POLITECHNIKA WARSZAWSKA

DYSCYPLINA NAUKOWA INFORMATYKA TECHNICZNA
I TELEKOMUNIKACJA
DZIEDZINA NAUK INŻYNIERYJNO-TECHNICZNYCH

Rozprawa doktorska

mgr inż. Piotr Zych

Wykrywanie awarii sieciowych oparte na korelacyjnej analizie danych w zakresie usług świadczonych przez operatora telekomunikacyjnego

Promotor
dr hab. inż. Przemysław Dymarski, prof. PW

WARSZAWA 2023

Pragnę złożyć serdeczne podziękowania
dr hab. inż. Przemysławowi Dymarskiemu, prof. PW
oraz
dr hab. inż. Marcinowi Zychowi
za nieocenioną pomoc i poświęcony czas
w trakcie przygotowywania niniejszej rozprawy doktorskiej

WYKRYWANIE AWARII SIECIOWCYH OPARTE NA KORELACYJNEJ ANALIZIE DANYCH W ZAKRESIE USŁUG ŚWIADCZONYCH PRZEZ OPERATORA TELEKOMUNIKACYJNEGO

Streszczenie

Skuteczna detekcja awarii sieci jest dla operatora telekomunikacyjnego jednym z istotnych aspektów zapewnienia wysokiej jakości dostarczanych usług. Rynek telekomunikacyjny zapewnia wiele rodzajów systemów monitorowania. Analiza aktualnego stanu wiedzy oraz kooperacja z wybranym operatorem sieciowym wykazały występujące niedoskonałości dotychczas dostępnych systemów monitorowania sieci telekomunikacyjnych. Większość z nich bazuje na bezpośrednim monitorowaniu zdefiniowanych zasobów sieciowych.

W ramach rozprawy doktorskiej przedstawiono podejście do monitorowania (algorytm *AWA*) bazujące na korelacji danych otrzymywanych z sieci w czasie rzeczywistym. Kluczowym źródłem są informacje dotyczące rozliczeń użytkowników (ang. *Accounting*) dostarczane za pomocą protokołu *RADIUS* (ang. *Remote Authentication Dial In User Service*), informujące o stanie sesji *PPP* (ang. *Point to Point Protocol*) i wysyłane do systemu monitorowania. Algorytm *AWA* może opierać się na dowolnym innym protokole, który zapewnia możliwość notyfikacji w czasie rzeczywistym (lub zbliżone do czasu rzeczywistego) o zmianie stanu każdej usługi pracującej w sieci. W pracy przedstawiono podstawową wersję algorytmu *AWA*, ale także jego rozszerzenia, których celem było zwiększenie skuteczności wykrywanych awarii. Jedną z zaprezentowanych opcji rozbudowy była także możliwość realizacji działań proaktywnych przy okazji wykrywania awarii.

Ponadto, częścią pracy autora było wdrożenie algorytmu *AWA* u wybranego operatora telekomunikacyjnego oraz ocena skuteczności wdrożenia, którego wyniki zaprezentowano w niniejszej rozprawie. Wykonano badania symulacyjne implementacji algorytmu pod kątem oceny jego wydajności i skalowalności. Algorytm *AWA* został także zaprezentowany w ramach artykułu opublikowanego na łamach czasopisma *International Journal of Electronics and Communications (AEÜ)*, a europejskie biuro patentowe (*European Patent Office*) nadało patent (*EP3252995*).

W podsumowaniu rozprawy omówiono skuteczność *AWA* pod kątem sformułowanych we wstępie tez rozprawy doktorskiej. Podsumowano efekty wdrożenia u operatora telekomunikacyjnego oraz przedstawiono zyski z wdrożenia. Przeprowadzone pomiary powdrożeniowe wskazują, że wprowadzone rozwiązanie pozwala na wykrycie większej liczby

zdarzeń, umożliwiając szybszą reakcję, celem przywrócenia usług dla użytkowników. Przekłada się to, na zapewnienie większej ciągłości oraz lepszej jakości świadczonych usług.

Słowa kluczowe: sieć telekomunikacyjna; monitorowanie; korelacja danych; wykrywanie awarii; *RADIUS*; *Accounting*; *Point to Point Protocol*

NETWORK FAILURE DETECTION BASED ON CORRELATION DATA ANALYSIS

Abstract

For a telecommunications operator, effective network failure detection is one of the important aspects of ensuring high quality of provided services. The telecommunications market provides many types of monitoring systems. The analysis of the current state of knowledge and cooperation with the selected network operator revealed the imperfections of the so far available monitoring systems for telecommunications networks. Most of them are based on direct monitoring of defined network resources.

This doctoral dissertation presents an approach to monitoring (*AWA* algorithm) based on the correlation of data obtained from the network in real-time. The key source is the accounting information retrieved using *Remote Authentication Dial In User Service* (*RADIUS*) protocol, concerning the status of users' *Point to Point Protocol* (*PPP*) sessions. This information is sent to the monitoring system. The *AWA* algorithm can be based on any other protocol that provides real-time (or near-real-time) notification capabilities of a change in the state of each service running on the network. This work presents the basic version of the *AWA* algorithm, as well as its extensions, the purpose of which was to increase the effectiveness of the functionality. One of the presented extensions was the possibility of implementing proactive measures when detecting failures.

Part of the author's work was the implementation of the *AWA* algorithm at the selected network operator and the assessment of the effectiveness of the implementation, the results of which are presented in this dissertation. The implementation of the algorithm was also simulation tested in terms of assessing its performance and scalability. The *AWA* algorithm was presented in the form of an article published in the *International Journal of Electronics and Communications* (*AEÜ*), and the method itself was patented in the *European Patent Office* (*EP3252995*).

In the summary of the dissertation, the effectiveness of *AWA* was discussed in terms of the theses presented in the introduction. The effects of the implementation for a telecommunications operator were summarized and the profits from the implementation were presented. The post-implementation measurements carried out indicate that the introduced solution allows for a detection of a larger number of events, enabling a faster response to restore services for users. This translates into greater continuity and better quality of provided services.

Keywords: telecommunication network; monitoring; data correlation; failures detection; *RADIUS*; *Accounting*; *Point to Point Protocol*

Życiorys

Piotr Zych

Urodziłem się 29 czerwca 1988 roku w Krośnie. W 2007 roku ukończyłem I Liceum Ogólnokształcące im. Mikołaja Kopernika w Krośnie oraz rozpocząłem studia dzienne na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej na Makrokierunku. Wybraną przeze mnie specjalnością była: Teleinformatyka i Zarządzanie w Telekomunikacji oraz specjalność dodatkowa: Systemy i Sieci Telekomunikacyjne. Od lipca 2010 roku pracuję w Orange Polska S. A. na stanowisku eksperckim, gdzie zajmuję się tworzeniem i wdrażaniem systemów monitorowania oraz diagnostyki sieci stacjonarnej. W czerwcu 2011 roku z wyróżnieniem ukończyłem studia I stopnia, a studia II stopnia zostały przeze mnie zakończone we wrześniu 2012 roku. Studia III stopnia rozpocząłem w październiku 2014 roku. W latach 2017 – 2018 byłem zatrudniony na Wydziale Elektroniki i Technik Informacyjnych na stanowisku Asystenta Naukowo-Dydaktycznego.

SPIS TREŚCI

WYKAZ UŻYWANYCH SKRÓTÓW

AAA	Authentication, Authorization and Accounting
ADSL	Asymmetric Digital Subscriber Line
ADSL/2/2+	rodzina standardów ADSL
ATM	Asynchronous Transfer Mode
AWA	Algorytm Wykrywania Awarii
BFD	Bidirectional Forwarding Detection
BNG	Broadband Network Gateway
BRAS	Broadband Remote Access Server
CDN	Content Delivery Network
CPE	Customer Premises Equipment
CPU	Central Processing Unit
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name Server
DSL	Digital Subscriber Line
DSLAM	Digital Subscriber Line Access Multiplexer
EPO	European Patent Office
FDEP	Failure Detection/Exploration Protocol
FTTH	Fibre To The Home
G.FAST	Fast Access to Subscriber Terminals
GNU	System operacyjny “GNU's Not Unix!”
GPON	Gigabit Passive Optical Network
GUI	Graphical User Interface
GZS	Grupowe Zerwanie Sesji
HDD	Hard Disk Drive
HFC	Hybrid Fibre-Coaxial
HMP	Host Monitoring Protocol
IP	Internet Protocol
IPDSLAM	DSLAM działający w technologii pakietowej
ISDN	Integrated Services Digital Network
JDK	Java Development Kit
L1	Warstwa 1 modelu ISO/OSI (ang. *Layer 1*) – warstwa fizyczna
L2	Warstwa 2 modelu ISO/OSI (ang. *Layer 2*) – warstwa łącza danych

LAN	Local Area Network
LOP	Loss of Power
MySQL	Rodzaj serwera baz danych SQL
MSAN	Multi-Service Access Node
NE	Network Element
NMS	Network Management System
NR	Network Resilience
OID	Object Identifier
OLT	Optical Line Termination
OSI	ISO Open Systems Interconnection Reference Model
PPP	Point-to-Point Protocol
PTM	Packet Transfer Mode
RADIUS	Remote Authentication Dial In User Service
RAM	Random-Access Memory
RNR	Relative Network Resilience
RODO	Rozporządzenie o ochronie danych osobowych
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UNEMS	Unified Network Element Monitoring System
VDSL2	Very High Speed Digital Subscriber Line 2
VLAN	Virtual LAN
VOIP	Voice over IP
VP	Virtual Path (w sieci ATM)
VPN	Virtual Private Network
XTU-R	XDSL Transmission Unit-Remote

1. WPROWADZENIE

Dynamiczny rozwój usług multimedialnych (Internet, telewizja, telefonia mobilna) oraz coraz większy stopień ich skomplikowania powoduje trudności w wykrywaniu każdego rodzaju awarii przez operatora telekomunikacyjnego. Klasyczne systemy monitorujące awarie sieci nie zawsze są w stanie wykryć wszystkie rodzaje występujących awarii. Dzieje się tak ze względu na oparcie monitorowania na wybranych, predefiniowanych zdarzeniach lub elementach sieci. Każda awaria jest jednak ostatecznie wykrywana poprzez abonentów zgłaszających problemy z dostępem do usług. W takim przypadku cierpi wizerunek operatora, który zawczasu nie wykrył awarii. Wydłuża się także czas usuwania awarii, a tym samym zwiększa się czas niedostępności usług.

Trudności z wykrywaniem awarii wynikają z coraz większego stopnia skomplikowania i zwiększania różnorodności usług sieciowych. Aktualnie do świadczenia usług często stosuje się różnego rodzaju enkapsulacje protokołów, sieci wirtualne (*VPN* – ang. *Virtual Private Network*) i tunele. Każdy z elementów sieci jest monitorowany oddzielnie. Istotne jest jednak monitorowanie rozwiązania jako całości, tak aby móc wykrywać awarie, które nie są widoczne z poziomu monitorowania pojedynczych elementów sieci. Przykładowo, systemy monitorujące status tuneli *VPN* realizują monitorowanie całościowe i wysyłają alarmy dopiero w momencie problemów widocznych globalnie (np. spadek aktywnych sesji o 10% na poziomie całej sieci). Każdy tunel jest realizowany poprzez wykreowanie odpowiedniego połączenia na warstwie 2. (łącza danych) modelu *ISO/OSI* [1]. Urządzenia realizujące niezbędne funkcje są monitorowane, a ich status jest weryfikowany. Monitorowany jest także poziom warstwy 2. ale, jako że urządzenia te nie służą do realizacji samego tunelu jako takiego, z ich poziomu jego status nie jest weryfikowany. Występują jednak awarie związane z brakiem możliwości zestawienia tunelu, które przez bezpośrednie monitorowanie urządzeń nie są wykrywane. Nie da się przewidzieć z góry wszystkich możliwych typów uszkodzenia urządzenia, czy błędów w konfiguracji, które są widoczne dopiero przy porównaniu konfiguracji różnych urządzeń. Z poziomu samego monitoringu może wydawać się, że urządzenie pracuje prawidłowo, ale różnego rodzaju usługi (np. takie jak sesja *PPP*, *DS-LITE*, czy *SSH*) z nieokreślonego powodu nie mogą być zestawione. Globalny monitoring nie wskaże problemu, ponieważ spadek liczby aktywnych sesji na poziomie ogólnym jest zbyt niski – dany problem występuje w ograniczonym zakresie sieci. Do wykrywania tego typu awarii występuje potrzeba wprowadzenia systemu monitorowania, który będzie znajdował się pomiędzy poziomem monitorowania pojedynczych urządzeń sieci, a poziomem monitoringu globalnego.

1.1. CELE PRACY

Zarysowane we wstępie problemy z monitorowaniem awarii sieciowych skłoniły autora rozprawy do sformułowania następujących celów:

- opracowanie algorytmu wykrywania awarii sieciowych,
- wprowadzenie usprawnień w analizie danych pod kątem wydajnościowym,
- weryfikacja algorytmu w rzeczywistym środowisku,
- statystyczna analiza zachowania się opracowanego algorytmu w oparciu o symulacje.

Ponadto zostały sformułowane następujące tezy:

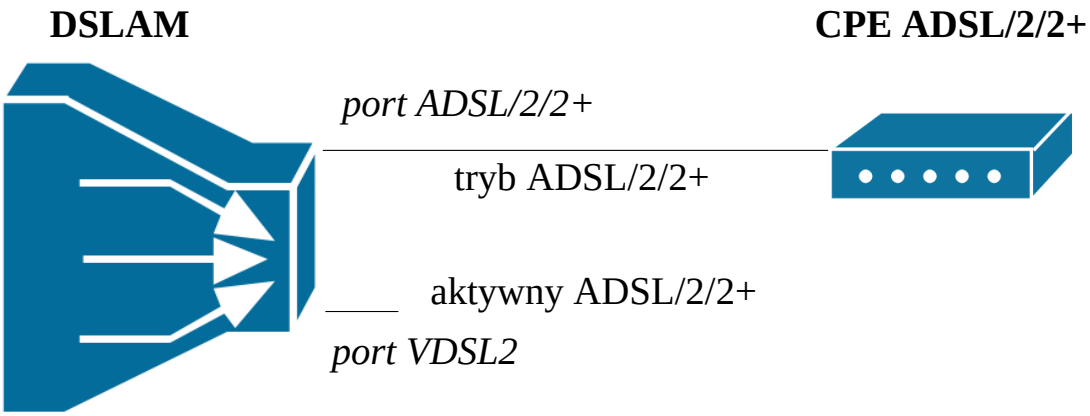
- korelacyjna analiza danych w zakresie usług świadczonych przez operatora telekomunikacyjnego zapewnia większą skuteczność wykrywania awarii w stosunku do innych typów monitorowania sieci (bazujących wyłącznie na informacjach pozyskanych bezpośrednio z urządzeń sieciowych),
- bazowanie na informacjach z sieci otrzymywanych jako notyfikacje wysyłane do systemu monitorującego zapewnia mniejsze opóźnienie w wykrywaniu awarii w stosunku do systemów cyklicznie monitorujących sieć oraz powoduje mniejsze obciążenie sieci,
- zaproponowany algorytm monitorowania sieci jest skalowalny – wydajność opracowanego algorytmu korelacyjnej analizy danych jest wystarczająca do jego implementacji w rzeczywistej sieci (w oparciu o dane udostępnione przez kooperującego z *PW* polskiego operatora szerokopasmowych sieci stacjonarnych),
- opracowana procedura monitorowania w przypadku usług świadczonych w oparciu o protokół *PPP* zapewnia możliwość wykrywania awarii zarówno urządzeń aktywnych (np. przełączniki), jak również komponentów pasywnych (np. przewody miedziane lub światłowodowe).

Przeprowadzone analizy dotyczą sieci dostępowej i agregacyjnej. Zawarte w pracy wnioski i informacje statystyczne pochodzą z wdrożeń oraz analiz przeprowadzanych przez autora w trakcie prac wykonywanych w *Orange Polska S.A.*

2. MONITOROWANIE ZASOBÓW SIECIOWYCH

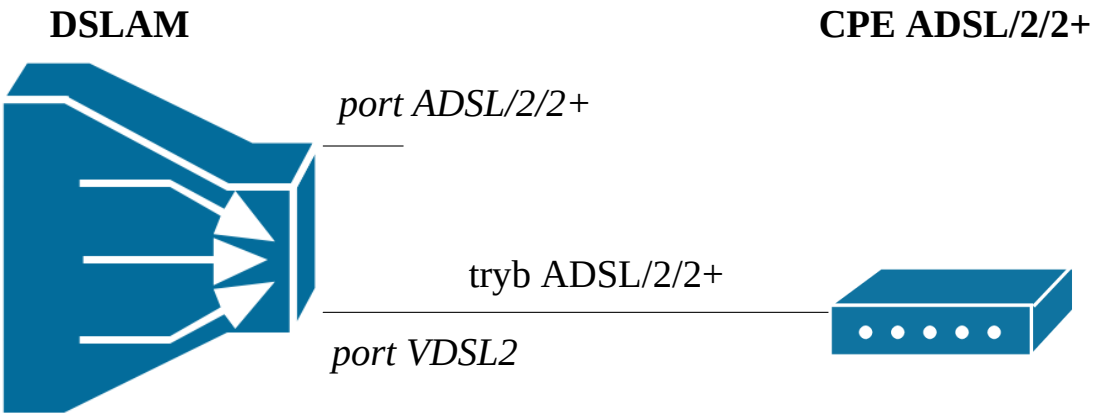
Częścią procesu wykrywania awarii jest monitorowanie stanu zasobów sieciowych. W tym rozdziale przedstawiono i porównano dwa najczęściej stosowane sposoby monitorowania na przykładzie funkcjonalności monitorowania rodzaju *CPE* (ang. *Customer Premises Equipment*) *DSL* (ang. *Digital Subscriber Line*) od strony urządzenia *DSLAM* (ang. *Digital Subscriber Line Access Multiplexer*). Celem analizy jest wypracowanie optymalnej metody monitorowania do wykrywania awarii.

Rozwój różnych usług biznesowych, portali informacyjnych, platform z rozrywką, itp., wymusza szybki rozwój transmisji szerokopasmowej w sieci dostępowej. Pierwszą specyfikacją cyfrowej transmisji danych w tego typu sieci była technika *ISDN* (ang. *Integrated Services Digital Network*) z 1993 roku [2]. Aktualnie w sieciach miedzianych wykorzystywane są techniki z rodziny *ADSL/2/2+* (ang. *Asymmetric Digital Subscriber Line/2/2+*) [3], [4], [5], [6], [7], [8] oraz *VDSL2* (ang. *Very High Speed Digital Subscriber Line 2*) [9], które są używane w sieci równolegle. Jeśli dołoży się do tego najnowsze techniki jak *G.FAST* (ang. *Fast Access to Subscriber Terminals*) [10], [11] oraz *G.mgfast* [12] okazuje się, że wykrywanie rodzaju *CPE* jest istotną czynnością w procesie zapewnienia ciągłości świadczenia usług. Weryfikacja rodzaju *CPE* jest ważnym aspektem w trakcie migracji między technikami *ATM* (ang. *Asynchronous Transfer Mode*) – wykorzystywanym przez rodzinę *ADSL/2/2+* oraz *PTM* (ang. *Packet Transfer Mode*) – wykorzystywanym przez technikę *VDSL2* [13]. Automatyzacja przełączania między tymi technikami oparta na wykrywaniu typu *CPE* jest wymaganym elementem do zapewnienia ciągłości usług w trakcie migracji z techniki *ADSL/2/2+* do *VDSL2*.



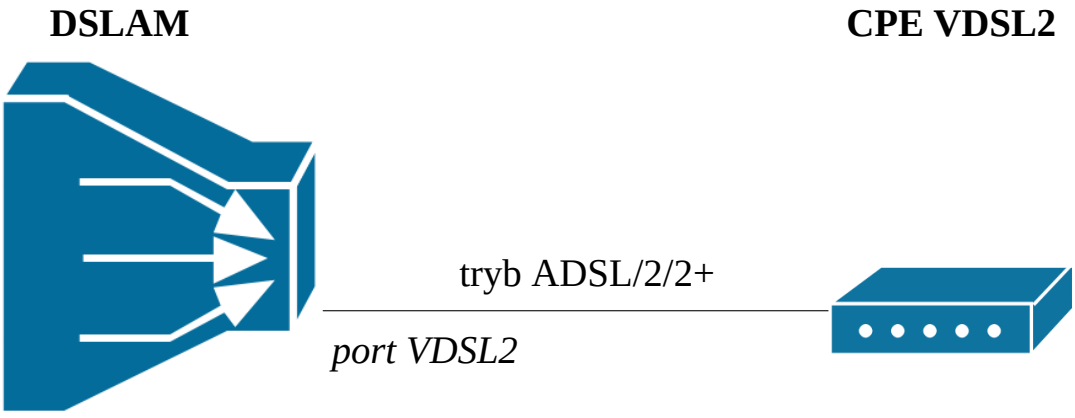
Rys. 2.1: Początkowy stan sieci

Rys. 2.1 pokazuje moment, gdy usługi klienta są świadczone za pomocą portu *ADSL/2/2+*. Do zapewnienia ciągłości świadczenia usług wykorzystuje się fakt, że mimo iż *VDSL2* nie może być aktywowany równocześnie z *ADSL/2/2+*, port tego typu może zostać skonfigurowany do pracy wyłącznie w trybie *ADSL/2/2+*. Taka konfiguracja została przedstawiona na Rys. 2.1.



Rys. 2.2: Stan po zmianie połączenia *CPE ADSL/2/2+* do portu *VDSL2*

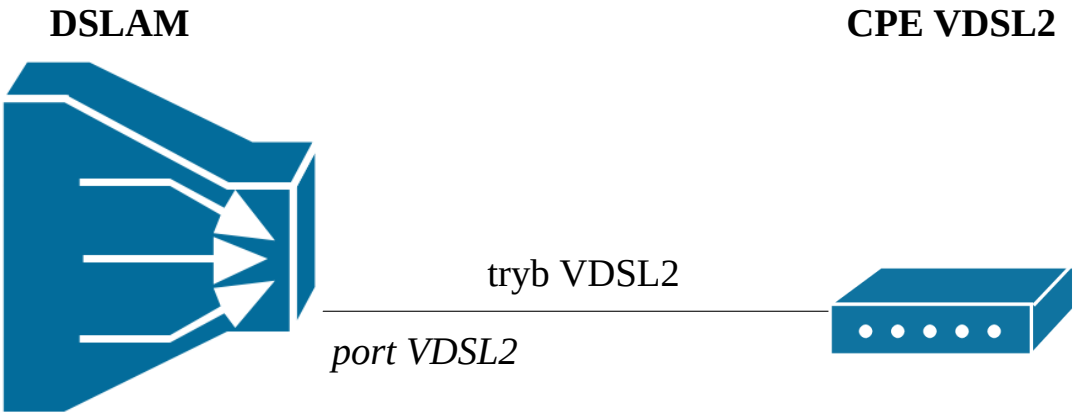
Odpowiednia konfiguracja portu *VDSL2* po zmianie połączenia fizycznego linii abonenckiej do portu na *DSLAM* na centrali *CPE ADSL/2/2+* pozwala poprawnie działać na porcie *VDSL2* (Rys. 2.2). Od tego momentu kluczowe jest jak najszybsze wykrycie podłączenia *CPE VDSL2*, po czym powinna nastąpić automatyczna rekonfiguracja portu do trybu *VDSL2*.



Rys. 2.3: Stan po podłączeniu *CPE VDSL2* (praca w trybie *ADSL/2/2+*)

Na Rys. 2.3 przedstawiono sytuację po podłączeniu *CPE VDSL2*. Uwzględniając, że port *VDSL2* jest aktywowany w trybie *ADSL/2/2+* linia pracuje wciąż w trybie *ADSL/2/2+*. Należy w sposób automatyczny wykryć pojawienie się stanu pokazanego na Rys. 2.3 w oparciu

o monitorowanie typu *CPE* (*VDSL2* vs *ADSL/2/2+*). Proces ten musi być niezauważalny dla klienta. Po wykryciu *CPE VDSL2* powinna nastąpić zmiana konfiguracji portu *VDSL2* do pracy w trybie *VDSL2*, do świadczenia usług docelowych (Rys. 2.4).



Rys. 2.4: Docelowy stan konfiguracji portu (po zmianie konfiguracji na *VDSL2*)

2.1. MONITOROWANIE CYKLICZNE

Pierwszą opcją monitorowania jest monitorowanie cykliczne, które umożliwia kontrolę dowolnie zdefiniowanej liczby atrybutów wybranego zasobu (w opisywanej sytuacji migracji jest to typ *CPE*). Jest ono aktywowane w zdefiniowanych interwałach czasowych, w trakcie których skanuje się predefiniowaną listę elementów sieciowych (ang. *NE – Network Element*) pod kątem określonej liczby atrybutów (np. stan operacyjny, stosunek sygnał-szum, liczba błędów *CRC*).

W trakcie monitorowania sieci pod kątem awarii, krytycznymi zjawiskami są: opóźnienie w pobieraniu danych oraz wydajność monitorowania. Typowy czas odpowiedzi urządzenia jest równy 200 ms, co odpowiada częstotliwości jednowątkowego odpytywania równej 5 Hz. W przypadku konieczności monitorowania typowej sieci dostępowej zawierającej 30 tys. urządzeń *DSLAM* (rząd wielkości urządzeń dla operatora obsługującego kraj wielkości Polski) z częstotliwością 5 Hz jednokrotny skan sieci zająłby 1h 40 minut. Z punktu widzenia monitorowania dostępności usług taki czas opóźnienia w wykryciu problemu jest nieakceptowalny. Powoduje to, że jedynym rozwiązaniem jest zaimplementowanie takiego monitorowania w przetwarzaniu równoległym, co generuje znacznie więcej ruchu sieciowego związanego z monitorowaniem.

W przypadku mniejszej liczby zasobów do monitorowania (do 1 tys.) czas jednokrotnego skanowania jest na poziomie 5 minut. Z kolei, gdyby taki czas był akceptowalny z punktu

widzenia zastosowania biznesowego, możliwa byłaby implementacja takiego rozwiązania bez wdrożenia przetwarzania równoległego. Zaletą monitorowania cyklicznego jest to, że nie bazuje ono odpowiedzi otrzymanej na wysłane żądanie, co zapewnia niezawodność rozwiązania. Ponadto, w przypadku monitorowania cyklicznego można definiować dowolne algorytmy pod kątem logiki wykonywanych zapytań i wnioskowania w oparciu o otrzymane odpowiedzi.

2.2. ODBIÓR ASYNCHRONICZNYCH NOTYFIKACJI

Alternatywną metodą monitorowania jest odbiór asynchronicznych notyfikacji. Wymaga on jednak konfiguracji wszystkich elementów sieci tak, aby powiadamiały system o zdefiniowanych typach zdarzeń (np. zmiana statusu operacyjnego, potwierdzenie podtrzymania statusu operacyjnego, zmiana przepustowości, wzrost liczby błędów *CRC* powyżej zadanego progu).

Ograniczeniem tego sposobu monitorowania jest lista możliwych typów zdarzeń, jakie będą wysyłane jako notyfikacje. Zdarzenia takie są definiowane przez dostawców urządzeń, a stopień ich zaawansowania ogranicza się zazwyczaj do zmiany stanu pojedynczego parametru – bez dodatkowych analiz. Z tego powodu, jeśli w ramach monitorowania wymagana jest bardziej zaawansowana funkcjonalność (np. zmiana przepływności poniżej danego progu, ale tylko w zadanym oknie czasowym i tylko dla wybranego modelu *CPE*), jedynie częściowo można ją zrealizować w oparciu o proste notyfikacje. Po odebraniu notyfikacji należy wykonać dodatkowe pobranie parametrów z urządzeń, celem realizacji logiki pełnej funkcjonalności.

W zakresie potrzeby monitorowania typu *CPE*, idealnym rozwiązaniem byłaby możliwość konfiguracji, na wybranych portach, zdarzeń informujących o zmianie typu *CPE* na kompatybilne z wybranym protokołem. W momencie pisania dysertacji żaden z producentów urządzeń wykorzystywanych w sieci analizowanego operatora nie zapewnia takiej możliwości. Istnieje możliwość wysyłania notyfikacji o zmianie statusu operacyjnego portu, jednak bez specyfikacji listy konkretnych portów. Wysyłka notyfikacji dla całej sieci w przypadku zapotrzebowania monitorowania jedynie niewielkiego wycinka całej sieci, jest niewydajnym procesem (powoduje wysoki nadmiar przesyłanych informacji). Z kolei dla funkcjonalności wykrywania awarii, gdzie zazwyczaj monitorowanie dotyczy całej sieci, a oczekiwane opóźnienie w przekazywaniu danych powinno być minimalne – takie rozwiązanie wydaje się być najlepsze.

Wadą asynchronicznych notyfikacji jest to, że bazuje się w nich na funkcjonalności samych urządzeń sieciowych, które mogą zawieść. Często też do notyfikacji na warstwie transportowej

modelu *ISO/OSI* [1] wykorzystuje się protokół *UDP* [14], który nie zapewnia gwarancji transmisji. To oznacza, że asynchroniczne notyfikacje nie gwarantują pewności rozwiązania. Ich niewątpliwą zaletą jest szybkość działania – opóźnienie jest rzędu maksymalnie dziesiątek milisekund, które w praktyce można uznać za pomijalne oraz minimalizacja ruchu sieciowego w przypadku, gdy uda się skonfigurować rodzaj notyfikacji odpowiadający konkretnej potrzebie.

2.3. PORÓWNANIE METOD MONITOROWANIA

Oba rodzaje monitorowania opisane w rozdziałach 2.1 i 2.2 mogą mieć swoje zastosowanie w zależności od wymagań przedstawionych przez operatora. Przykładowo, w przypadku potrzeby monitorowania całej sieci pod kątem prostego parametru jak status operacyjny portu, optymalnym rozwiązaniem jest odbiór notyfikacji. Z kolei dla sytuacji monitorowania wybranego fragmentu sieci pod kątem wykrywania zdarzeń zdefiniowanych w bardziej skomplikowany sposób (np. zmiana przepływności łącza w określonym okresie czasu i przy określonym poziomie błędów) lepszym rozwiązaniem jest monitorowanie cykliczne. Przy analizie rozwiązań należy też pamiętać o ich niezawodności.

Ruch generowany przez cykliczne monitorowanie można obliczyć z zależności:

$$F_C = A_C \cdot N \cdot f_C \tag{1}$$

gdzie:

F_C – ruch w sieci dla monitorowania cyklicznego,

A_C – rozmiar przesyłanych danych dla jednego elementu sieciowego,

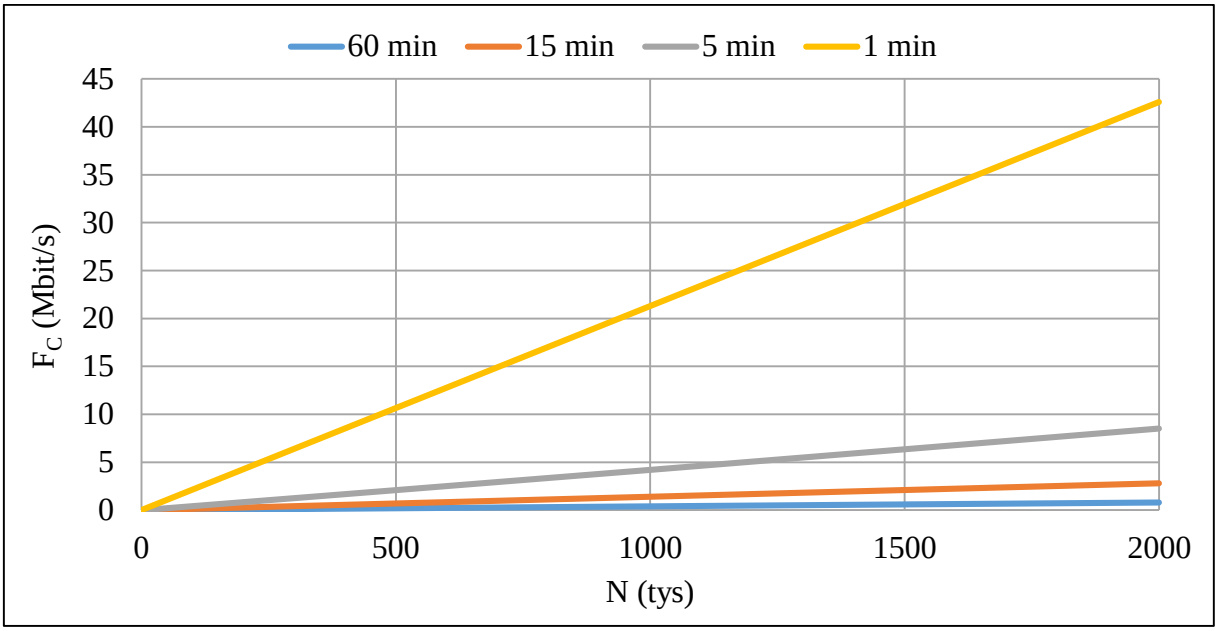
N – liczba monitorowanych elementów sieciowych,

f_C – częstotliwość monitorowania.

$$f_C = \frac{1}{T_C} \tag{2}$$

gdzie:

T_C – interwał monitorowania.



Rys. 2.5: Ruch w sieci zarządzania (F_C) dla cyklicznego monitorowania jednego parametru *SNMP* przy różnych częstotliwościach monitorowania (f_C podane jako interwał monitorowania T_C)

W oparciu o zależność (1) na Rys. 2.5 przedstawiono wykresy przykładowych przebiegów ruchu sieciowego generowanego w przypadku monitorowania cyklicznego. Założono przy tym średnią wielkość danych dla jednej operacji A_C =160B. Wartość tę oszacowano w oparciu o pomiar wielkości ruchu sieciowego, w przypadku pobierania jednego parametru (statusu operacyjnego) za pomocą protokołu *SNMP* w wersji 1. W przypadku monitorowania 2 mln elementów sieci z 1-minutowym interwałem średni ruch w sieci zarządzania wynosi 42,7 Mb/s. Sieci zarządzania mają ograniczoną przepustowość. Oznacza to, że taki poziom ruchu może nie być możliwy do zaakceptowania. Z kolei dla monitorowania 1 tys. portów w trakcie migracji ruch wynosi zaledwie 0,02 Mb/s. Wada monitorowania cyklicznego, w postaci generowania dużego poziomu ruchu, może zostać wyeliminowana za pomocą ograniczenia monitorowanego obszaru. Pozostałe cechy tego typu monitorowania to zalety (pewność transmisji, łatwa implementacja skomplikowanych algorytmów, itd.) co oznacza, że dzięki ograniczeniu obszaru monitorowania staje się ono rozwiązaniem rekomendowanym.

Z kolei dla funkcjonalności odbioru notyfikacji wielkość generowanego ruchu sieciowego można obliczyć ze wzoru:

$$F_N = A_N \cdot N \cdot f_N \tag{3}$$

gdzie:

F_N – wielkość ruchu w sieci w przypadku odbierania notyfikacji,

A_N – rozmiar przesyłanych danych w ramach jednej notyfikacji,

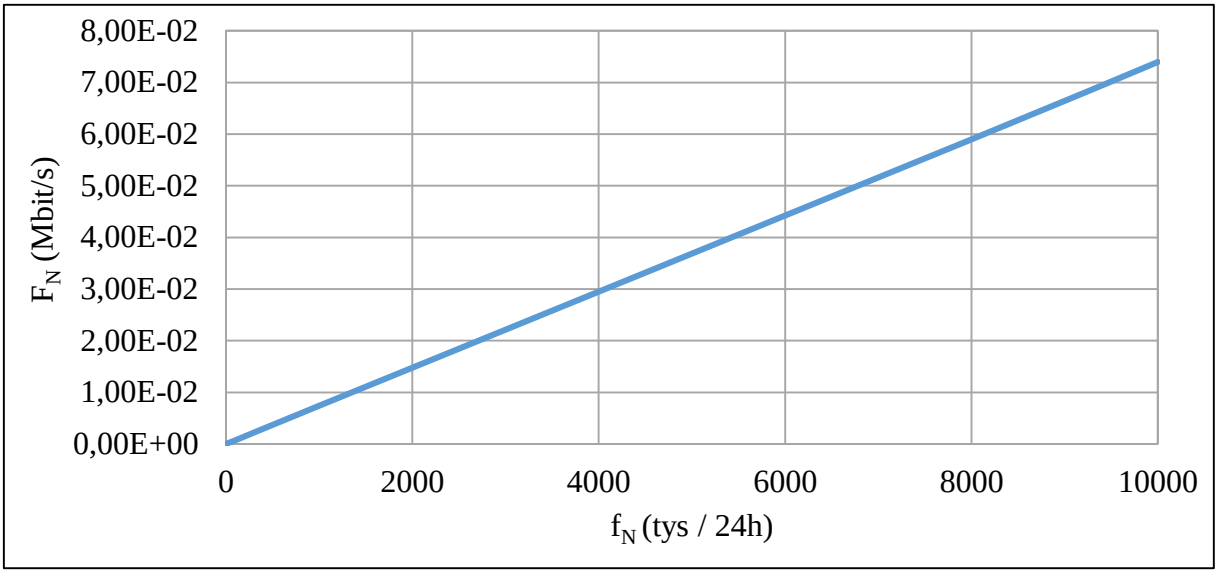
N – liczba monitorowanych elementów sieciowych,

f_N – średnia częstotliwość występowania zdarzeń dla jednego elementu sieciowego.

$$f_N = \frac{1}{T_N} \tag{4}$$

gdzie:

T_N – interwał występowania zdarzeń.



Rys. 2.6: Ruch w sieci zarządzania dla odbioru notyfikacji jednego parametru *SNMP*

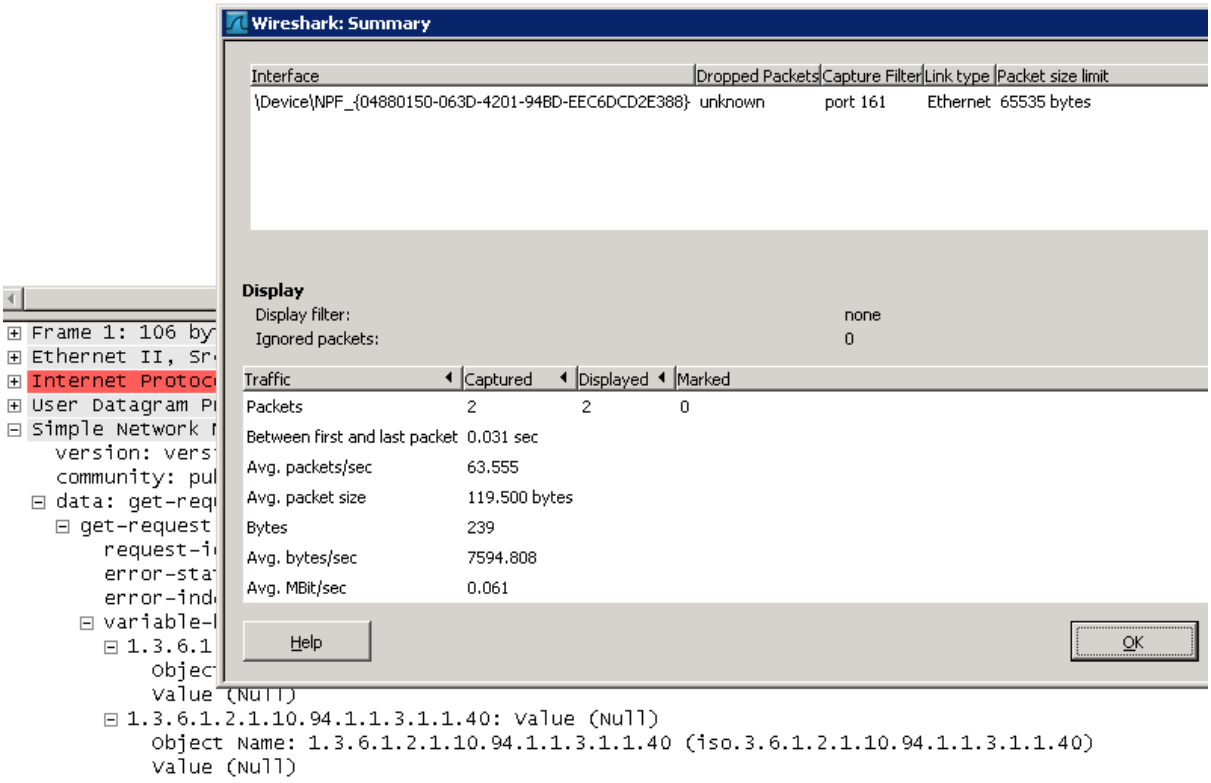
Rys. 2.6 prezentuje wielkość ruchu generowanego przy odbieraniu notyfikacji, w zależności od częstotliwości występowania zdarzeń. Ze względu na to, że notyfikacje są generowane tylko w przypadku wystąpienia zdarzenia w sieci, a monitorowanie cykliczne jest aktywowane cały czas dla całej sieci, niezależnie od tego, czy w sieci wystąpiły jakiekolwiek zdarzenia, obserwowany ruch dla notyfikacji jest znacznie mniejszy. W przedstawionych danych przyjęto założenie, że jeden element sieciowy zmienia stan 4 razy na dobę (f_N = 4 Hz), co dla 2 mln zasobów sieciowych daje dobowo 8 milionów zdarzeń. Cykliczne monitorowanie sieci takiej wielkości w interwale 5 minut generuje 576 mln operacji na dobę, co przekłada się na odpowiednio większy ruch sieciowy. Ponadto monitorowanie cykliczne generuje dodatkową porcję ruchu sieciowego w stosunku do odbioru notyfikacji. Wynika to z faktu, iż w skład jednej operacji wchodzą dwa pakiety: żądanie i odpowiedź. Z kolei w przypadku odbioru notyfikacji ruch jest jednostronny. Odbiór notyfikacji dla 10 mln zdarzeń na dobę generuje jedynie 0,074 Mb/s średniego ruchu transmisyjnego.

Ze względu na niski wolumen generowanego ruchu (F_N) notyfikacje są preferowane, jeśli można je właściwie skonfigurować, a liczba elementów do monitorowania na tyle duża, że eliminuje możliwość zastosowania monitorowania cyklicznego (np. ze względu na

przekroczenie maksymalnego ruchu sieciowego w sieci zarządzania). Z kolei w odwrotnej sytuacji, mimo tego, że cykliczne monitorowanie generuje nadmiar ruchu, może być ono rozwiązaniem preferowanym, jeśli jego zastosowanie może zostać ograniczone wyłącznie do wybranej puli zasobów sieci zapewniając niezawodność transmisji oraz większe możliwości funkcjonalne.

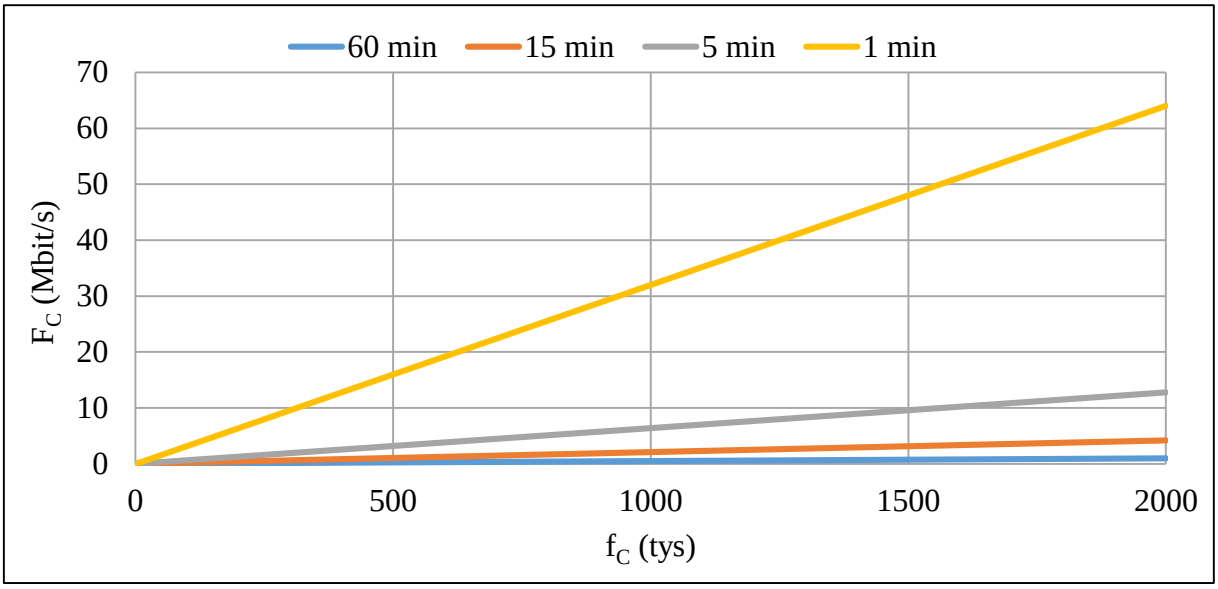
2.4. WERYFIKACJA TYPU CPE

W pracy [15] przedstawione zostały operacje wykonywane niezależnie od wybranego sposobu monitorowania. Są one realizowane po wykryciu zmiany stanu statusu operacyjnego portu *DSL* w trakcie monitorowania typu *CPE* celem podjęcia decyzji, czy doszło do zmiany typu *CPE*, czy nie. Wcześniejszy monitoring (możliwy do realizacji w oparciu o dwa opisane rodzaje monitorowania) wykrywał wyłącznie zdarzenie zmiany statusu operacyjnego portu *DSL*. Zatem zgodnie z procedurą zaprezentowaną w [15], w pierwszej kolejności należy pobrać szczegółowe informacje o *CPE* oraz potwierdzić zmianę *CPE* z wspierającego technologię *ADSL/2/2+* na wspierające technologię *VDSL2*. Poziom ruchu generowanego przez tę operację należy zsumować z poziomem ruchu generowanego przez cykliczne skanowanie statusu operacyjnego portu.



Rys. 2.7: Odczyt numeru seryjnego i stanu operacyjnego portu (*SNMP*) [15]

Na Rys. 2.7 zilustrowano pakiet *SNMP* służący do odczytu numeru seryjnego *CPE* wraz z informacją o jego wielkości.



Rys. 2.8: Ruch w sieci w przypadku zastosowania cyklicznego monitorowania z odczytem numeru seryjnego za pomocą protokołu *SNMP* (Mb/s) dla różnych interwałów monitorowania

W oparciu o równanie (1) zrealizowano wykresy przedstawione na Rys. 2.8, przedstawiające ruch wygenerowany w przypadku zastosowania cyklicznego monitorowania, po dodaniu numeru seryjnego do tego samego pakietu *SNMP* (pakiet ten służył do tej pory monitorowaniu jedynie statusu operacyjnego portu). W symulacjach przyjęto A_C=239B zgodnie z pokazanym na Rys. 2.7 odczytem. Zdecydowano się na dodanie drugiego *OID* (ang. *Object Identifier*) [16] w ramach tego samego pakietu sieciowego, celem optymalizacji wolumenu przesyłanego ruchu (dodanie danych w warstwie aplikacji zmniejsza sumaryczny ruch sieciowy, w porównaniu z sytuacją, w której dany parametr jest odczytywany w osobnym pakiecie – ze względu na narzut nagłówków). Dla 1 tys. portów na minutę w przypadku monitorowania typu *CPE* ruch wzrasta z 0,02 do 0,032 Mb/s.

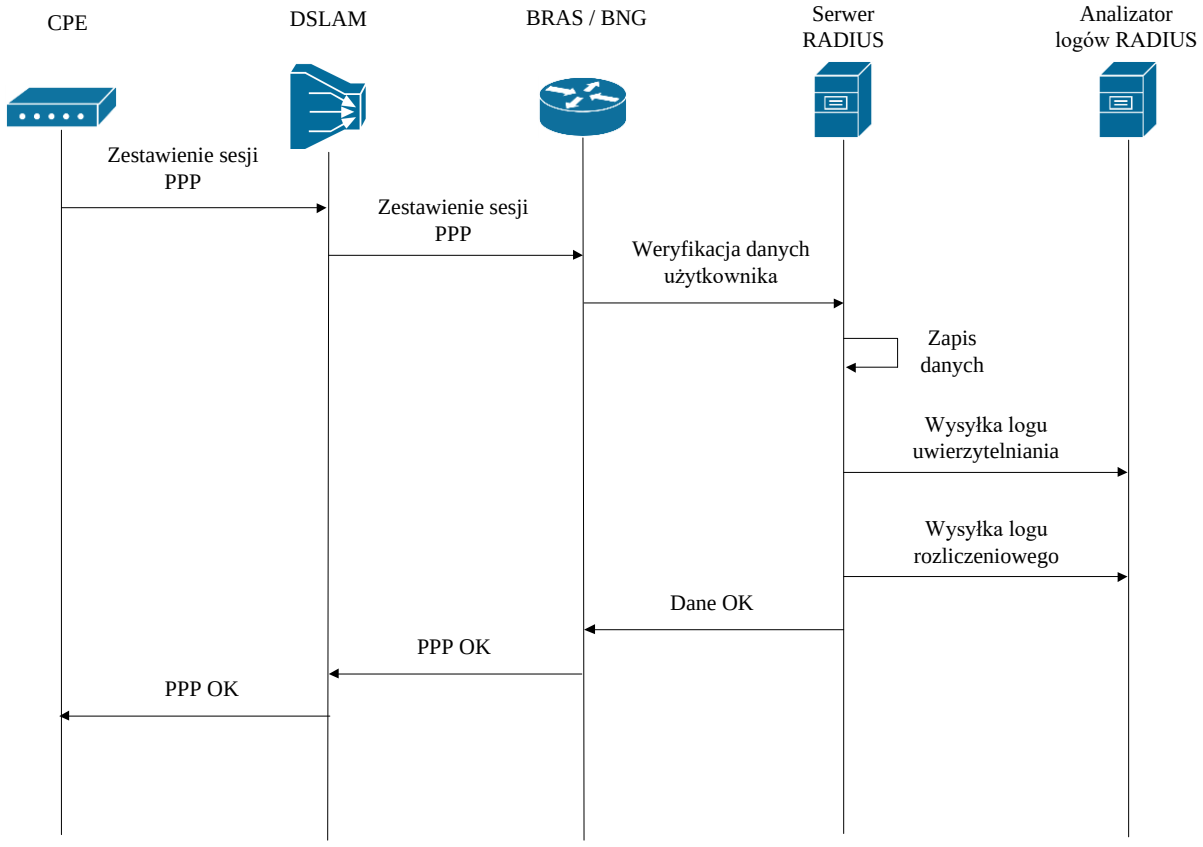
Analiza dwóch zaprezentowanych możliwości monitorowania zasobów sieciowych została wykonana pod kątem realizacji procesu migracji usług, szczegółowo opisanych w [15]. Wykorzystano przykład monitorowania typu *CPE*, ponieważ ilustruje on dobrze ogólne zagadnienie wykrywania zmian stanu elementów sieci. Wnioski wynikające z przedstawionej analizy można uogólnić. I tak, dla pewnych zastosowań odbiór notyfikacji będzie optymalnym rozwiązaniem. Nie generuje on znaczącego opóźnienia transmisji i wymaga mniej zasobów sieciowych. Założenie takie jest prawdziwe tylko w przypadku zastosowania do całej sieci oraz gdy dostawca urządzeń umożliwia notyfikację właściwych typów zdarzeń. W przeciwnym

przypadku, w warunkach ograniczenia monitorowanych zasobów lub konieczności wykonywania skomplikowanych algorytmów, cykliczne monitorowanie zapewnia lepsze wyniki z punktu widzenia generowanego ruchu sieciowego.

2.5. SIEĆ DOSTĘPOWA Z PROTOKOŁAMI PPP I RADIUS

W tym rozdziale przedstawiono opis protokołów wykorzystywanych w sieci dostępowej i agregacyjnej, które są podstawą analiz przeprowadzonych w ramach niniejszej dysertacji oraz na których bazuje zaprojektowany sposób monitorowania sieci.

W sieci dostępowo-agregacyjnej często wykorzystuje się protokół *PPP* (ang. *Point to Point Protocol*) [17]. Umożliwia on nawiązanie połączenia typu punkt-punkt pomiędzy dwoma dowolnymi elementami sieci, którymi w omawianym przypadku są *CPE* i *BRAS / BNG*.



Rys. 2.9: Nawiązanie sesji *PPP* pomiędzy *CPE*, a *BRAS / BNG* oraz wygenerowanie logu uwierzytelniania i rozliczeniowego

Przykładowy proces zestawienia sesji *PPP* pomiędzy tymi dwoma urządzeniami został przedstawiony na Rys. 2.9. Pod kątem detekcji awarii najważniejszą cechą protokołu *PPP* jest to, że na początku zestawia on sesję między dwoma końcowymi elementami, a następnie utrzymuje ją przez cały czas trwania połączenia. Stan sesji jest weryfikowany również

w przypadku, w którym urządzenia nie wymieniają między sobą danych. Protokół *PPP* wymaga cyklicznego przesyłania ramek typu *keep-alive* między urządzeniami końcowymi, celem potwierdzenia podtrzymania połączenia. Jeśli przesyłanie ramek tego typu ustaje, urządzenia końcowe uznają sesję za rozłączoną i zapisują tę informację w swoich rejestrach [17]. Informacja o nawiązaniu i rozwiązaniu połączenia *PPP* może być przesyłana w inne miejsca.

Ważnym elementem systemu są serwery *RADIUS* [18] (por. Rys. 2.9), które w swoim podstawowym przeznaczeniu realizują funkcje uwierzytelniania oraz rozliczania sesji *PPP* każdego użytkownika sieci. W większości przypadków do uwierzytelniania użytkowników stosuje się takie parametry jak nazwa użytkownika i hasło. Nazwa użytkownika może być następnie użyta w celu rozróżniania sesji różnych użytkowników między sobą w ramach analizy logów *RADIUS*. Każde nawiązanie i rozłączenie sesji *PPP* powinno być zapisywane w dziennikach zdarzeń. Są to dzienniki rozliczeniowe (ang. *accounting*) systemu *RADIUS*. Jeśli połączy się funkcjonalności utrzymania sesji *PPP* przy wykorzystaniu ramek *keep-alive* oraz notowania nawiązania każdego połączenia i zakończenia sesji *PPP* można wywnioskować, że analiza dzienników zdarzeń systemów *RADIUS* umożliwia monitorowanie statusu sesji *PPP* każdego użytkownika w sieci.

Sesje *PPP* mogą być też monitorowane bezpośrednio z poziomu urządzeń końcowych *BRAS* / *BNG*. Z wykorzystania systemów *RADIUS* do tego celu płynie jednak szereg zalet. Pierwszą z nich jest znacznie mniejsza liczba serwerów *RADIUS* w stosunku do routerów *BRAS* / *BNG*, co ułatwia budowę systemu monitorowania. Druga zaleta to fakt możliwości łatwego przekierowania ruchu typu *RADIUS Accounting* z serwerów *RADIUS* do serwerów monitorujących, dzięki czemu nie obciąża się łącz zarządzania urządzeń *BRAS* / *BNG*.

2.6. PRZYPISYWANIE ZDARZEŃ UŻYTKOWNIKOWI

W zwykle wykonywanej analizie dane *RADIUS* przypisywane są do poszczególnych użytkowników, celem zdiagnozowania stanu wybranej, pojedynczej usługi. Można wyróżnić dwie metody kojarzenia danych z użytkownikiem. Pierwsza z nich opiera się na przypisaniu konkretnego użytkownika w oparciu o wiedzę na temat jego loginu (pole *user-name*), która zapisywana jest w logach dotyczących uwierzytelnień oraz rozliczeń protokołu *RADIUS*. Jednak w sytuacji, w której występuje problem z uwierzytelnieniem użytkownika ze względu na nieprawidłową nazwę użytkownika – mechanizm ten nie pozwoli wykonać tej czynności.

Druga możliwość to wykorzystanie pola *calling-station-id* [18]. Atrybut ten jest nadawany przez urządzenia *BRAS* / *BNG*, a następnie przesyłany za pomocą protokołu *RADIUS* do

serwerów *RADIUS* oraz odkładany w logach wraz z innymi polami. Jest to unikatowy identyfikator *CPE* w sieci operatora. Sposób nadawania konkretnej wartości pola *calling-station-id* jest specyficzny dla operatora i nie ma tutaj standardu w jaki sposób należy wypełnić wartość tego pola. Istotne jest, aby była ona unikatowa dla różnych *CPE,* celem zidentyfikowania konkretnego *CPE*, należącego do odpowiedniego użytkownika sieci. Atrybut *calling-station-id* jest uzupełniany automatycznie przez sieć, dzięki czemu jego wartości nie mogą być błędnie wypełnione przez użytkowników (np. za pomocą wprowadzenia na *CPE*). Opisywany atrybut rozwiązuje problem ze skojarzeniem logów *RADIUS Accounting* z konkretnym *CPE* w sytuacji błędnie wprowadzonej nazwy użytkownika (*user-name*). Ponadto, pole to zapewnia anonimowość, ponieważ opisuje zasoby sieciowe w sposób techniczny, więc nie przenosi żadnych danych osobowych lub innych mogących zostać skojarzonymi z danymi osobowymi. Dopiero przy wykorzystaniu dodatkowych danych ewidencyjnych możliwe jest skojarzenie wartości z konkretnym *CPE*. Jeśli zatem wystarczy detekcja samej unikatowości poszczególnych zdarzeń, system nie jest obostrzony wymaganiami prawnymi jak np. *RODO* [19].

Każdy operator ma swój sposób wypełniania wartości atrybutu *calling-station-id*. Popularnym jest nadawanie inicjalnej wartości na urządzeniach dostępowych jak *OLT* czy *DSLAM*, a następnie przekazywanie tej wartości w stronę urządzenia *BRAS / BNG* w nagłówkach stosowanych protokołów. W przypadku omawianej sieci jest to *opcja 105* protokołu *PPP*, której wykorzystanie jest niezbędne ze względu na to, że w przypadku sieci agregacyjnej opartej na protokole *Ethernet* wraz z sieciami wirtualnymi *VLAN* (ang. *Virtual LAN*) sesje *PPP* różnych *CPE* przesyłane są w ramach jednej sieci *VLAN*. Powoduje to, że jedyną możliwością rozróżnienia ruchu poszczególnych użytkowników między sobą są dodatkowe dane przesyłane w nagłówkach wykorzystywanych protokołów (źródłowy adres *MAC* również może podlegać translacji w ramach funkcji *Virtual MAC*). W sieci *ATM* problem ten jest mniejszy, ponieważ tam *CPE* zestawia z *BRAS / BNG* dedykowaną, unikatową ścieżkę na warstwie *ATM* (*VPI/VCI*), dzięki czemu na poziomie *BRAS / BNG* specyfikacja ścieżki (wartości *VPI* i *VCI*) jednoznacznie identyfikują użytkownika. W takim przypadku nie ma potrzeby stosowania opcji 105 z protokołu *PPP*, mimo, że protokół ten i tak jest stosowany.

2.7. PROBLEMY ZWIĄZANE Z WYKRYWANIEM AWARII

Awarie mogą obejmować znaczną część sieci operatora, stąd mechanizmy ich wykrywania są bardzo ważne. Również reakcja na wykryte zdarzenia musi być tak szybka, jak to tylko możliwe, gdyż gwarantuje to wysoką jakość dostarczanych usług.

Systemy zarządzania siecią (ang. *NMS - Network Management System*) [15], [20], które zapewniają dostawcy elementów sieciowych (ang. *NE – Network Element*) mogą być elementem systemów monitorowania sieci pod kątem awarii. Dywersyfikacja dostawców *NE*, a co za tym idzie także systemów *NMS* powoduje, że na globalny system monitorowania składa się wiele podsystemów różnego typu, z często różnym zakresem funkcjonalnym. Celem osiągnięcia jak największego stopnia jednorodności operatorzy stosują systemy ujednolicające określane jako *UNEMS* (ang. *Unified Network Element Monitoring System*) [21].

Awarie w sieci różnią się między sobą stopniem skomplikowania. Im bardziej złożony problem, tym może być on trudniejszy do wykrycia. Przykładem trudno wykrywalnej awarii sieciowej jest awaria polegająca na zawieszeniu się modułu odpowiedzialnego za przekazywanie danych na warstwie *L2* modelu *ISO/OSI* na urządzeniu. Urządzenie *DSLAM* jest urządzeniem dostępowym realizującym w głównej mierze funkcję dostępu do sieci dla użytkowników na warstwie fizycznej (*L1*). Z tego powodu klasyczne systemy monitorowania analizują właśnie tę warstwę, która w przypadku opisywanej awarii działa poprawnie, stąd wspomniane systemy nie wykazują awarii. Awaria na warstwie *L2* uniemożliwia przesyłanie ruchu dalej w sieci w stronę routerów *BRAS / BNG*. Z poziomu urządzeń *BRAS / BNG*, agregują one znacznie więcej urządzeń *DSLAM*, dlatego awaria pojedynczego z nich (lub części urządzenia) może pozostać niezauważona.

Systemy *NMS* są w stanie obserwować zdefiniowane zasoby sieciowe jednego producenta sprzętu, jak np. *NE*, karta, port czy ścieżka logiczna (ang. *VP – Virtual Path*). Zasoby do monitorowania są zazwyczaj konfigurowane manualnie, co powoduje, że tylko predefiniowane rodzaje awarii zostaną wykryte. Te, których specyfika nie została przewidziana pozostaną niezauważone. Stopień skomplikowania aktualnie stosowanych rozwiązań sieciowych (sieci prywatne *VPN* – ang. *Virtual Private Network*, sieci *CDN* – ang. *Content Delivery Network*, wielokrotna enkapsulacja protokołów, stosowanie balansowania ruchu itd.) powoduje, że wykrywanie wszystkich rodzajów awarii staje się coraz trudniejsze.

2.8. DOSTĘPNE ROZWIĄZANIA WYKRYWANIA AWARII

Ze względu na rosnący stopień skomplikowania sieci nie ma możliwości zdefiniowania każdego parametru, który będzie monitorowany za pomocą systemów typu *NMS*, nawet jeśli zastosowane byłyby systemy adaptacyjne opisane w pracach [22] i [23]. Oznacza to, że występuje potrzeba monitorowania na wyższych warstwach modelu *ISO / OSI* [1] celem dostarczenia wiarygodnej informacji o awariach występujących w sieci, patrząc na sieć z poziomu usług.

Jeden ze sposobów monitorowania tego typu, opisany w pracy [24], specyfikuje wykrywanie awarii z wykorzystaniem grafów celem lepszego zobrazowania awarii w strukturze sieci. Natomiast nie bierze on pod uwagę możliwości odbioru asynchronicznych notyfikacji, a także specyfiki protokołów *PPP* [17] i *RADIUS* w sieci, która jest przedmiotem zainteresowania autora tej rozprawy.

W zakresie ogólnodostępnych rozwiązań interesujący jest patent z *USA* [25], który definiuje proces wykrywania awarii bazujący na liczbie żądań otrzymywanych z alternatywnych (zapasowych) ścieżek sieciowych przed aktywacją alarmu (co jest wykonywane w momencie przekroczenia odpowiedniego poziomu liczby żądań). Opatentowane rozwiązanie skupia się na aplikacjach webowych i żądaniach przekazywanych do serwerów nazw *DNS* (ang. *Domain Name Server*). Aktywacja alarmu następuje po przekroczeniu odpowiedniego poziomu liczby żądań z tej samej lokalizacji, a opatentowane rozwiązanie skupia się wyłącznie na serwerach *DNS* i nie jest dedykowane do sieci dostępowej oraz agregacyjnej wykorzystującej protokoły *PPP* i *RADIUS*. Rozwiązanie przytoczonego patentu [25] nie zakłada korelacji danych pod kątem sesji użytkowników.

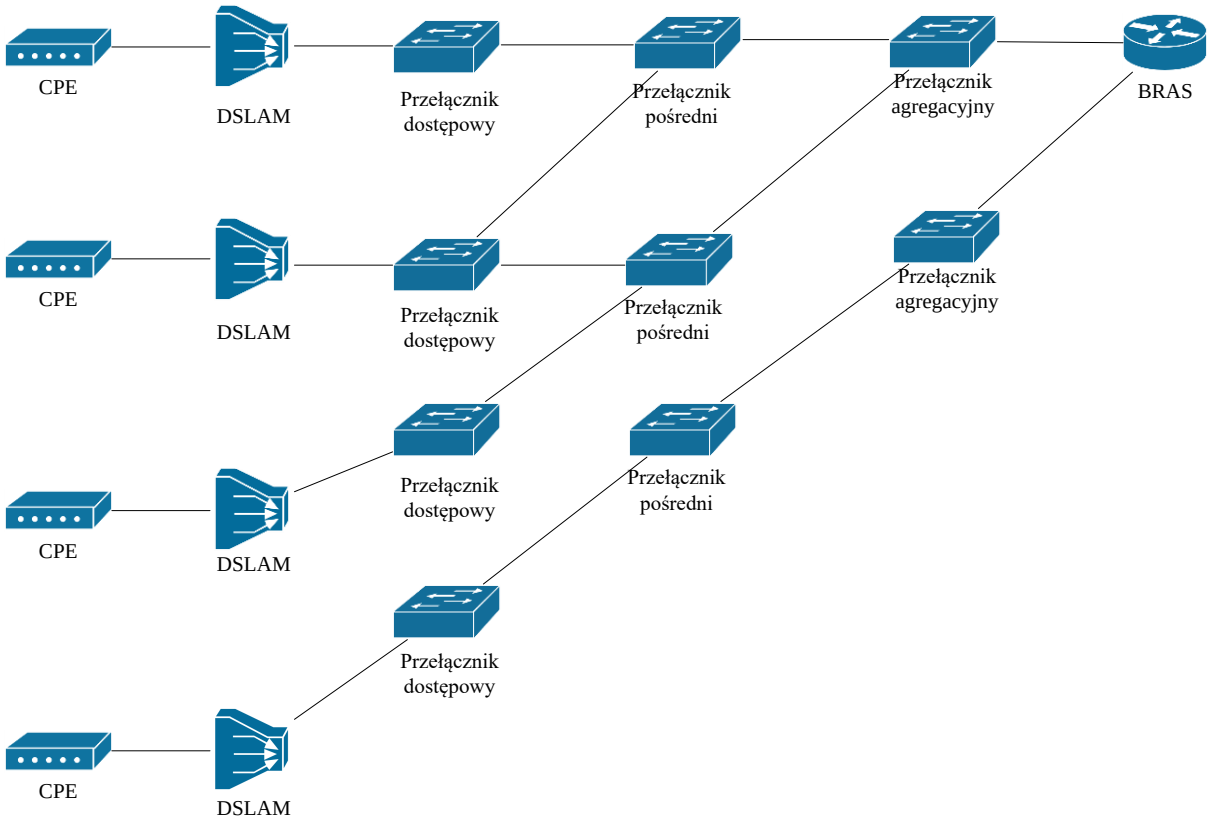
Elastyczność sieci jest jedną z istotnych zalet, którą można wykorzystać w procesie wykrywania awarii. Ważnym zagadnieniem w tej domenie jest odporność na awarie, która definiuje możliwość poprawnej pracy sieci pomimo awarii. Najjar i Gaudiot w [26] definiują oraz analizują prawdopodobieństwo rozłączenia usług w rodzinie grafów sieci. Wprowadzają współczynniki *NR(p)* (ang. *Network Resilience*) i *RNR(p)* (ang. *Relative Network Resilience*) jako probabilistyczne wskaźniki odporności sieci na awarie. Definicja współczynnika *NR(p)* została określona jako liczba awarii, którą sieć może obsłużyć, bez wpływu na użytkowników z prawdopodobieństwem $(1 - p)$. Analiza przedstawiona przez autorów pokazuje, że odporność sieci wzrasta wraz z rozmiarem grafów sieci.

Z kolei Sterbenz i inni w pracy [27] omawiają problemy odporności sieci. Definiują parametr N_k opisujący zachowanie sieci: normalne, częściowo pogorszone oraz poważnie pogorszone. Definiują także istotną grupę parametrów (P_k) określających stan operacyjny usług w sieci: stan akceptowalny, stan osłabiony (w omawianej pracy określony jako *impaired*) oraz stan nieakceptowalny. Pełny stan sieci opisywany jest przez parametr S_k, który jest funkcją N_k i P_k

$$S_k = (N_k, P_k) \tag{5}$$

Z innych zdefiniowanych parametrów przez autorów jest: odporność sieci R_{ij} na granicach B_{ij} pomiędzy dwoma warstwami L_i oraz L_j.

Wśród dostępnych ważnych artykułów oraz patentów opisujących ten problem należy wymienić [28], [29], [30], [31]. Każde z opisanych rozwiązań jest dedykowane dla specyficznego rodzaju sieci (różnego od analizowanej w tej rozprawie) i jej zachowaniu. Ponadto autorzy opisują mechanizmy kompensacji ruchu sieciowego w trakcie awarii, np. poprzez przełączenie ruchu na ścieżkę zapasową.



Rys. 2.10: Poglądowa (uproszczona) architektura sieci agregacyjnej

Dla sieci agregacyjnej przedstawionej na Rys. 2.10 występuje ograniczona możliwość utrzymania pełnej funkcjonalności sieci w przypadku awarii. Spowodowane jest to drzewiastą architekturą sieci. Jeśli jest możliwość pionowego połączenia węzłów to wtedy taka funkcjonalność byłaby możliwa do realizacji, a zatem istniałaby opcja zastosowania rozwiązania zaproponowanego przez Rak w [32]. Pomimo występowania różnorodności architektury sieci agregacyjnych automatyczne przełączanie ruchu na ścieżkę zapasową jest w tych sieciach stosowane rzadko. Główną przyczyną jest brak alternatywnych łączy na początkowych stopniach sieci, tj. od *CPE* do *DSLAM* oraz od *DSLAM* do pierwszego przełącznika w sieci agregacyjnej (tzw. przełącznik dostępowy, ang. *Access Switch*). Z tego powodu rozwiązania opisujące automatyczne przełączanie na ścieżki zapasowe rzadko może być zastosowane w sieciach agregacyjnych (ze względu na ich typową architekturę).

Analiza dostępnej obecnie literatury wskazuje, że problemy postawione w tezie tej rozprawy nie zostały dokładnie zgłębione. Istotnymi aspektami wartymi poruszenia są zdarzenia odbierane asynchronicznie, w szczególności pod kątem protokołów *PPP* oraz *RADIUS*, które to protokoły w swoim podstawowym sensie (zgodnie z definicją) nie służą do monitorowania sieci. Wszystkie elementy (pasywne i aktywne) występujące pomiędzy *CPE*, a *BRAS* (ang. *Broadband Remote Access Server*); takie jak *DSLAM, OLT* – ang. *Optical Line Termination*, przełączniki agregacyjne, kable; są skonfigurowane do przesyłania danych co najwyżej na warstwie łącza modelu *ISO / OSI* bez identyfikacji danych użytkownika. Protokół *PPP* jest jedną z możliwości enkapsulacji danych między *CPE*, a *BRAS*. Dane dotyczące rozliczeń protokołu *RADIUS* (ang. *RADIUS Accounting*) są automatycznie przesyłane przez serwery, co oznacza, że rozwiązanie to nie wymaga żadnych dodatkowych funkcjonalności do wdrożenia na *NE* oraz nie generuje dodatkowego obciążenia tych elementów. Komunikacja pomiędzy *NE* jest generowana w momencie wykrycia zdarzenia, a taki wolumen ruchu jest pomijalny (por. 2.3). Ostatnim istotnym czynnikiem każdego systemu jest jego wydajność. Kiedy liczba otrzymywanych zdarzeń dla klasycznej polskiej sieci sięga średnio tysięcy na sekundę, a każde z nich musi zostać przeanalizowane, zastosowane rozwiązania muszą być odpowiednio wydajne.

3. DETEKCJA AWARII W OPARCIU O DANE RADIUS

Rozwiązania opisane w rozdziale 2.8 nie są dostosowane do charakterystyki sieci dostępowej wykorzystującej protokoły *PPP* oraz *RADIUS*. Ponadto, rozwiązania te nie zapewniają możliwości monitorowania każdego elementu sieci dostępowej niezależnie od tego, czy te elementy są aktywne (jak przełączniki, urządzenia *DSLAM*), czy pasywne (jak kable). Ostatecznie, rozwiązania te nie umożliwiają monitorowania wspomnianych elementów niezależnie od tychże (bez komunikacji z nimi). Ze względu na opisane ograniczenia występuje potrzeba stworzenia niezawodnego sposobu monitorowania poszczególnych elementów sieci dostępowej i agregacyjnej, który nie miałby opisanych ograniczeń. Istotnym aspektem rozwiązania jest monitorowanie awarii przy całkowitym braku dostępu do monitorowanych urządzeń, co rozwiązuje problem monitorowania elementów pasywnych. Możliwość realizacji monitoringu pasywnego podnosi niezawodność rozwiązania także dla urządzeń aktywnych, z uwagi na fakt, że dla takich zasobów istnieje ryzyko, że w trakcie awarii nie będą zapewniać wiarygodnych informacji o samych sobie.

Celem opracowania przez autora nowej metody monitorowania sieci jest zatem zwiększenie skuteczności wykrywania awarii za pomocą podejścia do monitorowania od strony stanu samych usług, nie od strony urządzeń sieciowych. Dopiero w przypadku wykrycia problemów z usługami następuje próba wnioskowania, który element sieci powoduje awarię. Podejście to jest odwrotne w stosunku do opisanych wcześniej systemów *NMS*, które w pierwszej kolejności monitorują zasoby sieciowe, a w przypadku wykrycia problemów wnioskuje się, jakie usługi są objęte danym problemem.

Sesja *PPP* nawiązywana jest od *CPE* do *BRAS / BNG*, a między tymi urządzeniami występuje wiele pośredniczących elementów sieciowych. W przypadku wykrycia braku ciągłości sesji *PPP* można podejrzewać awarię dowolnego elementu na ścieżce pomiędzy punktami końcowymi. Jedna niedziałająca sesja *PPP* to zbyt mało, aby stwierdzić awarię sieci. Do wykrycia awarii konieczne jest potwierdzenie problemu na wielu, najlepiej wszystkich sesjach użytkowników przechodzących przez element sieciowy, na którym podejrzewa się wystąpienie awarii.

Celem detekcji awarii, należy zatem skojarzyć każde zdarzenie z dziennika zdarzeń z jego pełną architekturą sieciową oraz zmierzyć korelację tych zdarzeń między sobą, pod kątem współwystąpienia na tym samym elemencie sieci. Jeśli dodatkowo problem z działaniem wielu usług zaczyna się w tym samym czasie – występują przesłanki do tego, aby podejrzewać wystąpienie awarii. Można zatem wyobrazić sobie algorytm detekcji, który definiuje się jako

minimalną liczbę jednoczesnych zerwań sesji (na wspólnym zasobie sieciowym), powyżej której aktywowany jest alarm. Przykładowym zasobem może być karta urządzenia *DSLAM*. W przypadku rozłączenia wszystkich sesji użytkowników na karcie, w sytuacji, w której sesje na innych kartach nie zostały zerwane, z dużym prawdopodobieństwem można stwierdzić podejrzenie awarii wskazanej karty. Zdarzenie takie określa się mianem grupowego zerwania sesji *PPP* na monitorowanym zasobie sieciowym (*GZS*).

W danych *RADIUS* występują dwa atrybuty umożliwiające kojarzenie zdarzeń z użytkownikami. Dane te nie umożliwiają jednak w bezpośredni sposób wnioskowania na temat architektury sieciowej danego połączenia. Architektura musi być opracowana w oparciu o dodanie dodatkowych informacji pochodzących z systemów ewidencyjnych sieci. Jako, że każde zdarzenie jest analizowane w czasie rzeczywistym, dane te muszą być dodawane w locie w wydajny sposób. Jedną z możliwości takiego kojarzenia danych ewidencyjnych jest lokalna kopia danych ewidencyjnych, do której możliwy jest natychmiastowy dostęp. W przypadku cyklicznej replikacji danych rozwiązanie takie ma wadę w postaci opóźnienia w otrzymywaniu informacji o zmianach w architekturze sieciowej operatora, co może ograniczać skuteczność metody wykrywania zmian.

Jednoczesne zerwanie usług nigdy nie następuje dokładnie w tym samym momencie, natomiast rozciąga się w pewnym oknie czasowym. W przypadku sesji *PPP* okno czasowe zerwania oscyluje zazwyczaj w granicach 3 minut i jest ono wynikiem konfiguracji sieciowej operatora. Konfiguracja ta dotyczy liczby utraconych pakietów typu *keep-alive*, po której dane urządzenie uznaje sesję *PPP* za nieaktywną. Oznacza to, że system analizujący grupowe zerwania sesji musi analizować dane zawsze z co najmniej takiego okresu czasu. Zerwania sesji *PPP* widoczne są w ramach zdarzeń dotyczących rozliczeń użytkowników. Nie ma potrzeby analizy danych dotyczących uwierzytelniania w tym zakresie. Skoro zerwanie generuje asynchroniczne notyfikacje o rozłączonych usługach oznacza to, że w okresie awarii system monitorowania będzie otrzymywał większą liczbę żądań, niż poza tym okresem. Musi być on zatem gotowy na zwiększony ruch spowodowany awariami, a to powoduje konieczność jego przeskalowania w stosunku do standardowego ruchu w sieci. Inny aspekt problemu w zakresie wydajności, jaki należy brać pod uwagę, to zdarzenia pogodowe, w szczególności burze, które również generują zwiększoną liczbę rozłączeń sesji użytkowników. Z tego powodu systemy monitorowania sieci są wrażliwe na zjawiska atmosferyczne i zazwyczaj w okresie anomalii pogodowych odnotowuje się zwiększony napływ żądań.

Do detekcji awarii konieczna jest analiza zdarzeń ze zdefiniowanego przedziału czasu, co powoduje konieczność buforowania otrzymywanych informacji o zaistniałych zdarzeniach.

Rozmiar tego bufora jest bezpośrednio związany z wielkością okna, w ramach którego urządzenia sieciowe uznają sesję *PPP* za nieaktywną. Liczba jednoczesnych zerwań sesji użytkowników w ramach danej awarii jest różna i zależy od liczby usług obsługiwanych przez monitorowany element sieci. Przykładem może być urządzenie *DSLAM*, którego pojemność nierzadko wynosi nawet 1 tys. portów. Nigdy jednak urządzenie nie jest w pełni obsadzone, co powoduje, że najlepszym progiem jest odniesienie wielkości grupowego zerwania do liczby wcześniej aktywnych usług. Jeśli wszystkie aktywne sesje na zasobie sieciowym zostały zerwane, można podejrzewać problem z nim występujący. W ramach ustalania wielkości progu należy wziąć pod uwagę możliwe straty sieciowe w ruchu *RADIUS Accounting*. Poszczególne progi poziomu procentowego zerwań powinny być ustalane indywidualnie dla poszczególnych rodzajów monitorowanych zasobów sieciowych, ponieważ inne relatywne wartości stosunku zerwanych sesji do całości będzie miała utrata rekordu o jednym zerwaniu sesji na zasobie o pojemności 1 tys. usług, a inny na zasobie o pojemności 10 usług.

3.1. ALGORYTM WYKRYWANIA AWARII – AWA

Algorytm wykrywania awarii stworzony w ramach prac nad rozprawą doktorską został skrótowo określony jako *AWA*. Bazuje on na korelacyjnej analizie danych odbieranych w czasie rzeczywistym za pomocą zdarzeń typu *Accounting* protokołu *RADIUS*. Analiza polega na grupowaniu otrzymywanych zdarzeń pod kątem elementów sieciowych (*NE*), na których wystąpiły one w określonym momencie czasu. Następnie, jeśli na *NE* występuje określona liczba zdarzeń w zdefiniowanym zakresie czasowym, wnioskuje się o początku wystąpienia zdarzenia grupowego zerwania sesji *PPP* (*GZS*) na tymże *NE*. Częścią realizacji algorytmu *AWA* jest pozyskanie informacji o działaniu sieci, aby móc kojarzyć zdarzenia z każdym elementem sieciowym występującym na ścieżce realizacji usług. Ze względu na to, do działania algorytmu niezbędne jest posiadanie pełnej wiedzy o architekturze sieci. Podstawą procesu wykrywania awarii przez *AWA* jest detekcja zbieżnych zdarzeń w określonym przedziale czasowym.

W ramach algorytmu *AWA* zdefiniowano następujące parametry, które zostały użyte w dalej pokazanych schematach algorytmu:

- *T* – okres analizy *GZS*: przedział grupowania zdarzeń, tj. okno czasowe, w ramach którego przyjmuje się, że zdarzenia występujące w jego ramach dotyczą jednego konkretnego zdarzenia w sieci,
- *t* – czas trwania zdarzenia: od wykrycia *GZS*, do dowolnego wyjścia algorytmu,

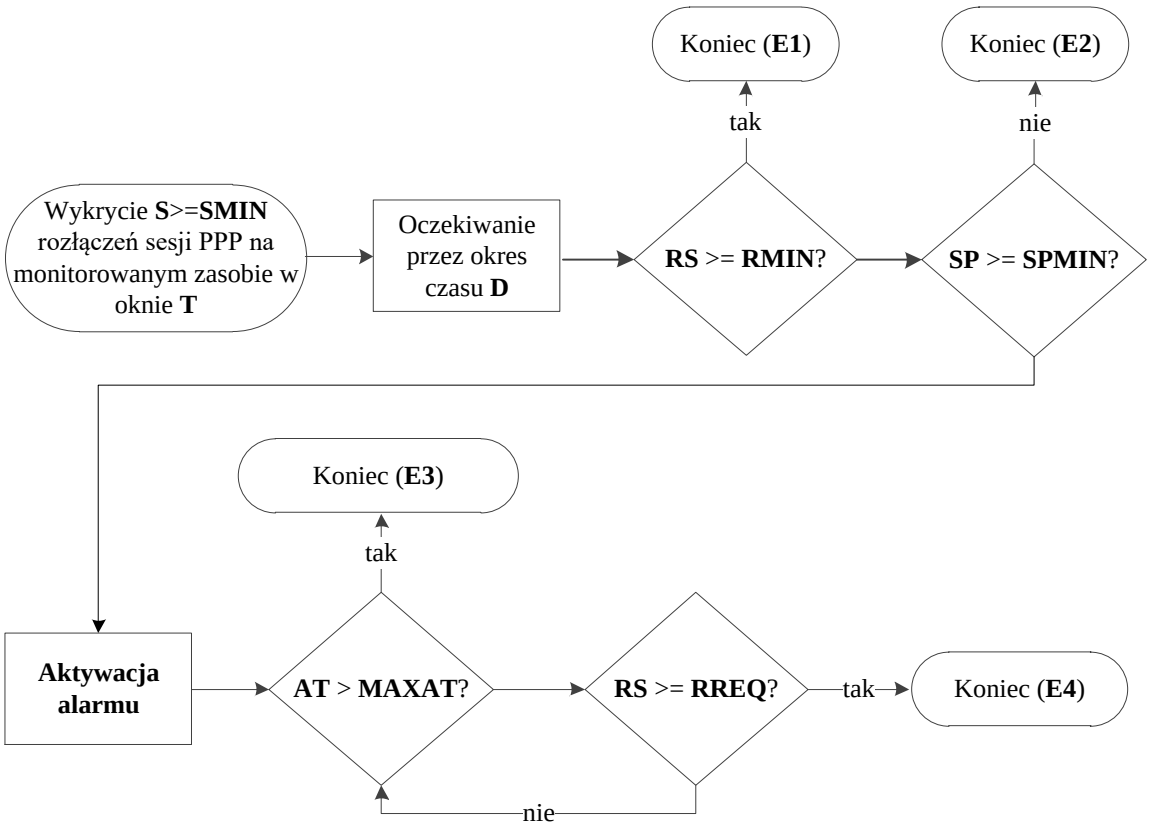
- *S* (*Stops*) – liczba zerwanych sesji w danym zerwaniu grupowym,
- *A* (*Active*) – liczba aktywnych sesji użytkowników przed wykryciem *GZS*,
- *SMIN* (*Stops Min*) – minimalna liczba zerwanych usług do kontynuacji analizy (poniżej tej liczby dane zdarzenie nie jest rozpatrywane jako *GZS*),
- *D* (*Delay*) – opóźnienie aktywacji alarmu po wykryciu zdarzenia *GZS*,
- *R* (*Returns*) – liczba sesji, które zostały nawiązane ponownie w trakcie obserwacji zdarzenia *GZS*,
- *RS* (*Returns to Stops*) – stosunek liczby powrotów do zerwanych sesji,

$$RS = \frac{R}{S} \tag{6}$$

- *RMIN* (*Returns Min*) – minimalny stosunek liczby powrotów sesji do liczby zerwanych (*RS*), powyżej którego zdarzenie jest odrzucane przed aktywacją alarmu o awarii (po odczekaniu czasu *D*),
- *SP* (*Stops Percentage*) – stosunek liczby zerwań do liczby aktywnych sesji przed zerwaniem – zawiera się w przedziale od 0 do 1 (włącznie),

$$SP = \frac{S}{A} \tag{7}$$

- *SPMIN* (*Stops Percentage Min*) – wartość *SP***,** powyżej której aktywowany jest alarm o awarii,
- *MAXT* – najdłuższy okres między dwoma zdarzeniami (np. *RADIUS Accounting*) generowanymi dla aktywnej usługi,
- *AT* (*Alarm Time*) – czas trwania alarmu,
- *MAXAT* (*Max Alarm Time*) – maksymalny możliwy czas trwania alarmu,
- *RREQ* (*Returns Required*) – minimalna wartość *RS*, powyżej której dezaktywuje się trwający alarm,
- *E1* (*Exit 1*) – pierwsze wyjście algorytmu (liczba nawiązanych sesji po okresie oczekiwania przekroczyła próg *RMIN*),
- *E2* (*Exit 2*) – drugie wyjście algorytmu (*SP* nie przekroczył progu *SPMIN*),
- *E3* (*Exit 3*) – trzecie wyjście algorytmu (przekroczono *MAXAT* – maksymalny czas trwania alarmu),
- *E4* (*Exit 4*) – czwarte wyjście algorytmu (awaria została usunięta, *RS* przekroczył *RREQ*).



Rys. 3.1: Algorytm wykrywania awarii (*AWA*) w oparciu o detekcję grupowych zerwań sesji *PPP*

W ramach algorytmu *AWA* na bieżąco analizuje się napływające dane *RADIUS Accounting* (Rys. 3.1). Po napływie każdego zdarzenia typu *Stop* algorytm cofa się w czasie o *T* jednostek i oblicza liczbę zdarzeń typu *Stop* w czasie *T* na analizowanym zasobie sieciowym. Liczba ta to parametr *S* (*Stops*). Następnie weryfikuje się, czy wartość *S* przekracza minimalną liczbę zdarzeń typu *Stop* (*S >= SMIN*). Jest to pierwszy krok weryfikacji liczby zerwań, a spełnienie warunku *S>=SMIN* powoduje wykrycie grupowego zerwania sesji *PPP* (*GZS*) oraz rozpoczęcie realizacji działań przedstawionych na Rys. 3.1. Na tym etapie mówimy o wystąpieniu zdarzenia *GZS*, które nie jest jeszcze zakwalifikowane jako alarm. Nie oblicza się wartości parametru *SP*, gdyż wymagałoby to większej mocy obliczeniowej. Następnie algorytm przechodzi w stan oczekiwania przez czas *D*. Jest to opóźnienie w generowaniu alarmu o awarii celem zmniejszenia wpływu krótkotrwałych zdarzeń na działanie algorytmu. W rzeczywistej sieci występuje duża liczba zdarzeń losowych i tymczasowych problemów, które znikają samoczynnie lub są wynikiem prowadzonych prac w sieci. Celem odrzucenia tego

typu zdarzeń algorytm dokonuje zatrzymania analizy na wspomniany czas *D* przed podjęciem dalszych czynności.

Opóźnienie *D* zależy od liczby alarmów, jakie w tym czasie ulegają samoczynnej dezaktywacji oraz od typu monitorowanego zasobu. Jeśli monitorowana jest karta urządzenia *DSLAM*, oznacza to, że usługowy zakres jej awarii jest stosunkowo niewielki (co oznacza, że opóźnienie w generowaniu alarmu nie wywiera skutku na dużą liczbę usług), a liczba alarmów dezaktywowanych automatycznie jest relatywnie duża. Istnieje zatem zasadność ustawienia wyższego czasu *D*: rzędu dziesiątek minut. Im dalej w stronę urządzenia *BRAS / BNG* w sieci agregacyjnej, tym urządzenia obsługują więcej usług i nie dopuszcza się na nich do występowania nawet krótkotrwałych przerw w świadczeniu usług. Liczba krótkotrwałych zdarzeń *GZS* spada wraz z przesuwaniem się w analizie danych w stronę urządzenia *BRAS / BNG*. Dla tego typu zasobu opóźnienie w generowaniu alarmu ma ponadto negatywne skutki w postaci zwiększającej się liczby usług, dla których awaria została przez czas *D* niewykryta. Stąd im zasoby są większe oraz znajdują się bliżej urządzenia *BRAS / BNG* – czas *D* powinien być krótszy.

Sesje użytkowników mają ustalony maksymalny czas trwania. To oznacza, że co najmniej w takim interwale muszą się zakończyć i być nawiązane ponownie. Jeśli z jakiegoś powodu (jak np. awaria zasilania) wszystkie sesje zostały nawiązane w tym samym momencie, będą one również nawiązywać ponowne połączenia w tym samym momencie czasu w przyszłości. To z kolei zawsze będzie generować *GZS*, ale powodem są zwykłe ponowne nawiązania sesji przez użytkowników, które zostały zsynchronizowane jakimś zdarzeniem z przeszłości. Oznacza to, że na sporych zasobach sieciowych nie można czasu *D* skrócić do zera, tak aby tego typu zdarzenia mogły filtrować się w tym oknie automatycznie. Czas ponownego nawiązania sesji *PPP* jest rzędu sekund, więc do zrealizowania takiej funkcjonalności wystarczy *D* również na poziomie kilku sekund.

Po upływie okresu *D* weryfikuje się stosunek powrotów sesji (*R*) z grupy sesji zerwanych do liczby sesji zerwanych (*S*) – jest to parametr *RS*. Jeśli przekracza on próg *RMIN* (minimalny stosunek powrotów do zerwań), wtedy algorytm kończy swoje działanie wyjściem *E1*. Przykładem są opisane wcześniej zsynchronizowane zerwania sesji *PPP*, gdzie po upływie czasu *D* stosunek *RS* wynosi 1.

W kolejnym kroku oblicza się wartość parametru *SP*. Jest to stosunek liczby zerwanych sesji do liczby sesji aktywnych przed tym zdarzeniem. Do obliczenia tego parametru, należy cofnąć się w historii zdarzeń o okno *MAXT* (opisany w tym podrozdziale okres, w którym występuje pewność, że każda aktywna sesja wysłała notyfikację o swojej aktywności, co daje

pewność poprawnego obliczenia liczby aktywnych sesji). Dalej weryfikuje się, czy wartość parametru *SP* jest większa lub równa od progu *SPMIN* i jeśli tak – aktywowany jest alarm. Od tego momentu dane zdarzenie *GZS* jest określane jako alarm, który może okazać się fałszywy w przypadku nie potwierdzenia awarii przez operatora. Gdy warunek *SPMIN* nie jest spełniony algorytm kończy swoje działanie wyjściem *E2*. Przyczyną zakończenia analizy za pomocą wyjść *E1* i *E2* są inne, skorelowane zdarzenia po stronie użytkowników (np. brak zasilania na pewnym obszarze) lub awarie na innych elementach sieci. Z punktu widzenia monitorowanego zasobu nie była to jednak awaria.

W trakcie trwania alarmu system cały czas monitoruje liczbę powrotów sesji *PPP* celem natychmiastowej dezaktywacji alarmu po usunięciu zdarzenia. Czas trwania alarmu został zdefiniowany jako parametr *AT* (ang. *Alarm Time*). Został także zdefiniowany próg *MAXAT*, który określa maksymalny czas *AT*, jaki może istnieć w systemie. Jego przekroczenie powoduje zakończenie analizy i wyjście z algorytmu wyjściem *E3*. Jest to zabezpieczenie przeciw nieskończonemu trwaniu alarmów. Taka sytuacja może wystąpić, gdy w trakcie trwania alarmu zmieniły się zasoby sieciowe, na których świadczone są usługi (służby usuwające awarię mogą taką operację przeprowadzić). Wtedy algorytm może nie wykryć powrotu usług do pracy. W innym przypadku, gdy z jakiegoś powodu system nie otrzyma notyfikacji o ponownym nawiązaniu sesji (przerwa w transmisji – dane są transmitowane za pomocą *UDP*; nie będzie więc retransmisji notyfikacji) również może dojść do niewykrycia końca awarii. Z tego powodu konieczne było wprowadzenie weryfikacji maksymalnego czasu *AT*.

W trakcie trwania alarmu weryfikuje się, czy stosunek *RS* przekracza *RREQ* (minimalny stosunek powrotów do zerwań, powyżej którego dezaktywuje się alarm). Jeśli tak, to algorytm kończy swoje działanie wyjściem *E4*. W przeciwnym razie algorytm cały czas czeka i obserwuje otrzymywane zdarzenia pod kątem powrotu zerwanych sesji.

Zaprezentowany algorytm *AWA* jest generyczny, dzięki czemu może zostać zastosowany do wykrywania awarii dowolnego elementu na ścieżce sesji *PPP*. Niezbędne jest skojarzenie każdego zdarzenia z zasobami sieci, których ono dotyczy, tak aby wiedzieć, które zdarzenia wskazują na awarie poszczególnych elementów sieci (por. architektura sieci na Rys. 2.10). Realizacja tych skojarzeń jest realizowana w pierwszym bloku schematu algorytmu *AWA* (por. Rys. 3.1) i jest ważnym zagadnieniem, wymagającym przeanalizowania w trakcie implementacji algorytmu. Powiązywanie zdarzeń z tym, których zasobów sieciowych dotyczą wymaga posiadania wiedzy o strukturze sieci danego operatora. Zgodnie z opisem przedstawionym w rozdziale 2.6 powiązania takie można stwierdzać w oparciu o pole nazwy użytkownika lub pole *calling-station-id* występujące w logu *RADIUS Accounting*. Wartości

tych pól, jak również struktura ewidencji sieci u operatora nie są ustandaryzowane. Każdy z operatorów może stosować różne rozwiązania ewidencji sieci oraz wypełniania wartości wspomnianych pól, odpowiadające jego potrzebom, a co za tym idzie mające różne cechy w zakresie możliwości i wydajności przeszukiwania globalnej bazy ewidencji tych danych. Z tego powodu wydajność algorytmów detekcji takich powiązań nie jest możliwa do przeanalizowania w sposób generyczny i może zostać zrealizowana wyłącznie na konkretnych przykładach.

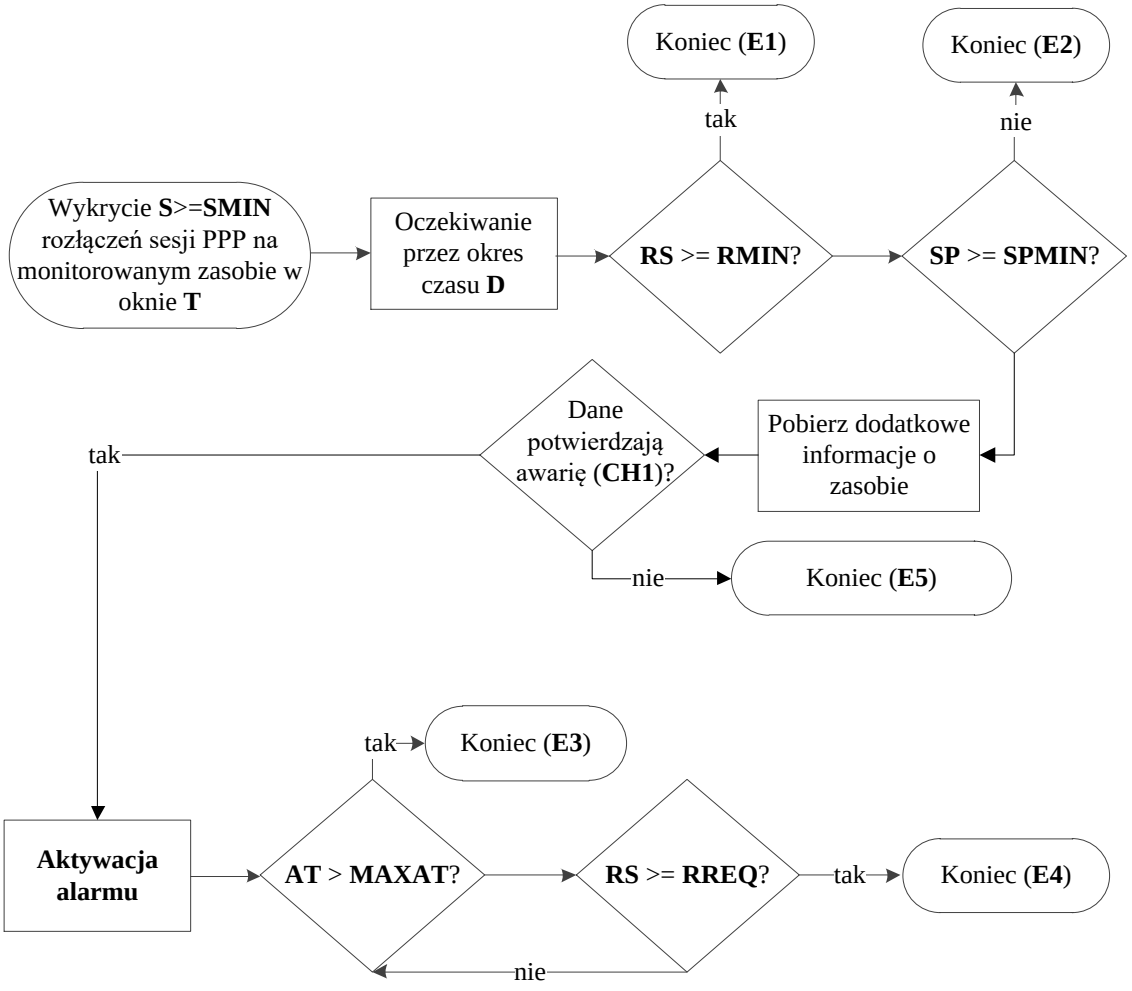
3.2. ALGORYTM AWA ROZSZERZONY O POBIERANIE INFORMACJI O ZASOBIE SIECIOWYM

Cechą algorytmu *AWA* przedstawionego w rozdziale 3.1 jest wykrywanie wszystkich zdarzeń powodujących grupowe zerwanie sesji na monitorowanym zasobie sieciowym. Nie każda taka sytuacja jest awarią. Im bliżej *CPE* położony jest monitorowany zasób, tym bardziej narażony jest on na fałszywe alarmy z powodu awarii zasilania w danym obszarze geograficznym, czy z powodu prowadzonych w danej lokalizacji prac planowych. Również zdarzenia burzowe generują dużą liczbę tego typu alertów. Z punktu widzenia operatora są to fałszywe alarmy niepotrzebnie absorbujące jego zasoby operacyjne. Z tego powodu, tak jak opisano wcześniej, dla takich zasobów ustala się wyższe wartości parametru *D*.

Na Rys. 3.2 przedstawiono algorytm wzbogacony o dodatkowy element pobierający informacje o zasobie sieciowym przed wygenerowaniem alarmu. Po pobraniu informacji dokonywane jest ich sprawdzenie (*CH1*), a w sytuacji, w której dane te nie potwierdzają awarii, algorytm kończy swoją pracę dodatkowym wyjściem *E5* (*Exit 5*). Jak wspomniano, największa liczba fałszywych alarmów występuje na *NE* znajdujących się blisko *CPE*, w szczególności na ostatniej pętli abonenckiej (tzw. ostatnia mila). Celem eliminacji alarmów dotyczących awarii zasilania można posłużyć się komunikacją z urządzeniami *DSLAM* lub *OLT* (dalej określane jako *MSAN* – ang. *Multi-Service Access Node*) i pobraniem z tych urządzeń informacji pochodzących z mechanizmu *„dying gasp”*. Funkcja ta działa w ten sposób, że w momencie gdy *CPE* traci zasilanie jest w stanie wysłać do urządzenia *MSAN* tzw. „pakiet ostatniego tchnienia” informujący o tym, że urządzenie właśnie wyłącza się z powodu braku zasilania. Taka informacja pozwala odfiltrować alarmy spowodowane awarią zasilania w danym obszarze geograficznym. Z kolei taka sytuacja nie jest dla operatora interesująca (w opisanej sytuacji wszystkie *CPE* utraciły zasilanie zasilane, ale samo urządzenie *MSAN* wciąż pracuje poprawnie – takie urządzenia mają zazwyczaj dodatkowe zasilanie bateryjne).

Opisywany algorytm może zostać rozszerzony o dodatkowe elementy (schemat na Rys. 3.2):

- *CH1* (*Check 1*) – weryfikacja, czy dane pobrane z urządzenia potwierdzają awarię,
- *E5* (*Exit 5*) – piąte wyjście algorytmu (dane pobrane z urządzenia nie potwierdzają awarii).



Rys. 3.2: Algorytm *AWA* rozszerzony o dodatkowe pobieranie informacji o zasobie sieciowym

Identycznie jest w przypadku monitorowania fizycznych zasobów pasywnych, którymi są połączenia kablowe. W takiej sytuacji niezwykle pomocną jest informacja z portu urządzeń aktywnych, do których te kable są podłączone. W przypadku potrzeby filtracji zdarzeń spowodowanych awarią zasilania, można użyć informacji pochodzącej z mechanizmu *dying*

gasp. Mechanizm ten aktywuje na urządzeniach dostępowych alarm o nazwie *LOP* (ang. *Loss of Power*) i/lub powoduje zwiększenie wartości licznika zdarzeń typu *LOP*.

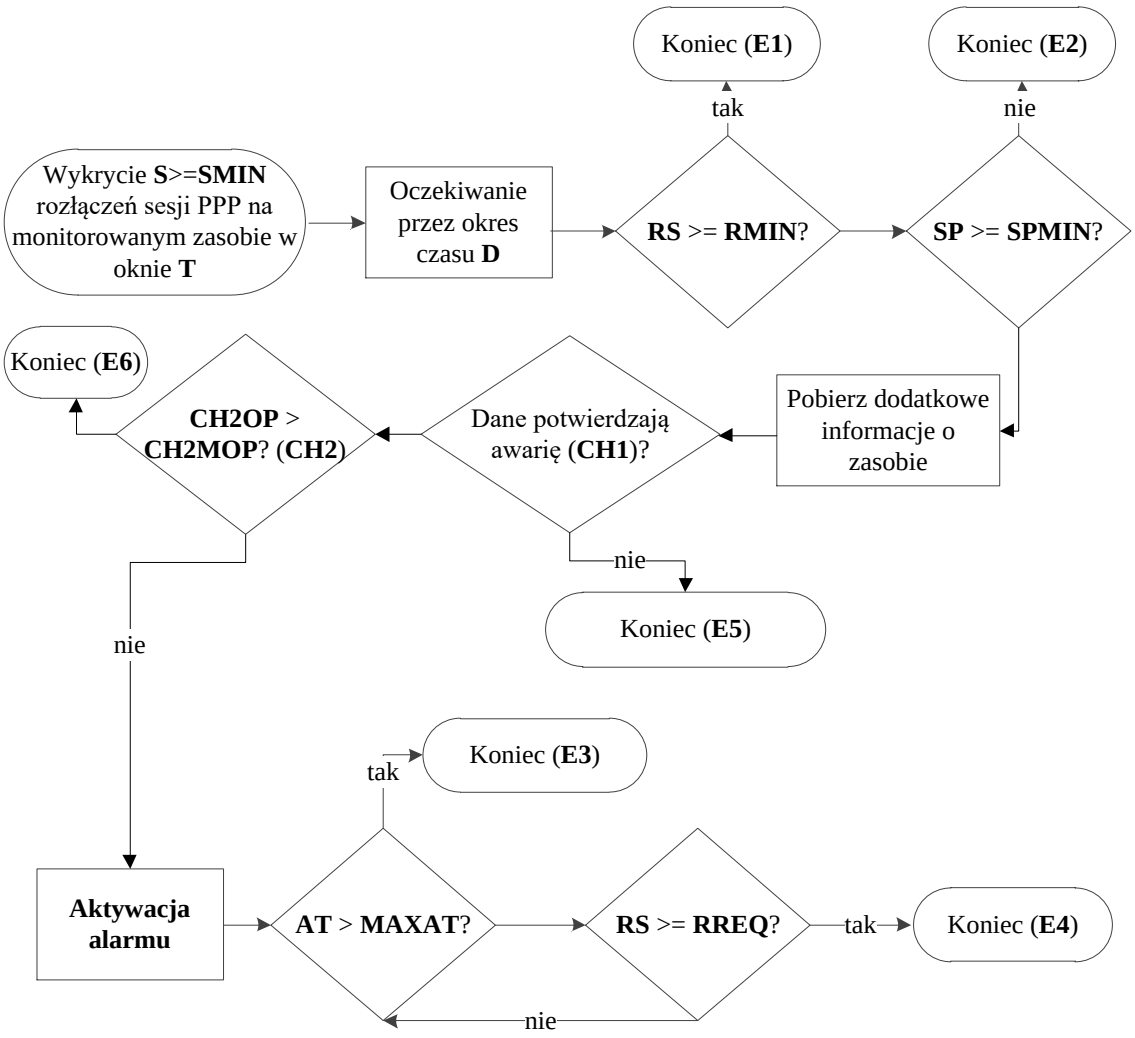
W przypadku, gdy także urządzenie *MSAN* jest objęte awarią zasilania, nie będzie możliwości pobrania z niego informacji o alarmach *LOP*. W takiej sytuacji jedyną możliwością wsparcia decyzyjności algorytmu jest informacja z urządzeń centralowych zasilających te urządzenia. Urządzenia zasilające również są podłączane do sieci zarządzania operatora i są w stanie wysłać alerty o utracie zasilania, jak też poinformować o przełączeniu na zasilanie bateryjne. To wszystko daje dodatkowe możliwości analityczne w zakresie polepszenia skuteczności generowania alarmów.

3.3. NAKŁADKA ALGORYTMU AWA (NAWA)

Zaletą *AWA* jest możliwość realizacji na dowolnym elemencie sieci, znajdującym się między urządzeniem *CPE*, a *BRAS / BNG* (por. Rys. 2.10). Przy założeniu aktywacji *AWA* na każdym elemencie sieci, występować będą zwielokrotnione alarmy związane z wykrywaniem grupowych zerwań sesji na każdym *NE* niżej w architekturze sieci (w lewą stronę na Rys. 2.10) od *NE*, na którym faktycznie wystąpiła awaria. Przykładowo, awaria urządzenia *DSLAM* spowoduje aktywację alarmu na tymże urządzeniu, ale także na wszystkich jego kartach oraz kablach podłączonych do tych kart. Tylko alarm na elemencie najbliższym w stronę urządzenia *BRAS / BNG* jest prawdziwy. Powoduje to, analogicznie jak w sytuacji z awariami zasilania, znaczą liczbę fałszywych alarmów, które powinny być filtrowane systemowo. Jedynym elementem sieci, który nie ma opisanej wady jest korzeń drzewa sieci, a w omawianej architekturze jest to *BRAS / BNG*.

Celem rozwiązania opisanego problemu należy zweryfikować, czy na elemencie znajdującym się wyżej w hierarchii sieciowej (w prawą stronę na Rys. 2.10) wykryto alarm. Rys. 3.3 przedstawia algorytm *AWA* wzbogacony o ten element weryfikacji, realizowany przed aktywacją alarmu (*CH2 – Check 2*). Weryfikacja *CH2* sprawdza, czy na elemencie znajdującym się wyżej w hierarchii sieci wykryto alarm dla tych samych usług, dla których analizowane jest *GZS*. W takim przypadku algorytm kończy swoje działanie wyjściem *E6* (*Exit 6*), a w przeciwnym razie alarm jest aktywowany. W celu realizacji porównania *CH2* definiuje się dodatkowy parametr *CH2OP* (*Check 2 Overlapping Percentage*), który opisuje stosunek usług powtarzających się między wykrytymi zdarzeniami na porównywanych stopniach monitorowania. *CH2OP* przyjmuje wartości z zakresu 0-1 (0% – 100%). Jeśli wszystkie usługi wykryte w ramach *GZS* są również wykryte w ramach alarmu aktywowanego na wyższym stopniu monitorowania – wtedy *CH2OP* przyjmuje wartość 1. Jeśli 80% usług znajduje takie

pokrycie, wtedy *CH2OP* = 0,8 itd. *CH2MOP* (*Check 2 Overlapping Min Percentage*) opisuje próg, którego przekroczenie przez parametr *CH2OP* oznacza uznanie dwóch awarii za skojarzonych ze sobą i algorytm kończy swoją pracę wyjściem *E6* (uznając nadrzędną awarię usytuowaną wyżej w hierarchii sieciowej). Podczas zliczania rodzajów zakończeń analiz zdarzenia typu *E6* zliczane są jako *ASO* (*All Stops Overlapped*).



Rys. 3.3: Algorytm *AWA* rozbudowany o weryfikację stopni monitorowanych zasobów

Do proponowanego algorytmu monitorowania sieci wartościowym rozwiązaniem jest wprowadzenie dodatkowych elementów (por. Rys. 3.3):

- *CH2* (*Check 2*) – weryfikacja, czy wykryto zdarzenie na elemencie wyższym w hierarchii sieciowej i zdarzenie to dotyczy tych samych usług,

- *E6* (*Exit 6*) – szóste wyjście algorytmu (wykryto zdarzenie na elemencie wyższym w hierarchii sieciowej),
- *CH2OP* (*Check 2 Overlapping Percentage*) – procent usług powtarzających się w zdarzeniu wykrytym na wyższym stopniu monitorowania,
- *CH2MOP* (*Check 2 Min Overlapping Percentage*) – minimalna wartość *CH2OP*, powyżej której uznaje się awarie za skojarzone ze sobą (przykładowo wartość 0,6 oznacza że wystarczy, że 60% usług jest wykrytych również w trakcie zdarzenia na wyższym stopniu), aby dezaktywować automatycznie alarm na niższym stopniu wyjściem *E6*. Parametr przyjmuje wartości z zakresu od 0 do 1.

3.4. KORELACJA Z INNYMI SYSTEMAMI MONITOROWANIA

W sytuacji, w której w sieci istnieje już pewien zestaw systemów monitorujących, algorytm *AWA* musi zostać wdrożony ze świadomością współistnienia z innymi systemami. Oznacza to konieczność integracji systemów monitorujących w jednym miejscu, celem usprawnienia pracy służb obsługujących awarie.

Może zostać to osiągnięte na kilka sposobów. Jednym z nich jest wdrożenie nadrzędnego (unifikującego) systemu, do którego awarie notyfikować będą wszystkie inne systemy monitorowania (por. *UNEMS* w 2.7). Dzięki temu jednostki operacyjne mają do obsłużenia wyłącznie jedno narzędzie, a każdy nowy system monitorowania może być podłączony do narzędzia centralnego jako dodatkowy komponent. Zwiększanie liczby systemów monitorujących sieć nie jest niczym nadzwyczajnym. Jednym z systemów, które u kooperującego operatora zwiększyły tę pulę jest system implementujący algorytm *AWA*. Innym momentem zwiększenia puli systemów jest wprowadzenie do sieci nowego typu urządzeń *NE*, wraz z którym dostarczony jest nowy typ *NMS* (por. 2.7).

Systemy monitorujące mogą także weryfikować awarie między sobą, tak aby nie generować wielokrotnych alarmów. W przypadku takiej integracji systemy są zintegrowane każdy z każdym, a w przypadku wykrycia awarii weryfikuje się, czy inne systemy jej nie wykryły (celem wyeliminowania konieczności manualnej obróbki wielu alarmów dot. tego samego zdarzenia). W takiej sytuacji, wraz z ze wzrostem liczności systemów wzrasta liczba koniecznych do wykonania integracji międzysystemowych, a liczba narzędzi, za pomocą których muszą pracować jednostki operacyjne operatora telekomunikacyjnego jest równa liczbie systemów monitorujących. Rozwiązanie to nie jest optymalne z punktu widzenia obsługi manualnej oraz koniecznej do wykonania liczby integracji międzysystemowych.

Ostatnim sposobem jest wybranie systemu unifikującego spośród już istniejących systemów monitorujących. Wtedy wybrany system zajmuje się właściwym mu monitorowaniem, ale dodatkowo staje się unifikującym wobec innych (zbiera i prezentuje informacje ze wszystkich innych systemów). System taki musi zatem zapewniać elastyczność integracji nowych źródeł danych.

3.5. INNE ROZWIĄZANIA PRZY BRAKU SESJI PPP W ARCHITEKTURZE SIECI

Rys. 2.9 oraz Rys. 2.10 zaprezentowane w rozdziale 2 pokazują referencyjną architekturę sieci z wykorzystaniem kluczowych, dla opracowanego algorytmu, elementów takich jak: *BRAS / BNG*, sesje *PPP* oraz protokół *RADIUS*. Nie wszyscy operatorzy w ramach swoich sieci dostępowo-agregacyjnych stosują sesje *PPP* do przydziału adresów *IP* urządzeniom *CPE* oraz podtrzymania sesji między *CPE*, a *BRAS* / *BNG*. Rozwiązanie to jest popularne wśród operatorów stosujących w swoich sieciach dostępowych techniki takie jak *DSL* lub *FTTH* (ang. *Fibre To The Home*), ale nie jest stosowane zawsze. W sporej grupie sieci tego typu do przydziału adresów stosuje się protokół *DHCP* (ang. *Dynamic Host Configuration Protocol*) [33]. Jest on popularny w sieciach *HFC* (ang. *Hybrid Fibre-Coaxial*). Protokół *PPP* można używać do dwóch celów jednocześnie: przydziału adresów oraz monitorowania łączności ścieżki, ponieważ zestawia on podtrzymywaną sesję (por. 2.5). W przypadku *DHCP* nie ma takiej sesji, zatem protokół ten nie zapewnia informacji o ciągłości połączenia. W *DHCP* na początku pracy urządzenie zgłasza się do serwera *DHCP* z prośbą o przydzielenie adresu (*DHCPREQUEST* [33]), a następnie po jego uzyskaniu wykorzystuje go przez ustalony okres czasu (tzw. *lease time*, np. równy 24h). W trakcie *lease time*, z punktu widzenia protokołu *DHCP*, nie ma żadnych informacji na temat osiągalności urządzenia *CPE* w sieci.

Sytuacja z protokołem *PPP* jest o tyle wygodna, że do monitorowania stanu sesji użytkownika nie trzeba wykonywać żadnych zmian w konfiguracji sieci, ponieważ protokół ten w naturalny dla siebie sposób wspiera funkcję podtrzymania połączenia. Można to wykorzystać na potrzeby monitorowania. W sytuacji, w której nie da się użyć protokołu przydziału adresu do monitorowania stanu urządzeń należy zastosować dodatkowy, inny protokół. Może nim być: *BFD* (ang. *Bidirectional Forwarding Detection*) [34], *FDEP* (ang. *Failure Detection/Exploration Protocol*) [35] lub *HMP* (ang. *Host Monitoring Protocol*) [36]. Istotne jest zastosowanie protokołu, który w swoim działaniu wykorzystuje pakiety typu *keep-alive*, analogicznie jak *PPP*. Takie rozwiązanie umożliwia pozyskiwanie informacji o ciągłości sesji w czasie zbliżonym do rzeczywistego. Ważna jest także centralizacja rozwiązania oraz

wykorzystanie danych w czasie rzeczywistym w systemie monitorującym. W przypadku protokołu *PPP* naturalnym jest, że operacje *AAA* (ang. *Authentication, Authorization and Accounting*) są realizowane w mocno scentralizowany sposób za pomocą systemu *RADIUS*. To w łatwy sposób umożliwia pozyskanie informacji z ograniczonej liczby punktów, bez konieczności integracji z dużą liczbą elementów sieci. W przypadku wykorzystania innych protokołów architektura jest inna, prawdopodobnie bez użycia protokołu *RADIUS* jako pośrednika w przekazywaniu informacji o stanie sesji. Może to oznaczać integrację np. z każdym routerem w sieci, co zwiększa stopień skomplikowania rozwiązania.

3.6. STRATY DANYCH RADIUS ACCOUNTING

Dane *RADIUS Accounting* są zazwyczaj przesyłane przy pomocy protokołu *UDP* w warstwie transportowej modelu *ISO / OSI*. Istnieje możliwość przesyłania tych danych za pomocą protokołu połączeniowego, jakim jest *TCP*, który zapewnia pewność transmisji (lub w przypadku braku możliwości transmisji poinformowanie o tym fakcie). Protokół ten wykorzystuje się jednak tylko w ramach wewnętrznych serwerów *RADIUS*. Wykorzystanie danych *Accounting* do monitorowania stanu sieci jest dla systemu *RADIUS* zadaniem dodatkowym (por. Rys. 2.9). Z tego powodu systemy *RADIUS* przekierowują te zdarzenia do systemu monitorującego z jak najmniejszym własnym zaangażowaniem, celem braku wpływu na jakość podstawowej usługi *RADIUS* w sieci. Dlatego do przekierowywania ruchu używa się protokołu *UDP*, który jest prostszy od *TCP*. Ruch ze źródła jest wysyłany w postaci ramek określanych jako *datagramy*, bez konieczności nawiązywania połączenia i negocjacji jego parametrów, jak to ma miejsce w przypadku zastosowania protokołu *TCP*. Odciąża to serwery *RADIUS*, które muszą ustawiać wyższe priorytety zadań na te związane z funkcjami *AAA*. Niestety protokół *UDP* nie zapewnia pewności transmisji, co powoduje, że normalnym zjawiskiem są straty pakietów i systemy wykorzystujące te dane powinny być na takie straty przygotowane.

Z punktu widzenia systemu monitorującego sieć straty danych mogą dawać różne skutki. W przypadku wystąpienia grupowego zerwania sesji *PPP* obserwuje się pik ruchu *RADIUS Accounting Stop* dotyczący wszystkich zerwanych sesji za uszkodzonym zasobie. Do poprawnej identyfikacji problemu dokonuje się oceny jak dużo sesji aktywnych zostało przerwanych. Ocena ta jest uzależniona od cofnięcia się w historycznych rekordach *RADIUS Accounting* i obliczenia jak dużo rekordów *Start* lub *Interim-Update* zostało otrzymanych w okresie *MAXT*. Należy pamiętać, że zarówno w ramach danych *Stop* z momentu zdarzenia jak i w ramach ruchu wcześniejszego mogły wystąpić straty danych. Jeśli wystąpiły one

w ruchu poprzednim może dojść do sytuacji, w której okazuje się, że system otrzymał rekord *Stop* dla sesji o której „nie wiedział", a to spowoduje obliczenie wartości parametru *SP* powyżej 100%. Jest to w rzeczywistości niemożliwe, jednak ze względu na straty danych obliczenia mogą dawać taki wynik. W przypadku strat na ruchu *Stop* z okresu grupowego zerwania błąd obliczeń występuje w przeciwną stronę. Wtedy parametr *SP* osiągnie wartość niższą niż rzeczywista. Dlatego wartość *SP*, która potwierdza awarię zawiera się w pewnym przedziale i musi to być uwzględnione przy ustawianiu progu *SPMIN*. Im większy jest monitorowany zasób, tym mniejszy błąd względny z tytułu pojedynczych utrat pakietów.

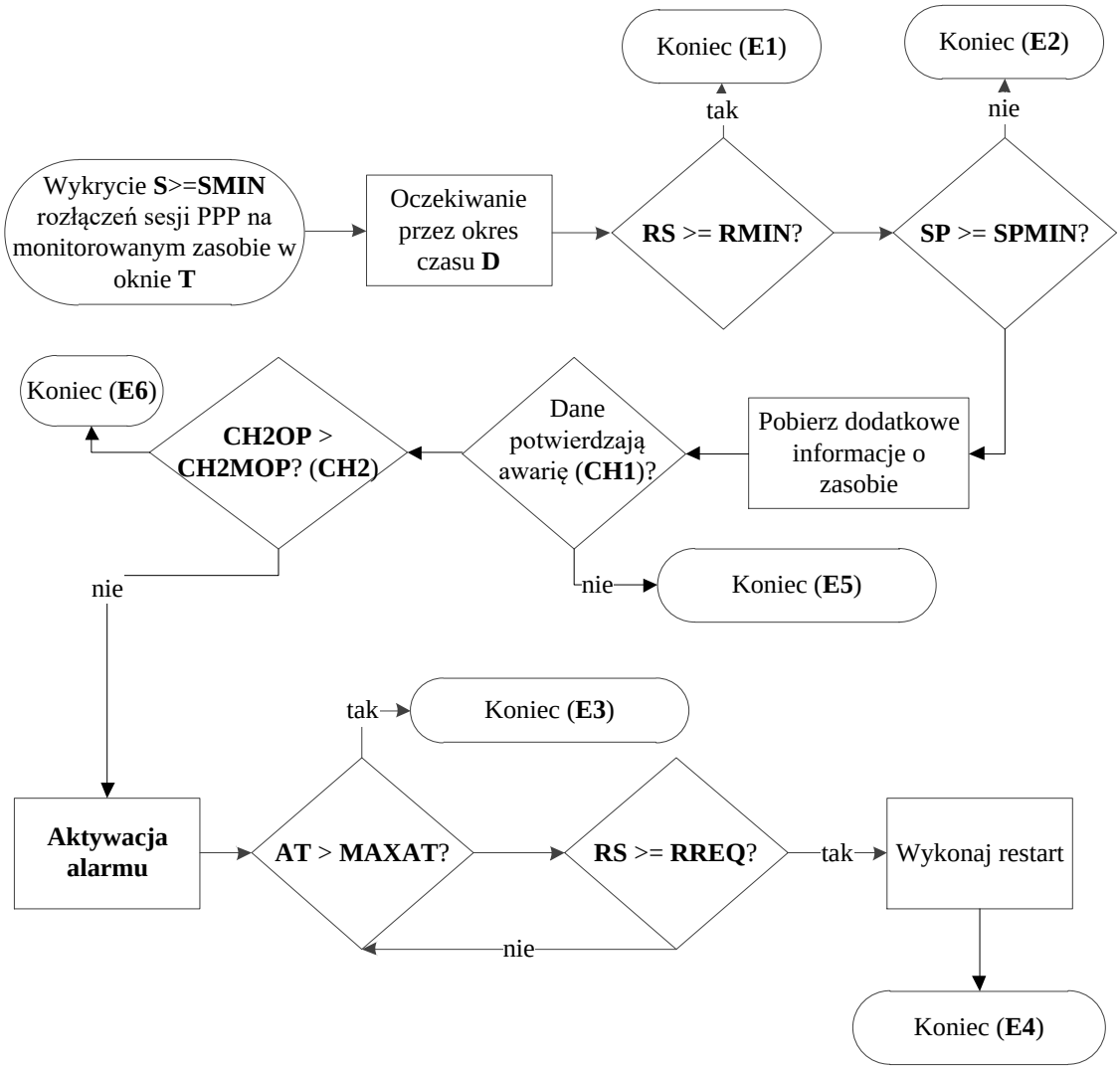
Celem poprawy sytuacji związanej z utratą pakietów w sieci stosuje się rekordy *Accounting* typu *Interim-Update*. Rekordy te informują o trwającej sesji i są uzupełnieniem rekordów występujących na jej końcach (*Start* i *Stop*). W przypadku, gdy rekordy *Interim-Update* są wysyłane z dużą częstotliwością można, po krótkim czasie, poprawić wcześniejsze, błędne wyniki obliczeń. Przykładowo przyjmijmy, że rekordy *Interim-Update* są przesyłane w 20 minutowym interwale aktualizując dane o każdej sesji. W przypadku utraty rekordu *Stop*, jeśli po 20 minutach od ostatniego rekordu *Interim-Update* nie następuje wysłanie kolejnego rekordu *Interim-Update* należy domniemać, że wystąpiło zdarzenie typu *Stop* (ale rekord o nim został utracony) i uznać sesję za rozłączoną. Analogicznie sytuacja przedstawia się dla utraty rekordu *Start*. Wtedy występuje sytuacja, w której system po 20 minutach otrzyma pierwszy rekord *Interim-Update* z nawiązanej sesji i powinien uznać ją za aktywną, mimo braku rekordu *Start*. Dane *Interim-Update* pozwalają zatem znacznie lepiej oceniać aktualny stan sesji, z tym że kluczowy jest tu interwał ich przesyłania, ponieważ dopiero po upływie interwału „naprawiane" są rzeczywiste stany sesji w systemie. Warto nadmienić, że dane *Interim-Update* nie mają znaczenia w zakresie poprawy jakości obliczania parametru *SP* pod kątem utraty ruchu typu *Stop* w ramach grupowego zerwania.

Straty danych *Accounting* mogą występować również z powodu specyficznego zachowania samych urządzeń *BRAS / BNG*. Dane *Accounting*, mimo odbioru przez system monitorujący z systemu *RADIUS* są pierwotnie inicjowane przez urządzenia *BRAS / BNG*, które generują je w momencie zestawiania sesji (*Start*), przesyłania informacji o podtrzymaniu sesji (*Interim-Update*) oraz w momencie zakończenia sesji (*Stop*). W przypadku usterki urządzenia *BRAS / BNG* sesje użytkowników są w rzeczywistości zrywane, ale w strumieniu danych *RADIUS Accounting* będzie brakować rekordów typu *Stop*, ponieważ uszkodzony *BRAS / BNG* ich nie wysyła. Dopiero z uwagi na brak następnych rekordów *Interim-Update* można wnioskować, że sesje zostały zerwane. Prowadzi to do wniosku, że algorytm *AWA* może być stosowany do monitorowania urządzeń *BRAS / BNG* (korzeń drzewa sieci, por. Rys. 2.10) jedynie

w ograniczonym zakresie. Skutecznie monitorowane mogą być wyłącznie zdarzenia, które nie prowadzą do strat danych *Accounting* (np. uszkodzenia portów, światłowodów lub pojedynczych kart). Sytuacje, w których urządzenie *BRAS / BNG* nie zapewnia pewności przesyłania danych *Accounting* są obarczone błędem grubym i ich monitorowanie należy wykluczyć.

3.7. DZIAŁANIA PROAKTYWNE W TRAKCIE AWARII

Algorytm *AWA* może zostać wzbogacony o dodatkowe działania proaktywne celem przywrócenia działania usług. Wystąpienie awarii prowadzi do niedostępności usług w jej okresie. W tym czasie urządzenia klienckie próbują ponownie nawiązać połączenie. Jeśli próby są nieskuteczne, okres pomiędzy kolejnymi próbami jest zwiększany, celem ograniczenia negatywnego wpływu na sieć spowodowanego dużą liczbą urządzeń próbujących nawiązać połączenie. W trakcie tych prób urządzenia mogą ulec zawieszeniu, co oznacza, że po usunięciu awarii nie uzyskają one dostępu do usług w sposób automatyczny. Mogą występować sytuacje, w których klienci w trakcie trwania awarii sami wyłączyli sprzęt (bo i tak mieli to zrobić, np. na noc). Występują też sytuacje, w których klienci nie wiedząc o awarii, próbując samodzielnie przywrócić usługę dokonują zmian uniemożliwiających jej dalsze działanie.



Rys. 3.4: Algorytm *AWA* rozbudowany o operację restartu usług

Niektóre z opisanych sytuacji mogą być rozwiązane za pomocą zdalnego restartu urządzenia. Rys. 3.4 przedstawia wersję *AWA* rozbudowaną o mechanizm restartu wykonywany zakończeniem *E4*. Restart może dotyczyć wielu elementów sieci, poczynając od *CPE*, poprzez porty urządzeń sieciowych, karty, aż do całych urządzeń sieciowych. Nie zawsze występuje możliwość wykonania restartów dla całej ścieżki awarii, ponieważ ze względu na występujące problemy może występować brak możliwości nawiązania łączności z zasobem do restartu lub restart spowodowałby zaburzenie działania innych usług. Przykładowo bardzo mało prawdopodobna jest możliwość restartu urządzeń *CPE,* ponieważ jeśli nie działają usługi nie ma dostępu zdalnego do tych urządzeń.

Port centralowy, który wykorzystywany jest do świadczenia usług ma możliwość dostępu zdalnego i może być zdalnie zrestartowany, co w niektórych przypadkach może przywrócić działanie usług. Może on być jednak wykorzystywany przez więcej niż jedno *CPE* (tak jest w sieci *GPON*) co czyni operację restartu problematyczną. Jednakże w przypadku technologii *DSL* relacja port – *CPE* jest jeden do jednego, więc w tym przypadku restart może być wykonany bezpiecznie. W zakresie proaktywnych akcji należy rozważyć całą ścieżkę świadczenia usługi, jednak im dalej od *CPE* w stronę *BRAS / BNG* (Rys. 2.10) tym urządzenia obsługują w sposób uniwersalny coraz więcej usług, a to oznacza że możliwości restartów są coraz mniejsze (w sytuacji, w której tylko część usług nie wróciła po awarii). Chyba, że awaria dotyczy wszystkich usług świadczonych na danym zasobie, wtedy mimo dużej liczby realizowanych usług – restart nie będzie miał negatywnego wpływu. Jednym z przykładów, poza portem centralowym, jest jeszcze ścieżka wirtualna w sieci *ATM* (*VP* – ang. *Virtual Path*). W sieci *ATM* często konfiguruje się dedykowane ścieżki dla każdego z abonentów, dzięki czemu możliwe jest również restartowanie tych ścieżek. Taka operacja jest operacją logiczną, ponieważ ścieżki te nie są tworami fizycznymi.

W sytuacji, w której wciąż występuje awaria i żadne usługi nie wróciły do pracy, akcje proaktywne mogą być prowadzone na większą skalę, ponieważ jeśli nie ma działających usług nie ma też ryzyka przerwania ich ciągłości.

3.8. FLUKTUACJE ZASOBÓW SIECIOWYCH

W rozdziałach 3.2, 3.3 i 3.4 omówione zostały pojedyncze zdarzenia grupowych zerwań sesji. Są jednak sytuacje, w których występują pojedyncze *GZS*, ale wszystkie usługi bardzo szybko wracają do pracy. Zgodnie z zasadami pracy algorytmu *AWA* (por. Rys. 3.1) w takiej sytuacji nie zostanie aktywowany alarm informujący o problemie, a algorytm skończy działanie wyjściem *E1* lub *E2*. Sytuacja taka została do tej pory określona jako nieistotna dla operatora, ponieważ wynika z niezdefiniowanych krótkich interwencji w sieć (czy to sił wyższych, czy ludzkich). Cykliczne powtarzanie się takiej sytuacji określa się jako fluktuacja zasobu sieciowego, a jej znaczenie jest już znacznie większe. Zaprojektowany algorytm *AWA* pomija jednak takie sytuacje. Celem rozwiązania tego problemu zaprojektowano algorytm nakładkowy, monitorujący liczbę wystąpień *GZS* na tym samym zasobie w zdefiniowanym okresie czasu (parametr ten określa się jako liczba fluktuacji), niezależnie od tego, czy te zdarzenia powodowały wyzwolenie alarmu, czy też nie.

Wykonuje się cykliczną analizę wszystkich historycznych zdarzeń *GZS* w przedziale czasu o szerokości *T_HIST*, w ramach której zlicza się liczbę wykrytych zdarzeń na tym samym

zasobie (*C*), niezależnie od tego w jaki sposób zakończyła się ich analiza. Jeśli liczba zdarzeń przekracza próg *C_MIN_HIST* aktywowany jest inny rodzaj alarmu – alarm fluktuacji zasobu sieciowego. Ze względu na inną charakterystykę tego typu alarmu, nie jest on dezaktywowany natychmiastowo po usunięciu problemu (jak w przypadku grupowego zerwania sesji), ponieważ nie ma jasnego czynnika (w postaci usług wracających do pracy) świadczącego o jego usunięciu. Alarm taki podlega dezaktywacji ręcznej lub znika samoczynnie, gdy parametr *C* spadnie poniżej progu *C_MIN_HIST*, ale tutaj musi upłynąć czas rzędu *T_HIST*, aby wartość parametru *C* spadała znacząco.

W ramach definiowania fluktuacji zasobów zdefiniowano następujące parametry:

- *T_HIST* – okres historyczny analizy fluktuacji zasobów sieciowych,
- *C* – liczba wykrytych grupowych zerwań sesji w okresie *T_HIST*,
- *C_MIN_HIST* – minimalna liczba *C* aktywująca alarm.

Fluktuacje zasobów sieciowych zdarzają się relatywnie rzadko, ale wprowadzenie obsługi fluktuacji jest konieczne, ponieważ jak wskazano w tym rozdziale – bez niego nakreślone problemy pozostaną przez system monitorowania niezauważone.

4. TESTOWANIE I WDROŻENIE ALGORYTMU AWA

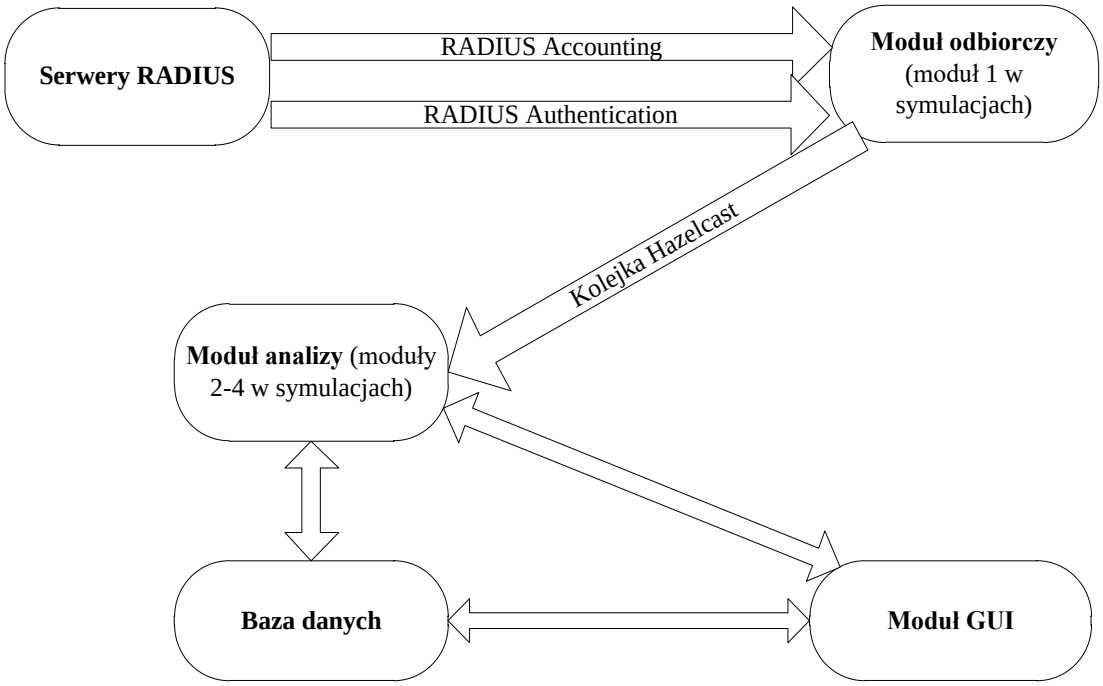
W tym rozdziale przedstawione zostały wyniki działania algorytmu *AWA* dla kart urządzeń *DSLAM* oraz kabli miedzianych w sieci dostępowej *DSL* zapewniającej usługi dla około 2 milionów klientów. Urządzenia *CPE* zestawiały sesję *PPP* z urządzeniami *BRAS / BNG* (Rys. 2.10). Do uwierzytelniania wykorzystywane były systemy *RADIUS*, przekazujące do systemu monitorowania dane *RADIUS Accounting* i *RADIUS Authentication*.

System *AWA* został wdrożony w sieci operatora *Orange Polska S.A.* w zakresie jego sieci dostępowej i agregacyjnej. W następnych podrozdziałach przedstawione zostały wyniki działania algorytmu z okresu 4 tygodni (od 30.12.2019 do 26.01.2020). Część sieci, której dotyczyła analiza, wykorzystywała parę miedzianą do transmisji na pętli abonenckiej, gdzie wykonywana była transmisja w standardach *ADSL/2/2+* lub *VDSL2*. Uruchomiony system analizował grupowe zerwania sesji *PPP* w kontekście kart urządzeń *DSLAM* (element aktywny) oraz kabli miedzianych (element pasywny).

Wolumen zdarzeń odbieranych z sieci w czasie rzeczywistym wynosił maksymalnie 5 tys. zdarzeń *RADIUS Accounting* na sekundę. Wdrożona aplikacja została napisana w technologii *Java* (uruchomiona na *Zulu JDK 1.8u212*) i dostosowana do pracy na serwerze aplikacyjnym *Payara Server* (*5.194*) pracującym pod kontrolą systemu operacyjnego *Debian GNU/Linux 7*. Jako bazę danych wykorzystano niekomercyjną bazę *MySQL* w wersji *8.0.19*.

4.1. DANE TECHNICZNE ALGORYTMU AWA

Wdrożone rozwiązanie składa się z czterech elementów przedstawionych na Rys. 4.1. Pierwszym z nich (moduł nr 1) jest moduł odbioru zdarzeń sieciowych (w tym przypadku *RADIUS Accounting*). Jest on odpowiedzialny za odbiór, analizę oraz wysłanie danych do modułu analizy. Obserwacje pokazały, że w przypadku sieci obejmującej około 2 mln usług i generującej zdarzenia typu *Interim-Update* co 20 minut, obserwowany średni ruch w module odbiorczym wynosi 1500 zdarzeń na sekundę, a w szczycie osiąga 5000 zdarzeń na sekundę. Wolumetria ta wynika głównie ze zdarzeń *Interim-Update*, które muszą być generowane co 20 minut dla każdej trwającej sesji użytkownika.



Rys. 4.1: Schemat logiczny systemu monitorowania

Obserwacje wielkości ruchu można potwierdzić poprzez przeprowadzenie symulacji. Średnią częstotliwość napływu zdarzeń *Y* można zdefiniować wzorem:

$$Y = \mathrm{A} \cdot \mathrm{X} \cdot \mathrm{f}\left[\frac{1}{s}\right] \tag{8}$$

gdzie:

Y – średnia częstotliwość odbieranego ruchu (zdarzenia na sekundę),

A – procent aktywnych usług w sieci (0 < A <= 1),

X – liczba usług w sieci,

f – częstotliwość generowania zdarzenia przez jedną aktywną usługę.

W przypadku omawianej sieci A = 0,9; X = 2 mln. Z kolei częstotliwość to 1/1200 Hz (jedno zdarzenie raz na 20 minut). Obliczenia te dają w wyniku 1500 zdarzeń na sekundę.

Dochodzą do tego dodatkowo zdarzenia typu *Start* i *Stop* związane z rozpoczynaniem oraz kończeniem pracy przez poszczególne sesje, ale te zdarzenia występują znacznie rzadziej. Rzeczywisty ruch potwierdza wykonane obliczenia, ponieważ 10 minutowa obserwacja przy pomocy programu *tcpdump* [37] dała w wyniku średni ruch *RADIUS Accounting* równy 1517 zdarzeń na sekundę.

Pełna analiza danych we wdrożonym rozwiązaniu wykonywana jest w pamięci operacyjnej *RAM*. Ze względów wydajnościowych baza danych nie byłaby w stanie obsłużyć analizy tak dużej liczby rekordów na sekundę.

Przyjęto kroczące okno analizy ze skokiem 15 sekund, co oznacza, że raz na 15 sekund aktywowana jest analiza danych pod kątem wystąpienia *GZS*. W okresie pomiędzy aktywacjami okna dane są gromadzone w buforze. Ze względu na to, że okno grupowego zerwania wynosi 3 minuty (parametr *T*), krok 15 sekund nie wprowadza znaczącego opóźnienia w wykryciu awarii. Umożliwia on jednak uruchamianie analizy w wybranych momentach czasu, a także rozdzielenie zasobów wykonujących operacje odbioru danych *RADIUS Accounting* od zasobów wykonujących analizę oraz znaczne ograniczenie liczby wykonywanych analiz bez znacznego pogorszenia jakości działania rozwiązania (występuje maksymalnie 15 sekundowe opóźnienie w pracy algorytmu). Dodatkowo zmienność sieci jest na takim poziomie, że nie ma potrzeby częstszego wykonywania analizy, ponieważ sesja *PPP* może być zerwana do 2 minut od zaprzestania aktywności *CPE*. Wiadomo, że różne sesje zostaną zerwane w różnych momentach czasu, jednak pewność, iż będą to wszystkie mamy w okresie 3 minut (2 minuty + minuta zapasu). Okno kroczące co 15 sekund umożliwia wykrycie pełnego dopełnienia okna zerwania przy optymalnym zużyciu posiadanych zasobów mocy obliczeniowej.

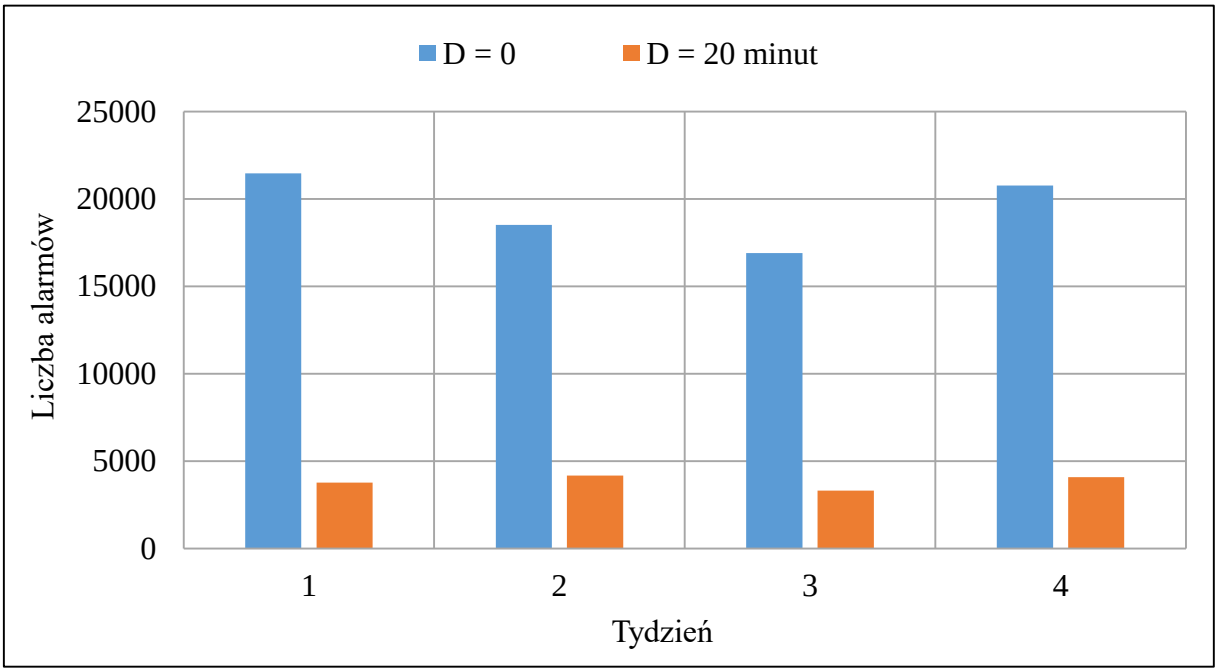
Ważną cechą wdrożonego rozwiązania jest przypisywanie każdego zdarzenia *RADIUS Accounting* do konkretnego zasobu sieciowego w momencie jego wystąpienia. Dzięki temu uzyskuje się spłaszczenie zużycia zasobów mocy obliczeniowej. W takim rozwiązaniu w pamięci *RAM* buforowana jest pełna mapa zdarzeń *RADIUS Accounting* na sieci z okresu ostatnich 3 minut, gdzie zdarzenia te są już przypisane do zasobów sieciowych, na których wystąpiły. W momencie uruchomienia analizy (raz na 15 sekund) algorytm ma już gotowe dane. Jedyne co pozostaje do zaaplikowania, to wykonanie odpowiednich obliczeń na zasobach sieciowych, celem oceny czy przekroczone zostały progi *GZS* (*S*>=*SMIN*). Analogiczne podejście zostało przyjęte w symulacjach wydajności algorytmu.

Ponadto dla kart *DSLAM* przyjęto, że w przypadku urządzeń z *uplink* (łącze w stronę urządzenia *BRAS / BNG*) typu *Ethernet* (tzw. *IPDSLAM*) minimalna liczba zerwań na karcie powodująca przejście do dalszego progu analizy wynosi 10 (*SMIN*=10), a dla urządzeń mniej obsadzonych (*ATM*) jest to 6 (*SMIN*=6). Przyjęte wartości progu *SMIN* są dość wysokie w stosunku do wielkości monitorowanego zasobu, co przekłada się na wadę w postaci braku wykrywania awarii w przypadku niewielkiego wypełnienia zasobu sieciowego aktywnymi usługami. Progi mogą zostać zmniejszone, jednak zwiększa to zapotrzebowanie algorytmu na

moc obliczeniową, ponieważ więcej zdarzeń *RADIUS Accounting* jest branych do analizy. Opisane progi zostały wypracowane w praktyce wraz z ekspertami od utrzymania sieci dostępowej jako najlepszy kompromis między wydajnością implementacji algorytmu *AWA*, a jego skutecznością biznesową dla danego konkretnego wdrożenia.

4.2. OPÓŹNIENIE PRZED AKTYWACJĄ ALARMU

Jednym z parametrów zaproponowanego algorytmu (Rys. 3.1) jest opóźnienie w generacji alarmu, którego czas trwania określa parametr *D*. W ramach wdrożenia porównano dwa statyczne progi *D*=0 i *D*=20 minut. Wyniki liczby generowanych alarmów dla tych dwóch progów zostały przedstawione na Rys. 4.2.



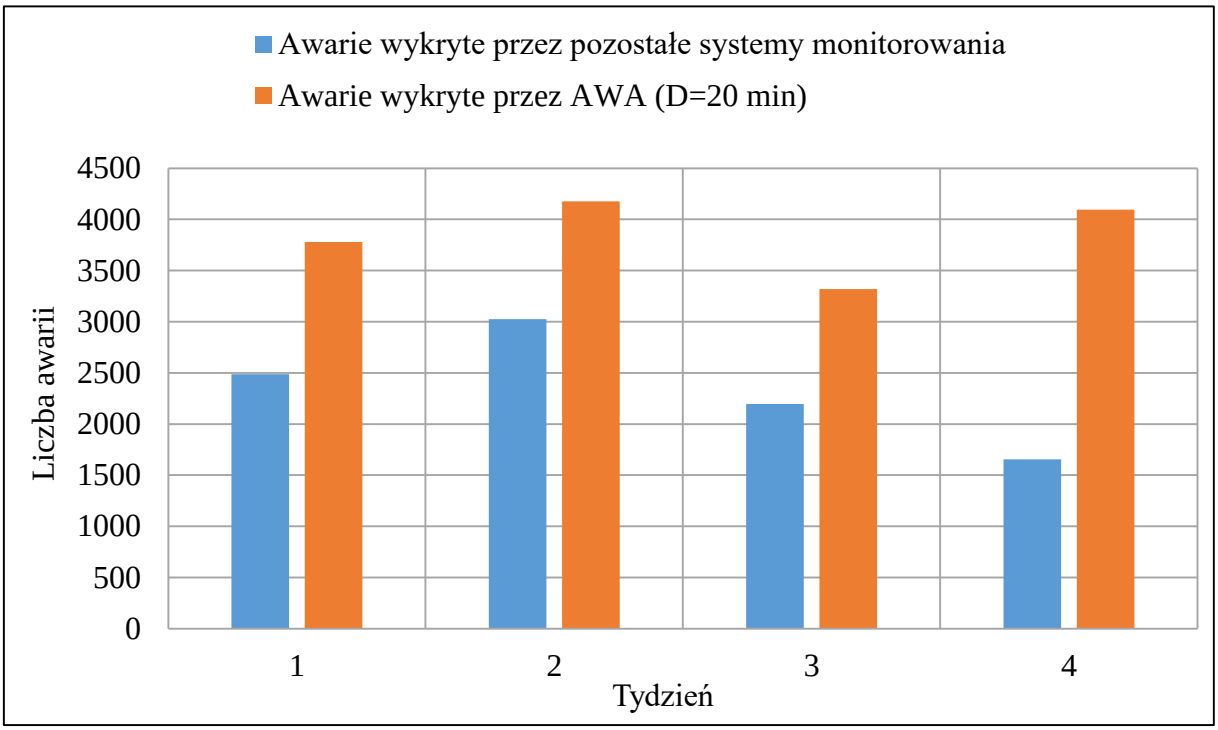
Rys. 4.2: Liczba generowanych alarmów dla *D* = 0 i *D* = 20 minut

W symulacjach zastosowano następujące ustawienia algorytmu: 0 lub 20 minutowe opóźnienie decyzji o aktywacji alarmu (parametr *D*), 3 minutowe okno grupowego zerwania sesji *PPP* (parametr *T*), 80-procentowy próg zerwanych sesji *PPP* na monitorowanym zasobie sieciowym (parametr *SPMIN*) oraz co najmniej 10% sesji (parametr *RMIN*), których nawiązanie ponowne po czasie *D* od wykrycia zdarzenia *GZS* powoduje automatyczne zakończenie analizy (*E1*). Dane z działającej sieci potwierdziły, że zastosowanie opóźnienia w generowaniu alarmu pozwala zmniejszyć liczbę alarmów o 82%. Wyeliminowane zdarzenia *GZS* są nieistotne dla operatora, ponieważ zanim odpowiednie służby zajęłyby się nimi – problem rozwiązałaby się sam. Samoczynnie znikające problemy nie mają dla operatora dużej wartości biznesowej

(problem „sam się naprawił”; jego przyczyną była prawdopodobnie krótka awaria zasilania, restart urządzeń, prace prowadzone na sieci). Inaczej jest, gdy tego typu zdarzenia *GZS* występują więcej niż jednokrotnie, ale do wykrywania problemów z fluktuacją zasobów sieciowych ma zastosowanie osobny algorytm (por. 3.8).

4.3. PORÓWNANIE Z INNYMI SYSTEMAMI WYKRYWANIA AWARII

Rys. 4.3 pokazuje porównanie liczby awarii wykrytych przez algorytm *AWA* w stosunku do pozostałych systemów monitorowania sieci kooperującego operatora. Systemy monitorujące, do których zostało wykonane porównanie, realizowały monitoring w sposób opisany w rozdziale 2.8. Bazowały one na weryfikacji stanu operacyjnego portów, kart oraz dostępności urządzeń sieciowych. Weryfikacja ta opierała się na cyklicznym skanowaniu sieci operatora, ale także na odbiorze zdarzeń asynchronicznych generowanych bezpośrednio przez urządzenia w sieci. Rejestracją tych zdarzeń zajmowały się systemy *NMS*. Ze względu na wielkość sieci operatora oraz liczbę różnych typów urządzeń alarmy były unifikowane za pomocą systemu kategorii *UNEMS* (por. 2.7), a następnie raportowane do systemów obsługiwanych przez jednostki operacyjne. Baza danych tych systemów, zawierająca informacje o wszystkich awariach wykrytych w sieci operatora była podstawą do porównania z algorytmem *AWA*, które to rozwiązanie zostało uruchomione testowo – obok istniejących już systemów monitorowania.

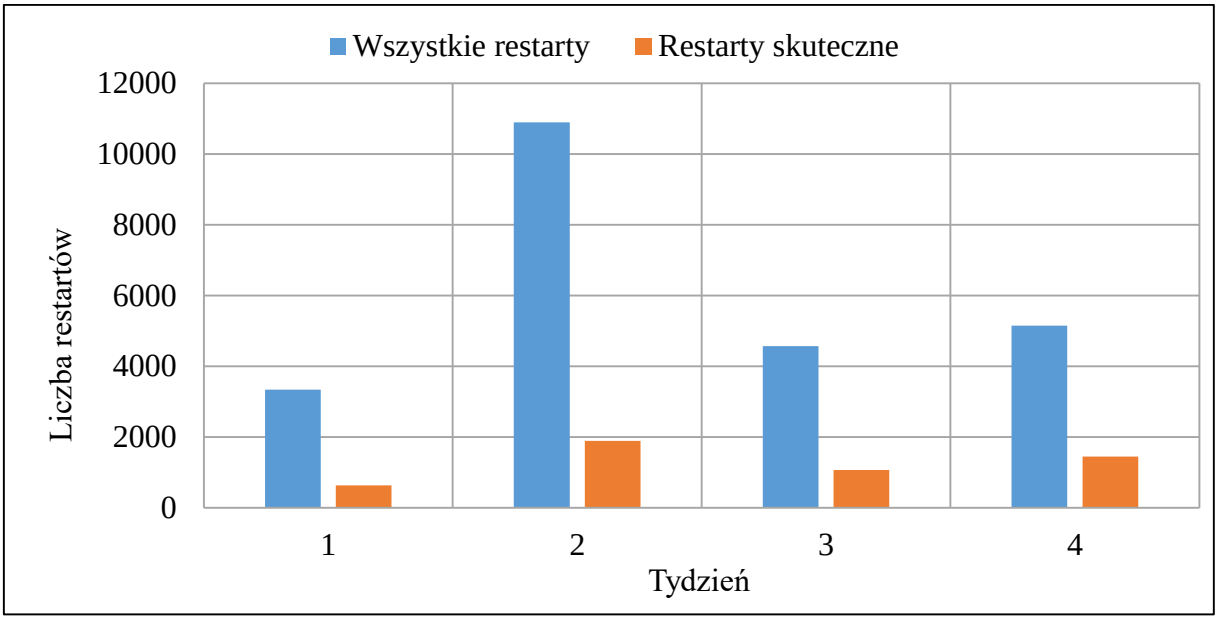


Rys. 4.3: Porównanie liczby awarii wykrytych przez *AWA* w stosunku do pozostałych systemów monitorowania

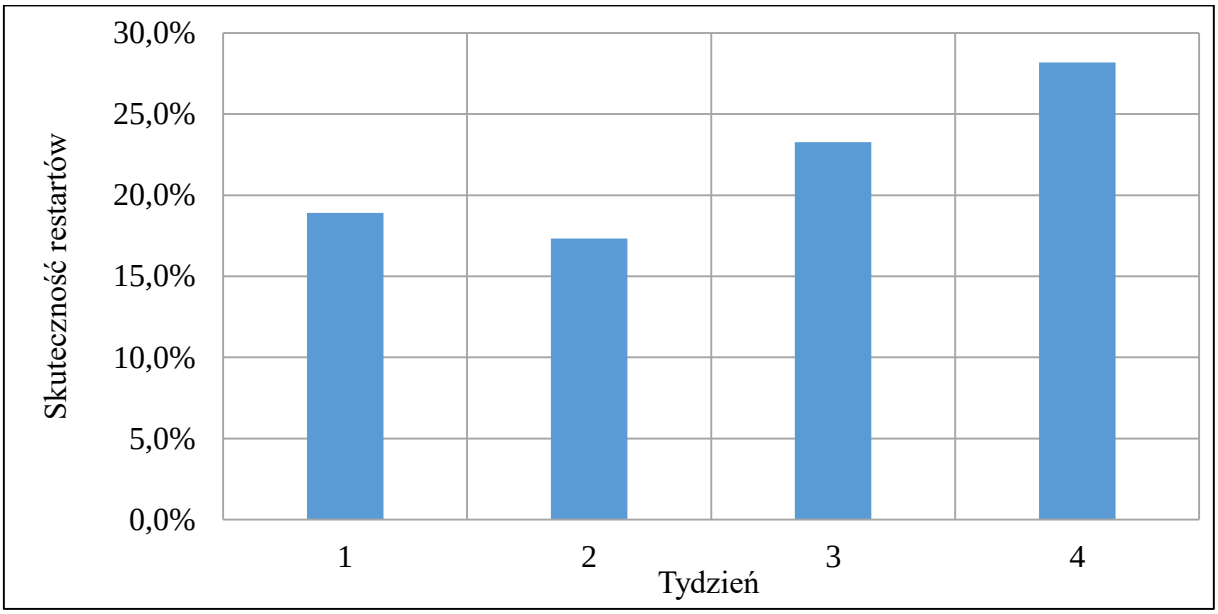
Zdarzenia z pozostałych systemów były kolekcjonowane w analogicznym okresie czasu, w jakim testowany był zaproponowany algorytm. W zależności od okresu implementacja algorytmu *AWA* wykrywała od 38% do 148% więcej awarii. Różnice między poszczególnymi okresami wynikają z różnych przyczyn awarii w różnych momentach czasowych. Zastosowane u operatora systemy (jak każde rozwiązanie) mają lepszą skuteczność w wykrywaniu pewnych typów awarii, a gorszą dla innych. W zależności od tego jak rozłożyły się w czasie przyczyny awarii uzyskane zostały inne wyniki porównania sposobów monitorowania sieci. Z tego powodu całkowitą skuteczność zaproponowanego algorytmu należy zbadać w odpowiednio długim okresie czasu, dzięki czemu jest pewność, że wystąpiły wszystkie typy zdarzeń, a co za tym idzie algorytmy zostały zweryfikowane pod każdym kątem. W przypadku zbyt krótkiego okresu czasu można dokonać nieprawidłowej oceny skuteczności zastosowanych metod.

4.4. DZIAŁANIA PROAKTYWNE

W ramach implementacji *AWA* wdrożonej dla kart urządzeń *DSLAM* zostały aktywowane akcje proaktywne na portach tychże kart. Wykryte zdarzenie *GZS* dotyczy określonej liczby portów na karcie. W ramach obsługi zdarzenia nadchodzi moment, w którym awaria jest usuwana i usługi wracają do pracy. W niektórych sytuacjach nie wszystkie usługi powracają do pracy samoczynnie po usunięciu awarii. Przyczyną takiego stanu rzeczy może być kilka czynników, które zostały opisane w rozdziale 3.7. Gdy usługa nie wraca do pracy, nie ma możliwości nawiązania zdalnej komunikacji ze sprzętem klienta, a tym samym zdalnego zrestartowania tegoż sprzętu. Istnieje jednak możliwość restartu portu na karcie urządzenia *DSLAM*, za pomocą którego świadczona jest usługa. Taka akcja ma na celu wykluczenie możliwości zawieszenia się portu, jak również ma na celu próbę odblokowania strony klienckiej transmisji danych (w przypadku sieci *DSL* jest to *XTU-R* [4]) poprzez tymczasowe wyłączenie i włączenie transmisji ponownie.



Rys. 4.4: Wartości bezwzględne wszystkich wykonanych restartów i restartów skutecznych



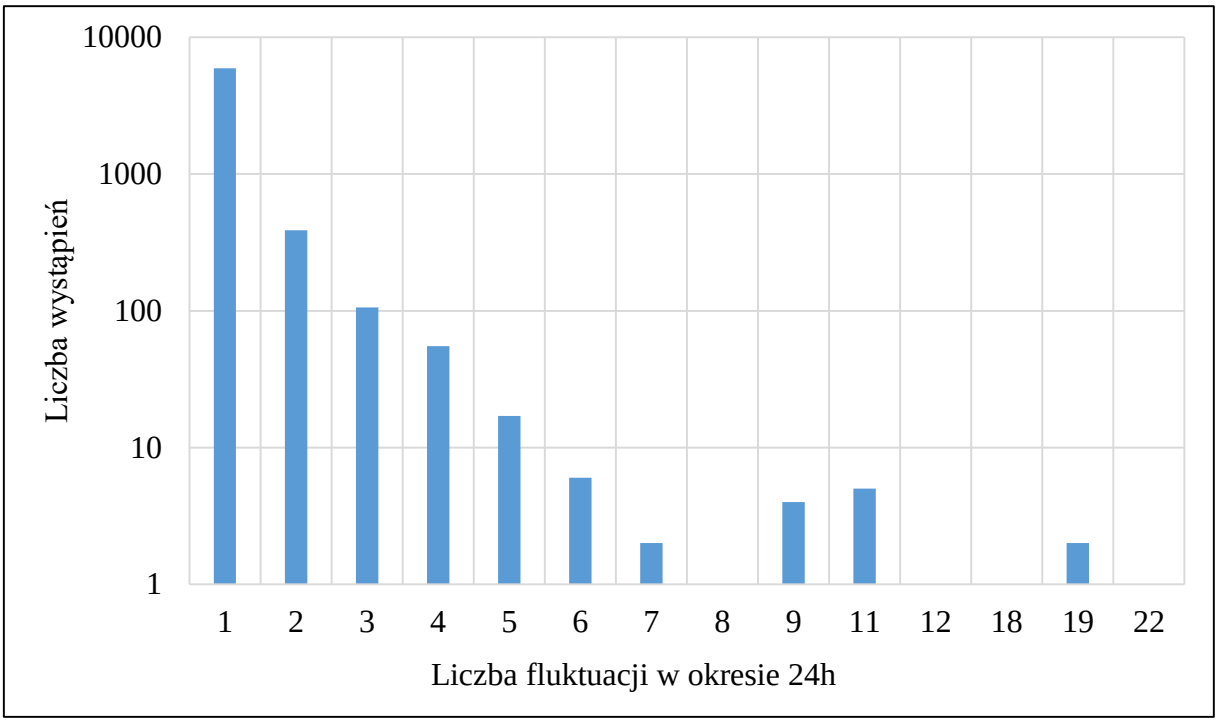
Rys. 4.5: Skuteczność restartów

Na Rys. 4.4 przedstawiono liczbę wszystkich przeprowadzonych restartów oraz tych zakończonych sukcesem. Restarty są wykonywane w momencie, w którym sytuacja po awarii została ustabilizowana, tj. wróciło do działania co najmniej 80% zerwanych sesji oraz liczba ta nie zmienia się w przeciągu 5 minut. Wtedy dla pozostałych usług wykonuje się restart portu. Za restart skuteczny uznaje się taki, po którego wykonaniu usługa wróciła do pracy. W poszczególnych tygodniach udało się przywrócić do pracy od 632 do 1889 usług. Rys. 4.5 prezentuje skuteczność restartów, która jest obliczana jako stosunek liczby restartów

skutecznych do wszystkich wykonanych restartów. Waha się ona między 17%, a 28%. Pozostałych usług nie udaje się przywrócić do pracy za pomocą restartu.

4.5. STATYSTYKI WDROŻENIA WYKRYWANIA FLUKTUACJI ZASOBÓW SIECIOWYCH

Wdrożenie *AWA* u kooperującego operatora zostało zrealizowane także dla funkcjonalności wykrywania fluktuacji kart *DSLAM* (por. rozdział 3.8). Parametr *T_HIST* został ustawiony na 24 godziny, a *C_MIN_HIST* na 5.



Rys. 4.6: Rozkład fluktuacji kart DSLAM

Na Rys. 4.6 przedstawiono rozkład liczby zdarzeń fluktuacji zasobów na monitorowanych zasobach sieciowych w funkcji liczby powtórzeń w okresie *T_HIST*=24h. Dane są pokazane za wspomniane 4 tygodnie czasu dla funkcjonalności monitorowania kart urządzeń *DSLAM*. Warto zwrócić uwagę na skalę logarytmiczną (o podstawie 10) na osi Y. Najwięcej było pojedynczych zdarzeń (5948), a następnie każda kolejna powtarzalność obejmuje coraz mniejszą liczbę zasobów sieciowych. Dla *C_MIN_HIST*=5 sumaryczna liczba zdarzeń do przeanalizowania w obrębie miesiąca wynosi 40.

Według opinii kooperującego operatora wykrywanie zdarzeń fluktuacji zasobów jest istotne, co wynika z faktu, że w tym aspekcie nie ma żadnego konkurującego systemu wykrywania tego typu zdarzeń. Inaczej jest w zakresie wcześniej omówionych funkcjonalności,

gdzie *AWA* uzupełnia inne systemy w zakresie wykrywania zdarzeń typu pojedyncza awaria. Dodatkowo, wdrożenie pokazało, że fluktuacje występują stosunkowo na niewielkiej liczbie zasobów sieciowych w stosunku do innych typów zdarzeń. Ułatwia to ich obsługę, ponieważ można ją zrealizować niewielkimi zasobami ludzkimi. Awarie typu fluktuacja zasobu sieciowego mają wyższy priorytet obsługi, niż wcześniej opisane pojedyncze zdarzenia, ze względu na ich przewlekłość (ze swojej definicji występują przez długi okres czasu) oraz zatajenie (do momentu wykrycia były odczuwalne przez klientów, ale operator nic o nich nie wiedział).

4.6. GRAFICZNE ELEMENTY WDROŻONEGO ROZWIĄZANIA

W tym rozdziale przedstawiono fragmenty graficznego interfejsu użytkownika (*GUI* – ang. *Graphical User Interface*) wdrożonego rozwiązania *AWA*.

Pierwsze zerwanie	Zerwania	Zerwania (%)	Dslam IP	Nazwa DSLAM	Typ DSLAM	M1400 DSLAM	Baza	Slot DSLAM	Pierwszy powrót	Powroty	Czy w analizie	Status DSLAM	Stan operacyjny portów	Cały DSLAM?
2020-04-16 15:31:10	**32**	**100**	**10.**	**962_**	**S**	**Pl**	**PC**	**1**		**0**	**true**	**TIMED_OUT**		**true**
2020-04-16 15:28:23	22	100	10	BS_I	IS	M	BI.	LT1		0	true	OK	DOWN	false
2020-04-16 14:21:43	37	100	10	Osie	S	O	SZ	1		0	true	TIMED_OUT		true
2020-04-16 14:21:22	13	100	10	SZ_\	IS	W	SZ	LT2		0	true	TIMED_OUT		false
2020-04-16 14:21:17	18	106	10	SZ_\	IS	W	SZ	LT3		0	true	TIMED_OUT		false
2020-04-16 14:21:16	19	100	10	SZ_\	IS	W	SZ	LT5		0	true	TIMED_OUT		false
2020-04-16 14:21:16	22	100	10	SZ_\	IS	W	SZ	LT1		0	true	TIMED_OUT		false
2020-04-16 14:21:16	35	100	10	SZ_I	IS	S\	SZ	LT5		0	true	TIMED_OUT		false
2020-04-16 14:21:16	11	100	10	SZ_I	IS	S\	SZ	LT1		0	true	TIMED_OUT		false
2020-04-16 14:21:15	38	100	10	SZ_\	IS	W	SZ	LT4		0	true	TIMED_OUT		false
2020-04-16 14:21:15	45	100	10	SZ_I	IS	S\	SZ	LT3		0	true	TIMED_OUT		false
2020-04-16 14:17:46	**21**	**100**	**10.**	**IPDS**	**IS**	**Pl**	**BI.**	**LT5**	**2020-04-16 14:29:00**	**19**	**false**	**OK**	**DOWN**	**false**
2020-04-16 14:14:54	**20**	**87**	**10.**	**BS_I**	**IS**	**B(**	**BI.**	**LT5**	**2020-04-16 14:14:57**	**20**	**false**	**OK**	**DOWN**	**false**
2020-04-16 13:22:17	**19**	**95**	**10.**	**LUB_**	**S**	**DI**	**LL**	**2**	**2020-04-16 13:24:11**	**17**	**false**	**OK**	**DOWN**	**false**
2020-04-16 13:21:49	**53**	**100**	**10.**	**LUB_**	**S**	**DI**	**LL**	**1**	**2020-04-16 14:19:58**	**52**	**false**	**OK**	**DOWN**	**false**
2020-04-16 13:20:42	55	115	10	LB_I	M	H.	LL	3	2020-04-16 14:15:16	55	false	OK	DOWN	false
2020-04-16 13:14:18	**18**	**138**	**10.**	**WAL**	**M**	**B(**	**JE**	**3**	**2020-04-16 13:25:55**	**18**	**false**	**OK**	**DOWN**	**false**
2020-04-16 13:04:39	21	91	10	Bs_S	S M	S:	BI.	1	2020-04-16 13:41:24	21	false	OK	DOWN	true
2020-04-16 12:51:15	37	100	10	IP_R	IS	G	RA	LT7	2020-04-16 13:24:39	36	false	OK	DOWN	false

Rys. 4.7: Zanonimizowany wycinek *GUI AWA* dla funkcji wykrywania awarii dla kart *DSLAM*

Rys. 4.7 przedstawia główny widok, na którym prezentowana jest lista aktualnie trwających zdarzeń dla funkcji detekcji awarii kart *DSLAM*. Dane wrażliwe dla kooperującego operatora zostały usunięte. Pierwsza kolumna pokazuje czas pierwszej zerwanej usługi, następna prezentuje liczbę zerwań (parametr *S*), kolejna stosunek liczby zerwań do liczby wcześniej aktywnych usług (parametr *SP*). W jednym przypadku wartość parametru *SP* przekracza 100%, co wynika ze strat danych *RADIUS Accounting* (por. rozdział 3.6). Następne kolumny prezentują dane ewidencyjne zasobu sieciowego, na którym wykryto awarię. Kolumna *„Pierwszy powrót”* prezentuje czas powrotu pierwszej sesji, a kolumna *„Powroty”* liczbę powrotów (sesji nawiązanych ponownie; parametr *R*). Kolumna *„Czy w analizie”* informuje o tym, czy dany rekord jest wciąż w analizie przez algorytm (rekordy po jej zakończeniu są wciąż przez pewien czas prezentowane na graficznym interfejsie użytkownika, stąd ta informacja jest istotna).

Kolumna o nazwie *„status DSLAM”* prezentuje dane pobrane z urządzenia, ponieważ wdrożenie *AWA* zostało wykonane z dodatkowym krokiem pobierania informacji o zasobie sieciowym (3.2). W omawianej kolumnie przekazuje się informację o tym, czy dane urządzenie jest osiągalne sieciowo (*OK* oznacza, że tak; *TIMED_OUT* oznacza, że nie).

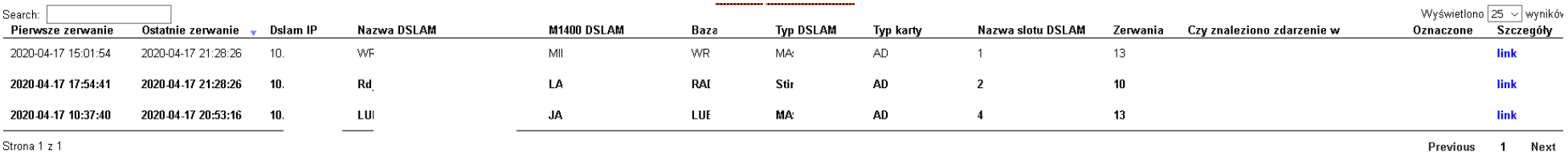
Obserwowane na Rys. 4.7 zdarzenie to grupowe zerwanie sesji (*GZS*) na określonej liczbie portów karty *DSLAM*. Celem oceny danego zdarzenia prezentuje się informacje o samych portach. Pierwszy wiersz pokazuje zerwanie 32 usług, co oznacza, że wystąpiło ono na takiej liczbie portów. Każdy z portów może mieć inny status operacyjny, co jest szczegółowo wyświetlone po wejściu w dalszą zakładkę, a na widoku głównym (kolumna *„status operacyjny portów”*) prezentowana jest informacja statystyczna o stanie większości portów. Kolumna *„cały DSLAM”* informuje o tym, czy algorytm *AWA* na wyższym stopniu monitorowania wykrył awarię całego urządzenia *DSLAM* (jest to realizacja porównania stopni algorytmu, opisana w rozdziale 3.3).

Oprogramowanie realizujące funkcjonalność *AWA* pozwala także na wyróżnienie informacji o efektywności wykrywania awarii. Pogrubiona czcionka w danym wierszu tabeli oznacza, że zdarzenie jest jeszcze nieobsłużone i niewykryte przez inne systemy monitorowania. Natomiast nazwy o niepogrubionej czcionce są już wykryte przez inne systemy i obsługiwane przez jednostki operacyjne operatora telekomunikacyjnego.

Pierwsze zerwanie	Zerwania	Zerwania (%)	Aktywne wcz.	Typ kabla	Nazwa kabla	Początek kabla typ	Początek kabla nazwa	Koniec kabla typ	Koniec kabla nazwa	Baza	Powroty	Powroty (%)	W analizie?	W kartach DSLAM?	Oznaczone	Szczegóły
2020-04-17 13:51:30	14	70	20	kabel magistralny	MAN	PG	MAN	SZAFKA	MAN	PO	4	28	true	false		link
2020-04-17 13:47:30	8	100	8	kabel rozdzielczy	YB9\	SZAFKA	YB9	PD	YB9\	RZ	0	0	true	true		link
2020-04-17 13:47:30	21	105	20	kabel magistralny	YB9\	PG	RAD	SZAFKA	YB9\	RZ	0	0	true	true		link
2020-04-17 13:47:28	7	100	7	kabel rozdzielczy	YB9\	SZAFKA	YB9	PD	YB9\	RZ	0	0	true	true		link
2020-04-17 13:47:28	9	100	9	kabel rozdzielczy	YB9\	SZAFKA	YB9	PD	YB9\	RZ	0	0	true	true		link
2020-04-17 13:47:17	21	100	21	kabel magistralny	YB9\	PG	RAD	SZAFKA	YB9\	RZ	0	0	true	true		link
2020-04-17 13:47:13	26	104	25	kabel magistralny	YB9\	PG	RAD	SZAFKA	YB9\	RZ	0	0	true	true		link
2020-04-17 13:33:57	18	64	28	kabel magistralny	LU1	PG	LU1	SZAFKA	LU2	LO	0	0	true	false		link
2020-04-17 13:33:27	6	100	6	kabel rozdzielczy	DOE	SZAFKA	DOE	PD	DOB	RA	0	0	true	false		link
2020-04-17 13:02:17	7	100	7	kabel rozdzielczy	TOR /14/(	SZAFKA	TOF	PD	TOR	BYI	0	0	true	false		link
2020-04-17 12:50:55	6	120	5	kabel rozdzielczy	ZYJA	PG	ZYJ.	PD	ZYJA	BIE	5	83	false	true		link
2020-04-17 12:50:38	7	100	7	kabel rozdzielczy	ZYJA	PG	ZYJ.	PD	ZYJA	BIE	6	85	false	true		link
2020-04-17 12:50:29	8	100	8	kabel rozdzielczy	ZYJE	PG	ZYJ	PD	ZYJE	BIE	7	87	false	false		link
2020-04-17 12:50:28	9	100	9	kabel rozdzielczy	ZYJE	PG	ZYJ	PD	ZYJE	BIE	8	88	false	true		link
2020-04-17 12:50:27	9	100	9	kabel rozdzielczy	ZYJE	SZAFKA	ZYJ	PD	ZYJE	BIE	8	88	false	false		link
2020-04-17 12:50:24	19	90	21	kabel magistralny	ZYJE	PG	ZYJ	SZAFKA	ZYJE	BIE	17	89	false	false		link
2020-04-17 12:50:24	12	100	12	kabel rozdzielczy	ZYJE	SZAFKA	ZYJ	PD	ZYJE	BIE	11	91	false	false		link
2020-04-17 12:50:24	6	120	5	kabel rozdzielczy	ZYJA	PG	ZYJ.	PD	ZYJA	BIE	6	100	false	true		link
2020-04-17 12:50:17	17	100	17	kabel rozdzielczy	ZYJA	PG	ZYJ.	PD	ZYJA	BIE	15	88	false	true		link

Rys. 4.8: Zanonimizowany wycinek *GUI AWA* dla funkcji wykrywania awarii kabli miedzianych

Rys. 4.8 prezentuje *GUI AWA* dla funkcjonalności detekcji awarii na zasobie pasywnym, jakim są kable miedziane w sieci dostępowej. Analogicznie jak poprzednio kolumna *„Pierwsze zerwanie”* prezentuje znacznik czasowy momentu utraty usług przez pierwszą linię w ramach danej awarii. Następnie pokazane są parametry *S* oraz *SP*. W kilku przypadkach (analogicznie jak na Rys. 4.7) wartości parametru *SP* przekraczają 100%, co wynika ze strat danych *RADIUS Accounting* (por. rozdział 3.6). *„Aktywne wcz.”* to liczba usług aktywnych przed zerwaniem (parametr *A*). Kolejne 7 kolumn prezentuje informacje operatora o faktycznej lokalizacji kabla zgodnie z jego ewidencją i te informacje zostały częściowo ukryte, ze względu na ich wrażliwość. Kolumna *„powroty”* prezentuje liczbę powrotów usług do działania w ramach obserwowanej awarii (parametr *R*). *„Powroty (%)”* pokazuje tę samą daną z tym, że odniesioną do liczby zerwań (parametr *RS*). Informacja czy *„W analizie”* mówi o tym, czy dane zerwanie podlega wciąż analizie. Kolumna *„w kartach DSLAM?”* prezentuje informację o tym, czy dane zerwanie zostało wykryte na wyższym stopniu monitorowania (3.3). Jest to rozwiązanie analogiczne jak opisane wcześniej dla kart *DSLAM*, gdzie kolumna ta odnosiła się do jeszcze wyższego stopnia monitorowania w hierarchii sieciowej (całego urządzenia *DSLAM*).



Search:

Wyświetlono 25 wyników

Pierwsze zerwanie	Ostatnie zerwanie	Dslam IP	Nazwa DSLAM	M1400 DSLAM	Baza	Typ DSLAM	Typ karty	Nazwa slotu DSLAM	Zerwania	Czy znaleziono zdarzenie w	Oznaczone	Szczegóły
2020-04-17 15:01:54	2020-04-17 21:28:26	10.	WF	MII	WR	MA	AD	1	13			link
2020-04-17 17:54:41	2020-04-17 21:28:26	10.	Rd	LA	RAI	Stir	AD	2	10			link
2020-04-17 10:37:40	2020-04-17 20:53:16	10.	LUI	JA	LUE	MA	AD	4	13			link

Strona 1 z 1

Previous 1 Next

Rys. 4.9: Zanonimizowany wycinek *GUI AWA* dla funkcji wykrywania fluktuacji kart *DSLAM*

Rys. 4.9 prezentuje zanonimizowany fragment *GUI AWA* dla funkcji wykrywania fluktuacji kart urządzeń *DSLAM*. W momencie zapisu zrzutu ekranu wykryte były trzy fluktuacje. Pierwsza karta rozpoczęła fluktuację (pierwsza kolumna) o godzinie 15:01:54 dnia 17.04.2020, a ostatnie zerwanie tej karty było datowane na godzinę 21:28:26 tego samego dnia (kolumna *„Ostatnie zerwanie”*). Następnie, analogicznie jak dla poprzednio przedstawionych zrzutów ekranu występują dane dotyczące rzeczywistych lokalizacji zasobów, które zostały ukryte. Kolumna *„Zerwania”* prezentuje wartość parametru *C*. Kolumna *„Czy znaleziono zdarzenie w”* informuje o tym, czy inne systemy monitorowania również wykrywają problem z daną kartą. *„Oznaczone”* jest kolumną, za pomocą której użytkownicy oznaczają dany rekord jako wzięty do analizy, a *„Szczegóły”* umożliwia przejście do szczegółów danej fluktuacji.

5. ANALIZA DZIAŁANIA ALGORYTMU

Celem przeprowadzenia analizy działania algorytmu *AWA*, w tym jego wydajności, zrealizowany został zestaw symulacji. Niezbędne było posiadanie identycznego zestawu danych wejściowych umożliwiających wielokrotne uruchamianie algorytmu *AWA*. Zostało to osiągnięte poprzez zapisanie w bazie danych wszystkich rekordów *RADIUS Accounting* z wdrożenia w sieci za okres 4 tygodni (od 30.12.2019 do 26.01.2020). Zaletą wdrożenia na rzeczywistej sieci jest możliwość weryfikacji jak system zachowuje się w realnym świecie, ale też możliwość zapisu tych danych oraz ich modyfikacji celem weryfikacji warunków brzegowych algorytmu. Dane zostały zapisane w lokalnej bazie *MySQL* i przy każdej kolejnej symulacji były z niej pobierane.

Przeprowadzone symulacje zostały podzielone na dwie grupy. Pierwsza z nich bazowała na danych z sieci i symulacje te weryfikowały zachowanie algorytmu przy zmianie parametrów konfiguracyjnych *AWA*. Celem tych testów była próba znalezienia optymalnych parametrów konfiguracyjnych, ponieważ jak wspomniano wcześniej – większość z nich była we wdrożeniu ustalona bez takich analiz (co nie oznacza, że te ustawienia były błędne – często sama wiedza ekspercka wystarcza).

Po ustaleniu optymalnych wartości parametrów wykonano drugą grupę symulacji, mającą na celu poznanie granicznych możliwości *AWA* lub zastosowanej technologii implementacji. W tym celu dane wzorcowe zapisane w bazie były zwielokrotniane w różny sposób, aby dokonać zwiększenia obciążenia systemu i obserwacji jego zachowania dla większej wolumetrii odbieranych informacji.

Część przedstawionych analiz zawiera porównanie z danymi z wdrożenia. Dane te były odnoszone do liczby awarii potwierdzonych przez operatora jako faktycznie występujące (celem oceny działania algorytmu w stosunku do rzeczywiście potwierdzonych alarmów).

5.1. OPÓŹNIENIE PRZED AKTYWACJĄ ALARMU (PARAMETR D)

Opóźnienie w generacji alarmu we wdrożeniu opisanym w rozdziale 4.6 zostało ustalone na 20 minut (por. także 3.1). Mając zebraną próbkę danych z 4 tygodni wykonano analizę histogramową liczby wykrytych zdarzeń (n_i) w funkcji przedziału czasu ich trwania (t_i), dla parametru D=0. Analiza została przedstawiona dla zakresu wykrywania grupowych zerwań sesji *PPP* na kartach urządzeń *DSLAM*. Wyniki zostały przedstawione w Tab. 5.1, gdzie pokazano wartości n_i dla t_i w różnych przedziałach czasowych (określono 21 przedziałów czasowych). Np. n_1 to liczba zdarzeń zakończonych przed upływem 5 min od ich rozpoczęcia,

n_2 to liczba zdarzeń o czasie trwania od 5 do 10 min, itd. Tab. 5.1 prezentuje także stosunek n_i do wszystkich zdarzeń, tj. parametr ns_i:

$$ns_i = \frac{n_i}{\sum_{k=1}^{k=21} n_k} \tag{9}$$

gdzie:

i – numer przedziału czasowego (od 1 do 21),

ns_i – stosunek n_i do wszystkich zdarzeń,

n_i – liczba zdarzeń z danego przedziału,

t_i – przedział czasowy czasu trwania zdarzenia rozciągający się od $t_{i,min}$ do $t_{i,max}$,

$\sum_{k=1}^{k=21} n_k$ – suma n_k ze wszystkich przedziałów,

oraz stosunek skumulowany nss_i:

$$nss_i = \frac{\sum_{j=1}^{j=i} n_j}{\sum_{k=1}^{k=21} n_k} \tag{10}$$

gdzie:

i – numer przedziału czasowego (od 1 do 21),

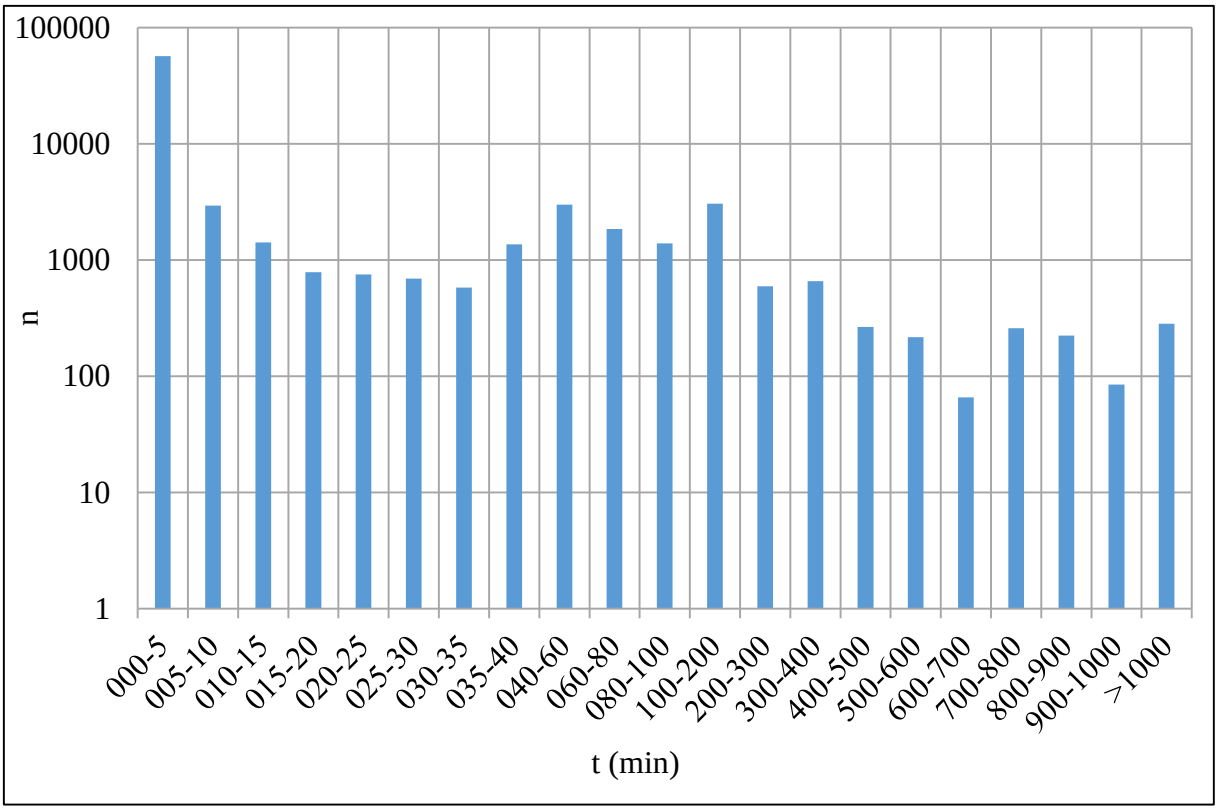
$\sum_{j=1}^{j=i} n_j$ – suma n_j z przedziałów od *1* do i (czyli liczba zdarzeń o czasie trwania krótszym niż maksymalny czas w przedziale t_i),

nss_i – stosunek powyższej sumy do liczby wszystkich zdarzeń.

Tab. 5.1: Analiza zdarzeń w przedziałach czasowych (n_i, ns_i i nss_i w funkcji t_i)

Numer przedziału (*i*)	**Przedział czasu trwania zdarzenia t_i**	**Liczba zdarzeń w przedziale (n_i)**	**Stosunek zdarzeń (ns_i)**	**Stosunek skumulowany (nss_i)**
1	000-05	57140	73,6%	73,6%
2	005-10	2938	3,8%	77,4%
3	010-15	1418	1,8%	79,2%
4	015-20	786	1,0%	80,2%
5	020-25	755	1,0%	81,2%
6	025-30	695	0,9%	82,1%
7	030-35	579	0,7%	82,8%
8	035-40	1371	1,8%	84,6%
9	040-60	3001	3,9%	88,5%
10	060-80	1852	2,4%	90,8%
11	080-100	1397	1,8%	92,6%
12	100-200	3062	3,9%	96,6%
13	200-300	597	0,8%	97,3%
14	300-400	659	0,8%	98,2%
15	400-500	266	0,3%	98,5%

16	500-600	217	0,3%	98,8%
17	600-700	66	0,1%	98,9%
18	700-800	260	0,3%	99,2%
19	800-900	224	0,3%	99,5%
20	900-1000	85	0,1%	99,6%
21	>1000	283	0,4%	100,0%



Rys. 5.1: Rozkład liczby zdarzeń (n_i) w przedziałach czasowych (t_i)

Rys. 5.1 pokazuje wartości n_i w wybranych przedziałach czasowych w skali logarytmicznej. Skala logarytmiczna została zastosowana dla uwypuklenia wartości t z zakresu powyżej 5. Można wyznaczyć, że 73,6% zdarzeń ma czas trwania do 5 minut. Między 5, a 20 minut dochodzi kolejne 6,6% zdarzeń, co daje sumarycznie 80,2% zdarzeń, z czego najwięcej (3,8%) jest w przedziale 5-10 minut. Przedziały 10-15 i 15-20 minut zawierają odpowiednio 1,8% oraz 1% zdarzeń. Oznacza to, że opóźnienie w generacji alarmu jest niezbędne, a minimalna jego wartość to 5 minut. Przedział 5-10 minut zawiera podwyższoną liczbę zdarzeń, co oznacza, że również ten przedział powinien być wzięty pod uwagę w rozważaniach na temat ustawienia parametru D. Następne przedziały zawierają z kolei bardzo mało zdarzeń. Celem wyeliminowania kolejnych 2,8% zdarzeń trzeba zwiększyć D o kolejne 10 minut (z 10 do 20), a należy pamiętać, że im większa wartość D tym mniej zdarzeń do analizy, ale też większe opóźnienie w rozpoczęciu prac nad usuwaniem awarii.

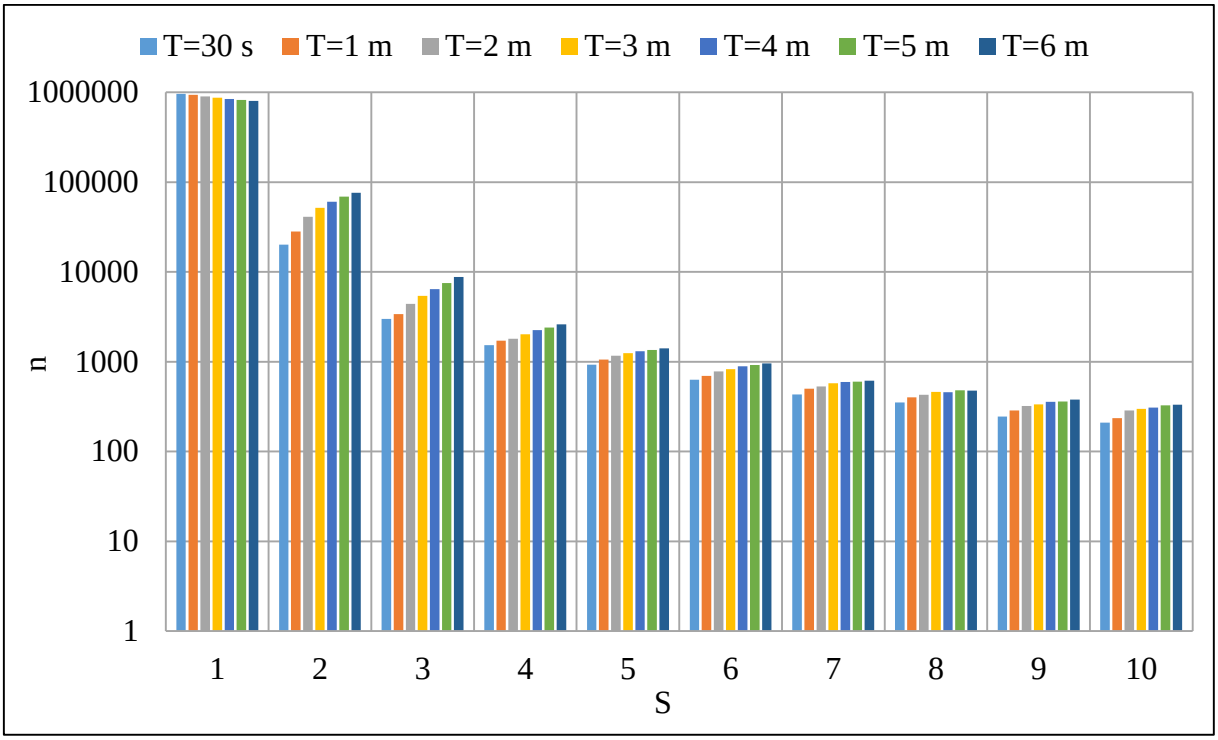
Na podstawie przeprowadzonych testów można rekomendować skrócenie 20 minutowego progu *D*, jaki został przyjęty we wdrożeniu do 10 minut, celem zmniejszenia opóźnienia w zgłaszaniu awarii.

Pokazany na Rys. 5.1 histogram można zinterpretować także jako statystyczny czas usuwania awarii przez służby operacyjne. Krótkie czasy trwania odnoszą się do zdarzeń, które usuwają się samoczynnie (dotyczą, jak wspomniano w rozdziale 3.1, prowadzonych prac w sieci, krótkich zaników zasilania, itp.). Dłuższe czasy trwania odnoszą się do rzeczywistych awarii. Należy zauważyć zwiększoną liczbę zdarzeń o czasie trwania od 35 do 400 minut. Są to najczęstsze przedziały, w których służby operatora były w stanie dotrzeć do źródła awarii i je usunąć. Liczba zdarzeń trwających powyżej 400 minut jest już bardzo mała.

5.2. PRZEDZIAŁ ANALIZY GRUPOWYCH ZERWAŃ SESJI (PARAMETR T)

W opisie algorytmu przedstawionym w rozdziale 4.1 przyjęto przedział analizy grupowych zerwań sesji równy 3 minuty. Taki przedział został wybrany w oparciu o informacje o „zrywaniu" sesji *PPP* po utracie transmisji danych w sieci kooperującego operatora. Są to trzy kolejno utracone ramki *keep-alive*, które przesyłane są cyklicznie w interwale 30 sekund. Oznacza to, że przedział maksymalny od utraty transmisji do rozłączenia sesji na urządzeniu *BRAS* wynosi 1,5 minuty. Należy jednak założyć przypadek negatywny, w którym zerwanie transmisji nastąpiło zaraz po wysłaniu ostatniej ramki *keep-alive*, co oznacza, że teoretycznie maksymalny czas rozłączenia sesji to 2 minuty od utraty danych. Ze względu na to, że następnie dane muszą zostać jeszcze przesłane przez systemy *RADIUS* oraz fakt, że nie wszystkie utraty transmisji przy awarii muszą następować jednocześnie, został wybrany arbitralnie przedział $T = 3$ minuty. Nie zanotowano uwag ze strony operatora w zakresie wyboru szerokości przedziału *T*.

Celem potwierdzenia postawionych założeń należy zbadać, jak inne przedziały *T* wpływają na zachowanie się algorytmu *AWA*. W tym celu dokonano analizy symulacyjnej różnych ustawień implementacji *AWA* w zakresie szerokości okna *T*. Badane były za każdym razem te same dane zebrane z sieci, a przedstawione wyniki, jak poprzednio, dotyczą wykrywania grupowych zerwań na kartach urządzeń DSLAM. Do symulacji zostały wybrane następujące przedziały *T*: 30 s, 1 m, 2 m, 3 m, 4 m, 5 m i 6 m.



Rys. 5.2: Liczba wykrytych zdarzeń (*n*) w funkcji liczby zerwań w danym zdarzeniu (*S*) oraz w zależności od szerokości okna grupowego zerwania (*T*)

Wyniki zostały pokazane na Rys. 5.2, który prezentuje liczbę wykrytych zdarzeń (*n*) zawierających od 1 do 10 zerwań sesji (*S*). Przedział zerwań sesji *S* od 1 do 3 jest czysto teoretyczny, ponieważ tak niskie progi algorytmu nie mają sensu rzeczywistego – nie wiadomo, czy jest to pojedyncze zerwanie usługi (przypadkowo skojarzone z zerwaniem innej sesji), czy jednak awaria. Dane pokazują, że im szersze okno *T* tym mniej jest wykrywanych awarii składających się z pojedynczych zdarzeń (*S*=1). Jest to spodziewane zachowanie, ponieważ w szerszym oknie mamy znacznie większe prawdopodobieństwo uwzględnienia innych zdarzeń (skorelowanych losowo) co przesuwa daną awarię do takiej, której przyczyną było więcej zdarzeń.

Przedziały wartości *S* interesujące dla operatora wynoszą od 4 wzwyż. Zanotowano dla nich wzrost liczby wykrytych grupowych zerwań sesji o danej liczbie zdarzeń wraz ze wzrostem okna analizy (*T*). W szczególności największy wzrost jest widoczny między przedziałami 30 s, 1 m i 2 m. Jest to spowodowane opisaną charakterystyką badanej sieci, gdzie dopiero od 2 minut możemy mówić o właściwej konfiguracji przedziału. Przedziały poniżej 2 minut powodują nieuwzględnienie wszystkich zerwań w obrębie danej awarii. Dalszy wzrost szerokości okna, do 6 minut powoduje kolejne zwiększanie liczby wykrytych zdarzeń, co wynika zarówno ze zdarzeń zaraportowanych z opóźnieniem (co zostało wspomniane w związku z koniecznością przesłania ich z *BRAS* do *RADIUS* oraz systemu analizującego

zdarzenia), jak też przez przypadkowe zdarzenia nieskorelowane z awarią, które po prostu statystycznie trafiły w okno wystąpienia zdarzenia.

Analizując Rys. 5.2 pod kątem rozkładu wykrytych zdarzeń z liczbą zdarzeń równą 8 (S=8) można dojść do wniosku, że przedział T równy 3 minuty wybrany arbitralnie jest bliski optymalnemu. Szersze przedziały T mogą zapewnić wyższą wykrywalność zdarzeń, jednak są obarczone większym błędem zakwalifikowania do awarii zdarzeń jej niedotyczących, co negatywnie wpływa na dalszy etap algorytmu. Skoro dane zdarzenie nie jest skorelowane z awarią, przy jej usunięciu jest bardzo mało prawdopodobne, aby usługa wróciła do pracy, a to wpływa negatywnie na ocenę skuteczności usunięcia awarii. Nie oznacza to jednak, że krótkie przedziały T są pozbawione tego ryzyka, aczkolwiek maleje ono wraz ze skracaniem przedziału T.

Ostateczny przedział (T) analizy grupowych zerwań sesji *PPP* musi być wybrany bazując na przestawionych w tym rozdziale symulacjach oraz uwarunkowaniach technicznych sieci operatora. Dłuższy przedział ma większą skuteczność wykrywania, ale może zawierać więcej zdarzeń nieprawidłowych. Zbyt krótki przedział oznacza z kolei zbyt niską skuteczność wykrywania awarii. Celem wybrania optymalnego przedziału T wyniki symulacji powinny być połączone z wiedzą o charakterystyce sieci. Analizując Rys. 5.2 znając sposób zachowania się sesji *PPP* można dojść do wniosku, że wybór T powinien paść na przedział z zakresu 2-4 minut, a okres 3 minutowy jest w środku tego przedziału i taka wartość została przyjęta w dalszych symulacjach. Wartość ta jest zgodna z pierwotnie wybraną u kooperującego operatora.

Należy zaznaczyć, że zdarzeń zawierających poniżej 3 (S<3) zerwanych usług nie można zakwalifikować jako potencjalne awarie. Są to pojedyncze zerwania usług występujące samoczynnie (S=1) lub przypadkowo skorelowane ze sobą dwa zdarzenia w oknie T (S=2). Mogą zdarzyć się sytuacje, kiedy dla zasobów sieciowych z trzema lub mniej podłączonymi usługami będą to awarie, ale w zaproponowanym rozwiązaniu nie da się analizować tego typu sytuacji ze względu na dużą liczbę fałszywych alarmów. Z tego powodu w dalszych symulacjach przyjęto wartość parametru *SMIN*=3 (por. Rys. 3.1). W opisanym wcześniej wdrożeniu ten parametr był ustawiany znacznie wyżej (6 lub 10), ale w celu wykonania symulacji wartość została obniżona do najniższej uzasadnionej logicznie, celem weryfikacji pełnego zakresu działania algorytmu. Wartość 6 we wspomnianym wdrożeniu była ustawiana dla zasobów sieciowych *ATM*, gdzie liczba usług na węzłach tego typu (ze względu na wygaszanie tej technologii) była mniejsza w stosunku do węzłów działających w technologii *IP*. Tam próg *SMIN* wynosił 10. Sam poziom wartości (6 i 10) był dodatkowo ustawiony ze

względu na znaną wielkość monitorowanych zasobów sieciowych, gdzie liczba portów na danej karcie to minimum 24.

5.3. ANALIZA PARAMETRU SP

Jednym z kluczowych parametrów *AWA* jest stosunek liczby zerwanych sesji do wszystkich aktywnych na analizowanym zasobie sieciowym, przed wystąpieniem zdarzenia (parametr *SP*). W teorii wartość *SP* dla awarii powinna wynosić 100%. W rzeczywistości występują jednak utraty danych *RADIUS Accounting* oraz inne sytuacje opisane w rozdziale 3.6 powodujące, że należy rozważyć ustawienie progu *SPMIN* na wartość mniejszą niż 100%. W ramach badań odporności algorytmu przeprowadzono analizę statystyczną rozkładu liczby wykrytych zdarzeń *GZS* w funkcji przedziału wartości parametru *SP*. W tym celu wprowadzono parametr:

$$nsp_i = \frac{n_i}{\sum_{k=1}^{k=10} n_k} \tag{11}$$

gdzie:

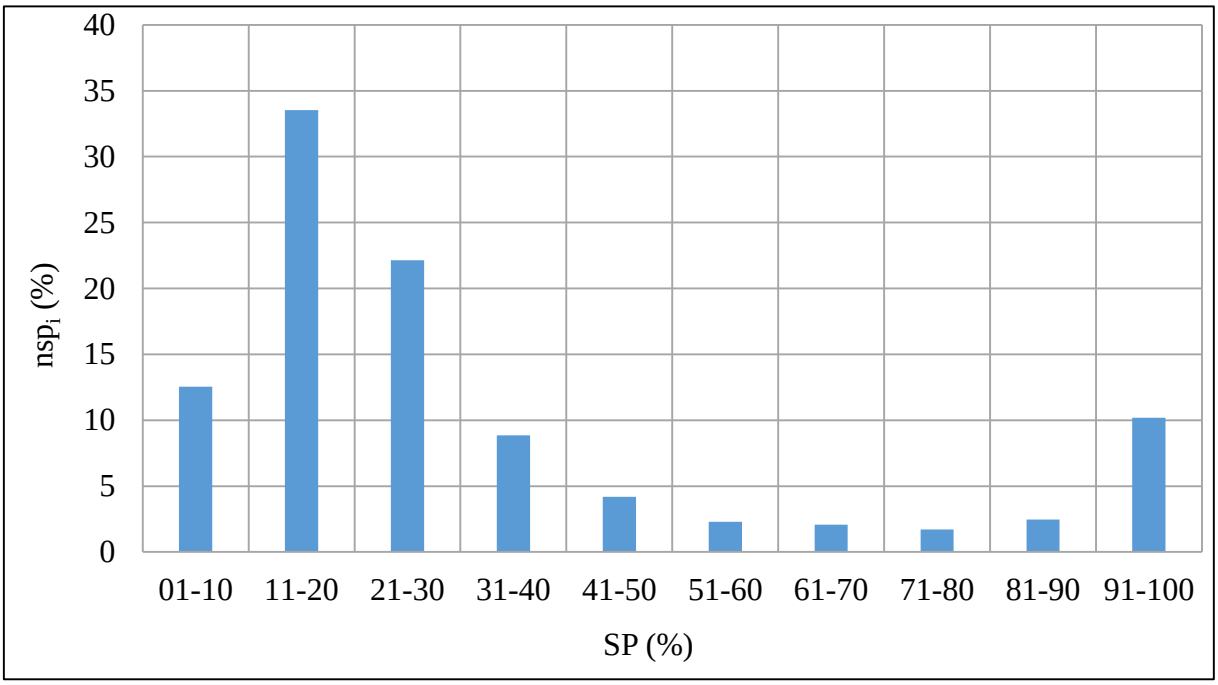
i – numer przedziału SP_i (od 1 do 10),

SP_i – przedział *SP* rozciągający się od $SP_{i,min}$ do $SP_{i,max}$,

n_i – liczba wykrytych zdarzeń w przedziale SP_i,

nsp_i – liczba wykrytych zdarzeń dla przedziału SP_i w stosunku do wszystkich wykrytych zdarzeń,

$\sum_{k=1}^{k=10} n_k$ – suma wszystkich wykrytych zdarzeń.



Rys. 5.3: Liczba wykrytych zdarzeń w stosunku do wszystkich (nsp_i) w funkcji SP_i

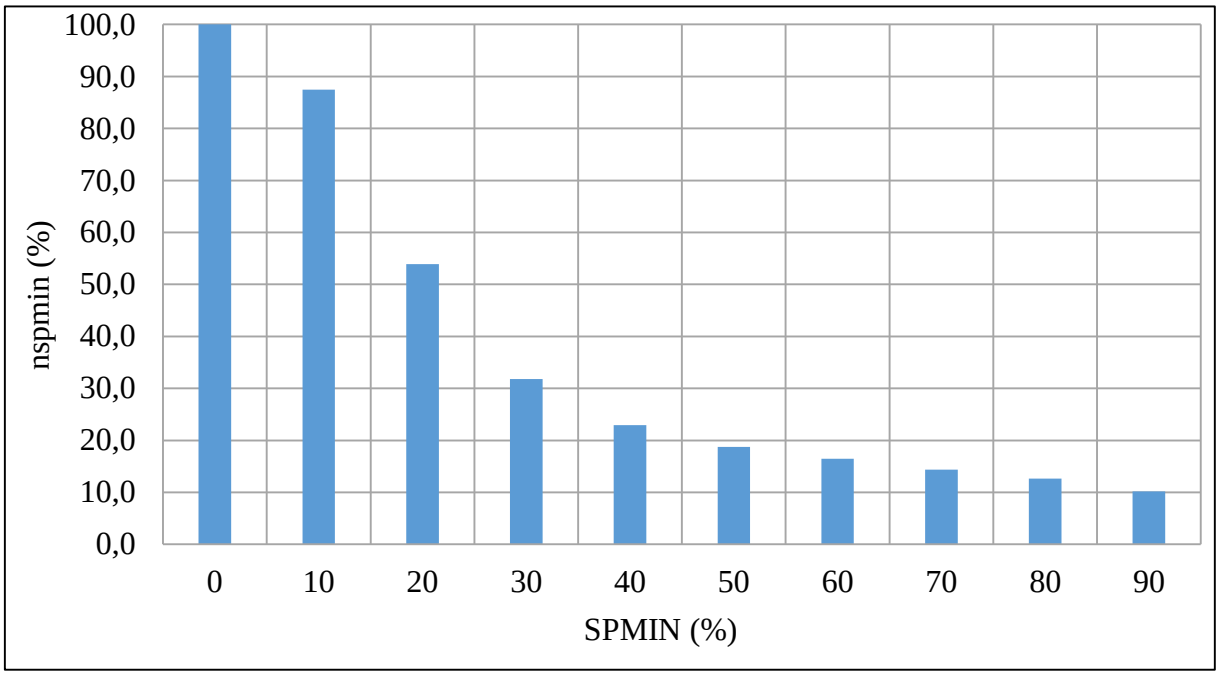
W przypadku spełnienia warunku *SP>=SPMIN* podstawowy algorytm *AWA* (Rys. 3.1) wchodzi w stan alarmu. Dla zobrazowania liczby alarmów w funkcji parametru *SPMIN* wprowadzono parametr:

$$nspmin = \frac{\mathrm{n(SPMIN)}}{\sum_{k=1}^{k=10} n_k} \tag{12}$$

gdzie:

n(SPMIN) – liczba alarmów dla wskazanego *SPMIN*,

nspmin – liczba alarmów w stosunku do wszystkich wykrytych zdarzeń *GZS* dla wskazanego *SPMIN*.



Rys. 5.4: Liczba alarmów w stosunku do wszystkich wykrytych zdarzeń *GZS* dla wskazanego *SPMIN*

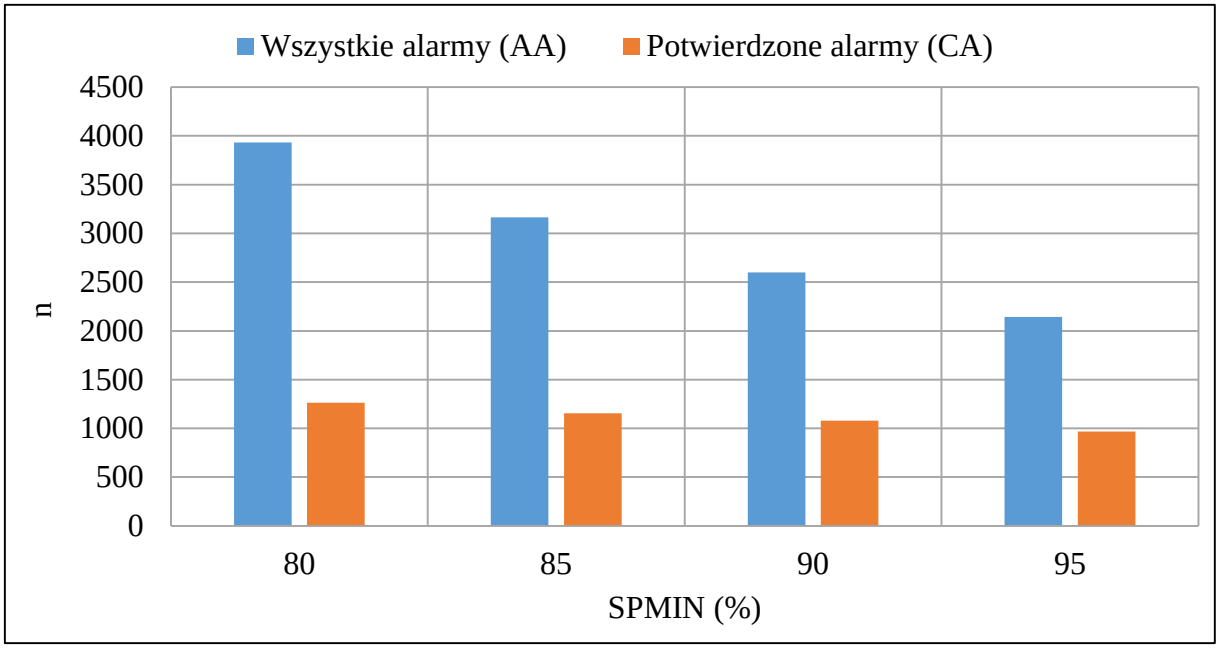
Rys. 5.3 przedstawia histogram wartości *SP*. Ustawiając parametr *SPMIN* na początku i-go przedziału ($SPMIN=SP_{i,min}$) i sumując wartości nsp_i od i-go przedziału w prawo, otrzymuje się wartości przedstawione na Rys. 5.4. Można zauważyć, że bardzo duża liczba alarmów występuje w przedziale *SP* poniżej 50%. Są to fałszywe alarmy wynikające z zerwań pojedynczych usług przypadkowo skorelowane w czasie lub awarie innych elementów sieci (na architekturze sieci w stronę *CPE* – por. Rys. 2.10), które powinny być wykryte przez *AWA* na elementach sieci niżej w hierarchii (tam SP_i powinien osiągnąć wartość zbliżoną do 100%). Faktyczne awarie danego zasobu sieciowego opisują zdarzenia o wartościach *SP* zbliżonych do 100%. Aby zbadać ten zakres wykonana została analiza w większej rozdzielczości (5%) dla

zakresu 80-100% (Rys. 5.5 i Rys. 5.6). Dla wykonania analizy otrzymanych wyników zostały zdefiniowane następujące parametry:

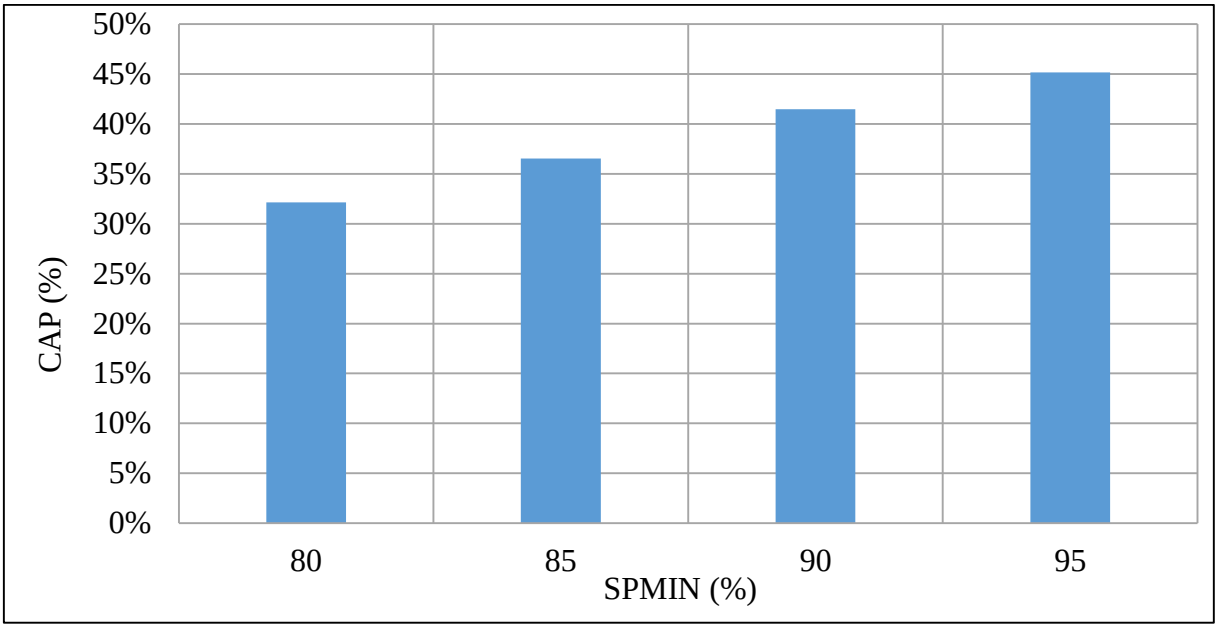
- *CA* (ang. *Confirmed Alarms*) – liczba potwierdzonych alarmów algorytmu *AWA*,
- *AA* (ang. *All Alarms*) – liczba wszystkich alarmów algorytmu *AWA*,
- *CAP* (ang. *Confirmed Alarms Percentage*) – stosunek *CA* do *AA*; zawiera się w przedziale (0, 1>.

$$CAP = \frac{CA}{AA} \tag{13}$$

Parametr *CA* wynika z rzeczywistych faktów – prezentuje on liczbę potwierdzonych przez operatora alarmów, jakie wystąpiły na sieci (rzeczywiste awarie).



Rys. 5.5: Liczba wszystkich alarmów (*AA*) oraz potwierdzonych (*CA*) w funkcji *SPMIN*



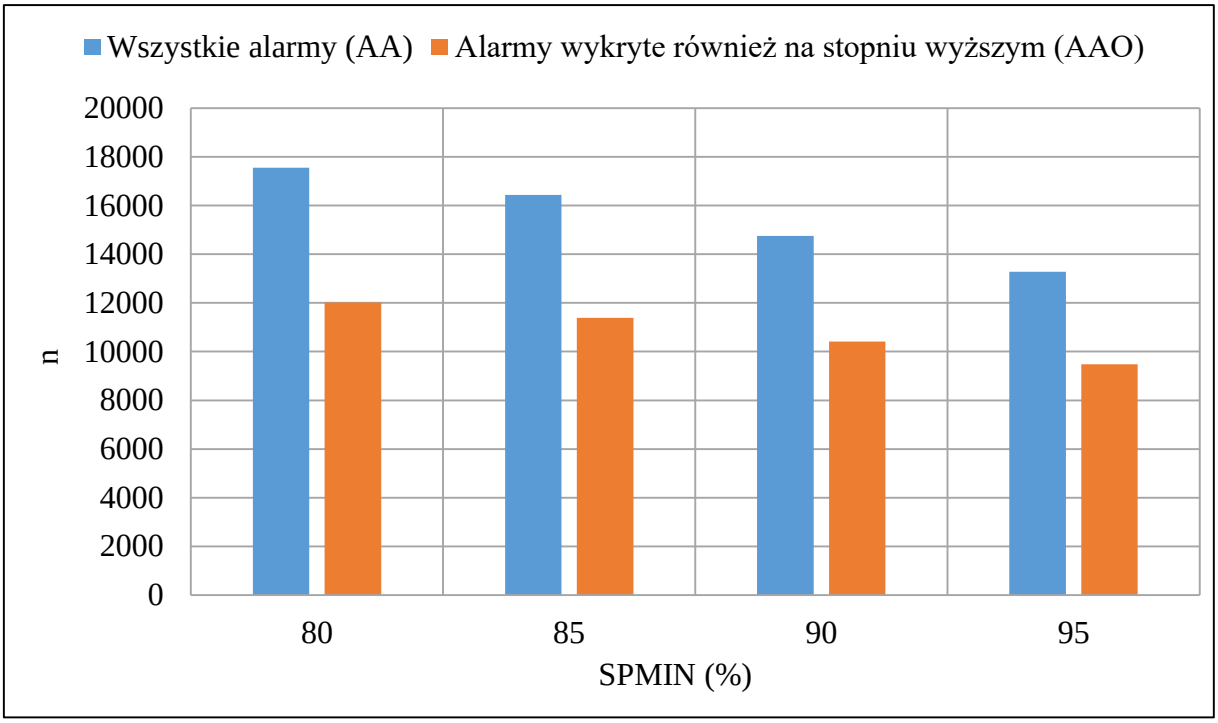
Rys. 5.6: Stosunek potwierdzonych do generowanych alarmów (*CAP*) w funkcji *SPMIN*

Wyniki potwierdzają, że najwyższa skuteczność *AWA* występuje dla *SPMIN* = 95%, gdzie osiąga on 45% potwierdzalności. Dla mniejszych wartości *SPMIN* skuteczność spada, a dla okresów z przedziału 80% jest najmniejsza. Patrząc jedynie na dane względne można byłoby zaakceptować do analizy cały przedział 80 – 100% (tj. ustawić *SPMIN* = 80%), jednak ustawienie *SPMIN* = 80% w stosunku do *SPMIN* = 95% spowodowałoby około dwukrotne zwiększenie liczby alarmów (96-100% to sumarycznie 2143 zdarzenia, a reszta przedziałów 1791). Byłyby to głównie fałszywe alarmy. Z drugiej strony dla przedziałów 81-95% wciąż występuje pewna część zdarzeń potwierdzonych, więc ich eliminacja spowodowałaby zmniejszenie skuteczności algorytmu.

Wyniki pokazane w tym podrozdziale są niejednoznaczne. Stąd nasuwa się pytanie: dlaczego wartości z Rys. 5.6 nie zbliżają się do 100% potwierdzonych alarmów? Wynika to z faktu, że dane te dotyczą jednego elementu logicznego sieci. Awarie innych elementów sieci w algorytmie korelacyjnym powodują, że wykrywa on alarmy na wszystkich elementach zależnych (por. rozdział 3.3). Przykładowo, rozważmy monitorowanie całego urządzenia *DSLAM* oraz niezależnie jego kart liniowych. Wtedy w przypadku awarii urządzenia *DSLAM* zostaną również aktywowane alarmy każdej z kart liniowych. W przedstawionej powyżej analizie – aktywowane alarmy kart zostaną niepotwierdzone, ponieważ przyczyna jest wyżej w hierarchii sieciowej, co znacznie zaniża wyniki skuteczności działania algorytmu. Z tego samego powodu przedstawione w tym rozdziale dane powinny być skorelowane z innymi stopniami algorytmu (por. 3.3).

5.4. BADANIE ZACHOWANIA KORELACJI STOPNI MONITOROWANIA

Korelacja stopni monitorowania została opisana w rozdziale 3.3. W rozdziale 5.3 wysunięto tezę, że brak takiej korelacji negatywnie wpływa na skuteczność algorytmu. W ramach testowania *AWA* zostały przeprowadzone symulacje wykrywania awarii na dwóch sąsiednich stopniach monitorowania (karta *DSLAM* – stopień niższy oraz całe urządzenie *DSLAM* – stopień wyższy).

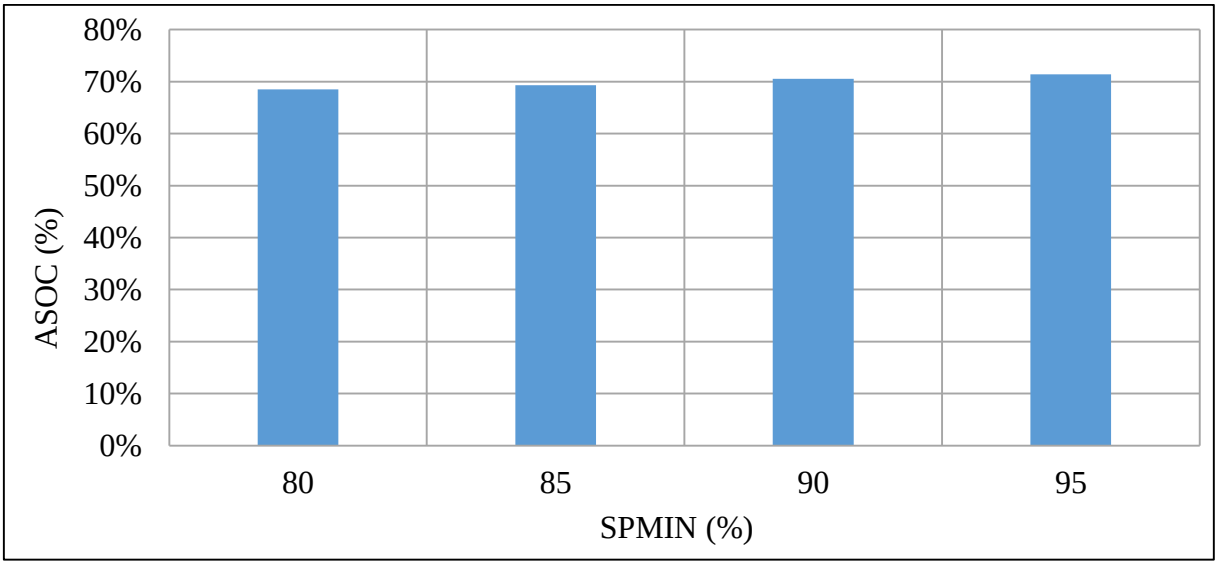


Rys. 5.7: Porównanie wszystkich alarmów do wykrytych na wyższym stopniu monitorowania w funkcji *SPMIN*

Parametr *AA* (*All Alarms*) oznacza wszystkie wykryte zdarzenia, a *AAO* (*All Alarms Overlapped*) to liczba tych zdarzeń, które w ramach nakładki *NAWA* zostały skojarzone jako takie, dla których wykryto również zdarzenie na wyższym stopniu monitorowania (bliżej w hierarchii sieciowej w stronę *BRAS / BNG* – por. Rys. 3.3, koniec *E6*). Stosunek tych wartości zdefiniowano jako *ASOC* (*All Stops Overlapped Compared*):

$$ASOC = \frac{AAO}{AA} \tag{14}$$

Wartości *ASOC* zostały przedstawione na Rys. 5.8. Przy porównywaniu zdarzeń pomiędzy stopniami monitorowania zastosowano próg *CH2MOP* równy 0,8 (80%).

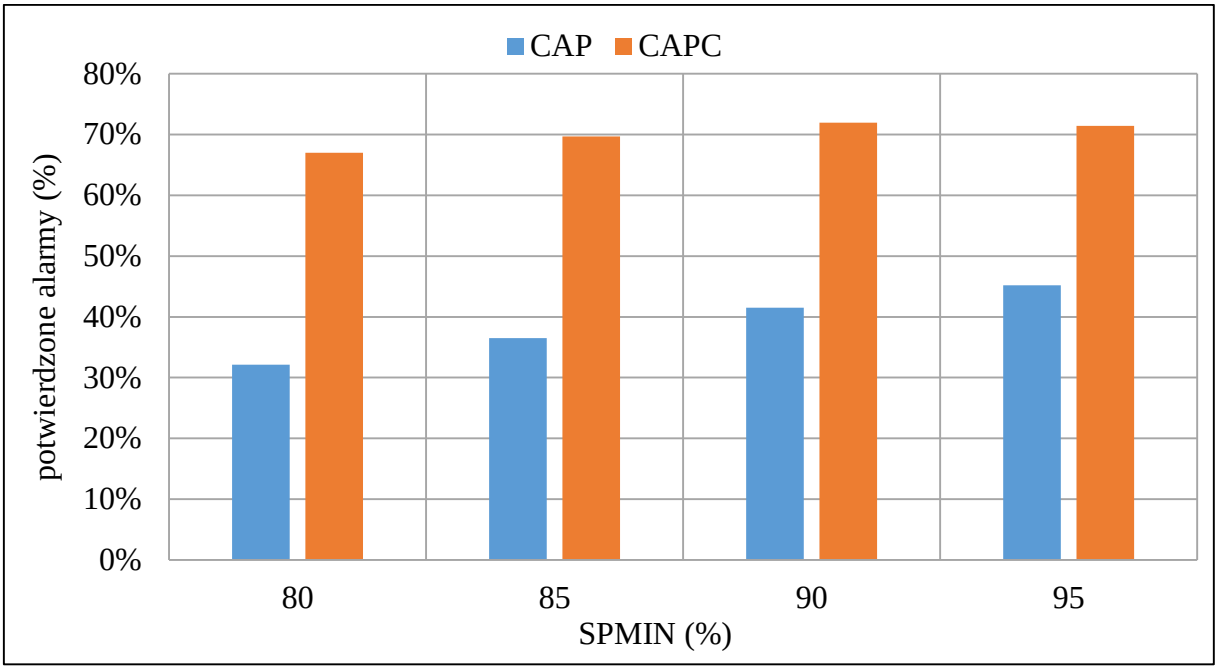


Rys. 5.8: Stosunek liczby aktywowanych alarmów na sąsiednich stopniach monitorowania do wszystkich alarmów (*ASOC*)

Analizując dane przedstawione na Rys. 5.8 można zauważyć, że znacznie powyżej połowy alarmów na niższym stopniu monitorowania było wykrywane także na wyższym stopniu, co potwierdza konieczność korelacji stopni monitorowania między sobą. Dla *SPMIN* w przedziale 95% aż 71,4% zdarzeń na kartach było spowodowanych awarią urządzenia *DSLAM* (lub innych, jeszcze wyższych w hierarchii elementów logicznych sieci). Dzięki korelacji można w tym przypadku wyeliminować aż 71,4% fałszywych alarmów. Co również warte odnotowania, stosowanie korelacji stopni monitorowania ma praktycznie taką samą skuteczność, niezależnie od przyjętego *SPMIN*.

Celem porównania liczby alarmów potwierdzonych przez operatora (*CAP*) do liczby alarmów wykrytej przez nakładkę *NAWA* zdefiniowano modyfikację parametru *CAP*:

- *CAPC* (ang. *Confirmed Alarms Percentage Correlated*) – wartość parametru *CAP* po odrzuceniu zdarzeń wykrytych na wyższym stopniu monitorowania (zgodnie z funkcjonalnością jaką wprowadza nakładka *NAWA*).



Rys. 5.9: Porównanie wartości *CAP* (*AWA*) do *CAPC* (*NAWA*)

Rys. 5.9 pokazuje, że poziom potwierdzalności alarmów wzrasta po zastosowaniu korelacji, którą wprowadza nakładka *NAWA*. Poziom potwierdzalności alarmów dla *NAWA* jest praktycznie taki sam w całym przedziale *SPMIN* od 80% do 95%. Różnice potwierdzalności w funkcji *SPMIN* są widoczne dla podstawowego algorytmu *AWA*, zaś *NAWA* eliminuje ten efekt. Dzięki temu można wysunąć wniosek, że zastosowanie *NAWA* umożliwia stosowanie niższych progów *SPMIN* przy utrzymaniu niezmiennej skuteczności biznesowej algorytmu. Niższe progi *SPMIN* są z kolei bardziej odporne na utraty pakietów w sieci, co jest efektem pożądanym (por. 3.6).

Mimo zastosowania korelacji parametr *CAPC* dla żadnej wartości *SPMIN* nie osiąga wartości zbliżonych do 100% – jest zawsze oddalony o co najmniej 20%. Prezentowane na wykresach dane są odnoszone do rzeczywiście potwierdzonych awarii przez operatora telekomunikacyjnego. Nie ma pewności, czy wszystkie awarie były faktycznie zarejestrowane w systemach ewidencyjnych awarii operatora. Część awarii zostać obsłużona bez rejestracji, co spowodowało, że nie wszystkie aktywowane alarmy *AWA* mają odzwierciedlenie w ewidencji awarii operatora telekomunikacyjnego. Poza tym niektóre alarmy mogą być fałszywe. Żaden system nie ma 100% skuteczności i mogą zdarzyć się sytuacje, że mimo zastosowania wszystkich teoretycznych porównań i analiz w sieci nie było awarii, ale dane ułożyły się losowo w ten sposób, iż system taki układ uznał za alarm. Zakładając, że pierwszy argument o braku ewidencji wszystkich awarii przez operatora jest niemierzalny należy przyjąć, że poziom generowania fałszywych alarmów przez algorytm *NAWA* wynosi 28% dla progu *SPMIN* (90%).

5.5. MODEL BADANIA WYDAJNOŚCI ALGORYTMU

Dla określenia możliwości skalowalności algorytmu, analizowano próbkę danych z jednego dnia z omówionych wcześniej zamrożonych 4 tygodni danych z rzeczywistej sieci. Do analizy wybrany został dzień 26.01.2019. Dane źródłowe pochodziły zarówno z 26.01, jak i 25.01, ponieważ celem obliczenia wartości parametru *SP* niezbędne jest wykorzystanie danych z wcześniejszej doby. Stworzony kod symulacyjny algorytmu został napisany w języku *Java* i składał się z kilku modułów (por. Rys. 4.1):

1. Moduł importu danych z bazy do tablic zoptymalizowanych pod kątem analizowanego zasobu sieciowego.

   Oznacza to, że dane są od razu grupowane w mapy, których kluczem jest analizowany zasób sieciowy. W przypadku zasobu jakim jest karta urządzenia *DSLAM* definicja mapy wygląda tak:

```
1  Map<DslamCard, List<RadiusAcctLog>> acctLogsPerDslamCardMap = new HashMap<>();
```

   Rys. 5.10: Przykładowa deklaracja mapy powiązań kart w języku *Java*

   Mapa na Rys. 5.10 zawiera listy logów *RADIUS Accounting* pogrupowanych per karta *DSLAM*. Taka struktura optymalizuje dalsze obliczenia, ponieważ dane dla wybranego zasobu znajdują się już we właściwym miejscu.

   Drugą zoptymalizowaną strukturą jest mapa:

```
1  Map<String, List<RadiusAcctLog>> acctLogsPerLoginMap = new HashMap<>();
```

   Rys. 5.11: Przykładowa deklaracja mapy powiązań loginów w języku *Java*

   Mapa ta zawiera logi *RADIUS Accounting* pogrupowane per login użytkownika (w wybranej sieci jest to unikatowy identyfikator łącza), dzięki czemu dalsza ocena aktywności sesji przed zerwaniem jest dokonana wydajnie za pomocą loginu.

   Podwójne mapy nie powodują podwojenia wymaganych zasobów pamięci *RAM*, ponieważ język programowania *Java* działa w oparciu o referencje. Oznacza to, że obie mapy odnoszą się do tych samych obiektów w pamięci.

   Czas przetwarzania danych przez moduł nr 1 jest oznaczony jako t_1.

2. Moduł wykrywania grupowych zerwań sesji.

   Moduł ten, dla każdego zerwania sesji w okresie *T*, dokonuje przeszukania logów *RADIUS Accounting* w przód i zapisuje wyniki w postaci listy wykrytych zerwań dla każdej karty. Struktura przestawia się następująco:

```
1  Map<DslamCard, List<StopsResult>> stopsPerCard = new HashMap<>();
```

Rys. 5.12: Przykładowa deklaracja mapy powiązań kart ze stopami w języku *Java*

Obiekt *StopsResult* wygląda w następujący sposób:

```
1  @Getter
2  @Setter
3  @ToString
4  public class StopsResult {
5      private RadiusAcctLog firstStop;
6      private RadiusAcctLog lastStop;
7      private List<RadiusAcctLog> stopsList;
8      private int stopsCnt;
9      private int stopsPercentage = -1;
10
11 }
```

Rys. 5.13: Przykładowy wygląd obiektu z wynikami wykrytych zatrzymań w języku *Java*

Obiekt ten zawiera informacje niezbędne do dalszych analiz wykrytych zerwań. Pole *firstStop* opisuje pierwszą zerwaną sesję, *lastStop* opisuje ostatnią, *stopsList* zawiera listę wszystkich zerwań w danym zdarzeniu, *stopsCnt* ich liczbę (*S*), a *stopsPercentage* jest wartością parametru *SP*.

Czas przetwarzania danych przez moduł nr 2 jest oznaczony jako t_2.

3. Moduł usuwający dane nachodzące na siebie.

   Moduł nr 2 wykrywa wszystkie grupowe zerwania dla każdego, nawet pojedynczego zerwania sesji *PPP*. Oznacza to, że w obrębie jednego logicznego zerwania grupowego wykrytych jest wiele różnych zerwań sesji będących częścią właściwego, większego zdarzenia. Opisywany moduł usuwa duplikaty z listy wykrytych zdarzeń. Usuwanie polega na przeszukaniu wszystkich wykrytych zdarzeń, posortowaniu ich w grupy nakładających się, wybraniu zdarzenia o największej liczbie zerwań i usunięciu pozostałych, z grupy nakładających się.

   Czas przetwarzania danych przez moduł nr 3 jest oznaczony jako t_3.

4. Moduł obliczający wartości parametru *SP*.

   Ostatni moduł oblicza wartości parametru *stopsPercentage* (*SP*). Do obliczenia wartości tego parametru, dla każdego grupowego zerwania moduł cofa się w czasie o minimalny okres generacji zdarzeń w sieci (*MAXT*) i analizuje wszystkie zdarzenia *RADIUS Accounting* (*Start* / *Stop* / *Interim-Update*) występujące w tym okresie.

   Czas przetwarzania danych przez moduł nr 4 jest oznaczony jako t_4.

Sumaryczny czas przetwarzania modułów t_1-t_4 jest oznaczony jako t_{14}:

$$t_{14} = \sum_1^4 t_n \tag{15}$$

Dane z wybranych dwóch dni zwierają około 13 mln wzorcowych rekordów *RADIUS Accounting*. Do przeprowadzenia symulacji różnych sytuacji napływu danych, były one zwielokrotniane, a także ograniczone do okrągłych dziesiątek milionów rekordów. Symulacje zostały przeprowadzone na komputerze o następujących parametrach:

Procesor *CPU*: *Intel(R) Core(TM) i7-6820HQ*

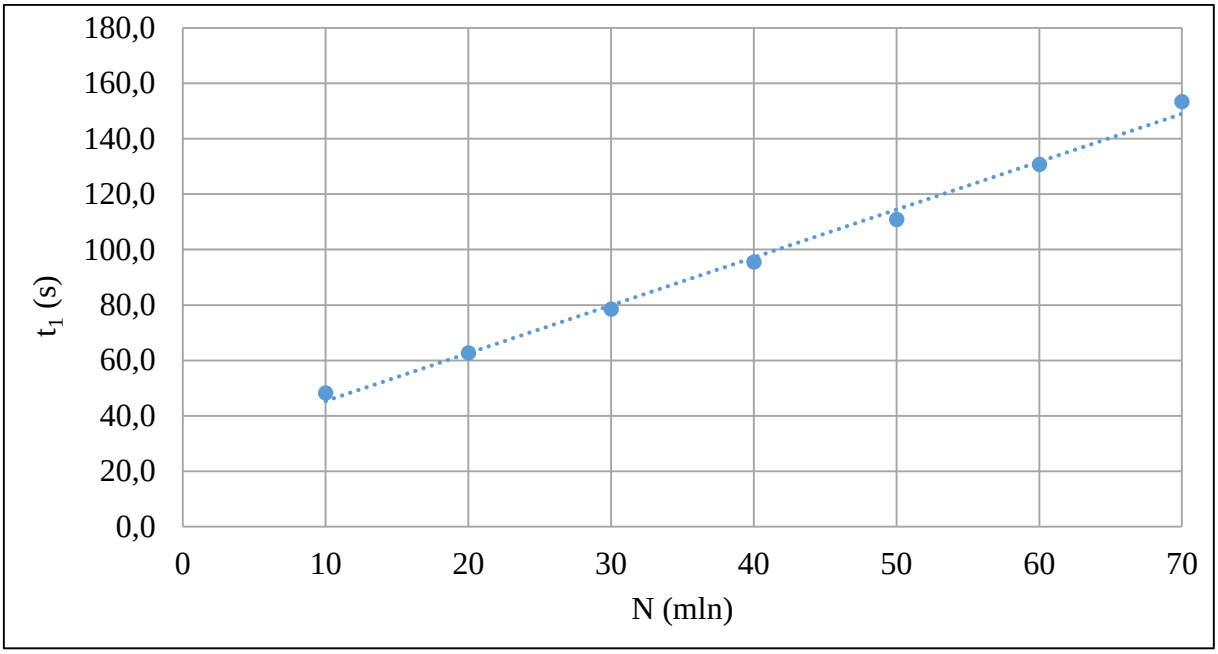
Pamięć *RAM*: *32GB*

Dysk *HDD*: *LITEON CV-3-8D512-41 SATA 512GB SED*

Zarówno kod symulacyjny, jak i baza danych znajdowała się na opisanym komputerze. Maksymalna wielkość stosu pamięci *Javy* została określona na 16 GB. Jest to maksymalna wielkość (w przybliżeniu), jaką język *Java* obsługuje poprawnie, stąd mimo, że komputer ma więcej pamięci, nie były stosowane wyższe wartości.

5.6. WYNIKI BADANIA WYDAJNOŚCI ALGORYTMU W FUNKCJI LICZBY REKORDÓW WEJŚCIOWYCH

Ważnym testem proponowanego rozwiązania jest pomiar wydajności poszczególnych modułów w funkcji liczby wejściowych danych *RADIUS Accounting* (*N*). Rekordy ładowane z bazy danych były zwielokrotniane celem uzyskania żądanej liczby rekordów wejściowych – 10 mln, 20 mln itd. Najwyższa zmierzona wartość *N* wynosiła 70 mln, ponieważ powyżej tej wartości brakowało pamięci i symulacja kończyła się błędem braku pamięci (ang. *Out Of Memory Error*).



Rys. 5.14: Czas ładowania danych t_1 (moduł 1) w funkcji liczby rekordów wejściowych

W ramach przedstawionych symulacji widać, że czas ładowania danych rośnie liniowo wraz ze wzrostem liczby rekordów (Rys. 5.14) zgodnie ze wzorem:

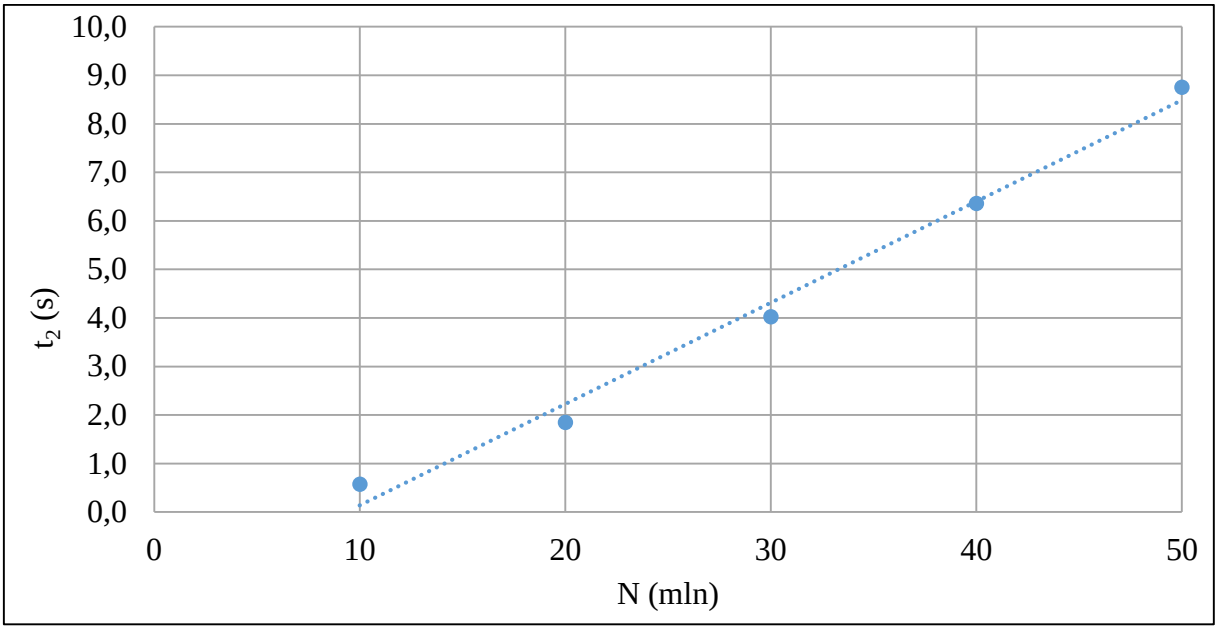
$$t_1 = 1{,}7282 \cdot \mathrm{N} + 28{,}049 \qquad (16)$$

gdzie:

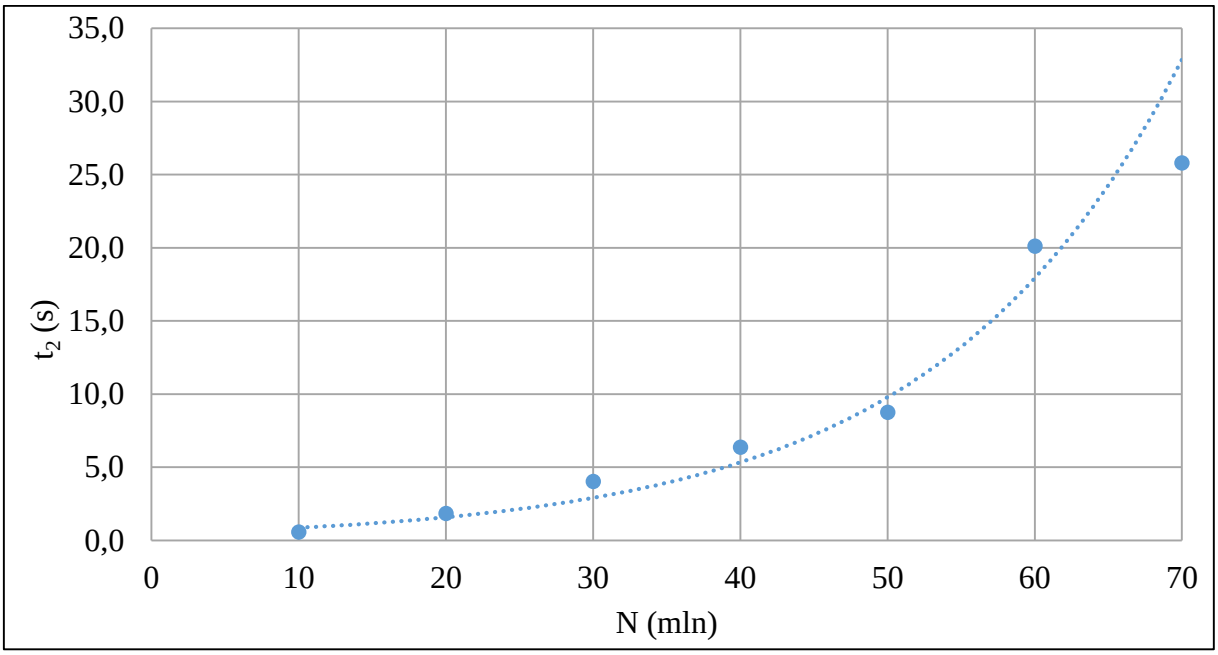
N – liczba rekordów wejściowych (w milionach),

t_1 – czas ładowania danych w sekundach (moduł 1).

Współczynnik korelacji Pearsona (R^2) [38] wynosi 0,997, a wariancja jest na poziomie 6,7 s. Stała czasowa modułu 1 wynosi 1,7282. Zależność liniowa pracy modułu 1 wynika z faktu, że liczba operacji do wykonania jest liniowo skorelowana z wielkością ładowanego zbioru danych.



Rys. 5.15: Czas wykrywania grupowych zerwań t_2 (moduł 2) w funkcji liczby rekordów wejściowych dla N z zakresu 10-50 mln



Rys. 5.16: Czas wykrywania grupowych zerwań t_2 (moduł 2) w funkcji liczby rekordów wejściowych dla pełnego zakresu N

Czas wykrywania grupowych zerwań sesji *PPP* został przedstawiony na Rys. 5.15 i Rys. 5.16. Na Rys. 5.15 pokazano wartości w zakresie N od 10 do 50 mln rekordów. W zakresie tym czas t_2 zachowuje się w sposób liniowy zgodnie z zależnością:

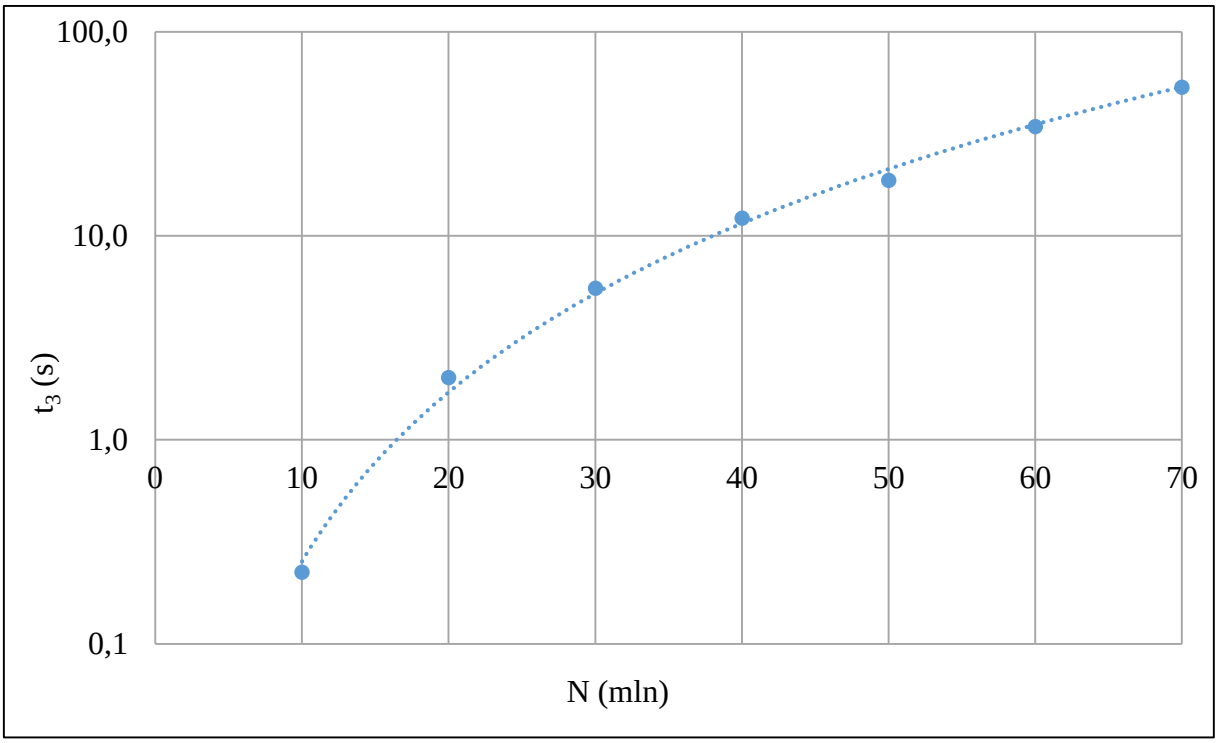
$$t_2 = 0{,}2087 \cdot \mathrm{N} - 1{,}9489 \qquad (17)$$

Współczynnik korelacji Pearsona wynosi 0,994 z wariancją na poziomie 0,1 s i stałą czasową 0,2087. Powyżej 50 mln rekordów wyniki zaczynają zachowywać się w sposób wykładniczy co ma związek z ograniczeniami pamięci *heap* w języku *Java* ustawionymi w trakcie testów na poziomie 16 GB. Krzywa na wykresie zamieszczonym na Rys. 5.16 jest opisana równaniem:

$$t_2 = 0{,}4727 \cdot exp(0{,}0606 \cdot N) \quad (18)$$

Przy czym współczynnik Pearsona przyjmuje wysoką wartość $R^2 = 0{,}961$.

W momencie, w którym pamięć zaczyna osiągać limit, maszyna wirtualna *Java* zaczyna większość operacji procesora zużywać do czyszczenia pamięci, co powoduje brak mocy obliczeniowej dla realizacji właściwych zadań obliczeniowych, a co za tym idzie wykładniczy wzrost czasu operacji.

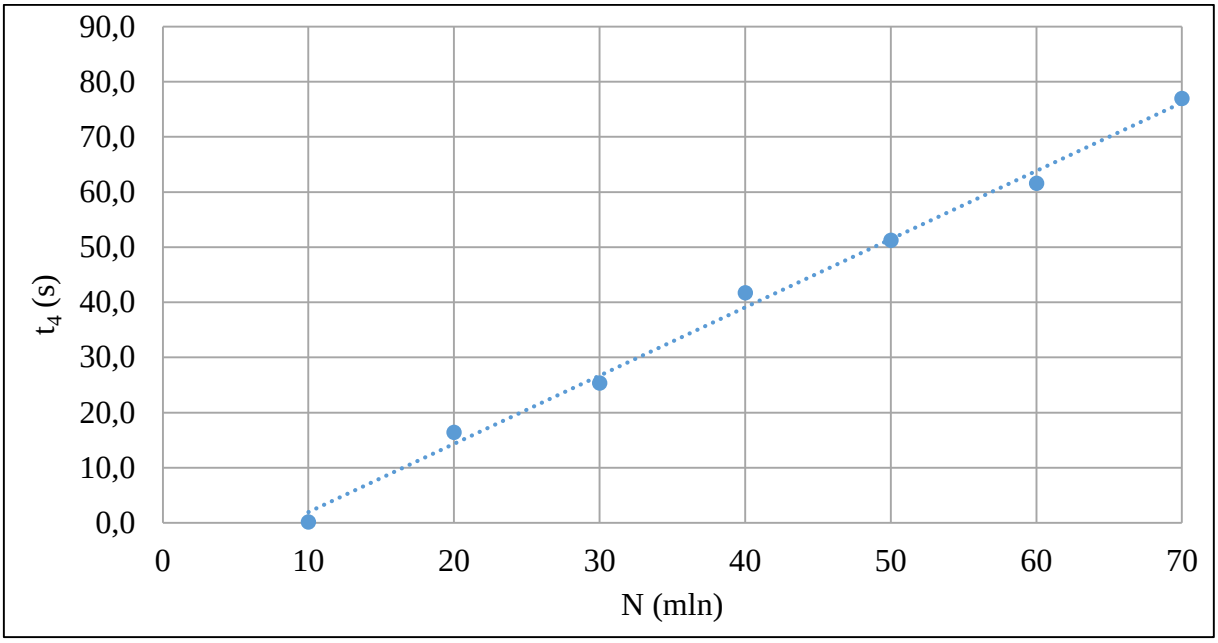


Rys. 5.17: Czas usuwania duplikatów t_3 (moduł 3) w funkcji liczby rekordów wejściowych

Na Rys. 5.17 przedstawiono czas usuwania duplikatów (t_3). Charakteryzuje się on przebiegiem w postaci funkcji potęgowej o wzorze:

$$t_3 = 0{,}0004 \cdot N^{2{,}7521} \quad (19)$$

Potęgowy czas usuwania duplikatów jest spowodowany tym, że algorytm przy każdym zdarzeniu skanuje cały zbiór danych co powoduje, że złożoność obliczeniowa rośnie co najmniej z kwadratem wielkości zbioru wejściowego. Współczynnik wyliczony w oparciu o regresję jest większy niż 2 (wynosi 2,7521), co wynika z omówionego zbliżania się do limitu pamięci *heap*.

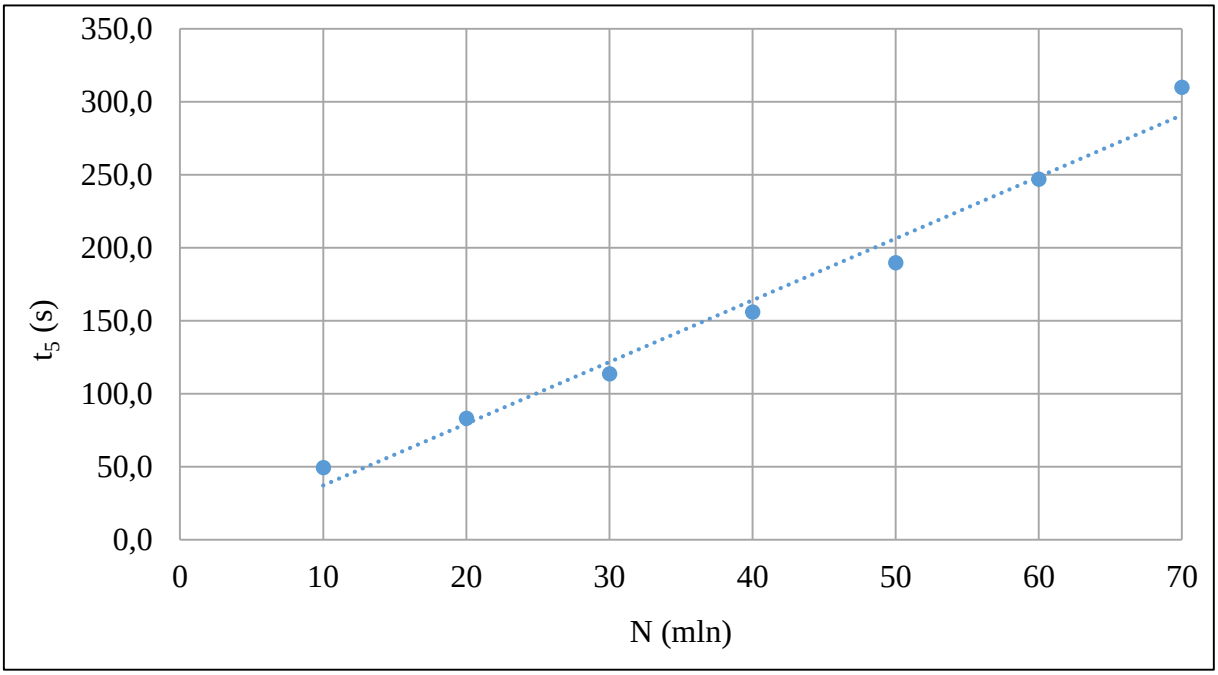


Rys. 5.18: Czas obliczania parametru *SP* t_4 (moduł 4) w funkcji liczby rekordów wejściowych

Na Rys. 5.18 przedstawiono czas obliczania parametru *SP* (t_4). Można przedstawić go jako funkcja liniowa:

$$t_4 = 1{,}2381 \cdot \mathrm{N} + 10{,}436 \qquad (20)$$

R^2 wynosi 0,997 z wariancją na poziomie 3,1 s. Stała czasowa tego modułu jest na poziomie 1,2381.

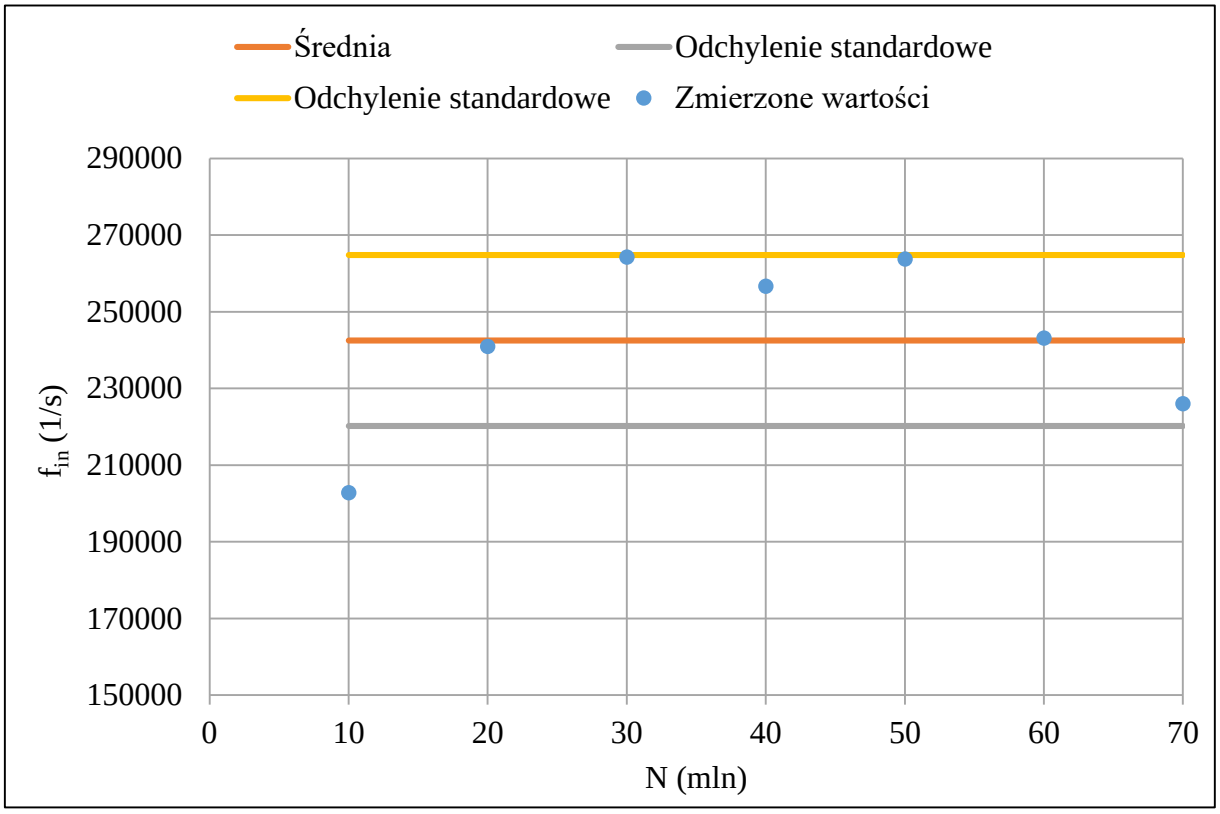


Rys. 5.19: Sumaryczny czas przetwarzania t_5

Sumaryczny czas przetwarzania t_5 (Rys. 5.19) jest sumą czasów t_1 – t_4. Czasy t_1, t_2 i t_4 estymowane funkcją liniową, jednak czas t_3 nie zachowywał się jak funkcja liniowa. Ze względu na niewielki wpływ czasu t_3 sumaryczny czas t_5 może być estymowany funkcją liniową:

$$t_5 = 4{,}2322 \cdot \mathrm{N} + 5{,}3177 \tag{21}$$

Współczynnik R^2 wynosi 0,998 z wariancją na poziomie 133,5 s. Współczynnik nachylenia prostej wskazuje, że stała czasowa na opracowanie 10 milionów rekordów przez algorytm wynosi 42,32 s, w rozpatrywanym przedziale liczby rekordów z serwera *RADIUS*.



Rys. 5.20: Liczba obsługiwanych rekordów *RADIUS Accounting* na sekundę (f_{in}) w funkcji *N*

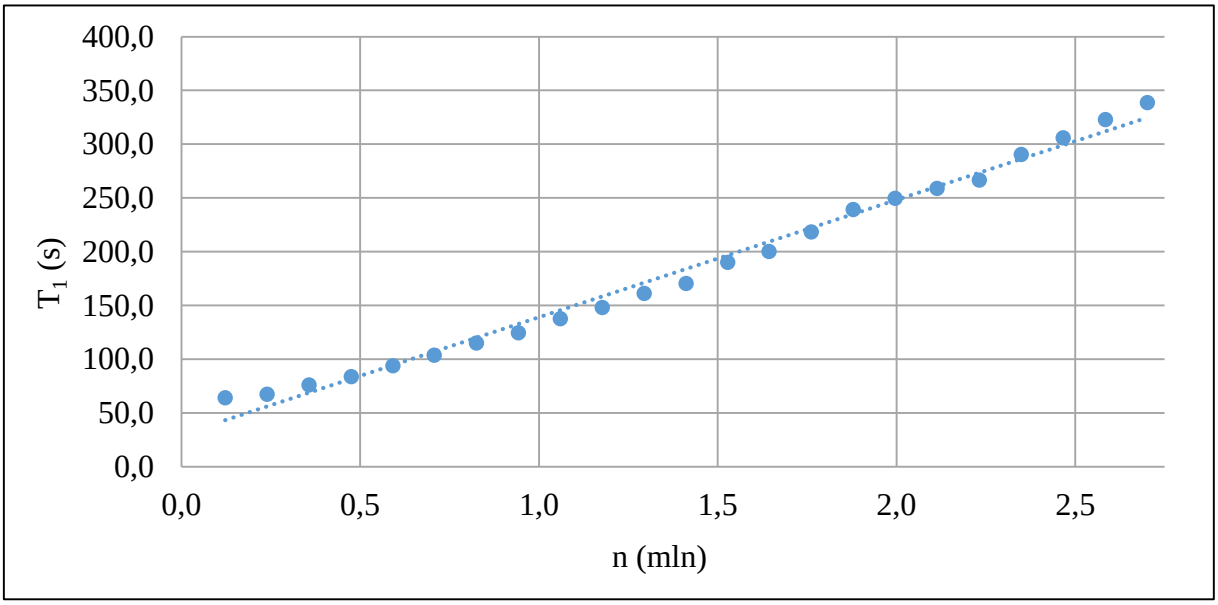
Wydajność *AWA* w postaci liczby rekordów *RADIUS Accounting* możliwych do obsłużenia w jednostce czasu (f_{in}) została przedstawiona na Rys. 5.20, gdzie przeliczono sumaryczny czas obsługi w stosunku do liczby rekordów wejściowych prezentując liczbę możliwych do obsługi zdarzeń na sekundę. Wyniki wahają się od 200 do 264 tys. zdarzeń na sekundę ze średnią na poziomie 242 tys. i odchyleniem standardowym na poziomie 22 tys. Należy przy tym uwzględnić to, że algorytm, który został użyty do symulacji jest algorytmem podstawowym – obliczającym główne parametry i zapisującym minimalny zestaw parametrów celem prawidłowej oceny zdarzeń w trakcie symulacji. Implementacje wdrażane produkcyjnie wymagają zapisu większej ilości informacji, wykonywania większej liczby operacji (jak

wysyłka zdarzeń do systemów trzecich, pobieranie dodatkowych informacji z urządzeń aktywnych, czy kolekcja większej ilości parametrów dot. zdarzenia celem wyświetlenia ich na *GUI* użytkownika). To wszystko wpływa negatywnie na skuteczność algorytmu i jest niemożliwe do oceny przy braku wymagań biznesowych konkretnego operatora.

Na podstawie przeprowadzonych analiz udało się potwierdzić, że rdzeń *AWA* przy optymalnym zbudowaniu modelu danych działa wydajnie. Napływ rekordów *RADIUS Accounting* na poziomie 200 tys. rekordów na sekundę nie ma szansy wystąpić dla operatorów o zasięgu europejskim.

5.7. WYDAJNOŚĆ ALGORYTMU W FUNKCJI LICZBY GENEROWANYCH ALARMÓW

Dla przeprowadzenia oceny wydajności algorytmu należy zbadać, jak wygląda czas obsługi w funkcji liczby alarmów generowanych przez algorytm *AWA*. Tym razem przeprowadzone testy polegały również na zwiększeniu liczby rekordów wejściowych, ale duplikowano tylko rekordy dotyczące zasobów, na których występowały alarmy *AWA*, tym samym zwiększając liczbę tychże alarmów konieczną do wykrycia. Dlatego liczba rekordów wejściowych wzrastała (większa liczba awarii wymaga większej liczby rekordów wejściowych), ale liniowo wzrastała też liczba generowanych alarmów. Przedstawione dalej wykresy mają na osi *X* liczbę alarmów w danej symulacji (*n*).

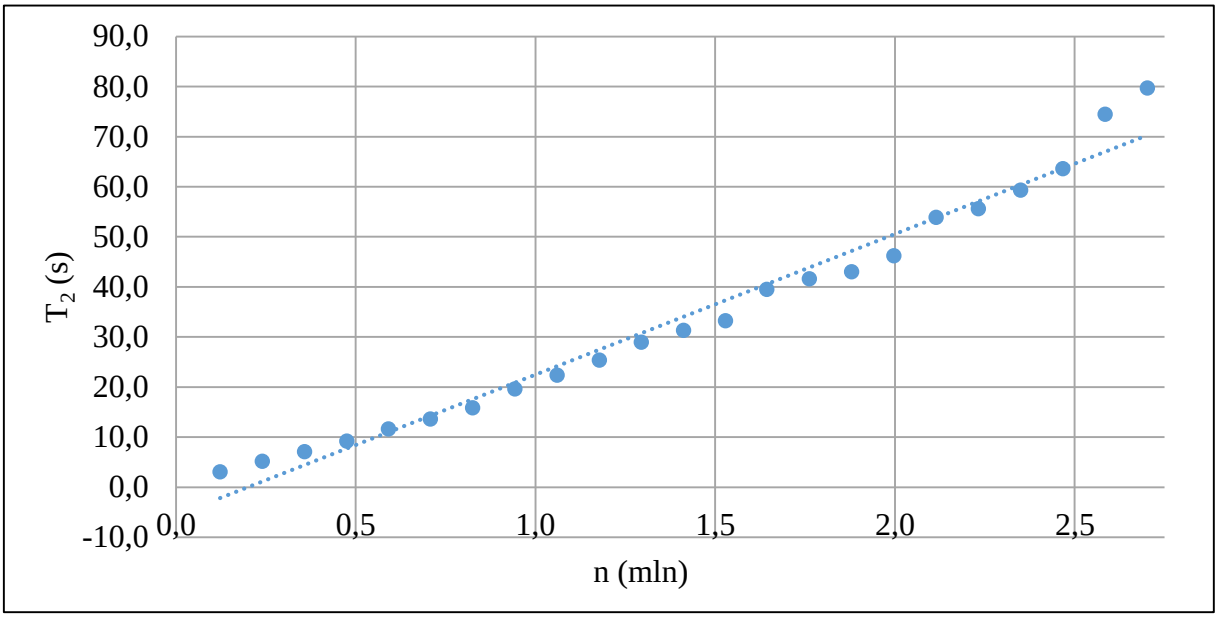


Rys. 5.21: Czas ładowania danych T_1 (moduł 1) w funkcji liczby wykrywanych alarmów (n)

Czas ładowania danych T_1, przedstawiony na Rys. 5.21 jest funkcją liniową:

$$T_1 = 109{,}13 \cdot \mathrm{n} + 29{,}981 \qquad (22)$$

ze stałą czasową równą 109,13 oraz R^2 na poziomie 0,995 (n jest liczbą alarmów wyskalowaną w milionach).

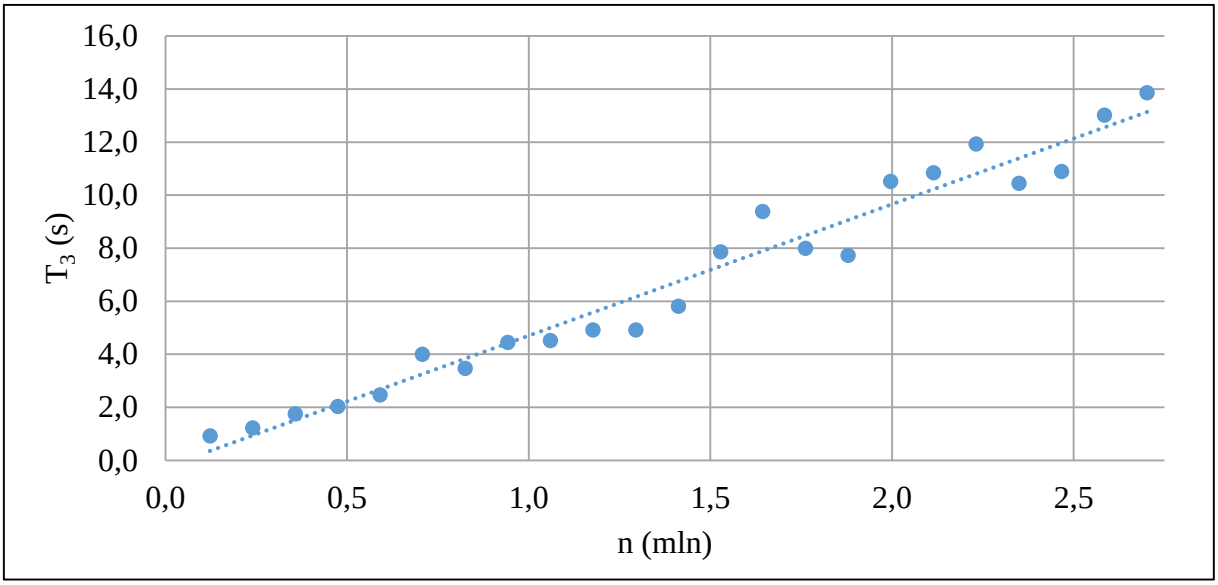


Rys. 5.22: Czas wykrywania grupowych zerwań T_2 (moduł 2) w funkcji liczby alarmów (n)

Czas wykrywania grupowych zerwań T_2 przedstawiony na Rys. 5.22 jest funkcją liniową:

$$T_2 = 28{,}082 \cdot \mathrm{n} + 5{,}5767 \qquad (23)$$

ze stałą czasową równą 28,082 i R^2 = 0,987.

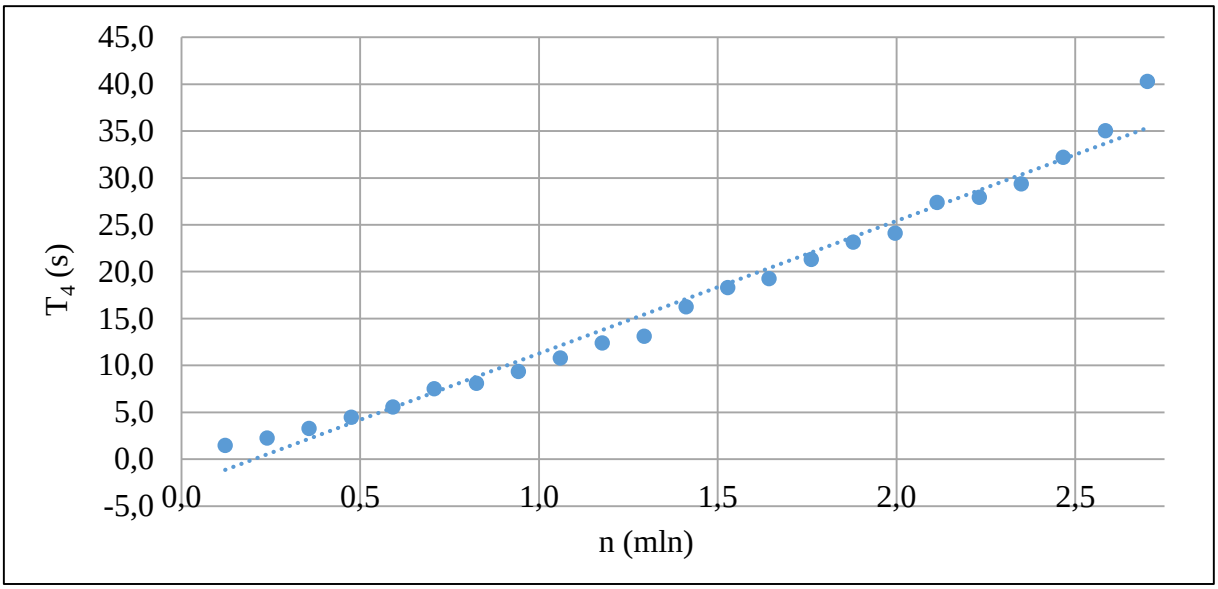


Rys. 5.23: Czas usuwania duplikatów T_3 (moduł 3) w funkcji liczby alarmów (n)

Czas usuwania duplikatów T_3 przedstawiony na Rys. 5.23 jest funkcją liniową:

$$T_3 = 4{,}9568 \cdot \mathrm{n} + 0{,}2498 \tag{24}$$

ze stałą czasową równą 4,9568 i R^2 = 0,980.

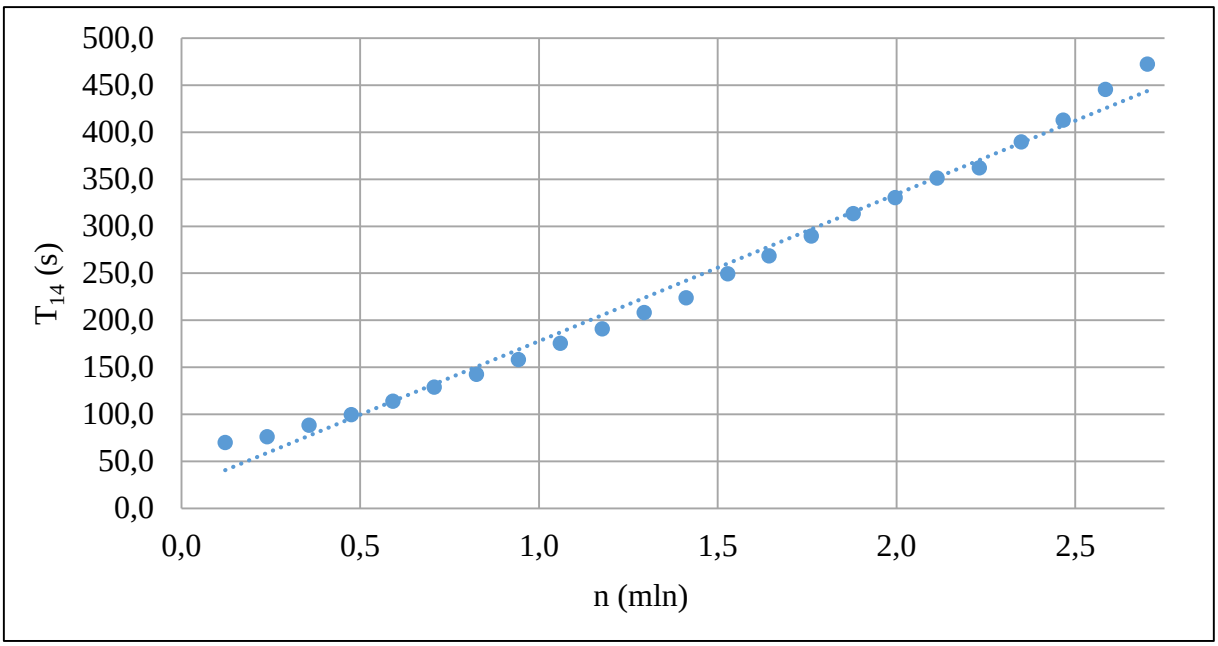


Rys. 5.24: Czas obliczania parametru *SP* T_4 (moduł 4) w funkcji liczby alarmów (*n*)

Czas obliczania parametru *SP* T_4 przedstawiony na Rys. 5.24 jest funkcją liniową:

$$T_4 = 14{,}133 \cdot \mathrm{n} + 2{,}8533 \tag{25}$$

ze stałą czasową równą 14,133 i R^2 = 0,990.



Rys. 5.25: Sumaryczny czas obliczeń modułów 1-4 (T_{14}) w funkcji liczby alarmów (*n*)

Sumaryczny czas T_{14} (przedstawiony na Rys. 5.25) można zdefiniować następująco:

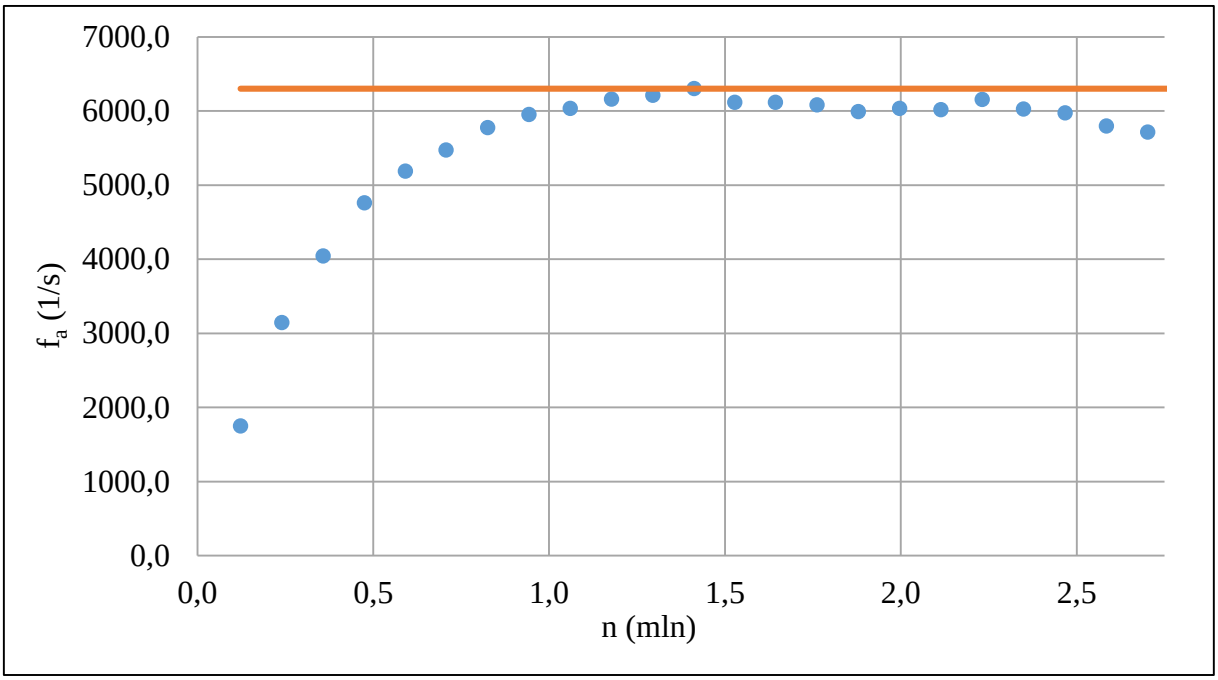
$$T_{14} = \sum_1^4 T_n \tag{26}$$

Wszystkie składowe czasu T_{14} były funkcjami liniowymi, zatem również T_{14} można scharakteryzować za pomocą funkcji liniowej:

$$T_{14} = 156{,}31 \cdot \mathrm{n} + 21{,}301 \tag{27}$$

gdzie stała czasowa wynosi 156,31, a R^2 = 0,994.

Wyniki czasu przetwarzania każdego modułu są zbliżone do liniowych, z dokładnością do wahań związanych z procesem czyszczenia pamięci wirtualnej maszyny *Java* lub innymi procesami systemu operacyjnego, które wprowadziły drobne anomalie czasów przetwarzania danych. Przedstawione wyniki czasów przetwarzania są zgodne ze spodziewanymi – w przypadku optymalnej implementacji do momentu napotkania limitów (jak chociażby limit pamięci) wzrost czasu przetwarzania danych jest liniowy.



Rys. 5.26: Częstotliwość generacji alarmów na sekundę (f_a) w funkcji liczby alarmów (n)

Na końcu przeanalizowano częstotliwość generacji alarmów (f_a) w funkcji liczby napływających alarmów (n), a wyniki zostały zilustrowane na Rys. 5.26. Początek wykresu pokazuje niższą niż asymptota wydajność, ponieważ stosunkowo najwięcej czasu algorytmu było spędzane w module ładowania danych (moduł 1). Moduł ten zawiera zestaw funkcjonalności, który jest mniej zależny od liczby zdarzeń (n) i przy małej wolumetrii zdarzeń przeważa on w całym zakresie przetwarzania danych, obniżając całkowitą wydajność detekcji. Wraz ze wzrostem liczby alarmów maleje jego znaczenie na tle reszty operacji do przetworzenia. Funkcjonalności, jakie są niezależne od liczby alarmów to sama aktywacja modułu, podłączenie do bazy danych, czy zbudowanie kluczy tablic, które przy

zwielokrotnianiu rekordów (co było użyte do zmiany liczby zdarzeń *n*) nie muszą być ponownie przeliczane. Istotnym czynnikiem był sam sposób przeprowadzenia badania, gdzie zwielokrotniano wyłącznie rekordy związane z alarmami, co oznacza, ze statystycznie malał stosunek liczby rekordów niezwiązanych z alarmami, co również przyczyniło się do zwiększania częstotliwości detekcji. Uzyskano maksymalną liczbę 6300 alarmów na sekundę do wykrycia, przy założeniu, że praktycznie cały zestaw przetwarzanych danych jest związany z alarmami. Wartości *n* powyżej 6300 są wartościami teoretycznymi i uzyskanymi w ramach symulacji, gdzie częstotliwość napływu alarmów była większa, niż możliwa do przetworzenia. Gdyby została napotkana taka sytuacja w rzeczywistym świecie, oznaczałoby to konieczność odrzucania napływających rekordów (ze względu na przekroczenie maksymalnej częstotliwości napływu) lub awarię implementacji *AWA* z powodu np. braku pamięci, procesora, etc. W przedstawionych symulacjach wyniki zostały uzyskane dzięki temu, że analizowany zbiór danych był ograniczony, a zatem po odczekaniu odpowiednio długiego czasu, mimo przekroczenia maksymalnej częstotliwości napływu, analiza zakończyła się.

W oparciu o przedstawione dane można stwierdzić niewielkie pogorszenie parametrów wydajnościowych dla ostatnich badanych przypadków. Jest to spowodowane (analogicznie jak poprzednio) zbliżaniem się to limitu przydzielonej pamięci *heap* w ramach tego rodzaju symulacji.

6. PODSUMOWANIE

Niniejsza praca jest skupiona na zagadnieniu monitorowania sieci dostępowej i agregacyjnej operatorów sieci stacjonarnych w celu detekcji awarii. Skuteczne wykrywanie zdarzeń w sieci, które mają wpływ na ciągłość świadczenia usług jest dla operatorów jednym z kluczowych zadań w zakresie jakości świadczonych usług. Proces ten wpływa bezpośrednio na zadowolenie użytkowników, co z kolei przekłada się na wyniki finansowe operatora ze względu na stabilną (lub nawet rosnącą) bazę świadczonych usług.

Dynamiczny rozwój usług sieciowych spowodował, że skuteczne wykrywanie wszystkich rodzajów awarii przy wykorzystaniu systemów monitorujących opartych na kolekcji predefiniowanych zbiorów danych jest trudne do realizacji. Z tego powodu w ramach realizacji niniejszej dysertacji opracowano system monitorowania sieci określony mianem *AWA – Algorytm Wykrywania Awarii* (por. 3.1). Opracowane rozwiązanie wyróżnia się na tle innych tym, że bazuje na odbieraniu w czasie rzeczywistym zdarzeń rozliczeniowych (*RADIUS Accounting*) ze wszystkich usług działających w sieci. Kolekcjonowane notyfikacje są następnie kojarzone z zasobami sieci, których mogą dotyczyć (co wynika ze ścieżki sieciowej, w oparciu o którą świadczona jest konkretna usługa). Po skojarzeniu z zasobami fizycznymi i logicznymi realizuje się korelację czasową pod kątem wykrycia jednoczesności wystąpienia grupy zdarzeń na poszczególnych zasobach sieci (zdarzenie takie określono mianem *Grupowego Zerwania Sesji – GZS*). To pozwala wnioskować, że na danym zasobie sieciowym mogło dojść do jakiegoś zdarzenia, potencjalnie awarii.

Realizacja niniejszej pracy została zrealizowana w oparciu o zdefiniowane cele. Pierwszym z nich było opracowanie algorytmu, którego podstawowa wersja (*AWA*) została przedstawiona w rozdziale 3.1. W rozdziale 3.3 przedstawiono nakładkę *NAWA*, która rozszerza bazową wersję *AWA* o pobieranie informacji o monitorowanym zasobie sieciowym oraz dokonuje analizy korelacji różnych stopni algorytmu *AWA*. W rozdziale 3.7 przedstawiono wersję *NAWA* wzbogaconą o funkcje proaktywnej naprawy zasobów sieciowych, zaś w 3.8 zdefiniowano pojęcie fluktuacji zasobów sieciowych oraz przedstawiono wersję algorytmu wzbogaconą o detekcję tego typu zdarzeń.

Drugim ze zdefiniowanych celów było wprowadzenie usprawnień implementacyjnych pod kątem wydajnościowym. Zostały one dokonane w ramach wdrożenia dla *Orange Polska S.A.*, gdzie zastosowano zoptymalizowane struktury danych oraz adekwatne metody ich analizy (por. 5.5). W trakcie prac implementacyjnych autor tej rozprawy dokonał trzech implementacji algorytmu, z czego dopiero druga była optymalna pod kątem wydajnościowym, a trzecia

została wzbogacona o funkcjonalności *NAWA* oraz generyczne monitorowanie zasobu sieciowego (wersja druga nie była generyczna).

Trzecim z celów pracy była weryfikacja w rzeczywistym środowisku, czego dokonano w oparciu o wdrożenie algorytmu *AWA* dla Orange Polska S.A. Implementacja *AWA* skutecznie działa do roku 2023 i nie ma planów na jej eliminacji lub zastąpienia innym rozwiązaniem. Omówiona sytuacja ma miejsce, mimo braku znaczących zmian w działaniu algorytmu od momentu jego wdrożenia, czyli w okresie około 6 lat. Potwierdza to potrzebę istnienia tego rozwiązania, jego niezawodności oraz braku innych tego typu rozwiązań na rynku.

Ostatnim celem tej pracy było wykonanie statystycznej analizy działania algorytmu, która została zrealizowana, a jej wyniki zostały zaprezentowane w rozdziale 5 niniejszej dysertacji.

Potwierdzone zostały także zdefiniowane na wstępie tezy. Pierwszą z tez była weryfikacja, czy algorytm *AWA*, realizujący korelacyjną analizę danych napływających z sieci, zapewnia większą skuteczność wykrywania awarii w stosunku do innych typów monitorowania sieci (bazujących wyłącznie na informacjach pozyskanych bezpośrednio z urządzeń sieciowych). Potwierdzenie tej tezy zostało przedstawione w rozdziale 4.3, gdzie na przykładzie danych z wdrożenia w *Orange Polska S.A.* potwierdzono, że liczba zdarzeń wykrywanych przez algorytm *AWA* jest większa niż suma zdarzeń wykrytych przez pozostałe systemy monitorujące u operatora. Poza tym, drugim potwierdzeniem skuteczności detekcji zdarzeń jest wspomniana wieloletnia praca implementacji *AWA* u operatora. Żadna firma ponosi kosztów utrzymania rozwiązań nieprzydatnych, stąd utrzymanie działania systemu jest automatycznym potwierdzeniem jego skuteczności i przydatności.

Drugą do zweryfikowania tezą było sprawdzenie, czy bazowanie na informacjach otrzymywanych jako notyfikacje wysyłane do systemu monitorującego zapewnia mniejsze opóźnienie w wykrywaniu awarii w stosunku do systemów cyklicznie monitorujących sieć oraz powoduje mniejsze obciążenie sieci. Badania w tym zakresie były prowadzone przez autora na samym początku pracy nad rozprawą doktorską, a wyniki tych badań ukazały się w publikacjach [13], [15], [39] oraz w tej pracy w rozdziale 2. Wyniki potwierdziły sformułowaną tezę w postaci mniejszego opóźnienia przy otrzymywaniu notyfikacji asynchronicznych oraz mniejszego obciążenia sieci za pomocą tego typu monitorowania. Jako, że algorytm *AWA* bazuje na obsłudze zdarzeń z urządzeń sieciowych w czasie rzeczywistym, tego typu weryfikacja była konieczna do przeprowadzenia.

Trzecią tezą do weryfikacji było potwierdzenie wydajności zaproponowanego rozwiązania *AWA*. Wystarczająca dla największego polskiego operatora sieci stacjonarnej została niejako automatycznie potwierdzona w oparciu o zrealizowane wdrożenie tego rozwiązania. Ponadto

w oparciu o symulacje zweryfikowano maksymalną wydajność zrealizowanej implementacji, która została przedstawiona w rozdziale 5.7. Maksymalna wydajność jednego serwera na poziomie 6300 zdarzeń na sekundę przekracza o co najmniej dwa rzędy wielkości potrzebę największego polskiego operatora, co oznacza, że zaimplementowane rozwiązanie jest możliwe do wdrożenia u każdego operatora. W oparciu o dane z wdrożenia oraz badania wydajności za pomocą dwóch niezależnych weryfikacji potwierdzono zdefiniowaną trzecią tezę.

Ostatnią tezą, jaką należało sprawdzić, była możliwość zastosowania *AWA* do monitorowania zasobów pasywnych, tj. takich, które nie pobierają żadnej energii lub nie komunikują się z innymi elementami sieci. Potwierdzenie tej tezy zostało zrealizowane w oparciu o wdrożenie w *Orange Polska S.A.* rozwiązania *AWA* do detekcji awarii kabli miedzianych (por. 4.6). Możliwość wdrożenia *AWA* dla tego typu zasobów wynika z pasywności zaprojektowanego rozwiązania, gdzie awarie nie są wykrywane w oparciu o bezpośrednią komunikację z urządzeniami, a o korelację informacji o architekturze sieci wraz z informacjami o stanie działania usług dla całej sieci. Dzięki temu można wnioskować o awarii poszczególnych elementów nie mając żadnej komunikacji z nimi.

Cele pracy zostały zatem zrealizowane, a tezy potwierdzone. Na końcu warto przypomnieć, że algorytm *AWA* w dniu 6.06.2016 został zgłoszony do Urzędu Patentowego Rzeczypospolitej Polskiej pod tytułem *„Sposób wykrywania awarii sieciowych”* [40]. W dniu 1.06.2017 zostało wysłane zgłoszenie do Europejskiego Urzędu Patentowego (ang. *EPO*) pod tytułem *„Method for Detecting Network Failures”* [41], a patent *EPO* został przyznany w dniu 27.03.2019. Publikacje, które były związane z realizowaną pracą doktorską to [13], [15], [39] oraz [42].

BIBLIOGRAFIA

[1] **ISO.** Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model. 1994. ISO standard 7498-1:1994.

[2] **ITU-T.** Integrated services digital networks (ISDNs). 1993. Recommendation I.120 (03/93).

[3] **ANSI.** Network and Customer Installation Interfaces - Asymmetric Digital Subscriber Line (ADSL) Metallic Interface. 1998. T1.413-1998.

[4] **ITU-T.** Asymmetric digital subscriber line (ADSL) transceivers. 1999. G.992.1.

[5] **ITU-T.** Splitterless asymmetric digital subscriber line (ADSL) transceivers. 1999. G.992.2.

[6] **ITU-T.** Asymmetric digital subscriber line transceivers 2 (ADSL2). 2002. G.992.3.

[7] **ITU-T.** Splitterless asymmetric digital subscriber line transceivers 2 (splitterless ADSL2). 2002. G.992.4.

[8] **ITU-T.** Asymmetric digital subscriber line 2 transceivers (ADSL2) - Extended bandwidth ADSL2 (ADSL2plus). 2003. G.992.5.

[9] **ITU-T.** Very high speed digital subscriber line transceivers 2 (VDSL2). 2007. G.993.2.

[10] **ITU-T.** Fast access to subscriber terminals (G.fast) - Power spectral density specification. 2014. G.9700.

[11] **ITU-T.** Fast access to subscriber terminals (G.fast) - Physical layer specification. 2014. G.9701.

[12] **ITU-T.** Multi-Gigabit broadband access over copper (MGfast). 2021. G.9701.

[13] **Zych P., Obrycki P., Obrycka J., Dąbrowski K. i Kula S.** Migracja usług z rodziny technil ADSL/2/2+ do techniki VDSL2. *Prace Seminarium Naukowego.* 2015, Tom 1, 69-83.

[14] **IETF.** User datagram protocol. 1980. RFC 768.

[15] **Zych P., Obrycki P., Obrycka J., Kula S. i Zych M.** Monitoring of xDSL CPE type from DSLAM side. *Przegląd Telekomunikacyjny.* 2015, Tomy 8-9, 1040-1045.

[16] **IETF.** A simple network management protocol (SNMP). 1990. RFC 1157.

[17] **IETF.** The Point-to-Point Protocol (PPP). 1994. RFC 1661.

[18] **IETF.** Remote Authentication Dial In User Service (RADIUS). 1997. RFC 2865.

[19] **Xinyu, Y., Xuebin, R., Shusen, Y. and McCann, J.** A novel temporal perturbation based privacy-preserving scheme for real-time monitoring systems. *Computer Networks.* 2015, Vol. 88, 72-88.

[20] **Naim N., F., Bakar A., Ashrif i A. Mohammad, S.** Real-time monitoring in passive optical access networks using L-band ASE and varied bandwidth and reflectivity of fiber Bragg gratings. *Optics & Laser Technology.* 2016.

[21] **Laijun, L. and Shanhong, S.** ZTE Unified Network Management Solution: From Distributed to Centralized Management. *Shen Shanhong Click:2301*. 2008.

[22] **Macías-Escrivá, F., D., Haber, R., del Toro, R. and Hernandez, V.** Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications.* 2013, Vol. 40 (18), 7267 – 7279.

[23] **Psiuk, M. and Zieliński, K.** Goal-driven adaptive monitoring of SOA systems. *Journal of Systems and Software.* 2013, Vol. 110, 101 - 121.

[24] **Kleinberg, J., Sandler, M. and Slivkins, A.** Network failure detection and graph connectivity. *Philadelphia: Society for Industrial and Applied Mathematics.* 2004.

[25] **Presotto, D., Jain, A., Siganporia, A. and Jain, S.** *Network failure detection. US 9106518 B1*. US Patent, 2007.

[26] **Najjar, W. and Gaudiot, J.** Network resilience: a measure of network fault tolerance. *IEEE Trans Comput 1990.* 1990, Vol. 39(2).

[27] **Sterbenz, J.P.G., Çetinkaya, E.K. and Hameed, M.A.** Evaluation of network resilience, survivability, and disruption tolerance: analysis, topology generation, simulation, and experimentation. *Telecommun Syst 2013.* 2013, Vol. 52(705).

[28] **Pi-Yu, C., Yennun, H., Deron, L., Chia-Yen, S. and Shalini, Y.** *Method and apparatus for providing failure detection and recovery with predetermined replication style for distributed applications in a network. US 6266781 B1*. US Patent, 1998.

[29] **Goyal, M., Ramakrishnan, K. K. and Wu-chi, F.** Achieving faster failure detection in OSPF networks. s.l. : IEEE Xplore, 2003.

[30] **Massa, D. and Lehner, O.** *Failure detection and failure handling in cluster controller networks. US 6934878 B2*. US Patent, 2002.

[31] **Gertler, J. and Singer, D.** A new structural framework for parity equation-based failure detection and isolation. *Automatica.* 1990, Vol. 26(2).

[32] **Rak, J.** Resilient routing in communication networks. *Comput Commun Networks 2015.* 2015.

[33] **IETF.** Dynamic Host Configuration Protocol. 1997. RFC 2131.

[34] **IETF.** Bidirectional Forwarding Detection (BFD). 2010. Tom FGCN 2010. RFC 5880.

[35] **IETF.** Failure Detection and Locator Pair Exploration Protocol for IPv6 Multihoming. 2009. RFC 5534.

[36] **IETF.** A Host Monitoring Protocol. 1983. RFC 869.

[37] **Group, Tcpdump.** TCPDUMP & LIBPCAP. [Online] https://www.tcpdump.org/.

[38] **Wright, S.** Correlation and causation. *Journal of agricultural research.* 20 (7), 1921.

[39] **Zych P.** Monitorowanie typu modemu XDSL od strony DSLAM. *Prace Seminarium Naukowego Instytutu Telekomunikacji Politechniki Warszawskiej. Działalność naukowa 2015/2016.* 2016, Tom 2, 21-35.

[40] **Zych P.** *Sposób wykrywania awarii sieciowych. P.417410.* UPRP, 2016.

[41] **Zych P.** *Method for detecting network failures. EP3252995.* European Patent Office, 2017.

[42] **Zych P.** Network failure detection based on correlation data analysis. *International Journal of Electronics and Communications (AEÜ).* 2016, Tom 77, 27-35.

SPIS ILUSTRACJI

SPIS TABEL

ZAŁĄCZNIKI

1. KOD ANALIZY DANYCH OFFLINE – WYKRYWANIE AWARII KART DSLAM

1.1. Obiekt CardFailAlgorithm (realizujący główny algorytm wykrywania awarii):

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Timestamp;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.*;
import java.util.concurrent.TimeUnit;
import java.util.logging.Logger;


public class CardFailAlgorithm2 {

    private static final Logger log =
Logger.getLogger(CardFailAlgorithm2.class.getName());
    private static final DateFormat df = new
SimpleDateFormat("yyyy-MM-dd HH:mm:ss", Locale.UK);
    private static final CardFailAlgorithm2 instance = new
CardFailAlgorithm2();
    private long lastRunningTime = -1;
    private List<CardFailAlgOutput2> lastRunningResult =
null;
    private double MIN_PERCENT_STOPS = 0.8;
    private static final int MINIMUM_ACTIVE_PORTS = 10;
    private static final long STARTS_LOOKUP_MAX_WINDOW_MS =
TimeUnit.MINUTES.toMillis(5 * 60);
    private static final long STOP_WINDOW_MS =
TimeUnit.MINUTES.toMillis(5);
    private static final long MIN_SESSION_UP_TIME_MS =
TimeUnit.HOURS.toMillis(21);
    private static final long MAX_SESSION_REFRESH_TIME_MS =
TimeUnit.SECONDS.toMillis(60);
    private static final long MAX_PACKET_SWITCHED_DIFF_MS =
TimeUnit.SECONDS.toMillis(1);
    public static final int GUI_HOURS_DISPLAYED = 2;
    private static final long GUI_HOURS_DISPLAYED_MS =
```

```
TimeUnit.HOURS.toMillis(GUI_HOURS_DISPLAYED);

    private CardFailAlgorithm2() {
    }

    public static CardFailAlgorithm2 getInstance() {
        return instance;
    }

    public double getMIN_PERCENT_STOPS() {
        return MIN_PERCENT_STOPS;
    }

    public void setMIN_PERCENT_STOPS(double
MIN_PERCENT_STOPS) {
        this.MIN_PERCENT_STOPS = MIN_PERCENT_STOPS;
    }

    private Set<Integer> buildNotStartedPorts(Set<Integer>
startedPorts, Set<Integer> stoppedPorts) {
        Set<Integer> notStartedPorts = new TreeSet<>();
        if (startedPorts != null && !startedPorts.isEmpty())
{
            for (Integer stoppedPort : stoppedPorts) {
                if (!startedPorts.contains(stoppedPort)) {
                    notStartedPorts.add(stoppedPort);
                }
            }
        } else {
            notStartedPorts = new TreeSet<>(stoppedPorts);
        }
        return notStartedPorts;
    }

    private Map<DslamPort, List<CfdRadiusAcct>>
buildPortMap(List<CfdRadiusAcct> list) {
        Map<DslamPort, List<CfdRadiusAcct>> portMap = new
HashMap<>();
        for (CfdRadiusAcct acct : list) {
            DslamPort port = new
DslamPort(acct.getDslamIp(), acct.getShelf(),
acct.getSlot(), acct.getPort());

            List<CfdRadiusAcct> portList =
portMap.get(port);
            if (portList == null) {
                portList = new ArrayList<>();
                portMap.put(port, portList);
            }
            portList.add(acct);
        }
```

```
        return portMap;
    }

    private void downloadDataAndInsertInMapByCard(DBConnect
db, Map<DslamCard, List<CfdRadiusAcct>> slotMap, int
daysAgo, boolean gui) throws SQLException {
        for (int i = daysAgo + 1; i >= daysAgo; --i) {
            String sql = "SELECT `timestamp`,
`dslam_ip_105`, `dslam_shelf_105`, `dslam_slot_105`,
`dslam_port_105`, `type`, `acct_terminate_cause` FROM `"
                    + Database.FD_DB_NAME_RADIUS + "`.`" +
DbUtilCommon.getAccountingEthTableName(DbUtil.buildDateTable
Name(i)) + "`";
            if (gui && i == 1) {
                sql += " WHERE `timestamp`>'" + new
Timestamp(System.currentTimeMillis() -
TimeUnit.HOURS.toMillis(26)) + "'";
            }
            log.finest("select: " + sql);
            long sqlStart = System.currentTimeMillis();
            try (Statement s =
db.getConnection().createStatement(); ResultSet rs =
s.executeQuery(sql)) {
                log.finest("Db download time: " +
(System.currentTimeMillis() - sqlStart) + ", for: " + sql);

                long mapInsert = System.currentTimeMillis();
                while (rs.next()) {
                    Timestamp time =
rs.getTimestamp("timestamp");
                    String type = rs.getString("type");

                    String dslamIp =
rs.getString("dslam_ip_105");
                    int shelf =
rs.getInt("dslam_shelf_105");
                    int slot = rs.getInt("dslam_slot_105");
                    int port = rs.getInt("dslam_port_105");
                    String acctTerminateCause =
rs.getString("acct_terminate_cause");

                    if (type.equals("Stop") &&
acctTerminateCause.equals("User-Error")) {
                        continue;
                    }

                    DslamCard dslamSlot = new
DslamCard(dslamIp, 1, shelf, slot);

                    CfdRadiusAcct acct = new
CfdRadiusAcct(time, DEFAULT_RACK_NR, shelf, slot, port,
```

```
dslamIp, type, RadiusAcctLog.AcctType.ATM);
                    List<CfdRadiusAcct> list =
getListFromMap(slotMap, dslamSlot);
                    list.add(acct);
                }
                log.finest("Map insert time: " +
(System.currentTimeMillis() - mapInsert));
            }
        }
    }

    private void filterResults(boolean gui,
List<CardFailAlgOutput2> toReturn) {
        if (gui) {
            filterResultsGui(toReturn);
        } else {
            filterResultsOnlyNight(toReturn);
        }
    }

    private List<CfdRadiusAcct>
getListFromMap(Map<DslamCard, List<CfdRadiusAcct>> map,
DslamCard slot) {
        List<CfdRadiusAcct> list = map.get(slot);
        if (list == null) {
            list = new ArrayList<>();
            map.put(slot, list);
        }
        return list;
    }

    public void getPFCRadiusAcctForDslamSlot(String
dslamIpInput, int shelfNr, int slotNr) throws SQLException,
ClassNotFoundException {
        try (DBConnect db = new DBConnect()) {
            List<CfdRadiusAcct> list = new ArrayList<>();
            for (int i = 8; i >= 0; --i) {
                String select = "SELECT * FROM
`fault_detector_radius`.`" + DbUtil.buildDateTableName(i) +
"_acct_eth` WHERE `dslam_ip`='" + dslamIpInput
                        + "' AND `shelf`=" + shelfNr + " AND
`slot`=" + slotNr + " ORDER BY `index`";
                try (Statement s =
db.getConnection().createStatement(); ResultSet rs =
s.executeQuery(select)) {

                    while (rs.next()) {
                        Timestamp time =
rs.getTimestamp("timestamp");
                        String dslamIp =
rs.getString("dslam_ip");
```

```
                        String type = rs.getString("type");
                        int shelf = rs.getInt("shelf");
                        int slot = rs.getInt("slot");
                        int port = rs.getInt("port");

                        CfdRadiusAcct acct = new
CfdRadiusAcct(time, DEFAULT_RACK_NR, shelf, slot, port,
dslamIp, type, RadiusAcctLog.AcctType.ATM);
                        list.add(acct);
                    }
                }
            }
            printlnActives(list);
        }
    }

    private void printlnActives(List<CfdRadiusAcct> list) {
        Set<Integer> activePorts = new TreeSet<>();
        for (CfdRadiusAcct acct : list) {
            updateActivePortsSet(activePorts, acct);
        }
    }

    public synchronized List<CardFailAlgOutput2>
testEthOnlyPorts(int daysAgo, boolean gui, BufferedWriter
writer) throws SQLException, ClassNotFoundException,
IOException {
        long start = System.currentTimeMillis();
        Calendar c = Calendar.getInstance();
        c.setTimeInMillis(start -
TimeUnit.DAYS.toMillis(daysAgo));
        int dayOfYearToCheck = c.get(Calendar.DAY_OF_YEAR);

        String date = DbUtil.buildDate(daysAgo, "-");

        try (DBConnect db = new
DBConnect(Database.FD_DB_NAME_RADIUS)) {
            if (gui) {
                long slotAlgMinIntervalSec =
Integer.valueOf(DBGetter.getSettingsValue(db,
DbSetting.SLOT_ALG_MIN_INTERVAL_SEC));
                if (start -
TimeUnit.SECONDS.toMillis(slotAlgMinIntervalSec) <
lastRunningTime) {
                    log.info("Last running time: " +
df.format(lastRunningTime) + ", now: " + df.format(start) +
", returning last calculated results");

                    return lastRunningResult;
                }
            }
```

```
            log.finest("testEthOnly DBConnect time: " +
(System.currentTimeMillis() - start) + ", days: " +
daysAgo);

            List<CardFailAlgOutput2> toReturn = new
ArrayList<>();
            Map<DslamCard, List<CfdRadiusAcct>> slotMap =
new HashMap<>();

            downloadDataAndInsertInMapByCard(db, slotMap,
daysAgo, gui);

            long analyzeStart = System.currentTimeMillis();
            writer.append("DATE,DSLAM IP,DSLAM
NAME,M1400,SHELF,SLOT,PORT,STOPS CNT\r\n");

            for (Map.Entry<DslamCard, List<CfdRadiusAcct>>
entry : slotMap.entrySet()) {
                DslamCard dslamCard = entry.getKey();

                List<CfdRadiusAcct> cardList =
entry.getValue();


removeSessionRefreshAndRepairSwitched(cardList);
                String dslamName = null;
                String m1400 = null;
                String dslamIp = dslamCard.getDslamIp();
                int shelf = dslamCard.getShelf();
                int slot = dslamCard.getSlot();
                String select = "SELECT * FROM
`fault_detector`.`ksp_device_info` WHERE `DSLAM_IP`='" +
dslamIp + "' AND `SHELF`='" + shelf + "' AND `SLOT`='" +
slot + "' LIMIT 1";

                try (Statement s =
db.getConnection().createStatement(); ResultSet rs =
s.executeQuery(select)) {
                    if (rs.next()) {
                        dslamName =
rs.getString("DSLAM_NAME");
                        m1400 =
rs.getString("M1400_SUFIKS");
                    }
                }

                Map<Integer, Integer> stopsMap = new
TreeMap<>();
                for (CfdRadiusAcct acct : cardList) {
                    long timestamp1 =
acct.getTimestamp().getTime();
```

```
                    c.setTimeInMillis(timestamp1);
                    int dayOfYear =
c.get(Calendar.DAY_OF_YEAR);

                    if (dayOfYear == dayOfYearToCheck &&
acct.getType().equals("Stop")) {
                        Integer portNr = acct.getPort();
                        Integer cnt = stopsMap.get(portNr);
                        if (cnt == null) {
                            stopsMap.put(portNr, 1);
                        } else {
                            stopsMap.put(portNr, ++cnt);
                        }
                    }
                }

                for (Map.Entry<Integer, Integer> entry1 :
stopsMap.entrySet()) {
                    Integer port = entry1.getKey();
                    Integer cnt = entry1.getValue();
                    writer.append(date + "," + dslamIp + ","
+ dslamName + "," + m1400 + "," + shelf + "," + slot + "," +
port + "," + cnt + "\r\n");
                }
            }

            writer.flush();

            log.finest("Analyze time: " +
(System.currentTimeMillis() - analyzeStart));
            return toReturn;
        } finally {

            log.finest("testEth time: " +
(System.currentTimeMillis() - start));
        }
    }

    public synchronized List<CardFailAlgOutput2> testEth(int
daysAgo, boolean gui) throws SQLException,
ClassNotFoundException {
        long start = System.currentTimeMillis();

        try (DBConnect db = new
DBConnect(Database.FD_DB_NAME_RADIUS)) {
            if (gui) {
                long slotAlgMinIntervalSec =
Integer.valueOf(DBGetter.getSettingsValue(db,
DbSetting.SLOT_ALG_MIN_INTERVAL_SEC));
                if (start -
TimeUnit.SECONDS.toMillis(slotAlgMinIntervalSec) <
```

```
lastRunningTime) {
                    log.info("Last running time: " +
df.format(lastRunningTime) + ", now: " + df.format(start) +
", returning last calculated results");

                    return lastRunningResult;
                }
            }
            log.finest("testEthOnly DBConnect time: " +
(System.currentTimeMillis() - start) + ", days: " +
daysAgo);

            List<CardFailAlgOutput2> toReturn = new
ArrayList<>();
            Map<DslamCard, List<CfdRadiusAcct>> slotMap =
new HashMap<>();
            downloadDataAndInsertInMapByCard(db, slotMap,
daysAgo, gui);
            long analyzeStart = System.currentTimeMillis();
            for (Map.Entry<DslamCard, List<CfdRadiusAcct>>
entry : slotMap.entrySet()) {
                DslamCard dslamCard = entry.getKey();
                List<CfdRadiusAcct> cardList =
entry.getValue();


removeSessionRefreshAndRepairSwitched(cardList);
                testCard(cardList, dslamCard, toReturn, db);
            }

            filterResults(gui, toReturn);
            log.finest("Analyze time: " +
(System.currentTimeMillis() - analyzeStart));

            setLastRunningTimeData(toReturn);
            return toReturn;
        } finally {
            log.finest("testEth time: " +
(System.currentTimeMillis() - start));
        }
    }

    private void
setLastRunningTimeData(List<CardFailAlgOutput2> toReturn) {
        lastRunningTime = System.currentTimeMillis();
        lastRunningResult = toReturn;
    }

    private int getStopsNr(List<CfdRadiusAcct> list) {
        int toReturn = 0;
        Iterator<CfdRadiusAcct> it = list.iterator();
```

```
        CfdRadiusAcct firstAcct = null;
        long _24hMillis = TimeUnit.HOURS.toMillis(24);
        while (it.hasNext()) {
            CfdRadiusAcct acct = it.next();
            if (firstAcct == null) {
                firstAcct = acct;
                continue;
            } else if (acct.getTimestamp().getTime() -
firstAcct.getTimestamp().getTime() >= _24hMillis) {
                if (acct.getType().equals("Stop")) {
                    ++toReturn;
                }
            }
        }
        return toReturn;
    }

    private void testCard(List<CfdRadiusAcct> cardList,
DslamCard dslamCard, List<CardFailAlgOutput2> toReturn,
DBConnect db) throws SQLException {
        long _24hMillis = TimeUnit.HOURS.toMillis(24);
        CfdRadiusAcct firstAcct = null;
        Set<Integer> activePorts = new TreeSet<>(); //this
will be different for different scopes
        for (int i = 0; i < cardList.size(); ++i) {
            CfdRadiusAcct currentPointAcct =
cardList.get(i);
            boolean updateInFinally = true;
            try {
                if (firstAcct == null) {
                    firstAcct = currentPointAcct;
                    continue;
                } else if
(currentPointAcct.getTimestamp().getTime() -
firstAcct.getTimestamp().getTime() < _24hMillis) {
                    continue;
                }

                if
(!currentPointAcct.getType().equals("Stop")) {
                    continue;
                }

                CfdRadiusAcct firstStopAcct =
currentPointAcct;

                Set<Integer> activePortsInStopWindow = new
TreeSet<>(activePorts);
                StopsLookupResult stopLookupResult =
lookupForStopsInWindow(i, cardList, firstStopAcct,
activePortsInStopWindow);
```

```
                if (stopLookupResult.getLastAcctStop() ==
null || activePorts.size() < MINIMUM_ACTIVE_PORTS) {
                    continue;
                }

                int stopsInWindow =
stopLookupResult.getStoppedPorts().size(); //to have unique
this per port
                double percentageStops = new
BigDecimal((double) stopsInWindow / (double)
activePortsInStopWindow.size()).setScale(2,
RoundingMode.HALF_UP).doubleValue();

                if (percentageStops >= MIN_PERCENT_STOPS) {
                    lookupForStartsAndBuildVerdict(i,
cardList, stopLookupResult, dslamCard, db, firstStopAcct,
toReturn, percentageStops, activePortsInStopWindow);


updateActivePortsAndJump(stopLookupResult, i, cardList,
activePorts);

                    i = stopLookupResult.getLastIndex();
                    updateInFinally = false;

                }
            } finally {
                if (updateInFinally) {
                    updateActivePortsSet(activePorts,
currentPointAcct);
                }
            }
        }
    }

    private int updateActivePortsAndJump(StopsLookupResult
stopLookupResult, int i, List<CfdRadiusAcct> cardList,
Set<Integer> activePorts) {
        int lastIndexToLookup =
stopLookupResult.getLastIndex();
        for (; i <= lastIndexToLookup; ++i) {
            CfdRadiusAcct acct = cardList.get(i);
            updateActivePortsSet(activePorts, acct);
        }
        --i;
        return i;
    }

    public static void updateActivePortsSet(Set<Integer>
activePorts, CfdRadiusAcct currentPointAcct) {
```

```
        String type = currentPointAcct.getType();
        switch (type) {
            case "Start":
                activePorts.add(currentPointAcct.getPort());
            case "Alive":
                activePorts.add(currentPointAcct.getPort());
                break;
            case "Stop":

activePorts.remove(currentPointAcct.getPort());
                break;
            default:
                break;
        }
    }

    private void lookupForStartsAndBuildVerdict(int i,
List<CfdRadiusAcct> cardList,

StopsLookupResult stopLookupResult, DslamCard dslamCard,
DBConnect db, CfdRadiusAcct firstStopAcct,

List<CardFailAlgOutput2> toReturn, double percentageStops,
Set<Integer> activePortsInStopWindow) throws SQLException {
        Set<Integer> stoppedPorts =
stopLookupResult.getStoppedPorts();

        Set<Integer> startedPorts = new TreeSet<>();
        List<CfdRadiusAcct> acctInStartWindow = new
ArrayList<>();
        CfdRadiusAcct firstStartInWindow = null;
        CfdRadiusAcct lastStartInWindow = null;
        Set<String> brases = new HashSet<>();

        for (int j = i; j < cardList.size(); ++j) {
            CfdRadiusAcct acct = cardList.get(j);
            if (acct.getType().equals("Start")) {
                if (firstStartInWindow == null) {
                    firstStartInWindow = acct;
                }

                if (acct.getTimestamp().getTime() -
firstStartInWindow.getTimestamp().getTime() <=
STARTS_LOOKUP_MAX_WINDOW_MS) {
                    acctInStartWindow.add(acct);
                    startedPorts.add(acct.getPort());

                    lastStartInWindow = acct;

                    if (startedPorts.equals(stoppedPorts)) {
```

```
                        break;
                    }
                } else {
                    break;
                }
            }
        }
        int startsInWindow = startedPorts.size();
        String verdict = "disabled";
        String dslamIp = dslamCard.getDslamIp();
        String dslamName = null;
        String m1400 = null;
        String dslamType = null;
        String select = "SELECT * FROM
`fault_detector`.`ksp_device_info` WHERE `DSLAM_IP`='" +
dslamIp + "' LIMIT 1";

        try (Statement s =
db.getConnection().createStatement(); ResultSet dslam =
s.executeQuery(select)) {
            if (dslam.next()) {
                dslamName = dslam.getString("DSLAM_NAME");
                m1400 = dslam.getString("M1400_SUFIKS");
                dslamType = dslam.getString("DSLAM_TYPE");
            }
        }

        Set<Integer> notStartedPorts =
buildNotStartedPorts(startedPorts, stoppedPorts);

        CardFailAlgOutput2 out = new CardFailAlgOutput2();
        out.setDslamIp(dslamIp);
        out.setDslamName(dslamName);
        out.setDslamType(dslamType);
        out.setM1400(m1400);
        out.setDslamShelf(dslamCard.getShelf());
        out.setDslamSlot(dslamCard.getSlot());

out.setFirstStopTimestamp(firstStopAcct.getTimestamp().getTi
me());

out.setLastStopTimestamp(stopLookupResult.getLastAcctStop().
getTimestamp().getTime());

out.setStopsInWindow(stopLookupResult.getStoppedPorts().size
());
        out.setPercentageStops(percentageStops);
        out.setActivePortsBefore(new
TreeSet<>(activePortsInStopWindow));

out.setActivePortsBeforeSize(activePortsInStopWindow.size())
```

```
;
        out.setStartedPorts(startedPorts);
        out.setStoppedPorts(stoppedPorts);
        out.setNotStartedPorts(notStartedPorts);
        out.setNotStartedPortsSize(notStartedPorts.size());
        out.setBrases(brases);

        if (firstStartInWindow != null) {

out.setFirstStartTimestamp(firstStartInWindow.getTimestamp()
.getTime());
        }
        if (lastStartInWindow != null) {

out.setLastStartTimestamp(lastStartInWindow.getTimestamp().g
etTime());
        }
        out.setStartsInWindow(startsInWindow);
        out.setVerdict(verdict);

out.setAcctInStopWindow(stopLookupResult.getAcctInStopWindow
());
        if (!acctInStartWindow.isEmpty()) {
            out.setAcctInStartWindow(acctInStartWindow);
        }

        toReturn.add(out);
    }

    private StopsLookupResult lookupForStopsInWindow(int i,
List<CfdRadiusAcct> cardList, CfdRadiusAcct firstStopAcct,
Set<Integer> activePortsInStopWindow) {
        List<CfdRadiusAcct> acctInStopWindow = new
ArrayList<>();
        Set<Integer> stoppedPorts = new TreeSet<>();
        CfdRadiusAcct lastStopAcctInWindow = null;
        int j = i;
        int lastStopIndex = j;
        OUTER:
        for (; j < cardList.size(); ++j) {
            CfdRadiusAcct acct = cardList.get(j);
            if (acct.getTimestamp().getTime() -
firstStopAcct.getTimestamp().getTime() <= STOP_WINDOW_MS) {
                String type = acct.getType();
                if (type.equals("Stop")) {
                    stoppedPorts.add(acct.getPort());
                    acctInStopWindow.add(acct);
                    lastStopAcctInWindow = acct;
                    lastStopIndex = j;
                } else {
```

```
updateActivePortsSet(activePortsInStopWindow, acct);
                    }
                } else {
                    break;
                }
            }
            return new StopsLookupResult(lastStopAcctInWindow,
lastStopIndex, stoppedPorts, acctInStopWindow);
        }

        private void
removeSessionRefreshAndRepairSwitched(List<CfdRadiusAcct>
cardList) {
            Map<DslamPort, List<CfdRadiusAcct>> portMap =
buildPortMap(cardList);
            List<CfdRadiusAcct> sessionRefresh =
buildSessionRefreshList(portMap);
            cardList.removeAll(sessionRefresh);
        }

        private List<CfdRadiusAcct>
buildSessionRefreshList(Map<DslamPort, List<CfdRadiusAcct>>
portMap) {
            List<CfdRadiusAcct> toRemove = new ArrayList<>();

            for (Map.Entry<DslamPort, List<CfdRadiusAcct>>
entry1 : portMap.entrySet()) {
                List<CfdRadiusAcct> portList =
entry1.getValue();

                for (int i = 1; i < portList.size() - 1; ++i) {
                    CfdRadiusAcct beforeAcct = portList.get(i -
1);
                    CfdRadiusAcct middleAcct = portList.get(i);
                    CfdRadiusAcct nextAcct = portList.get(i +
1);
                    if (beforeAcct.getType().equals("Start")
                            &&
middleAcct.getType().equals("Stop") &&
nextAcct.getType().equals("Start")) {
                        long sessionTime =
middleAcct.getTimestamp().getTime() -
beforeAcct.getTimestamp().getTime();
                        long refreshTime =
nextAcct.getTimestamp().getTime() -
middleAcct.getTimestamp().getTime();
                        if (sessionTime >=
MIN_SESSION_UP_TIME_MS && refreshTime <=
MAX_SESSION_REFRESH_TIME_MS) {
                            toRemove.add(middleAcct);
                            toRemove.add(nextAcct);
```

```
                        i += 2;
                    }
                }
            }
        }
        return toRemove;
    }

    private long calculateAvgTime(List<CfdRadiusAcct> list)
{
        BigDecimal avg = BigDecimal.ZERO;
        int i;
        for (i = 0; i < list.size(); ++i) {
            avg =
avg.add(BigDecimal.valueOf(list.get(i).getTimestamp().getTim
e()));
        }
        avg = avg.divide(BigDecimal.valueOf(i), 0,
RoundingMode.HALF_UP);

        return avg.longValue();
    }

    private void filterResultsGui(List<CardFailAlgOutput2>
list) {
        Iterator<CardFailAlgOutput2> it = list.iterator();
        long currentTime = System.currentTimeMillis();
        long minimumTimeBetweenStopStart =
TimeUnit.MINUTES.toMillis(10);
        long minimumTimeBetweenStopStartNight =
TimeUnit.MINUTES.toMillis(2 * 60);
        int minHourOfDayNight = 0;
        int maxHourOfDayNight = 5;

        while (it.hasNext()) {
            CardFailAlgOutput2 out = it.next();
            long firstStop =
out.getFirstStopTimestampLong();
            long firstStart =
out.getFirstStartTimestampLong();
            int starts = out.getStartsInWindow();
            Calendar calendar = Calendar.getInstance();
            calendar.setTimeInMillis(firstStop);
            int hourOfDay =
calendar.get(Calendar.HOUR_OF_DAY);

            if (currentTime - firstStop <
minimumTimeBetweenStopStart) {
                it.remove();
            } else if (starts > 0) {
                if (firstStart - firstStop <
```

```
minimumTimeBetweenStopStart) {
                    it.remove();
                } else if (hourOfDay >= minHourOfDayNight &&
hourOfDay <= maxHourOfDayNight && firstStart - firstStop <
minimumTimeBetweenStopStartNight) {
                    it.remove();
                }
            } else if (currentTime - firstStop >
GUI_HOURS_DISPLAYED_MS) {
                it.remove();
            }
        }
    }

    private void
filterResultsOnlyNight(List<CardFailAlgOutput2> list) {
        Iterator<CardFailAlgOutput2> it = list.iterator();
        long minimumTimeBetweenStopStartNight =
TimeUnit.MINUTES.toMillis(2 * 60);
        int minHourOfDayNight = 0;
        int maxHourOfDayNight = 5;

        while (it.hasNext()) {
            CardFailAlgOutput2 out = it.next();
            long firstStop =
out.getFirstStopTimestampLong();
            long firstStart =
out.getFirstStartTimestampLong();
            int starts = out.getStartsInWindow();
            Calendar calendar = Calendar.getInstance();
            calendar.setTimeInMillis(firstStop);
            int hourOfDay =
calendar.get(Calendar.HOUR_OF_DAY);
            if (starts > 0) {
                if (hourOfDay >= minHourOfDayNight &&
hourOfDay <= maxHourOfDayNight && firstStart - firstStop <
minimumTimeBetweenStopStartNight) {
                    it.remove();
                }
            }
        }
    }

    public String getLastRunningTime() {
        return df.format(lastRunningTime);
    }
}

class StopsLookupResult {

    private CfdRadiusAcct lastAcctStop;
```

```
    private int lastIndex;
    private Set<Integer> stoppedPorts;
    private List<CfdRadiusAcct> acctInStopWindow;

    public StopsLookupResult(CfdRadiusAcct lastAcctStop, int
lastIndex, Set<Integer> stoppedPorts, List<CfdRadiusAcct>
acctInStopWindow) {
        this.lastAcctStop = lastAcctStop;
        this.lastIndex = lastIndex;
        this.stoppedPorts = stoppedPorts;
        this.acctInStopWindow = acctInStopWindow;
    }

    public List<CfdRadiusAcct> getAcctInStopWindow() {
        return acctInStopWindow;
    }

    public void setAcctInStopWindow(List<CfdRadiusAcct>
acctInStopWindow) {
        this.acctInStopWindow = acctInStopWindow;
    }

    public CfdRadiusAcct getLastAcctStop() {
        return lastAcctStop;
    }

    public void setLastAcctStop(CfdRadiusAcct lastAcctStop)
{
        this.lastAcctStop = lastAcctStop;
    }

    public int getLastIndex() {
        return lastIndex;
    }

    public void setLastIndex(int lastIndex) {
        this.lastIndex = lastIndex;
    }

    public Set<Integer> getStoppedPorts() {
        return stoppedPorts;
    }

    public void setStoppedPorts(Set<Integer> stoppedPorts) {
        this.stoppedPorts = stoppedPorts;
    }
}
```

1.2. Obiekt CfdRadiusAcct (reprezentacja logu Radius Accounting):

```
import java.sql.Timestamp;
import java.util.Objects;

public class CfdRadiusAcct {

    private final Timestamp timestamp;
    private final int rack;
    private final int shelf;
    private final int slot;
    private final int port;
    private final String dslamIp;
    private final String type;
    private final RadiusAcctLog.AcctType acctType;

    public CfdRadiusAcct(Timestamp timestamp, int rack, int
shelf, int slot, int port, String dslamIp, String type,
RadiusAcctLog.AcctType acctType) {
        this.timestamp = timestamp;
        this.rack = rack;
        this.shelf = shelf;
        this.slot = slot;
        this.port = port;
        this.dslamIp = dslamIp;
        this.type = type;
        this.acctType = acctType;
    }

    public RadiusAcctLog.AcctType getAcctType() {
        return acctType;
    }

    public Timestamp getTimestamp() {
        return timestamp;
    }

    public int getPort() {
        return port;
    }

    public String getDslamIp() {
        return dslamIp;
    }

    public int getShelf() {
        return shelf;
    }

    public int getSlot() {
        return slot;
```

```
    }

    public int getRack() {
        return rack;
    }

    public String getType() {
        return type;
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 17 * hash + Objects.hashCode(this.timestamp);
        hash = 17 * hash + this.rack;
        hash = 17 * hash + this.shelf;
        hash = 17 * hash + this.slot;
        hash = 17 * hash + this.port;
        hash = 17 * hash + Objects.hashCode(this.dslamIp);
        hash = 17 * hash + Objects.hashCode(this.type);
        hash = 17 * hash + Objects.hashCode(this.acctType);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final CfdRadiusAcct other = (CfdRadiusAcct) obj;
        if (!Objects.equals(this.timestamp,
other.timestamp)) {
            return false;
        }
        if (this.rack != other.rack) {
            return false;
        }
        if (this.shelf != other.shelf) {
            return false;
        }
        if (this.slot != other.slot) {
            return false;
        }
        if (this.port != other.port) {
            return false;
        }
        if (!Objects.equals(this.dslamIp, other.dslamIp)) {
            return false;
```

```
        }
        if (!Objects.equals(this.type, other.type)) {
            return false;
        }
        return this.acctType == other.acctType;
    }

    @Override
    public String toString() {
        return "CfdRadiusAcct{" + "timestamp=" + timestamp +
", rack=" + rack + ", shelf=" + shelf + ", slot=" + slot +
", port=" + port + ", dslamIp=" + dslamIp + ", type=" + type
+ ", acctType=" + acctType + '}';
    }

}
```

1.3. Obiekt DslamCard (reprezentacja karty urządzenia DSLAM):

```
import java.io.Serializable;
import java.util.Objects;

public class DslamCard implements Serializable {

    private String dslamIp;
    private int rack;
    private int shelf;
    private int slot;

    public DslamCard(String dslamIp, int rack, int shelf,
int slot) {
        this.dslamIp = dslamIp;
        this.rack = rack;
        this.shelf = shelf;
        this.slot = slot;
    }

    public String getDslamIp() {
        return dslamIp;
    }

    public void setDslamIp(String dslamIp) {
        this.dslamIp = dslamIp;
    }

    public int getRack() {
        return rack;
    }

    public void setRack(int rack) {
        this.rack = rack;
```

```
}

public int getShelf() {
    return shelf;
}

public void setShelf(int shelf) {
    this.shelf = shelf;
}

public int getSlot() {
    return slot;
}

public void setSlot(int slot) {
    this.slot = slot;
}

@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final DslamCard other = (DslamCard) obj;
    if (!Objects.equals(this.dslamIp, other.dslamIp)) {
        return false;
    }
    if (this.rack != other.rack) {
        return false;
    }
    if (this.shelf != other.shelf) {
        return false;
    }
    if (this.slot != other.slot) {
        return false;
    }
    return true;
}

@Override
public int hashCode() {
    int hash = 7;
    hash = 29 * hash + Objects.hashCode(this.dslamIp);
    hash = 29 * hash + this.rack;
    hash = 29 * hash + this.shelf;
    hash = 29 * hash + this.slot;
    return hash;
}
```

```
    @Override
    public String toString() {
        return "DslamCard{" + "dslamIp=" + dslamIp + ",
rack=" + rack + ", shelf=" + shelf + ", slot=" + slot + '}';
    }

}
```

1.4. Obiekt CardFailAlgOutput (obiekt do wyświetlenia rekordu zdarzenia na GUI użytkownika):

```
2. import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Iterator;
import java.util.List;
import java.util.Locale;
import java.util.Set;

public class CardFailAlgOutput {

    private static final DateFormat df = new
SimpleDateFormat("yyyy-MM-dd HH:mm:ss", Locale.UK);
    private String dslamIp;
    private String dslamName;
    private String m1400;
    private int dslamShelf;
    private int dslamSlot;
    private long firstStopTimestamp;
    private long lastStopTimestamp;
    private int stopsInWindow;
    private long firstStartTimestamp = -1;
    private long lastStartTimestamp = -1;
    private int startsInWindow;
    private List<CfdRadiusAcct> acctInStopWindow;
    private List<CfdRadiusAcct> acctInStartWindow;
    private Set<Integer> stoppedPorts;
    private Set<Integer> startedPorts;
    private String verdict;

    public CardFailAlgOutput() {
    }

    public CardFailAlgOutput(String dslamIp, String
dslamName, String m1400, int dslamShelf, int dslamSlot, long
firstStopTimestamp, long lastStopTimestamp,
                             int stopsInWindow, int
startsInWindow, String verdict, Set<Integer> stoppedPorts,
Set<Integer> startedPorts) {
```

```
        this.dslamIp = dslamIp;
        this.dslamName = dslamName;
        this.m1400 = m1400;
        this.dslamShelf = dslamShelf;
        this.dslamSlot = dslamSlot;
        this.firstStopTimestamp = firstStopTimestamp;
        this.lastStopTimestamp = lastStopTimestamp;
        this.stopsInWindow = stopsInWindow;
        this.startsInWindow = startsInWindow;
        this.verdict = verdict;
        this.stoppedPorts = stoppedPorts;
        this.startedPorts = startedPorts;
    }

    public String getDslamIp() {
        return dslamIp;
    }

    public void setDslamIp(String dslamIp) {
        this.dslamIp = dslamIp;
    }

    public String getDslamName() {
        return dslamName;
    }

    public void setDslamName(String dslamName) {
        this.dslamName = dslamName;
    }

    public String getFirstStartTimestamp() {
        return formatTime(firstStartTimestamp);
    }

    public long getFirstStartTimestampLong() {
        return firstStartTimestamp;
    }

    public void setFirstStartTimestamp(long
firstStartTimestamp) {
        this.firstStartTimestamp = firstStartTimestamp;
    }

    public String getFirstStopTimestamp() {
        return formatTime(firstStopTimestamp);
    }

    public long getFirstStopTimestampLong() {
        return firstStopTimestamp;
    }
```

```
    public void setFirstStopTimestamp(long
firstStopTimestamp) {
        this.firstStopTimestamp = firstStopTimestamp;
    }

    public String getLastStartTimestamp() {
        return formatTime(lastStartTimestamp);
    }

    public void setLastStartTimestamp(long
lastStartTimestamp) {
        this.lastStartTimestamp = lastStartTimestamp;
    }

    public String getLastStopTimestamp() {
        return formatTime(lastStopTimestamp);
    }

    public void setLastStopTimestamp(long lastStopTimestamp)
{
        this.lastStopTimestamp = lastStopTimestamp;
    }

    public String getM1400() {
        return m1400;
    }

    public void setM1400(String m1400) {
        this.m1400 = m1400;
    }

    public int getDslamShelf() {
        return dslamShelf;
    }

    public void setDslamShelf(int dslamShelf) {
        this.dslamShelf = dslamShelf;
    }

    public int getDslamSlot() {
        return dslamSlot;
    }

    public void setDslamSlot(int dslamSlot) {
        this.dslamSlot = dslamSlot;
    }

    public int getStartsInWindow() {
        return startsInWindow;
    }
```

```
    public void setStartsInWindow(int startsInWindow) {
        this.startsInWindow = startsInWindow;
    }

    public int getStopsInWindow() {
        return stopsInWindow;
    }

    public void setStopsInWindow(int stopsInWindow) {
        this.stopsInWindow = stopsInWindow;
    }

    public String getVerdict() {
        return verdict;
    }

    public void setVerdict(String verdict) {
        this.verdict = verdict;
    }

    private String formatTime(long time) {
        if (time > 0) {
            return df.format(time);
        } else {
            return null;
        }
    }

    public List<CfdRadiusAcct> getAcctInStartWindow() {
        return acctInStartWindow;
    }

    public void setAcctInStartWindow(List<CfdRadiusAcct>
acctInStartWindow) {
        this.acctInStartWindow = acctInStartWindow;
    }

    public List<CfdRadiusAcct> getAcctInStopWindow() {
        return acctInStopWindow;
    }

    public void setAcctInStopWindow(List<CfdRadiusAcct>
acctInStopWindow) {
        this.acctInStopWindow = acctInStopWindow;
    }

    public Set<Integer> getStartedPorts() {
        return startedPorts;
    }

    public String getStartedPortsStr() {
```

```
        return setToString(startedPorts, "; ");
    }

    public void setStartedPorts(Set<Integer> startedPorts) {
        this.startedPorts = startedPorts;
    }

    public String getStoppedPortsStr() {
        return setToString(stoppedPorts, "; ");
    }

    public Set<Integer> getStoppedPorts() {
        return stoppedPorts;
    }

    public void setStoppedPorts(Set<Integer> stoppedPorts) {
        this.stoppedPorts = stoppedPorts;
    }

    private String setToString(Set in, String split) {
        Iterator it = in.iterator();
        String toReturn = "";
        while (it.hasNext()) {
            Object o = it.next();
            toReturn += o.toString() + split;
        }
        if (!toReturn.isEmpty()) {
            toReturn = toReturn.substring(0,
toReturn.length() - split.length());
        }
        return toReturn;
    }

    public String toCsvString() {
        return dslamIp + "," + dslamName + "," + m1400 + ","
+ dslamShelf + "," + dslamSlot + "," +
df.format(firstStopTimestamp) + "," +
df.format(lastStopTimestamp)
                + "," + stopsInWindow + "," +
setToString(stoppedPorts, ";") + "," + (firstStartTimestamp
!= -1 ? df.format(firstStartTimestamp) : null)
                + "," + (lastStartTimestamp != -1 ?
df.format(lastStartTimestamp) : null) + "," + startsInWindow
+ "," + setToString(stoppedPorts, ";") + "," + verdict;
    }
}
```