Krzysztof ŚWIDRAK*

* Wojskowa Akademia Techniczna, Wydział Elektroniki, Eksploatacja Systemów Łączności
ul. W. Urbanowicza 2, 01-476 Warszawa
tel.: +48 667409329, e-mail: krzysztof.swidrak@student.wat.edu.pl

PROCES TWORZENIA MODELU W ŚRODOWISKU OMNET++ NA PRZYKŁADZIE KLIENTA PROTOKOŁU SIP

1. Wstęp

Obecnie szerzy się trend do wirtualizacji i symulacji zarówno danych jak również zdarzeń rzeczywistych. Począwszy od symulacji ruchu drogowego, poprzez różne modele fizyczne w tym zachowania ciał niebieskich z uwzględnieniem reakcji na poziomie świata kwantowego (np. promieniowanie Hawkinga). Trendowi temu ulega także przemysł sieciowy. Odwołując się do prawa Moore'a przybywa coraz więcej urządzeń, które w coraz większym stopniu mają udostępniać Internet człowiekowi.

Niniejszy artykuł stanowi o procesie tworzenia modeli symulacyjnych i symulacji w środowisku OMNeT++.

Pierwszy rozdział stanowi wstęp oraz określa porządek procesu. Drugi rozdział prezentuje główną tematykę i opisane w nim są: pierwszy etap - zapoznanie z problemem oraz syntezę założeń dotyczących rozwiązania danego problemu. Następnie, przedstawiono praktyczną realizacje projektu. Zwieńczeniem procesu tworzenia jest przeprowadzenie symulacji weryfikacyjnych i generacja wniosków.

Zwrócić należy uwagę, że cały proces w rzeczywistości scala w sobie pracę trzech osób: projektanta oprogramowania, programisty oraz testera programów.

Artykuł jest bezpośrednim odniesieniem do pracy dyplomowej: „Opracowanie modelu klienta SIP dla środowiska OMNeT++." realizowanej w latach 2017/2018 na Wojskowej Akademii Technicznej pod opieką i nadzorem pana mjr. dr. Inż. Jerzego Dołowskiego.

2. Rozwinięcie

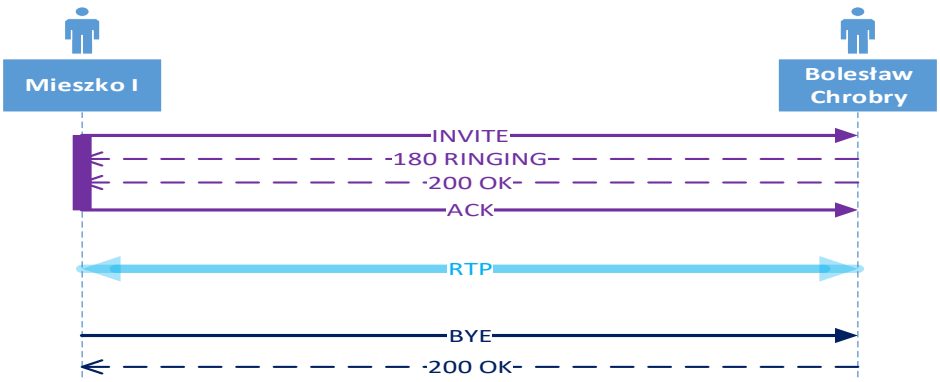
2.1. Zapoznanie z problematyką

Istnieje powiedzenie: „Potrzeba matką wynalazków”, w myśl którego początkiem każdego wynalazku, każdego procesu tworzenia winna być potrzeba a co za nią idzie zapoznanie z problemem jaki należy rozwiązać i określenie planu działania (synteza założeń).

2.1.1. Środowisko OMNeT++ i Session Initiation Protocol

Zapoznając się z problematyką, należy poznać środowisko w jakim problem ma zostać rozwiązany. W niniejszym artykule środowiskiem jest OMNeT++. Jest to symulator zdarzeń dyskretnych . Jego głównym przeznaczeniem jest symulacja procesów zachodzących w sieciach głównie komputerowych. IDE stanowi Eclipse poszerzony o nakładkę OMNeT++. W celu odwzorowania jak najbardziej rzeczywistych zachowań sieci wykorzystywana jest biblioteka INET. Zawiera ona w sobie modele wielu protokołów sieciowych oraz urządzeń obsługujących je (np. standardowy host który obsługuje protokoły IP zarówno v4 i v6, TCP, UDP, ARP itp.).

Problemem jest stworzenie modelu symulacyjnego klienta protokołu zarządzania sesją multimedialną (Session Inintiation Protocol) usytuowanego w warstwie Aplikacji modelu TCP/IP. Może on wykorzystywać wiele protokołów transportowych, jednakże w związku z tym, że sieci multimedialne w celu zapewnienia QoS najczęściej wykorzystują UDP, w niniejszej implementacji klient SIP korzysta z socketu UDP, co związane jest z koniecznością implementacji odpowiednich mechanizmów w celu zapewnienia niezawodności na poziomie SIP (retransmisje). Komunikacja w SIP oparta jest na bazie klient-serwer, jednakże budowa logiczna klientów pozwala na tworzenie rozproszonych rozwiązań i połączeń bezpośrednich bez udziału serwerów SIP Proxy. Protokół SIP określa ŻĄDANIA (inaczej METODY) oraz **Odpowiedzi** (**xxx Responses**). Przykładowa sesja, stanowiąca podstawę połączeń SIP przedstawiona jest na rysunku 1. Więcej informacji o SIP oraz zasadach działania protokołu opisane jest w normie [2] a także w pracy dyplomowej [1].



Rys. 1. Podstawowa sesja SIP

2.1.2. Przyjęcie założeń

W związku z tym, że SIP jest obszernie rozbudowanym protokołem (liczba norm odnoszących się do niego obecnie przekracza 500), narzucone zostały odpowiednie ograniczenia. Założenia nie ograniczają funkcjonalności protokołu w kwestii podstawowej zarządzania sesją. Co więcej taka synteza oraz obiektowość OMNeT++ opartego na C++ umożliwia łatwą rozbudowę modelu o dodatkowe funkcje. Oto najważniejsze z założeń:

1. Klient stanowi agenta użytkownika (UA), który obsługuje metody proste:
    a. INVITE,
    b. BYE,
    c. CANCEL,
    d. ACK.
2. Serwery SIP nie są modelowane.
3. Zachowanie klientów umożliwia zestawienie jednego połączenia z udziałem danych użytkowników w tym samym czasie, pozostałe połączenia przychodzące są odrzucane (**486 Busy Here**).
4. Dla danego urządzenia prowadzący badanie określa czas symulacyjny początku i końca połączenia oraz adres SIP urządzenia do którego kierowane jest połączenie.
5. Klient reaguje na błędnie podane parametry symulacji.
6. Sieć złożona może być z dowolnej liczby klientów, którzy realizować mogą dowolną liczbę połączeń.
7. Strumieniowanie RTP nie jest realizowane.
8. Zasoby przetwarzania dźwięku (w negocjacji) może być zmieniany przez prowadzącego. W razie niezgodności odpowiedź **406 Not Acceptable**.
9. Brak dodatkowych usług (gorąca linia, przekierowania, usługi bezpieczeństwa).
10. Wykorzystanie uproszczonego protokołu SDP przez SIP.
11. Uproszczona struktura wiadomości SIP [Rys. 1].

```
INVITE sip:user1@10.0.0.1:53001 SIP/2.0
Via: SIP/2.0/UDP 10.0.0.2:5060;
branch=1sdasxsadxsa
CSeq: 1 INVITE
Call-ID: 1qaz2wsx3EDC
To: UserName1 <sip:user1@10.0.0.1>
From: UserName2 <sip:user2@10.0.0.2>
Max-Forwards: 10
Content-Type: application/SDP
Contetn-Length: 44

v=0
m=audio 53001 RTP 1
a=rtp: 1 PCMU/8000
```

Rys. 2. Wiadomość SIP (Żądanie INVITE) modelu klienta SIP

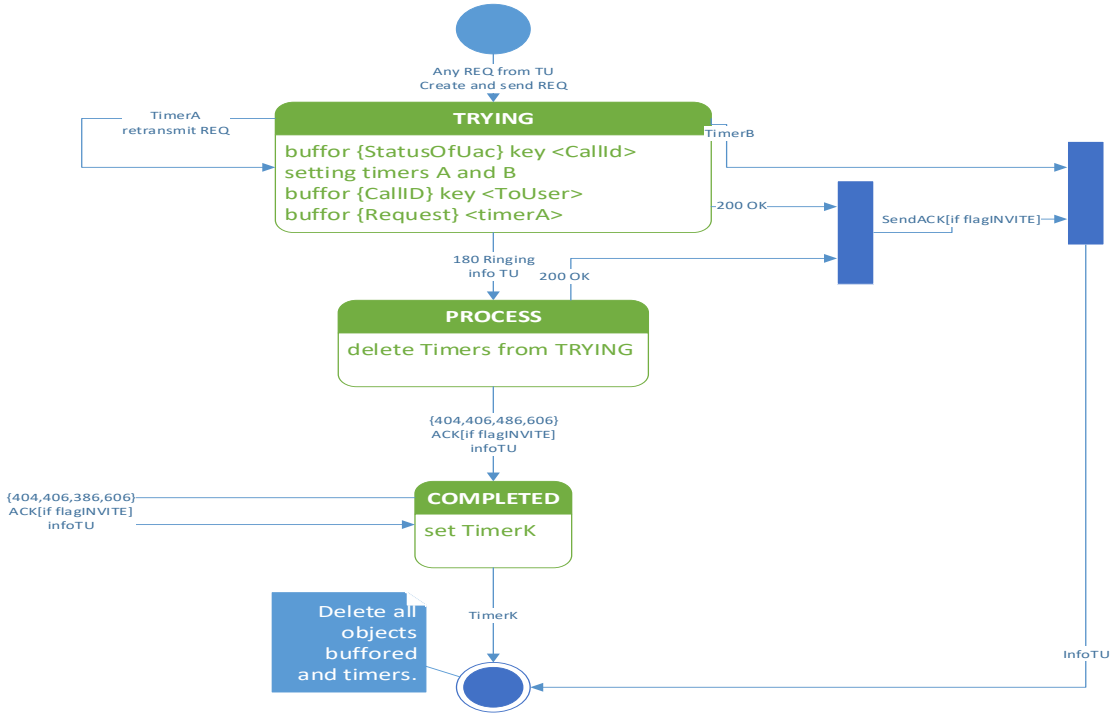
12. Ograniczony zbiór odpowiedzi:
    a. **180 Ringing,**
    b. **200 OK,**
    c. **406 Not Acceptable,**
    d. **606 Not Acceptable,**
    e. **404 Not Found,**
    f. **486 Busy Here.**
13. Ograniczenie zbioru timerów zgodnie z tabelą:

TIMERY MODELU KLIENTA SIP		
Nazwa	**Wartość domyślna**	**Znaczenie**
T1	0,5 s	Oszacowany RTT.
T2	4s	Maksymalny możliwy odstęp pomiędzy retransmisją INVITE.
T4	5s	Maksymalny okres, podczas którego wiadomość SIP może przebywać w sieci.
Timer A	T1	Odstęp pomiędzy retransmisją żądań z wykluczeniem ACK.
Timer B	64·T1	Maksymalny okres nawiązywania transakcji.
Timer K	T4	Okres oczekiwania na retransmisję odpowiedzi.

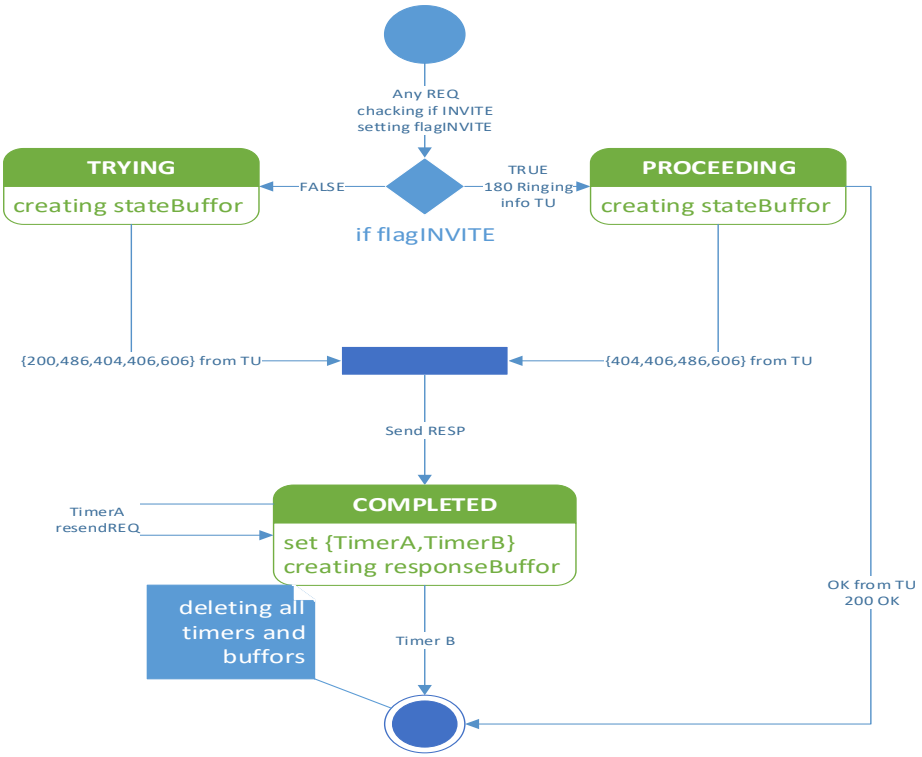
14. Wyposażenie klienta w statystyki liczące wysyłane i odbierane wiadomości każdego rodzaju oraz informacje o przyjmowaniu odpowiednich stanów przez klientów.
15. Logowanie wysyłanych wiadomości przez odpowiednie moduły silnika SIP do plików.

2.1.3. Maszyny stanów klienta

Na podstawie maszyn stanów z [2] oraz opracowanych założeń zaproponowano następujące maszyny stanów dla klienta SIP [Rys. 2., Rys. 3.].



Rys. 3. Maszyna stanów UAC modelu klienta SIP



Rys. 4. Maszyna stanów UAS modelu klienta SIP

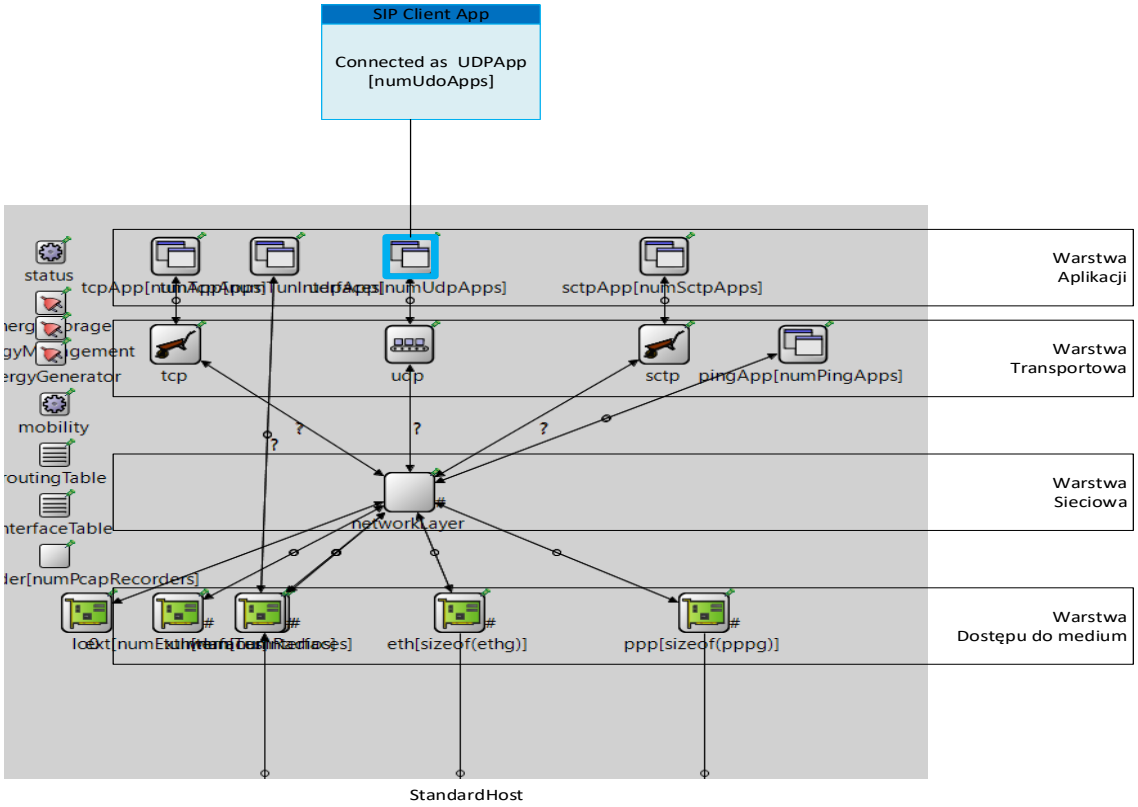
W celu przygotowania elementów silnika protokołu klienta SIP na obsługę wielu sesji utworzone są specjalne obiekty przechowujące stan oraz dane każdego połączenia. Każdy stan skorelowany jest z sesją poprzez odpowiednie parametry identyfikujące. Dzięki temu każdy moduł jest w stanie przebywać w wielowymiarowym stanie, który tak naprawdę stanowi zbiór stanów poszczególnych sesji. Zgodnie jednak z założeniami UAC może zainicjować tylko jedną sesję w określonym momencie czasu. Jednakże powyższe podejście umożliwia przyszłą rozbudowę funkcjonalności klienta. UAS jest w stanie obsługiwać wiele połączeń przychodzących.

2.2. Praktyczna realizacja modelu klienta SIP

Model klienta został stworzony przy wykorzystaniu wersji OMNeT++ 5.2.1 oraz modelu INET 3.6.3.

2.2.1. Koncepcja modelu klienta SIP.

Klient SIP stanowi aplikację, której socket binduje do UDP. W związku z tym klient stanowi element standardowego hosta biblioteki INET [Rys. 4.].



Rys. 5. Model klienta SIP w Standard Host

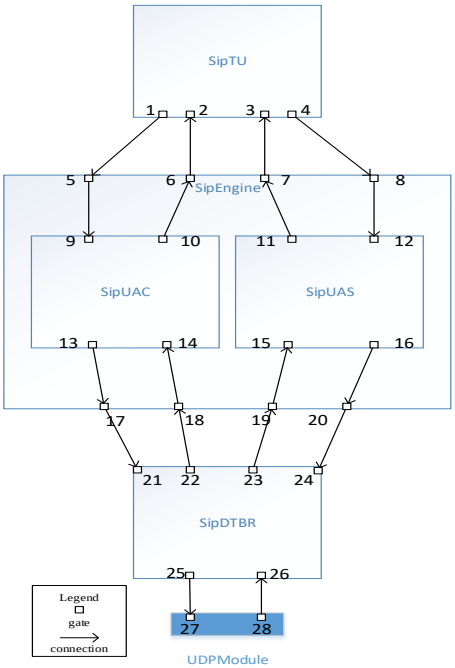
Klient złożony jest z trzech modułów:

- **SipEngine** – silnika SIP, który przetwarza i zarządza wiadomościami SIP. Składa się on z dwóch podmodułów **SipUAC** i **SipUAS**. Moduły te zapewniają funkcjonowanie klienta w połączeniach typu peer2peer. Jednakże należy podkreślić, że lokalny server agenta klienta (UA- User Agent odpowiednio S-Server, C-Client) nie powinien być

utożsamiany z serwerami globalnymi (SipProxy) ze względu na funkcjonalność i świadczenie usług w ograniczonym zakresie.

- **SipDTBR** – dystrybutor, odpowiedzialny za odpowiednie rozdzielanie danych. Po otrzymaniu danych od warstwy transportowej kieruje je do odpowiedniego podmodułu silnika SIP. W druga strone po otrzymaniu wiadomosci z silnika SIP, kieruje ja poprzez socket do odpowiedniego portu UDP.
- **SipTU** – jest to komponent sterująco-symulacyjny. Z jednej strony pełni rolę układu decyzyjnego, który podaje odpowiednie polecenia od użytkownika. W związku z tym, iż jest to symulacja. SipTU symuluje również zachowanie użytkownika. Należy tutaj nadmienić, iż nie chodzi tutaj o posługiwanie się sztuczną inteligencją lub różnego rodzaju algorytmami decyzyjnymi. SipTU jest raczej interpretatorem danych wpisanych przez prowadzącego badanie w pliku inicjalizacyjnym.

Dodatkowo zdefiniowano odpowiednie obiekty, które reprezentują sobą wiadomości SIP oraz jedną wiadomość sterującą silnikiem SIP przez SipTU. Te wiadomości to odpowiednio: **RequestSIP** (żądania SIP), **AnswerSIP** (odpowiedzi SIP), **ControlTU** (wiadomość sterująca). Logiczny schemat modułu przedstawiony jest na rysunku 6.



Rys. 6. Struktura modelu klienta SIP

2.2.2. Realizacja struktury klienta w NED.

W celu utworzenia szkieletu modelu opisanego w poprzednim rozdziale, utworzono w środowisku OMNeT++ projekt o nazwie „sipklientwat” a w nim dwa moduły złożone (*compound modules*) oraz cztery moduły proste (*simple modules*). Moduły proste i złożone są definiowane w plikach *.ned. Projekt posiada referencję do projektu „inet”. W OMNeT++ moduły złożone

stanowią jedynie szkielet, bez odpowiednich kodów opisujących ich działanie. Struktura modułów opisywana jest za pomocą języka Network Descryption (NED). OMNeT++ umożliwia także wizualne przedstawianie modułów. Przykładowy kod modułu SipEngine oraz Sip Uac przedstawione są poniżej:

SipEngine.ned:

```
package sipwatklient;
module SipEngine
{
    gates:
        input inUacTu;
        output outUacTu;
        input inUacDtbr;
        output outUacDtbr;
        input inUasTu;
        output outUasTu;
        input inUasDtbr;
        output outUasDtbr;
    submodules:
        sipUAC: SipUAC {
            @display("p=73,95");
        }
        sipUAS: SipUAS {
            @display("p=146,95");
        }
    connections:
        inUacDtbr  --> sipUAC.inDtbr;
        inUacTu  --> sipUAC.inTu;
        sipUAC.outDtbr  --> outUacDtbr;
        sipUAC.outTu  --> outUacTu;
        sipUAS.outDtbr  --> outUasDtbr;
        sipUAS.outTu  --> outUasTu;
        inUasDtbr --> sipUAS.inDtbr;
        inUasTu  --> sipUAS.inTu;
}
```

SipUac.ned:

```
package sipwatklient;
simple SipUAC
{
    parameters:
        double T1 @unit(s)=default(500ms);
        double T2 @unit(s)=default(4s);
        double T4 @unit(s)=default(5s);
        @signal[StateS](type=long);
        @statistic[StateT](source=StateS; record=vector);
    gates:
        input inTu;
        output outTu;
        input inDtbr;
        output outDtbr;

}
```

Moduły posiadają zdefiniowane parametry, bramy oraz w przypadku modułów złożonych (takim jest SipEngine) podmoduły oraz połączenia pomiędzy bramami. W celu połączenia z modułami zewnętrznymi podmoduły korzystają z mostkowania na bramie modułu złożonego.

Wiadomości zostały zdefiniowane w plikach .msg. Na ich podstawie tworzone są automatycznie kody klas wiadomości w języku C++.

ControlTU.msg:

```
cplusplus
{{
     #include "inet/networklayer/common/L3Address.h"
}}
class noncobject inet::L3Address;
namespace inet;
```

```
enum DestOfMsg //Where is message destinated
    {
        Tu=30;  //INFERIORED UAC/UAS->TU
        Uas=20; //SUPERIORED TU->UAS
        Uac=10; //SUPERIORED TU->UAC
    };
enum InfoCarried //What info message is carrying
    {
        CALL=1;
        TERMINATE=2;
        CANTALK=3;
        CANCELreq=3;
        OKansw=200;
        BUSYansw=486;
        RINGINGansw=180;
        NOT_ACCEPTABLE4answ=406;
        NOT_ACCEPTABLE6answ=606;
        NOT_FOUNDansw=404;
    };
message ControlTU {
    int infoCarried @enum(InfoCarried);
    int destOfMsg @enum(DestOfMsg);
    L3Address call2Address;     //where to call/from where the callee is calling
    string myVoiceCodec="";     //codec i want to negotiate/ codec which callee is proposing
    string call2User="";        //the name of user where i want to call/ which are calling
        string localUser="";            //the name of user whos is using the device
}
```

2.2.2. Realizacja funkcjonalności klienta w C++.

W środowisku symulacyjnym OMNeT++ każdy z modułów prostych posiada podstawowe metody jakimi są *initialize()*, *finish()* oraz *handleMessage()*. Odpowiadają one kolejno za utworzenie i inicjalizacje modułu podczas uruchamiania symulacji, zwieńczenie symulacji (zebranie statystyk, zwolnienie pamięci) i przetwarzanie zdarzeń, których odpowiednikiem są wiadomości otrzymywane przez moduły. Najczęściej metody są nadpisywane.

W związku z usytuowaniem klienta w najwyższej warstwie sieciowej, konieczna jest inicjalizacja jego modułów w momencie, gdy wszystkie niższe warstwy są aktywne, ponieważ świadczą one usługi na rzecz klienta. Wszystkie moduły klienta dziedziczą po *ApplicationBase* dlatego dla nich odpowiednikiem metody przetwarzania zdarzeń jest *handleMessageWhenUp(cMessage* msg)*, co powiązane jest również z powyższym. Główna struktura metody opiera się na warunkach zależnych od bramy na którą przyszła odpowiednia wiadomość. Przykładowa struktura dla SipTU:

```
        void SipTU::handleMessageWhenUp(cMessage *msg) {
                        if (msg->isSelfMessage()) {
                                …
                        } else {
                if (msg->arrivedOn("inUasSipEngine")) {
                                …
                } else if (msg->arrivedOn("inUacSipEngine")) {
                                …
                }
                }
        }
```

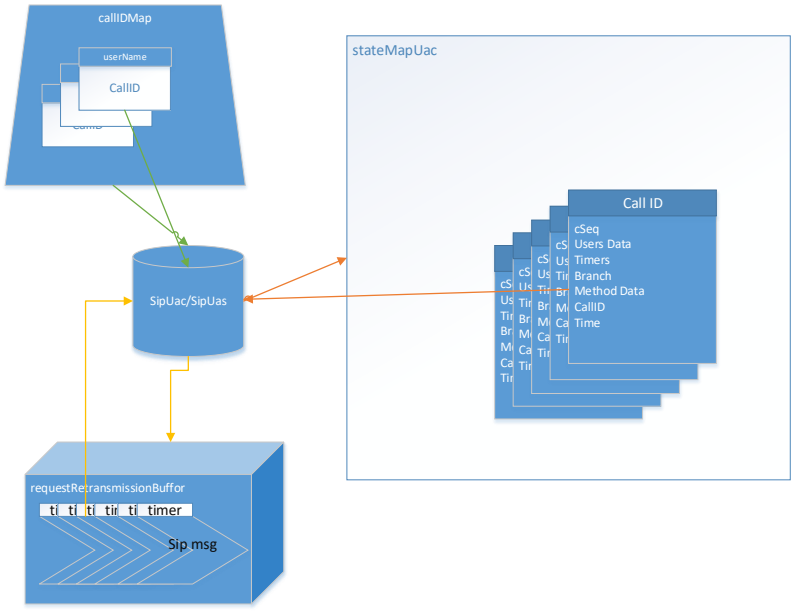
Większość modułów (wszystkie poza Dtbr) oparte są na maszynach stanów. W tym celu dla SipEngine zostały zadeklarowane i zdefiniowane odpowiednie obiekty, przechowywujące stan skorelowany z odpowiednią sesją oraz informacje o niej:

```
        enum StateOfSipEngine {
         TRYING = 11, PROCEEDING = 12, COMPLETE = 13, PROCESS = 14
```

```
};
class StatusOfUac {
…
}
class StatusOfUas {
…
}
```

Stany są przechowywane w mapach *std:map*. W celu powiązania nazwy użytkownika z sesją utworzone zostały również mapy łączące użytkowników z odpowiednim „Call-ID" sesji. Istnieje możliwość logowania stanów modułów silnika do pliku podobnie jak wiadomości SIP. Architektura modułów SipEngine przedstawiona jest na rysunku 7 przedstawione są również elementy zawarte w odpowiednich obiektach. W celu zapewnienia niezawodności SIP zapewnia retransmisje wiadomości. W związku z tym zostały utworzone odpowiednie bufory przetrzymujące wiadomości do retransmisji. Zasady retransmisji są zgodne z przyjętą maszyną stanów i normą [2].

SipTU jako moduł decyzyjny i symulujący zachowania użytkownika oparty jest na warunkowym przetwarzaniu przychodzących zdarzeń ale również w funkcji initialize zawarty został proces odczytywania parametrów podanych przez prowadzącego badanie. SipTU tworzy na podstawie podanych informacji o połączeniach planowanych strukturę *std:set* z obiektami *CallUser* reprezentującymi dane potrzebne do zestawienia połączenia z konkretnym klientem (nazwę użytkownika, adres IP, kodek wymagany do połączenia itp.). Warto podkreślić, że SipTU realizuje jawną translacje nazw hostów na adresy sieciowe z wykorzystaniem *addressResolver* będącego elementem biblioteki INET. Jest to konieczne, ponieważ brak serwerów SIP wymusza na użytkownikach deklarowanie konkretnych adresów sieciowych.



Rys. 7. Architektura modułów silnika SIP

2.3. Proces weryfikacji modelu klienta SIP.

Każda architektura oprogramowania, w ogóle każdy wynalazek jaki człowiek wymyśla i tworzy poddawany jest serii prób w celu weryfikacji poprawności działania. Często może się zdarzyć, że zamierzone cele znacząco odbiegają od tych osiąganych praktycznie. Proces weryfkacji podzielić można na trzy podprocesy: przyjęcie założeń scenariuszy, realizacje symulacji weryfikujących oraz analiza wyników i wnioskowanie.

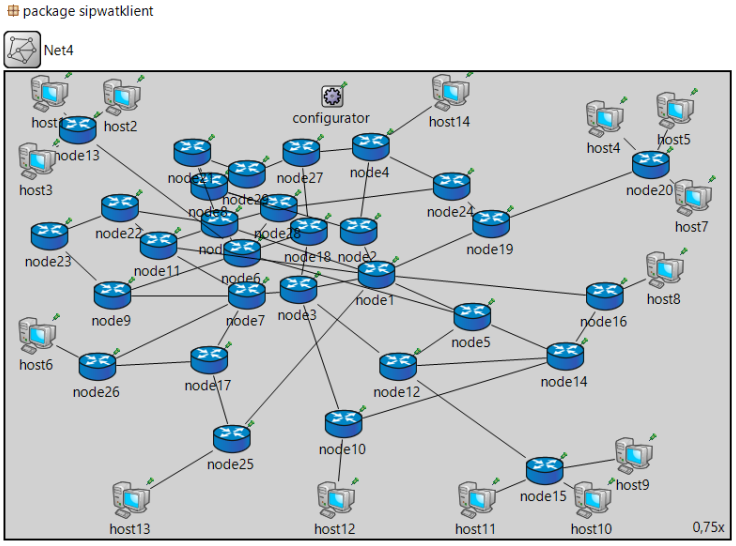
2.3.1. Przyjęcie scenariuszy:

Przyjęto 25 scenariuszy z czego kilka jest rozwiniętych do kilku wariantów co łącznie daje 35 scenariuszy weryfikujących. Przykładowe scenariusze:

1. *Scenario1:* Uruchomienie symulacji, sprawdzenie, czy w UDPApp znajduje się SipClient oraz jego struktura jest zgodna z oczekiwaną.
2. *Scenario7:* Sprawdzenie czy klient reaguje na błąd, gdy czasy zakończenia połączeń są mniejsze od czasów ich rozpoczęcia np. [*startCallingTimes>stopCallingTimes*].
3. *Scenario16:* Komunikacja pomiędzy dwoma hostami w trybie obydwa hosty inicjują do siebie połączenie w tym samym czasie (zgodność nazw użytkowników i kodeka).
4. *Scenario25:* Komunikacja pomiędzy dwoma hostami w trybie jeden inicjuje, drugi tylko odbiera. Realizacja jednego połączenia (zgodność nazw użytkowników i kodeka, włączenie logów).

2.3.2. Realizacja symulacji:

W celu realizacji symulacji weryfikujących stworzono inicjalizacyjny plik *omnetpp.ini* wraz z odpowiednimi scenariuszami badań oraz odpowiednie sieci. Przykładowa sieć przedstawiona jest na rysunku 8.



Rys. 2. Przykładowa sieć weryfikacyjna

Fragment pliku inicjalizacyjnego *omnetpp.ini:*

```
[Config Scenario23]
network=sipwatklient.Net4
**.host*.numUdpApps = 1
**.host*.udpApp[0].typename= "SipClient"
**.host1.udpApp[*].sipTU.localUser = "Job"
**.host2.udpApp[*].sipTU.localUser = "Bob"
…
**.host14.udpApp[*].sipTU.localUser = "CppFun"

**.host2.udpApp[*].sipTU.call2Users = "Mark@host3 Job@host1"
**.host2.udpApp[*].sipTU.startCallingTimes = "32 44"
**.host2.udpApp[*].sipTU.stopCallingTimes ="34 55"

…

**.host5.udpApp[*].sipTU.call2Users = "Dad2@host8 Job@host11 Manus@host11 Albert@host7"
**.host5.udpApp[*].sipTU.startCallingTimes = "12 26 43 83"
**.host5.udpApp[*].sipTU.stopCallingTimes ="24 41 61 112"
…
```

2.3.3. Analiza wyników i wnioskowanie:

Analizowano odpowiednie zachowania, ilość wiadomości wysłanych i odebranych oraz retransmitowanych, kolejność i rodzaj przyjmowanych stanów w różnorakich połączeniach. Dane porównywano z zamierzeniami i założeniami. Na podstawie zebranych danych w postaci tabel, wykresów oraz logów stanów i wiadomości stwierdzono, że klient przeszedł pozytywnie wszystkie scenariusze weryfikacyjne oraz spełnia wszystkie założenia.

3. Podsumowanie

Proces umożliwił stworzenie symulacyjnego modelu klienta SIP, który w pełni spełnia zamierzenia oraz jest zgodny z normą [2]. Co więcej na podstawie zabranych wyników wnioskować można, że istnieje możliwość rozwoju struktury modelu SIP w OMNeT++ oraz mechanizmy które wykorzystane są w kliencie potrafią wygenerować wiadomości, które z pewnością mogłyby być identyfikowane przez rzeczywiste urządzenia SIP. Opisany proces wydaje się być wzorcowym podczas tworzenia różnego typu oprogramowania.

Bibliografia

[1] K. Świdrak praca dyplomowa pod opieką dr. inż. J. Dołowskiego: „Opracowanie modelu klienta SIP dla środowiska OMNeT++”, Warszawa 2018, Wojskowa Akademia Techniczna

[2] J. Rosenberg, H. Schulzrinne, G. Cammarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler „SIP: Session Initiation Protocol”, IETF, RFC 3261, June 2002