

Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH



Instytut Telekomunikacji

Praca dyplomowa magisterska

na kierunku Telekomunikacja
w specjalności Teleinformatyka i Cyberbezpieczeństwo

Środowisko wykonawcze dla komponentów aplikacji IoT

Mykyta Vovk

numer albumu 309729

promotor

dr inż. Jarosław Domaszewicz

WARSZAWA 2023

Streszczenie

Środowisko wykonawcze dla komponentów aplikacji IoT

Celem tej pracy magisterskiej była implementacja i badania środowiska wykonawczego NAPES dla systemu Android, który ma za zadanie emulację ruchu sieciowego przybliżonego do ruchu generowanego przez dowolną aplikację IoT, wykorzystując protokoły TCP i UDP. NAPES wczytuje konfiguracje zdefiniowane w komponentach RCR (Runtime Component Representation) i na podstawie podanych parametrów zaczyna emulację ruchu sieciowego. Takie rozwiązanie pozwoli na zbadanie wybranych sieci IoT pod względem ich wydajności i możliwości dostosowania do nowych zmian w sieci (np. dodanie nowego węzła do skonfigurowanej sieci). W tym celu NAPES wykonuje symulację maszyny stanów i zgodnie z stanem, w którym obecnie znajduje się system, generowany jest przepływ sieciowy z odpowiednimi parametrami. Warunkami na zmianę stanów w maszynie są zdarzenia aplikacyjne, które zostały zaimplementowane za pomocą protokołu MQTT.

Zakres pracy obejmuje całkowitą implementację środowiska wykonawczego za pomocą języka Java oraz gruntowne przetestowanie emulatora pod kątem dokładności generacji ruchu sieciowego. Część badawcza pracy zawiera wyniki miar dokładności generacji przepływów sieciowych o różnych parametrach oraz w różnych warunkach pracy środowiska. Wszystkie miary dokładności zostały obliczone na podstawie odstępów czasu między wysłaniem kolejnych pakietów. Wyniki pomiarów wykazały, że zmiana parametrów ma dość niewielki wpływ na dokładność systemu, natomiast obserwowane są niektóre tendencje. Na przykład, wraz ze wzrostem ilości danych przesyłanych na jednostkę czasu, zmniejsza się dokładność wysłania pakietów sieciowych. Także został zbadany wpływ zmiany priorytetu wątku w systemie na dokładność generacji przepływów. Wyniki tych badań wykazały, że zmiana priorytetu ma efekt niejednoznaczny. W niektórych przypadkach zmiana priorytetu poprawiła dokładność generacji, jednak w innych pogorszyła ją.

Słowa kluczowe: środowisko wykonawcze, komponent, RCR, NAPES, Android, generacja ruchu sieciowego, IoT, TCP, UDP, MQTT, Java, maszyna stanów, priorytety wątków

Abstract

Runtime for IoT application components

The aim of this master's thesis was to implement and test the NAPES runtime for Android, which is designed to emulate network traffic similar to that generated by any IoT application, using TCP and UDP protocols. NAPES loads the configurations defined in the RCR (Runtime Component Representation) components and based on the given parameters, starts emulating network traffic. This solution allows to test selected IoT networks for their performance and the ability to adapt to new changes in the network (e.g., adding a new node to the configured network). For this purpose, NAPES performs a state machine simulation and, according to the current state of the system, a network flow with appropriate parameters is generated. The conditions for changing states in the state machine are application events that have been implemented using the MQTT protocol.

The scope of work includes the complete implementation of the runtime environment using Java and thorough testing of the emulator in terms of the accuracy of network traffic generation. The research part of the thesis contains the results of the accuracy measurements of network flow generation with various parameters and in various operating conditions of the environment. These accuracy measures were computed from a set of inter-packets times. The results of the measurements showed that the change of parameters has quite a small effect on the accuracy of the system, but some trends are observed. For example, as the amount of data transmitted per unit of time increases, the accuracy of sending network packets decreases. The impact of changing the thread priority in the system on the accuracy of network flow generation was also examined. The results of these studies showed that changing the priority has an ambiguous effect. In some cases, changing the priority improved generation accuracy, but worsened it in others.

Keywords: runtime, component, RCR, NAPES, Android, network traffic generation, IoT, TCP, UDP, MQTT, Java, state machine, thread priorities



Politechnika Warszawska

Warszawa, dd.mm.rrrr

miejscowość i data

imię i nazwisko studenta

numer albumu

kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4. lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz. U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płyce kompaktowej) oraz treść pracy dyplomowej w systemie iSOD są identyczne.

czytelny podpis studenta

Spis treści

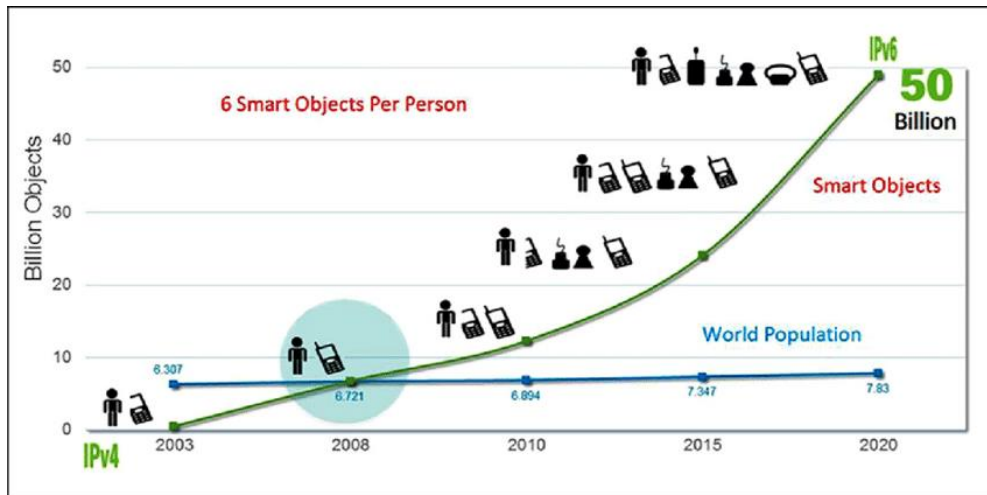
1. Wprowadzenie.....	10
1.1. Motywacja	10
1.2. Cel pracy	10
1.3. Organizacja aplikacji NAPES Runtime	11
1.4. Architektura i implementacja.....	11
1.5. Badania	13
1.6. Wkład własny	13
1.7. Układ pracy	14
2. Stan wiedzy	15
3. System NAPES	17
3.1. Ogólna architektura systemu NAPES	17
3.2. Format RCR: składnia i semantyka	18
4. Wykorzystane API, protokoły i narzędzia.....	19
4.1. Android, Java	19
4.2. Protokół MQTT	19
4.3. MQTT Broker Mosquitto	20
4.4. MQTT Paho.....	21
4.5. Narzędzia do analizy ruchu sieciowego.....	21
5. Model komponentu NAPES Runtime dla systemu Android.....	22
6. Implementacja komponentu NAPES Runtime dla systemu Android.....	24
6.1. Instalacja obsługi protokołów.....	24
6.1.1. Instalacja oraz uruchomienie brokera MQTT	24
6.1.2. Tworzenie klienta MQTT w Android Studio.....	27
6.1.3. Tworzenie oraz uruchomienie serwera UDP.....	28
6.1.4. Tworzenie klienta UDP w aplikacji Android.....	29
6.1.5. Tworzenie oraz uruchomienie serwera TCP	33
6.1.6. Tworzenie klienta TCP w aplikacji Android	34
6.2. Implementacja parsera formatu RCR	36
6.3. Implementacja środowiska wykonawczego	41
6.3.1. Logika działania emulatora	42
6.3.2. Organizacja kolejki zdarzeń.....	43

6.3.3.	Implementacja obsługi stanów	44
6.3.4.	Implementacja obsługi portów	45
6.3.5.	Obsługa zdarzeń lokalnych	46
6.3.6.	Tworzenia obsługi logów systemu	46
7.	Testowanie komponentu NAPES Runtime dla systemu Android	48
8.	Eksperyment ze zmianą parametrów w zakresie jednego przepływu	52
8.1.	Zmiana długości pakietów jednego przepływu dla $T = 25$ [ms]	54
8.2.	Zmiana długości pakietów jednego przepływu dla $T = 5$ [ms]	55
8.3.	Wpływ priorytetu wątków na dokładność generacji ruchu sieciowego	56
8.3.1.	Przepływ generowany przez wątek o domyślnym priorytecie	56
8.3.2.	Przepływ generowany przez wątek o maksymalnym priorytecie	58
8.3.3.	Przepływ generowany przez wątek o minimalnym priorytecie	59
9.	Badania kilku przepływów uruchomionych jednocześnie	61
9.1.	Dwa jednocześnie uruchomione przepływy opisane funkcją $F = (5\text{ms}, 1024)$..	62
9.2.	Dwa jednocześnie uruchomione przepływy opisane funkcją $F = (25\text{ms}, 1024)$..	63
9.3.	Dwa jednocześnie uruchomione przepływy opisane funkcjami: $F_1 = (5\text{ms}, 1024)$ i $F_2 = (25\text{ms}, 1024)$	64
9.4.	Dwa jednocześnie generowane przepływy przez wątki o różnych priorytetach: $F = (5\text{ms}, 1024)$	65
9.5.	Dwa jednocześnie generowane przepływy przez wątki o różnych priorytetach: $F = (25\text{ms}, 1024)$	66
9.6.	Dwa jednocześnie generowane przepływy przez wątki o różnych priorytetach: $F_1 = (5\text{ms}, 1024)$ oraz $F_2 = (25\text{ms}, 1024)$	67
10.	Podsumowanie i kierunki dalszych prac	69
	Literatura	71
	Spis rysunków	73
	Spis tabel	76
	Spis listingów	77
	Załącznik I. Fragment kodu wykorzystywany podczas badań dokładności wysyłania pakietów	78

1. Wprowadzenie

1.1. Motywacja

Urządzenia **IoT** podlegają nieustannej rewolucji, a ilość tych urządzeń rośnie w postępie geometrycznym [1]. Na rys. 1.1 jest pokazany wykres zależności, jak zwiększała się liczba inteligentnych urządzeń zgodnie z czasem. Jak widać, w 2020 roku ta liczba przekroczyła wartość w 50 miliardów urządzeń, które mogą łączyć się z siecią (z tym jest związane przejście na protokół IPv6).



Rysunek 1.1 Rozwój IoT na osi czasu [2]

Moc obliczeniowa tych urządzeń coraz większa się a aplikacje dla nich stają się coraz bardziej złożone. Z tego powstają nowe problemy przy projektowaniu nowych aplikacji lub przy wdrażaniu nowych zmian do już istniejącej sieci IoT. Projektanta systemu powinien przewidzieć, czy jego nowe rozwiązanie nie będzie zakłócało pracy działającego systemu i sieci. Dla tych celów istnieje wiele rozwiązań komercyjnych dla systemów **Unix**, które są obsługiwane przez urządzenia IoT. Jednym z najbardziej popularnych rozwiązań jest narzędzie do pomiaru i konfiguracji wydajności sieci **Iperf**.

Wraz ze wzrostem wydajności samych urządzeń IoT – zaczynają one obsługiwać systemy wyższego poziomu na bazie Unix, na przykład **Android**. Na dzień dzisiejszy, na rynku są mało rozwiązań analogicznych do Iperf dla systemów Android. Dlatego motywacją tej pracy jest stworzenie takiego narzędzia, które pozwoliłoby emulować aplikację rzeczywistą, przy tym robiąc różne pomiary, które będą służyć do statystyki wydajności sieci oraz samego urządzenia.

1.2. Cel pracy

Celem pracy jest realizacja i gruntowne przetestowanie środowiska wykonawczego NAPES Runtime dla systemu Android. Ten system musi być na tyle elastyczny, że jego zachowanie w czasie rzeczywistym musi być jak najbardziej przybliżone do aplikacji IoT z punktu widzenia sieci [3].

NAPES Runtime otrzymuje opis zachowania komponentu dla aplikacji emulującej w formie tzw. pliku RCR (Runtime Component Representation), a następnie realizuje

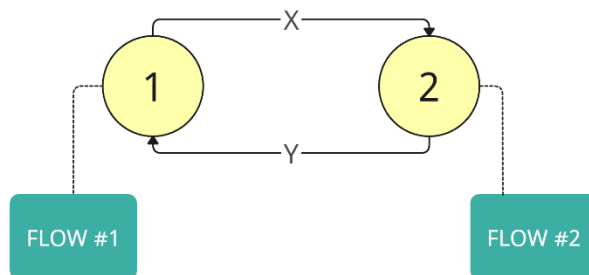
opisane zachowania (generacja ruchu, interakcja z innymi komponentami za pomocą zdarzeń, itp.). Na węźle NAPES może być uruchomiony komponent, który może komunikować się z komponentami działającymi na innych węzłach. Taki system pozwoli na tworzenie sieci węzłów NAPES komunikujących się między sobą podczas emulacji.

Po stworzeniu środowiska NAPES Runtime, może zostać ono wykorzystane do zbadania wydajności wybranych sieci. Na końcu emulacji środowiska wykonawczego z każdego węzła będą zbierane statystyki, które będą wykazywać, jak dokładny jest zaprojektowany system i czy będzie on przydatny na potrzeby użytkownika.

1.3. Organizacja aplikacji NAPES Runtime

Środowisko wykonawcze NAPES to system, w którym kluczową jednostką jest komponent. Każdy komponent składa się z zdarzeń, maszyny stanów, przepływów oraz portów, a jego konfiguracja jest opisana w plikach o rozszerzeniu RCR. Środowisko wykonawcze interpretuje te konfiguracje w oddzielnym wątku aplikacji [4].

Także w komponencie są zdefiniowane parametry przepływów sieciowych (rozmiar pakietu oraz odstęp czasowy między kolejnymi pakietami). W jednym komponencie mogą być zdefiniowane wiele przepływów z różnymi parametrami. Te przepływy są generowane względem maszyny stanów rys.1.2.



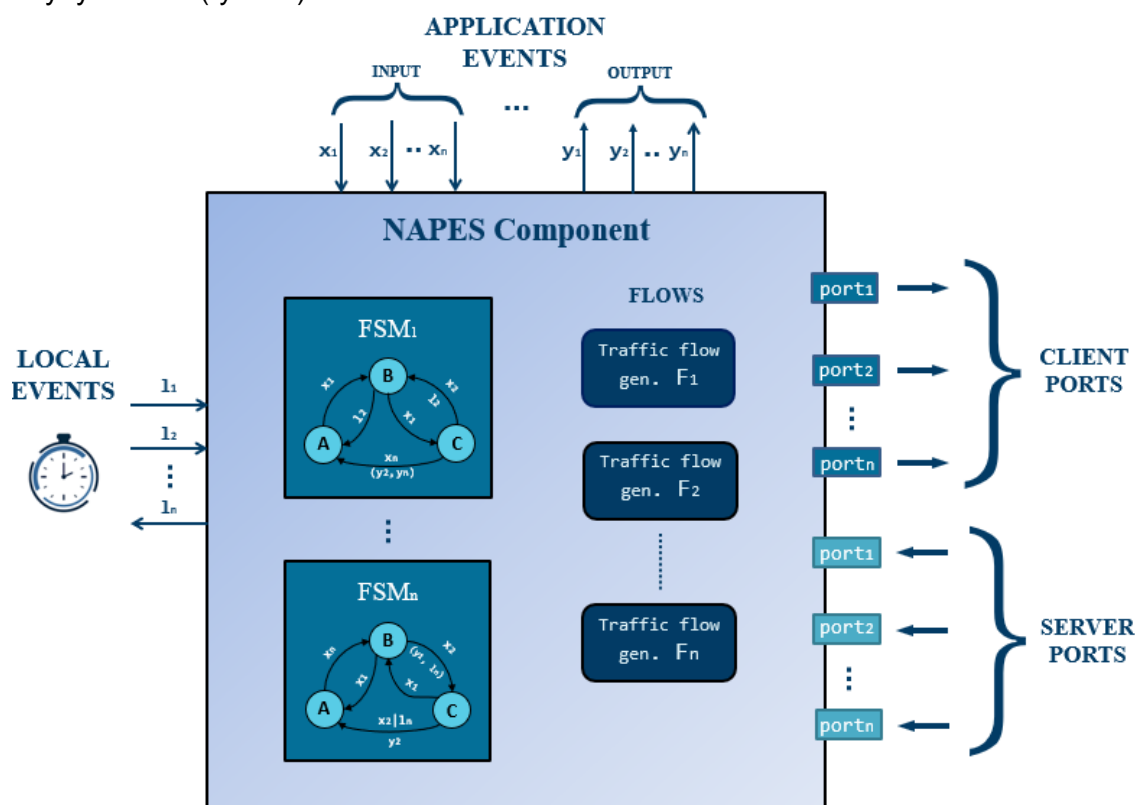
Rysunek 1.2 Podstawowa zasada działania systemu NAPES

Powyższy schemat pokazuje podstawową zasadę działania systemu NAPES. Jak widać, w stanie 1 system generuje przepływ o nazwie „Flow #1”, a w stanie 2 system generuje „Flow #2”. Znaczniki X i Y są warunkami przejścia między kolejnymi stanami, mogą to być wchodzące zdarzenia aplikacyjne. Tworząc wiele takich przepływów oraz stanów, można emulować zachowanie aplikacji działającej na rzeczywistym węźle IoT. Dodatkowo można powiedzieć, że komponent może zawierać wiele maszyn stanów, które uruchamiają się w oddzielnych wątkach programu, co robi system bardziej elastycznym.

1.4. Architektura i implementacja

Podstawową jednostką architektury środowiska wykonawczego NAPES jest komponent [5]. W tym komponencie jest opisane zachowania maszyny stanów oraz portów działających na węźle. Implementacja takiego systemu wymaga stosowania programowania wielowątkowego, ponieważ generacja ruchu sieciowego jest innym procesem, niż proces zarządzania *maszyną stanów*. Dlatego architekturę komponentu

można podzielić na dwie główne warstwy: warstwę obsługi portów oraz warstwę obsługi maszyny stanów (rys 1.3).



Rysunek 1.3 Model komponentu NAPES

Warstwa obsługi portów decyduje o tym, jaki generator przepływu musi być uruchomiony w zależności od tego, w jakim stanie znajduje się maszyna stanów. Ta warstwa ma za zadanie robić walidację parametrów przepływu po każdym przejściu między stanami. **Warstwa obsługi maszyny stanów** posiada zadeklarowane w sobie różne stany, ale posiada tylko jeden stan aktualny. Po uruchomieniu emulacji, warstwa obsługi FSM ma za zadanie ustawić aktualny stan na stan początkowy, który jest zadeklarowany w komponencie.

Każdy port ma zdefiniowane w sobie reguły, w których każdy stan maszyny może mieć przypisany do siebie dowolny generator przepływu. Komponent NAPES rozróżnia dwa typy zdarzeń: lokalne i aplikacyjne. Zdarzenia aplikacyjne są realizowane za pomocą protokołu MQTT. Wysyłanie wiadomości MQTT na „topic”, który jest subskrybowany przez dowolny inny komponent powoduje dla niego obsługę zdarzenia wejściowego. Zdarzenia lokalne wykonują funkcję timera, który może zostać uruchomiony tylko po przejściu między stanami, wykonując odpowiednie akcje.

Dzięki temu komponenty mogą komunikować się między sobą i reagować na jakiegokolwiek zmiany w całym ekosystemie NAPES. Taka implementacja pozwala na zmianę stanu i generację dowolnego przepływu sieciowego. Ogólnie rzecz biorąc, system NAPES i jego komponenty są zaprojektowane w sposób, który umożliwia dynamiczne tworzenie i uruchamianie komponentów na węzle Android, co pozwala na emulację aplikacji prawdziwiej z punktu widzenia sieci [6].

Tak jak system musi działać na systemach Android, to do jego tworzenia zostało wybrane środowisko programistyczne Android Studio, które zawiera wszystkie niezbędne narzędzia do tworzenia takich aplikacji.

Podczas implementacji systemu zostały wykorzystane następujące protokoły sieciowe: MQTT, TCP i UDP. Za pomocą protokołu MQTT system dostaje informacje o wchodzących zdarzeniach aplikacyjnych lub sam tworzy akcje wyjściowe. A protokoły TCP i UDP zostały wykorzystane w celu generacji obciążającego ruchu sieciowego.

1.5. Badania

Część badawcza tej pracy będzie skupiona na dokładności generacji ruchu sieciowego na węźle NAPES. Obserwowany będzie rzeczywisty odstęp czasowy między wysłanymi pakietami i jak on różni się od oczekiwanego odstępu [7]. Im mniejsza będzie ta różnica tym większa będzie dokładność systemu. Część badawczą tej pracy można podzielić na trzy główne części: badania wstępne, badania z różnymi parametrami przepływów oraz badania dokładności NAPES pod innymi warunkami pracy systemu.

Celem **badania wstępnego** jest sprawdzenie poprawności działania systemu, czy zaimplementowane zachowanie systemu jest zgodne z oczekiwanym. W tym eksperymencie ruch sieciowy będzie obserwowany z poziomu aplikacji obcej i porównywany z logami NAPES w celu sprawdzenia poprawności zapisywania czasów wysłanych pakietów podczas runtime. W tym badaniu wstępnym będą przedstawione główne funkcje emulatora:

- jak działa maszyna stanów,
- jak system przechodzi od jednego stanu do drugiego
- i jak zmienia przepływy (flows) przy różnych warunkach pracy.

Celem **drugiej części badawczej** będzie obserwowanie reakcji systemu na zmianę parametrów generującego się ruchu sieciowego. W tym teście do uwagi będą brane odstępy czasowe między pakietami (*inter-packet time*). Na podstawie tych odstęgów czasowych będą obliczone podstawowe miary dokładności systemu:

- Średnia,
- Wariancja,
- Odchylenie standardowe,
- Minimalny odstęp między pakietowy,
- Maksymalny odstęp między pakietowy.

Posiadając te parametry można będzie wnioskować, jak dokładny jest system i jak różne parametry wpływają na dokładność generacji ruchu sieciowego na zwykłym urządzeniu IoT.

Badania dokładności NAPES pod innymi warunkami pracy systemu - ten eksperyment ma na celu zbadać, jaki wpływ ma zmiana priorytetu wątku, przez który jest generowany przepływ sieciowy. Android API posiada narzędzia, pozwalające na zmianę priorytetów wątków na poziomie systemowym.

1.6. Wkład własny

W trakcie realizacji pracy osiągnięto następujące wyniki:

- analiza formatu RCR oraz tworzenie programu parsującego dane konfiguracyjne komponentu,
- konfiguracja i uruchomienie lokalnego brokera MQTT,
- implementacja serwerów TCP i UDP,
- implementacja klientów MQTT, TCP oraz UDP dla Android,

- implementacja obsługi portów (klienckich oraz serwerowych),
- implementacja maszyny stanów,
- implementacja obsługi kolejki zdarzeń,
- analiza formatu reprezentacji danych Trace Event Format oraz implementacja zapisywania logów zgodnie z tym formatem,
- analiza algorytmu Rate Monotonic Scheduling i wdrożenie go do działającego systemu,
- analiza i odczytywanie statystyki wyjściowych danych systemu,

Koncepcja, architektura oraz formaty danych wejściowych i wyjściowych NAPES zostały opracowane przez dr inż. Jarosława Domaszewicza i dr inż. Andrzeja Bąka [3-6]. W opracowaniu formatów danych uczestniczył R. Sztelmach, który również opracował narzędzia do translacji formatów [13].

1.7. Układ pracy

W pierwszym rozdziale są przedstawione podstawowe zagadnienia dotyczące tematu pracy. Jest opisana motywacja i cel pracy oraz są opisane główne elementy architektury systemu NAPES. Także wkrótce zostały opisane badania, które zostały wykonane w tej pracy.

Drugi rozdział porusza tematykę generatorów ruchu. W tym rozdziale są pokazane inne narzędzia służące do generacji ruchu sieciowego i jaka jest różnica między tymi narzędziami a systemem NAPES.

Trzeci rozdział przedstawia w bardziej szczegółowy sposób ogólną architekturę systemu NAPES, z jakich warstw składa się system. Także ten rozdział przedstawia zapoznanie się z formatem pliku konfiguracyjnego RCR.

W czwartym rozdziale są wykazane wszystkie technologie oraz narzędzia, które zostały użyte podczas implementacji systemu.

Piąty rozdział zawiera szczegółową informację o komponencie NAPES i jak elementy komponentu wchodzą w interakcje ze sobą.

W szóstym rozdziale pracy przedstawiona i opisana jest całkowita implementacja środowiska wykonawczego NAPES. W tej części są pokazane różne decyzje projektowe oraz poszczególne kroki, po których powstał system.

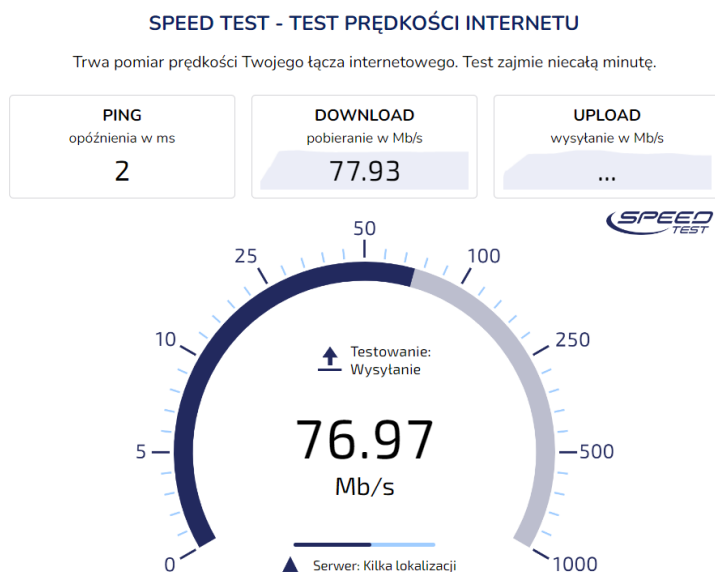
Siódmy rozdział przedstawia metody testowania oraz weryfikacji poprawności wytworzonego systemu. Są pokazane wyniki badań dla różnych parametrów komponentu.

Ósmy i dziewiąty rozdziały też dotyczą części badawczej oraz mają na celu pokazać, jak zachowuje się system przy innych warunkach pracy aplikacji.

W ostatnim rozdziale przedstawione są podsumowanie i wnioski wynikające z uzyskanych wyników, a także porusza się tematyka rozwoju systemu NAPES oraz kierunki dalszych prac.

2. Stan wiedzy

W tym rozdziale będą przedstawione istniejące narzędzia, które generują ruch sieciowy tak, jak robi to NAPES. Większość generatorów ruchu służą do pomiaru wydajności sieci lub wybranego łącza [8]. Najbardziej popularne z tych narzędzi są testy prędkości Internetu domowego, które można łatwo znaleźć w Internecie i uruchomić test z poziomu przeglądarki. Na przykład, taka aplikacja jest dostępna pod linkiem: <https://www.speedtest.pl/> [9]. Ta aplikacja mierzy trzy podstawowe parametry łącza: **PING** (wartość opóźnień przesyłania najmniejszej możliwej ilości danych na trasie klient-serwer-klient), prędkość **pobierania** danych oraz **wysyłania** danych (rys. 2.1).



Rysunek 2.1 Podstawowy generator ruchu, służący do pomiaru prędkości Internetu [9]

Ten test jest bardzo ścisły, ponieważ generowany jest przepływ o jednakowych parametrach, zdefiniowanych wcześniej na potrzeby testu. Także badane jest tylko jedno łącze pomiędzy klientem a wybranym serwerem zdalnym.

Bardziej przydatnym narzędziem dla programistów może być **lperf**, który jest bardziej elastycznym narzędziem do generacji ruchu sieciowego. Bieżąca wersja, czasami określana jako **lperf3**, obsługuje konfigurowanie różnych ustawień związanych z synchronizacją, protokołami (*można ustalić rozmiar wysyłającego pakietu*) i buforami. Dla każdego testu raportuje przepustowość, straty w sieci i inne parametry. Także główną cechą **lperf** jest to, że istnieje możliwość uruchomienia serwera na wybranym hoście (rys 2.2 i 2.3). To pozwala na badania przepustowości łącza w jednej wybranej sieci, czy to w prywatnej czy w lokalnej [10].

```

root@192.168.1.149
[root@mini:~] /usr/lib/vmware/vsan/bin/iperf -m -i t300 -c nuc -fm
WARNING: interval too small, increasing from 0.00 to 0.5 seconds.
Client connecting to nuc, TCP port 5001
TCP window size: 0.03 MByte (default)
-----
[ 3] local 192.168.1.149 port 46678 connected with 192.168.1.50 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3] 0.0- 0.5 sec  56.1 MBytes 112 MBytes/sec
[ 3] 0.5- 1.0 sec  56.0 MBytes 112 MBytes/sec
[ 3] 1.0- 1.5 sec  56.0 MBytes 112 MBytes/sec
[ 3] 1.5- 2.0 sec  56.0 MBytes 112 MBytes/sec
[ 3] 2.0- 2.5 sec  55.9 MBytes 112 MBytes/sec
[ 3] 2.5- 3.0 sec  56.1 MBytes 112 MBytes/sec
[ 3] 3.0- 3.5 sec  55.9 MBytes 112 MBytes/sec
[ 3] 3.5- 4.0 sec  55.9 MBytes 112 MBytes/sec
[ 3] 4.0- 4.5 sec  55.0 MBytes 110 MBytes/sec
[ 3] 4.5- 5.0 sec  56.0 MBytes 112 MBytes/sec
[ 3] 5.0- 5.5 sec  55.9 MBytes 112 MBytes/sec
[ 3] 5.5- 6.0 sec  56.1 MBytes 112 MBytes/sec
[ 3] 6.0- 6.5 sec  56.0 MBytes 112 MBytes/sec
[ 3] 6.5- 7.0 sec  56.0 MBytes 112 MBytes/sec
[ 3] 7.0- 7.5 sec  56.0 MBytes 112 MBytes/sec
[ 3] 7.5- 8.0 sec  56.0 MBytes 112 MBytes/sec
[ 3] 8.0- 8.5 sec  56.1 MBytes 112 MBytes/sec
[ 3] 8.5- 9.0 sec  55.9 MBytes 112 MBytes/sec
[ 3] 9.0- 9.5 sec  56.1 MBytes 112 MBytes/sec
[ 3] 9.5-10.0 sec  56.0 MBytes 112 MBytes/sec
[ 3] 0.0-10.0 sec 1119 MBytes 112 MBytes/sec
[ 3] MSS size 1448 bytes (MTU 1500 bytes, ethernet)
[root@mini:~]

```

Rysunek 2.2 Widok terminali z klienta Iperf

```

root@nuc
[root@nuc:~] /usr/lib/vmware/vsan/bin/iperf, copy -s -0 192.168.1.50
Server listening on TCP port 5001
Binding to local address 192.168.1.50
TCP window size: 64.0 KByte (default)
-----
[ 4] local 192.168.1.50 port 5001 connected with 192.168.1.149 port 46678
[ ID] Interval      Transfer    Bandwidth
[ 4] 0.0-10.0 sec  1.09 GBytes 938 Mbits/sec
[  ]

```

Rysunek 2.3 Widok z terminali serwera Iperf

Jak widać, na jednej maszynie jest uruchomiony serwer a na drugiej klient, który generuje sparametryzowany przepływ sieciowy. Zasada działania systemu NAPES jest bardzo podobna (rys. 2.4 - 2.5). Istotną różnicą jest to, że NAPES jest bardziej złożonym generatorem ruchu, który obsługuje maszynę stanów (tak jak na zwykłym węźle IoT) [11].

W Internecie można znaleźć wiele rozwiązań, jak komercyjnych tak i niekomercyjnych, które generują ruch sieciowy w warunkach aplikacji rzeczywistych. Są to raczej bardzo ściśle narzędzia, ponieważ większość z takich rozwiązań emulują ruch sieciowy tylko dla jednej wybranej aplikacji rzeczywistej [12], a NAPES Runtime ma na celu emulację pracy aplikacji wielokomponentowej o złożonej logice.



Rysunek 2.4 Widok klienta z aplikacji NAPES



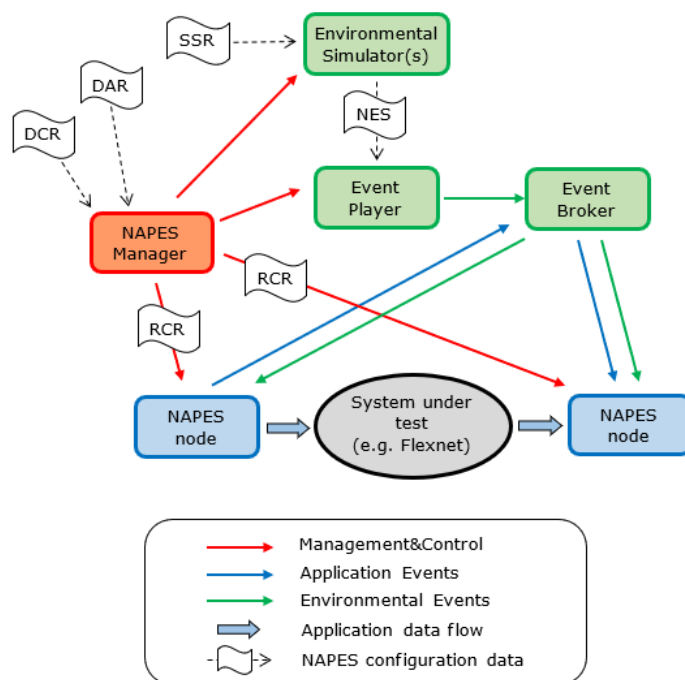
Rysunek 2.5 Widok serwera z aplikacji NAPES

3. System NAPES

NAPES jest system przeznaczonym do wydajnego testowania sieci FLEXNET [4]. Główną jednostką tego systemu jest środowisko wykonawcze, które ma za zadanie generować obciążający ruch sieciowy. Środowisko wykonawcze interpretuje dane komponentu, konfiguracja którego jest zdefiniowana w plikach o formacie RCR. Komponenty komunikują się ze sobą poprzez przepływy i są określane przez maszyny stanów. Mogą również reagować na zdarzenia aplikacyjne lub lokalne. NAPES ma na celu umożliwienie szybkiego testowania sieci bez znacznych inwestycji w rozwój i wdrażanie aplikacji rzeczywistych. Przewidywano jest, że NAPES może być również przydatny dla programistów lub integratorów systemów lub aplikacji, aby szybko sprawdzić, czy dostępna infrastruktura sieciowa jest wystarczająca dla ich systemu lub aplikacji.

3.1. Ogólna architektura systemu NAPES

Rysunek 3.1 przedstawia ekosystem NAPES, który składa się z węzłów NAPES, menedżera NAPES, symulatorów środowiskowych oraz brokera zdarzeń. Węzły NAPES służą do emulowania zachowania komponentów aplikacji IoT przy użyciu środowiska uruchomieniowego NAPES. Menedżer NAPES zarządza systemem i kontroluje sesję emulacji, a symulatory środowiskowe generują zdarzenia środowiskowe, które symulują sygnały czujników odbierane przez komponenty. Odtwarzacz zdarzeń planuje i wysyła zdarzenia środowiskowe do brokera zdarzeń, który następnie przesyła je do komponentów, które je zasubskrybowały.



Rysunek 3.1 Architektura systemu NAPES [4]

Architektura systemu NAPES składa się z trzech głównych warstw:

- warstwa interfejsu użytkownika, która zawiera elementy wykorzystywane przez użytkownika do tworzenia danych konfiguracyjnych sesji emulacji lub do

przetwarzania dzienników i statystyk generowanych podczas emulacji (przechowywane w rejestrze dziennika). Użytkownik określa ustawienia emulatora NAPES, podając następujące dane konfiguracyjne: Design-time Component Representation (DCR), Design-time Application Representation (DAR), Simulated Session Representation (SSR), zbiór węzłów NAPES, na których można emulować aplikację.

- warstwa translacji i symulacji, która implementuje elementy używane do tworzenia danych wejściowych w czasie wykonywania dla NAPES lub gromadzenia wyjściowych statystyk i dzienników. Dane wykonawcze obejmują o Runtime Component Representation (RCR), NAPES Event Script (NES).
- warstwa środowiska wykonawczego, która implementuje elementy używane podczas sesji emulacji, w tym środowisko wykonawcze NAPES, odtwarzacz zdarzeń i broker zdarzeń.

3.2. Format RCR: składnia i semantyka

Kluczowym zadaniem środowiska wykonawczego jest interpretacja struktury danych opisującej zachowanie komponentu działającego na węźle. Struktura danych jest nazywana „runtime component representation” (RCR). RCR reprezentuje konfigurację poszczególnych komponentów, które mają być uruchamiane przez węzeł NAPES przy użyciu środowiska wykonawczego NAPES. Dane te są tworzone z reprezentacji DCR i DAR i zawierają wszystkie informacje wymagane przez środowisko wykonawcze do utworzenia wystąpienia i uruchomienia komponentu emulowanej aplikacji [13].

Reprezentacja konfiguracji poszczególnych komponentów w plikach RCR jest realizowana i opisana za pomocą składni EBNF (extended Backus–Naur form). Gramatykę i poprawność zapisu pliku można zweryfikować przez: <https://bnfplayground.pauliankline.com/>. [14]. Składnia ta pozwala na zapis poszczególnych elementów oraz stanów systemu do pliku, rozmiar, którego jest bardzo mały w rezultacie końcowym. Gramatyka EBNF nie pozwala na wykorzystania spacji i tabulacji. Plik RCR można stworzyć ręcznie za pomocą zwykłego edytora tekstowego i zapisać go w postaci czytelnej (z ładnym formatowaniem). Żeby wykryć błędy w takim pliku, należy przepuścić plik przez filtr usuwający spacje i wkleić go zawartość do strony, link do której jest podany wyżej (jest to porównywalne z wykrywaniem błędów podczas kompilacji). Parser w środowisku wykonawczym NAPES powinien zezwalać na spacje i usuwać je podczas wykonywania programu.

Poniżej jest przykładowo przedstawiono ułamek semantyki RCR:

```
1 /* flows (named and anonymous) */
2
3 <F_object>                ::= "{" "F" ";" <flows> "}"
4 <flows>                    ::= <flow_named> | "[" <flow_named> ("," <flow_named>)+ "]"
5 <flow_named>               ::= "{" <f_name> ";" <flow_anonymous> "}"
6 <flow_anonymous>          ::= "{" <f_type> ";" <f_parameter>* "}"
7 <f_type>                   ::= "simple" | "on_off"
8 <f_parameter>             ::= <parameter>
9 <flow_instance>          ::= <f_name> | <flow_anonymous>
```

Rysunek 3.2 Przykładowy ułamek pliku RCR

Na tym fragmencie przedstawiona jest definicja zapisywania różnych parametrów (szczególnie definicja parametrów przepływu) do ich interpretacji przez środowisko wykonawcze.

4. Wykorzystane API, protokoły i narzędzia

4.1. Android, Java

Android jest używany na wielu różnych urządzeniach. Są to smartfony, tablety, telewizory, inteligentne zegarki i szereg innych gadżetów. Więc tworzenie aplikacji dla systemu Android jest perspektywiczne.

Można tworzyć aplikacje na Androida przy użyciu różnych frameworków i języków programowania. Tak więc, Java, Kotlin, Dart (framework Flutter), C ++, Python, C # (platforma Xamarin) itp. mogą być używane jako języki programowania. W tym przypadku został wybrany język Java, który jest jednym z najbardziej popularnych języków używanych w tym celu.

Istnieją różne środowiska programistyczne dla Androida. Zalecanym środowiskiem programistycznym jest Android Studio, które jest specjalnie zaprojektowane do programowania na Androida. Można pobrać plik instalatora z oficjalnej strony internetowej: <https://developer.android.com/studio> [15].

Oprócz samego środowiska Android Studio programowanie będzie wymagało również zestawu narzędzi zwanych Android SDK. Na przykład, jeśli Android SDK nie był wcześniej zainstalowany, to przy pierwszym kontakcie z Android Studio będzie komunikat, że nie ma Android SDK. Można ręcznie pobrać Android SDK z oficjalnej strony i zainstalować go, lub można to zrobić bezpośrednio z Android Studio.

W celu tworzenia oraz rozwoju środowiska wykonawczego został wybrany język programowania Java, który pozwala na tworzenie takiej aplikacji dla platformy Android. Zaletą tego języka jest to, że istnieje wiele usług oraz bibliotek, pozwalających na pracę z systemem Android i protokołem MQTT. Także dobrym aspektem tego języka jest to, że on pozwala na *wielowątkowość*, co bardzo się przyda w trakcie rozwoju aplikacji.

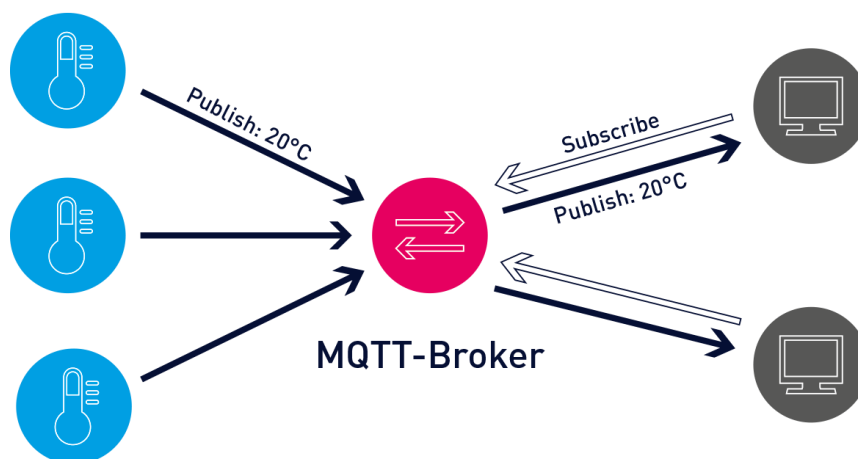
Do uruchomienia i przetestowania aplikacji można wykorzystać *emulatory* lub prawdziwe urządzenia. Najlepiej jednak testować na prawdziwych urządzeniach. Ponadto emulatory wymagają dużych zasobów sprzętowych, a nie każdy komputer poradzi sobie z wymaganiami emulatorów. Aby przetestować urządzenie mobilne, może być konieczne zainstalowanie tylko niezbędnego sterownika.

W zakresie potrzebnym do realizacji środowiska NAPES Runtime, ważnym elementem pracy jest wysyłanie i odbieranie pakietów za pomocą portów. Dla umożliwienia pracy z tymi komponentami można użyć dwóch głównych bibliotek, które pozwalają na to, nawet dla systemów Android: **java.net.DatagramPacket**, **java.net.DatagramSocket**. Po realizacji potrzebnych interfejsów z tych bibliotek, można wysyłać i odbierać wiadomości na konkretnych portach różnych urządzeń [16].

4.2. Protokół MQTT

Jeszcze jednym ważnym składnikiem potrzebnym do zapoznania i do realizacji emulatora jest protokół MQTT. MQTT jest protokołem do przesyłania danych między urządzeniami z ograniczoną mocą procesora i żywotnością baterii oraz dla sieci o drogiej lub niskiej przepustowości, nieprzewidywalnej stabilności lub dużym opóźnieniu. Dlatego MQTT jest znany jako idealny transport dla IoT [17]. Jest zbudowany na protokole TCP/IP, ale istnieją wersje protokołu do pracy przez *Bluetooth*, *UDP* i inne sieci *IoT*.

System komunikacyjny oparty na MQTT składa się z wydawcy, serwera brokera i subskrybenta. Wydawca nie wymaga żadnych korekt dotyczących liczby lub lokalizacji subskrybentów otrzymujących wiadomości. Ponadto subskrybenci nie muszą być konfigurowani dla konkretnego wydawcy. W systemie może być wiele brokerów, które rozpowszechniają komunikaty. Na poniższym rysunku pokazano schemat działania protokołu MQTT:



Rysunek 4.1 Schemat działania protokołu MQTT [18]

MQTT umożliwia przesyłanie wiadomości pomiędzy urządzeniami w sieci, zwłaszcza w środowiskach Internetu Rzeczy (IoT). Jest to protokół komunikacyjny oparty na modelu publikacji/subskrypcji, co oznacza, że urządzenia mogą publikować wiadomości na określonych kanałach, a inne urządzenia mogą subskrybować te kanały i odbierać wiadomości. MQTT jest lekki, prosty w użyciu i wymaga minimalnego zużycia zasobów, co czyni go popularnym wyborem w IoT.

Do zaimplementowania aplikacji emulującej należało użyć protokołu MQTT. W tym celu można wybrać dowolny MQTT-broker serwer, który będzie znajdować się w chmurze i dostęp do tego brokera będzie możliwy na dowolnym węźle klienckim. W Internecie istnieje wiele takich serwisów, które pozwalają zdefiniować swojego brokera (darmowych i płatnych), na przykład: <https://test.mosquitto.org/> [19].

4.3. MQTT Broker Mosquitto

Jak już wiadomo, system komunikacyjny oparty na MQTT składa się z co najmniej jednego serwera (często nazywanym brokerem MQTT). W tej pracy, broker MQTT będzie zainstalowany i uruchomiony na komputerze lokalnym. Po uruchomieniu brokera, można będzie sprawdzić, czy wszystko działa poprawnie, tworząc klienta (subskrybenta) MQTT w środowisku programistycznym Android Studio.

Dla celów tej pracy został wybrany jeden z najbardziej popularnych zasobów – broker „Mosquitto” [20]. Także w celu testowania brokera zostały użyte istniejące aplikacje klienckie, pozwalające na połączenie z brokerem:

- MQTT.fx – desktopowy klient,
- MQTTTool – aplikacja dla urządzeń mobilnych.

4.4. MQTT Paho

Dla tego, żeby stworzyć klienta MQTT dla systemu Android, należy dołączyć do projektu bibliotekę, pozwalającą na to. W tym celu została wybrana jedna z najczęściej używanych bibliotek Eclipse Paho. Ta biblioteka obsługuje wszystkie szczegóły protokołu niskopoziomowego, co pozwala skupić się na innych aspektach aplikacji, pozostawiając jednocześnie dużo miejsca na dostosowanie ważnych aspektów jego wewnętrznej funkcjonalności, takich jak zapis wiadomości [21].

Biblioteka Paho umożliwia tworzenie klienta MQTT. Żeby to zrobić należy realizować interfejs *IMqttClient*. Ten interfejs zawiera wszystkie metody potrzebne aplikacji do nawiązania połączenia z serwerem, wysyłania i odbierania wiadomości. Na poniższym rysunku jest pokazano tworzenie klienta oraz go połączenie z brokerem:

```
String publisherId = UUID.randomUUID().toString();  
IMqttClient publisher = new MqttClient("tcp://iot.eclipse.org:1883",publisherId);
```

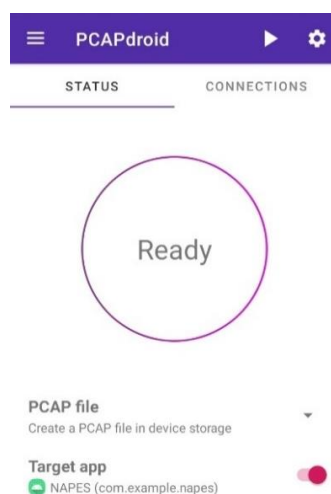
Rysunek 4.2 Tworzenie obiektu klienta poprzez łączenia klienta do brokera MQTT

Posiadając obiekt klasy *MqttClient* można wysłać oraz odbierać wiadomości za pomocą funkcji zaimplementowanych w tej klasie. Dla tego żeby wysłać wiadomość korzystamy z funkcji *publish()*, natomiast dla odbioru wiadomości stosujemy funkcję *subscribe()*.

4.5. Narzędzia do analizy ruchu sieciowego

PCAPdroid to aplikacja typu open source, która umożliwia śledzenie, analizowanie i blokowanie połączeń nawiązywanych przez inne aplikacje na urządzeniu. Pozwala także eksportować zrzut ruchu PCAP, sprawdzać HTTP, odszyfrowywać ruch TLS i wiele więcej [22].

Podczas emulacji NAPES Runtime ruch sieciowy jest zapisywany za pomocą PCAPdroid, a następnie ten ruch jest analizowany za pomocą Wireshark, który także służy do analizy przechwyconego ruchu sieciowego.



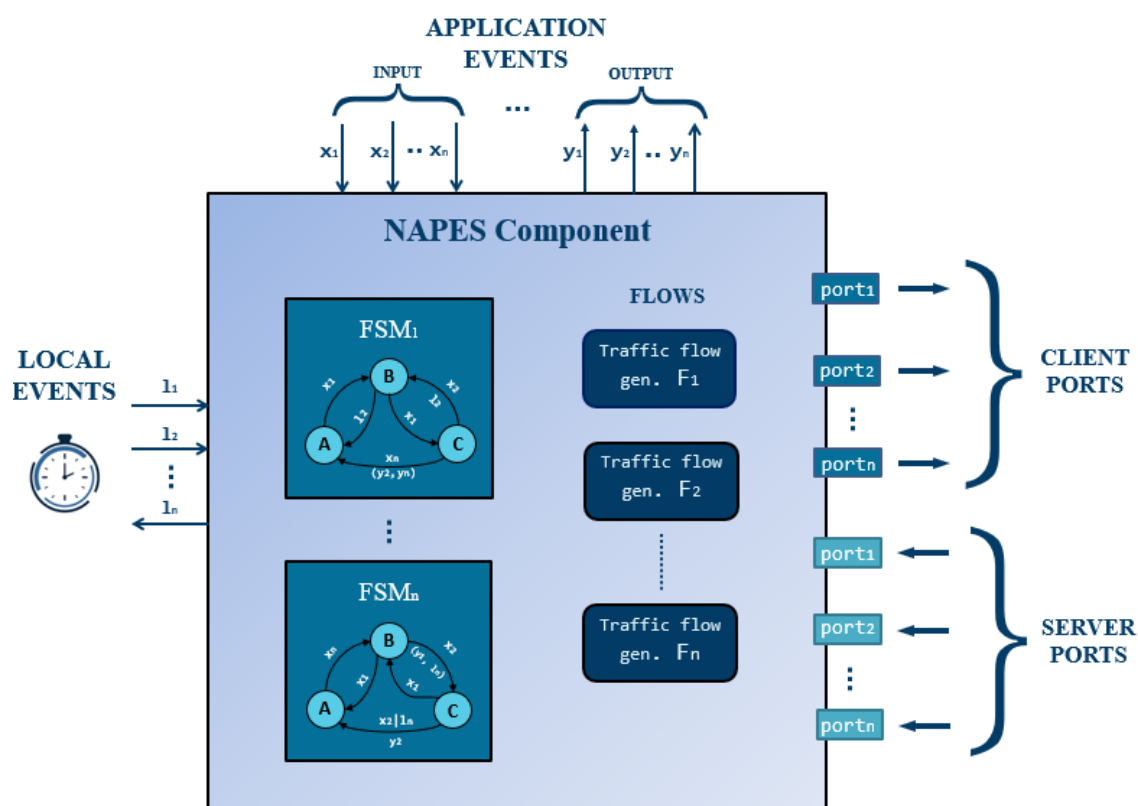
Rysunek 4.3 Widok głównego menu PCAPdroid

5. Model komponentu NAPES Runtime dla systemu Android

Komponent jest podstawową jednostką systemu NAPES [4], w którym jest opisana konfiguracja zachowania poszczególnych elementów:

- zdarzenia (lokalne i środowiskowe),
- maszyna stanów,
- przepływy,
- porty.

Konfiguracja komponentu powinna znajdować się w plikach o rozszerzeniu RCR (*Runtime Component Representation*). Środowisko wykonawcze interpretuje konfiguracje każdego komponentu w oddzielnym wątku aplikacji. Taka decyzja projektowa pozwala na uruchomienie wielu komponentów na jednym węźle Android. Na rys. 5.1 przedstawiony jest model komponentu NAPES.



Rysunek 5.1 Model komponentu NAPES

Implementacja takiego modelu wymaga wielowątkowości w aplikacji, która będzie dynamiczna z czasem (aplikacja może tworzyć oraz uruchamiać wątki podczas runtime'a). Architekturę komponentu można podzielić na dwie główne warstwy:

- **Warstwa obsługi portów** decyduje o tym, jaki generator przepływu musi być uruchomiony w zależności od tego w jakim stanie znajduje się maszyna stanów. Ta warstwa ma za zadanie robić walidację parametrów przepływu po każdym przejściu między stanami.
- **Warstwa obsługi maszyny stanów** posiada zadeklarowane w sobie różne stany, ale posiada tylko jeden *stan aktualny* (ten w którym aktualnie się znajduje maszyna stanów). Po uruchomieniu emulacji, warstwa obsługi FSM ma za zadanie ustawić aktualny stan na stan początkowy, który jest

zadeklarowany w komponencie. Natomiast głównym zadaniem tej warstwy jest walidacja zdarzeń wchodzących i ewentualna, zmienia aktualnego stanu FSM, jeżeli spełniają się określone warunki.

W modelu na rys. 5.1 są pokazane wszystkie składniki dowolnego komponentu, które mają wpływ na logikę działania emulatora i są wykorzystywane podczas symulacji. Głównym zadaniem interpretacji komponentu na środowisku wykonawczym jest uruchomienie maszyny stanów i obsługa zdefiniowanych portów.

Każdy port ma zdefiniowane w sobie reguły, w których każdy stan maszyny może mieć przypisany do siebie dowolny generator przepływu. W definicjach parametrów komponentu takie reguły mają nazwę „StateFlow”. Na poniższym listingu znajduje się deklaracja przykładowego portu wraz z jego regułami.

Listing 5.1 Przykładowa deklaracja portu wraz z jego regułami

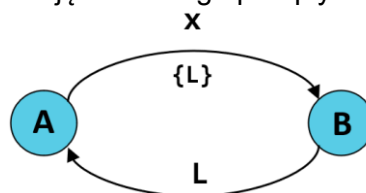
```
{port1; c; U; {;8011}; {0.0.0.0;8002}; FSM1; [{A; F1}, {B; F2}]}
```

Powyższy fragment deklaracji ma na celu pokazać, jak są ustawiane reguły na dowolnych portach. Dwa ostatnie parametry odpowiadają za interakcję warstwy obsługi portów z warstwą obsługi maszyny stanów. W tym przykładzie **port1** monitoruje aktualny stan maszyny **FSM1**. Ostatnim parametrem w tej definicji jest lista obiektów „StateFlow”, która definiuje generację przepływu **F1** w stanie **A** i generację przepływu **F2** w stanie **B**.

Nie mniej ważnym składnikiem tego modelu są zdarzenia, które mogą być *wejściowe* i *wyjściowe*. Za pomocą zdarzeń wejściowych realizowane są przejścia między kolejnymi stanami. A zdarzenia wyjściowe – są to raczej akcje, które powinien wykonać komponent w trakcie zmiany stanu na FSM.

Komponent NAPES rozróżnia dwa typy zdarzeń: lokalne i aplikacyjne. **Zdarzenia aplikacyjne** są realizowane za pomocą protokołu MQTT. Wysyłanie wiadomości MQTT na „topic”, który jest subskrybowany przez dowolny inny komponent powoduje dla niego obsługę zdarzenia wejściowego. Dla przykładu: po przejściu między stanami w FSM, komponent może wykonać akcję, wysyłając wiadomość MQTT na podany „topic”, a komponenty, które ten „topic” subskrybują otrzymają zdarzenie wejściowe. W taki sposób komponenty mogą komunikować się między sobą i reagować na jakiegokolwiek zmiany w całym ekosystemie NAPES.

Zdarzenia lokalne wykonują funkcję timera, który może zostać uruchomiony tylko po przejściu między stanami, wykonując odpowiednie akcje. Na rysunku 5.2 jest przedstawiony przykład działania zdarzeń lokalnych. Jak widać dla stanu **A** warunkiem przejścia do stanu **B** jest **X**. Gdy spełnia się ten warunek – wykonuje się akcja **L**, która uruchamia timer na podany czas. Po upływie czasu przychodzi to samo zdarzenie **L**, które jest warunkiem dla przejścia z powrotem do stanu **A**. Taka implementacja pozwala na zmianę stanu, a z tym i na generację dowolnego przepływu określony czas.



Rysunek 5.2 Przykład działania zdarzeń lokalnych

6. Implementacja komponentu NAPES Runtime dla systemu Android

6.1. Instalacja obsługi protokołów

6.1.1. Instalacja oraz uruchomienie brokera MQTT

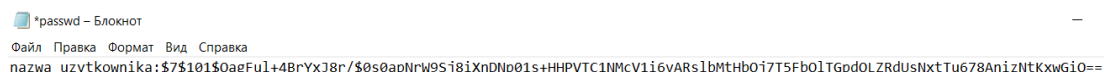
Można go ściągnąć z oficjalnej strony, która znajduje się pod linkiem: <https://mosquitto.org/download/> [20]. Także, do poprawnej instalacji oraz uruchomienia tej technologii niezbędnym jest instalacja zasobu Microsoft Visual C++ 2019.

Po instalacji tych składników można przejść do konfiguracji brokera. W pierwszym etapie konfiguracji, zostały stworzone nazwa użytkownika oraz jego hasło. Jeśli domyślne broker Mosquitto został zainstalowany na dysku C, można to zrobić za pomocą poniższego polecenia w terminali:

```
C:\>"\Program Files\mosquitto\mosquitto_passwd" -c "C:\Program Files\mosquitto\passwd" nazwa_uzytkownika
Password:
Reenter password: _
```

Rysunek 6.1 Tworzenie nazwy użytkownika oraz jego hasło

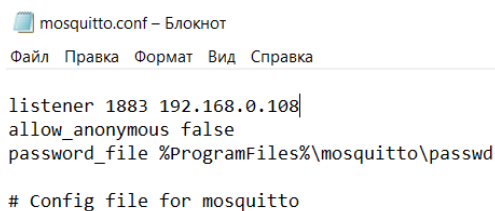
Patrząc na to polecenie, można zobaczyć, że najpierw podawana jest ścieżka do zasobu, który pomaga stworzyć plik z danymi użytkownika. W parametrze „-c” podawana jest ścieżka, gdzie plik z danymi uwierzytelnienia będzie zapisany. Na poniższym rysunku jest przykładowo pokazano zawartość tego pliku:



```
*passwd - Блокнот
Файл  Правка  Формат  Вид  Справка
nazwa_uzytkownika:$7$101$QagFul+4BrYxJ8r/$0s0apNrW9Sj8iXnDNp01s+HHPVTC1NMcv1i6vARslbmthbQj7T5FbQlTgpdQLZRdUsNxtTu678AnizNtKxwGiQ==
```

Rysunek 6.2 Zawartość pliku wygenerowanego pliku „passwd”, zawierającego dane uwierzytelniające do brokera

Następnym etapem konfiguracji jest wpisanie odpowiednich reguł do pliku „mosquitto.conf”. Ten plik jest specjalnie przeznaczony do konfiguracji brokera. Reguły wpisane do tego pliku, pomagają zabezpieczyć się przed anonimowymi połączeniami do brokera oraz można ustawiać w jaki sposób będzie on działać. Zawartość tego pliku jest pokazana na poniższym rysunku:



```
mosquitto.conf - Блокнот
Файл  Правка  Формат  Вид  Справка

listener 1883 192.168.0.108|
allow_anonymous false
password_file %ProgramFiles%\mosquitto\passwd

# Config file for mosquitto
..
```

Rysunek 6.3 Zawartość pliku „mosquitto.conf”

Jak widać, w pierwszej linijce tego pliku jest ustawiono, że broker nasłuchuje na porcie 1883 pod adresem sieci lokalnej 192.168.0.108. W tym punkcie należałoby ustawić adres komputera IPv4 na statyczny, ponieważ za każdym uruchomieniem komputera serwis DHCP może przydzielać inny adres, co powoduje, że potrzebnym będzie za każdym razem zmieniać adres w pliku konfiguracyjnym. W następnej linijce została dodana reguła,

która nie pozwala na połączenie z brokerem bez danych uwierzytelniających. A w kolejnej linijce wskazano jest, gdzie znajduje się plik z nazwą użytkownika oraz jego hasłem.

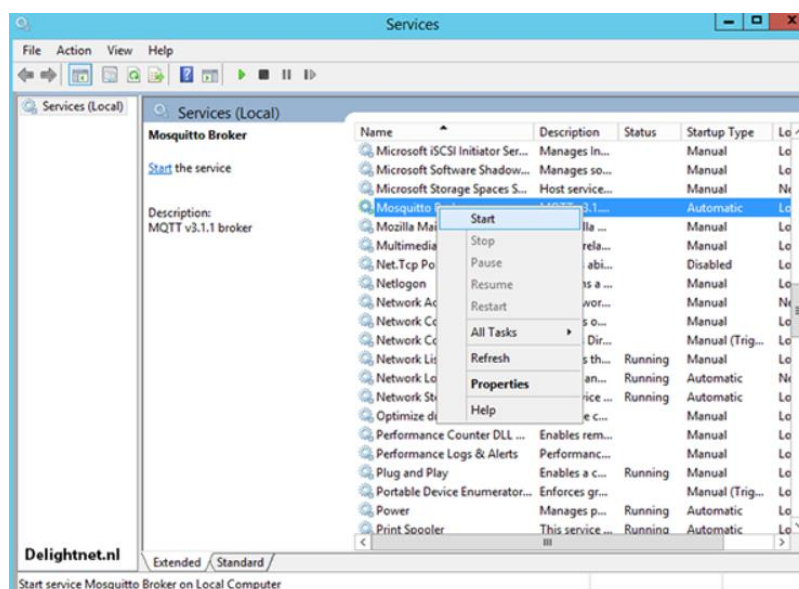
W kolejnym kroku konfiguracji potrzebnym jest tworzenie reguły zapory w zaporze systemu Windows. Pozwoliłoby to na połączenie innych urządzeń w sieci lokalnej do portu 1883. Można to zrobić za pomocą poniższego polecenia:

```
C:\WINDOWS\system32>netsh advfirewall firewall add rule name="MQTTBroker" protocol=TCP dir=in localport=1883 action=allow  
Ok.
```

Rysunek 6.4 Polecenie służące do tworzenia reguły zapory

W wyniku tego polecenia otrzymywany jest komunikat „OK”, który świadczy o tym, że wszystko zostało dodane poprawnie. Także poprawność dodania tej reguły można sprawdzić w systemie operacyjnej, gdzie także można zmieniać niektóre parametry we właściwościach reguły.

Po wykonaniu powyższych kroków konfiguracji, można sprawdzić, czy wszystko działa poprawnie. W tym celu należy uruchomić usługę „Mosquitto Broker” (rys. 6.5) w systemie Windows.



Rysunek 6.5 Uruchomienie usługi Mosquitto Broker

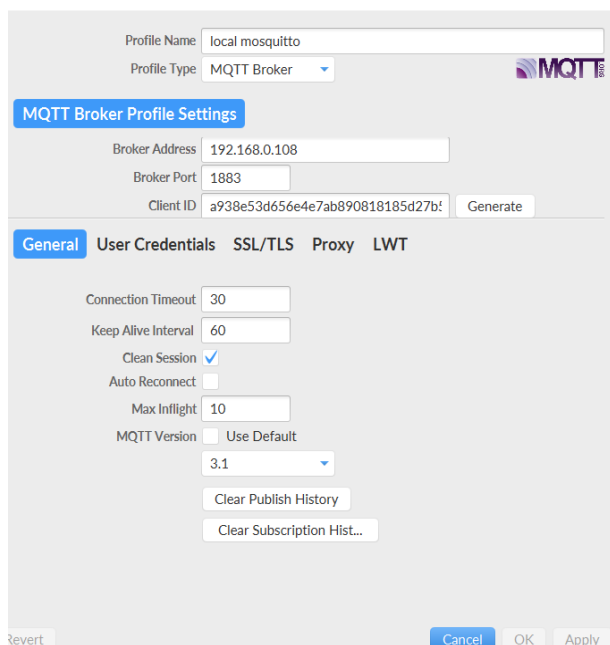
W tym menu, także można ustawić niektóre parametry dla odpowiedniej usługi. W tym przypadku został zmieniony parametr uruchomienia usługi, żeby ona włączała się ręcznie, a nie automatycznie przy włączeniu komputera (nie jest to niezbędnym punktem do wykonania, więc można to opuścić).

Do sprawdzenia poprawności działania brokera, będą użyte dwie programy umożliwiające połączenia z brokerem MQTT. Jeden program będzie uruchomiony na komputerze, na którym jest uruchomiony broker, a drugi będzie uruchomiony na dowolnym innym urządzeniu połączonym do sieci lokalnej (w tym przypadku będzie to urządzenia mobilne iPhone).

Na komputerze został uruchomiony program MQTT.fx, za pomocą którego można połączyć się z brokerem. W tym programie należy podać parametry niezbędne do połączenia z brokerem (rys. 6.6).

Na urządzeniu mobilnym został zainstalowany program MQTTTool, który też pozwala na pracę z brokerem. Żeby nawiązać połączenia należy podać te same parametry, jak i w wersji desktopowej.

Gdy zostały nawiązane połączenia na obu urządzeniach, można sprawdzić poprawność działania serwera. W tym celu na komputerze będzie subskrybowany topik „test/”, a na urządzeniu mobilnym zostanie wysłana dowolna wiadomość na ten topik (rys. 6.7).



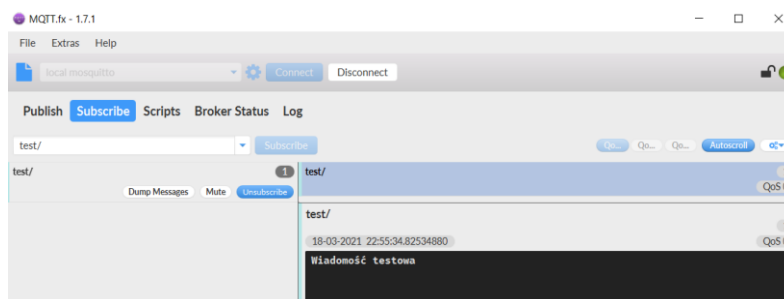
Rysunek 6.6 Widok okna konfiguracji programu MQTT.fx



Rysunek 6.7 Wysłanie wiadomości na topik „test/”

Jak widać, żeby umożliwić połączenie z brokerem potrzebnym jest wpisanie adresu komputera, na którym jest uruchomiony broker, port, na którym broker nasłuchuje oraz podać dane uwierzytelniające. Po wprowadzeniu tych parametrów można próbować połączyć się z brokerem. Sprawdzić, czy połączenie zostało nawiązane, można, patrząc na logi.

Po wysłaniu wiadomości został otrzymany komunikat, że wszystko zostało wysłano prawidłowo. I w następnym kroku, można sprawdzić, czy została ta wiadomość doszła do subskrybenta, czyli do komputera (rys. 6.8).



Rysunek 6.8 Widok ze strony subskrybenta

Po wykonaniu tych operacji, można zobaczyć, że wiadomość została odebrana poprawnie i można ją odczytać. To świadczy o tym, że broker został poprawnie skonfigurowany i uruchomiony, i wszystko działa poprawnie.

6.1.2. Tworzenie klienta MQTT w Android Studio

Po uruchomieniu brokera w sieci lokalnej, można przystąpić do tworzenia klienta MQTT na urządzeniach Android. W tym celu zostało wykorzystane narzędzie Android Studio jako środowisko programistyczne.

W pierwszym etapie tworzenia klienta, należy stworzyć nowy projekt w Android Studio. Następnie, za pomocą technologii Gradle należy podłączyć bibliotekę pozwalającą na pracę z MQTT. W tym przypadku została wybrana biblioteka Eclipse Paho, pozwalająca na pracę z protokołem MQTT. Dołączyć tę bibliotekę można poprzez dodawanie linii kodu do pliku „build.gradle”:

Listing 6.1 Dodanie biblioteki Paho do projektu

```
1 repositories {
2     maven {
3         url "https://repo.eclipse.org/content/repositories/paho-snapshots/"
4     }
5     google()
6     jcenter()
7 }
```

Po dołączeniu biblioteki do projektu, można zacząć tworzyć klienta MQTT, za pomocą interfejsu oraz klas, znajdujących się w bibliotece paho. W tym celu należy zaimplementować interfejs „MqttCallback” i realizować trzy metody tego interfejsu: „messageArrived()”, „deliveryComplete()” oraz „connectionLost()”. Jak wynika z nazw tych metod, można dowiedzieć się za co odpowiada każda metoda kolejno. Po implementacji, można przystąpić do tworzenia samego klienta i ustawianie parametrów niezbędnych na połączenia z brokerem (list. 6.2) [21].

Listing 6.2 Konfiguracja klienta MQTT w metodzie onCreate()

```
1 // tworzenie obiektu klienta MQTT
2 client = new MqttClient("tcp://" + ipAddressBroker + ":" + mqttPort,
3                        MqttAsyncClient.generateClientId(),
4                        new MemoryPersistence());
5
6 // Za pomocą MqttConnectOptions można ustawiać parametry połączeniowe
7 MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();
8 mqttConnectOptions.setUserName("username");
9 mqttConnectOptions.setPassword("passwd".toCharArray());
10
11 // ustawienie klasy callback
12 client.setCallback(this);
13
14 //nawiązanie połączenia
15 client.connect(mqttConnectOptions);
```

Jak widać, tworzony jest obiekt klasy „MqttClient”, do konstruktora, którego jest podawany adres wraz z portem maszyny, na której jest uruchomiony broker. Następnie jest tworzony obiekt klasy „MqttConnectOptions”, za pomocą którego zostały wpisane dodatkowe parametry dla połączenia, takie jak nazwa użytkownika oraz hasło. W 33 linijce kodu jest bezpośrednio wywołana funkcja, która tworzy połączenie urządzenia Android z

brokerem. Poniżej tej linii można zobaczyć, że domyślne - to urządzenie staje się subskrybentem topiku „test/”.

Jeszcze jedną rzecz, którą należy zrobić przed badaniem klienta, to w pliku „AndroidManifest.xml” dodać dwie linie, które są pokazane na list. 6.3. Te linijki pozwalają aplikacji na dostęp do Internetu oraz do sieci lokalnej.

Listing 6.3 Niezbędne linijki konfiguracji do pliku „AndroidManifest.xml”

```
1 <uses-permission android:name="android.permission.INTERNET" />
2 <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

W kolejnym kroku można przejść do sprawdzenia poprawności działania stworzonego klienta, wysyłając testowe wiadomości na topik „test/”. Aplikacja została uruchomiona na emulatorze urządzenia Android:



Rysunek 6.9 Badanie aplikacji klienckiej

Jak widać wszystkie wiadomości testowe zostały otrzymane i wyświetlone przez aplikację kliencką. To znaczy, że udało się poprawnie skonfigurować klienta MQTT, i w następnych etapach tworzenia środowiska wykonawczego, ten klient będzie bardziej rozbudowany i złożony.

6.1.3. Tworzenie oraz uruchomienie serwera UDP

Dla umożliwienia komunikacji za pośrednictwem protokołu UDP, niezbędnym jest stworzenie serwera oraz klienta, które będą komunikować się między sobą. Protokół UDP (User Datagram Protocol) ma bardzo mało reguł, więc przyspiesza komunikację, ale nie gwarantuje dokładności przesyłanych wiadomości. Gdy priorytetem jest szybka komunikacja, a nie dokładność, należy wybrać UDP jako główny protokół przeznaczony do komunikacji [23]. W tym podrozdziale będzie przedstawiono, jak można zaimplementować własny serwer UDP, za pomocą języka Java oraz środowiska programistycznego IntelliJ IDEA.

Do implementacji serwera UDP został stworzony nowy projekt oraz została stworzona klasa Java, która dziedziczy z klasy Thread. Na poniższym rysunku jest pokazany kod serwera UDP:

Listing 6.4 Fragment kodu serwera UDP

```

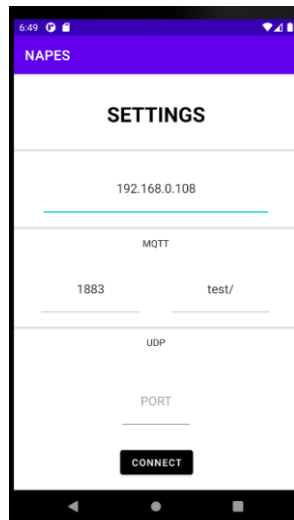
1  @Override
2  public void run() {
3      // alokacja pamięci dla przychodzącego pakietu
4      byte[] buf = new byte[1024];
5      DatagramPacket packet = new DatagramPacket(buf, buf.length);
6
7      // pętla działająca podczas trwania symulacji
8      while(Config.simulating){
9          try {
10             // w tym miejscu działanie wątku zatrzymuje się
11             // do póki serwer nie otrzyma pakietu
12             socket.receive(packet);
13
14             // zapisywanie czasu otrzymania wiadomości do listy
15             times.add(System.currentTimeMillis());
16
17             // uzyskiwanie danych klienta
18             InetAddress address = packet.getAddress();
19             int port = packet.getPort();
20
21             // wyświetlanie informacji
22             System.out.println("Request from: " + address + ":" + port);
23             System.out.println("Received data:" + new String(packet.getData()));
24         } catch (IOException ex) {
25             System.out.println(ex.toString());
26         }
27     }
28 }

```

Początkowym parametrem niezbędnym do poprawnego działania serwera jest numer portu, na którym serwer będzie przyjmował wchodzące pakiety. Ten parametr jest podawany do konstruktora klasy `DatagramSocket`. Gniazdo to programowy (logiczny) punkt końcowy, który ustanawia dwukierunkową komunikację między serwerem a co najmniej jednym programem klienckim. Także ważnym elementem jest obiekt `DatagramPacket`, za pomocą którego można odczytać bajty przychodzącej wiadomości [16]. Żeby to zrobić, wywoływana jest funkcja „`socket.receive(packet)`”, która zatrzymuje działanie programu do póki nie przyjdzie jakiś pakiet. Po przyjściu pakietu UDP ta funkcja zapisuje bajty wiadomości do obiektu klasy `DatagramPacket`. W kolejnym kroku tego programu, tekst tej wiadomości jest wyświetlany w konsoli i można go zobaczyć. \ Cały ten kod jest zapętlony, aby serwer mógł nasłuchiwać wiadomości, dopóki program wykonuje się.

6.1.4. Tworzenie klienta UDP w aplikacji Android

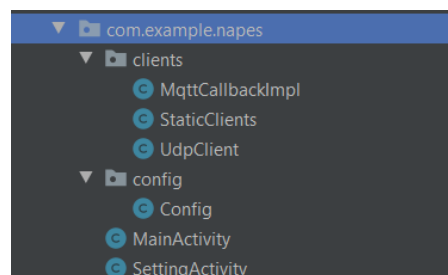
W celu tworzenia klienta UDP struktura kodu aplikacji wykonawczej, została nieco zmieniona. Bardziej szczegółowo mówiąc, to zostały dodane nowe pakiety Java oraz klasy. Także została stworzona nowa strona konfiguracji „`SettingActivity`”, na której będą podawane parametry związane z różnym rodzajem połączeń. Na poniższym rysunku jest pokazano, jaki jest widok tej strony:



Rysunek 6.10 Strona ustawień połączeniowych

Jak widać, została dodana możliwość ręcznego wpisania IP komputera, na którym są uruchomione serwery. W przypadku protokołu MQTT można podawać numer portu, na którym jest uruchomiony broker oraz można ustawić topik, który będzie subskrybowany. A w przypadku protokołu UDP można ustawić numer portu, na którym będzie nasłuchiwał serwer.

W celu oddzielenia kodu oraz łatwości jego odczytywania i redakcji zostały stworzone nowe klasy. Struktura drzewa projektu zmieniła się w następujący sposób:



Rysunek 6.11 Struktura drzewa projektu

Został stworzony pakiet „clients”, który zawiera klasy implementujące działanie poszczególnych klientów. Pakiet „config”, w którym będą znajdować się różne klasy odpowiedzialne za konfigurację różnych parametrów projektu.

Realizacja klienta UDP jest zaimplementowana w klasie „UdpClient”. Tak samo jak w przypadku serwera, ta klasa dziedziczy z klasy „Thread”. Kod implementacji klienta jest bardzo podobny do kodu serwera, tylko że na wejście funkcji przyjmuje takie parametry początkowe: IP adres serwera, numer portu oraz tekst wiadomości potrzebnej do wysłania:

Listing 6.5 Kod klienta UDP

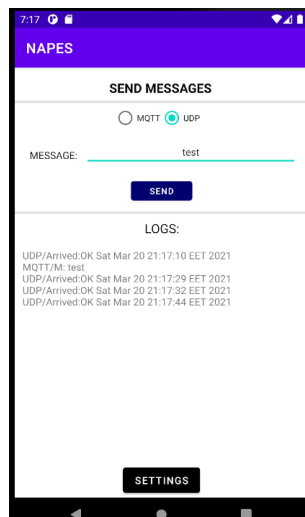
```

1 public class UdpClient extends Thread {
2
3     List<Long> sentTimeList; // Lista czasów wysłanych pakietów
4
5     DatagramSocket socket; // Obiekt portu
6     DatagramPacket packet; // pakiet
7
8     // Konfiguracja wstępna klienta
9     public UdpClient(Port port, Flow flow) {
10         super();
11         // ustawienia parametrów klienta
12         sentTimeList = new ArrayList<>();
13         dstPort = port.getClientInfo().getEndPoint().getPort();
14         dstAddress = port.getClientInfo().getEndPoint().getIP();
15         this.flow = flow;
16
17         // alokacja pamięci dla wysyłanych pakietów
18         byte[] buf = new byte[flow.getFParametr()];
19         address = InetAddress.getByNames(dstAddress);
20
21         // tworzenie pakietu (tutaj podają się parametry węzła docelowego)
22         packet = new DatagramPacket(buf, buf.length, address, dstPort);
23         // otwarcie portu podanego w konfiguracji komponentu
24         socket = new DatagramSocket(port.getEndPointHere().getPort());
25     }
26
27     // otwarcie portu podanego w konfiguracji komponentu
28     synchronized public void sendThroughLink() {
29
30         socket.send(packet); // wysyłanie pakietu
31
32         long sentTime1 = System.nanoTime() ; //zapisywanie czasu wysłania
33
34         sentTimeList.add(sentTime1); // dodanie czasu do listy
35     }
36 }

```

Widać, że kod jest bardzo podobny kodu serwera. Główną różnicą tutaj jest to, że obiekt klasy „DatagramPacket” jest tworzony za pomocą innego konstruktora, parametrami którego są adres IP, port oraz ciąg znaków, który trzeba wysłać.

Posiadając stworzonego klienta oraz serwera można zbadać, czy wszystko działa i czy będą wiadomości dostarczone w poprawny sposób. Dla przykładu, z urządzenia Android będzie wysłana wiadomość z tekstem „test”:



Rysunek 6.12 Widok aplikacji po wysłaniu wiadomości UDP

Jak można zobaczyć w logach aplikacji, została otrzymana wiadomość od serwera z aktualną datą i czasem, co świadczy o poprawnym nawiązaniu połączenia. Także można zobaczyć wiadomość od strony serwera, która wyświetlają się w terminalu:

```
Request from: /192.168.0.108:58961
Received data: test
OK Sat Mar 20 21:17:44 EET 2021 31
```

Rysunek 6.13 Widok z konsoli serwera po otrzymaniu wiadomości UDP

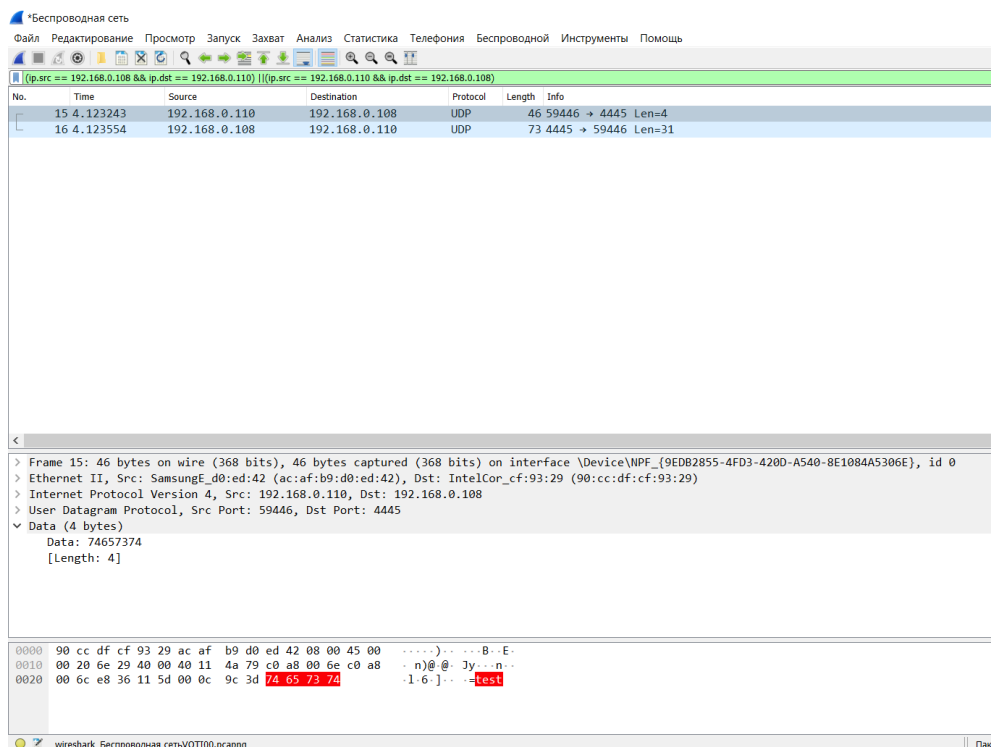
Jak widać, po stronie serwera można odczytać wiadomość oraz informację o nadawcę tej wiadomości. Można zobaczyć IP adres urządzenia, z którego została wysłana wiadomość oraz z którego portu.

Żeby upewnić się, że komunikacja odbywa się za pośrednictwem protokołu UDP, a nie innego, można użyć programu Wireshark, który pozwoli na odczytywanie pakietów podczas komunikacji klienta i serwera. Podczas odczytywania całego ruchu w sieci lokalnej została wysłana wiadomość z fizycznego urządzenia Android do serwera. Dla łatwości odnalezienia potrzebnych pakietów, został zastosowany filtr:

Listing 6.6 Filtr stosowany do wyświetlenia pakietów po emulacji

```
(ip.src == 192.168.0.108 && ip.dst == 192.168.0.110) || (ip.src == 192.168.0.110 && ip.dst == 192.168.0.108)
```

Za pomocą tego filtru udało się odnaleźć tylko potrzebne pakiety. Na poniższym rysunku, można zobaczyć, że komunikacja odbyła się bezpośrednio za pomocą protokołu UDP:



Rysunek 6.14 Widok z konsoli serwera po otrzymaniu wiadomości UDP

Jak można zobaczyć, najpierw jest wysłana wiadomość z adresu 192.168.0.110 do 192.168.0.108, czyli najpierw został wysłany pakiet od urządzenia Android do komputera, na którym jest uruchomiony serwer, a później został wysłany pakiet od komputera do urządzenia mobilnego z komunikatem, że wiadomość została otrzymana poprawnie. Za pomocą Wireshark da się odczytać nawet dane, które zostały wysłane (w tym przypadku tekst „test” rys. 6.14).

6.1.5. Tworzenie oraz uruchomienie serwera TCP

Tak samo, jak w przypadku z protokołem UDP, do umożliwienia komunikacji za pośrednictwem protokołu TCP niezbędnym jest realizacja klienta oraz serwera. Jak już wiadomo, TCP (Transmission Control Protocol) to jest jeden z głównych protokołów transmisji danych w Internecie, przeznaczony do sterowania transmisją danych [24]. W tym podrozdziale będzie przedstawiono jak można stworzyć i uruchomić własny serwer, który będzie działał na zasadzie protokołu TCP. W tym celu został stworzony nowy projekt w środowisku programistycznym IntelliJ IDEA, który w nowym wątku aplikacji uruchamia TCP serwer na wybranym porcie. Kod serwera TCP jest przedstawiony na poniższym listingu 6.7.

Listing 6.7 Fragment kodu serwera TCP

```

1  @Override
2  public void run() {
3      // tworzenie portu TCP
4      ServerSocket ss = new ServerSocket(port.getPort());
5
6      // pętla działająca podczas trwania symulacji
7      while (Config.simulating) {
8          // wczytywanie pobranych danych na porcie
9          Socket s = ss.accept();
10         isr = new InputStreamReader(s.getInputStream());
11         br = new BufferedReader(isr);
12         message = br.readLine();
13
14         // wyświetlanie informacji o otrzymanym pakiecie
15         System.out.println("Request from:" + s.getInetAddress() + ":" + s.getPort());
16         System.out.println("Received data: " + message);
17         // zamykanie połączenia oraz strumienia wyjściowego
18         isr.close();
19         br.close();
20         ss.close();
21         s.close();
22     }
23 }
24

```

Powyższy kod nieco różni się od kodu serwera UDP, ale zasada działania jest bardzo podobna. Głównym elementem tego programu jest obiekt klasy „ServerSocket”. Ten obiekt odpowiada za nasłuchiwanie wiadomości na określonym porcie, numer, którego jest podawany do konstruktora tego obiektu. Po wywołaniu funkcji „ss.accept()” serwer oczekuje na przyjście wiadomości od klienta. Po nawiązaniu połączenia przez klienta zostanie utworzone wystąpienie strumienia wyjściowego. Może to służyć do wysyłania danych z serwera do podłączonego klienta.

6.1.6. Tworzenie klienta TCP w aplikacji Android

Tworząc klienta wysyłającego wiadomości za pomocą TCP, do już istniejącej aplikacji zostały dodane niektóre nowe elementy i pliki. Klasa „TcpClient” została dodana do pakietu „clients”, gdzie są przechowywane klasy wszystkich klientów. Do widoku strony konfiguracji zostało dodane nowe pole, w którym można wskazać numer portu serwera TCP. Klient uruchamia się w osobnym wątku, tak jak to było w przypadku klienta UDP. Dlatego klasa „TcpClient” też dziedziczy z klasy „Thread”. Kod klienta wysyłającego wiadomości na port serwera jest pokazany na listingu 6.8 [24].

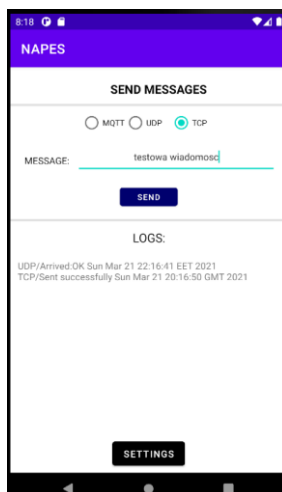
Listing 6.8 Fragment kodu klienta TCP

```

1  @Override
2  public void run() {
3
4      // ustawienie parametrów serwera
5      dstAddress = Config.ipAddressTcp;
6      dstPort = Config.tcpPort;
7
8      // tworzenie nowego portu TCP
9      s = new Socket();
10
11     // ustawianie portu klienckiego
12     s.bind(new InetSocketAddress(port.getPortNum()));
13
14     // nawiązywanie połączenia z serwerem TCP
15     SocketAddress sockaddr = new InetSocketAddress(dstAddress, dstPort);
16     s.connect(sockaddr);
17
18     // wysłanie wiadomości do serwera
19     printWriter = new PrintWriter(s.getOutputStream());
20     printWriter.write(message);
21     printWriter.flush();
22 }

```

Przedstawiony powyżej program działa jako klient, nawiązując połączenie z portem serwera. Po nawiązaniu połączenia klient wysyła podaną wiadomość do serwera. W przypadku, gdy wszystko udało się i wiadomość została wysłana użytkownik otrzymuje komunikat, o poprawnym wysłaniu:



Rysunek 6.15 Widok aplikacji po stronie użytkownika

Po stronie serwera w terminalu wyświetla się następujący komunikat:

```

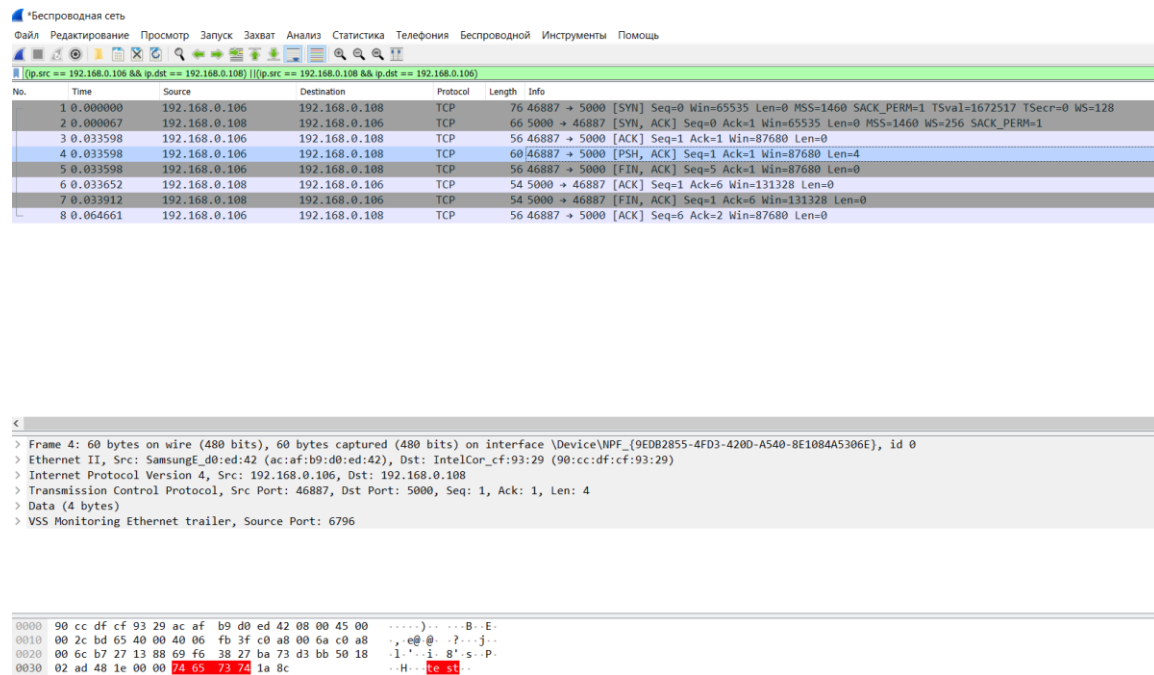
Request from: /192.168.0.108:55644
Received data: testowa wiadomosc

```

Rysunek 6.16 Widok z terminali po stronie serwera

Jak widać, w konsoli wyświetla się IP adres klienta wraz z portem, z którego została wysłana wiadomość. Także jest wyświetlana zawartość tej wiadomości. Żeby upewnić się

w tym, że komunikacja odbyła się za pomocą protokołu TCP, można znowu użyć programu Wireshark, wraz z odpowiednim filtrem.



Rysunek 6.17 Widok z programu Wireshark po wysłaniu wiadomości

Z powyższego rysunku można wnioskować, że komunikacja odbyła się bezpośrednio za pomocą protokołu TCP. Także można zobaczyć w jakiej kolejności były wysyłane pakiety wraz z ich flagami. W pakiecie z flagą „PSH” można znaleźć informację o wysłanej wiadomości tekstowej oraz można odczytać ją.

6.2. Implementacja parsera formatu RCR

W celu interpretacji struktury danych opisującej zachowanie komponentu działającego na węźle niezbędnym jest tworzenie parsera, który po wczytaniu pliku z rozszerzeniem „.rcr” mógłby zapisywać poszczególne parametry różnych elementów do instancji klasów Java. Zapisując różne parametry do instancji klas pojawia się możliwość interpretacji tych parametrów z pewną logiką działania oraz zachowania, która pozwoli na pracę samego środowiska wykonawczego. W tym podrozdziale będą przedstawione wszystkie etapy opisujące tworzenia parsera.

W pierwszym etapie implementacji niezbędnym było zapoznanie się z formatem zapisu plików RCR, który jest zapisany za pomocą składni EBNF. Wiedząc format zapisu oraz kolejność, w której są zapisane odpowiednie parametry, można poprawnie go odczytać. Posługując się szablonem, który jest przedstawiony na poniższym listingu, można zobaczyć wszystkie parametry komponentu oraz jaka jest ich kolejność.

Listing 6.9 Szablon pliku RCR [6]

```

1 <component> ::= "{" <c_name> ";" <pid> ";" <E_object> ";" <S_object> ";" <F_object>";"
2               <P_object> ";" <MQTT_broker> "}"
3 <pid>      ::= <number>
4
5 /* events */
6 <E_object> ::= "{" "E" ";" <events> "}"
7 <events>   ::= <event> | "[" <event> ( ";" <event>)+ "]"
8 <event>    ::= "{" <e_name> ";" "e" ";" <MQTT_e_name> "}" |
9             "{" <e_name> ";" "i" ";" <MQTT_e_name> "}" |
10            "{" <e_name> ";" "o" ";" <MQTT_e_name> "}" |
11            "{" <e_name> ";" "l" ";" <timeout>? "}"
12 <timeout>  ::= <parameter>
13
14 /* state machines */
15 <S_object> ::= "{" "S" ";" <fsms> "}"
16 <fsms>     ::= <fsm> | "[" <fsm> ( ";" <fsm>)+ "]"
17 <fsm>      ::= "{" <m_name> ";" <states> ";" <initial> "}"
18 <states>   ::= <state> | "[" <state> ( ";" <state>)+ "]"
19 <state>    ::= "{" <s_name> ";" <on_entry>? ";" <on_exit>? ";" <transitions>? "}"
20 <on_entry> ::= <actions>
21 <on_exit>  ::= <actions>
22 <transitions> ::= <transition> | "[" <transition> ( ";" <transition>)+ "]"
23 <transition> ::= "{" <e_name> ";" <s_name> ( ";" <actions>)? "}"
24 <actions>   ::= <action> | "[" <action> ( ";" <action>)+ "]"
25 <action>    ::= <e_name>
26 <initial>   ::= <s_name>
27
28 /* flows */
29 <F_object> ::= "{" "F" ";" <flows> "}"
30 <flows>    ::= <flow> | "[" <flow> ( ";" <flow>)+ "]"
31 <flow>     ::= "{" <f_name> ";" <f_type> ( ";" <f_parameter>)* "}"
32 <f_type>   ::= "simple" | "on_off"
33 <f_parameter> ::= <parameter>
34
35 /* ports */
36 <P_object> ::= "{" "P" ";" <ports> "}"
37 <ports>    ::= <port> | "[" <port> ( ";" <port>)+ "]"
38 <port>     ::= "{" <p_name> ";" <p_type> ";" <p_transport> ";" <local_end> ( ";" <client_info>)? "}"
39 <p_type>   ::= "c" | "s"
40
41 /* client or server */
42 <p_transport> ::= "T" | "U"
43
44 /* TCP or UDP */
45 <client_info> ::= <remote_end> ";" <state_flows>
46 /* <client_info> included only for client ports; client port can connect to only one server
47 port */
48
49 <state_flows> ::= <state_flow> | "[" <state_flow> ( ";" <state_flow>)+ "]"
50 <state_flow>  ::= "{" <s_name> ";" <f_name> "}"
51 <local_end>   ::= <endpoint_here>
52 <remote_end>  ::= <endpoint>
53

```

```

54 /* MQTT broker */
55 <MQTT_broker>      ::= "{" "M" ";" <endpoint_defp> "}"
56 /* default MQTT port:1883 (defined at IANA as MQTT over TCP) */
57 /* auxiliary */
58
59 <endpoint> ::= "{" <ip_address> ";" <port_no> "}"
60 <endpoint_here> ::= "{" <ip_address>? ";" <port_no> "}"
61 <endpoint_defp> ::= "{" <ip_address> ";" <port_no>? "}"
62 <ip_address> ::= <number> "." <number> "." <number> "." <number>
63 <port_no> ::= <number>
64 <c_name> ::= <name>
65 <e_name> ::= <name>
66 <m_name> ::= <name>
67 <s_name> ::= <name>
68 <f_name> ::= <name>
69 <p_name> ::= <name>
70 <parameter> ::= <number><unit>?
71 <number> ::= <digit><digit>*
72 <unit> ::= "s" | "ms"
73 <name> ::= <letter> (<letter> | <digit> | "_")*
74 <MQTT_e_name> ::= <name_plus>? ("/" <name_plus>)* "/"#"?
75 <name_plus> ::= <name> | "+"
76 <letter> ::= [a-z]
77 <digit> ::= [0-9]

```

Patrząc na szablon powyżej, można zobaczyć, że komponent posiada takie parametry jak:

- *nazwa komponentu* ----> („<c_name>”),
- *identyfikator procesu* ----> („<pid>”),
- *zdarzenia* ----> („<E_object>”),
- *maszyna stanów* ----> („<S_object>”),
- *przepływy* ----> („<F_object>”),
- *porty* ----> („<P_object>”),
- *konfiguracja MQTT* ----> („<MQTT_broker>”).

Elementy z dopiskiem „_object” są obiektami i posiadają własne parametry. Także warto zwrócić uwagę na samą składnię tego pliku, na przykład wszystkie obiekty są opisane w nawiasach klamrowych „{ }”, a różne parametry tego obiektu są oddzielone od siebie średnikiem „;”. Także można zobaczyć, że w niektórych miejscach są używane listy obiektów, które zapisują się w nawiasach prostokątnych „[]” i poszczególne elementy listy są oddzielone od siebie przecinkiem „,”.

Wiedząc format zapisu pliku RCR można zaprojektować algorytm, który pozwoli na zapisywanie wartości parametrów do instancji klas. W następnym kroku zostały stworzone klasy zawierające poszczególne parametry zgodnie z formatem RCR. Dla przykładu na następnym rysunku jest pokazana zawartość klasy „Event”:

Listing 6.10 Zawartość klasy „Event”

```

1 public class Event {
2
3     // parametry zdarzenia
4     String eName;
5     String eType;
6     String mqtt_eName;
7     int timeout;
8
9     public Event() {
10    }
11
12    public Event(String eName, String eType, String mqtt_eName) {
13        this.eName = eName;
14        this.eType = eType;
15        this.mqtt_eName = mqtt_eName;
16    }
17
18    public Event(Event event) {
19        this.eName = event.eName;
20        this.eType = event.eType;
21        this.mqtt_eName = event.mqtt_eName;
22    }
23
24    public Event(String eName, String eType, String mqtt_eName, int timeout) {
25        this.eName = eName;
26        this.eType = eType;
27        this.mqtt_eName = mqtt_eName;
28        this.timeout = timeout;
29    }
30 }

```

W tej klasie znajdują się wszystkie parametry, który zawiera obiekt „<E_objects>” z odpowiednim typem danych. Także można zauważyć, że ta klasa posiada dwa konstruktora. Jest tak zrobiono dlatego, że parametr „timeout” jest opcjonalny i w niektórych przypadkach nie jest on podawany. W standardzie zapisu RCR takie opcjonalne zmienne są oznaczone pytajnikiem.

Posiadając instancje klas niezbędnych do pracy parsera został zaimplementowany algorytm wczytywania danych z pliku. Algorytm ten działa na zasadzie pętli wczytującej każdy znak po kolei. W zależności od tego jaki znak odczyta program, algorytm zachowuje się w pewien sposób. Na przykład, jeśli program odczyta otwarty prostokątny nawias, to będzie otworzona pętla odczytująca wszystkie parametry oddzielone przecinkiem, dopóki nie pojawi się nawias zamykający listę. Dla przykładu na poniższym listingu jest pokazana jedna z funkcji odczytująca zdarzenia:

Listing 6.11 Przykład funkcji wczytującej listę zdarzeń

```

1  public class EventsParser extends PayloadParser{
2      // wczytywanie listy zdarzeń
3      public EventList parseEventList(){
4          EventList eventList = new EventList();
5          // zmienna 'till' przyjmuje wartość ostatniego znaku,
6          // do którego należy wczytywać plik
7          char till='0';
8
9          // sprawdzenie, czy RCR zawiera listę zdarzeń,
10         //czy posiada pojedynczy obiekt zdarzenia
11         if (linkedList.get(1).equals(""))
12             till = '0';
13         else if (linkedList.get(1).equals("{}"))
14             till='}';
15
16         // pętla zapisująca zdarzenia do listy
17         // do póki nie napotka się na znak końca obiektu 'till'
18         while (!linkedList.getFirst().equals(String.valueOf(till)) &&
19             !linkedList.get(1).equals(String.valueOf(till))){
20
21             // zapisywanie wczytanego zdarzenia do listy
22             eventList.getEvents().add(parseEvent());
23         }
24         return eventList;
25     }
26     ...
27 }

```

Wszystkie dane po wczytywaniu są zapisywane do instancji wcześniej stworzonych klas Java. dla łatwości odczytania i sprawdzenia danych, komponent wyświetla się w terminalu w sposób, jak to przedstawiono na rysunku 6.18.

```

Component{

cName='client[3]',

pid=105,

eventList=EventList{events=[
    Event{eName='TokenIn', eType='i', mqtt_eName='app/token2to3', timeout=0},
    Event{eName='TokenOut', eType='o', mqtt_eName='app/token3toS', timeout=0},
    Event{eName='Local', eType='l', mqtt_eName='null', timeout=1}],

stateMachineList=StateMachineList{stateMachines=[
    StateMachine{mName='fsm1', stateList=StateList{states=[
        State{sName='Low', onEntry=OnEntry{actionList=ActionList{actions=[Action{eName='tokenIn'}, Action{eName='TokenOut'}]}}, onExit=OnExit{actionList=Ac
        State{sName='High', onEntry=OnEntry{actionList=ActionList{actions=[Action{eName='Local'}]}}, onExit=OnExit{actionList=ActionList{actions=[]}}, trans

flowList=FlowList{flows=[
    Flow{fName='f1', fType='simple', timeParam=500, fParametr=100, unit='ms'},
    Flow{fName='f2', fType='simple', timeParam=4, fParametr=256, unit='s'}]},

portList=PortList{ports=[
    Port{pName='port1', pType='c', pTransport='U', endPointHere=EndPoint{IP='', port=8004}, clientInfo=ClientInfo{endPoint=EndPoint{IP='127.0.0.1', port=8000},
    Port{pName='port2', pType='c', pTransport='U', endPointHere=EndPoint{IP='', port=8004}, clientInfo=ClientInfo{endPoint=EndPoint{IP='127.0.0.1', port=8000}},

mqttBroker=MQTTBroker{endPointDefp=EndPoint{IP='127.0.0.1', port=1883}}

```

Rysunek 6.18 Widok z okna terminali po wczytywaniu pliku RCR

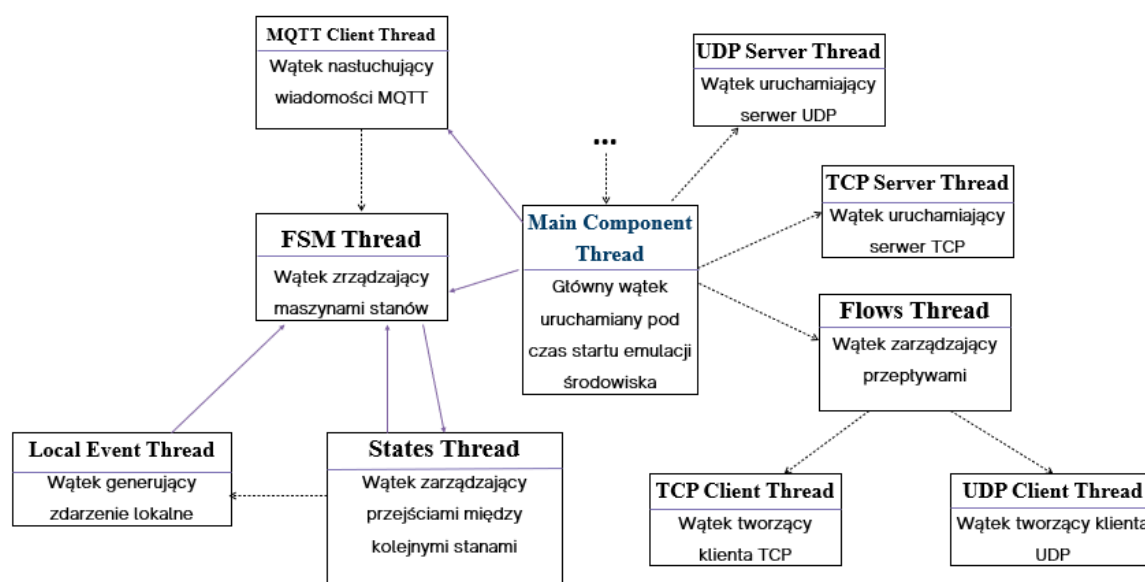
Dla sprawdzenia poprawności działania parsera zostały przeprowadzone badania dla wielu różnych plików RCR. W rezultacie końcowym widać wszystkie wartości różnych

parametrów zapisanych w pliku konfiguracyjnym. Dany algorytm został zintegrowany do projektu NAPES Runtime w Android Studio i pozwala na odczytywanie plików RCR z urządzeń mobilnych. W następnych krokach tworzenia środowiska wykonawczego te wartości będą wykorzystywane się w celu interpretacji różnych struktur danych z pewnym algorytmem i zachowaniem (tworzenie wątków, uruchomienie serwerów na określonych portach itd.).

6.3. Implementacja środowiska wykonawczego

W tym podrozdziale będzie omówiona najważniejsza część dotycząca implementacji środowiska wykonawczego, ponieważ w tym etapie pracy zostało zaprogramowane zachowanie całego systemu. Interpretując dane pobrane z pliku konfiguracyjnego, system uruchamia ten czy inny wątek zgodnie z podanymi danymi. Na przykład: jeżeli w pliku konfiguracyjnym znajdują się parametry dla tworzenia serwera, to system uruchomi oddzielny wątek, który za to odpowiada.

Głównym aspektem w implementacji systemu było tworzenie wielowątkowej aplikacji. Ogólnie, wielowątkowość jest podstawową i niezbędną rzeczą w programowaniu sieciowym. Także stosowanie wielowątkowości było wygodnym pod względem tworzenia maszyny stanów. Maszyna stanów systemu NAPES powinna być dynamiczną, taką, żeby móc zmienić swój stan w każdej chwili, kiedy zajdzie zdarzenie [25]. Dla przykładu, broker MQTT może otrzymać zdarzenie w dowolnej chwili, dlatego wątek nasłuchujący wiadomości MQTT współdzieli zmienne z wątkiem zarządzającym maszyną stanów. W rezultacie końcowym zostało skonfigurowane zachowanie całego systemu i zostało zaprogramowane wiele różnych wątków, z których można wydzielić kilka głównych, które są pokazane na poniższym schemacie wraz z ich zależnościami.



Rysunek 6.19 Schemat wątków aplikacji

Jak można było zauważyć, w systemie jest jeden główny wątek, który uruchamia pozostałe. Strzałki między oznaczeniami wątków pokazują w jakiej hierarchii mogą być uruchomione wątki. Strzałki z linią przerywaną pokazują, że wątek nie musi być konieczne uruchomiony, gdyż wszystko zależy od konfiguracji. Ciekawym jest to, że taka struktura

systemu pozwala na jednoczesne uruchomienia klienta i serwera na jednym urządzeniu, przy podaniu odpowiednich parametrów.

W poprzednich podrozdziałach już było omówiono sposób tworzenia procesów dotyczących ruchu sieciowego. Dlatego w następnych będzie krótko przedstawione główne kroki implementacji maszyny stanów.

6.3.1. Logika działania emulatora

Część programu dotycząca emulacji środowiska wykonawczego zaczyna się od klasy „Service”. Ta klasa zawiera w sobie metodę uruchamiającą emulację. Kod źródłowy wraz z komentarzami klasy „Service” są przedstawione na poniższym rysunku:

Listing 6.12 Fragment kodu klasy Service

```
1  public class Service extends Thread {
2
3      EventService eventService;    //obiekt uruchamiający wątek obsługi zdarzeń
4      ServiceFsm serviceFsm;        //obiekt uruchamiający wątek obsługi maszyny stanów
5      ServicePorts servicePorts;    //obiekt uruchamiający wątek obsługi portów
6      Component component;          //obiekt zawierający dane komponentu
7
8
9      //metoda zaczynająca symulację
10     public void serviceMain() {
11
12         // zmiana flagi mówiącej o starcie symulacji
13         Config.simulating = true;
14
15         // tworzenie obiektu obsługi zdarzeń
16         // oraz tyczenia się z brokerem MQTT
17         Config.ipAddressBroker = component.getMqttBroker().getEndPointDefp().getIP();
18         eventService = new EventService(handler);
19         eventService.startMqttClient();
20
21         // start wątku obsługującego maszynę stanów
22         serviceFsm = new ServiceFsm(eService, component, component.getStateMachineList());
23         serviceFsm.start();
24
25         // rozmiar tej listy będzie równy ilości maszyn stanów zadeklarowanych w RCR
26         // lista przechowuje obiekty,
27         // zawierające aktualne informacje o uruchomionych FSM (ich bieżące stany)
28         ArrayList<ServiceStates> serviceStatesArrayList =
29             new ArrayList<>(serviceFsm.getServiceStatesArrayList());
30
31
32         // start wątku obsługującego podane porty
33         servicePorts = new ServicePorts(component, handler, serviceStatesArrayList);
34         servicePorts.start();
35
36     }
37     ...
38 }
```

Patrząc na powyższy kod, można zauważyć, że ważnym elementem tutaj jest obiekt komponentu „Component”, zawierający dane konfiguracyjne, które są niezbędne do

podania na wejściu funkcji obsługi maszyny stanów oraz obsługi portów. Ogólnie, zasada działania kodu jest prosta i można ją opisać krok po kroku:

- najpierw za pomocą obiektu „EventService” uruchamia się obsługa zdarzeń (nawiązuje się połączenie z brokerem MQTT, po czym wątek czeka na wiadomość)
- następnie obiekt obsługi zdarzeń jest przekazywany do obiektu obsługi maszyny stanów. Jest tak zrobiono dlatego, żeby po otrzymaniu zdarzenia system mógł sprawdzić, czy maszyna nie musi przejść do kolejnego stanu.

W kolejnych liniach kodu, do listy są zapisywane aktualne stany każdej maszyny. Ta lista jest potrzebna przy obsłudze portów, ponieważ praca portu zależy od aktualnego stanu FSM. Także obsługa portów decyduje czy musi być uruchomiony klient, czy serwer (*dlatego także jest podawany obiekt komponentu*).

6.3.2. Organizacja kolejki zdarzeń

Ważnym było zrobić system takim, żeby każde nowe przychodzące zdarzenie było sprawdzane. W tym celu, każde nowe zdarzenie jest zapisywane do listy, a później elementy tej kolejki są przetwarzane zgodnie z algorytmem FIFO [26]. Na poniższym rysunku jest pokazany fragment kodu z klasy, która obsługuje maszynę stanów. Jak można zobaczyć w 8 linii kodu tworzy się nowa kolejka zdarzeń za pomocą „new LinkedList<>()”, która później jest uzupełniana po przyjściu jakiegokolwiek zdarzenia. Taka implementacja wymaga użycia semaforów, ponieważ w czasie, gdy pojawia się nowe zdarzenie, warstwa obsługi maszyny stanów powinna sprawdzać, czy ono nie spełnia warunków przejścia do innego stanu na FSM.

Listing 6.13 Fragment kodu, w którym system czeka na zdarzenie wchodzące

```
1      ...
2
3      synchronized (eventService) {
4
5          // jeśli lista kolejki zdarzeń nie zmieniła się,
6          // to system czeka na nowe zdarzenie wejściowe
7          if (!eventService.isChanged()) {
8              eventService.setArrivedQueueEvents(new LinkedList<>());
9              eventLinkedList = null;
10
11              // semafor, oczekujący na nowe zdarzenie
12              eventService.wait();
13          }
14      }
15      ...
```

Te linijki kodu pokazują, jak system czeka na zdarzenia wejściowe, w przypadku, gdy kolejka jest pusta. W roli semafora występuje obiekt klasy „EventService”. Funkcja oczekiwania przerywa się za pomocą funkcji „notify()”, wywołaną po otrzymaniu wiadomości MQTT (list. 6.14), lub po przyjściu zdarzenia lokalnego.

Listing 6.14 Fragment kodu, w którym system zapisuje zdarzenie wejściowe do kolejki

```
1 // przechwytywanie wchodzących wiadomości MQTT
2 @Override
3 public void messageArrived(String topic, MqttMessage message) throws Exception {
4     // tekst wiadomości
5     String payload = new String(message.getPayload());
6
7     // blok synchronized w tym przypadku nie pozwala na dodawanie
8     // nowych zdarzeń wielu wątkom jednocześnie
9     synchronized (eventService) {
10
11         // wyszukiwanie obiektu Event za nazwą 'topic'
12         // oraz jego umieszczenie do kolejki
13         eventService.getArrivedQueueEvents()
14             .add(searchEventByTopic(component.getEventList(), topic));
15
16         // zmiana wartości flagi, która wskazuje
17         // o dodaniu nowego zdarzenia do kolejki
18         eventService.setChanged(true);
19
20         // powiadomienie o konieczności kontynuacji przetwarzania
21         // dla innych wątków, które znajdują się w stanie oczekującym
22         eventService.notify();
23     }
24 }
```

W powyższym fragmencie kodu, można także zobaczyć jak do kolejki dodaje się nowe zdarzenie w bezpieczny sposób (stosowanie zsynchronizowanego bloku kodu). Obiekt klasy „EventService” dodaje do kolejki nie tylko zdarzenia MQTT, a także zdarzenia lokalne (implementacja jest bardzo podobna i jest przedstawiona w podrozdziale 6.3.5).

6.3.3. Implementacja obsługi stanów

Zgodnie z zasadami emulacji: w każdym stanie jest określone zdarzenie, po przyjściu którego, stan maszyny zmienia się na inny. Takie zdarzenie jest parametrem obiektu „Transitions” [25]. Podczas przejścia do kolejnego stanu, system może wykonywać określone akcje, jeśli zostały one zadeklarowane w pliku konfiguracyjnym RCR. Poniższy listing 6.15 przedstawia fragment kodu, w którym system sprawdza, czy przetwarzane zdarzenie znajduje się na liście transakcji bieżącego stanu, jeśli tak, to sprawdza czy spełnia ono warunek na przejście do innego stanu FSM.

Listing 6.15 Fragment kodu, w którym odbywa się obsługa maszyny stanów

```

1  // po przyjściu nowego zdarzenia uruchamia się poniższa pętla,
2  // która przetwarza wszystkie transakcje dla bieżącego stanu
3  for (Transition currentTransition :
4      currentState.getTransitionList().getTransitions()) {
5
6      // poniższa linia sprawdza, czy nowe zdarzenie nie jest zdefiniowane
7      // na którymś obiekcie Transaction dla bieżącego stanu
8      // jeśli tak, to znaczy, że został spełniony warunek na zmianę stanu
9      if (currentTransition.getName().equals(arrivedEventTemp.getName())) {
10
11         // wykonywanie akcji wyjściowych, jeśli takie zostały zdefiniowane w RCR
12         sa.doActions(currentState.getOnExit().getActionList());
13
14         // zmiana bieżącego stanu na stan zadeklarowany
15         // w przetwarzanym obiekcie Transaction
16         currentState = getStateByName(stateMachine, currentTransition.getName());
17
18         // wykonywanie akcji przejściowych zdefiniowanych na przetwarzanym Transaction
19         sa.doActions(currentTransition.getActionList());
20
21         // wykonywanie akcji wejściowych zdefiniowanych na nowym stanie
22         sa.doActions(currentState.getOnEntry().getActionList());
23
24         // zapisywanie do mapy bieżącego stanu dla potocznej FSM
25         map.put(stateMachine.getName(), currentState.getName());
26     }
27 }

```

Jak widać, algorytm szuka takiej samej nazwy zdarzeń w liście „Transitions”. Jeżeli nazwa będzie taka sama, to system wykona określone wyjściowe i wejściowe akcje i przejdzie do kolejnego stanu. W końcu, aktualnie zmieniony stan jest zapisywany do zabezpieczonej zmiennej. Jest tak zrobiono po to, żeby w każdej chwili i w każdym miejscu programu można było zwrócić aktualny stan maszyny.

6.3.4. Implementacja obsługi portów

Każdy port podany w konfiguracji jest sprawdzony na jego typ. W zależności od jego typu, program uruchamia klienta albo serwera odpowiednio. Na poniższym rysunku jest przedstawiony blok kodu demonstrujący tę realizację:

Listing 6.16 Fragment kodu, w którym odbywa się obsługa portów na węźle NAPES

```

1  public void run() {
2
3      // sprawdzenie każdego portu na jego typ
4      PortList portList = component.getPortList();
5      for (Port port : portList.getPorts()) {
6          if (port.getType().equals("s"))
7              startServer(port.getTransport(),
8 port.getEndPointHere().getPort());
9          if (port.getType().equals("c"))
10             startClient(port);
11     }
12 }

```

W przypadku uruchomienia klienta, program zaczyna przepływ danych z odpowiednimi parametrami. Zgodnie z założeniami środowiska wykonawczego, dane są wysyłane okresowo z zadanymi odstępami czasu. Także przy zmianie stanu system powinien szybko reagować, zmieniając swój przepływ. Okres czasu pomiędzy pakietami jest ustawiony za pomocą funkcji „*wait(Long timeout)*”.

6.3.5. Obsługa zdarzeń lokalnych

Zdarzenie lokalne może być utworzone tylko podczas wykonywania akcji, która generowana jest w trakcie przejścia do kolejnego stanu. Również może to być akcja wyjściowa lub wejściowa, któregoś z stanów. Zdarzenia lokalne zostały zdefiniowane w celu imitacji czynników zewnętrznych. W rzeczywistości generacja takiego zdarzenia, to jest zwykle uruchomienia timera czasu, po którym przyjdzie wyimaginowane zdarzenie, nazywane zdarzeniem lokalnym. Niżej jest przedstawiona implementacja generacji zdarzeń lokalnych:

Listing 6.17 Fragment kodu, realizujący zdarzenia lokalne

```
1  @Override
2  public void run() {
3
4      // uruchomienie timera za pomocą funkcji sleep()
5      Thread.sleep(localEvent.getTimeout() * 1000);
6
7      // dodawanie zdarzenia do kolejki
8      // oraz wzbudzenie pracy innych wątków
9      synchronized (eventService) {
10         eventService.getArrivedQueueEvents().add(localEvent);
11         eventService.setChanged(true);
12         eventService.notifyAll();
13     }
14
15 }
```

Czas, po którym musi przyjść zdarzenie, jest podawany w pliku konfiguracyjnym. Okres czasu oczekiwania, w tym przypadku, jest zrealizowany za pomocą funkcji „*Thread.sleep(timeout);*”. Także widać tutaj stosowanie semafora oraz zmianę flagi, mówiącej o tym, że przyszło nowe zdarzenie, co pozwala na zapisywanie zdarzenia do kolejki i na jego późniejsze przetwarzanie.

6.3.6. Tworzenia obsługi logów systemu

W trakcie emulacji system tworzy plik o formacie JSON, do którego zapisuje logi zgodnie z formatem *Trace Event Format*. Ten format pozwala na wizualizację różnych zdarzeń na osi czasu. Jest on bardzo przydatnym formatem do celów systemu NAPES, ponieważ pozwala na wizualizację działań maszyny stanów oraz generatorów przepływności. Taka wizualizacja pozwoli zobaczyć, jak zachowywał się system podczas emulacji. Dokumentację tego formatu można znaleźć w bibliografii [27].

Na rysunku 6.32 jest pokazany przykładowy JSON, który powstaje po zakończeniu symulacji. W tych logach można odnaleźć informację o czasie rozpoczęcia i zakończenia różnych procesów uruchomionych podczas runtime.

W tym przykładzie można zobaczyć, że został uruchomiony serwis obsługi maszyny stanów dla „fsm1” (2 wiersz). Także widać jak system od razu po uruchomieniu maszyny stanów ustawił stan początkowy „A”, a po niektórym czasie zmienił stan aktualny na „B” (4-5 wiersze).

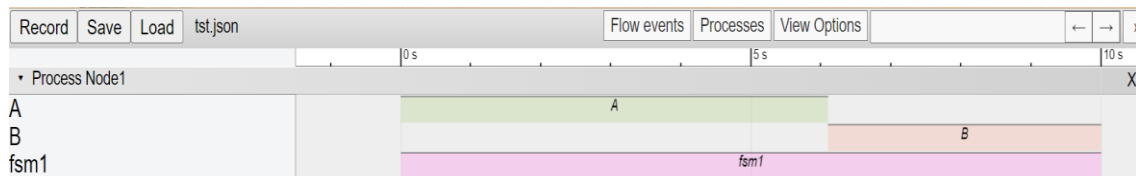
```

1 [{"traceEvents":[{"pid":"Node1","tid":"fsm1","ts":1585820123394,"ph":"b","cat":"service_states","name":"fsm1","id":1,"args":{}},
2 {"pid":"Node1","tid":"fsm1","ts":1585820123394,"ph":"b","cat":"state","name":"A","id":1,"args":{}},
3 {"pid":"Node1","tid":"fsm1","ts":1585826223394,"ph":"e","cat":"state","name":"A","id":1,"args":{}},
4 {"pid":"Node1","tid":"fsm1","ts":1585826223394,"ph":"b","cat":"state","name":"B","id":1,"args":{}},
5 {"pid":"Node1","tid":"fsm1","ts":1585830123393,"ph":"e","cat":"state","name":"B","id":1,"args":{}},
6 {"pid":"Node1","tid":"fsm1","ts":1585830123394,"ph":"e","cat":"service_states","name":"fsm1","id":1,"args":{}}
7 ]}
8 ]
9 ]

```

Rysunek 6.20 Przykładowy JSON z logami NAPES

Istnieje wiele różnych rozwiązań do wizualizacji zapisu w takiej postaci (narzędzia z otwartym kodem źródłowym). Ten format ma takie szerokie zastosowanie, ponieważ wykorzystuje się w zwykłych przeglądarkach. Dla przykładu, wpisując polecenie <chrome://tracing/> do pola wyszukiwania w przeglądarce Google Chrome – otworzy się strona, na której można wczytać i wizualizować JSON. Na rysunku 6.33 przedstawiona jest wizualizacja do wcześniej podanego pliku z logami.



Rysunek 6.21 Przykład wizualizacji logów

Za pomocą tej wizualizacji można obserwować uruchomione procesy systemu w skali czasu. Implementacja zapisu takich logów odbywa się bezpośrednio w kodzie programu. Na list. 6.18 jest pokazany ułamek kodu pokazujący tą implementację. Ten przykład pokazuje jak po zmianie stanu (5 linia), system notuje czas, a następnie wstawia rekord do pliku z logami.

Listing 6.18 Ułamek kodu realizującego zapisywanie logów do pliku JSON

```

1 // zmiana bieżącego stanu
2 currentState = getStateByName(stateMachine, currentTransition.getName());
3
4 // dodanie czasu zmiany stanu do logów aplikacji
5 String currentTime = (Long.toString(System.currentTimeMillis()));
6
7 handler.addLog("{\"pid\":\"Node1\",\"tid\":\"fsm1\",\"ts\":\"" + currentTime +
8               "\",\"ph\":\"b\",\"cat\":\"service_states\",\"name\":\"" +
9               currentState.getName()+"\", \"id\":1,\"args\":{}}", handler);

```


7. Testowanie komponentu NAPES Runtime dla systemu Android

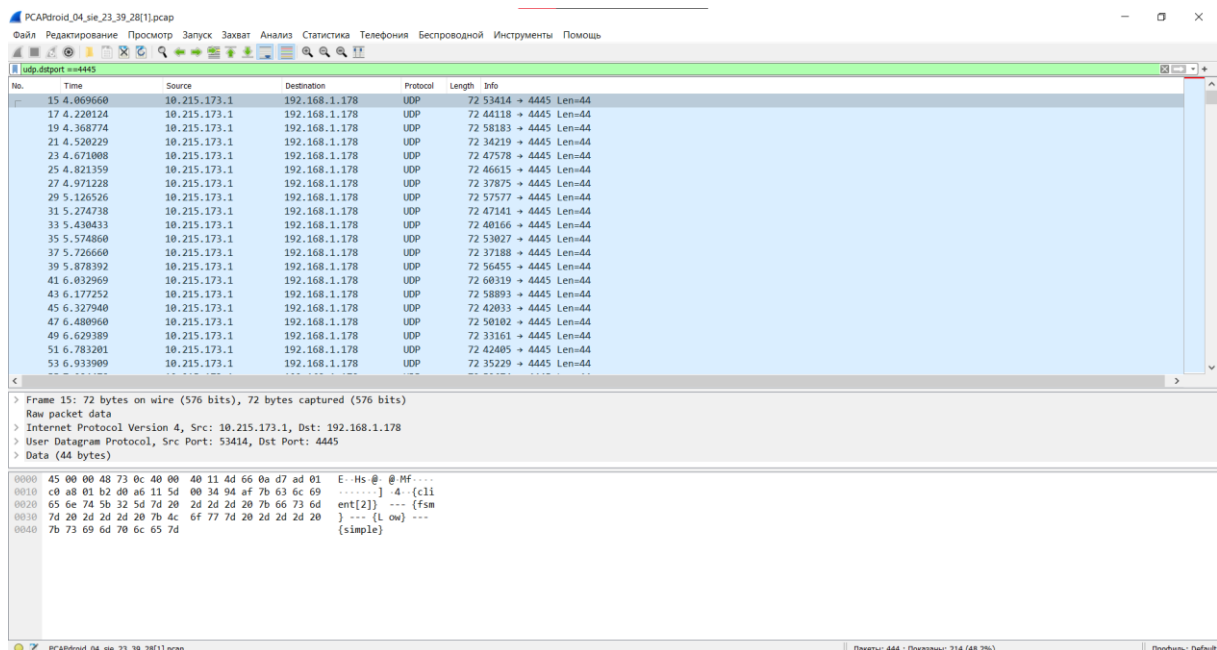
W tym podrozdziale będzie opisane badanie wstępne, które pozwoliłoby sprawdzić poprawność działania środowiska wykonawczego czy dany system zachowuje się zgodnie z oczekiwaniami. W tym celu, należałoby sprawdzić odstępy czasowe pomiędzy kolejnymi pakietami oraz czas reakcji systemu na zmiany środowiskowe. Dlatego został stworzony testowy konfiguracyjny plik RCR, po wczytywaniu, którego system zaczyna emulację aplikacji. Zawartość tego pliku jest przedstawiona niżej:

Listing 7.1 Konfiguracja klienta zgodnie z wymaganiami plików RCR

```
1 {
2     client[0];
3     104;
4     {E;
5         [{TokenIn;i;app/tokenI},
6          {TokenOut;i;app/tokenO},
7          {Local;l;5}]];
8     {S;
9         {fsm;[{Low;;;{TokenIn;High}},
10            {High;Local;;;{Local;Low;TokenOut}}]];
11         Low}};
12     {P;
13         {port;c;U;{;8004};{192.168.1.178;4445};
14          fsm;[{Low;{simple;150ms;3}},{High;{fast;50ms;5}}]}};
15     {M;
16         {192.168.1.178;1883}}
17 }
```

Jak widać, w tym pliku są opisane dwa przepływy, które środowisko generuje w zależności od aktualnego stanu FSM. W stanie „Low” system wysyła pakiety UDP do serwera z częstotliwością każdych 150 [ms], a w stanie „High” z częstotliwością każdych 50 [ms]. System zaczyna swoje działania w stanie „Low”, który jest wskazany jako domyślny. Żeby przejść od stanu „Low” do stanu „High” musi zajść zdarzenie środowiskowe „TokenIn”. Po przyjsciu tego zdarzenia system uruchamia zdarzenie lokalne, które przyjdzie przez 5 sekund. Warunkiem przejścia od stanu „High” do stanu „Low” jest zdarzenie lokalne. To oznacza, że zawsze po pięciu sekundach w stanie „High”, system będzie zmieniał stan z powrotem na „Low”.

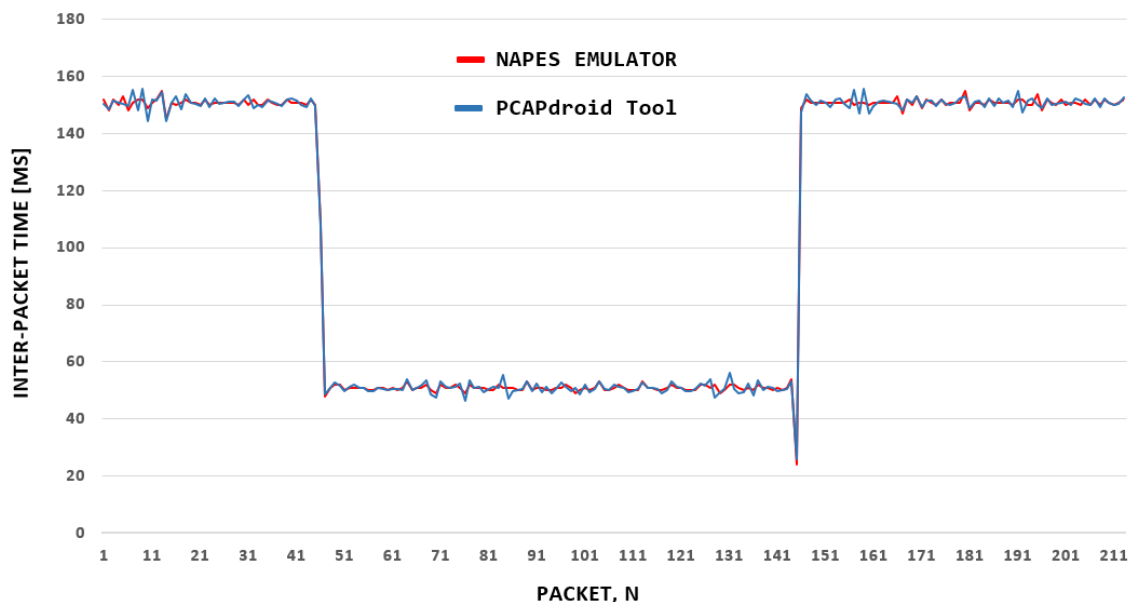
Dlatego żeby zbadać dokładność systemu, do programu została dodana linia kodu, która po każdym wysłanym pakiecie zapisuje czas go wysłania do pliku. Także podczas symulacji został zapisany ruch sieciowy na urządzeniu fizycznym Android, za pomocą już istniejącej aplikacji **PCAPdroid**. Ta aplikacja pozwala na zapisywanie całego ruchu sieciowego na telefonie Android oraz na eksportowanie tego ruchu do pliku z rozszerzeniem „.pcap”. Następnie po otwarciu oraz po odpowiednim filtrowaniu pakietów (jak to pokazano na rysunku niżej), udało się wyciągnąć tylko potrzebne do zaobserwowania pakiety.



Rysunek 7.1 Zebrany ruch sieciowy po wykonaniu emulacji

Czasy tych pakietów zostały sparsowane oraz zostały obliczone odstępy czasowe pomiędzy kolejnymi pakietami, tak samo jak to odbyło się z czasami generowanymi przez system. Te odstępy czasowe, zostały między sobą porównane i są przedstawione na poniższym wykresie:

ODSTĘPY CZASOWE MIĘDZY KOLEJNYMI PAKIETAMI UDP

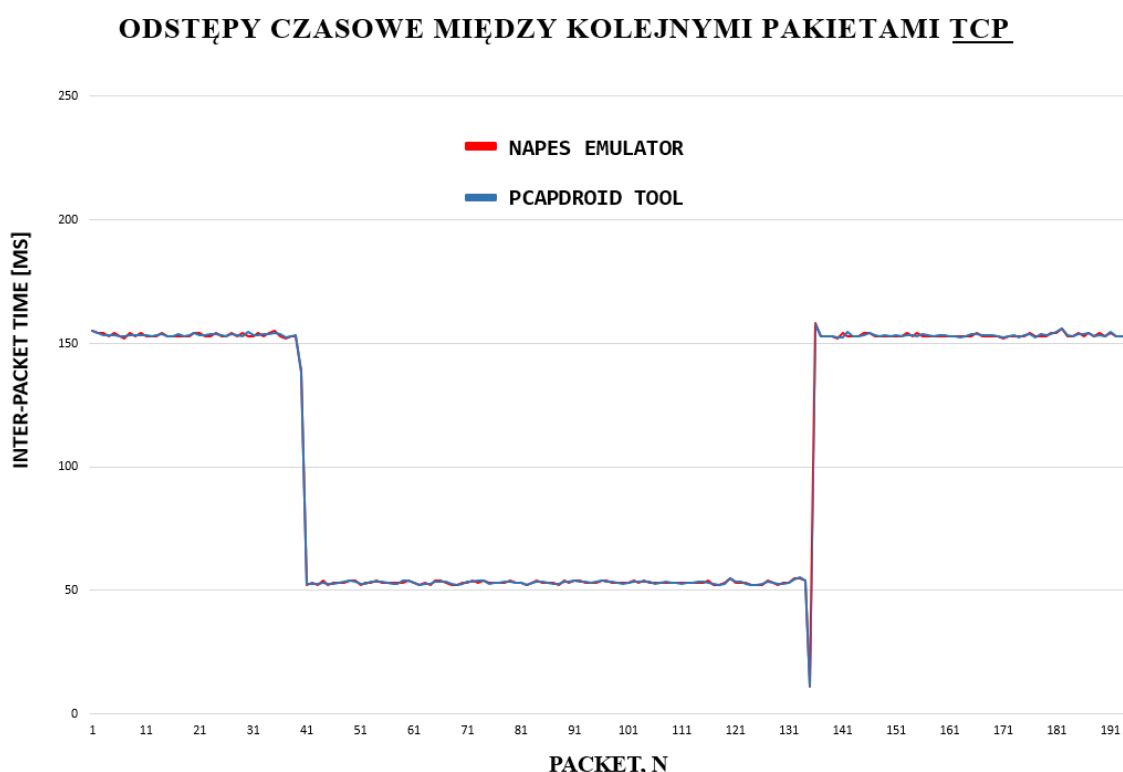


Rysunek 7.2 Wykres odstępów czasowych pomiędzy kolejnymi pakietami UDP

Wnioskując ten wykres, można powiedzieć, że system zachowuje się poprawnie. Ponieważ jego zachowanie jest zgodne z oczekiwaniami (dobrze widać, jak środowisko

zmienia przepływ na szybszy, a po pięciu sekundach, kiedy zachodzi zdarzenie lokalne zmienia przepływ na wolniejszy). Odstępy czasowe trochę różnią się od wartości oczekiwanej, ale taka odchyłka jest niewielka (w skali mikrosekund), co jest zadowalającym wynikiem. Także, można stwierdzić, że czasy, które są zapisywane do pliku przez system są jak najbardziej dokładne (środowisko poprawnie notuje czas). Są one nawet bardziej dokładne niż czasy zapisane przez program PCAPdroid, ponieważ na wykresie widać, jak w niektórych momentach odstępy czasów między pakietami mogą mieć niewielkie opóźnienie, które może być spowodowane obciążeniem procesora na urządzeniu.

Taki sam zestaw danych został zebrany dla tej samej konfiguracji, tylko przy użyciu protokołu TCP. Na poniższym wykresie są porównane odstępy czasowe między kolejnymi pakietami TCP, dla danych zebranych bezpośrednio z emulatora oraz z danych zapisanych za pomocą narzędzia PCAPTTool:

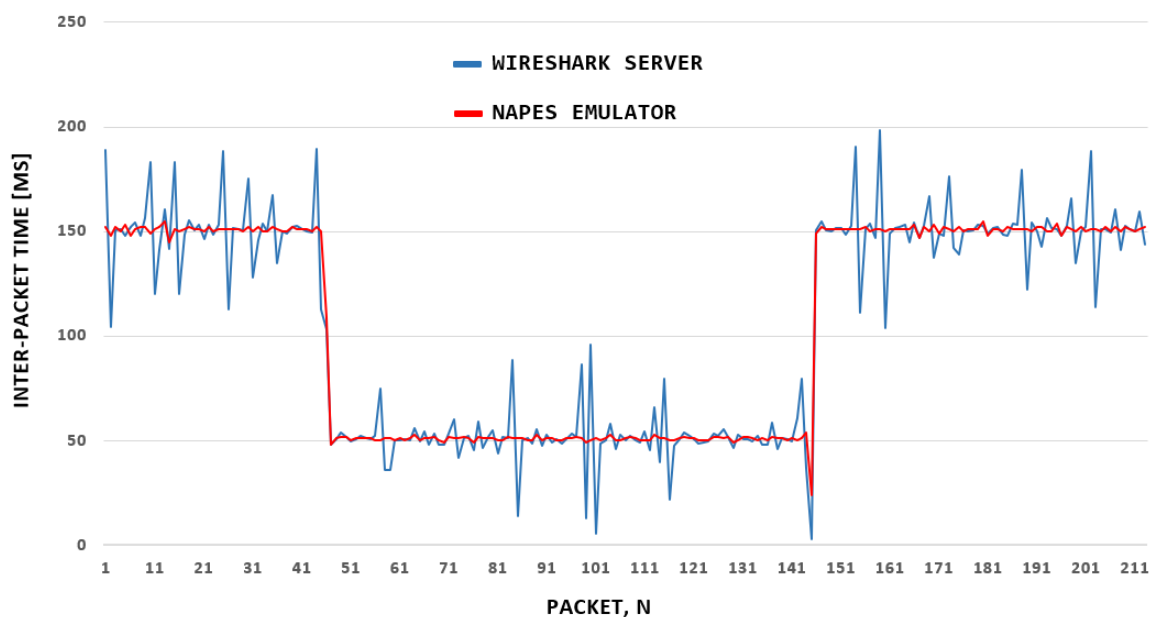


Rysunek 7.3 Wykres odstępow czasowych pomiędzy kolejnymi pakietami TCP

W przypadku użycia protokołu TCP, widać, że środowisko wykonawcze zachowuje się w sposób poprawny. Odchyłka od wartości oczekiwanych jest dość niewielka, co także jest zadowalającym wynikiem. Także można stwierdzić, że czasy wysłania pakietów zapisane przez system są bardzo dokładnymi i w następnych eksperymentach można będzie posługiwać się logami bezpośrednio ze środowiska wykonawczego.

Także, podczas tych symulacji, został zapisany ruch pakietów na serwerze, do którego były wysyłane pakiety. Po użyciu odpowiednich filtrów zostały wyciągnięte czasy przyjścia pakietów TCP i UDP. Za pomocą tych danych, także zostały obliczone odstępy czasowe, które zostały porównane z klienckimi odstępami czasowymi i te porównania są przedstawione na poniższych wykresach (rys. 7.4).

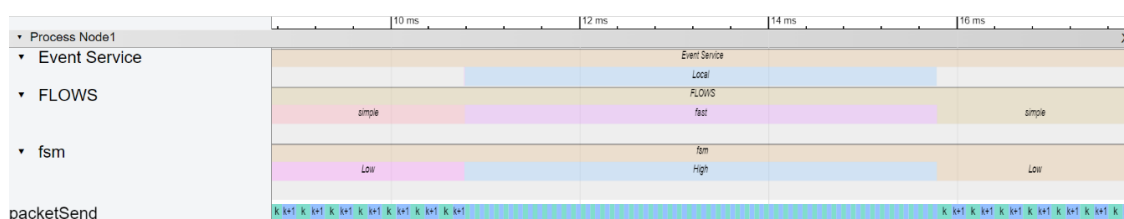
ODSTĘPY CZASOWE MIĘDZY KOLEJNYMI PAKIETAMI UDP



Rysunek 7.4 Wykres odstępów czasowych pomiędzy kolejnymi pakietami UDP na porcie serwerowym

Tutaj dobrze widać, że na serwerze odstępy czasowe między pakietami mogą bardzo się różnić od wartości oczekiwanej. Taka odchyłka jest spowodowana opóźnieniami w samej sieci i na tych wykresach można łatwo zobaczyć to opóźnienie [28].

Po upewnieniu, że danymi z środowiska wykonawczego można posługiwać się, w kodzie zostały dodane linijki tworzące logi systemu. Logi te dodają się do pliku z rozszerzeniem „.json” w formacie Trace Event Format. Ten format pozwala na wizualizację poszczególnych zdarzeń i zjawisk zachodzących w skali czasu. Na poniższym rysunku jest przedstawiona wizualizacja pracy środowiska wykonawczego dla wcześniej podanej konfiguracji:



Rysunek 7.5 Wizualizacja logów za pomocą Trace Event Visualisation

Taka wizualizacja pozwala zobaczyć jakie procesy są w systemie oraz w jakim czasie oni odbywają się podczas trwania symulacji. Na powyższym przykładzie widać, że z przejściem do stanu „High” system zaczyna przepływ „fast” i odpowiednio widać odstępy czasowe między pakietami na samym dole. Także widać, że po przyjsciu zdarzenia lokalnego maszyna stanów zmienia swój stan prawie od razu. Za pomocą takich logów można obliczyć czas reakcji systemu na zmiany. Odejmując od czasu startu przepływu „simple”, czas przyjscia zdarzenia lokalnego, można obliczyć ten czas reakcji. W tym przypadku ten czas wynosi: 5 [ms]. Po otrzymaniu takiego wyniku można stwierdzić, że jest on zadowalający i czas reakcji systemu nie jest zbyt duży (system prawie momentalnie reaguje na zmiany).

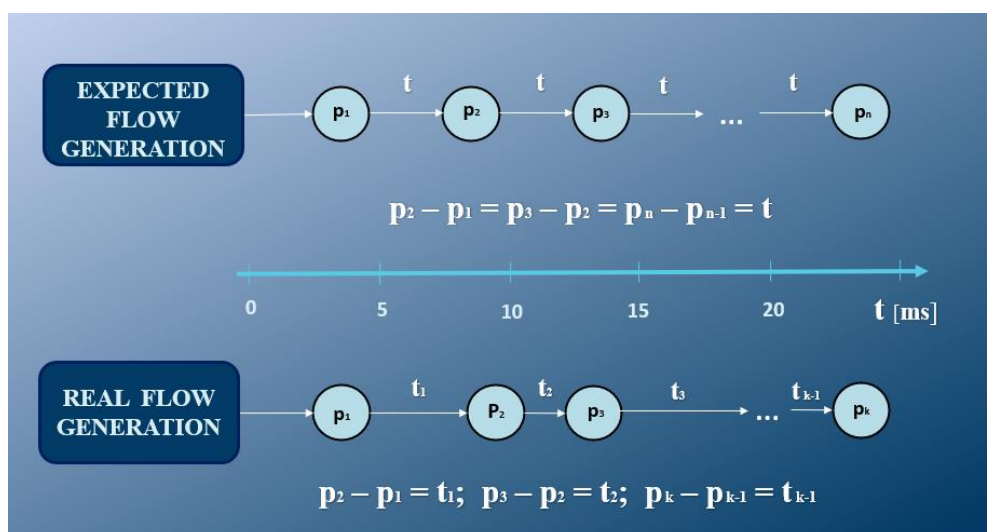
8. Eksperyment ze zmianą parametrów w zakresie jednego przepływu

Ta część badawcza będzie skupiona na sprawdzeniu dokładności zaimplementowanego systemu. Badania polegają na zmianie podstawowych parametrów przepływów:

- odstęp między pakietowy - $T_n [ms]$
- rozmiar wysyłającego pakietu – $L_n [byte]$

Problem jest w tym, że przy idealnych warunkach pracy oczekiwano jest uzyskanie jednakowych odstępów czasowych między wszystkimi wygenerowanymi pakietami. Zaś w rzeczywistości ten odstęp będzie zawsze różny dla kolejnych sekwencji pakietów [29]. Takie zachowanie może być spowodowane różnymi czynnikami, na przykład: sposób implementacji aplikacji oraz innymi czynnikami zewnętrznymi, takimi jak warunki sieciowe, wykorzystywanie zasobów systemu itp. Procesory mają ograniczoną dokładność w zakresie czasu, ponieważ opierają się na sygnałach zegara, które mają pewien stopień niestabilności i zmienności ze względu na różne czynniki fizyczne [30].

Aby zminimalizować wpływ tych czynników, zostały przeprowadzane powtarzalne pomiary i uśredniane wyniki, aby uzyskać bardziej precyzyjne i obiektywne dane. Ponadto, różne warianty implementacji zostały przetestowane, aby zobaczyć, w jaki sposób wpływają one na dokładność systemu. Na koniec, wyniki badań będą porównywane z oczekiwaniami, aby ocenić skuteczność i dokładność zaimplementowanego systemu.



Rysunek 8.1 Schemat, który ilustruje problem badawczy

Statystyki obliczone na podstawie zbioru zebranych danych pozwalają zobaczyć, jak rzeczywiste zachowanie aplikacji odchyła się od zachowania oczekiwanego. Dla zbioru odstępów czasowych między kolejnymi pakietami $T = (t_1, t_2, \dots, t_n)$ zostały obliczone następujące miary dokładności systemu:

- $\mu_T (\mu s)$ – mediana (średnia arytmetyczna dla zbioru danych)
- $\sigma_T^2 (\mu s)^2$ - wariancja
- $\sigma_T (\mu s)$ – odchylenie standardowe
- $CV_T (\mu s)$ – współczynnik zmienności
- $\min_T (\mu s)$ – minimalna wartość ze zbioru
- $\max_T (\mu s)$ – maksymalna wartość ze zbioru.

Listing 8.1 zawiera konfiguracje ustawione w pliku RCR, które zostały użyte do celów tego eksperymentu. Jak można zobaczyć, w tej konfiguracji jest jedna maszyna stanów **fsm1**, w której zadeklarowano dwa stany: **On** i **Off**. Na porcie **port1** zdefiniowana jest reguła, która definiuje generację przepływu o nazwie **f1**, gdy maszyna **fsm1** przebyła w stanie **On**.

Listing 8.1 Zawartość pliku RCR, wykorzystywanego podczas wykonania testów

```

1 {
2     client;
3     104;
4     {E;
5         {TokenIn;i;app/tokenIn}};
6     {S;
7         {fsm1;
8             [{On;;;{TokenIn;Off}},
9             {Off;;;{TokenIn;On}}];
10        On}};
11    {F;
12        {f1;{test; Tn ms; Ln byte}}};
13    {P;
14        {port1;c;U;{8009};{192.168.1.105;8002};fsm1;{On;f1}}};
15    {M;
16        {192.168.1.217;1883}}
17 }
```

Deklaracja przepływu oraz jego parametrów znajduje się w 12 linii pliku. Na czerwono są zaznaczone parametry, które ulegały zmianom na potrzeby testu. Dla przepływów o różnych rozmiarach pakietów $L_{[byte]} = \{1, 512, 1024, 2048, 4096, 8192\}$ został zmieniony odstęp czasowy między pakietami:

- $T_1 = 25 \text{ ms}$
- $T_2 = 5 \text{ ms}$
- $T_3 = 1 \text{ ms}$

Ogólnie, wszystkie generatory strumieni są uruchamiane z domyślnym priorytetem wątku aplikacji. W celu uzyskania jak najlepszych wyników, zostały wykonane testy z zastosowaniem zmienionego priorytetu wątku. Generator ruchu sieciowego, gdzie odstęp między pakietami $T_3 = 1 \text{ ms}$, został zbadany pod kątem wpływu zmiany priorytetu bieżącego wątku.

W przypadku systemu Android, który jest mobilnym systemem operacyjnym opartym na zmodyfikowanej wersji jądra Linux, wszystkie nowe wątki posiadają priorytet domyślny, który ma wartość **NI = 0** [31]. Zwykły Android API pozwala na zmianę priorytetów na poziomie systemu Linux, ale niestety nie pozwala na zmianę trybu planowania przydziału procesora „CPU scheduling”, co pozwoliłoby zbadać NAPES pod kątem różnych trybów pracy samego urządzenia.

Testy ze zmianą priorytetu pozwoliły na ocenę, w jaki sposób zmiana priorytetu wątku wpłynęła na jakość i dokładność przepływu. W celu zapewnienia jednoznacznych wyników, testy te zostały powtórzone kilkakrotnie. W końcowej analizie wyników, będą uwzględnione nie tylko wartości średnie, ale także odchylenie standardowe i inne miary statystyczne, aby lepiej zrozumieć i interpretować wyniki.

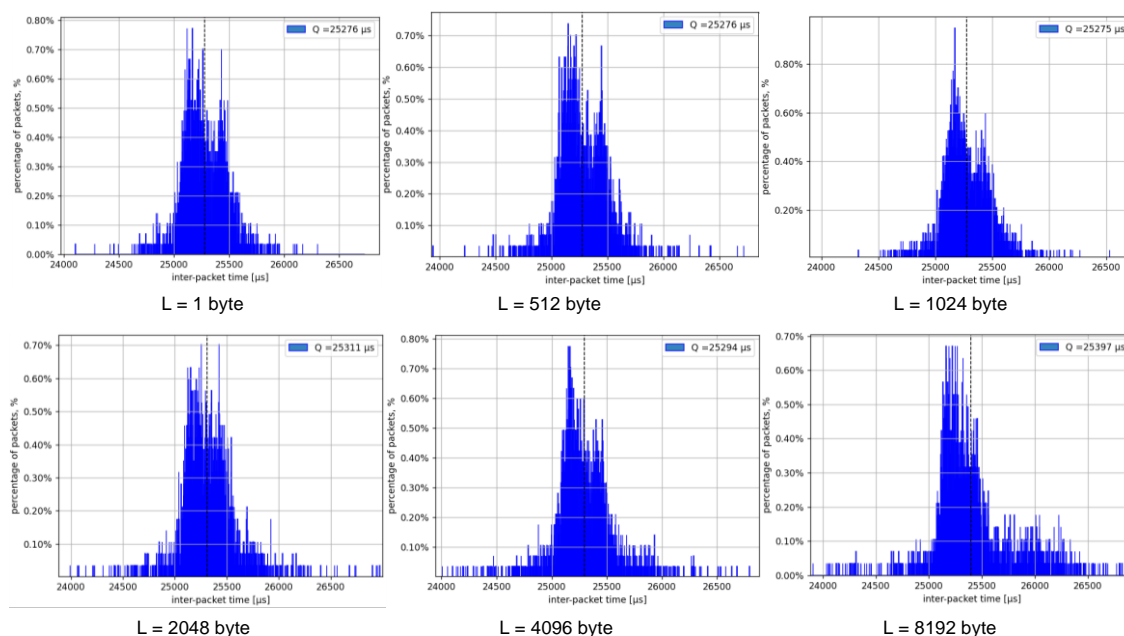
8.1. Zmiana długości pakietów jednego przepływu dla $T = 25$ [ms]

Tabela 8.1 zawiera miary dokładności przepływów o różnych parametrach. Wartości średnie dla tych przepływów μ_T są bardzo bliskie do wartości oczekiwanej 25 ms, ich różnica może sięgać do kilkaset mikrosekund. Można zaobserwować, że wraz ze wzrostem wielkości transferu danych rośnie również średni czas potrzebny na ich wykonanie. Wariancja, odchylenie standardowe i współczynnik zmienności również mają tendencję do zwiększania się wraz ze wzrostem rozmiaru transferu danych.

Tabela 8-1 Statystyki dla $T = 25$ ms i $L = 1/512/1024/2048/4096/8192$ [byte]

Parametry	1 byte	512 byte	1024 byte	2048 byte	4096 byte	8192 byte
$\mu_T(\mu s)$	25 275.77	25 276.12	25 275.36	25 311.32	25 294.14	25 396.66
$\sigma_T^2(\mu s)^2$	39 761.09	50 192.27	44 551.22	85 222.82	81 058.91	177 622.03
$\sigma_T(\mu s)$	199	224	211	292	285	421
$CV_T(\mu s)$	1	1	1	1	1	2
$\min_T(\mu s)$	24 104	23 470	23 046	20 271	23 453	20 427
$\max_T(\mu s)$	26 731	26 995	27 711	30 257	28 711	30 078

Rysunek 8.2 przedstawia histogramy dla różnych L . Wraz ze zwiększeniem rozmiaru pakietu obserwowana jest pogorszona dokładność wysłania, z czego można wnioskować, że system traci więcej czasu na wysłanie większej ilości danych.



Rysunek 8.2 Histogramy dla $T = 25$ ms i $L = 1/512/1024/2048/4096/8192$ [byte]

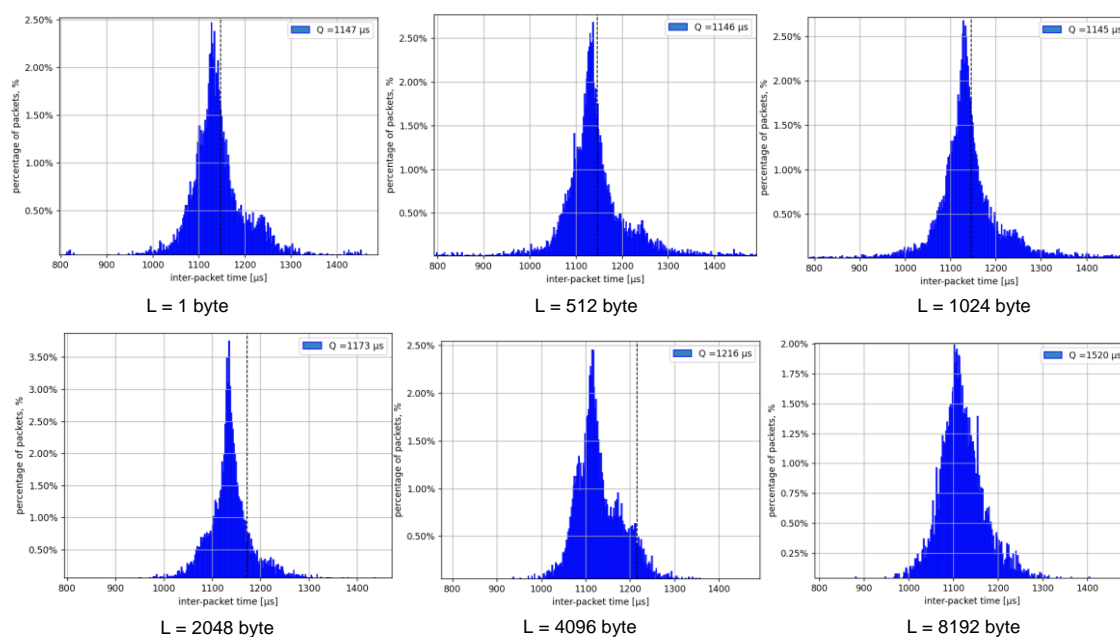
8.2. Zmiana długości pakietów jednego przepływu dla $T = 5$ [ms]

Ten przypadek testowy obejmuje takie same długości pakietów, tylko został zmieniony czas pomiędzy ich wysłaniem $T = 5$ [ms]. Tabela 8.2 przedstawia statystyki zebrane dla tego przypadku testowego.

Tabela 8-2 Statystyki dla $T = 5$ ms i $L = 1/512/1024/2048/4096/8192$ [byte]

Parametry	1 byte	512 byte	1024 byte	2048 byte	4096 byte	8192 byte
$\mu_T(\mu s)$	5 242.78	5 252.1	5 243.92	5 248.95	5 266.32	5307.09
$\sigma_T^2(\mu s)^2$	20 9577.7	102 480.9	35 658.36	56 006.55	84 829.7	1 904 993.1
$\sigma_T(\mu s)$	458	320	189	237	291	1380
$CV_T(\mu s)$	9	6	4	5	6	26
$\min T(\mu s)$	2 735	966	3 032	2 548	326	542
$\max T(\mu s)$	36 937	10 600	9 261	8 448	17 288	88 372

Wyniki tego testu są bardzo podobne do wyników pierwszego przypadku testowego, widać zachowanie tendencji na pogorszenia dokładności wysłania dla większych L . Na rysunku 8.3 są przedstawione histogramy dla poszczególnych długości pakietów.



Rysunek 8.3 Histogramy dla $T = 5$ ms i $L = 1/512/1024/2048/4096/8192$

8.3. Wpływ priorytetu wątków na dokładność generacji ruchu sieciowego

Ten podrozdział ma na celu sprawdzić, jak wpływa priorytezaacja wątków systemu Android na dokładność pracy NAPES. Android API umożliwia zmieniać priorytet wątku bezpośrednio na poziomie systemu Linux za pomocą funkcji:

```
android.os.Process.setThreadPriority(int priority);
```

Jako argument ta funkcja przyjmuje liczby w zakresie **[-20;19]**, gdzie -20 oznacza najwyższy priorytet, a +19 najniższy. Są to tak zwane „nice values” - wartości przestrzeni użytkownika, których można użyć do kontrolowania priorytetu procesu. Domyślnie nowe procesy uruchamiają się o priorytecie **0** [31].

Wadą standardowej biblioteki jest to, że ma ona pewne ograniczenia związane z ustawieniem różnych parametrów wątku (np. nie można zmieniać „scheduling policy”). Począwszy od Androida 5.0, platforma Android implementuje nowy harmonogram wątków oparty na Linux Completely Fair Scheduler (CFS), a korzystanie z „setThreadScheduler()” nie jest już zalecane [32]. Takie ograniczenia zostały wprowadzone w celu zabezpieczenia programisty przed nieporządnym korzystaniem zasobów procesora. Więc w tych badaniach jako parametr procesu, ulegał zmianom tylko priorytet wątku.

W celu obserwacji jakichkolwiek zmian dokładności po zmianie priorytetu, został uruchomiony przepływ o wysokiej częstotliwości generacji pakietów **T = 1 [ms]** (dla zbioru *L* ze wcześniejszych przypadków). Taki sam test został powtórzony dla różnych priorytetów wątku, generującego ruch sieciowy. Najpierw został uruchomiony generator o **domyślnym** priorytecie 0, później o **maksymalnym** i **minimalnym** priorytetami. Poniższe podrozdziały przedstawiają wyniki dla testów opisanych wyżej.

8.3.1. Przepływ generowany przez wątek o domyślnym priorytecie

Dla obserwacji zmian priorytetów zostało wykorzystane narzędzie Android Debug Bridge (adb), które pozwala na uruchomienie wiersza poleceń z poziomu systemu Android. To z kolei pozwala monitorować różne własności wątków w czasie rzeczywistym za pomocą polecenia:

```
adb shell top -H
```

Na poniższym rysunku jest przedstawiony wynik tego polecenia dla przypadku z domyślnym priorytetem. Kolumna **NI** pokazuje priorytet bieżącego wątku, w tym przypadku jest on 0. Także można zobaczyć nazwę wątku i z jakich zasobów systemowych on korzysta [33].

Cmd Командная строка - adb shell top -H

Threads: 2226 total, 14 running, 2212 sleeping, 0 stopped, 0 zombie

Mem: 16E total, 16E used, 162M free, 65M buffers

Swap: 1.5G total, 506M used, 1.0G free, 1.2G cached

800%cpu 22%user 0%nice 50%sys 720%idle 0%iow 0%irq 8%sirq 0%host

TID	USER	PR	NI	VIRT	RES	SHR	S[%CPU]	%MEM	TIME+	THREAD
12247	u0_a176	20	0	1.2G	126M	97M	S 23.3	4.4	0:01.18	Flow T = 1 ms

Rysunek 8.4 Wynik polecenia "top -H" dla przypadku z domyślnym priorytetem

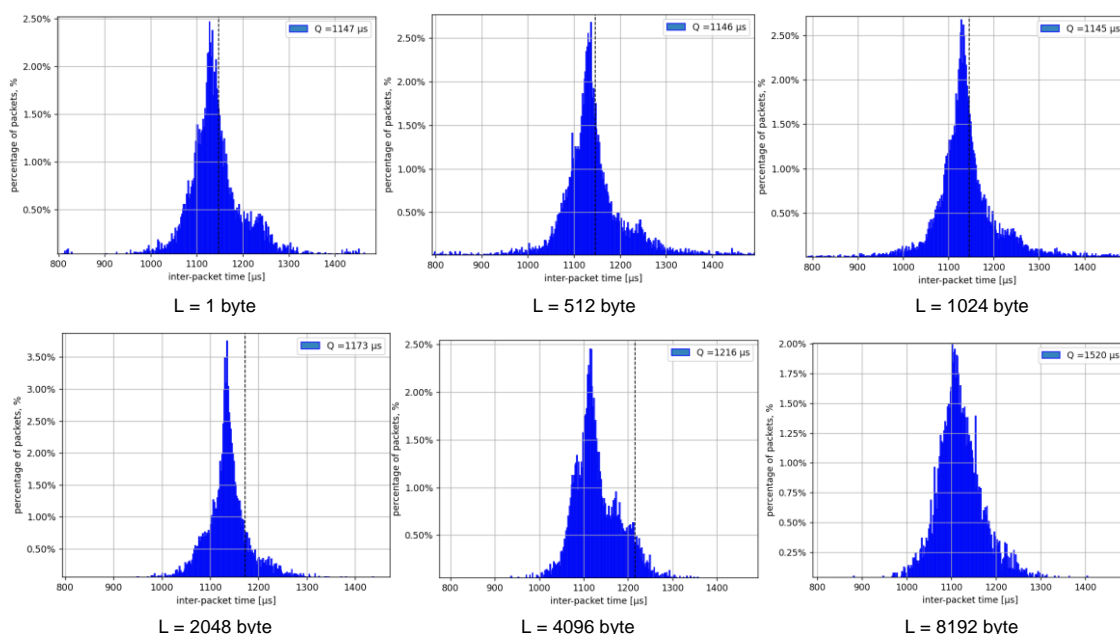
Tabela 8.3 zawiera miary dokładności dla aktualnego przypadku testowego. Widać, że dla *L* = 512 [byte] wartość odchylenia standardowego jest większa niż dla *L* = 1024 [byte].

To mówi o tym, że własność każdego przeprowadzanego testu jest indywidualna. Zatem można wnioskować, że pogorszenie dokładności systemu wraz ze zwiększeniem rozmiaru pakietu jest bardzo nieznacząca dla mniejszych wartości L . Natomiast dla $L > 1024$ [byte] obserwowany jest wzrost wartości odchylenia standardowego. Może to być związane z fragmentacją, dla której w większości przypadków rozmiar MTU (*Maximum Transmission Unit*) wynosi 1500 bajtów, więc fragmentacja zwykle występuje, gdy pakiety przekraczają ten rozmiar.

Tabela 8-3 Statystyki dla $T = 1$ ms; $L = 1/512/1024/2048/4096/8192$ byte; $NI = Q$

Parametry	1 byte	512 byte	1024 byte	2048 byte	4096 byte	8192 byte
$\mu_T(\mu s)$	1 147.25	1 145.73	1 145.44	1 172.64	1 215.90	1 520.03
$\sigma_T^2(\mu s)^2$	13 781.88	28 045.96	15 184.41	1 613 797.3	3 917 606.21	13 829 624.8
$\sigma_T(\mu s)$	117	167	123	1270	979	3719
$CV_T(\mu s)$	10	15	11	108	163	245
$min_T(\mu s)$	229	398	278	348	254	215
$max_T(\mu s)$	5 058	13 770	3 587	93 246	94 914	149 838

Na poniższym rysunku znajdują się histogramy dla przeprowadzonego testu. Patrząc na te histogramy można wnioskować, że nawet dla takiej szybkiej transmisji dokładność wysłanych pakietów jest dość zadowalająca i większość z tych pakietów zostały wysłane zgodnie z oczekiwaniami. Względem innych testów wartość odchyłki jest nawet mniejsza dla małych L , natomiast w przypadkach, gdzie $L > 1024$ wartość odchylenia mocno wzrasta.



Rysunek 8.5 Histogramy dla $T = 1$ ms; $L = 1/512/1024/2048/4096/8192$ byte; $NI = 0$

8.3.2. Przepływ generowany przez wątek o maksymalnym priorytecie

Wątek aplikacji, który generuje przepływ, został uruchomiony z maksymalnym priorytetem systemowym. Na rys. 8.6 widać, że w kolumnie NI wątek posiada wartość -20.

```

Командная строка - adb shell top -H
Threads: 2283 total, 13 running, 2270 sleeping, 0 stopped, 0 zombie
Mem: 16E total, 16E used, 128M free, 66M buffers
Swap: 1.5G total, 506M used, 1.0G free, 1.2G cached
800%cpu 28%user 16%nice 64%sys 679%idle 1%iow 0%irq 12%siq 0%host
TID USER PR NI VIRT RES SHR S [%CPU] %MEM TIME+ THREAD
12753 u0_a176 0 -20 1.2G 126M 97M S 31.0 4.4 0:00.12 Flow T = 1 ms

```

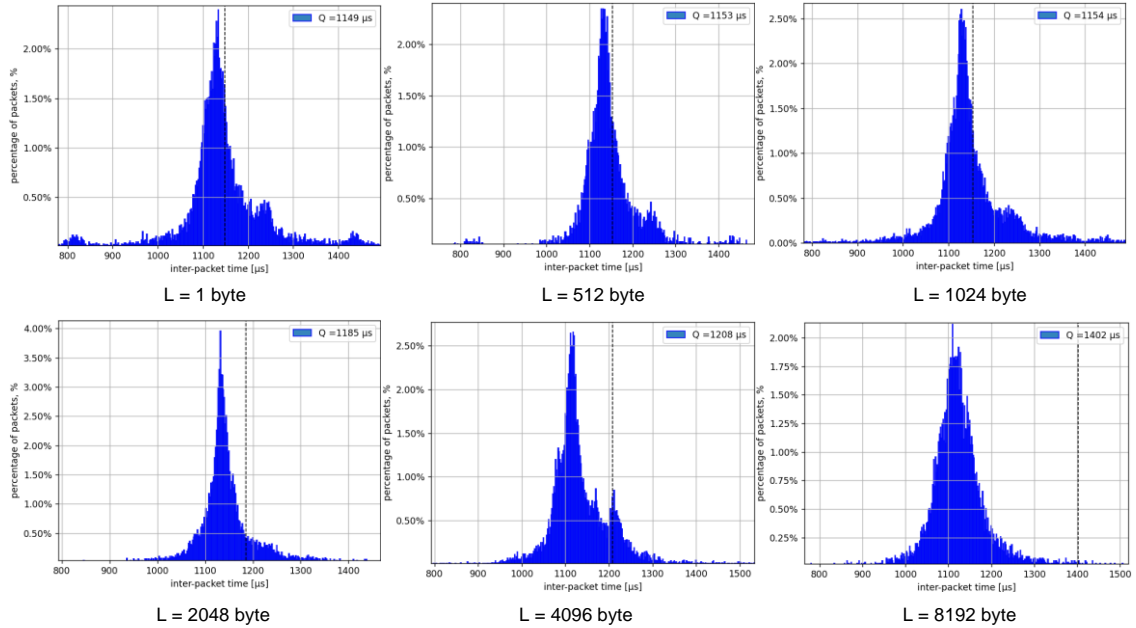
Rysunek 8.6 Wynik polecenia "top -H" dla przypadku z maksymalnym priorytetem

Miary dokładności dla zebranych danych tego testu są przedstawione na tabeli 8.4. Porównując poniższe wyniki z wynikami tabeli 8.3, można zaobserwować, że zmiana priorytetu wątku na wyższy *nie powoduje znacznego polepszenia dokładności wysłania pakietów*. Widać, że nawet dla niektórych przypadków wartość odchylenia jest wyższa, niż w przypadku przepływu uruchomianego z priorytetem domyślnym. Tylko nieznaczne polepszenia wartości średniej jest obserwowane dla przypadków z dużymi rozmiarami pakietów (np. dla $L = 8192$ [byte]). Czyli ustawienie wyższego priorytetu pomaga uzyskać trochę lepsze wyniki dla przypadków, w których zachodzi fragmentacją [34].

Tabela 8-4 Statystyki dla $T = 1$ ms; $L = 1/512/1024/2048/4096/8192$ byte; $NI = -20$

Parametry	1 byte	512 byte	1024 byte	2048 byte	4096 byte	8192 byte
$\mu_T(\mu s)$	1 148.6	1 152.91	1 153.88	1 184.84	1 207.93	1 402.21
$\sigma_T^2(\mu s)^2$	17 807.15	24 804.17	18 456.04	1 415785.8	3 082 003	12 367 037
$\sigma_T(\mu s)^2$	133	157	136	1 190	1755	3 516
$CV_T(\mu s)$	12	14	12	100	145	251
$min_T(\mu s)$	341	276	297	397	289	277
$max_T(\mu s)$	2740	4978	3410	61822	65145	11 0830

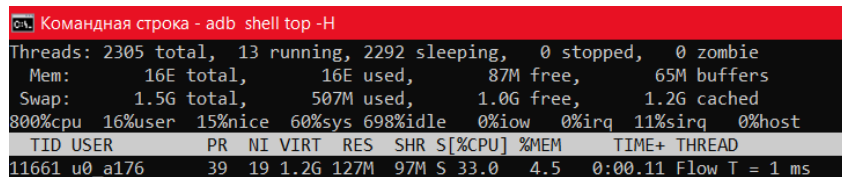
Histogramy dla zebranych danych są przedstawione na poniższym rysunku. Dla przypadku $L = 8192$ [byte] widać, jak nieznacznie poprawiła się sytuacja niż w przypadku z priorytetem domyślnym (większy procent dokładnie wysłanych pakietów). Chociaż, porównując wyniki dla $L = 4096$ [byte] oraz $L = 2048$ [byte] nie jest obserwowana większa dokładność w przypadku wyższego priorytetu, a nawet odwrotnie. Dla przepływu z długością $L = 4096$ [byte]



Rysunek 8.7 Histogramy dla $T = 1$ ms; $L = 1/512/1024/2048/4096/8192$ byte; $NI = -20$

8.3.3. Przepływ generowany przez wątek o minimalnym priorytecie

W tym podrozdziale są przedstawione wyniki dla przepływu generowanym na wątku o najniższym priorytecie (rys. 8.8).



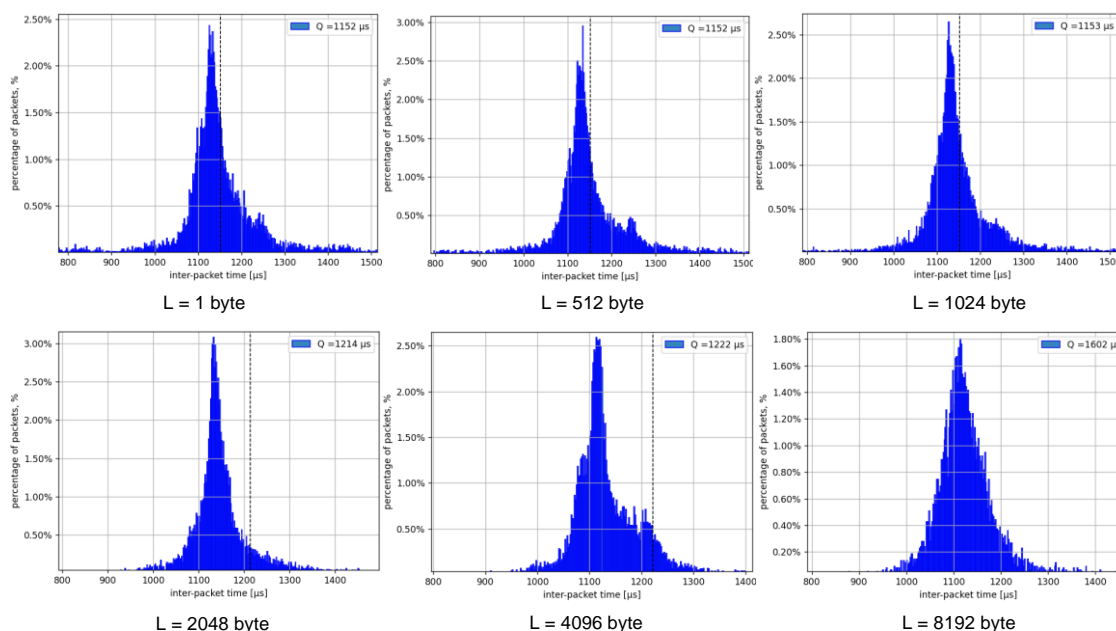
Rysunek 8.8 Wynik polecenia "top -H" dla przypadku z minimalnym priorytetem

Obliczone miary dokładności dla zebranych danych są przedstawione na tabeli 8.5. W porównaniu z poprzednimi przypadkami, gdzie priorytet był wyższy, ten eksperyment pokazuje, że dokładność generacji nieznacznie pogorszyła się dla wszystkich rozmiarów pakietu L . Z tego można wnioskować, że wartość priorytetu ma znaczenie, ale w przypadku jednego przepływu widać, że to znaczenie jest bardzo małe.

Tabela 8-5 Statystyki dla $T = 1$ ms; $L = 1/512/1024/2048/4096/8192$ byte; $NI = 19$

Parametry	1 byte	512 byte	1024 byte	2048 byte	4096 byte	8192 byte
$\mu_T(\mu s)$	1 151.7	1 152.17	1 152.55	1 213.55	1 222.05	1 601.51
$\sigma_T^2(\mu s)^2$	17 089.1	17 190.25	20 184.78	4 212 432.2	3 589 672.4	17 357 245.8
$\sigma_T(\mu s)$	131	131	142	2 052	1 895	4 166
$CV_T(\mu s)$	11	11	12	169	155	260
$\min_T(\mu s)$	343	332	308	374	249	188
$\max_T(\mu s)$	3 009	4 574	5 260	115 146	84 182	15 0339

Poniższy rysunek zawiera histogramy dla zebranych danych. Można tutaj zaobserwować nieznaczne zmniejszenie procentu dokładnie wysłanych pakietów. Na przykład dla $L = 8192$ [byte] zakres procentu dokładnie wysłanych pakietów zmniejszył się o 0.2%, w porównaniu do takiego samego przepływu uruchomionego z priorytetem maksymalnym (rys. 8.7)



Rysunek 8.9 Histogramy dla $T = 1$ ms; $L = 1/512/1024/2048/4096/8192$ byte; $NI = 19$

9. Badania kilku przepływów uruchomionych jednocześnie

Test ma na celu zbadać, jak zmieniają się miary dokładności, gdy zostaną uruchomione wybrane przepływy **jednocześnie**. Założeniem jest to, że generatory wszystkich strumieni **działają w różnych wątkach aplikacji**.

Poniższy listing zawiera konfigurację ustawioną w pliku RCR, który został użyty do celów tego eksperymentu. Zawartość pliku jest bardzo podobna do konfiguracji z poprzedniej części badawczej. Jedyną różnicą jest to, że został dodany **port2**, w którym zdefiniowana reguła na generację przepływu **f2** w momencie, gdy maszyna stanów znajduje się w stanie **on**.

Przepływ można opisać funkcją $F = (T, L)$, gdzie T – czas, a L – rozmiar pakietu.

Listing 9.1 Zawartość pliku RCR, wykorzystywanego podczas wykonania testów

```
1 {
2     client;
3     104;
4     {E;
5         {TokenIn;i;app/tokenIn};
6     {S;
7         {fsm1;
8             [{On;;{TokenIn;Off}},
9             {Off;;{TokenIn;On}}];
10        On}};
11    {F;
12        [{f1;{test; Tn ms; Ln byte }},
13        {f2;{test; Tk ms; Lk byte } }]];
14    {P;
15        [{port1;c;U;{;8009};{192.168.1.105;8002};fsm1;{On;f1}},
16        {port2;c;U;{;8011};{192.168.1.105;8002};fsm1;{On;f2}}]];
17    {M;
18        {192.168.1.217;1883}}
19 }
```

Ogólnie, ta część badawcza składa się z sześciu głównych przypadków testowych:

- Pierwszy przypadek (*symetryczny*) – dwa jednocześnie uruchomione przepływy opisane funkcją $F = (5\text{ms}, 1024)$.
- Drugi przypadek (*symetryczny*) – dwa jednocześnie uruchomione przepływy opisane funkcją $F = (50\text{ms}, 1024)$.
- Trzeci przypadek (*asymetryczny*) - dwa jednocześnie uruchomione przepływy opisane funkcjami: $F_1 = (5\text{ms}, 1024)$, $F_2 = (50\text{ms}, 1024)$.
- Czwarty przypadek (*symetryczny*) – generacja dwóch strumieni $F = (5\text{ms}, 1024)$ na wątkach o różnych priorytetach.
- Piąty przypadek (*symetryczny*) – generacja dwóch strumieni $F = (25\text{ms}, 1024)$ na wątkach o różnych priorytetach.
- Szósty przypadek (*asymetryczny*) – generacja dwóch strumieni opisanych funkcjami: $F_1 = (5\text{ms}, 1024)$, $F_2 = (25\text{ms}, 1024)$ na wątkach o różnych priorytetach.

9.1. Dwa jednocześnie uruchomione przepływy opisane funkcją $F = (5\text{ms}, 1024)$

Dodatkowy przepływ powoduje zwiększenia obciążenia procesora. Test ma na celu sprawdzić, czy praca jednego strumienia nie zakłóca pracy drugiego. W tym przypadku zostały uruchomione dwa przepływy o jednakowych parametrach F (5ms, 1024) oraz priorytetach (rys. 9.1).

```

Komandная строка - adb shell top -H
Threads: 2382 total, 13 running, 2369 sleeping, 0 stopped, 0 zombie
Mem: 16E total, 16E used, 64M free, 56M buffers
Swap: 1.5G total, 485M used, 1.0G free, 1.2G cached
800%cpu 24%user 0%nice 37%sys 737%idle 0%iow 0%irq 2%irq 0%host
TID USER PR NI VIRT RES SHR S[%CPU] %MEM TIME+ THREAD
26383 u0_a176 20 0 1.2G 127M 99M S 8.6 4.5 0:01.34 f2(5 ms, 1024)
26382 u0_a176 20 0 1.2G 127M 99M S 8.3 4.5 0:01.32 f1(5 ms, 1024)

```

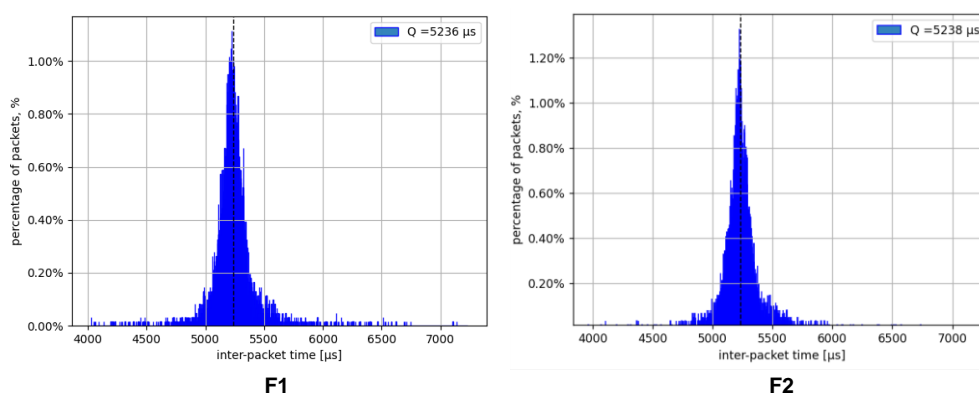
Rysunek 9.1 Wynik polecenia "top -H" podczas generacji dwóch strumieni F (5ms, 1024) o tym samym priorytecie

Tabela 9.1 zawiera wyniki dla zebranych danych. W przypadku, gdy został uruchomiony tylko jeden przepływ o takich samych parametrach wartość średniej wynosi $\mu_T = 5243.92$ [μs] (tabela 8.2). Porównując te wyniki, widać, że generacja dwóch strumieni nie pogorsza dokładności, a ich praca nie zakłóca siebie nawzajem.

Tabela 9-1 Miary dokładności po badaniu symetrycznym, gdzie F (5ms, 1024)

Przepływ	NI	$\mu_T(\mu\text{s})$	$\sigma_T^2(\mu\text{s})^2$	$\sigma_T(\mu\text{s})^2$	$CV_T(\mu\text{s})$	$\min_T(\mu\text{s})$	$\max_T(\mu\text{s})$
F₁ (5ms, 1024)	0	5 236.49	38 486.44	196	4	4 025	7 234
F₂ (5ms, 1024)	0	5 238.18	54 291.40	233	4	245	13 995

Poniższy rysunek przedstawia histogramy dla zebranych danych. Są one bardzo podobne i nie obserwowany jest wpływ dodatkowego obciążenia procesora, czy jakiegokolwiek zakłócenia.



Rysunek 9.2 Histogramy dla przypadku symetrycznego, gdzie F (5ms; 1024)

9.2. Dwa jednocześnie uruchomione przepływy opisane funkcją $F = (25\text{ms}, 1024)$

Żeby upewnić się w tym, że dodatkowe obciążenia procesora nie pogorsza dokładności wysłania, zostały zmienione parametry przepływu - $F(25\text{ms}, 1024)$. Rysunek 9.3 przedstawia parametry wątków, na których odbywa się generacja strumieni.

```

Командная строка - adb shell top -H
Threads: 2313 total, 13 running, 2300 sleeping, 0 stopped, 0 zombie
Mem: 16E total, 16E used, 107M free, 55M buffers
Swap: 1.5G total, 486M used, 1.0G free, 1.2G cached
800%cpu 12%user 0%nice 25%sys 76%idle 0%iow 0%irq 1%irq 0%host
TID USER PR NI VIRT RES SHR S[%CPU] %MEM TIME+ THREAD
26162 u0_a176 20 0 1.2G 129M 98M S 2.6 4.6 0:00.19 f2(25 ms, 1024)
26161 u0_a176 20 0 1.2G 129M 98M S 2.0 4.6 0:00.20 f1(25 ms, 1024)

```

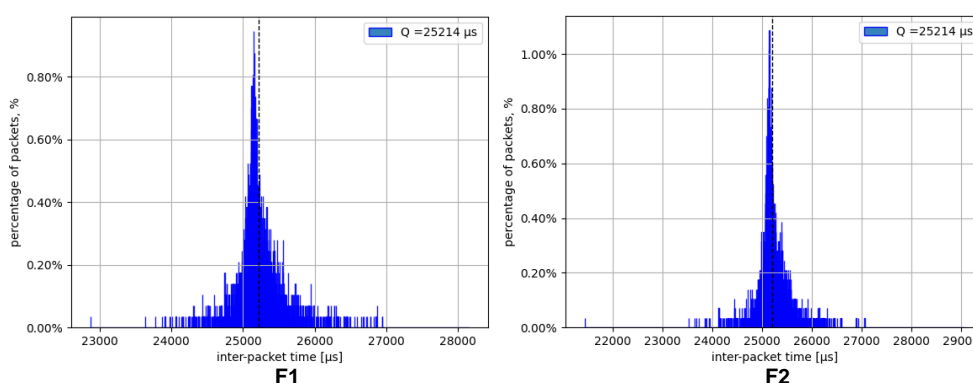
Rysunek 9.3 Wynik polecenia "top -H" podczas generacji dwóch strumieni $F(25\text{ms}, 1024)$ o tym samym priorytecie

Obserwowane miary dokładności są bardzo podobne dla obu strumieni i są przedstawione w tabeli 9.2. Takie podobne wyniki mogą wynikać z tego, że dany przypadek jest symetryczny. Także nie jest obserwowane żadne zakłócenia. Wartości odchylenia są nieco większe w stosunku do odchylenia z tabeli 1.1 dla $L = 1024$, gdzie $\sigma_T = 211 (\mu\text{s})^2$, co może być spowodowane małym obciążeniem.

Tabela 9-2 Miary dokładności po badaniu symetrycznym, gdzie $F(25\text{ms}, 1024)$

Przepływ	NI	$\mu_T(\mu\text{s})$	$\sigma_T^2(\mu\text{s})^2$	$\sigma_T(\mu\text{s})$	$CV_T(\mu\text{s})$	$\min_T(\mu\text{s})$	$\max_T(\mu\text{s})$
F₁ (25ms, 1024)	0	25 214.43	108 591.27	329	1	22 867	28 157
F₂ (25ms, 1024)	0	25 214.39	98 026.70	313	1	21 433	29 084

Dokładność obu przepływów jest prawie identyczna i jest to dobrze widocznym na histogramach poniżej. Tak jak w poprzednim eksperymencie, nie jest obserwowane żadne pogorszenie dokładności.



Rysunek 9.4 Histogramy dla przypadku symetrycznego, gdzie $F(25\text{ms}; 1024)$

9.3. Dwa jednocześnie uruchomione przepływy opisane funkcjami: F1 = (5ms, 1024) i F2 = (25ms, 1024)

Test ma na celu sprawdzić przypadek asymetryczny, to znaczy, że będą generowane dwa przepływy z różnymi odstępami czasowymi między pakietami. Na poniższym rysunku są przedstawione parametry wątków uruchomionych w teście.

```

Komandная строка - adb shell top -H
Threads: 2362 total, 13 running, 2349 sleeping, 0 stopped, 0 zombie
Mem: 16E total, 16E used, 58M free, 56M buffers
Swap: 1.5G total, 484M used, 1.0G free, 1.2G cached
800%cpu 18%user 1%nice 28%sys 75%idle 0%iow 0%irq 3%irq 0%host
TID USER PR NI VIRT RES SHR S [%CPU] %MEM TIME+ THREAD
26429 u0_a176 20 0 1.3G 129M 99M S 9.0 4.5 0:02.55 f1(5 ms, 1024)
26430 u0_a176 20 0 1.3G 129M 99M S 2.0 4.5 0:00.51 f2(25 ms, 1024)

```

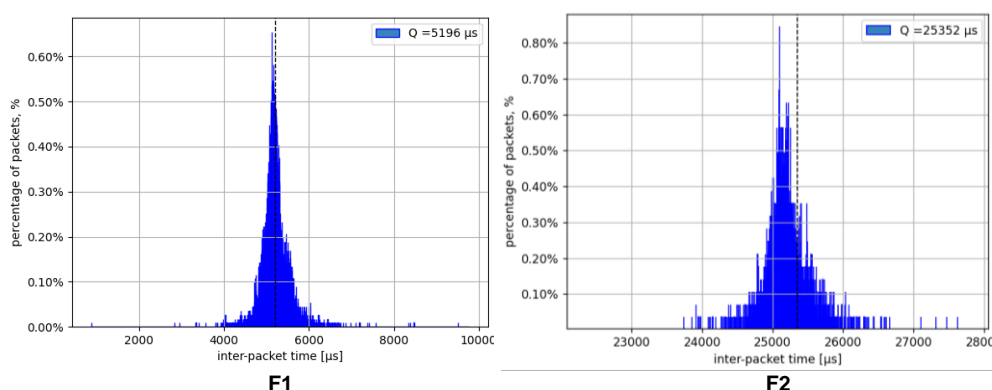
Rysunek 9.5 Wynik polecenia "top -H" podczas testu asymetrycznego z priorytetem domyślnym

Tabela 2.3 zawiera obliczone miary dokładności dla przypadku asymetrycznego. Wartości średnie są podobne do wcześniejszych przypadków. Natomiast dla F₂ obserwowane jest zwiększenie wartości odchylenia standardowego, co może być spowodowane wpływem generacji ruchu o mniejszym odstępie czasowym między pakietami. Z tego można wnioskować, że przepływ o mniejszym odstępie czasowym może częściowo zakłócać pracę generatora F₂ z większym odstępem.

Tabela 9-3 Miary dokładności po badaniu asymetrycznym z priorytetem domyślnym

Przepływ	NI	$\mu_T(\mu s)$	$\sigma_T^2(\mu s)^2$	$\sigma_T(\mu s)$	CV _T (μs)	min _T (μs)	max _T (μs)
F ₁ (5ms, 1024)	0	5 195.73	91 140.06	302	6	867	9 779
F ₂ (25ms, 1024)	0	25 351.58	61 391 729.2	7834	31	23744	441 548

Histogramy poniżej nie wykazują żadnej anomalii podczas trwania symulacji. Widać tylko bardzo małe zwiększenia wartości odchylenia w porównaniu z wcześniejszymi przypadkami.



Rysunek 9.6 Histogramy dla przypadku asymetrycznego z priorytetem domyślnym

9.4. Dwa jednocześnie generowane przepływy przez wątki o różnych priorytetach: $F = (5\text{ms}, 1024)$

Test ma na celu sprawdzić, czy zmieni się zachowanie aplikacji po zmianie priorytetów wątków, które generują ruch. Dla wcześniejszego przypadku symetrycznego z przepływem $F(5\text{ms}, 1024)$ – zostały ustawione priorytety wątków (rys. 9.7). Jak można zobaczyć, wątek, generujący strumień F_1 ma priorytet $NI = -20$, a F_2 $NI = 19$.

```

Выбрать Командная строка - adb shell top -H
Threads: 2289 total, 13 running, 2276 sleeping, 0 stopped, 0 zombie
Mem:      16E total,      16E used,      109M free,      45M buffers
Swap:     1.5G total,     549M used,     987M free,      1.3G cached
800%cpu  47%user  26%nice  83%sys 640%idle  0%iow  0%irq  4%sig  0%host
TID USER      PR  NI VIRT  RES  SHR S[%CPU] %MEM    TIME+  THREAD
31549 u0_a176     0  -20 1.2G 126M 97M S 23.0   4.4   0:00.49 f1(5 ms, 1024)
31550 u0_a176    39   19 1.2G 126M 97M S 22.6   4.4   0:00.49 f2(5 ms, 1024)

```

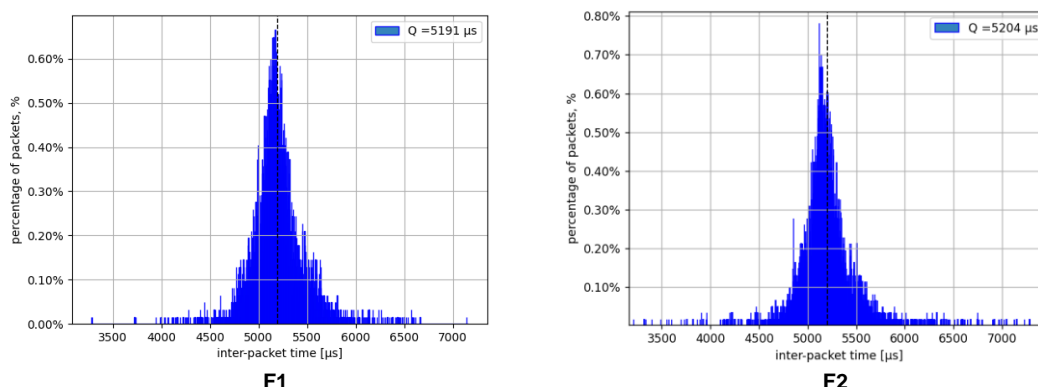
Rysunek 9.7 Wynik polecenia "top -H" podczas generacji dwóch strumieni $F(5\text{ms}, 1024)$ o różnych priorytetach

Tabela 2.4 zawiera miary dokładności dla zebranych danych. Widać, że wyniki dla przepływu o wyższym priorytecie są bardziej dokładne w stosunku do wyników przepływu o niższym priorytecie. Choć pogorszenie dokładności jest bardzo małe, można stwierdzić, że ustawianie priorytetów ma wpływ na wyniki końcowe. Eksperyment został powtórzony kilka razy i po każdym teście były otrzymywane podobne wyniki.

Tabela 9-4 Miary dokładności po badaniu symetrycznym z różnymi priorytetami $F(5\text{ms}, 1024)$

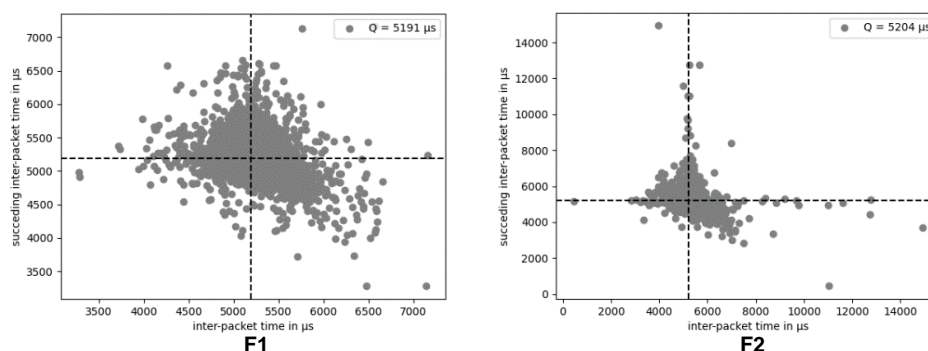
Przepływ	NI	$\mu_T(\mu\text{s})$	$\sigma_T^2(\mu\text{s})^2$	$\sigma_T(\mu\text{s})^2$	$CV_T(\mu\text{s})$	$\min_T(\mu\text{s})$	$\max_T(\mu\text{s})$
$F_1(5\text{ms}, 1024)$	-20	5 190.94	68 025.39	261	5	3 276	7 163
$F_2(5\text{ms}, 1024)$	19	5 204.28	160 257.34	400	8	441	14 936

Histogramy wraz z wykresami punktowymi dla danego testu są przedstawione na poniższym rysunku. Porównując wykresy punktowe widać, że w przypadku niższego priorytetu uzyskiwane są znacznie większe odstępy czasowe między wysłaniem kolejnych pakietów.



Rysunek 9.8 Histogramy dla przypadku symetrycznego z różnymi priorytetami $F(5\text{ms}, 1024)$

Rysunek 9.9 Wykresy punktowe dla przypadku symetrycznego z różnymi priorytetami $F(5\text{ms}, 1024)$



9.5. Dwa jednocześnie generowane przepływy przez wątki o różnych priorytetach: $F = (25\text{ms}, 1024)$

Własności tego testu są analogiczne do poprzedniego z rozdziału 2.4, gdzie generatory obu przepływów zostały uruchomione o różnych priorytetach wątków. W tym przypadku generowany jest przepływ F z większym odstępem czasowym, który opisany **jest zestawem parametrów $(25\text{ms}, 1024)$** . Na poniższym rysunku przedstawione parametry wątków obserwowane podczas symulacji:

```

Komandная строка - adb shell top -H
Threads: 2358 total, 14 running, 2344 sleeping, 0 stopped, 0 zombie
Mem: 16E total, 16E used, 40M free, 47M buffers
Swap: 1.5G total, 547M used, 989M free, 1.3G cached
800%cpu 23%user 5%nice 26%sys 744%idle 0%iow 0%irq 2%irq 0%host

```

TID	USER	PR	NI	VIRT	RES	SHR	S[%CPU]	%MEM	TIME+	THREAD
32497	shell	20	0	16M	6.7M	3.1M	R 22.6	0.2	0:10.54	top
32667	u0_a176	39	19	1.2G	126M	97M	S 5.0	4.4	0:01.59	f2(25 ms, 1024)
32666	u0_a176	0	-20	1.2G	126M	97M	S 4.3	4.4	0:01.59	f1(25 ms, 1024)

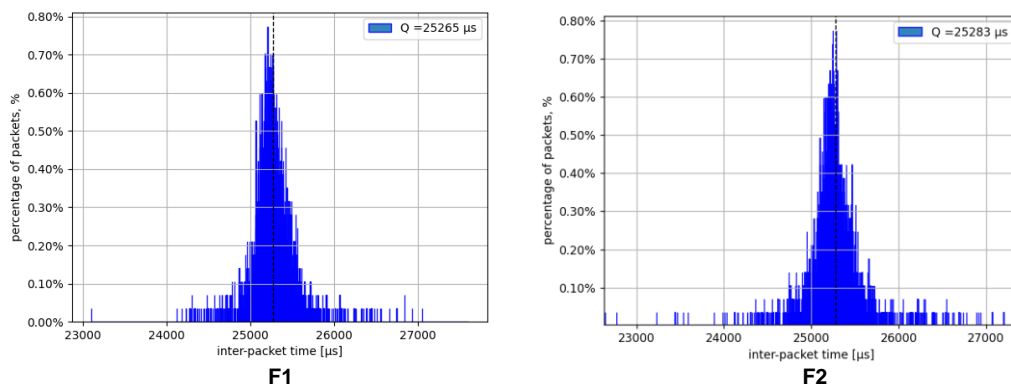
Rysunek 9.10 Wynik polecenia "top -H" podczas generacji dwóch strumieni $F(25\text{ms}, 1024)$ o różnych priorytetach

Wyniki przeprowadzonego testu są przedstawione na tabeli 2.5. Obserwując te pomiary, można zauważyć, że generacja ruchu z mniejszym priorytetem powoduje znaczne zwiększenie wartości odchylenia standardowego (wartość odchylenia dla F_2 jest większa w 2 razy) w stosunku do wartości odchylenia dla F_1 .

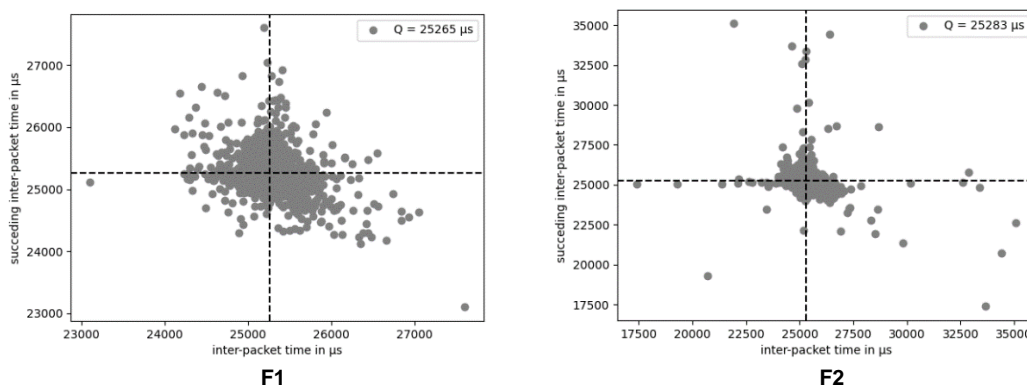
Tabela 9-5 Miary dokładności po badaniu symetrycznym z różnymi priorytetami $F(25\text{ms}, 1024)$

Przepływ	NI	$\mu_T(\mu\text{s})$	$\sigma_T^2(\mu\text{s})^2$	$\sigma_T(\mu\text{s})^2$	$CV_T(\mu\text{s})$	$\min_T(\mu\text{s})$	$\max_T(\mu\text{s})$
F₁ (25ms, 1024)	-20	25 265.30	64 781.47	254	1	23 100	27 601
F₂ (25ms, 1024)	19	25 282.78	337 254.71	581	2	17 390	35 089

Na poniższych rysunkach są przedstawione histogramy oraz wykresy punktowe dla przeprowadzonego testu. Obserwując histogram dla F_2 , można zauważyć pojawienie więcej małych słupków obok wartości średniej, co mówi o większym odchyleniu w porównaniu z wynikami dla F_1 . Na wykresach punktowych dobrze widać, jak zmniejszyła się dokładność w przypadku generacji przepływu F_2 (pojawiają się punkty w większym zakresie).



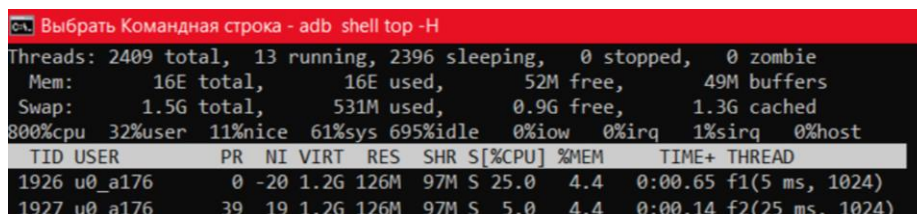
Rysunek 9.11 Histogramy dla przypadku symetrycznego z różnymi priorytetami F (25ms, 1024)



Rysunek 9.12 Wykresy punktowe dla przypadku symetrycznego z różnymi priorytetami F (25ms, 1024)

9.6. Dwa jednocześnie generowane przepływy przez wątki o różnych priorytetach: $F_1 = (5\text{ms}, 1024)$ oraz $F_2 = (25\text{ms}, 1024)$

Ten przypadek ma na celu zbadać, jak zmieni się zachowanie systemu dla testu asymetrycznego po zmianie priorytetów wątków. Generator przepływu F_1 został uruchomiony z maksymalnym priorytetem wątku, a generator F_2 startował z minimalnym priorytetem (rys. 9.13).



Rysunek 9.13 Wynik polecenia "top -H" podczas testu asymetrycznego z różnymi priorytetami wątków

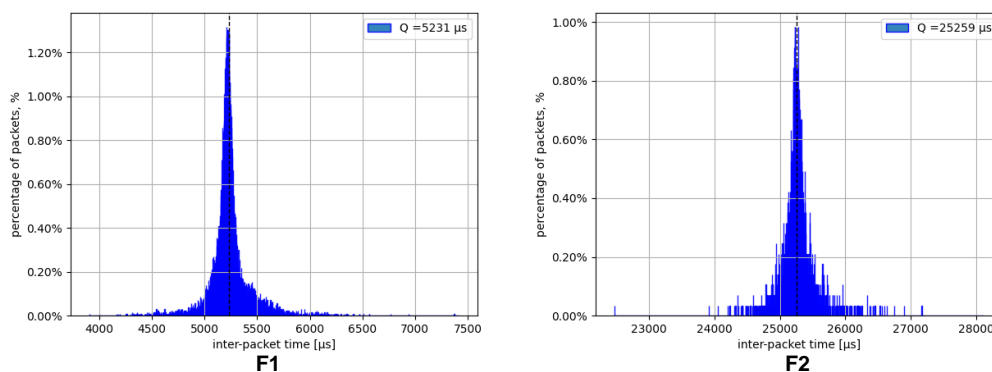
Poniższa tabela przedstawia wyniki dla przeprowadzonego testu. W porównaniu do takiego samego testu uruchomionego z priorytetami domyślnymi (podrozdział 9.3), obserwowane jest znaczne polepszenie dokładności w przypadku przepływu F_2 (wartość odchylenia jest o wiele mniejsza). Także zgodnie z oczekiwaniami [35]: generator o

wyższym priorytecie wysyła pakiety dokładniej niż to robi generator z minimalnym priorytetem. Wnioskując te wyniki można powiedzieć, że da się uzyskać większą dokładność wysłania pakietów ustawiając różne priorytety wątków, które generują ruch.

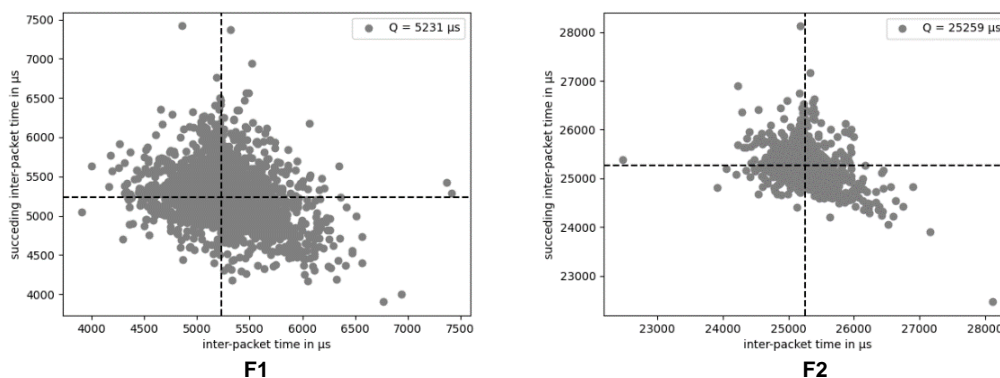
Tabela 9-6 Miary dokładności po badaniu asymetrycznym z różnymi priorytetami

Przepływ	NI	$\mu_T(\mu s)$	$\sigma_T^2(\mu s)^2$	$\sigma_T(\mu s)$	CV _T (μs)	min _T (μs)	max _T (μs)
F ₁ (25ms, 1024)	-20	5 231.36	31 344.13	177	3	3 902	7 419
F ₂ (25ms, 1024)	19	25 258.83	62 145.83	249	1	22 463	28 123

Na poniższych rysunkach znajdują się histogramy oraz wykresy punktowe dla przeprowadzonego testu asymetrycznego. Widać, że większość pakietów zostały wysłane z bardzo małym odchyleniem dla obu przepływów. Wykresy punktowe wykazują w miarę porządną dokładność w porównaniu do innych prowadzonych testów (nawet w przypadku przepływu F₂, dokładność którego była o wiele mniejsza przy uruchomieniu z priorytetem domyślnym).



Rysunek 9.14 Histogramy dla przypadku asymetrycznego z różnymi priorytetami



Rysunek 9.15 Wykresy punktowe dla przypadku asymetrycznego z różnymi priorytetami

10. Podsumowanie i kierunki dalszych prac

Celą tej pracy była implementacja oraz gruntowne przetestowanie systemu, generującego ruch sieciowy na podstawie podanych parametrów wejściowych. Założeniem było to, że aplikacja musi powstać dla systemów Android. W pierwszej kolejności, aby rozpocząć implementację, została podjęta analiza wymagań dotycząca projektowania i implementacji aplikacji. Następnie, po dokładnej analizie wymagań, należało przystąpić do projektowania i implementacji aplikacji. W tym celu, należało zdefiniować architekturę systemu, wybrać odpowiednie narzędzia i biblioteki oraz zaimplementować funkcjonalność generowania ruchu sieciowego.

Po zakończeniu implementacji, należało przeprowadzić szereg testów jakości, aby upewnić się, że aplikacja jest stabilna i spełnia wymagania. Następnie, należało przeprowadzić takie testy ruchu sieciowego, aby sprawdzić, czy aplikacja jest w stanie generować ruch sieciowy zgodnie z podanymi parametrami wejściowymi. Po badaniach wstępnych, została podjęta decyzja o badaniach skupionych wyłącznie na *dokładności generacji pakietów*.

W związku z decyzją o badaniach dokładności generacji pakietów, zostały przeprowadzone dodatkowe testy, których celem było dokładne sprawdzenie tej funkcjonalności. Testy te były oparte na porównaniu czasów wysłania wygenerowanych pakietów z czasami, które były oczekiwane. Wszystkie testy były powtarzane kilkakrotnie w celu uzyskania wyników reprezentatywnych.

Po upewnieniu się, że system zachowuje się poprawnie, została zbadana dokładność systemu pod kątem zmiany parametrów sieciowych oraz systemowych. Celem tych badań było sprawdzenie, jak zmieni się zachowanie systemu w innych warunkach pracy aplikacji. W tym celu zostały przeprowadzone badania, w których zmianom ulegały parametry przepływu: częstotliwość generacji pakietów oraz rozmiar wysyłanych pakietów.

Po zakończeniu tych badań, zostały przeprowadzone analizy wyników, po których zostało zaobserwowane, że zmiana tych parametrów praktycznie nie ma wpływu na dokładność generacji ruchu sieciowego. Tylko w przypadku wysłania pakietów o dużym rozmiarze powodowało częściowe zakłócenia dokładności. Jest to raczej związane z fragmentacją pakietów w sieci użytkowej.

Podsumowując wszystkie testy i obserwacje, została podjęta decyzja o ponownych przeprowadzeniach testów, zmieniając parametry systemowe na urządzeniu Android. W tym celu część testów została przeprowadzona ponownie z warunkiem zmiany priorytetu wątku, generującego ruch sieciowy.

Po analizie testów przeprowadzonych ze zmienionym priorytetem zostało stwierdzone, że zmiana priorytetu wątku miała wpływ na dokładność generacji ruchu sieciowego, jednak nie był to efekt jednoznaczny [37]. W niektórych przypadkach zmiana priorytetu poprawiała dokładność generacji, jednak w innych pogorszyła ją. To może być spowodowane przez wielu czynników, takich jak konkurencja o zasoby i wydajność, ograniczenia sprzętowe i systemowe.

W rezultacie, ustawienie wysokiego priorytetu dla wątku niekoniecznie gwarantuje poprawę wydajności, a może nawet prowadzić do jej pogorszenia [38]. Podczas generowania ruchu sieciowego w wątku o najwyższym priorytecie możliwe jest, że inne wątki o wysokim priorytecie również konkurują o zasoby i powodują dodatkowe zakłócenia, co prowadzi do zmniejszenia dokładności. Ponadto, jeśli system jest już obciążony zadaniami i dostępne są ograniczone zasoby sprzętowe, ustawienie wyższego priorytetu dla wątku niekoniecznie musi skutkować poprawą dokładności działania. Także

wątki o wysokim priorytecie mogą powodować częstsze przełączanie kontekstu i opróżnianie pamięci podręcznej, co prowadzi do wyższego obciążenia i niższej wydajności.

Wyniki moich badań wykazały, że zmiana priorytetu nie ma łatwego do przewidzenia wpływu na dokładność transmisji pakietów. Wpływ ten zależy od wielu czynników i jest trudny do określenia.

Podsumowując, badania wpływu zmian różnych parametrów na dokładność generowanego przepływu ruchu ujawniło kilka ważnych wniosków. Po pierwsze, zmiana długości wysyłanego pakietu ma ograniczony wpływ na dokładność generowanego ruchu. Jednak zmiana priorytetu wątku generującego przepływ ruchu ma znacznie bardziej złożony i nieprzewidywalny wpływ na dokładność. Wyniki pokazały, że poprawa lub pogorszenie dokładności zależy od wielu czynników. Stwierdzono, że ustawienie priorytetu wątku generowania ruchu na najwyższy poziom niekoniecznie musi prowadzić do poprawy wydajności.

Podsumowując, testy i obserwacje wykonane w celu sprawdzenia wpływu zmian parametrów na dokładność generacji ruchu sieciowego wykazały, że zmiany te nie mają znaczącego wpływu na dokładność, jednak istnieją jeszcze metody, które pozwoliłyby sprawdzić ten temat głębiej. Jedną z takich metod może być ustawienie innego algorytmu szeregowania na procesorze. To spowoduje inny sposób ustalania priorytetów i przydzielania czasu procesora do różnych wątków, co może mieć wpływ na dokładność czasów wysyłania pakietów. Niestety w przypadku systemów Android starszych niż wersja Android 5.0 nie ma możliwości, ustawić trybu dyspozytora na procesorze. Potrzebne są dalsze badania, aby w pełni zrozumieć wpływ zmiany priorytetów na dokładność generowanego przepływu.

Literatura

- [1] K. K. Patel, S. M. Patel, et al., "Internet of things IOT: definition, characteristics, architecture, enabling technologies, application future challenges," International journal of engineering science and computing, 2016, s. 6122–6131.
- [2] <https://insights2techinfo.com/iot-and-its-uses-in-security-surveillance/>, dostęp z dnia 21.01.2023.
- [3] J. Choque, R. Agüero, Z. Kopertowski i in., "FLEXNET: Flexible Networks for IoT-based services", w 2020 23rd International Symposium on Wireless Personal Multimedia Communications (WPMC), 2020, s. 1–6, DOI: 10.1109/WPMC50192.2020.9309486.
- [4] Partnerzy projektu FLEXNET, FLEXNET Deliverable 1.1 Use cases and initial architecture, lip. 2019, s. 47–66.
- [5] J. Domaszewicz i A. Bąk, "Koncepcja emulatora aplikacji Internetu Rzeczy dla projektu FLEXNET", w Prace Naukowe Wydziału Elektroniki i Technik Informacyjnych Politechniki Warszawskiej, t. 1, A. Jakubiak, 2020, s. 9–15.
- [6] Partnerzy projektu FLEXNET, FLEXNET Deliverable 1.1b Use cases and final architecture, lip. 2020, s. 24–52.
- [7] N. Piratla, A. Jayasumana, H. Smith, "Overcoming the Effects of Correlation in Packet Delay Measurements Using Inter-Packet Gaps", Proceedings. 2004 12th IEEE International Conference on Networks (ICON 2004), 2005, s. 1-6.
- [8] N. Feamster, J. Livingood, "Measuring internet speed: current challenges and future recommendations", Communications of the ACM, 2020, s. 72-80, DOI: 10.1145/3372135.
- [9] <https://www.speedtest.pl/wiki/jak-dziala-speed-test>, dostęp z dnia 21.01.2023.
- [10] J. Dugan, "Iperf Tutorial", Columbus: Summer JointTechs, 2010, s. 1-23.
- [11] J. Domaszewicz i A. Bąk, Prezentacja NAPES - FLEXNET Final Review, 2022.
- [12] T. Alstad, J. Riley Dunkin i in., "Game network traffic simulation by a custom bot", w 2015 Annual IEEE Systems Conference (SysCon) Proceedings, 2015, s. 1-6, DOI: 10.1109/SYSCON.2015.7116828.
- [13] Rafał Sztelmach, "Narzędzia do translacji i integracji komponentów aplikacji IoT", praca dyplomowa inżynierska, WEiTI PW, Warszawa 2022.
- [14] <https://bnfplayground.pauliankline.com/>, dostęp z dnia 21.01.2023.
- [15] <https://developer.android.com/studio>, dostęp z dnia 21.01.2023.
- [16] Esmond Pitt, "Fundamental Networking in Java", Springer, 2010, s 1-64
- [17] Gastón C. Hillar, "MQTT Essentials - A Lightweight IoT Protocol", Packt Publishing Ltd., Birmingham, 2017, s. 8-57
- [18] <https://www.comfortclick.com/News/Show/93>, dostęp z dnia 21.01.2023.
- [19] <https://test.mosquitto.org/>, dostęp z dnia 21.01.2023.
- [20] <https://mosquitto.org/>, dostęp z dnia 11.02.2023.
- [21] <https://www.eclipse.org/paho/files/javadoc/index.html>, dostęp z dnia 11.02.2023.

- [22] M Naser, Q Abu Al-Haija, "Spyware Identification for Android Systems Using Fine Trees", Information, 2023, <https://doi.org/10.3390/info14020102>
- [23] E. Farchi, Y. Krasny i Y. Nir, "Automatic Simulation of Network Problems in UDP-Based Java Programs", 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings, 2004, s. 1-8, DOI: 10.1109/IPDPS.2004.1303342
- [24] Kenneth L. Calvert, Michael J. Donahoo, "TCP/IP Sockets in Java: Practical Guide for Programmers", Morgan Kaufmann, 22.02.2008, s. 6-47
- [25] G. Wagner, "An abstract state machine semantics for discrete event simulation", 2017 Winter Simulation Conference (WSC), 2018, s. 1-12, DOI: 10.1109/WSC.2017.8247830
- [26] C. Cummings, "Simulation and Synthesis Techniques for Asynchronous FIFO Design", 2002 Synopsys Users Group Conference, 2002, s. 1-23
- [27] <https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6lOnSsKchNAySU/preview>, dostęp z dnia 11.02.2023.
- [28] K. Lai, M. Baker, "Measuring Bandwidth", w IEEE INFOCOM '99. Conference on Computer Communications. Proceedings, 2002, s. 235-245, DOI: 10.1109/INFCOM.1999.749288
- [29] M. Jaber, R. G. Cascella i C. Barakat, "Can We Trust the Inter-Packet Time for Traffic Classification?", 2011 IEEE International Conference on Communications (ICC), 2011, st. 1-5, DOI: 10.1109/icc.2011.5963024
- [30] L. Perneel, H. Fayyad-Kazan i M. Timmerman, "Can Android be used for real-time purposes?", 2012 International Conference on Computer Systems and Industrial Informatics, 2013, st. 1-6, DOI: 10.1109/ICCSII.2012.6454350.
- [31] [https://en.wikipedia.org/wiki/Nice_\(Unix\)](https://en.wikipedia.org/wiki/Nice_(Unix)), dostęp z dnia 11.02.2023.
- [32] A. Ruiz, M. Rivas, M. Harbour i in., "CPU Isolation on the Android OS for running Real-Time Applications", Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems, 2015, st. 1-7, DOI: 10.1145/2822304.2822317.
- [33] J. Han i S. Lee, "Performance Improvement of Linux CPU Scheduler Using Policy Gradient Reinforcement Learning for Android Smartphones", IEEE Access, 10.01.2020, s. 1-15, DOI: 10.1109/ACCESS.2020.2965548
- [34] G. Huston, "A Tale of Two Protocols: IPv4, IPv6, MTUs and Fragmentation", The ISP Column, 2009, st. 1-29.
- [35] C. Lian, Y. Quansheng, "Thread Priority Sensitive Simultaneous MultiThreading Fair Scheduling Strategy", w 2009 International Conference on Computational Intelligence and Software Engineering, 2009, s. 1-4, DOI: 10.1109/CISE.2009.5362844
- [36] S. Hemminger, "Network emulation with NetEm", Linux conf au, 2005, st. 1-9
- [37] S. Hahn, S. Lee i I. Yee, "Improving User Experience of Android Smartphones Using Foreground App-Aware I/O Management", Proceedings of the 8th Asia-Pacific Workshop on Systems, 2017, st. 1-8, DOI: 10.1145/3124680.3124721.
- [38] G. Stuer, K. Vanmechelen i in., "Sleeping in Java", Dept. of Mathematics and Computer Science, 2020, s. 1-5

Spis rysunków

Rysunek 1.1 Rozwój IoT na osi czasu [2]	10
Rysunek 1.2 Podstawowa zasada działania systemu NAPES	11
Rysunek 1.3 Model komponentu NAPES	12
Rysunek 2.1 Podstawowy generator ruchu, służący do pomiaru prędkości Internetu [9]	15
Rysunek 2.2 Widok terminali z klienta Iperf	16
Rysunek 2.3 Widok z terminali serwera Iperf	16
Rysunek 2.4 Widok klienta z aplikacji NAPES	16
Rysunek 2.5 Widok serwera z aplikacji NAPES	16
Rysunek 3.1 Architektura systemu NAPES [4]	17
Rysunek 3.2 Przykładowy ułamek pliku RCR	18
Rysunek 4.1 Schemat działania protokołu MQTT [18]	20
Rysunek 4.2 Tworzenie obiektu klienta poprzez łączenia klienta do brokera MQTT	21
Rysunek 4.3 Widok głównego menu PCAPdroid	21
Rysunek 5.1 Model komponentu NAPES	22
Rysunek 5.2 Przykład działania zdarzeń lokalnych	23
Rysunek 6.1 Tworzenie nazwy użytkownika oraz jego hasło	24
Rysunek 6.2 Zawartość pliku wygenerowanego pliku „passwd”, zawierającego dane uwierzytelniające do brokera	24
Rysunek 6.3 Zawartość pliku „mosquitto.conf”	24
Rysunek 6.4 Polecenie służące do tworzenia reguły zapory	25
Rysunek 6.5 Uruchomienie usługi Mosquitto Broker	25
Rysunek 6.6 Widok okna konfiguracji programu MQTT.fx	26
Rysunek 6.7 Wysłanie wiadomości na topic „test/”	26
Rysunek 6.8 Widok ze strony subskrybenta	26
Rysunek 6.9 Badanie aplikacji klienckiej	28
Rysunek 6.10 Strona ustawień połączeniowych	30
Rysunek 6.11 Struktura drzewa projektu	30
Rysunek 6.12 Widok aplikacji po wysłaniu wiadomości UDP	32
Rysunek 6.13 Widok z konsoli serwera po otrzymaniu wiadomości UDP	32
Rysunek 6.14 Widok z konsoli serwera po otrzymaniu wiadomości UDP	33
Rysunek 6.15 Widok aplikacji po stronie użytkownika	35
Rysunek 6.16 Widok z terminali po stronie serwera	35
Rysunek 6.17 Widok z programu Wireshark po wysłaniu wiadomości	36

Rysunek 6.18 Widok z okna terminali po wczytywaniu pliku RCR	40
Rysunek 6.19 Schemat wątków aplikacji	41
Rysunek 6.20 Przykładowy JSON z logami NAPES	47
Rysunek 6.21 Przykład wizualizacji logów	47
Rysunek 7.1 Zebrany ruch sieciowy po wykonaniu emulacji.....	49
Rysunek 7.2 Wykres odstępów czasowych pomiędzy kolejnymi pakietami UDP	49
Rysunek 7.3 Wykres odstępów czasowych pomiędzy kolejnymi pakietami TCP	50
Rysunek 7.4 Wykres odstępów czasowych pomiędzy kolejnymi pakietami UDP na porcie serwerowym	51
Rysunek 7.5 Wizualizacja logów za pomocą Trace Event Visualisation	51
Rysunek 8.1 Schemat, który ilustruje problem badawczy	52
Rysunek 8.2 Histogramy dla $T = 25$ ms i $L = 1/512/1024/2048/4096/8192$ [byte]	54
Rysunek 8.3 Histogramy dla $T = 5$ ms i $L = 1/512/1024/2048/4096/8192$	55
Rysunek 8.4 Wynik polecenia "top -H" dla przypadku z domyślnym priorytetem	56
Rysunek 8.5 Histogramy dla $T = 1$ ms; $L = 1/512/1024/2048/4096/8192$ byte; $NI = 0$	57
Rysunek 8.6 Wynik polecenia "top -H" dla przypadku z maksymalnym priorytetem.....	58
Rysunek 8.7 Histogramy dla $T = 1$ ms; $L = 1/512/1024/2048/4096/8192$ byte; $NI = -20$..	59
Rysunek 8.8 Wynik polecenia "top -H" dla przypadku z minimalnym priorytetem	59
Rysunek 8.9 Histogramy dla $T = 1$ ms; $L = 1/512/1024/2048/4096/8192$ byte; $NI = 19$...	60
Rysunek 9.1 Wynik polecenia "top -H" podczas generacji dwóch strumieni F (5ms, 1024) o tym samym priorytecie.....	62
Rysunek 9.2 Histogramy dla przypadku symetrycznego, gdzie F (5ms; 1024)	62
Rysunek 9.3 Wynik polecenia "top -H" podczas generacji dwóch strumieni F (25ms, 1024) o tym samym priorytecie.....	63
Rysunek 9.4 Histogramy dla przypadku symetrycznego, gdzie F (25ms; 1024).....	63
Rysunek 9.5 Wynik polecenia "top -H" podczas testu asymetrycznego z priorytetem domyślnym	64
Rysunek 9.6 Histogramy dla przypadku asymetrycznego z priorytetem domyślnym.....	64
Rysunek 9.7 Wynik polecenia "top -H" podczas generacji dwóch strumieni F (5ms, 1024) o różnych priorytetach	65
Rysunek 9.8 Histogramy dla przypadku symetrycznego z różnymi priorytetami F (5ms, 1024).....	65
Rysunek 9.9 Wykresy punktowe dla przypadku symetrycznego z różnymi priorytetami F (5ms, 1024).....	66
Rysunek 9.10 Wynik polecenia "top -H" podczas generacji dwóch strumieni F (25ms, 1024) o różnych priorytetach	66

Rysunek 9.11 Histogramy dla przypadku symetrycznego z różnymi priorytetami F (25ms, 1024)	67
Rysunek 9.12 Wykresy punktowe dla przypadku symetrycznego z różnymi priorytetami F (25ms, 1024)	67
Rysunek 9.13 Wynik polecenia "top -H" podczas testu asymetrycznego z różnymi priorytetami wątków	67
Rysunek 9.14 Histogramy dla przypadku asymetrycznego z różnymi priorytetami	68
Rysunek 9.15 Wykresy punktowe dla przypadku asymetrycznego z różnymi priorytetami	68

Spis tabel

Tabela 8-1 Statystyki dla $T = 25$ ms i $L = 1/512/1024/2048/4096/8192$ [byte]	54
Tabela 8-2 Statystyki dla $T = 5$ ms i $L = 1/512/1024/2048/4096/8192$ [byte]	55
Tabela 8-3 Statystyki dla $T = 1$ ms; $L = 1/512/1024/2048/4096/8192$ byte; $NI = 0$	57
Tabela 8-4 Statystyki dla $T = 1$ ms; $L = 1/512/1024/2048/4096/8192$ byte; $NI = -20$	58
Tabela 8-5 Statystyki dla $T = 1$ ms; $L = 1/512/1024/2048/4096/8192$ byte; $NI = 19$	59
Tabela 9-1 Miary dokładności po badaniu symetrycznym, gdzie $F(5\text{ms}, 1024)$	62
Tabela 9-2 Miary dokładności po badaniu symetrycznym, gdzie $F(25\text{ms}, 1024)$	63
Tabela 9-3 Miary dokładności po badaniu asymetrycznym z priorytetem domyślnym	64
Tabela 9-4 Miary dokładności po badaniu symetrycznym z różnymi priorytetami $F(5\text{ms}, 1024)$	65
Tabela 9-5 Miary dokładności po badaniu symetrycznym z różnymi priorytetami $F(25\text{ms}, 1024)$	66
Tabela 9-6 Miary dokładności po badaniu asymetrycznym z różnymi priorytetami	68

Spis listingów

Listing 5.1 Przykładowa deklaracja portu wraz z jego regułami.....	23
Listing 6.1 Dodanie biblioteki Paho do projektu	27
Listing 6.2 Konfiguracja klienta MQTT w metodzie onCreate().....	27
Listing 6.3 Niezbędne linijki konfiguracji do pliku „AndroidManifest.xml”	28
Listing 6.4 Fragment kodu serwera UDP	29
Listing 6.5 Kod klienta UDP	31
Listing 6.6 Filtr stosowany do wyświetlenia pakietów po emulacji	32
Listing 6.7 Fragment kodu serwera TCP.....	34
Listing 6.8 Fragment kodu klienta TCP	35
Listing 6.9 Szablon pliku RCR [6]	37
Listing 6.10 Zawartość klasy „Event”	39
Listing 6.11 Przykład funkcji wczytującej listę zdarzeń.....	40
Listing 6.12 Fragment kodu klasy Service.....	42
Listing 6.13 Fragment kodu, w którym system czeka na zdarzenie wchodzące	43
Listing 6.14 Fragment kodu, w którym system zapisuje zdarzenie wejściowe do kolejki	44
Listing 6.15 Fragment kodu, w którym odbywa się obsługa maszyny stanów	45
Listing 6.16 Fragment kodu, w którym odbywa się obsługa portów na węźle NAPES	45
Listing 6.17 Fragment kodu, realizujący zdarzenia lokalne	46
Listing 6.18 Ułamek kodu realizującego zapisywanie logów do pliku JSON.....	47
Listing 7.1 Konfiguracja klienta zgodnie z wymaganiami plików RCR	48
Listing 8.1 Zawartość pliku RCR, wykorzystywanego podczas wykonania testów	53
Listing 9.1 Zawartość pliku RCR, wykorzystywanego podczas wykonania testów	61

Załącznik I. Fragment kodu wykorzystywany podczas badań dokładności wysyłania pakietów

```
1 while (Config.simulating) { // do póki flaga symulacji == true
2
3 // pętla sprawdzająca każdy state flow
4 for(StateFlow stateFlow:port.getClientInfo().getStateFlowList().getStateFlows()){
5
6 // deklaracja parametrów StateFlow
7 String stateName = stateFlow.getName();
8 String stateMachineName = stateMachine.getName();
9 // sprawdź czy przetwarzany StateFlow zgadza się z aktualnym stanem FSM
10 if (map.get(stateMachineName).equals(stateName)) {
11 // zapis czasu rozpoczęcia przepływu do pliku JSON
12 String currentTime = (Long.toString(System.currentTimeMillis()));
13 // ustaw aktualny przepływ na porcie
14 Flow currentFlow = getCurrentFlow(stateFlow);
15 //ustawienie priorytetu wątków
16 android.os.Process.setThreadPriority(getPriority(currentFlow));
17 //tworzenie klienta
18 UdpClient udpClient = new UdpClient(handler, port, currentFlow);
19 // ustawienie odstępów między pakietami
20 long timeOut = currentFlow.getRealTimeDelay() * 1_000_000;
21 currentFlowBuf = currentFlow.getType();
22 // zapis czasu przed przetwarzaniem
23 long timer = System.nanoTime();
24 // pętla przetwarzająca przepływ
25 while (map.get(stateMachine.getName()).equals(stateName) &&
26 Config.simulating) {
27 udpClient.sendThroughLink(); // wysłanie pakietu
28 tempTime = System.nanoTime() - timer;
29
30 // Jeśli czas przetwarzania był większy niż
31 // odstęp między pakietami, to kontynuuj pętlę
32 if (tempTime - timeOut >= 0) {
33 timer = System.nanoTime();
34 continue;
35 } else {
36 long timeToSleepNs;
37 timer = System.nanoTime() - timer; // uwzględnianie czasu przetwarzania
38 timeToSleepNs = timeOut - timer; // odstęp czasowy - czas przetwarzania
39 try {
40 // czekanie
41 TimeUnit.NANOSECONDS.sleep(timeToSleepNs);
42 } catch (InterruptedException e) {
43 e.printStackTrace();
44 }
45 // ustaw timer przetwarzania
46 timer = System.nanoTime();
47 }
48
49 }
50 }
51 }
```