да учита комплетну страницу како би приказао нову поруку. Клијент-

ски кôд могао је да преузме само списак нових порука и дода их у постојећи кориснички интерфејс.

Назив *AJAX* садржи назив формата *XML*, али ова техника није ограничена на тај формат. Подаци се могу преносити као *JSON*, *HTML*, обичан текст или бинарни садржај. У савременим апликацијама формат *JSON* користи се знатно чешће од формата *XML*.

Примена приступа *AJAX* омогућила је:

- ажурирање само одређеног дела веб странице;
- комуникацију са сервером без прекида тренутне интеракције;
- смањење количине поновно пренетог садржаја;
- бржу реакцију корисничког интерфејса;
- развој сложенијих једностраничних веб апликација.

Ипак, *AJAX* није решио основни проблем серверски инициране комуникације. Захтев је и даље морао да пошаље клијент. Сервер није могао самостално да обавести веб прегледач да је настао нови догађај.

Због тога су прве апликације које су захтевале често ажурирање података почеле да користе периодично проверавање сервера.

2.4 Периодични полинг

Периодични полинг (енгл. *polling*), представља један од најједноставнијих начина приближног остваривања комуникације у реалном времену. Клијент у унапред дефинисаним временским интервалима шаље *HTTP* захтев серверу и проверава да ли су доступни нови подаци.

На пример, апликација може на сваких пет секунди да пошаље захтев и затражи све поруке које су настале након последње примљене поруке. Ако постоје нови подаци, сервер их враћа. Ако нови подаци не постоје, сервер враћа празан одговор или информацију да није било промена.

Основна предност периодичног полинга јесте једноставност. За његову примену нису потребни посебни протоколи нити трајне везе. Клијент шаље обичне *HTTP* захтеве, а сервер на њих одговара на уобичајен начин.

Такав приступ се лако интегрише са постојећим серверима, механизмима аутентификације и инфраструктуром за евидентирање захтева. Због тога периодични полинг и данас може бити оправдан код једноставних апликација у којима се подаци ретко мењају или у апликацијама где није претерано важно да одмах добијемо најновије податке.

Главни недостатак овог приступа представља однос између кашњења и броја захтева. Ако се захтеви шаљу у кратким интервалима, нови подаци ће бити брже примљени, али ће сервер бити изложен великом броју захтева. Ако се интервал повећа, број захтева ће бити мањи, али ће корисник дуже чекати да види нову информацију.

Уколико клијент шаље захтев на сваких десет секунди, догађај који настане непосредно након једног захтева може остати непримећен скоро десет секунди. Са друге стране, ако клијент шаље захтев сваке секунде, велики број одговора може бити празан.

Овај проблем постаје нарочито изражен када систем има велики број корисника. Ако десет хиљада клијената шаље захтев на сваких пет секунди, сервер у просеку прима око две хиљаде захтева у секунди, чак и када нема нових података.

Поред обраде самог захтева, при свакој размени преносе се *HTTP* заглавља, колачићи, подаци о аутентификацији и други метаподаци. Код малих апликационих порука ти додатни подаци могу бити већи од корисног садржаја. Краћи интервал полинга омогућава брже уочавање нових података, али повећава број захтева и мрежно оптерећење, док дужи интервал смањује овај трошак уз повећање кашњења испоруке [8].

Иако је периодични полинг омогућио аутоматско освежавање података, није био ефикасан за апликације којима су били потребни чести и брзи догађаји. Као одговор на ова ограничења развијен је *long polling*.

2.5 *Long polling*

Long polling, представља прилагођени облик полинга у којем сервер не враћа одговор одмах ако нема нових података. Клијент шаље *HTTP* захтев, а сервер га задржава отвореним све док се не појави нови догађај или док не истекне одређено временско ограничење.

Када нови податак постане доступан, сервер шаље тај податак као одговор клијенту. Након обраде одговора клијент одмах шаље нови захтев, који сервер поново задржава.

Животни циклус комуникације засноване на *long polling* приступу може се описати на следећи начин:

1. клијент шаље захтев серверу;
2. сервер задржава захтев отвореним;
3. на серверу настаје нови догађај;
4. сервер враћа одговор са новим подацима;
5. клијент обрађује примљени одговор;
6. клијент шаље нови захтев.

У поређењу са периодичним полингом, *long polling* смањује број празних одговора. Сервер не мора одмах да врати информацију да нема промена, већ може да сачека настанак догађаја. На тај начин нови подаци могу бити послати готово одмах након што постану доступни [1].

Ипак, овај приступ и даље користи низ одвојених *HTTP* захтева и одговора. После сваке испоруке клијент мора да пошаље нови захтев [11]. Код апликација у којима се догађаји појављују веома често, тај циклус може да се понавља велики број пута у кратком временском периоду.

Поред тога, дуготрајни захтеви могу бити прекинути због временских ограничења на серверу, реверзном проксију (енгл. *reverse proxy*), балансеру оптерећења (енгл. *load balancer*) или другој мрежној компоненти. Због тога клијент мора да буде спреман да понови захтев након прекида или истека времена.

Документ *RFC 6202* описује познате проблеме и препоручене праксе за употребу *long polling* приступа и стриминга у двосмерној комуникацији заснованој на протоколу *HTTP*. У њему се посебно разматрају проблеми мрежних посредника, временских ограничења, броја веза и поновног успостављања комуникације [8].

Long polling је био један од најчешће коришћених механизама за асинхрону испоруку података са сервера ка клијенту. Његова значајна практична пред-

ност била је могућност примене у оквиру постојеће инфраструктуре засноване на протоколу *HTTP* [8].

Ипак, потреба да се након сваког одговора отвори нови захтев довела је до развоја приступа у којима један одговор може да остане отворен и пренесе већи број догађаја.

2.6 *HTTP* стриминг и *Comet* приступи

HTTP стриминг заснива се на идеји да сервер не мора одмах да заврши одговор, већ може да га остави отвореним и постепено шаље нове делове садржаја. Клијент затим обрађује податке док одговор још увек траје.

За разлику од *long polling* приступа, где се одговор најчешће завршава након појаве новог догађаја, код стриминга један одговор може да садржи већи број догађаја. Сервер шаље нове податке кроз исту отворену везу.

Овим приступом смањује се потреба за поновним слањем *HTTP* захтева. Такође се смањује кашњење, јер сервер може да пошаље догађај чим он настане.

Током развоја раних интерактивних веб апликација, за различите технике дуготрајног одржавања *HTTP* веза коришћен је назив *Comet*[1]. *Comet* није представљао један прецизно дефинисан протокол, већ скуп техника које су омогућавале серверу да асинхроно испоручује податке веб прегледачу.

У *Comet* приступе убрајали су се:

- *long polling*;
- стриминг помоћу дуготрајних одговора;
- употреба скривених оквира;
- прогресивно учитавање скрипти;
- различите варијанте такозваног обрнутог приступа *AJAX*.

Главни проблем *HTTP* стриминга било је неуједначено понашање веб прегледача и мрежних посредника. Одређене компоненте могле су да задржавају

[1]Назив *Comet* представља хумористичку игру речи са термином *AJAX* [10]. У литератури се ова игра речи објашњава чињеницом да су *Ajax* и *Comet* називи средстава за чишћење [12, 5].

мање количине података у баферу, уместо да их одмах проследе клијенту. У том случају догађаји би били испоручени тек када се прикупи довољна количина садржаја, чиме би се повећало кашњење.

Прокси сервери и балансери оптерећења такође су могли да прекину везу ако током одређеног периода није било података. Због тога су сервери понекад слали празне коментаре или друге контролне податке како би веза остала активна.

Други проблем представљала је чињеница да је комуникација углавном била једносмерна. Клијент је могао да прима податке кроз отворени ток, али је за слање података серверу морао да користи посебне *HTTP* захтеве.

Упркос ограничењима, *Comet* и *HTTP* стриминг показали су да дуготрајна веза може значајно да унапреди испоруку догађаја. Искуства стечена применом ових техника допринела су развоју стандардизованог механизма серверски послатих догађаја.

2.7 *Server-Sent Events*

Server-Sent Events (SSE) механизам представља стандардизован начин једносмерног слања догађаја са сервера ка клијенту преко дуготрајне *HTTP* везе. У веб прегледачу комуникација се успоставља помоћу интерфејса *EventSource*, док сервер враћа одговор типа `text/event-stream` [14].

Након што клијент отвори везу, сервер може да пошаље већи број догађаја без завршавања *HTTP* одговора. Сваки догађај представљен је текстуалним пољима, као што су:

- подаци догађаја;
- назив типа догађаја;
- идентификатор догађаја;
- препоручено време до следећег покушаја повезивања.

Једна од најважнијих особина механизма *SSE* јесте уграђена подршка за поновно повезивање. Ако се веза прекине, веб прегледач може аутоматски покушати да је поново успостави.

Уколико је сервер уз догађај послао идентификатор, клијент при наредном повезивању може да проследи идентификатор последњег примљеног догађаја. Сервер затим може да настави испоруку од одговарајућег места.

Ова особина је значајна јер поједностављује реализацију система у којима привремени мрежни прекид не сме да доведе до трајног губитка података.

Механизам *SSE* погодан је за апликације у којима информације претежно путују са сервера ка клијенту. Типични примери су:

- системи обавештења;
- приказ вести;
- праћење напретка дуготрајног процеса;
- приказ системских логова;
- спортски резултати;
- берзанске цене;
- праћење стања сервера и уређаја.

Најважније ограничење овог приступа јесте једносмерност. Иста веза користи се за слање података са сервера ка клијенту, али не и за произвољно слање података са клијента ка серверу. Клијент за тај смер мора да користи посебне *HTTP* захтеве.

Поред тога, *SSE* је заснован на текстуалном формату. Бинарни подаци не преносе се непосредно, већ се морају претворити у текстуални облик или преузети посебним захтевом.

Ипак, једносмерност није увек недостатак. У многим системима сервер шаље велики број обавештења, док клијент само повремено шаље команду. У таквим случајевима *SSE* може бити једноставнији од потпуно двосмерног протокола.

Стандардизација овог механизма представљала је важан корак у односу на различите нестандардизоване *Comet* технике. Уместо ослањања на специфично понашање прегледача и скривене механизме, апликације су добиле јасно дефинисан клијентски интерфејс и формат догађаја.

Међутим, апликације као што су системи за размену порука, онлајн игре и заједничко уређивање докумената захтевале су интензивну комуникацију у

оба смера. За такве потребе било је неопходно увести протокол који омогућава да и клијент и сервер независно шаљу податке кроз исту везу.

2.8 *WebSocket*

Претходно описани приступи омогућили су веб апликацијама да добијају нове податке без ручног освежавања странице, али су задржали одређена ограничења комуникације засноване на протоколу *HTTP*. *Long polling* је захтевао успостављање новог захтева након сваког примљеног одговора, док су *HTTP* стриминг и *SSE* пре свега били усмерени на пренос података са сервера ка клијенту.

Развој апликација за размену порука, онлајн игара, заједничко уређивање садржаја и других интерактивних система створио је потребу за комуникационим механизмом у којем обе стране могу независно да шаљу податке преко исте трајне везе.

Као одговор на ову потребу развијен је *WebSocket* протокол. Његова основна карактеристика је успостављање трајног двосмерног комуникационог канала између клијента и сервера. Након успостављања везе, сервер може да пошаље податке клијенту чим се појави нови догађај, без чекања на нови клијентски захтев [2].

Појава *WebSocket* протокола представља значајан корак у еволуцији комуникације у реалном времену, јер је уведен стандардизован механизам за пуну двосмерну комуникацију прилагођен веб окружењу. Настанак, стандардизација и начин функционисања овог протокола детаљније су представљени у поглављу 3.

Глава 3

WebSocket протокол

3.1 Настанак и стандардизација протокола *WebSocket*

Развој интерактивних веб апликација довео је до потребе за комуникационим механизмом који би омогућио да клијент и сервер независно шаљу податке кроз исту трајну везу, без покретања новог *HTTP* захтева за сваку размену. Као одговор на ову потребу развијен је протокол *WebSocket*.

Један од раних формалних предлога протокола објавио је Ијан Хиксон (*Ian Hickson*) у оквиру *IETF Internet-Draft* документа почетком 2009. године. Нацрт под називом *The Web Socket Protocol* описивао је протокол за двосмерну комуникацију између корисничког агента, као што је веб прегледач, и удаљеног сервера [6].

Даљи развој и стандардизација одвијали су се у оквиру радне групе *HyBi* (*BiDirectional or Server-Initiated HTTP*) организације *IETF*. Као полазна основа за рад групе коришћен је рани нацрт протокола који је развијао Хиксон. Циљ радне групе био је дефинисање општег механизма за ефикасну двосмерну комуникацију који би могао да функционише у постојећој веб инфраструктури [7, 6].

У раду групе учествовао је већи број стручњака и представника различитих организација. Председавајући групе били су Габријел Монтенегро (*Gabriel Montenegro*) и Салваторе Лорето (*Salvatore Loreto*), док су коначни документ који дефинише основни протокол написали Ијан Фет (*Ian Fette*) и Алексеј Мељников (*Alexey Melnikov*) [7, 2].

Коначна спецификација објављена је у децембру 2011. године као *RFC 6455 – The WebSocket Protocol*. Овим документом *WebSocket* је стандардизован као протокол за двосмерну комуникацију између клијента и сервера. Протокол се састоји од почетног поступка успостављања везе, након којег следи размена порука у облику *WebSocket* оквира преко трајне транспортне везе [2].

Поред самог протокола *WebSocket*, веб прегледачи обезбеђују и програмски интерфејс (*WebSocket API*) који омогућава веб апликацијама да успоставе везу, шаљу и примају поруке и обрађују њено затварање. Овај интерфејс дефинисан је у оквиру стандарда организације *WHATWG* [13].

На тај начин програмер не мора непосредно да управља детаљима протокола, као што су формирање *WebSocket* оквира или поступак почетног успостављања везе. Док документ *RFC 6455* дефинише правила комуникације између клијента и сервера на нивоу протокола, *WebSocket API* представља начин на који клијентска веб апликација користи ту комуникацију.

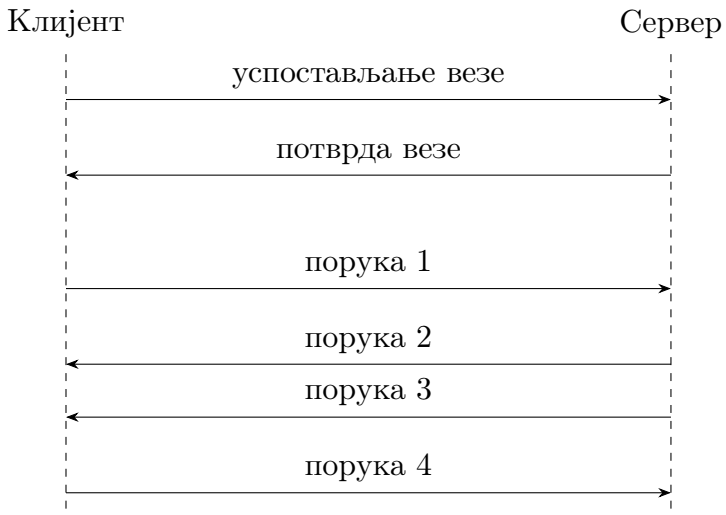
3.2 Мотивација и основне карактеристике

Основни циљ протокола *WebSocket* јесте да обезбеди двосмерну комуникацију између клијента и сервера преко једне дуготрајне везе. У документу *RFC 6455* као један од основних мотива наводи се потреба да веб апликације остваре двосмерну комуникацију без ослањања на више паралелних *HTTP* веза и технике као што је *long polling* [2].

Код традиционалне комуникације засноване на протоколу *HTTP*, комуникацију започиње клијент. Клијент шаље захтев, сервер га обрађује и затим враћа одговор. Ако сервер касније добије нови податак који треба проследити клијенту, неопходно је да постоји нови клијентски захтев или неки од механизама описаних у претходном поглављу.

Протокол *WebSocket* мења овај начин комуникације. Клијент иницијално успоставља везу, али након њеног успешног успостављања и клијент и сервер могу независно да шаљу поруке.

Таква комуникација назива се комуникацијом у пуном дуплексу (енгл. *full-duplex*). То значи да пренос података може да се одвија у оба смера преко исте везе.



Слика 3.1: Успостављање *WebSocket* везе и размена порука између клијента и сервера

Након успостављања везе више не постоји обавезан однос у којем једна порука мора представљати захтев, а друга одговор. Сервер може послати поруку клијенту зато што се на серверу догодио одређени догађај, чак и ако клијент непосредно пре тога није послао никакав захтев.

На пример, у апликацији за размену порука клијент не мора непрекидно да пита сервер да ли је стигла нова порука. Када сервер прими поруку намењену одређеном кориснику, може је одмах проследити кроз већ успостављену *WebSocket* везу.

Сличан модел може се користити код приказа промена цена, спортских резултата, стања уређаја, онлајн игара и апликација за заједничко уређивање садржаја. Примена протокола *WebSocket* у системима за комуникацију и приказ података у реалном времену разматрана је и у научној литератури [9].

Још једна важна карактеристика јесте трајност везе. Након почетног успостављања, иста веза може се користити за пренос већег броја порука. Није неопходно извршавати комплетан поступак успостављања комуникације за сваку нову апликациону поруку.

Протокол подржава два основна типа апликационих података:

- текстуалне поруке;
- бинарне поруке.

Текстуалне поруке морају представљати исправно кодиран *UTF-8* текст, док бинарне поруке могу садржати произвољан низ бајтова [2].

Сам протокол не одређује формат апликационих података. Веб апликација, на пример, може да користи *JSON*:

```
{
    "type": "message",
    "user": "marko",
    "text": "Zdravo"
}
```

За *WebSocket* је овај садржај само текстуална порука. Значење поља `type`, `user` и `text` дефинише сама апликација.

На тај начин *WebSocket* представља основу за комуникацију, али не дефинише конкретан апликациони протокол. Различите апликације могу користити потпуно различите формате и правила размене порука преко истог основног протокола.

3.3 Успостављање *WebSocket* везе

Иако *WebSocket* након успостављања везе користи сопствени формат порука, почетак комуникације је пројектован тако да се уклопи у постојећу веб инфраструктуру. У основном случају дефинисаном документом *RFC 6455*, клијент започиње комуникацију слањем посебног *HTTP/1.1* захтева [2].

Овај поступак назива се *opening handshake*. Његова сврха је да клијент и сервер потврде да желе да успоставе *WebSocket* комуникацију.

Поједностављени захтев клијента може да изгледа овако:

```
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

У овом захтеву посебно су значајна заглавља `Upgrade` и `Connection`. Вредност `Upgrade: websocket` означава да клијент тражи прелазак са комуникације засноване на протоколу *HTTP* на протокол *WebSocket*.

Поље `Sec-WebSocket-Version: 13` означава верзију протокола дефинисану документом *RFC 6455* [2].

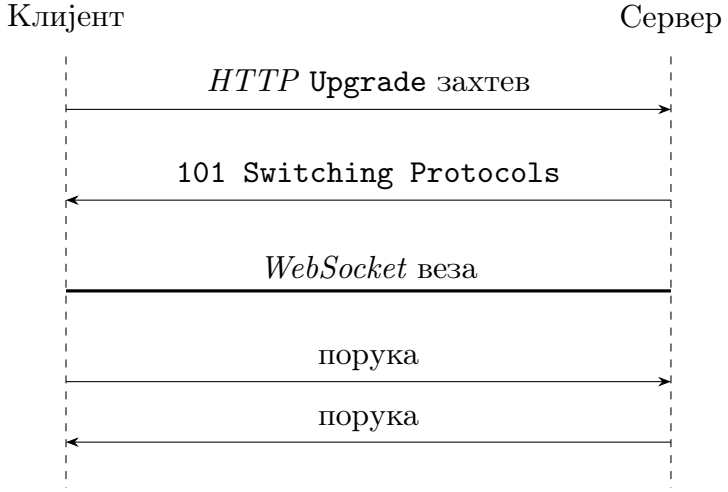
Поље `Sec-WebSocket-Key` садржи насумично генерисану вредност коју сервер користи приликом формирања одговора. Она помаже клијенту да провери да ли је сервер заиста разумео захтев за успостављање *WebSocket* везе.

Ако сервер прихвати захтев, враћа одговор:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Статусни кôд `101 Switching Protocols` означава да је сервер прихватио промену протокола [2].

Након успешног завршетка овог поступка иста транспортна веза остаје отворена, али се подаци више не размењују као класични *HTTP* захтеви и одговори. Уместо тога, обе стране размењују *WebSocket* поруке.



Слика 3.2: Успостављање *WebSocket* везе путем *HTTP Upgrade* механизма

Важно је напоменути да се протокол *HTTP* овде користи за почетно успостављање комуникације. Након успешног *handshake*-а свака наредна порука није нови *HTTP* захтев.

Управо је ово једна од суштинских разлика у односу на *long polling*. Код *long polling* приступа након завршеног одговора клијент мора да пошаље нови захтев, док *WebSocket* задржава већ успостављен комуникациони канал.

3.4 Размена података

Након успешног успостављања везе почиње размена података. На мрежном нивоу подаци се преносе у облику *WebSocket* оквира [2].

Оквир садржи заглавље и корисни садржај. Заглавље је представљено у бинарном формату и садржи информације које омогућавају другој страни да одреди тип поруке, њену дужину и друге особине потребне за правилну обраду.

За основно разумевање протокола најважније је да *WebSocket* разликује:

- текстуалне оквире;
- бинарне оквире;
- контролне оквире.

Текстуални и бинарни оквири преносе податке апликације. Контролни оквири користе се за управљање самом везом.

Протокол дефинише контролне поруке `Ping` и `Pong`. Једна страна може да пошаље `Ping`, а друга страна одговара поруком `Pong`. Овај механизам може да се користи за проверу да ли је друга страна и даље доступна [2].

Постоји и контролни оквир `Close`, који служи за уредно затварање везе. На тај начин клијент и сервер могу да сигнализирају да желе да заврше комуникацију пре затварања основне транспортне везе.

Једна од карактеристика протокола јесте и маскирање порука које клијент шаље серверу. Према документу *RFC 6455*, оквири које клијент шаље серверу морају бити маскирани, док серверски оквири нису маскирани [2].

Маскирање не представља шифровање. Кључ потребан за враћање оригиналног садржаја налази се у самом оквиру. Његова сврха повезана је са заштитом мрежних посредника од одређених начина злоупотребе протокола.

За заштиту поверљивости података користи се *TLS* (*Transport Layer Security*) протокол. Због тога постоје две основне шеме адреса:

```
ws://
wss://
```

Шема `ws` означава незаштићену *WebSocket* комуникацију, док `wss` означава комуникацију заштићену протоколом *TLS* [2].

Однос између њих сличан је односу између *HTTP* и *HTTPS* комуникације. У практичним веб апликацијама које се користе преко Интернета примењује се заштићена варијанта `wss`.

3.5 Пример *WebSocket* комуникације

Начин рада протокола најједноставније се може приказати на примеру апликације за размену порука.

Приликом отварања странице клијентски кôд може да креира везу на следећи начин:

```
const socket = new WebSocket("wss://example.com/chat");
```

Позивањем конструктора *WebSocket* веб прегледач покреће поступак успостављања везе. Апликација не мора ручно да формира заглавља `Upgrade`, `Sec-WebSocket-Key` и остала протоколска поља. Тај део поступка обавља имплементација у веб прегледачу.

Када је веза успешно успостављена, активира се догађај `open`:

```
socket.addEventListener("open", () => {
    console.log("Veza je uspostavljena");
});
```

Након тога клијент може да шаље податке серверу. Претпоставимо да су корисници Ана и Марко пријављени у апликацију и да обоје имају отворену веб страницу и успостављену WebSocket везу са сервером. Ако Ана пошаље поруку Марку, апликација може да формира објекат:

```
{
    "type": "message",
    "to": "marko",
    "text": "Zdravo"
}
```

и затим га пошаље кроз већ успостављену везу:

```
socket.send(JSON.stringify({
    type: "message",
    to: "marko",
    text: "Zdravo"
```

```
}));
```

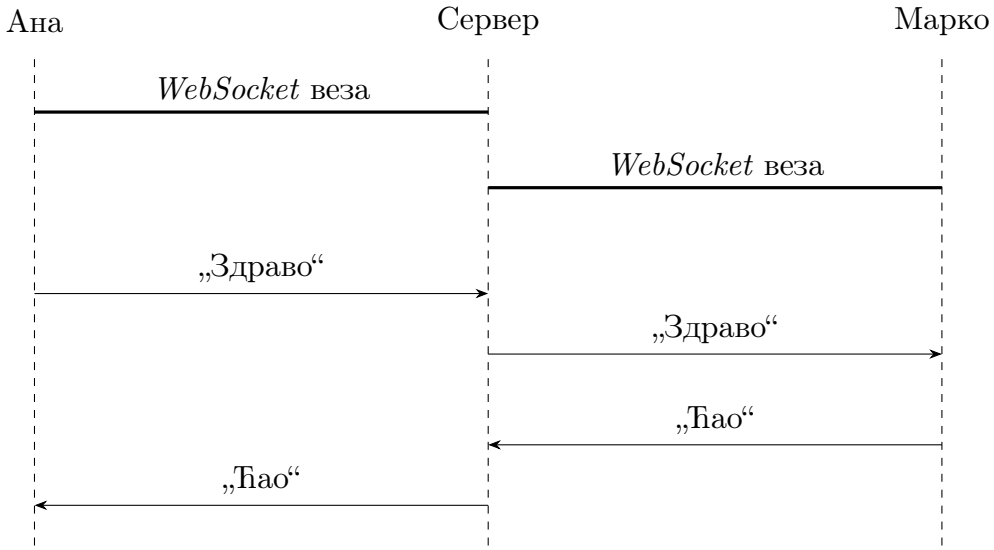
Сервер прима поруку, обрађује њен садржај и утврђује да је намењена Марку. Ако Марко има активну *WebSocket* везу, сервер може одмах да пошаље поруку кроз ту везу.

Марков клијент не мора претходно да пошаље питање серверу да ли постоји нова порука. Сервер сам иницира слање кроз већ отворени канал.

На Марковој страни порука се може обрадити преко догађаја `message`:

```
socket.addEventListener("message", event => {
    const message = JSON.parse(event.data);
    console.log(message);
});
```

Комплетан ток комуникације може се видети на слици 3.3.



Слика 3.3: Пример *WebSocket* комуникације у апликацији за размену порука

Овај пример показује основну предност модела. Након што су везе успостављене, сервер и клијенти могу слати поруке кад год је то потребно. Не постоји потреба за периодичним слањем захтева само ради провере да ли се појавио нови податак.

Исти принцип може да се примени и на друге типове апликација. У систему за приказ спортских резултата сервер може послати нови резултат свим заинтересованим клијентима. У апликацији за заједничко уређивање документа измена једног корисника може се одмах проследити осталим корисни-

цима. У онлајн игри клијент и сервер могу често размењивати информације о променама стања.

3.6 Затварање и прекид везе

WebSocket веза не мора да остане отворена током целог рада сервера. Када клијент или сервер више немају потребу за комуникацијом, веза може бити уредно затворена.

Протокол за то дефинише контролни оквир `Close`. Страна која жели да прекине комуникацију шаље `Close` оквир другој страни. Друга страна затим одговара сопственим `Close` оквиром, након чега се основна транспортна веза затвара [2].

На клијентској страни веза се може затворити методом:

```
socket.close();
```

Клијент такође може да обради догађај затварања:

```
socket.addEventListener("close", event => {
    console.log("Veza je zatvorena");
});
```

Поред уредног затварања, могућ је и неочекивани прекид комуникације, на пример услед губитка мрежне везе, престанка рада сервера или проблема у некој од посредних мрежних компоненти.

Сам *WebSocket API* не обезбеђује аутоматско поновно успостављање прекинуте везе. Ако апликација жели да након прекида настави комуникацију, потребно је да сама реализује механизам поновног повезивања.

Слично томе, протокол не дефинише на који начин треба надокнадити поруке које су настале док клијент није био повезан. Такво понашање припада апликационом слоју.

Због тога је важно разликовати транспорт који пружа *WebSocket* од додатне логике система. Протокол омогућава размену порука док веза постоји, али апликација одређује шта поруке значе, како се чувају и шта се дешава након прекида комуникације.

Глава 4

Упоредна техничка анализа приступа

У претходним поглављима представљен је развој приступа за комуникацију у реалном времену, као и детаљније карактеристике *WebSocket* протокола. Иако *long polling*, *Server-Sent Events* (*SSE*) и *WebSocket* протокол могу да се користе за испоруку података кориснику са малим кашњењем, њихов начин рада се значајно разликује.

Због тога избор одговарајућег приступа не зависи само од тога да ли апликација захтева комуникацију у реалном времену, већ и од смера и учесталости размене података, броја клијената, количине пренетих података и захтева саме апликације.

У овом поглављу три приступа биће упоређена према заједничким техничким критеријумима. Циљ није да се унапред прогласи један приступ најбољим, већ да се уоче особине које могу имати утицај на резултате мерења у експерименталном делу рада.

4.1 Модел комуникације и животни циклус везе

Једна од најважнијих разлика између посматраних приступа односи се на начин успостављања и одржавања комуникације.

Код *long polling* приступа клијент шаље *HTTP* захтев који сервер задржава отвореним док се не појави нови податак или док не истекне временско

ограничење. Након што сервер пошаље одговор, та размена се завршава и клијент мора да пошаље нови захтев како би наставио да прима наредне догађаје [8]. Због тога се комуникација састоји од низа узастопних *HTTP* захтева и одговора.

Механизам *SSE* такође користи протокол *HTTP*, али се веза понаша другачије. Клијент успоставља један дуготрајни захтев, а сервер кроз исти одговор може континуирано да шаље већи број догађаја. Веза остаје активна све док је једна страна не затвори или док не дође до прекида [14].

За разлику од претходна два приступа, *WebSocket* протокол након почетног успостављања прелази на сопствени модел размене порука. Клијент и сервер затим користе исту трајну везу и могу независно да шаљу податке [2].

Разлика је посебно значајна у погледу смера комуникације. *SSE* је намењен преносу догађаја са сервера ка клијенту. Ако клијент треба да пошаље податке серверу, за то се користи засебан *HTTP* захтев. Код *long polling* приступа клијент такође користи посебне захтеве за слање података серверу.

Протокол *WebSocket*, са друге стране, омогућава пуну двосмерну комуникацију преко исте везе. Ова особина је нарочито значајна код апликација у којима обе стране често генеришу нове податке.

Због тога се већ на основу модела комуникације могу издвојити различите области примене. *Long polling* може бити довољан када су догађаји релативно ретки и када је пожељно остати у оквиру класичног *HTTP* модела. *SSE* природно одговара апликацијама код којих већина података путује са сервера ка клијенту, док је *WebSocket* погоднији када је потребна честа комуникација у оба смера.

4.2 Протоколски трошак и количина мрежног саобраћаја

Поред самих апликационих података, сваки приступ преноси и одређену количину додатних протоколских информација. Њихов удео може бити нарочито значајан када апликација размењује велики број малих порука.

Код *long polling* приступа сваки нови циклус комуникације подразумева нови *HTTP* захтев и одговор. Након што сервер пошаље одговор, клијент типично шаље нови захтев како би наставио да чека наредни догађај [8]. Ако се догађаји појављују често, овај циклус се може понављати велики број пута.

Овај трошак је мање изражен када су догађаји ретки, јер један захтев може дуго остати отворен. Међутим, ако сервер непрекидно генерише нове догађаје, сваки примљени одговор доводи до покретања новог циклуса.

Код механизма *SSE*, након почетног *HTTP* захтева, више догађаја може да буде пренето кроз исти ток. Није потребно поново слати комплетан захтев за сваки нови догађај. Сваки догађај, међутим, има текстуалну структуру дефинисану форматом `text/event-stream`, која такође представља одређени додатни трошак [14].

Протокол *WebSocket* након почетног успостављања комуникације користи сопствене оквире. Свака порука садржи заглавље оквира, али се за сваку нову поруку не шаљу комплетна *HTTP* заглавља. Код порука које клијент шаље серверу присутан је и кључ за маскирање, у складу са правилима протокола [2].

Због ових разлика може се очекивати да ће се у системима са великим бројем малих и честих порука удео протоколских података значајније разликовати него у системима у којима се ретко шаљу велике поруке.

Управо због тога у практичном поређењу није довољно мерити само број успешно пренетих порука. Корисно је посматрати и укупну количину пренетих података, односно однос корисног садржаја и додатног протоколског саобраћаја.

4.3 Кашњење и учесталост размене података

За апликације које захтевају комуникацију у реалном времену једна од најважнијих особина је време које протекне од настанка догађаја на серверу до његовог пријема на клијенту.

Код *long polling* приступа сервер може да одговори непосредно након настанка догађаја ако клијент већ има активан захтев. На тај начин се избегава кашњење карактеристично за периодични полинг. Ипак, након сваког одговора постоји потреба за успостављањем новог захтева, што уводи додатни корак у животни циклус комуникације [8].

Код механизма *SSE* активна веза остаје отворена и сервер може да пошаље нови догађај чим он постане доступан. Слично томе, код протокола *WebSocket* сервер може непосредно да пошаље поруку кроз већ успостављену везу.

Због тога оба приступа имају архитектонску предност у ситуацијама у којима сервер генерише велики број догађаја у кратким интервалима, јер није потребно успостављати нови апликациони циклус за сваки догађај.

Код двосмерне комуникације разлика је израженија. У случају механизма *SSE*, подаци са клијента ка серверу захтевају додатне *HTTP* захтеве. Код протокола *WebSocket* клијент може да пошаље поруку непосредно преко постојеће везе.

Међутим, само на основу начина рада протокола није могуће прецизно одредити стварно кашњење. На њу утичу и мрежни услови, имплементација клијента и сервера, обрада порука, број активних корисника, величина поруке и други фактори. Због тога стварне разлике треба утврдити мерењем у контролисаном окружењу.

4.4 Коришћење серверских ресурса и скалабилност

Начин успостављања и одржавања комуникације непосредно утиче на ресурсе које сервер користи за комуникацију са повезаним клијентима. Код посматраних приступа разликују се број и трајање активних захтева, начин поновног успостављања комуникације и количина обраде која се извршава приликом размене података.

Код *long polling* приступа сервер задржава примљени *HTTP* захтев отвореним све док се не појави нови догађај или док не истекне временско ограничење. Након што сервер пошаље одговор, клијент типично шаље нови захтев како би наставио да чека следећи догађај [8]. Због тога велики број клијената подразумева истовремено одржавање великог броја активних захтева, али и поновљену обраду нових захтева након сваког примљеног одговора.

Код механизма *SSE* клијент такође успоставља *HTTP* везу са сервером, али се кроз исти дуготрајни одговор може пренети већи број догађаја. Након што је веза успостављена, није потребно отварати нови захтев за сваки наредни догађај [14]. Сервер, међутим, мора да одржава активну везу за сваког повезаног клијента.

Слично томе, WebSocket протокол користи дуготрајну везу која након почетног успостављања може да се користи за размену већег броја порука у

оба смера [2]. Сервер зато мора да одржава активну везу и одређено стање за сваког повезаног клијента током трајања комуникације.

На основу самог начина рада протокола није могуће закључити који приступ ће увек користити најмање процесорског времена или меморије. На потрошњу ресурса утичу и конкретна имплементација сервера, број активних клијената, учесталост догађаја, величина порука и количина података који се обрађују.

У контексту овог рада, скалабилност се односи на способност система да задржи прихватљиве перформансе са повећањем броја истовремено активних клијената и обима комуникације. Са растом броја клијената расте и број комуникационих веза или активних захтева које сервер мора да одржава, док већа учесталост догађаја повећава број операција које сервер мора да обради у јединици времена.

Скалабилност зависи од конкретног оптерећења и начина имплементације система, а за њену процену посебно су значајни следећи параметри:

- број истовремено повезаних клијената;
- учесталост генерисања и слања догађаја;
- величина порука;
- број обрађених захтева или порука у јединици времена;
- потрошња процесорског времена;
- потрошња меморије;
- количина пренетих података;
- промена кашњења са повећањем оптерећења.

4.5 Прекид комуникације и поновно повезивање

Дуготрајне везе могу бити прекинуте услед проблема у мрежи, временских ограничења посредних компоненти, престанка рада сервера или других разлога. Због тога је начин поновног успостављања комуникације важна разлика између посматраних приступа.

Код *long polling* приступа поновно слање захтева већ представља саставни део начина рада. Ако захтев истекне или се веза прекине, клијент мора да пошаље нови захтев. Апликација притом мора да води рачуна о томе који су догађаји већ примљени како не би дошло до губитка или дуплирања података [8].

Механизам *SSE* има уграђену подршку за поновно успостављање везе. Интерфејс *EventSource* након прекида може аутоматски да покуша поновно повезивање. Стандард дефинише и идентификатор последњег примљеног догађаја, који може да помогне наставку испоруке [14].

Код протокола *WebSocket* аутоматско поновно успостављање везе није део основног програмског интерфејса. Ако апликација жели да настави комуникацију након прекида, потребно је да сама дефинише начин поновног повезивања и, ако је потребно, механизам за надокнаду пропуштених порука.

Због тога *SSE* може бити једноставнији у системима у којима је потребно само континуирано примање серверских догађаја. *WebSocket* даје већу слободу у дефинисању комуникације, али део те флексибилности подразумева и већу одговорност апликације.

4.6 Сложеност имплементације и избор приступа

Са становишта имплементације, *long polling* има предност у томе што се заснива на стандардном моделу *HTTP* захтева и одговора и може се релативно једноставно уклопити у постојећу веб инфраструктуру [8]. За апликације са малом учесталошћу догађаја ова једноставност може бити значајнија од потенцијалних уштеда које нуде дуготрајне везе.

Механизам *SSE* је такође заснован на протоколу *HTTP*, а клијентски интерфејс *EventSource* обезбеђује релативно једноставан начин отварања везе и пријема догађаја. Због једносмерног модела посебно је погодан за обавештења, надзор стања, приказ резултата и друге системе у којима сервер најчешће шаље податке клијенту.

WebSocket протокол обезбеђује најопштији комуникациони модел од три посматрана приступа. Омогућава двосмерну размену текстуалних и бинарних порука преко исте трајне везе [2]. Та флексибилност је корисна за системе

за размену порука, интерактивне апликације и друге случајеве у којима обе стране често шаљу податке.

Ипак, употреба *WebSocket* протокола може захтевати додатну логику за поновно повезивање, управљање стањем клијента и обраду прекида. Избор технологије зато не треба заснивати искључиво на томе који приступ теоријски има најмањи протоколски трошак, већ на захтевима конкретне апликације.

Основне разлике између посматраних приступа приказане су у табели 4.1.

Критеријум	**long polling**	**SSE**	**WebSocket**
Смер комуникације	Сервер ка клијенту кроз задржани захтев; клијент користи засебне *HTTP* захтеве	Једносмерно, сервер ка клијенту; клијент користи засебне *HTTP* захтеве	Пуна двосмерна комуникација
Трајање комуникације	Захтев се завршава након одговора и затим се поново успоставља	Дуготрајни *HTTP* одговор	Трајна *WebSocket* веза
Протоколски трошак	Поновљени *HTTP* захтеви и одговори	Један дуготрајни *HTTP* ток	Почетно успостављање, затим *WebSocket* оквири
Бинарни подаци	Подржани преко *HTTP*-а	Нису непосредно подржани	Подржани
Поновно повезивање	Реализује апликација поновним захтевом	Уграђена подршка	Реализује апликација
Типична примена	Ређа ажурирања и једноставни системи	Обавештења и серверски догађаји	Честа двосмерна комуникација

Табела 4.1: Поређење основних техничких карактеристика комуникационих приступа *long polling*, *Server-Sent Events* и *WebSocket*

На основу претходне анализе могу се формулисати очекивања која ће бити

проверена експериментално. Са повећањем учесталости порука очекује се већи додатни саобраћај код *long polling* приступа због поновљених *HTTP* захтева, док је *SSE* погоднији за једносмерну испоруку већег броја серверских догађаја, а *WebSocket* за честу двосмерну комуникацију преко трајне везе.

Ова очекивања биће проверена у наредном поглављу кроз мерно окружење (енгл. *benchmark*) у којем ће сва три приступа бити упоређена.

Глава 5

Методологија и имплементација мерног окружења

5.1 Циљ истраживања

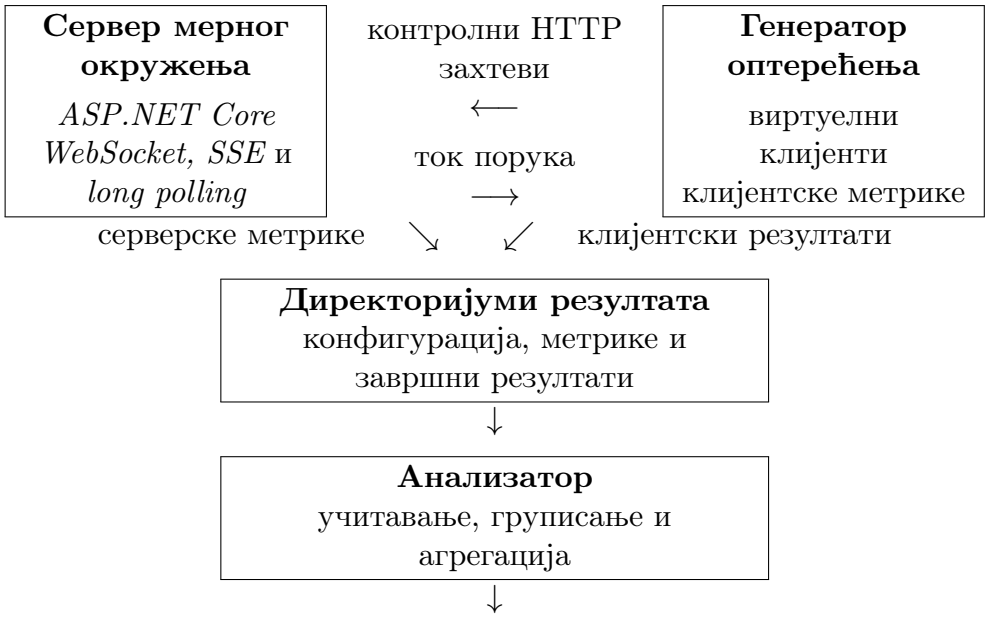
Практични део рада заснован је на контролисаном мерном окружењу намењеном упоредној анализи три приступа за испоруку порука од сервера ка клијентима у реалном времену: *WebSocket* протокола, *Server-Sent Events* приступа и *long polling* приступа. Основни циљ окружења је да омогући поређење наведених механизама у истим условима оптерећења, тако да уочене разлике у резултатима потичу од карактеристика самог механизма комуникације. Посматрају се, између осталог, кашњење испоруке, проток, поузданост испоруке, време успостављања везе, употреба процесора и меморије, као и процењени протоколски додатак.

За сва три приступа сервер користи заједнички генератор порука и исти формат поруке. На тај начин *WebSocket, SSE* и *long polling* клијенти примају исти логички ток порука, при истој учесталости генерисања и са истом величином корисног садржаја. Ова одлука је кључна за валидност поређења, јер спречава да разлике у резултатима буду последица различитог оптерећења које се шаље појединим протоколима.

Практични резултат истраживања је мерно окружење које програмски симулира већи број клијената, покреће поновљиве сценарије, прикупља метрике током извршавања и чува резултате у *JSON* и *CSV* датотекама.

5.2 Архитектура система

Систем је реализован у програмском окружењу *.NET 8* и састоји се од три одвојене апликације: *ASP.NET Core* сервера мерног окружења, конзолног генератора оптерећења и конзолног анализатора. Сервер реализује сва три комуникациона приступа и производи један заједнички ток порука. Генератор оптерећења програмски симулира клијенте и прикупља метрике пријема. Анализатор се покреће након експеримената и обрађује сачуване резултате без додатног оптерећења система који се мери.



Слика 5.1: Логичка архитектура мерног окружења и ток података

Све компоненте су написане у *C#*, чиме се избегава увођење различитих окружења за извршавање и олакшава изградња и репродукција. Библиотека *SignalR* се не користи јер би њен механизам преговарања, сопствени формат порука и додатни механизми за управљање транспортом утицали на резултате мерења.

5.3 Организација решења

Репозиторијум је организован као једно *.NET* решење са одвојеним извршним пројектима и тестовима:

```
server/              ASP.NET Core сервер мерног окружења
load-generator/      конзолна апликација за симулацију клијената
analyzer/            конзолна апликација за анализу резултата
tests/               аутоматизовани тестови
results/             резултати извршавања експеримената
BenchmarkRealtimeProtocols.sln
```

Унутар пројеката кôд је подељен према одговорностима. Сервер има моделе, сервисе, протоколске крајње тачке (енгл. *endpoint*), контролне руте и помоћне класе. Генератор оптерећења има слој за аргументе командне линије, три клијентске имплементације, компоненте за прикупљање и израчунавање метрика током извршавања и сервис који управља целокупним током мерења. Анализатор раздваја читање, агрегацију и форматирање излаза.

5.4 Сервер мерног окружења

Сервер мерног окружења представља *ASP.NET Core* апликацију која реализује сва три посматрана механизма комуникације. Сервер прихвата клијентске везе, покреће и зауставља појединачна мерења, генерише заједнички ток порука и доставља га повезаним клијентима преко одговарајућег комуникационог механизма. За управљање једним мерењем сервер излаже контролне *HTTP* крајње тачке, док су засебне крајње тачке намењене *WebSocket, SSE* и *long polling* комуникацији.

Модел поруке и серверски генератор порука

Да би поређење комуникационих приступа било фер, све поруке се формирају на једном месту и имају исти формат, независно од протокола којим ће бити испоручене. Модел поруке садржи јединствени идентификатор, временску ознаку тренутка генерисања поруке на серверу и корисни садржај. Идентификатор поруке омогућава да се на страни клијента утврде изгубљене, дуплиране и поруке примљене ван очекиваног редоследа, док временска ознака служи за израчунавање кашњења.

Сва три протокола добијају објекте истог облика:

```
{
 "id": 12345,
 "created_at": 1717000000123,
 "payload": "AAAA..."
}
```

Корисни садржај је детерминистички низ знакова `A`. Генерише се задате величине, при чему се за сваки експеримент свим протоколима шаље садржај исте дужине. На тај начин величина поруке представља контролисани параметар експеримента. У оквиру сваког извршавања поруке добијају узастопне идентификаторе, почевши од вредности 1, што омогућава откривање недостајућих, дуплираних и порука примљених ван редоследа, а време генерисања је *UTC Unix* време у милисекундама. Компресија се не користи, јер би различит степен компресије детерминистичког садржаја могао да прикрије стварни трошак протокола.

Генерисање порука заснива се на задатој учесталости. Уместо ослањања на један периодични тајмер, систем у сваком тренутку израчунава колико је порука требало да буде генерисано у односу на протекло време. Када је број већ генерисаних порука у складу са планираном учесталошћу, генератор се кратко успављује, чиме се избегава активно чекање и непотребно оптерећење процесора. Ако је генератор привремено успорен, формира ограничену групу порука како би достигао планирану учесталост. Ограничење величине групе спречава да краткотрајно кашњење доведе до прекомерног оптерећења сервера.

Кôд 5.1: Основни део алгоритма за генерисање порука

```
var target = (long)Math.Floor(
    sw.Elapsed.TotalSeconds * config.MessageRatePerSecond);

if (target <= generated)
{
    await Task.Delay(
        c.MessageRatePerSecond >= 100
            ? 1
            : (int)(500 / c.MessageRatePerSecond),
    token);
    continue;
}
```

```
var batch = Math.Min(target - generated, 1000);
for (var i = 0; i < batch && generated < max; i++)
{
    generated++;
    state.Publish(
        state.CreateMessage(generated, config.PayloadSizeBytes));
}
```

Генерисање се завршава када истекне задато трајање експеримента или када се достигне задати укупан број порука. Ако су постављена оба ограничења, генерисање се завршава када се прво од њих достигне, или када је мерење експлицитно отказано.

Управљање стањем мерења и објављивање порука

Класа `BenchmarkState` представља централно стање активног мерења. Исти објекат користе контролне крајње тачке, протоколске крајње тачке, генератор порука и компонента за прикупљање серверских метрика. Он садржи конфигурацију активног мерења и серверске бројаче, као и одвојене структуре података које користе комуникациони приступи. Клијенти протокола *WebSocket* и механизма *Server-Sent Events* чувају се у засебним колекцијама, док приступ *long polling* користи заједнички бафер генерисаних порука.

При покретању мерења проверава се конфигурација, празни се бафер и бројачи се постављају на почетне вредности. Затим се креира објекат `CancellationTokenSource`, чији се токен прослеђује генератору порука и компоненти за прикупљање серверских метрика. При заустављању мерења активира се сигнал за прекид, након чега обе компоненте контролисано завршавају рад.

Операције покретања и заустављања заштићене су закључавањем објекта `_gate`. Тиме се обезбеђује да провера постојања активног мерења и промена његовог стања не могу истовремено да се извршавају у више захтева, па у једном тренутку може бити активно највише једно мерење.

Свака генерисана порука најпре се додаје у бафер који користи приступ *long polling*. Затим се ажурира број генерисаних порука, а порука се прослеђује редовима повезаних *WebSocket* и *Server-Sent Events* клијената, као што је приказано у кôду 5.2.

Кôд 5.2: Објављивање заједничке поруке

```
public void Publish(BenchmarkMessage message)
{
    Buffer.Append(message);
    Interlocked.Exchange(ref MessagesGenerated, message.Id);

    PublishTo(WebSockets, message);
    PublishTo(SseClients, message);
}
```

При објављивању поруке, приступ *long polling* користи заједнички бафер ограниченог капацитета, док сваки повезани *WebSocket* и *Server-Sent Events* клијент има засебан ограничени ред порука. Метод приказан у кôду 5.3 покушава да упише поруку у ред сваког клијента без блокирања генератора.

Кôд 5.3: Неблокирајуће објављивање и евидентирање повратног притиска

```
private void PublishTo(
    ConcurrentDictionary<Guid, Channel<BenchmarkMessage>> clients,
    BenchmarkMessage message)
{
    foreach (var channel in clients.Values)
    {
        if (!channel.Writer.TryWrite(message))
        {
            Interlocked.Increment(ref BackpressureEvents);
        }
    }
}
```

Ако је ред неког клијента попуњен, метод `TryWrite` враћа вредност `false`, порука се не уписује у тај ред и бележи се догађај повратног притиска (енгл. *backpressure event*). Исти серверски бројач увећава се и када слање поруке клијенту траје дуже од 100 милисекунди. Због тога његова вредност представља показатељ оптерећења, а не тачан број изгубљених испорука. Стварни губитак накнадно се утврђује за сваког клијента анализом идентификатора примљених порука.

Серверска имплементација комуникационих приступа

У овој имплементацији сва три приступа користе се само за слање порука од сервера ка клијентима, како би били поређени под истим условима.

Због тога двосмерна комуникација коју подржава протокол *WebSocket* није коришћена.

WebSocket приступ

Сервер излаже крајњу тачку `/ws`. Након успешног успостављања везе сервер прихвата *WebSocket* везу и додељује јој јединствени идентификатор. За сваког клијента формира се засебан ограничени канал у којем на слање може да чека највише 4096 порука. Поруке се из канала преузимају редом и шаљу клијенту преко успостављене везе.

Одвојени канал за сваког клијента раздваја генерисање порука од њиховог слања преко мреже. Генератор покушава да упише нову поруку у канале свих повезаних клијената без блокирања целокупног система. Уколико је канал неког клијента попуњен, упис не успева и бележи се догађај повратног притиска. На тај начин спор клијент не зауставља генерисање порука за остале клијенте.

Сервер асинхроно чита поруке из канала, серијализује их у *JSON* формат и шаље као *WebSocket* текстуалне поруке. Додатно, као показатељ повратног притиска бележи се свако слање поруке које траје дуже од 100 милисекунди.

Кôд 5.4: Слање порука *WebSocket* клијенту

```
await foreach (var message in
    channel.Reader.ReadAllAsync(context.RequestAborted))
{
    var bytes = JsonSerializer.SerializeToUtf8Bytes(message);
    var before = Environment.TickCount64;

    await socket.SendAsync(bytes, WebSocketMessageType.Text, true,
        context.RequestAborted);

    if (Environment.TickCount64 - before > 100)
        Interlocked.Increment(ref state.BackpressureEvents);
}
```

По прекиду везе клијент се уклања из заједничког стања, док се евентуалне грешке приликом слања евидентирају у серверским метрикама.

Server-Sent Events приступ

Сервер излаже крајњу тачку `/sse` и поставља тип садржаја `text/event-stream`. Кеширање и посредничко бафероване су искључени како би се догађаји прослеђивали клијенту без непотребног задржавања. Након започињања *HTTP* одговора шаље се почетни *SSE* коментар, који клијенту потврђује да је веза успостављена.

Сваком *SSE* клијенту додељују се јединствени идентификатор и засебан ограничени канал капацитета 4096 порука. На тај начин *WebSocket* и *SSE* користе исти механизам интерног прослеђивања порука. Уколико је канал неког клијента попуњен, бележи се догађај повратног притиска, док се испорука осталим клијентима наставља.

Сервер асинхроно чита поруке из канала и серијализује их у *JSON* формат. Свака порука се затим обликује као *SSE* догађај који садржи идентификатор, тип догађаја и податке. Празан ред означава крај једног догађаја. Након уписа позива се операција `FlushAsync`, како порука не би остала задржана у баферу *HTTP* одговора. Ако уписивање и прослеђивање *SSE* догађаја траје дуже од 100 милисекунди, бележи се догађај повратног притиска.

Кôд 5.5: Формирање и слање *SSE* догађаја

```
var json = JsonSerializer.Serialize(message);
var sseEvent = $"id: {message.Id}\n" + $"event: message\n"
    + $"data: {json}\n\n";

var before = Environment.TickCount64;

await context.Response.WriteAsync(sseEvent, context.RequestAborted);
await context.Response.Body.FlushAsync(context.RequestAborted);

if (Environment.TickCount64 - before > 100)
    Interlocked.Increment(ref state.BackpressureEvents);
```

Long polling приступ

Сервер излаже крајњу тачку `/lp`. Сваки захтев садржи идентификатор последње поруке коју је клијент примио, максимално време чекања и највећи број порука који може бити враћен у једном одговору.

Поруке се чувају у ограниченом ротирајућем баферу који је заједнички

свим *long polling* клијентима. Његов капацитет износи 4096 порука. Читање порука не уклања их из бафера, већ се при попуњавању капацитета додавањем нових порука уклањају најстарије. За сваки захтев издвајају се поруке чији је идентификатор већи од вредности параметра `lastId`.

По пријему захтева сервер проверава да ли су после вредности параметра `lastId` већ доступне нове поруке. Ако јесу, одмах их враћа у одговору. У супротном, захтев остаје отворен до додавања нове поруке, истека задатог времена чекања или прекида захтева од стране клијента. Након завршетка чекања сервер поново издваја доступне поруке и формира одговор. Ако истекне задато време чекања, операција чекања се отказује и повећава се бројач истеклих *long polling* захтева.

Кôд 5.6: Обрада *long polling* захтева

```
var initial = state.Buffer.ReadAfterAndWatch(lastId, maxBatch ?? 100);

var result = (initial.Messages, initial.Truncated);

if (result.Messages.Count == 0 && (timeoutMs ?? 30000) > 0)
{
    using var timeout = CancellationTokenSource
        .CreateLinkedTokenSource(context.RequestAborted);

    timeout.CancelAfter(timeoutMs ?? 30000, 1, 120000);

    try
    {
        await initial.ChangeTask.WaitAsync(timeout.Token);
    }
    catch (OperationCanceledException)
        when (!ctx.RequestAborted.IsCancellationRequested)
    {
        Interlocked.Increment(ref state.LongPollTimeouts);
    }

    result = state.Buffer.ReadAfter(lastId, maxBatch ?? 100);
}
```

Метод `ReadAfterAndWatch` атомски обавља две операције: читање тренутно доступних порука и преузимање сигнала за следећу промену бафера. Ако би се ове операције извршавале одвојено, нова порука би могла да стигне између читања и почетка чекања. Тада би сервер пропустио сигнал о ње-

ном додавању, па би захтев непотребно остао отворен до истека временског ограничења.

Одговор садржи доступне поруке и вредност `Truncated`. Ова вредност показује да ли су неке поруке које клијент још није преузео већ уклоњене из ротирајућег бафера. Сервер додатно евидентира укупан број захтева, захтеве који тренутно чекају, истекла чекања, празне одговоре и одговоре код којих је дошло до губитка старијег дела бафера.

Прикупљање серверских метрика

Током активног мерења сервер једном у секунди бележи податке о употреби ресурса. Употреба процесора израчунава се на основу промене укупног процесорског времена серверског процеса између два узастопна мерења:

$$C = \frac{\Delta t_{\mathrm{CPU}}}{\Delta t_{\mathrm{elapsed}} \cdot N_{\mathrm{processors}}} \cdot 100,$$

где је Δt_{CPU} процесорско време које је серверски процес потрошио између два узорка, $\Delta t_{\mathrm{elapsed}}$ стварно протекло време, а $N_{\mathrm{processors}}$ број доступних логичких процесора. На овај начин добија се удео укупног процесорског капацитета који користи серверски процес.

Радни скуп процеса (енгл. *working set*) очитава се преко својства `WorkingSet64`. Он представља количину физичке меморије (*RAM*) коју серверски процес користи у тренутку мерења.

Поред ових показатеља, бележе се величина управљане динамичке меморије (енгл. *managed heap*), укупан број алоцираних бајтова, број извршавања сакупљача отпада и број нити.

Уз системске метрике бележе се број активних *WebSocket* и *Server-Sent Events* клијената, број *long polling* захтева који чекају одговор, број генерисаних порука и број догађаја повратног притиска.

Узорци се записују у формату *JSONL*, при чему сваки ред представља стање сервера у једном тренутку. Овакав запис омогућава накнадно израчунавање просечних и максималних вредности, као и праћење промене ресурса током трајања експеримента.

5.5 Генератор оптерећења

Генератор оптерећења реализован је као засебна *.NET* конзолна апликација. Његов задатак је да симулира задати број клијената, управља током експеримента и прикупља податке о примљеним порукама. Конфигурација једног мерења обухвата комуникациони приступ, број клијената, величину корисног садржаја, брзина генерисања порука и ограничење трајања или укупног броја порука.

Клијентске имплементације

WebSocket клијент

Сваки виртуелни *WebSocket* клијент користи засебан објекат класе `ClientWebSocket`. Веза се успоставља пре почетка генерисања порука, при чему се мери време потребно за њено успостављање. Ако је порука примљена у више делова, клијент их обједињује до ознаке `EndOfMessage`, након чега десеријализује *JSON* садржај. За сваку примљену поруку бележе се идентификатор клијента, број примљених бајтова и локална временска ознака пријема.

Server-Sent Events клијент

Сваки виртуелни *SSE* клијент успоставља *HTTP* везу коришћењем опције `ResponseHeadersRead`. Ова опција омогућава да се веза сматра успостављеном након пријема заглавља, без чекања на завршетак одговора, који током *SSE* комуникације остаје отворен. Клијент чита *SSE* ток ред по ред, а из поља `data` издваја *JSON* садржај и десеријализује га у заједнички модел поруке. Уз локалну временску ознаку пријема бележи се величина целог *SSE* догађаја.

Long polling клијент

Long polling клијент најпре шаље захтев са временом чекања једнаким нули. Тај захтев служи за проверу доступности крајње тачке и мерење времена успостављања комуникације. Током експеримента клијент у сваком новом захтеву шаље идентификатор последње примљене поруке, чита групу порука из одговора и затим одмах шаље следећи захтев.

За сваки одговор клијент бележи његову величину, број послатих захтева и број празних одговора. Ако одговор садржи више порука, његова величина се равномерно распоређује на све поруке у групи ради процене трошка по поруци. Примљене поруке се затим прослеђују истом механизму за израчунавање кашњења, пропусности и поузданости испоруке који користе *WebSocket* и *SSE* клијенти. Кључни серверски механизам читања и чекања приказан је у кôду 5.6.

Оркестрација тока експеримента

Сва три типа клијента имплементирају заједнички интерфејс `IBenchmarkClient`, који дефинише операције неопходне за њихово покретање и заустављање, као и приступ основним резултатима извршавања. Захваљујући томе, главни део генератора оптерећења може на исти начин да покреће, надгледа и зауставља све клијенте, независно од конкретног комуникационог приступа. Метрике специфичне за поједине приступе прикупљају се у одговарајућим клијентским имплементацијама.

Пре почетка мерења генератор проверава доступност сервера преко крајње тачке `/health`. Након тога, коришћењем крајње тачке `/time`, процењује се разлика између часовника генератора оптерећења и сервера. Ова корекција је потребна јер се временска ознака генерисања поруке бележи на серверу, док се време њеног пријема бележи у генератору оптерећења.

Затим се формира задати број клијената и за сваког од њих се позива метод `ConnectAsync`. Серверско генерисање порука не започиње док сви клијенти не буду припремљени за пријем порука. Након припреме клијената примењује се кратак период загревања, а затим се серверу шаље конфигурација и захтев за почетак експеримента. Успешност тог захтева се проверава пре наставка извршавања.

Кôд 5.7: Припрема клијената и покретање експеримента

```
var clients = Enumerable.Range(1, options.Clients)
    .Select(CreateClient)
    .ToArray();

await Task.WhenAll(
    clients.Select(client => client.ConnectAsync(setupCts.Token)));

if (options.WarmupSeconds > 0)
```

```
    {
        await Task.Delay(
            TimeSpan.FromSeconds(options.WarmupSeconds),
            setupCts.Token);
    }

    var config = new
    {
        runId = options.RunId,
        payloadSizeBytes = options.PayloadSize,
        messageRatePerSecond = options.Rate,
        durationSeconds = options.Duration,
        totalMessages = options.TotalMessages
    };

    using var start = await _http.PostAsJsonAsync(
        options.ServerUrl.TrimEnd('/') + "/control/start",
        config);

    start.EnsureSuccessStatusCode();
```

Након покретања експеримента сви клијенти паралелно примају поруке. За сваку примљену поруку бележе се њен идентификатор, време пријема, величина и клијент коме је испоручена. За сваког клијента се независно прате недостајуће, дуплиране и поруке примљене ван очекиваног редоследа.

Експеримент се извршава до истека задатог трајања или достизања укупног броја генерисаних порука. Ако су задата оба ограничења, мерење се завршава када прво од њих буде достигнуто. Генератор затим шаље захтев `/control/stop`, преузима завршне серверске статистике и оставља кратак период за обраду већ послатих порука. Након тога се клијентске операције заустављају, везе затварају и израчунавају се резултати експеримента.

Клијентске метрике

Генератор оптерећења прикупља клијентске метрике о броју примљених порука, кашњењу, пропусности, поузданости испоруке, времену успостављања комуникације и количини пренетих података.

Кашњење и синхронизација времена

Кашњење поруке одређује се као разлика између времена њеног пријема код клијента и временске ознаке њеног генерисања на серверу. Пошто ова два времена могу потицати са различитих рачунара, пре мерења се процењује разлика њихових часовника.

За сваки захтев ка крајњој тачки `/time` бележе се време слања захтева, време одговора сервера и време пријема одговора. Разлика часовника процењује се изразом

$$\Delta = t_{\text{server}} - \frac{t_{\text{before}} + t_{\text{after}}}{2}.$$

Да би се умањио утицај мрежних осцилација, за израчунавање коначне корекције користи се половина узорака са најмањим временом повратног пута. Кориговано кашњење поруке рачуна се као

$$L = t_{\text{received}} - t_{\text{sent}} + \Delta.$$

За добијене вредности кашњења порука израчунавају се просек, медијана, минимум, максимум, као и 95. и 99. перцентил. Перцентил се одређује након сортирања измерених кашњења од најмањег ка највећем. Вредност p95 представља границу испод које се налази приближно 95% измерених кашњења, док се преостали део расподеле односи на испоруке са већим кашњењем. Слично томе, вредност p99 обухвата 99% испорука, док само 1% испорука има веће кашњење.

На пример, ако је примљено 100 порука, p95 приближно одговара кашњењу 95. поруке у сортираном низу. То не представља просек 95% најбржих порука, већ највећу вредност унутар тог дела расподеле. Перцентили су значајни јер показују понашање споријег дела испорука, које просечна вредност може да прикрије.

Пропусност и поузданост испоруке

Пропусност представља број забележених испорука у јединици времена:

$$T = \frac{N_{\text{received}}}{t}.$$

Пошто се свака серверска порука шаље свим клијентима, теоријски број испорука једнак је производу броја генерисаних порука и броја клијената:

$$N_{\text{theoretical}} = N_{\text{generated}} \cdot N_{\text{clients}}.$$

Однос успешно испоручених порука израчунава се на основу броја јединствених порука које су клијенти примили:

$$R_{\text{delivery}} = \frac{N_{\text{unique}}}{N_{\text{theoretical}}}.$$

Недостајуће, дуплиране и поруке примљене ван редоследа откривају се анализом растућих идентификатора. Основна логика класе `LossTracker` приказана је у кôду 5.8.

Кôд 5.8: Евидентирање губитка и касног доласка поруке

```
if (id > _accountedThrough)
{
    AddMissingRange(_accountedThrough + 1, id - 1);
    _accountedThrough = id;
    if (_first == 0) _first = id;
    _last = id;
    return;
}

if (RemoveMissing(id))
{
    if (_first == 0) _first = id;
    if (id < _last) _outOfOrder++;
    _last = Math.Max(_last, id);
    return;
}

_duplicates++;
```

Недостајући идентификатори не чувају се појединачно, већ као сортирани интервали. Када стигне идентификатор већи од свих до тада обрађених, празнина испред њега додаје се као нови интервал. Ако касније стигне порука чији је идентификатор у неком од тих интервала, метод `RemoveMissing` уклања тај идентификатор, по потреби дели интервал и умањује број недостајућих порука. Порука се истовремено бележи као примљена ван редоследа. Ако идентификатор није ни нов ни означен као недостајући, реч је о дупликату. По завршетку мерења метод `Complete` додаје и поруке између последњег обрађеног идентификатора и коначног броја порука које је сервер генерисао.

Време успостављања комуникације

За сваког виртуелног клијента мери се време потребно да комуникациони механизам постане спреман за пријем порука. На основу појединачних вредности израчунавају се просечно време и 95. перцентил времена успостављања комуникације.

Протоколски трошак

Количина података прати се на нивоу корисног садржаја и процењене величине поруке са протоколским подацима. Додатни трошак протокола представља број додатних бајтова и добија се као разлика између процењене укупне количине података и величине корисног садржаја:

$$B_{\text{overhead}} = B_{\text{protocol}} - B_{\text{payload}}.$$

Релативни протоколски трошак показује колики је удео ових додатних бајтова у односу на корисни садржај и рачуна се као:

$$R_{\text{overhead}} = \frac{B_{\text{overhead}}}{B_{\text{payload}}}.$$

За *WebSocket* се у процену укључује величина оквира, за *SSE* комплетан текстуални догађај, док се код *long polling* приступа сабирају величине тела *HTTP* одговора и процењени трошак заглавља сваког захтева. Ове вредности представљају процену трошка на апликационом нивоу, а не егзактну количину саобраћаја на *TCP* или физичком мрежном нивоу.

Чување резултата мерења

Резултати сваког експеримента чувају се у засебном директоријуму чији назив одговара јединственом идентификатору мерења. На тај начин се спречава мешање података из различитих извршавања и омогућава њихова накнадна анализа.

Директоријум једног завршеног мерења може да садржи следеће датотеке:

`config.json` конфигурација генератора оптерећења;

`clock_sync.json` резултати синхронизације часовника;

`server_config.json` конфигурација коју је прихватио сервер;

`server_resources.jsonl` периодични узорци серверских ресурса;

`server_final_stats.json` завршни серверски бројачи;

`client_metrics.json` метрике појединачних клијената;

`final_summary.json` збирни резултати мерења у *JSON* формату;

`final_summary.csv` збирни резултати у *CSV* формату.

JSON формат користи се за структуриране конфигурације и резултате, док је *CSV* погодан за даљу статистичку обраду и формирање графикона. Серверски узорци и необавезни записи о појединачним порукама чувају се у *JSONL* формату. Овај формат омогућава постепено записивање података током експеримента.

5.6 Анализатор резултата

Анализатор је засебна конзолна апликација која се извршава након завршетка експеримента. Његова улога је учитавање резултата завршених мерења, груписање упоредивих извршавања и припрема агрегираних података за табеле и графиконе.

Учитавање и агрегација мерења

Анализатор проналази директоријуме који садрже датотеку `final_summary.json` и из њих учитава конфигурацију и завршне резултате мерења. Мерење се прескаче ако потребни подаци недостају или се не могу исправно учитати. Из датотеке са серверским узорцима додатно се израчунавају просечна и највећа употреба процесора и меморије за свако појединачно извршавање.

Поновљена мерења групишу се само ако имају исти комуникациони приступ, број клијената, величину корисног садржаја, учесталост генерисања, трајање и ограничење укупног броја порука.

Кôд 5.9: Груписање поновљених мерења

```
runs.GroupBy(run => new AggregateKey(
    run.Protocol,
    run.Clients,
    run.PayloadSizeBytes,
```

```
        run.MessageRatePerSecond,
        run.DurationSeconds,
        run.TotalMessages))
    .Select(CreateAggregate);
```

За сваку групу израчунавају се просечне вредности главних метрика и њихова стандардна девијација. Користи се узорачка стандардна девијација:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}.$$

Стандардна девијација се израчунава само када група садржи најмање два поновљена мерења. Резултати појединачних извршавања се задржавају, што омогућава проверу агрегираних вредности и уочавање одступајућих мерења.

Излаз анализатора

Анализатор формира збирне *JSON* и *CSV* датотеке, као и посебне *CSV* датотеке прилагођене цртању графикона. Оне садрже податке за поређење кашњења, пропусности, односа испоруке, губитка порука, употребе процесора и меморије и протоколског трошка при различитом броју клијената, брзини генерисања порука и величини порука. Излазни слој анализатора чине компоненте за упис *JSON* и *CSV* датотека, компонента за припрему *CSV* података за графиконе и модели који дефинишу структуру њихових редова.

Главни излази анализатора су:

- `run_summaries.json` и `run_summaries.csv`, са по једним објектом, односно редом, за свако појединачно мерење;
- `analysis_summary.json`, који садржи метаподатке анализе и агрегиране резултате;
- `protocol_aggregates.csv`, са статистикама група поновљених мерења;
- директоријум `chart-data`, са засебним *CSV* датотекама за кашњење, пропусност, испоруку, губитак порука, ресурсе и протоколски трошак.

Глава 6

Резултати и дискусија

У овом поглављу представљени су резултати поређења приступа *WebSocket*, *Server-Sent Events* и *long polling*. Анализа је организована према три независно мењана параметра: броју клијената, брзини генерисања порука и величини корисног садржаја. Посебно су разматрани кашњење, проток, удео успешне испоруке и губитак порука, употреба серверских ресурса и процењени додатак у односу на корисни садржај.

6.1 Експериментално окружење

Главни скуп резултата обухвата 240 успешних покретања: 16 различитих сценарија, три комуникациона приступа и пет понављања сваке комбинације. Након главног скупа спроведено је још 45 покретања за проширене сценарије са 1.000, 2.000 и 5.000 клијената, као и 45 додатних покретања ради испитивања понашања при већим брзинама генерисања од 4.000, 5.000 и 10.000 порука у секунди. Анализирани резултати тако обухватају укупно 330 успешних покретања.

Активна фаза генерисања порука у сваком покретању трајала је 30 секунди. Повезивање клијената, синхронизација часовника и припремна фаза извршавани су пре почетка тог интервала. Након престанка генерисања остављен је период од пет секунди за пријем порука које су већ биле послате или су чекале на обраду, после чега су формирани коначни резултати мерења.

За сваког повезаног *WebSocket* и *Server-Sent Events* клијента коришћен је засебан ред капацитета 4.096 порука. Приступ *long polling* користио је заједнички бафер истог капацитета, док је један одговор могао да садржи највише

100 порука. Ова ограничења изабрана су као практичан компромис између апсорбовања краткотрајних застоја и спречавања неконтролисаног раста употребе меморије и величине одговора. Она нису својства самих комуникационих приступа, већ фиксни параметри посматране имплементације, који су остали непромењени у свим експерименталним сценаријима.

Сви експерименти извршени су у контролисаном *Docker* окружењу на истом физичком рачунару. Сервер и генератор оптерећења покретани су на том рачунару, чиме је обезбеђено да сва три комуникациона приступа буду испитана под истим хардверским и софтверским условима.

Експерименти су извршени на рачунару чија је конфигурација приказана табели 6.1.

Табела 6.1: Конфигурација рачунара коришћеног за експерименте

Компонента	Спецификација
Процесор	Intel Core Ultra 7 255H
Меморија	64 GB RAM
Оперативни систем	Microsoft Windows 11 Business
Docker	29.6.1
.NET	8.0

У целокупном скупу експеримената нису забележене клијентске грешке, неочекивани прекиди веза, дупликати нити поруке примљене ван редоследа. Сва покретања садрже узорке серверских ресурса. Најмање остварење циљног броја генерисаних порука износило је приближно 99,93%, што показује да су сва три приступа била изложена упоредивом генерисаном оптерећењу. Пад удела успешно испоручених порука забележен је при већем броју клијената и при највећој испитаној брзини генерисања, а ови резултати анализирани су у одговарајућим одељцима.

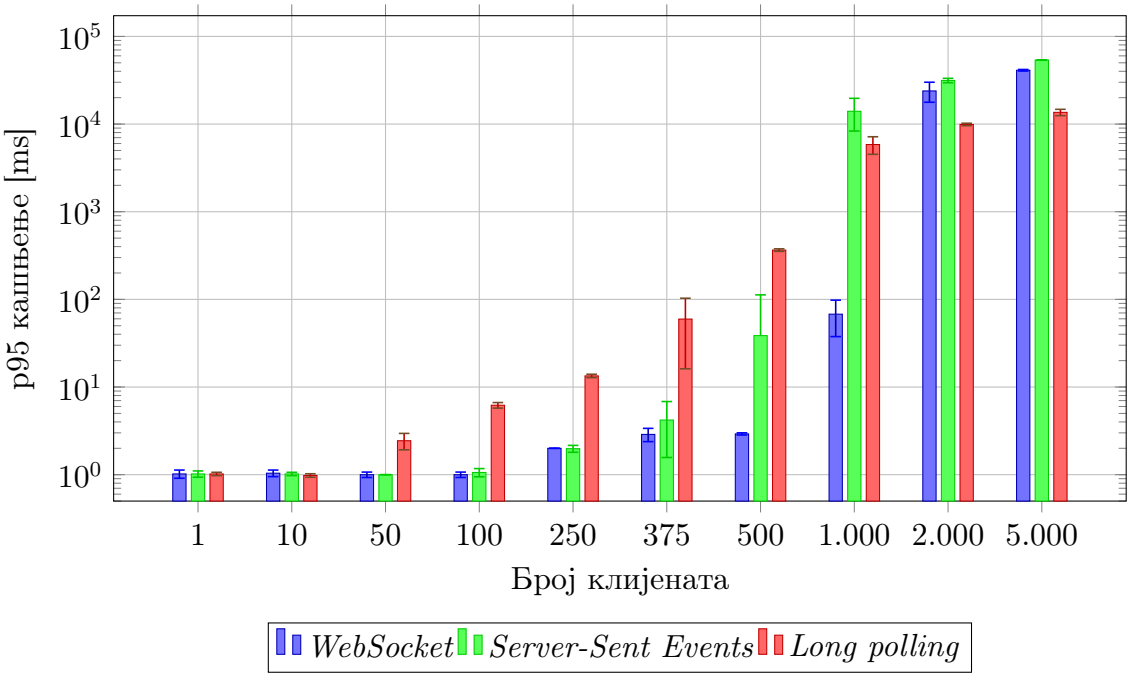
Приказане вредности представљају просек пет поновљених мерења, чиме је умањен утицај појединачних одступања и омогућено упоредиво посматрање резултата различитих комуникационих приступа.

6.2 Утицај броја клијената

У овом делу експеримента величина корисног садржаја износила је 1.024 бајта, док је сервер генерисао 500 порука у секунди. Испитани су сценарији са 1, 10, 50, 100, 250, 375, 500, 1.000, 2.000 и 5.000 истовремених клијената.

Сценарији са више од 500 клијената додати су ради испитивања понашања комуникационих приступа при већем оптерећењу и уочавања области у којој долази до израженијег пада перформанси.

Кашњење



Слика 6.1: Поређење p95 кашњења у зависности од броја клијената. Вертикалне линије представљају стандардну девијацију између поновљених мерења.

При малом броју клијената разлике између посматраних приступа биле су мале. За једног клијента p95 кашњење износило је приближно 1,02 ms код сва три приступа. Са повећањем броја клијената *WebSocket* је задржао ниско и стабилно кашњење, које је при 500 клијената износило 2,92 ms. Код механизма *Server-Sent Events* кашњење је остало релативно ниско до 375 клијената, али је при 500 клијената просечна вредност порасла на 38,60 ms. Овај раст последица је једног одступајућег понављања и указује на већу варијабилност резултата у том сценарију.

Код приступа *long polling* раст кашњења био је израженији већ у овом опсегу. Вредност p95 кашњења износила је 2,44 ms при 50 клијената, 13,40 ms при 250, 59,62 ms при 375 и 365,24 ms при 500 клијената. Последња вредност била је приближно 125 пута већа од одговарајуће вредности за *WebSocket*.

Ипак, до 500 клијената није забележен пад удела испоручених порука, што показује да се деградација приступа *long polling* у овом опсегу најпре испољила кроз повећање кашњења.

При 1.000 клијената дошло је до израженије промене понашања. *WebSocket* је задржао релативно ниско p95 кашњење од 67,82 ms и потпуну испоруку порука. Код приступа *long polling* p95 кашњење порасло је на 5,84 s, а код механизма *Server-Sent Events* на 14,00 s. У ова два случаја забележен је и пад удела испоручених порука. Резултати показују да је у посматраном окружењу *WebSocket* боље поднео повећање броја клијената са 500 на 1.000.

При 2.000 и 5.000 клијената сва три приступа показала су знаке преоптерећења. Вредности p95 кашњења кретале су се од више секунди до више десетина секунди, уз истовремени пад удела испоручених порука. Најниже p95 кашњење у овим сценаријима измерено је код приступа *long polling*, али се тај резултат не може тумачити као показатељ бољег укупног понашања. Кашњење се израчунава само за примљене поруке, док поруке које нису испоручене немају измерену вредност кашњења. Просечан удео испоруке код овог приступа износио је 59,55% при 2.000 и 22,94% при 5.000 клијената. Због тога резултате кашњења при великом оптерећењу треба посматрати заједно са показатељима поузданости испоруке.

Резултати протокола *WebSocket* и механизма *Server-Sent Events* били су слични до 375 клијената. При 500 клијената код механизма *Server-Sent Events* уочена је већа варијабилност кашњења, док је при 1.000 клијената *WebSocket* задржао знатно ниже кашњење и већи удео успешно испоручених порука.

Проток и поузданост испоруке

Сва три приступа остварила су потпуну испоруку у сценаријима са највише 500 клијената. При 500 клијената теоријски проток износио је приближно 250.000 испорука у секунди. Измерени проток износио је приближно 249.977 испорука у секунди за *WebSocket* и *long polling*, односно 249.980 за *Server-Sent Events*. То показује да су сва три приступа у овом опсегу могла да испрате задато оптерећење.

При 1.000 клијената *WebSocket* је и даље остварио потпуну испоруку. Просечан удео испоруке код приступа *long polling* износио је 98,96%, док је код механизма *Server-Sent Events* износио 95,45%. При 2.000 и 5.000 клијената удео успешно испоручених порука опао је код сва три приступа. *WebSocket* је

у оба сценарија задржао највећи удео испоруке, затим *Server-Sent Events*, док је најизраженији пад забележен код приступа *long polling*. Детаљне вредности приказане су у табели 6.2.

Табела 6.2: Удео успешно испоручених порука у зависности од броја клијената

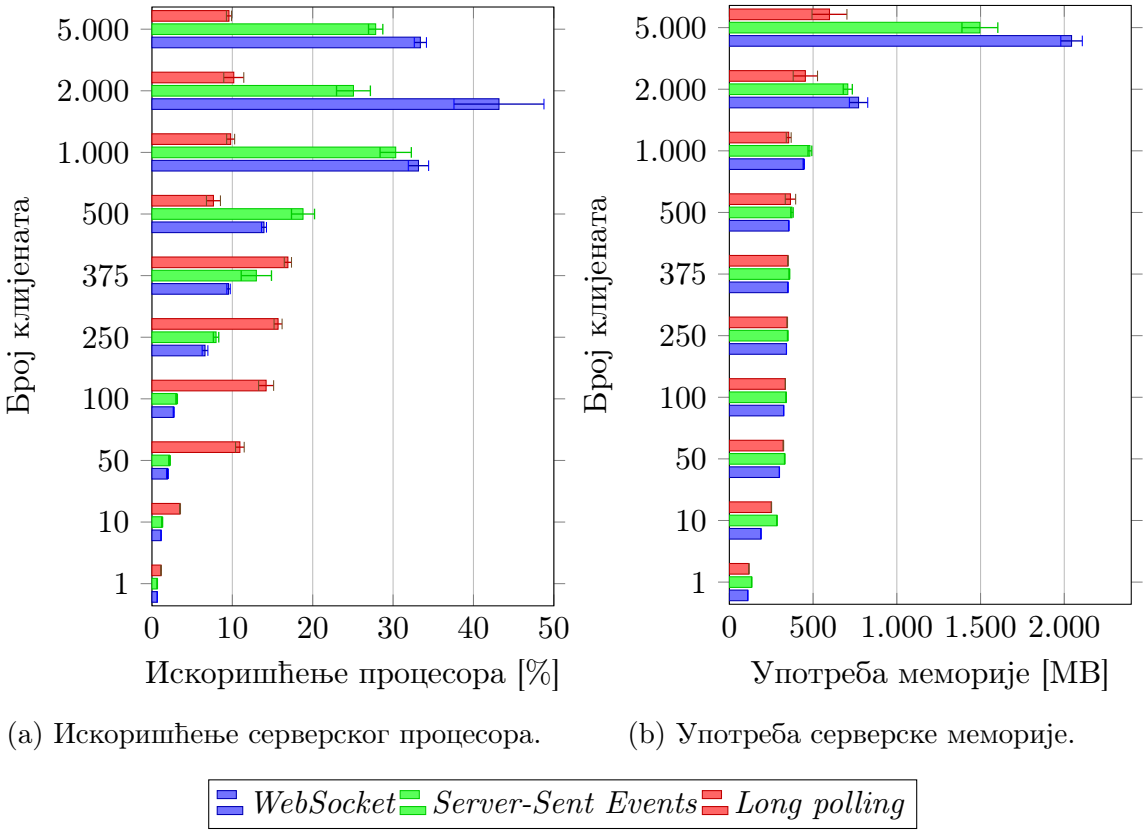
Број клијената	*WebSocket*	*Server-Sent Events*	*long polling*
1–500	100,00	100,00	100,00
1.000	100,00	95,45 ± 4,33	98,96 ± 2,27
2.000	80,24 ± 8,15	67,98 ± 1,90	59,55 ± 3,74
5.000	53,85 ± 1,41	46,11 ± 0,89	22,94 ± 2,82

Вредности су приказане у процентима као средња вредност ± стандардна девијација.

Измерени број испорука у секунди наставио је да расте са бројем клијената, јер се свака генерисана порука шаље сваком повезаном клијенту, па расте и укупан број очекиваних испорука. Због тога већи проток не значи нужно и поузданију испоруку: број забележених испорука може да расте иако се њихов удео у укупном броју очекиваних испорука смањује.

Метрика протока такође не показује колико је појединачна порука чекала на испоруку, па је треба тумачити заједно са уделом успешно испоручених порука и p95 кашњењем. До 500 клијената проток и удео испоруке били су готово једнаки код сва три приступа, али је *long polling* већ у том опсегу показивао знатно веће кашњење.

Употреба серверских ресурса



(a) Искоришћење серверског процесора.

(b) Употреба серверске меморије.

Слика 6.2: Просечно искоришћење процесора и просечна употреба меморије серверског процеса у зависности од броја клијената. Хоризонталне линије представљају стандардну девијацију између поновљених мерења.

До 500 клијената просечно искоришћење процесора код протокола *WebSocket* и механизма *Server-Sent Events* углавном је расло са бројем клијената. Приступ *long polling* већ при мањем броју клијената имао је веће искоришћење процесора, што се може повезати са понављањем обраде *HTTP* захтева.

При 500 клијената просечно искоришћење процесора код приступа *long polling* опало је, иако је остварена потпуна испорука порука. Истовремено је p95 кашњење порасло на 365,24 ms, па нижа употреба процесора у овом сценарију не означава и боље укупне перформансе. При 1.000 и више клијената искоришћење процесора није расло сразмерно оптерећењу, док су истовремено забележени раст кашњења и пад удела успешно испоручених порука. Због тога употребу процесора треба тумачити заједно са клијентским метрикама.

Просечна употреба меморије серверског процеса приказана је на слици 6.2b.

До 500 клијената она је углавном била највећа код механизма *Server-Sent Events*, док је приступ *long polling* при 500 клијената показао већу варијабилност између поновљених мерења. При 1.000 клијената употреба меморије израженије је порасла код протокола *WebSocket* и механизма *Server-Sent Events*, а при 2.000 и 5.000 клијената раст је забележен код сва три приступа. При 5.000 клијената највећа просечна употреба меморије измерена је код протокола *WebSocket*, затим код механизма *Server-Sent Events*. Мања употреба меморије приступа *long polling* при овом оптерећењу мора се посматрати у контексту знатно мањег удела испоручених порука.

Ове вредности представљају количину физичке меморије коју користи серверски процес, измерену преко својства `WorkingSet64`. Оне не обухватају меморију других процеса, контејнера или оперативног система.

Процењени протоколски додатак

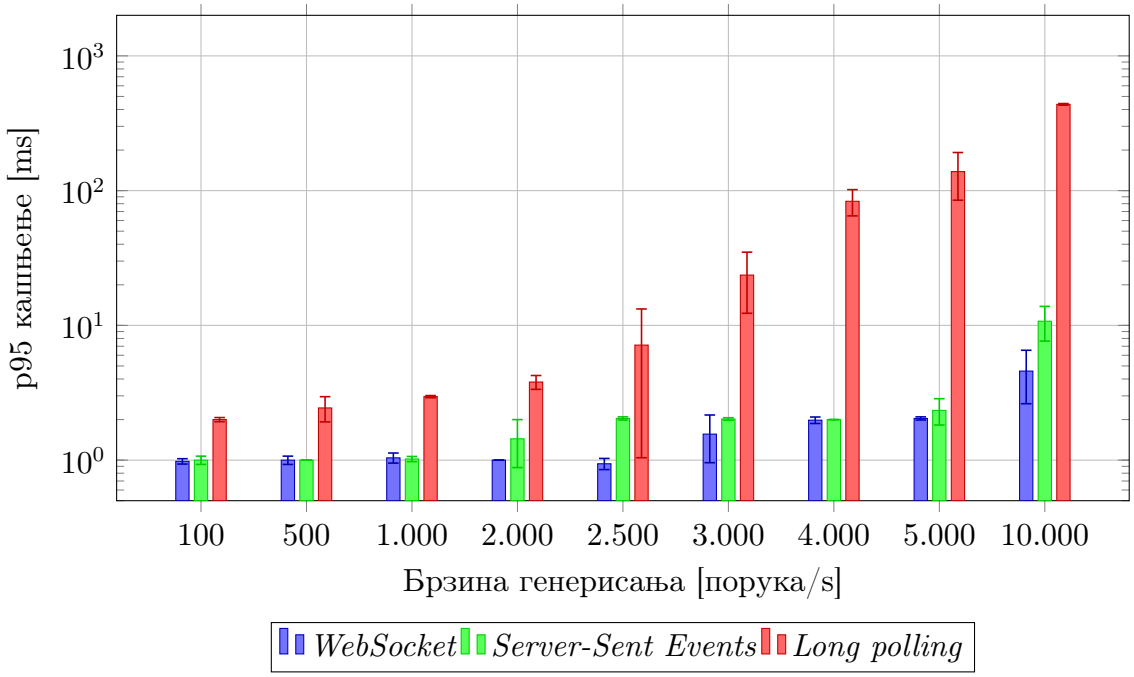
При промени броја клијената процењени додатак протокола *WebSocket* и механизма *Server-Sent Events* остао је приближно константан и износио је око 5,1%, односно 7,9%. Код приступа *long polling* додатак се смањио са 21,55% при једном клијенту на 5,72% при 500 и 5,11% при 5.000 клијената.

Овај пад не значи да је сам комуникациони механизам постао ефикаснији. Са повећањем оптерећења клијенти су чешће преузимали више порука у једном одговору, па се процењени трошак *HTTP* заглавља распоређивао на већи број испорука. При већем броју клијената то је истовремено било праћено растом кашњења и падом удела успешно испоручених порука.

6.3 Утицај брзине генерисања порука

У другом скупу сценарија коришћено је 50 клијената и корисни садржај величине 1.024 бајта. Испитане су брзине генерисања од 100, 500, 1.000, 2.000, 2.500, 3.000, 4.000, 5.000 и 10.000 порука у секунди. Сценарији са више од 3.000 порука у секунди додати су ради испитивања понашања комуникационих приступа при интензивнијем току порука и уочавања области у којој долази до израженијег пада перформанси.

Кашњење



Слика 6.3: Поређење p95 кашњења у зависности од задате брзине генерисања порука. Вертикалне линије представљају стандардну девијацију између поновљених мерења.

Протокол *WebSocket* и механизам *Server-Sent Events* задржали су ниско p95 кашњење до 5.000 порука у секунди. При 3.000 порука у секунди оно је износило 1,56 ms за *WebSocket* и 2,02 ms за *Server-Sent Events*, док је при 5.000 порука у секунди износило 2,04 ms, односно 2,34 ms. У свим овим сценаријима остварена је потпуна испорука. При 10.000 порука у секунди p95 кашњење порасло је на 4,58 ms за *WebSocket* и 10,74 ms за *Server-Sent Events*, али је удео успешно испоручених порука код оба приступа остао већи од 99,98%.

Код приступа *long polling* p95 кашњење било је релативно ниско до 2.000 порука у секунди, након чега је почело изражење да расте. При 3.000 порука у секунди износило је 23,62 ms, при 4.000 порука у секунди 83,36 ms, а при 5.000 порука у секунди 138,38 ms. Највећа варијабилност између поновљених мерења уочена је при 5.000 порука у секунди, када је стандардна девијација p95 кашњења износила 53,48 ms. При 10.000 порука у секунди p95 кашњење достигло је 436,76 ms, уз изражен пад удела успешно испоручених порука. Ови резултати показују да је код приступа *long polling* деградација најпре постала видљива кроз раст кашњења, а затим и кроз губитак испорука.

Губитак порука

До брзине од 5.000 порука у секунди сва три приступа остварила су потпуну испоруку. Губитак је забележен при 10.000 порука у секунди, али се његов обим знатно разликовао између комуникационих приступа. Просечни резултати пет поновљених мерења приказани су у табели 6.3.

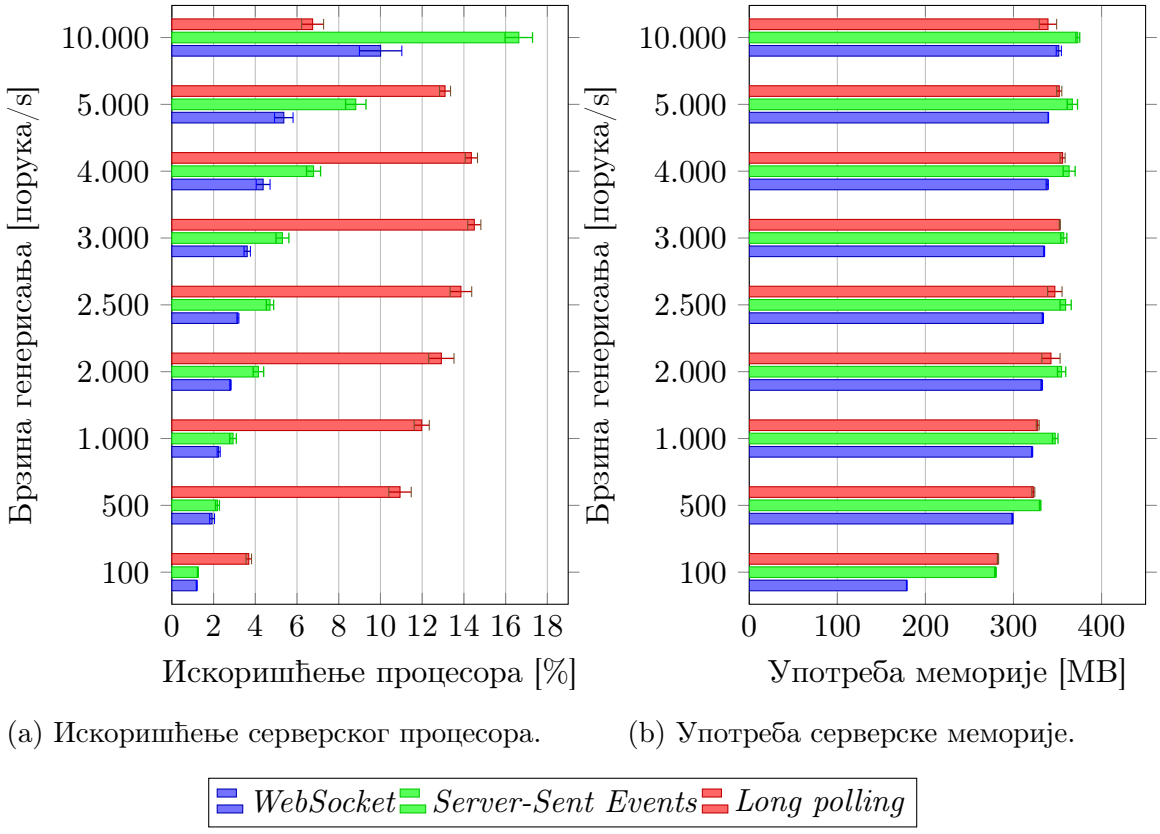
Табела 6.3: Губитак порука при 10.000 порука у секунди

Комуникациони приступ	**Испорука [%]**	**Губитак [%]**	**Понављања са губитком**
WebSocket	99,9868	0,0132	1/5
Server-Sent Events	99,9998	0,0002	2/5
long polling	76,4856	23,5144	5/5

Код приступа *long polling* губитак је забележен у свих пет понављања. Удео изгубљених испорука у појединачним покретањима кретао се од 15,95% до 30,95%, што показује да је систем при овој брзини ушао у област изражене деградације. За сваког клијента засебно су пребројане поруке које није примио, а затим су добијене вредности сабране. На тај начин је у пет покретања укупно евидентирано 17.634.920 недостајућих испорука. Ова вредност представља збир губитака свих клијената, а не број различитих порука које сервер није генерисао.

Код протокола *WebSocket* губитак је забележен у једном понављању, док се код механизма *Server-Sent Events* појавио у два понављања, али је у оба случаја просечан удео испоруке остао веома близу 100%. Сервер је у свим покретањима генерисао најмање 99,987% циљног броја порука, па забележени губитак није последица недовољне производње порука на серверу.

Употреба серверских ресурса



(a) Искоришћење серверског процесора.

(b) Употреба серверске меморије.

Слика 6.4: Просечно искоришћење процесора и просечна употреба меморије серверског процеса у зависности од брзине генерисања порука. Хоризонталне линије представљају стандардну девијацију између поновљених мерења.

Искоришћење процесора код протокола *WebSocket* и механизма *Server-Sent Events* постепено је расло са повећањем брзине генерисања порука. Код приступа *long polling* раст је био израженији већ при нижим брзинама, а највеће вредности забележене су у области од 3.000 до 4.000 порука у секунди. Након тога искоришћење процесора почело је да опада, па је при 10.000 порука у секунди било ниже него код преостала два приступа. Овај пад не указује на већу ефикасност, јер је истовремено забележен изражен губитак испорука.

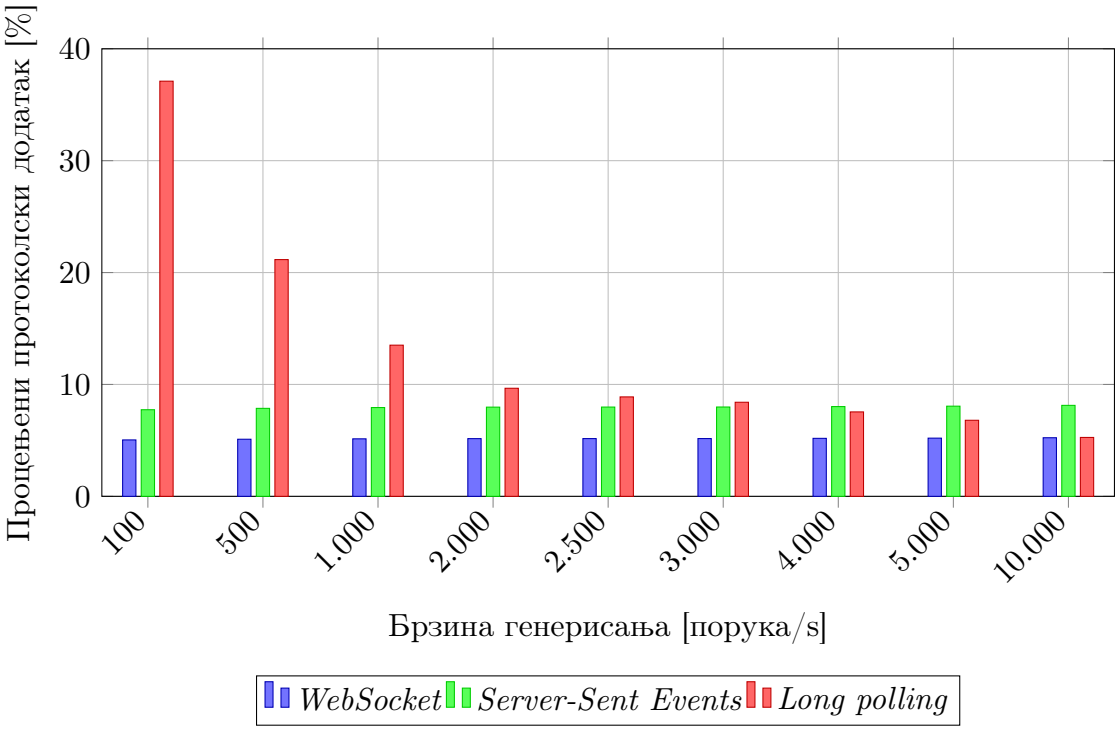
Просечна употреба меморије мењала се мање изражено од искоришћења процесора. Након почетног раста, код сва три приступа остала је у релативно уском опсегу, без наглог повећања при највећој брзини. Код приступа *long polling* пад перформанси стога није био праћен растом просечне употребе меморије, већ пре свега повећањем кашњења и губитком испорука.

Ови резултати показују да се област засићења не може утврдити само на основу употребе серверских ресурса. Искоришћење процесора и меморије потребно је тумачити заједно са кашњењем, протоком и уделом успешно испоручених порука.

Процењени протоколски додатак

При промени брзине генерисања процењени додатак протокола *WebSocket* и механизма *Server-Sent Events* остао је приближно константан. Код протокола *WebSocket* кретао се приближно од 5,04% до 5,24%, а код механизма *Server-Sent Events* од 7,74% до 8,13%.

Код приступа *long polling* додатак се смањио са 37,10% при 100 порука у секунди на 13,51% при 1.000, 6,80% при 5.000 и 5,26% при 10.000 порука у секунди. Са повећањем брзине у једном одговору преносило се више порука, па се трошак *HTTP* заглавља распоређивао на већи број испорука.



Слика 6.5: Процењени протоколски додатак у зависности од брзине генерисања порука.

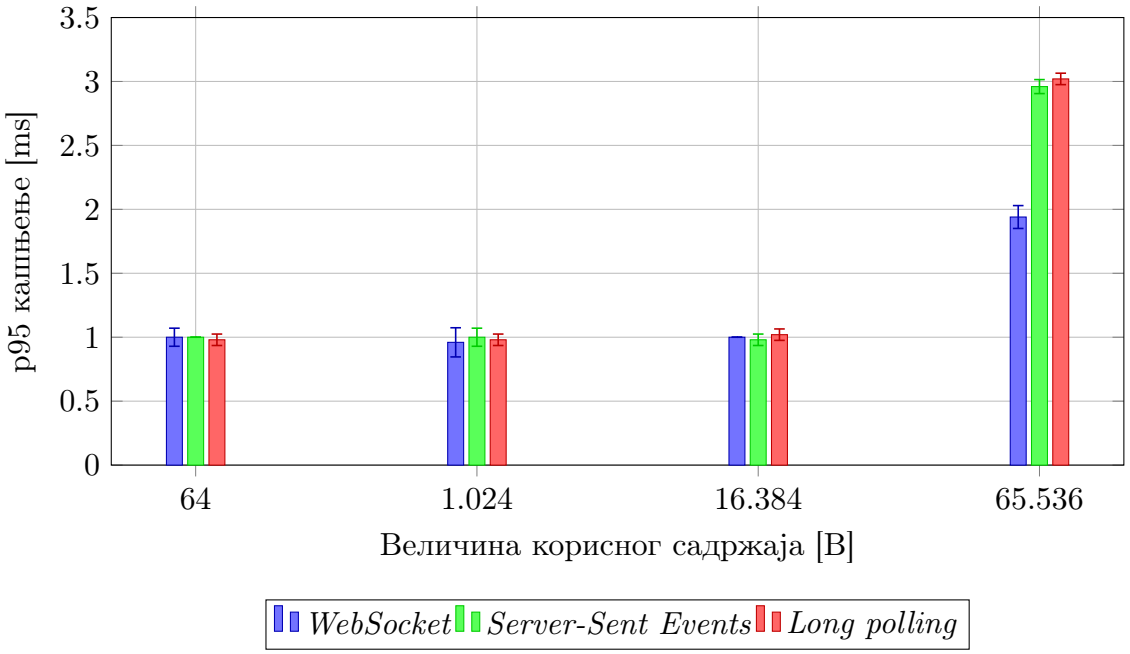
Нижа вредност додатка при највећим брзинама не представља боље укупне перформансе приступа *long polling*. При 10.000 порука у секунди она је била

праћена великим растом кашњења и просечним губитком од 23,51% очекиваних испорука.

6.4 Утицај величине корисног садржаја

У трећем скупу сценарија коришћено је 10 клијената, а брзина генерисања износила је 100 порука у секунди. Испитане су величине корисног садржаја од 64, 1.024, 16.384 и 65.536 бајтова. Изабрани опсег омогућава поређење утицаја веома малих и знатно већих порука на протоколски додатак, проток и остале показатеље перформанси.

Кашњење

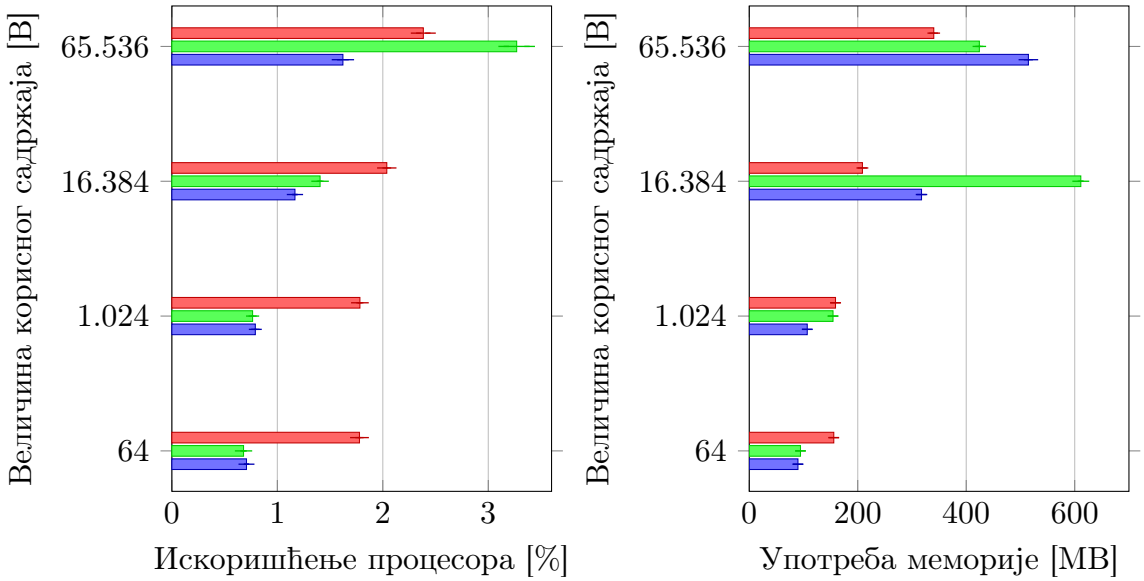


Слика 6.6: Поређење p95 кашњења у зависности од величине корисног садржаја. Вертикалне линије представљају стандардну девијацију између поновљених мерења.

До величине корисног садржаја од 16.384 бајта p95 кашњење код сва три приступа остало је приближно 1 ms, без значајних разлика између њих. При величини од 65.536 бајтова кашњење је порасло на 1,94 ms за *WebSocket*, 2,96 ms за *Server-Sent Events* и 3,02 ms за *long polling*.

Иако је обрада већег садржаја повећала кашњење, удео успешно испоручених порука остао је 100% код сва три приступа. Протокол *WebSocket* при највећој испитаној величини задржао је нешто ниже кашњење од преостала два приступа.

Употреба серверских ресурса



(a) Искоришћење серверског процесора.

(b) Употреба серверске меморије.

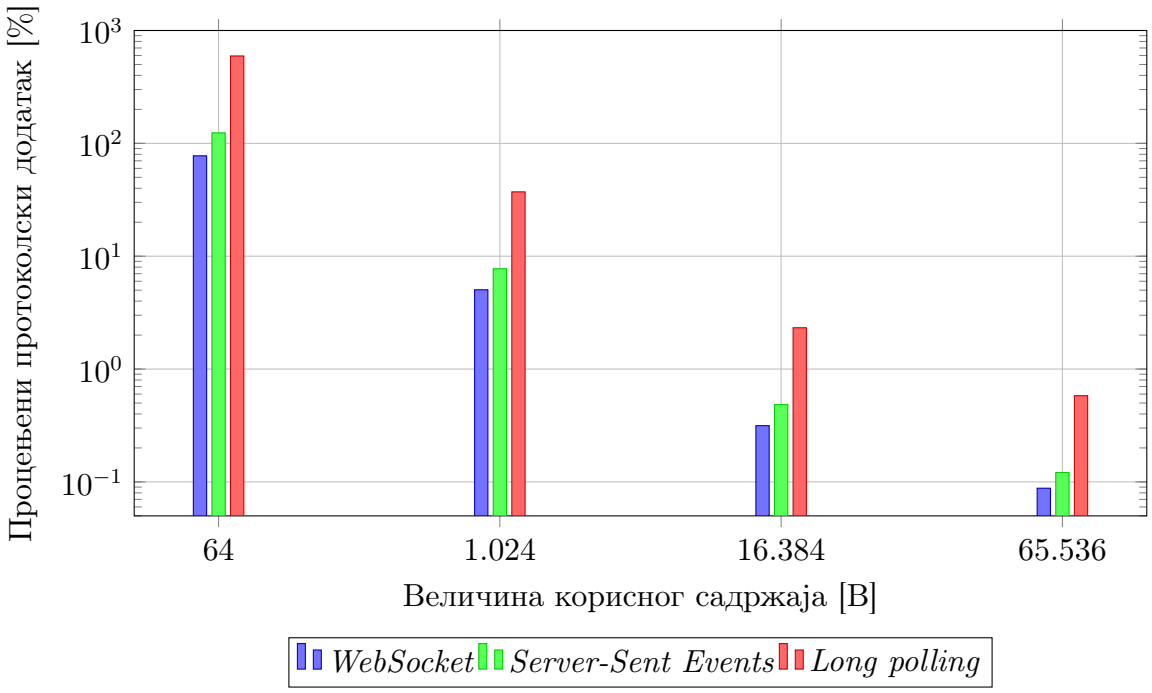
Слика 6.7: Просечно искоришћење процесора и просечна употреба меморије серверског процеса у зависности од величине корисног садржаја. Хоризонталне линије представљају стандардну девијацију између поновљених мерења.

Са повећањем величине корисног садржаја искоришћење процесора углавном је расло код сва три приступа. Најизраженији раст забележен је код механизма *Server-Sent Events*, чије је просечно искоришћење процесора при садржају од 65.536 бајтова било веће него код преостала два приступа.

Употреба меморије такође је углавном расла са величином корисног садржаја, али образац није био потпуно монотон. Код протокола *WebSocket* и приступа *long polling* највећа просечна вредност забележена је при садржају од 65.536 бајтова. Код механизма *Server-Sent Events* највећа вредност забележена је при 16.384 бајта, након чега је при највећем садржају била нижа.

Овај резултат показује да употреба меморије не зависи само од величине појединачне поруке, већ и од тренутка узорковања, алокација и рада сакупљача отпада.

Процењени протоколски додатак



Слика 6.8: Процењени протоколски додатак у зависности од величине корисног садржаја.

Процењени додатак најизраженији је код малих порука. За корисни садржај од 64 бајта износи 77,55% за *WebSocket*, 123,85% за *Server-Sent Events* и 594,74% за *long polling*. Код садржаја од 65.536 бајтова одговарајуће вредности опадају на 0,088%, 0,121% и 0,581%.

Ова промена је очекивана јер метаподаци кодиране поруке и додатни подаци комуникационог механизма имају велики релативни удео када је корисни садржај мали. Са повећањем корисног садржаја њихов удео у укупној количини података постаје знатно мањи.

Приступ *long polling* има највећи процењени додатак због понављања *HTTP* захтева и одговора, за које се сваки пут преносе одговарајућа заглавља [8]. Механизам *Server-Sent Events* користи један дуготрајни *HTTP* одговор, али

сваку поруку обликује као текстуални догађај. Протокол *WebSocket* након успостављања везе преноси поруке у оквирима и не понавља комплетна *HTTP* заглавља при свакој испоруци [2], због чега има најмањи процењени додатак.

6.5 Ограничења истраживања

Сервер и генератор оптерећења користили су ресурсе истог физичког рачунара. Због тога при високим оптерећењима није могуће потпуно искључити међусобни утицај њиховог коришћења процесора, меморије и других системских ресурса. Пре свега, у сценаријима са 1.000 и више истовремених клијената могуће је да су расположиви хардверски ресурси постали један од ограничавајућих фактора.

Експерименти су изведени у контролисаном локалном окружењу, па резултати не обухватају кашњење, губитак пакета, посредничке сервере и друге услове који се могу јавити при комуникацији преко јавне мреже. Клијенти су програмски симулирани и не обухватају трошак приказивања и обраде података у стварном веб прегледачу.

Измерено кашњење представља време од настанка поруке на серверу до њеног пријема на клијенту. Мерење не обухвата *TLS*, који би увео додатни трошак успостављања и заштите комуникације.

Свако мерење трајало је 30 секунди, па резултати пре свега описују краткорочно понашање система. Они не омогућавају процену дуготрајне стабилности, утицаја продуженог рада сакупљача отпадака и могућег нагомилавања ресурса током дужег временског периода.

Коначно, у овом раду користи се процењена количина пренетих података на апликационом нивоу, која не представља укупан број бајтова стварно пренетих преко мреже.

6.6 Дискусија

Резултати показују да сва три приступа могу да обезбеде поуздану испоруку уз ниско кашњење при нижем оптерећењу. Разлике постају израженије када се систем приближи границама својих могућности. Повећање броја клијената и брзине генерисања открива разлике у скалабилности приступа, док

промена величине корисног садржаја најјасније показује разлике у процењеном протоколском додатку.

Протокол *WebSocket* и механизам *Server-Sent Events* показали су слично понашање при умереном броју клијената и задржали ниско кашњење и готово потпуну испоруку при повећању брзине генерисања. Међутим, повећање броја истовремених клијената показало је границу ове сличности. *WebSocket* је дуже задржао ниско кашњење и већи удео успешно испоручених порука, док су при 2.000 и 5.000 клијената сва три приступа показала знаке преоптерећења. То указује да је у посматраном окружењу повећање броја истовремених веза представљало захтевнији облик оптерећења од повећања брзине генерисања при умереном броју клијената.

Код приступа *long polling* знаци деградације појавили су се раније. При промени броја клијената кашњење је значајно порасло пре појаве губитка, док је при промени брзине потпуна испорука задржана до 5.000 порука у секунди, али уз постепени раст кашњења. При 10.000 порука у секунди забележен је изражен губитак испорука. Понављање *HTTP* захтева и одговора ствара додатну обраду, а ограничени број порука у једном одговору отежава надокнађивање заостатка. Конкретна област у којој долази до губитка зависи и од параметара посматране имплементације, пре свега капацитета бафера и највећег броја порука у једном одговору, па је не треба тумачити као општу границу приступа *long polling*.

Резултати при великом оптерећењу показују да појединачне метрике могу довести до погрешног закључка ако се посматрају изоловано. Ниже p95 кашњење не мора да означава боље понашање ако велики део порука није испоручен, јер се кашњење израчунава само за примљене поруке. Већи проток не подразумева већи удео испоруке, док нижа употреба процесора не представља већу ефикасност ако систем обрађује мањи део очекиваног оптерећења. Због тога је неопходно заједно тумачити кашњење, проток, поузданост испоруке и употребу серверских ресурса.

Промена величине корисног садржаја највише је утицала на процењени додатак у односу на корисни садржај. Када је корисни садржај мали, подаци који се додају ради формирања и преноса поруке чине значајан део њене укупне величине. Са повећањем корисног садржаја њихов удео у укупној величини поруке се смањује. У овом скупу сценарија *WebSocket* је имао најмањи процењени додатак, док је он био највећи код приступа *long polling*. Повећање

садржаја утицало је и на кашњење при највећој испитаној величини, али су проток и удео испоруке остали једнаки код сва три приступа.

При повећању броја клијената и брзине генерисања додатак протокола *WebSocket* и механизма *Server-Sent Events* остао је приближно константан. Код приступа *long polling* он се смањивао јер је један одговор садржао више порука, па се трошак *HTTP* заглавља распоређивао на већи број испорука. Такав пад додатка не представља самостално побољшање перформанси, јер је при већем оптерећењу био праћен растом кашњења и губитком испорука.

Поред детаљно разматраних показатеља, анализирани су остварење циљног броја генерисаних порука, клијентске грешке, неочекивани прекиди веза, дупликати и поруке примљене ван редоследа. Нису забележене грешке, неочекивани прекиди, дупликати нити поруке ван редоследа. У свим сценаријима сервер је генерисао најмање 99,93% циљног броја порука, без значајних разлика између приступа. Просечна вредност кашњења, медијана и p99 углавном су пратили обрасце уочене на основу p95 кашњења, па нису засебно разматрани.

Глава 7

Закључак

У овом раду приказана је еволуција комуникације у реалном времену у веб апликацијама, са посебним осврtом на протокол *WebSocket* и његово поређење са механизмом *Server-Sent Events* и приступом *long polling*. Циљ рада био је да се размотре техничке карактеристике ових приступа и да се експериментално испита њихово понашање при различитим условима оптерећења.

У теоријском делу анализирани су развој посматраних комуникационих приступа, смер комуникације и начин успостављања и одржавања везе. Практични део обухватио је реализацију мерног окружења које се састоји од сервера, генератора оптерећења и анализатора резултата. Окружење је омогућило поређење сва три приступа при промени броја клијената, брзине генерисања порука и величине корисног садржаја, уз поновљена мерења и заједничко посматрање клијентских и серверских метрика.

Експериментални резултати потврдили су да разлике у моделу комуникације постају значајније са повећањем оптерећења. При умереном оптерећењу *WebSocket* и *Server-Sent Events* показали су слично понашање у испитаном једносмерном току. При већем броју клијената *WebSocket* дуже је задржао повољне вредности кашњења и поузданости испоруке, док су се код посматране имплементације приступа *long polling* ограничења модела заснованог на понављању *HTTP* циклуса испољила ранијим растом кашњења и појавом губитка испорука. Мерење додатка у односу на корисни садржај такође је показало предност дуготрајних комуникационих механизама, нарочито при слању малих порука.

Добијени резултати не издвајају један приступ као најбољи за све врсте веб апликација. *Server-Sent Events* представља погодан избор када је потре-

бан једносмерни ток од сервера ка клијентима, док је *WebSocket* примеренији када је потребна двосмерна комуникација или када се очекује интензивнија размена порука. *Long polling* може бити оправдан у мање захтевним системима и окружењима у којима трајнији механизми нису доступни. Избор зато треба заснивати на смеру комуникације, очекиваном оптерећењу, захтевима за кашњењем и поузданошћу, као и прихватљивом трошку имплементације.

Изведени закључци односе се на конкретну имплементацију и контролисано експериментално окружење. При највећем броју клијената на резултате су могла да утичу ограничења хардвера, док локално извршавање није обухватило услове јавне мреже, заштићене везе и стварне клијенте у веб прегледачима. Упркос овим ограничењима, резултати пружају релевантан увид у међусобне разлике посматраних приступа и њихово понашање са повећањем оптерећења. Даље истраживање може обухватити раздвајање сервера и генератора оптерећења на различите рачунаре, испитивање двосмерних сценарија, као и проверу утицаја капацитета редова и бафера и максималног броја порука у једном *long polling* одговору.

Библиографија

[1] Robin Christen and Etienne Gobeli. Event Based Real-Time Synchronization of Web Applications. Bachelor's thesis, University of Applied Sciences and Arts Northwestern Switzerland, March 2020. https://dierk.github.io/Home/data/documentation_P6_Etienne_Gobel_Robin_Christen.pdf.

[2] Ian Fette and Alexey Melnikov. The WebSocket Protocol. RFC 6455, Internet Engineering Task Force, December 2011. https://www.rfc-editor.org/rfc/rfc6455.html.

[3] Roy T. Fielding, Mark Nottingham, and Julian Reschke. HTTP Semantics. RFC 9110, Internet Engineering Task Force, June 2022. https://www.rfc-editor.org/rfc/rfc9110.html.

[4] Jesse James Garrett. Ajax: A New Approach to Web Applications. Adaptive Path, February 2005. https://designftw.mit.edu/lectures/apis/ajax_adaptive_path.pdf.

[5] Amos Q. Haviv. *MEAN Web Development*. Packt Publishing, 2 edition, November 2016. Поглавље 9, одељак "Introducing WebSockets", https://subscription.packtpub.com/book/web-development/9781785886300/9/ch09lvl1sec67/introducing-websockets, приступљено 6. августа 2026.

[6] Ian Hickson. The Web Socket Protocol. IETF Internet-Draft, draft-hixie-thewebsocketprotocol-00, January 2009. https://datatracker.ietf.org/doc/html/draft-hixie-thewebsocketprotocol-00.

[7] Internet Engineering Task Force. BiDirectional or Server-Initiated HTTP (HyBi) Working Group Charter. IETF Datatracker, 2010. https://datatracker.ietf.org/wg/hybi/charter/.

[8] Salvatore Loreto, Peter Saint-Andre, Stefano Salsano, and Greg Wilkins. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. RFC 6202, Internet Engineering Task Force, April 2011. `https://www.rfc-editor.org/rfc/rfc6202.html`.

[9] Victoria Pimentel and Bradford G. Nickerson. Communicating and Displaying Real-Time Data with WebSocket. *IEEE Internet Computing*, 16(4):45–53, July 2012. `https://ieeexplore.ieee.org/document/6197172`.

[10] Dylan Schiemann. Scaling with Comet. In Steve Souders, editor, *Even Faster Web Sites: Performance Best Practices for Web Developers*, chapter 8. O'Reilly Media, June 2009. Поглавље 8, `https://www.oreilly.com/library/view/even-faster-web/9780596803773/ch08.html`, приступљено 6. августа 2026.

[11] Pavel Smolka. Real-time Communication in Web Browser. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2013. `https://is.muni.cz/th/qfmmv/`.

[12] Darryl K. Taft. Microsoft Scrubs Comet from AJAX Tool Set. eWeek, May 2006. `https://www.eweek.com/development/microsoft-scrubs-comet-from-ajax-tool-set/`, приступљено 6. августа 2026.

[13] WHATWG. WebSockets Standard. Living Standard. `https://websockets.spec.whatwg.org/`.

[14] WHATWG. HTML Standard: Server-Sent Events. Living Standard, 2026. `https://html.spec.whatwg.org/multipage/server-sent-events.html`, приступљено 6. августа 2026.

Додатак А: Упутство за репродукцију експеримената

Овај додатак описује поступак изградње мерног окружења, покретања експерименталних матрица и обраде добијених резултата. Све команде извршавају се из коренског директоријума репозиторијума[1].

Предуслови

За репродукцију експеримената потребни су:

- *Docker Desktop* са доступном компонентом *Docker Compose*;
- *PowerShell*;
- довољно процесорских и меморијских ресурса за изабране сценарије.

Апликације су развијене за извршно окружење *.NET 8.0*. При покретању помоћу контејнера није потребно посебно инсталирати *.NET SDK*, јер се одговарајуће извршно окружење налази у сликама контејнера.

Ради обезбеђивања упоредивих услова, препоручује се затварање других ресурсно захтевних програма, повезивање рачунара на напајање и одржавање непромењене конфигурације система током свих покретања.

Покретање експеримената

За репродукцију коначних експеримената није потребно ручно покретати појединачне компоненте. Скрипта `scripts/run-docker-matrix.ps1` аутома-

[1]Линк ка GitHub репозиторијуму: `https://github.com/NevenaM12/Realtime-communication-protocol-benchmark`.

тизује изградњу слика контејнера, покретање сервера и генератора оптерећења, проверу насталих резултата, заустављање употребљених контејнера и покретање анализатора након завршетка матрице. Слике се подразумевано изграђују на почетку сваке матрице. Ако су већ изграђене и изворни кôд није мењан, наредне матрице могу се покренути са параметром `-SkipBuild`.

Експериментална матрица покреће се командом:

```
powershell -ExecutionPolicy Bypass -File ‘
.\scripts\run-docker-matrix.ps1 ‘
-ConfigPath .\experiments\MATRIX_FILE ‘
-BatchName BATCH_NAME
```

Вредност `MATRIX_FILE` одређује скуп сценарија који ће бити извршен, док `BATCH_NAME` задаје јединствени назив директоријума у који се смештају резултати. За експерименте приказане у раду коришћене су следеће конфигурационе матрице:

Табела 1: Коришћене конфигурационе матрице

Скуп експеримената	`MATRIX_FILE`
Главни скуп сценарија	`main-matrix.json`
Проширени број клијената	`client-extension-matrix.json`
Проширене брзине генерисања	`rate-extension-matrix.json`

За сваки сценарио скрипта извршава по пет понављања сва три комуникациона приступа, при чему се њихов редослед мења између понављања. Пре активне фазе обављају се провера доступности сервера, синхронизација часовника, повезивање клијената и припремни интервал. Активна фаза генерисања порука траје 30 секунди, након чега следе завршни интервал, провера и чување резултата.

Ако се извршавање прекине, постојећи скуп може се наставити додавањем параметра `-Resume`. При настављању се морају користити исти назив скупа и иста конфигурациона датотека.

Резултати и анализа

Резултати се смештају у директоријум:

```
results/BATCH_NAME/
```

За свако појединачно покретање чувају се конфигурација, клијентски резултати, завршни резиме и узорци серверских ресурса. Скрипта за покретање матрице након завршетка експеримената аутоматски покреће анализатор. Агрегирани резултати налазе се у директоријуму:

```
results/BATCH_NAME/analysis/
```

Услови поновљивости

За непосредно поређење са резултатима приказаним у раду потребно је користити исту верзију изворног кода, исте експерименталне матрице и упоредиву конфигурацију хардвера.

Вредности добијене на другачијем хардверу не морају бити бројчано једнаке резултатима из рада. Ипак, уз непромењене сценарије требало би да буде могуће проверити уочене обрасце понашања и релативне разлике између комуникационих приступа.

Додатак Б: Употреба великих језичких модела у изради рада

Током израде овог мастер рада коришћени су велики језички модели (ВЈМ; енгл. *Large Language Model - LLM*) као помоћни алат у писању рада и развоју софтверског решења представљеног у раду.

Употреба при писању рада

ВЈМ су коришћени за проналажење језичких и стилских грешака у тексту. ВЈМ нису коришћени као замена за стручну и научну литературу. Аутор је проверио коришћене изворе и преузео одговорност за тачност тврдњи, интерпретација резултата и закључака изнетих у раду.

Употреба при развоју пројекта

ВЈМ су повремено коришћени као додатни алат за проверу имплементације, слично поступку провере кода (енгл. *code review*), као и за разјашњавање могућих узрока грешака. Одређене сугестије добијене на овај начин су прихваћене, размотрене и додатно прилагођене од стране аутора. ВЈМ су повремено коришћени као помоћ при изради аутоматских тестова за проверу исправности одређених делова имплементације. Ови тестови имали су помоћну улогу и нису били део експерименталног поступка на коме се заснивају резултати и закључци рада, већ су део добре праксе развоја софтвера. Дизајн система, избор технологија, имплементација кључних функционалности и покретање експерименталног решења били су одговорност аутора.

Коришћени алати

У процесу израде рада и развоја пројекта коришћен је *ChatGPT (OpenAI)*.

Одговорност аутора

ВЈМ су коришћени искључиво као помоћни алати. Аутор је самостално доносио одлуке о садржају рада и дизајну и имплементацији система, проверавао и прилагођавао генерисане предлоге и извршио коначну проверу текста, програмског кода, експерименталних резултата и закључака рада.

Биографија аутора

Невена Мијаиловић рођена је 12.9.2000. године у Чачку. Одрасла је у Лучанима, где је завршила Основну школу „Милан Благојевић". Након тога, уписује и завршава друштвено-језички смер Гимназије „Свети Сава" у Пожеги, као носилац дипломе „Вук Караџић". Године 2019. уписује Математички факултет Универзитета у Београду, смер Рачунарство и информатика. Основне академске студије завршава 2023. године са просечном оценом 8.73, чиме стиче звање дипломираног математичара. Исте године уписује мастер студије на истом факултету, на студијском програму Математика, модул Математика и рачунарство. Своју професионалну каријеру започиње 2024. године на позицији *Full-Stack Web Developer*-а у компанији *TeleTrader*.