Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Tímový projekt II

Dokumentácia k projektu

Tímový e-mail

tp.team8.22@gmail.com

Členovia tímu

Bc. Andrej Dubovský
xdubovsky@stuba.sk

Bc. Samuel Kačeriak
xkaceriak@stuba.sk

Bc. Ondrej Martinka
xmartinkao@stuba.sk

Bc. Vojtech Fudaly
xfudaly@stuba.sk

Bc. Richard Andrášik
xandrasik@stuba.sk

Bc. Roman Grom
xgromr@stuba.sk

VANETLAB – Dokumentácia

O projekte

Tento projekt bol robený pomocou development systému agile. V agile development systéme na rozdiel od waterfall si ustanovujeme dvojtýždňové "sprinty" počas ktorý pracujeme na úlohách vybraných na začiatku šprintu. Vyberanie úloh a synchronizáciu členov tímu riadi scrum master.

Úlohou scrum mastera je dohliadnuť aby boli úlohy správne pridelené a aby na konci sprintu všetky úlohy dokončené. Agile development systém nám pomohol pri každom stretnutí uviesť tým na rovnakú stránku a napredovať s projektom omnoho rýchlejšie. O našich sprintoch sa dá bližšie dočítať v desiatich jednotlivých reflexiách sprintov.

Reflexie tímu sa skladajú z piatich častí. Na začiatku je zhrnutie sprintu. Obsahuje tabuľku, ktorá v krátkosti opisuje našu prácu počas sprintu. Pod tabuľkou je krátke textové zhrnutie sprintu. Nasleduje paragraf o začiatku sprintu. Štandardne obsahuje ambície pre tento sprint, čo scrum master zmenil od nového sprintu a spomenuteľné informácie k začiatku sprintu. Potom "mid-sprint review", ktorý je písaný v strede sprintu, obsahuje informácie o našom progrese. Na konci je "end of sprint" review, v ktorom je celkové zhodnotenie ako sprint dopadol a čo je možné zmeniť pre ďalší sprint aby prebiehal lepšie. Reflexie sprintov boli písané počas tímových stretnutí.

Stretnutia tímu prebiehali každý týždeň. Na týchto stretnutiach sme diskutovali s vedúcim tímu o priebežnom stave projektu. Obsah týchto stretnutí je v bodoch zaznamenaný v zápisniciach stretnutí. Zápisnice boli tiež písané scrum masterom tímu.

NS3 a jeho limitácie

NS-3 (Network Simulator 3) je výpočtový simulátor, používaný na modelovanie a simuláciu sietí a komunikačných systémov. Je to voľne dostupný simulátor, vyvinutý pre výskumné a vývojové účely v oblasti počítačových sietí.

NS-3 má niekoľko významných vlastností a výhod:

- Flexibilita: NS-3 poskytuje vysokú úroveň flexibility pri vytváraní a konfigurácii sietí. Umožňuje detailné modelovanie rôznych typov sietí a komponentov.
- Realizmus: Simulácie v NS-3 sú založené na reálnych protokoloch a technológiách používaných v sietiach. To umožňuje presné modelovanie a analýzu výkonu sietí.
- Rozsiahlosť: NS-3 obsahuje rozsiahlu knižnicu modelov a protokolov, ktoré pokrývajú široké spektrum komunikačných technológii a sietových architektúr.
- Podpora výskumu: NS-3 je populárny medzi výskumníkmi a poskytuje možnosti experimentovania a vývoja nových protokolov a algoritmov.

Niektoré z limitácií NS-3 sú:

- Náročnosť na ovládanie: NS-3 vyžaduje určitú mieru technickej znalosti a skúseností na úspešné vytvorenie a spustenie simulácií. Jeho ovládanie môže byť náročné pre nových používateľov.
- Výpočtová náročnosť: Simulácie v NS-3 môžu byť náročné na výpočtové prostriedky a čas. Simulácie s veľkým počtom uzlov alebo rozsiahlymi sieťami môžu trvať dlhý čas na vykonanie.
- Obmedzený vizualizačný nástroj: NS-3 má obmedzenú podporu pre vizualizáciu a monitorovanie priebehu simulácie. To môže značne ovplyvniť schopnosť užívateľa sledovať a analyzovať výsledky simulácie.
- Chýbajúca kompatibilita s existujúcimi sieťovými systémami: NS-3 je samostatný simulátor a nie je vždy jednoduché prepojiť ho s existujúcimi sieťovými systémami alebo nástrojmi používanými v reálnych sieťach.

NS-3 spolu s potrebnými komponentami je dosť obtiažne nainštalovať správne, každá verzia ns-3 má nejaké muchy a každá inštalácia má náhodné chyby, ktoré je pre smrteľníkov nemožné debugovať.

Navyše, pre naše potreby potrebujeme NS-3 s podporou pythonu, lebo máme robiť modulárny scenár, ktorý sa mení podľa konfigurácie. Tá pochádza z celkom veľkého json-u, takže v C++ by toto bolo kódiť ako za trest. Bohužiaľ NS3 scenárov písaných v pythone je len zopár a celý doxygen je pre C++. Preto ponúkame top 5 super tipov ako písať NS3 v pythone:

1. Keďže python podpora je realizovaná cez cppy, volania z doxygenu sú viac-menej 1 ku 1.
2. Ak si neviem rady, viem si v našom python zdrojáku definovať funkciu napísanú v cpp:
   ns.cppyy.cppdef("""
   using namespace ns3;
   Ptr<Ipv4> getNodeIpv4(Ptr<Node> node) {
     return node->GetObject<Ipv4>();
   };""")
   A následne ju volať cez `ns.cppyy.gbl.getNodeIpv4(node)`. Takýmto spôsobom viem doimplementovať do našeho zdroja aj module nepodporujúce cppy preklad, ak by sa niekto takýmto smerov vydal…
3. V príkladoch v stiahnutom ns3 priečinku nie sú všetky python scenáre. Keď hľadáme `.py` súbory v doxygene, tak toho nájdeme viac, aj keď nie vždy scenárov, dajú sa odtiaľ naučiť taktiky.
4. Keď mi padá scenár tak je možné že sa premenná neinicializuje rovnako ako v Cpp. Niekedy treba `CreateObject`, niekedy stačí `Object()`, niekedy treba `Object.Default()`
5. Obyčajné `print(val)` a prípadne `print(dir(val))` vie dosť pomôcť keď neviem s čím to vôbec pracujem.

Zistené chyby v jednotlivých verziách NS3:

- < NS3.36 - python podpora je realizovaná cez `pybindgen`, v ktorom sme nenašli bindingy na openflow switche, ktoré boli v požiadavkách
- < NS3.37 - openflow modul nemá pyhton bindingy, takže sa dal scenár so SDNkami kódiť len v C++.

- > NS3.37 - python podpora cez cppy, dá sa priamo z pythonu volať a definovať CPP kód, zmena syntaxu - scenáre napísané v nižších verziách treba prepísať.
- NS3.37 - myslím že nefungoval python port pre netanim

Z týchto dôvodov sme pre potreby projektu vybrali aktuálne najnovšiu verziu - 3.38.

Manuálna inštalácia NS3 - v3.38

1. ideme na `nsnam.org`, tapneme na download, v lavom draweri vyberieme `3.38` a stiahneme ten krásny kus softvéru.
   `wget https://www.nsnam.org/releases/ns-allinone-3.38.tar.bz2`
2. ďalej to extrahujeme cez `tar -xf ns-allinone-3.38.tar.bz2`
3. nainštalujeme dependencies
   `apt install python3 python3-pip python3-dev`
   `apt install libboost-dev`
   `pip install cppyy`
   prípadne aj ďalšie, ak bude čosi kričať
4. stiahneme openflow, **nie** do ns3 folderu, aj keď v tutoriáloch to chcú presne tam. Klamú. nefunguje to. `cd ~ && hg clone http://code.nsnam.org/openflow`
5. kompilujeme openflow:
   a. `cd openflow`
   b. `./waf configure`
   c. `./waf build`
6. opravíme ns3 `CMakeLists.txt` súbor tak aby sa nakonfiguroval s python podporou
   a. na riadok ~17 (kludne aj iný ale na tomto to funguje a s ns3 na to treba ísť opatrne s rukavičkami) pridáme riadok:
   b. `set(NS3_BINDINGS_INSTALL_DIR "*/home/$USER/.local/lib/python3.X/site-packages*")` - **path zmeniť podľa vlastnej potreby!**
   c. ak neviem čo tam presne dať, lebo som napríklad na windowsoch alebo tak, tak to najprv ns3 nakonfigurujem bez toho a hľadám žltý riadok, kde ns3 hovorí, že nemá kam dať bindingy a navrhne nejaký path. ten tam dám.
7. nakonfigurujeme `ns3`
   a. `cd ns-allinone-3.38/`
   b. `./ns3 configure –enable-python-bindings –with-openflow=/home/${USER}/openflow`
8. zkompilujeme ns3
   a. `./ns3 build`
9. prekopírujeme openflow knižnice do ns3
   a. `cp -r ~/openflow/include/openflow ~/ns3-allinone-3.38/ns-3.38/build/include`
10. všetko by malo fičať. hurá.

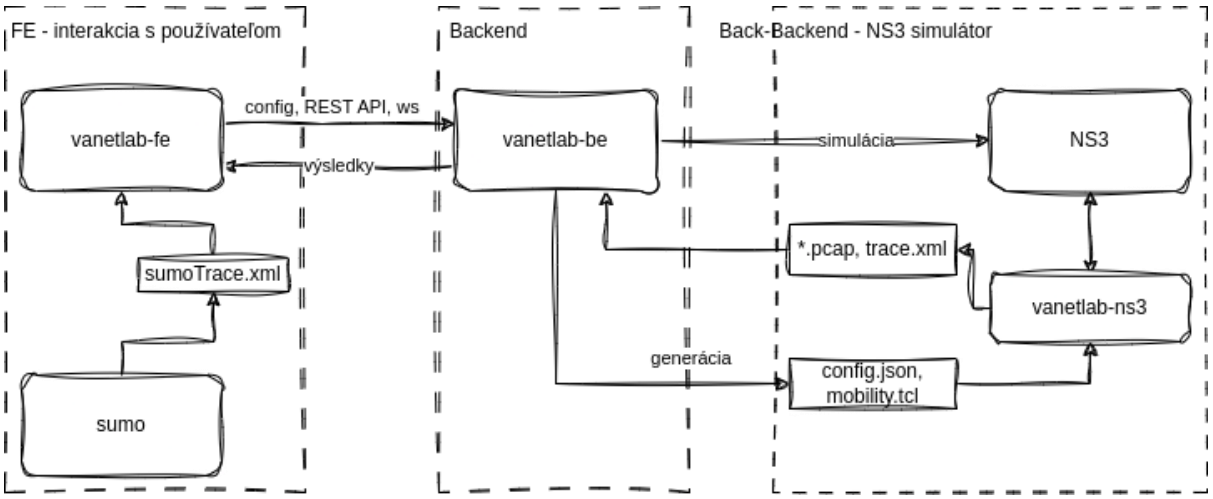
Celý tento proces sme zdokumentovali a vytvorili Dockerfile, takže to je preferovaný spôsob spúšťania. Venujeme sa mu neskôr, v sekcii o spúštaní development prostredia.

Architektúra systému a technická dokumentácia

Aplikácia je zložená z troch hlavných častí

- Frontend (https://github.com/vanetnuggets/vanetlab-fe)
- Backend (https://github.com/vanetnuggets/vanetlab-be)
- NS3 scenár (https://github.com/vanetnuggets/vanetlab-ns3)

High-level koncept fungovania softvéru a spolupráca komponentov je znázornená na nasledujúcom obrázku.



Celkom chaos. Vysvetlenie:

1. Používateľ si cez nástroj SUMO nechá vygenerovať `sumoTrace.xml` súbor, v ktorom je scenár vehikulárnej premávky vrátane mobility vozidiel.
2. Tá sa importuje do front-endu, pomocou API callu na BE sa pretransformuje na náš vanetlab json konfiguračný súbor.
3. Ten sa cez reaktívne webové UI vykreslí a upravuje. Každá časť konfigurácie je zaznamenaná v konfiguračnom súbore.
4. Pre simuláciu sa pošle POST požiadavka na BE. V tele požiadavky je celá konfigurácia.
5. Z konfigurácie sa na BE spraví ns2 mobility TCL (ako sa pohybujú jednotlivé uzly) a spolu s konfiguračným súborom sa uložia do súborového systému na mieste, ktoré je čitateľné z NS3 simulátora.
6. Backend zavolá NS3, ktoré odsimuluje náš scenár (vanetlab-ns3), v ktorom sa cez argumenty pošle cesta na konfiguračný súbor a mobility TCL.
7. Náš NS3 scenár si vyčíta konfiguráciu zo súborov a odsimuluje sa v NS3 simulátore.
8. Výstupy sú potom prenesené do priečinku scenáru (ten, kam sa uložil json config a mobility tcl). Odtiaľ sa dajú čítať Back-end-om.
9. Cez `GET /api/name/summary` vieme získať výsledky simulácie.

Modulárne scenáre - VanetLab JSON

Každý jeden konfigurovateľný scenár vytvorený vo vanetlabe je unikátny a teda ns3 scenár, ktorý ho simuluje, musí byť natoľko modulárny, aby ho vedel odsimulovať. Funguje tak, že náš scenár má definované triedy a metódy, pomocou ktorých vie odsimulovať každú z podporovaných technológií. To, či bude v scenári zahrnutá, sa zistí cez náš vanetlab json. Má takúto štruktúru:

1. Networks - zoznam konfigurácie sietí. Každá sieť má ako kľúč ID siete a ako obsah má takéto atribúty
    a. id - rovnaké ako klúč
    b. color - farba, ktorou je na front-ende vykresľovaná
    c. ssid - meno siete - pre vykreslenie na front-ende, nie len pri wifi, ssid sa to volá z historických dôvodov
    d. addr - IP adresa siete vo CIDR notácii.
    e. type - typ siete - LTE, WAVE, WIFI alebo ETH (TODO SDN?)
2. Nodes - Zoznam konfigurácii jednotlivých uzlov. Najväčší objekt v konfigurácii. Ako kľúč má podobne ako siete svoje ID. Každý objekt uzlu má nasledovné parametre:
    a. id - ID uzla
    b. mobility - Vnorený slovník, kde ako kľúč je sekunda a ako obsah objektu je x, y súradnice a rýchlosť uzla v danú sekundu. Slúži na vygenerovanie ns2mobility TCL súboru, ktorý sa aplikuje na simuláciu.
    c. l2 - aká technológia sa používa v uzle - LTE, WIFI, WAVE alebo ETH
    d. l2id - ID siete, do ktorej patrí.
    e. l2conf - Konfiguriácia na druhej TCP/IP vrstve. Atribúty konfigurácie:
    v závislosti od typu siete sú odlišné l2 configy:
    WIFI
        i. standard - wifi standard(napríklad 802.11a)
        ii. type - či sa jedná o ap alebo stanice
        iii. txgain - optional, zmena vysielacej sily
        iv. rxgain - optional, zmena prijímajúcej sily
    LTE
        v. type - ue, en , pgw
    ETH
        vi. type - definovaný iba jeden typ, generic node
    WAVE
        vii. type - definovaný iba jeden typ, generic node
        viii. txgain - optional, zmena vysielacej sily
        ix. rxgain - optional, zmena prijímajúcej sily
    f. l3 - Typ aplikácie, ktorá beží na uzly. (udpclient, udpserver, tcpclient, tcpserver)
    g. l3conf - Konfigurácia na štvrtejTCP/IP vrstve. Atribúty konfigurácie
        i. udpclient
            1. port - číslo portu, na ktorom bude prebiehať komunikácia
            2. start - sekunda v ktorej sa začne komunikácia
            3. stop - sekunda v ktorej sa skončí komunikácia
            4. communication - číslo uzlu, s ktorým bude komunikácia prebiehať
            5. interval - optional, interval odosielania
            6. packet_size - optional
            7. max_packets - optional
        ii. udpserver
            1. port - číslo portu, na ktorom bude prebiehať komunikácia
            2. start - sekunda v ktorej sa začne komunikácia
            3. stop - sekunda v ktorej sa skončí komunikácia
        iii. tcpclient

1. port - číslo portu, na ktorom bude prebiehať komunikácia
2. start - sekunda v ktorej sa začne komunikácia
3. stop - sekunda v ktorej sa skončí komunikácia
4. communication - číslo nodu, s ktorým bude komunikácia prebiehať
5. max_bytes - optional, maximálny počet bytov

iv. tcpserver

1. port - číslo portu, na ktorom bude prebiehať komunikácia
2. start - sekunda v ktorej sa začne komunikácia
3. stop - sekunda v ktorej sa skončí komunikácia

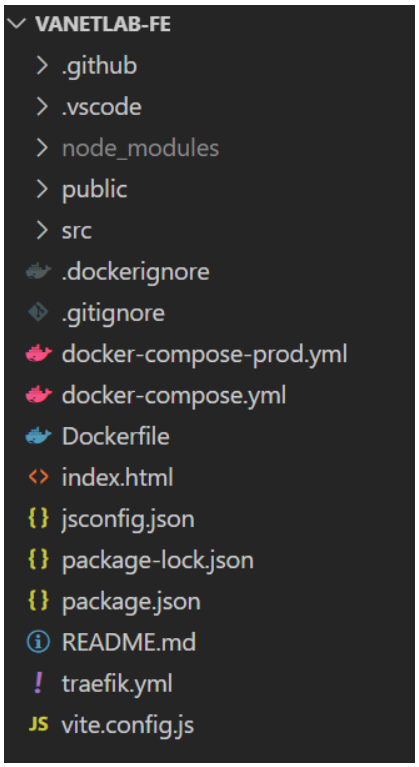
3. Connections - Zoznam Point-to-point spojení medzi 2mi uzlami. Realizuje sa cez neho prepojenie sietí medzi sebou.
4. Max_at - číslo v sekundách, kedy sa simulácia skončí. Vo front-ende aplikácie sa automaticky sa vypočíta posledným záznamom o pohybe uzla.
5. Routing - Myšlienka bola, že podľa tohto parametra sa bude v simulácii realizovať smerovanie medzi bezdrôtovými uzlami. V simulácii ale nefungoval, tak sa v tejto verzii aplikácie nepoužíva.
6. Labels - ide o text, ktorý sa vypisuje na našom frontende pre popis scenáru, priamo v scenári sa nepoužíva. Jeho atribúty sú `id`, `x`, `y` a `text`.

Príkladové scenáre prikladáme v zdrojových kódoch inžinierskeho diela. Priložené scenáre reprezentujú nasledujúce preddefinované scenáre:

- vienna-ethernet - scenár znázorňuje funkcionalitu eth a sdn
- new-york-wifi - scenár znázorňuje funkcionalitu wifi
- texas-wave - scenár znázorňuje funkcionalitu wave
- lte-example - scenár znázorňuje funkcionalitu lte
- zanzibar-mix - scenár znázorňuje funkcionalitu všetkých použitých technológií

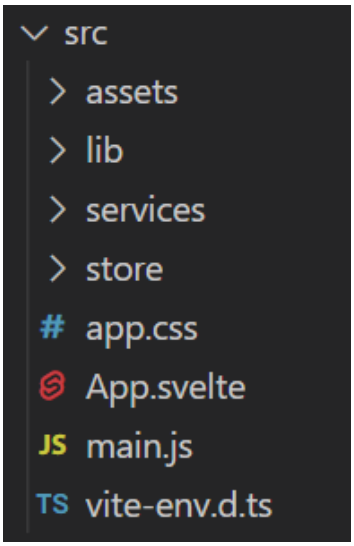
Frontend

Frontend bol naprogramovaní vo frameworku Svelte. Svelte je bezplatný a open-source front-end komponentný rámec, ktorý využíva jazyky JS, HTML a CSS.



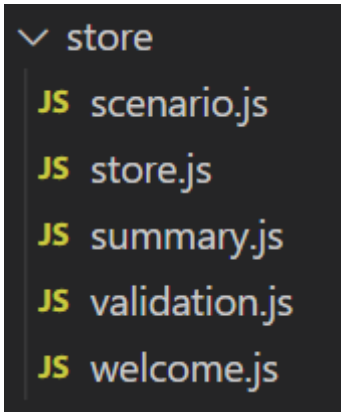
Základná kompozícia kódu

Po otvorení súboru VANETLAB-FE, ktorý obsahuje zdrojový kód nášho frontendu, sa nám ukážu súbory, ktoré vidíme na obrázku vľavo. Väčšina súborov sú originál pridružené od svelte už pri prvotnej inštalácií svelte modulu. Sú tu však dôležité súbory ako Dockerfile, docker-compose-prod.yml a docker-compose.yml. Tieto súbory slúžia na nastavenie konfigurácie Docker filu. Ďalej môžeme vidieť aj .gitignore subor, v ktorom máme zapísané všetky nepotrebné veci, ako node_modules a ďalšie podobné priečinky a súbory, ktoré nemajú opodstatnenie nahrávať ich na github. Ďalej si postupne rozoberieme zvyšné priečinky, ktoré môžeme vidieť na obrázku. Priečinok node_modules si môžete predstaviť ako vyrovnávaciu pamäť pre externé moduly, na ktorých závisí náš projekt. Keď ich nainštalujeme cez *npm install*, stiahnu sa z webu a kopírujú sa do priečinka *node_modules* a *Svelte* je naučený, aby ich tam hľadal, keď ich importujeme (bez špecifickej cesty).
Priečinok *.github* slúži na definovanie pull request templatu, a taktiež je tu nastavené automatické nasadenie zmien na produkčné prostredie po mergnutí zmien do hlavnej vetvy, ktorou je *main.* Ďalej sa budeme venovať už iba priečinku *src*, nakoľko sa v tomto priečinku nachádza celá naprogramovaná logika programu.



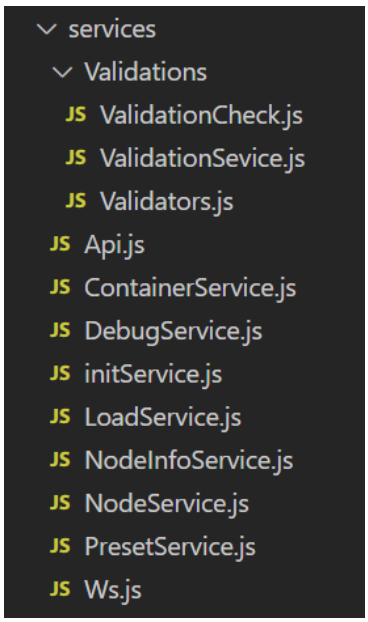
Priečinok ***src***

V priečinku *src* sa nachádzajú 4 priečinky. Priečinok *assets* obsahuje všetky obrázky a ikony potrebné pri vizuálnom zobrazovaní našej stránky. Taktiež sa tu nachádza aj súbor *default.css*, ktorý obsahuje všetky potrebné globálne css štýly. Zvyšné priečinky nachádzajúce sa v *src* si prejdeme postupne neskôr. Súbor *App.svelte* slúži na nastavenie smerovania url cesty. Sú tu nastavené niektoré cesty, ktoré slúžia na prepínanie obrazoviek a motívov.



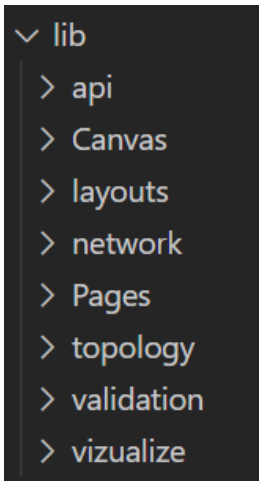
Priečinok ***src/store***

Priečinok *src/store* obsahuje 5 súborov js, ktoré predstavujú story pre rôzne použitia. V súbore *welcome.js* si ukladáme zoznam všetkých scenárov, ktoré má používateľ otvoriť s remote úložiska. Súbor *validation.js* si ukladá jednotlivé validácie pre rôzne vstupy používateľa. *summary.js* slúži ako úložisko na uchovanie výsledkov zo simulácie. Taktiež sa tu ukladajú aj informácie o aktuálnom priebehu procesu simulácie. *store.js* predstavuje úložisko, v ktorom sú uložené všetky potrebné informácie pre beh programu. Nachádzajú sa tu všetky veci, ktoré nepatrili do ostatných storov. *scenario.js* predstavuje základný Json, ktorý sa posiela na backend. To znamená, že tento store predstavuje základnú konfiguráciu celého programu a ukladajú sa tu informácie o scenárie, na ktorom sa aktualne pracuje.



Priečinok ***src/services***

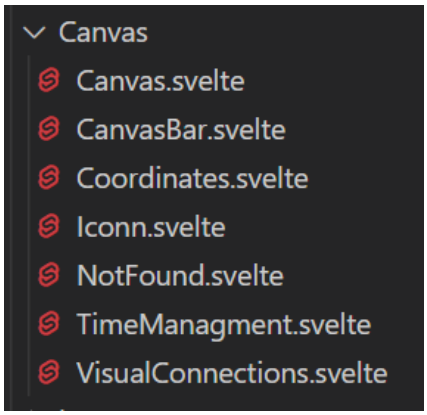
V tomto priečinku sa nachádza iba jeden ďalší priečinok *Validation*, ktorý ponúka services na skontrolovanie celeho základného JSON-u, či je v poriadku, vytváranie služby na validovanie a samotné preddefinované validátori. Ďalej sú tu ostatné servisy, ktoré slúžia na bežný beh programu.*ContainerService.js* bližšie popisuje, aký tvar JSON-u budú mať *networks. initService.js* a *LoadService.js* slúžia na načítanie dát z uvodnej obrazovky do nášho scenario JSON-u. Servisy s nodami slúžia na získavanie aktuálneho nastavenie určenej nody.



Priečinok ***src/lib***

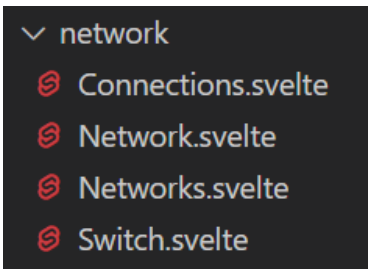
Tento priečinok obsahuje ďalšie adresáre, ktoré zabezpečujú rôzne funkcionality nášho systému. *api* slúži na api volania do storu, ktorý nám spätne vracia hodnoty. *Canvas* popisuje správanie sa plátna, na ktorom sa vizuálne zobrazujú nody. *layouts* popisuje rozloženie jednotlivých komponentov na obrazovke. *network* popisuje lavy drawer, to znamená že bližšie určuje správanie sa konfigurácie sietí. *network* zasa naopak špecifikuje správanie sa pravého drawera. To znamená samotnú konfiguráciu jednotlivých nodov. *validation* obsahuje jednotlivé komponenty, ako napríklad input alebo select, ktoré priamo

obsahujú validátory ich vstupov. A *vizualize* predstavuje správanie sa počas simulácie a vykresľovanie vysledkov.



Priečinok ***src/lib/Canvas***

Súbor *Canvas.svelte* popisuje samotné správanie sa plátna, na ktorom je vizuálne predstavený scenár. Popisuje sa tu pohyb nodov po osi, vytvára sa samotná os a pridáva sa možnosť zoomovania. *CanvasBar.svelte* zasa pridáva na plátno zopár klikacích možností, ktoré je možné vykonať na plátne. Ako napríklad mazanie, vytváranie peer to peer prepojení a podobne. *Coordinates.svelte* zabezpečuje prepočet koordinátov, aby sme pri zoomovaní vedeli reálnu pozíciu nody. *TimeManagment.svelte* pridáva na plátno možnosť prepínať aktuálny čas a meniť aktuálny maximálny čas. Taktiež zabezpečuje, aby sa nody pohybovali po plátne podľa času. *VisualConnections.svelte* vytvára vizuálne čiary, ktoré reprezentujú prepojenie medzi dvoma nodami.

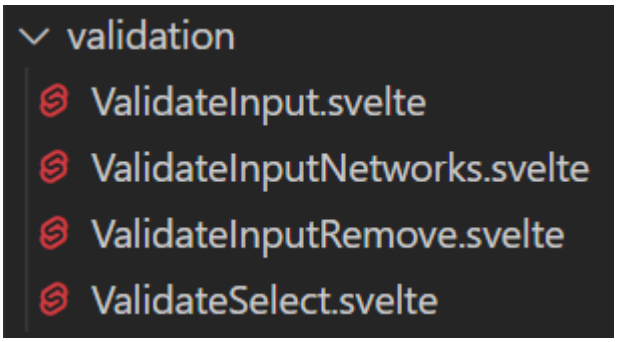


Priečinok ***src/lib/network***

Connections.svelte vizuálne reprezentuje peer to peer prepojenie ako zoznam usporiadaných dvojíc, medzi ktorými sa nachádza toto prepojenie. *Network.svelte* popisuje samotnú sieť, jej vlastnosti. *Networks.svelte* vytvára list netwokov. *Switch.svelte* je komponent, ktorý určuje typ siete (LTE, WIFI …)

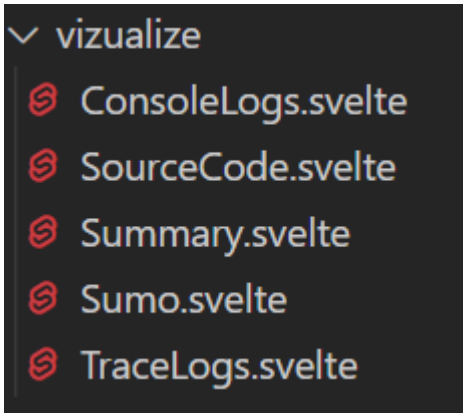
Priečinok ***src/lib/topology***

L2.svelte predstavuje konfiguráciu nody na druhej vrstve, to predstavuje výber siete a samotný typ nody, špecifický pre zadaný typ siete. *L3.svelte* reprezentuje nastavenie nody na tretej sieťovej vrstve, čiže pridáva rozdelenie na udp a tcp komunikáciu. Pre každý jeden typ z týchto komunikácií existujú povinné atribúty, ktoré je užívateľ povinný zadať. *L2attributes.svelte* a *L3attributes.svelte* pridávajú voliteľné atribúty pre jednotlivé vrstvy, ktoré používateľ nie je povinný zadať. *Mobility.svelte* pridáva možnosť užívateľovi manuálne vytvárať key framy, ktoré sú potom aj vizuálne zobrazené v tabuľke pre každú nodu zvlášť. *NodeInfo.svelte* popisuje základné informácie o node, ako napríklad pozícia nody v priestore alebo id. *SdnController.svelte* a *SdnNeigbors.svelte* popisujú správanie sa špecialneho typu nody, čo je sdn. Taktiež popisujú vzťahy, ktoré táto noda má.



Priečinok ***src/lib/validation***

Každý z pridružených súborov predstavuje samotný komponent, či už input alebo select. Každý komponent obsahuje svoje vlastné validácie a taktiež inputy sú rozdelené podľa formátu, aký tam užívateľ zadáva. Tieto komponenty riešia, aby používateľa upozornili a nepustili simulovať scenár, ak by napríklad v input políčke, ktorý očakáva IP adresu bol nájdený iný typ údaju.

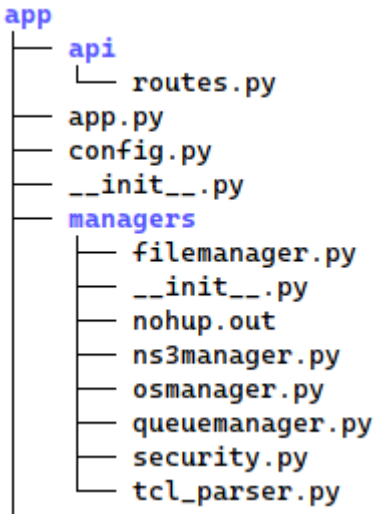


Priečinok ***src/lib/vizualize***

V tomto priečinku sa nachádzajú komponenty pre “vizualizáciu” výstupu scenáru. Všetko robí súbor “Summary.svelte”, ktorý len číta hodnoty zo store-u a pri úspešnej simulácii zobrazí tlačidlá pre stiahnutie výsledkov. Zvyšok file-ov sú pozozstatky z predošlích iterácii, kedy sme zisťovali limitácie NS3. Ale v backlogu sme mali tasky na kódenie, nie refaktorovanie a mazanie, takže máme tu zopár súborov na ktoré padá prach a čakajú len na ich neodvrátiteľnú záhubu.

Backend

Naprogramovaný v pythone s knižnicou flask. Je veľmi drobný a jednoduchý, nakoľko ide len o rozhranie, ktoré nám umožní ovládať ns3 simulátor cez webové API. Má takúto súborovú štruktúru:



Routes.py

V tomto súbore sú definované všetky API cally. Máme 2 typy

- REST api cally, ktoré majú prefix `/api`
- Websocket spojenie s prefixom `/ws`

Majú takúto funkcionalitu

- `ws: /status/<name>` - vráti status simulácie scenáru <name>. Či sa práve simuluje, koľkáty je v poradovníku na simuláciu.
- `get: /key/check` - vráti true ak je správne nastavený api kľúč
- `get: /isalive` - jednoduchá kontrola či api beží
- `post /from-sumo` v tele požiadavky cez multipart form so súborom s názvom `file` berie súbor, očakáva formát `sumoTrace.xml`. Spraví z neho základný vanetlab json konfiguráciu a vráti status.
- `get: get/<name>/<file>` vráti nám súbor `<file>` zo scenáru `<name>`. `<name>` je ID scenára a `file` môže byť
    - `output` - konzolový výstup ns3 simulácie
    - `mobility` - ns3 tcl súbor o mobilite zlov
    - `trace` - trace.xml súbor pre analýzu v Netanime
    - `config` - vanetlab konfigurácia
    - `pcap` - zozipovaná sieťová premávka v scenári
    - `ascii`- zozipované ascii logy pre analýzu v ďalších nástrojoch
- `post: simulate/<name>` pridá scenár s ID <name> do poradovníka pre simuláciu. Scenár berie z tela požiadavky v json formáte, očakáva vanetlab json.
- `get: /summary/<name>` vráti zhrnutie o simulácii s ID == <name>. Ide o názvy vygenerovaných súborov spolu s ich veľkosťami.
- `get: /list` vráti zoznam existujúcich scenárov uložených na stroji
- `get: /scenario/<name>` vráti scenár uložený na stroji
- `post: /scenario/<name>` uloží scenár na stroj, konfigurácia je v tele požiadavky
- `delete: /scenario/<name>` vymaže scenár zo stroju, pokiaľ nie je zaznačený ako príkladový
- `get: /exists/<name>` otestuje, či scenár existuje alebo nie.

Všetky API cally vracajú dáta vo formáte
```
{
 “error”: True/False,
 “message”: “error message” # len ak je error = true
 “data”: {} # dáta, ktoré vracia volanie
}
```

Okrem api volaní máme na backende zopár “manažérov”. Ide o pomocné triedy, ktoré implementujú nejakú špecifickú funkcionalitu. Cieľ bol, aby bol v API cestách poriadok, ale nepodarilo sa a ajtak tam je všade bordel.

FileManager

V tomto súbore sa rieši práca so súborovým systémom. NS3 je zvláštny nástroj, a pólku výstupu vie ukladať na ľubovoľnú cestu a druhú polku len do priečinka s ns3 binárkov. Trieda v tomto súbore implementuje funkcie na prenášanie týchto výplodov ns3 simulátora. Taktiež je použitý na zipovanie sieťovej premávky vygenerovanej ns3 simuátorom v ascii aj pcap formáte.

Ns3Manager

Táto trieda je len prietokový prehrievač medzi NS3 simulátorom, prípadne inými criptami ako sumo traceExporter a API endpointami. Má v sebe implementované funkcie ako `from_sumo()` a `simulate()`, ktoré cez `subprocess` modul pythonu spúštajú systémový príkaz a následne spracujú jeho výstup.

OSManager

Istý členovia tímu to mali rozbehnuté na windowse, kde sú opačné lomítka. Takže tu máme krásnu funkciu `l()`, ktorá ich vymení. Deň potom sme radšej spravili dockerfile, ale toto je pamiatka a pripomienka, že windows nie je dobrý operačný systém.

QueueManager

NS3 vie simulovať len 1 simuláciu naraz, lebo výstupy ukladá do priečinku s ns3 binárkou. To je problém, lebo ak zavolám simulate api call viac krát, tak sa to tam pomieša. Preto máme naimplementovanú queue, kam sa vkladajú simulácie a postupne sa vykonávajú. Mali by sme tam niekam pridať mutexy a určite tam je niekde race condition, ale pravdepodobnosť že sa to stane je zanedbatlná.

Security

Je tu definícia dekorátorov pre validovanie polí a pre autorizované volania. Autorizované sú tie, ktoré zapisujú na disk a všeobecne využívajú systémové prostriedky. Bez nich by hocikto mohol popridávať 3458435P23 simulácii a CPUčko by šlo na 100 percent až do spálenia. Preto definujeme API kľúč, ktorý sa vo front-ende pridá do lokálneho úložiska a posiela sa pri každej požiadavke. Niečo na štýl Bearer Authorization.

TclParser

Súbor, v ktorom je implementovaný parser, ktorý z ns2 mobility TCL-ka spraví jednoduchý, prázdny scenár, ktorý vie vyrkreslovať náš front-end a ďaľej ho upravovať. Vlastne len cez zopár šikovných regexov prečíta a zaznamená počet uzlov a ich pozíciu v danej sekunde.

Premenné prostredia

Aby bol back-end spustiteľný, potrebuje nastaviť nasledovné premenné prostredia:

- `VANETLAB_API_KEY`= Api kľúč, podľa ktorého budú povolené autorizované volania. Týmto sa nastavuje, a klienti ho musia použiť ak ich chcú použiť (m0skFfbELmUsq6g0YmOAU8kXjnFq7vOJ)
- `FLASK_APP` názov súboru, ktorý sa spustí pri príkaze `flask run` (main.py)
- `NS3_SCENARIO_PATH` cesta k `main.py` súbore repozitáru `vanetlab-ns3`, ktorý obsahuje ns3 náš scenár (/vlns3/main.py)
- `NS3_WAF_PATH` cesta k ns3 binárke (bez mena binárky). (/usr/ns-allinone-3.38/ns-3.38/)
- `NS3_PATH` rovnaké ako WAF_PATH, v novších verziach ns3 je už ns3 a nie waf. (/usr/ns-allinone-3.38/ns-3.38/)

- `SUMO_TRACE_EXPORTER` cesta k python scriptu (vrátane mena súboru), ktorý vie z `sumoTrace.xml` súboru vygenerovať ns2 mobility súbor. (/usr/share/sumo/tools/traceExporter.py)

NS3 scenár

Ako vyplíva z názvu, ide o ns3 scenár. Keďže ho potrebujeme modulárny, aby sa v ňom dali spúšťať unikátne scenáre, je rozdelený do viacerých častí. V prípade budúceho rozšírenia aplikácie o nové technológie, toto bude komponent, v ktorom bude najviac práce. Ak teda v tíme budú weboví developeri. Ak takí v tíme nebudú, tak najviac práce bude v boji s javascriptom. Ale o tom po tom. Alebo predtým. Má takúto štruktúru:

```
vanetlab-ns3/
├── application
│   ├── __init__.py
│   ├── tcp.py
│   └── udp.py
├── app_util.py
├── attribute_manager.py
├── config2.py
├── config.py
├── context.py
├── dubak_translator.py
├── examples
│   ├── lte-eth-wifi-routing.py
│   └── sdn.py
├── ip_util.py
├── log_helper.py
├── main.py
├── network
│   ├── eth.py
│   ├── __init__.py
│   ├── lte.py
│   ├── wave.py
│   └── wifi.py
├── ns2mobility.tcl
├── ns2_node_util.py
├── phy_util.py
├── sdn_manager.py
├── tracehelper.py
└── util.py
```

Scenár je realizovaný do modulov. Každý z modulov si prejde cez konfiguračný súbor (vanetlab json), zistí, či v ňom je ḱonfigurácia, ktorá sa ho týka a ak hej, tak ho pridá do simulácie. Keď nie, tak sa logicky nepridá.

Main

Ako vstup do simulácie sa cez argumenty programu prenesú parametre

- config - cesta do súboru, v ktorom je vanetlab config

- tracepath - cesta, do ktorej sa ukladá výstup simulácie, ak sa to dá špecifikovať (pcap-y sa nedajú a ukladajú sa do ns3 priečinku, ale ASCII logy sa ukladajú tam kam poviem)
- mobility - cesta k súboru, kde je ns2mobility TCL
- validate - boolean, či sa scenár má spustiť, alebo len validovať. Pri validovaní sa scenár ”spustí”, ale na 0 sekúnd, čiže sa overí, či sa simulácia vie správne zostaviť.

Najprv sa z vanetlab json konfiguračného súboru vyčíta počty uzlov. Na základe toho sa vygeneruje globálny NodeContainer objekt, v ktorom sa inicializujú prázdne uzly. Aplikuje sa na ne mobiltia z mobility TCL súboru zadaného cez parametre a následne sa zavolajú všetky moduly popísané nižšie, čím sa na uzly v konrajneri aplikujú ďaľšie konfigurácie.Na koniec sa simulácia spustí.

Apputil a priečinok application

V priečinku application sú v každom súbore zvlášť moduly pre konfiguráciu aplikácie. Aktuálne máme 2 - TCP a UDP. Každý z nich má implementované 2 funkcie

- add server - pridá na uzol server a správne ho nakonfiguruje z konfigurácie z vanetlab jsonu.
- add client - pridá na uzol klient a správne ho nakonfiguruje z konfigurácie z vanetlab jsonu.

Trieda AppUtil v súbore AppUtil inicializuje triedy zo súborov v priečinku `application` a týmto agreguje inštaláciu všetkých aplikácii na vyžších OSI vrstvách do volania jednej metódy z jednej triedy, ktorá je volaná v maine.

PhyUtil.py a priečinok network

PhyUtil je trieda, ktorá má na startosti inštalovanie sietí do scenáru. Volá sa výhradne iba z `main.py` súboru. V ňom sa volá metóda `install`, ktorá ako parameter dostane globálny zoznam uzlov `context.nodes`. Do nich nainštaluje siete. Siete sú definované v priečinku `network`.

Každý typ sietí, ktorý podporujeme je definovaný jedným súborom a jednou triedou v priečinku `networks`. Aktuálne to sú `eth`, `lte`, `wifi` a `wave`. Inštancie týchto tried sú inicializované v `phyutil` triede, Každý z týchto tried implementuje práve jednu metódu `install`, ktorá sa volá v ňom volá.

Každý z týchto sieťových tried má takéto atribúty

- nodemap - zoznam uzlov z konfiguračného súboru
- netmap - zoznam sietí z konfiguračného súboru
- xyz_nodes - uzly danej siete v `NodeContainer` triede. Nevytvárajú sa nové uzly, ale pridávajú sa referencie z globálnych uzlov `context.nodes` do kontajneru pre danú sieť
- xyz_devs - uzly, alebo teda sieťové rozhrania pre danú sieť v danom uzle, vyrobené ns3 metódou Helper.Install(xyz_nodes)
- xyz_helper - ns3 helper trieda pre inštaláciu siete
- zvyšok podľa potreby

Príklad pre ETH siete:

1. Inicializácia
    a. naplní sa `nodemap` uzlami, ktoré patria práve sietiam typu ETH. Keďže jeden typ môže byť vo viacerých, nezávyslých sieťach, `nodemap` je dvojitý dict. K uzlu sa dostaneme takto: nodemap[id siete][id uzla]
    b. naplní sa `netmap` konfiguráciou sietí
    c. nič sa zatiaľ nedeje s ns3
2. Inštalácia sietí
    a. Preiteruje sa cez všetky siete v `netmap`
    b. Ak ide o sieť, ktorá neni typu X (pri eth sietiach eth chápeme sa), tak sa nič nedeje, continue
    c. Vytvorí sa nový `NodeContainer`, uloží sa do atribúty objektu `xyz_nodes` a naplní sa uzlami cez metódu `.Add(node)`
    d. Vytvorí sa ns3 helper metóda, pri ETH sietach konkrétne `CsmaHelper`
    e. Cez metódy `.Set()` a `.SetAttribute` sa nakonfiguruje
    f. zavolá metódu `helper.Install(self.xyz_nodes)` a výsledné sieťové rozhrania uloží do triedy do `xyz_devs`
    g. Na vytvorené rozhrania nainštaluje IP stack cez `ip_helper`, ktorý získal v konštruktore
    h. Podľa konfiguračného súboru rozhraniam pridelí IP adresu
    i. zavolá sa `enable_trace` funkcia z utilít, aby komunikácia generovala pcap a ascii logy.

Logika inštalácie sietí sa realizuje následovne:

IPUtil

Súbor s triedou, ktorá obsahuje IP Stack, triedy na riešenie smerovania. Smerovanie je v tejto verzii aplikácie ešte veľmi limitované (čítaj nefunkčné), takže hlavná funkcionalita je realizácia P2P konekcíi medzi back-bone uzlami 2ch rôznych sietí. Slúži na to metóda `connect` a metóda `install_connections`, ktorá realizuje pripojenie rovno z konfiguračného objektu. Zároveň si pamätá rozsahy IP-čok, ktoré už použila, aby nedošlo k sietiam s rovnakými IP adresami.

SDN manager

Trieda SdnManager je zodpovedná za vytvorenie a inicializáciu SDN prepínačov na základe konfigurácie. V konštruktore triedy prechádza konfiguráciou a pre každý uzol zistí, či je typu "sdn". Ak áno, vytvorí inštanciu triedy SdnSwitch s identifikátorom uzla.

Trieda SdnSwitch predstavuje samotný SDN prepínač. V konštruktore triedy sa inicializujú potrebné premenné a zavolá sa privátna metóda _setup(), ktorá slúži na konfiguráciu prepínača a vytvorenie spojení so susednými uzlami.

Metóda _setup() najskôr inicializuje kontajnery pre uzly a susedné uzly získané zo spravovacieho kontextu. Potom sa pomocou triedy ns.csma.CsmaHelper() vytvárajú spojenia (links) typu csma medzi prepínačom a každým susedným uzlom. Konfigurujú sa parametre linky, ako dátová rýchlosť a oneskorenie.

Následne sa vytvárajú kontajnery pre sieťové zariadenia pre terminálové zariadenia a prepínač. Pre každého susedného uzla sa vytvorí linka csma a príslušné zariadenia sa pridajú do kontajnerov terminalDevices a switchDevices. Linky sa tiež ukladajú do zoznamu links.

Nakoniec sa vytvára SDN prepínač pomocou triedy ns.openflow.OpenFlowSwitchHelper(). Na základe konfigurácie sa vyberie typ ovládača (controller), buď DropController alebo LearningController. Prepínač sa nainštaluje do spravovacieho kontajnera s príslušnými sieťovými zariadeniami a ovládačom.

Trieda SdnSwitch teda slúži na inicializáciu a konfiguráciu SDN prepínača, vrátane vytvorenia spojení so susednými uzlami a inštalácie ovládača.

Pomocné súbory

- dubaktranslator.py
    - Prekladá názvy aplikácii ako sa posielajú z frontendu na také, ako očakávam v scenári. Užitočné, keď si každý kódi svoje a potom to chceme spojiť bez toho, aby sme prepisovali zvyšok kódu.
- util.py
    - súbor pre všeobecne pomocné funkcie. Zatiaľ má len 1 funkciu na preklad WIFI štandardu zo stringu na ns3 objekt.
- loghelper.py
    - súbor na výpis logov. Okrášlenie print() funkcie, len pred ne pridá `[info]`, `[log]` alebo `[error]` označenia, pre krajší výpis a prehľadnejší kód.
- attributemanager.py
    - trieda, ktorá inštaluje atribúty do XYZHelper tried z ns3. Tie sa volajú cez .Set(), prípadne .SetAttribute() metódy. Cieľ tejto triedy je zjednoduchšiť kód v moduloch na realizáciu sietí a aplikácii tak, že túto časť logiky premiestnime sem.
- context.py
    - Súbor, v ktorom sa ukladajú elementy, ktoré sa referencujú v ostatných moduloch. Čiže vlastne len súbor, v ktorom sú globálne premenné. Napríklad konfiguračný súbor alebo globálny kontajner uzlov.
- tracehelper.py
    - obsahuje funkciu pre pridanie ASCII a PCAP logov pre vstupný kontajner rozhraní uzlov.
- ns2_node_util.py
    - Parsuje ns2mobility tcl-ko. Má metódy na zistenie počtu uzlov v simulácii a kedy ktorý uzol začal byť aktívny a kedy odyšiel zo simulácie.

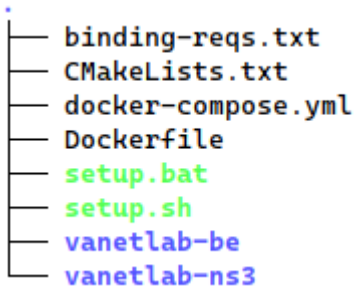
Backend && NS3 development prostredie

Keďže inštalácia NS3 je nepredvídateľná, tak sme pre development prostredie zostrojili Dockerfile, ktorý zaistí, že budeme používať tie rovnaké prostriedky. Priamo v docker kontajneri je nainštalované NS-3 a aj python s potrebnými knižnicami pre zbehnutie backendu.

Zdrojový kód “modulárneho” ns-3 scenáru a back-endu je namapovaný cez docker volumes, aby sa po každej zmene kódu nemusel rebuildiť docker image. Rebuild docker imagu je potrebný len ak sa na BE pridá nová pip knižnica, a aj to len posledné vrstvy obrazu, ns3 nie je potrebné kompilovať znovu, lebo docker je celkom rozumne navrhnutý a nakódený.

Spustenie je teda jednoduché (alebo určite jednoduchšie ako manuálne inštalovanie a konfigurovanie ns3)

- Nainštalujeme docker a docker-compose podľa oficiálnych inštrukcií
- Naklonujeme si vanetlab-deploy repozitár (https://github.com/vanetnuggets/vanetlab-deploy).
- Prejdeme do naklonovaného priečinku a do neho naklonujeme aj backend aj ns3 scenár tak, aby štruktúra priečinku vyzerala tak ako na obrázku:



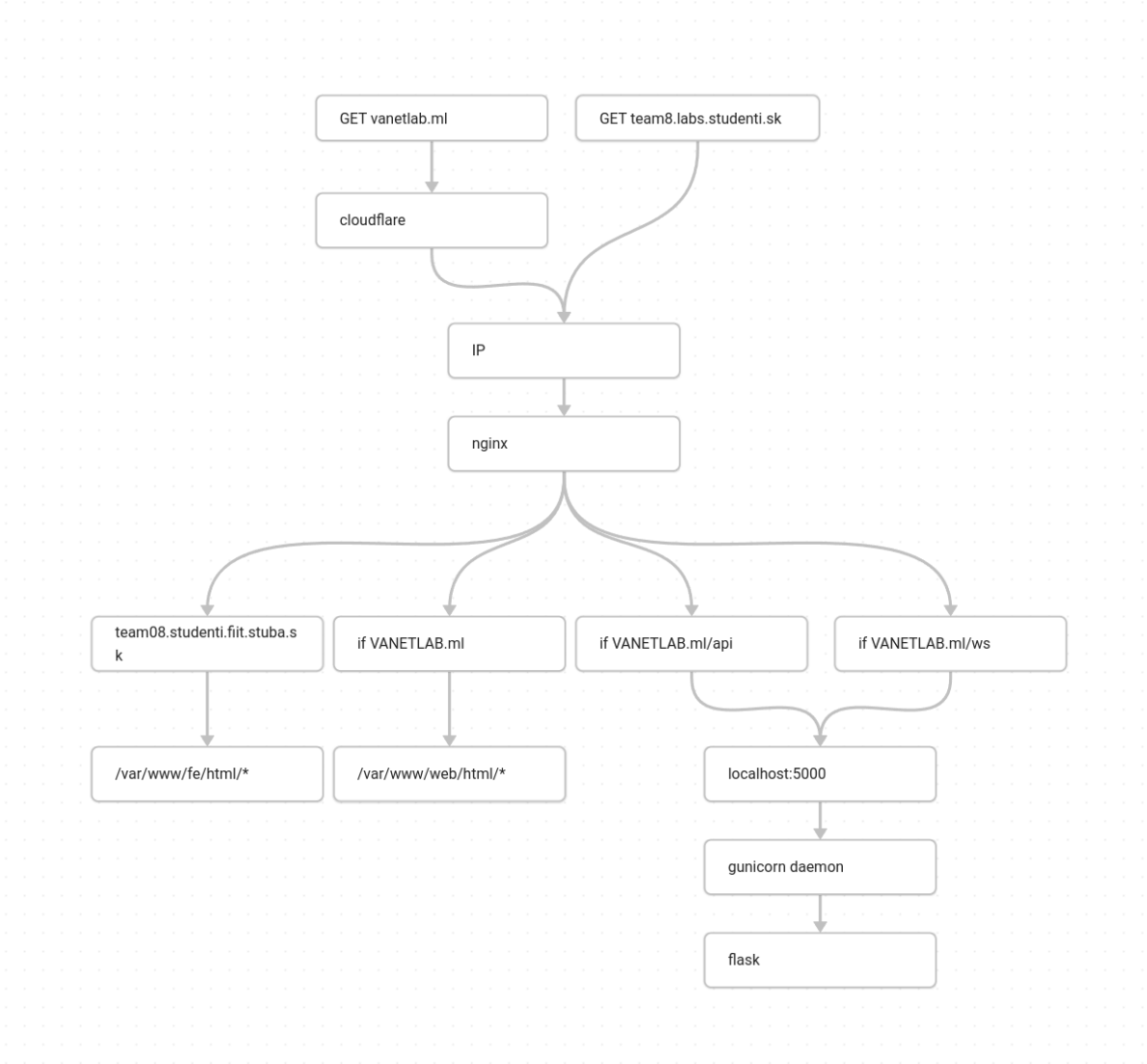
- príkazom `docker-compose up –build` sa začne vytvárať kontajner. prvý krát to trvá dosť dlho, keďže sa inštalujú všetky dependencies a ns3 samotné sa musí skompilovať.
- po dokončení procesu kontajner beží na pozadí, ale žiadna vanetlab aplikácia v ňom ešte nie je bežiaca
- príkazom `docker-compose run –rm app` spustíme v kontajneri bash.
- BE sa spustí príkazom `cd /app && flask run –host 0.0.0.0` v kontajneri
- pri nasledujúcich spusteniach pri príkaze `docker-compose up` už nie je potrebné zadávať prepínač `--build`, lebo image je už zbuildený.

Tento proces je možné prípadne upraviť tak aby nevyžadovalo žiadnu interakciu, ale pri našom vývoji sme zistili že potrebujeme debugovať veci v kontajneri (ns3 ehm ehm) častejšie ako len používať BE, takže sme to pripravili tak aby sa pri spustení kontajneru spustil bash a nie rovno BE.

Pre modifikáciu zdrojového kódu ľubovoľným textovým editorom editujeme súbory vo `vanetlab-be` a `vanetlab-ns3` priečinkoch. Keďže sa mapujú do docker kontajneru ako volumes, tak lokálne zmeny budú prenesené rovno do kontajnera. NS3 scenár stačí uložiť a pri jeho volaní sa zmeny aplikujú, ale po zmene back-end kódu treba flask server reštartovať (CTRL-C a znovu `flask run –host 0.0.0.0`). Mal by to riešiť flask development mód, ale stáva sa nám, že flask nedetekuje zmeny a potom debugujeme neaplikované zmeny, čo je blbé. Preto vždy reštart.

Nasadenie

Keďže na študentskom stroji je málo miesta a už len docker image pre ns3 s back-endom má 6GB, aplikáciu nasaďujeme na študenstký stroj bez kontajnerov, cez nginx, s manuálne nainštalovaným sumo, ns3 a všetkými dependencies.



Prikladáme aj nginx konfiguráciu. Je nutné poznamenať, že back-end beží pod gunicorn deamonom na lokalhoste na porte 9000 a nginx na neho preposiela všetky požiadavky s path prefixom `/api` a `/ws`.

```
map $http_upgrade $connection_upgrade {
        default upgrade;
        '' close;
    }

server {
```

```
        listen 80;
        server_name vanetlab.ml www.vanetlab.ml;

        root /var/www/fe/html;
        index index.html index.htm index.nginx-debian.html;
        location /api {
                proxy_pass http://localhost:9000;
                proxy_set_header Host $host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
           }
           location /ws {
                proxy_pass http://localhost:9000;
                proxy_http_version 1.1;
                proxy_set_header Upgrade $http_upgrade;
                proxy_set_header Connection $connection_upgrade;
                proxy_set_header Host $host;
        }
           location / {
                try_files $uri $uri/ =404;
        }
}
```

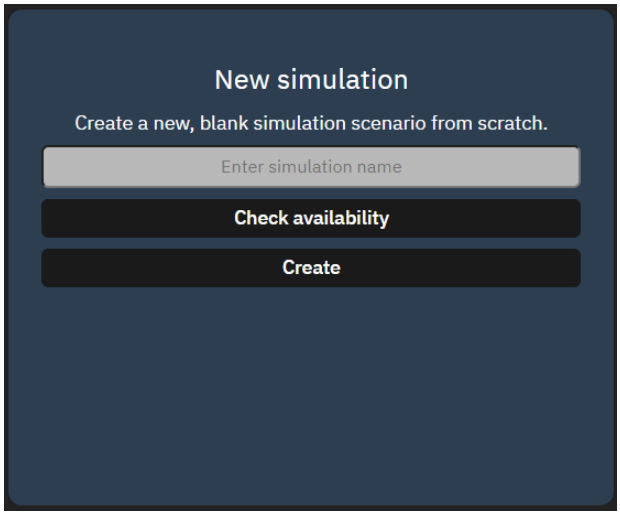
Používateľská príručka

Táto sekcia popisuje akú funkciu a ako sa používajú jednotlivé časti simulátor VanetLAB.

Úvodná strana

Po otvorení vanetlabu sa používateľovi zobrazí úvodná stránka s menu.

New simulation

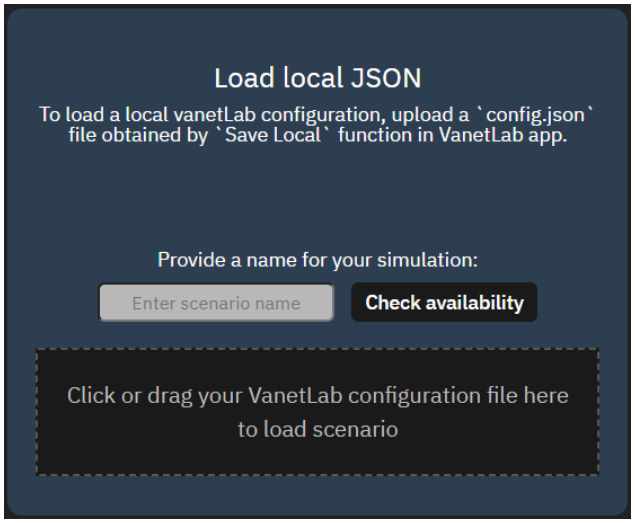


Kliknutím na novú simuláciu si používateľ vie vytvoriť nový simulačný scenár. Najprv je potrebné zadať meno simulácie a overiť dostupnosť, či sa takéto menu už nepoužíva. Ak áno pri uložení bude prepísaný.

Load XML from SUMO

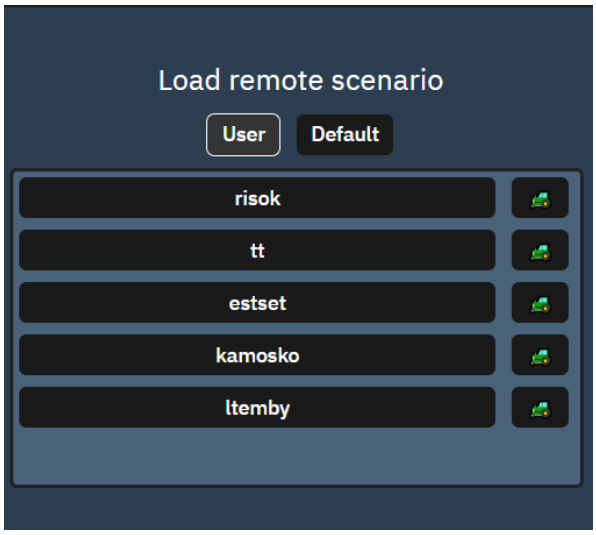
Možnosť load XML from SUMO slúži na načítanie scenára vygenerovaného nástrojom SUMO. Do nástroja sa vkladá xml súbor.

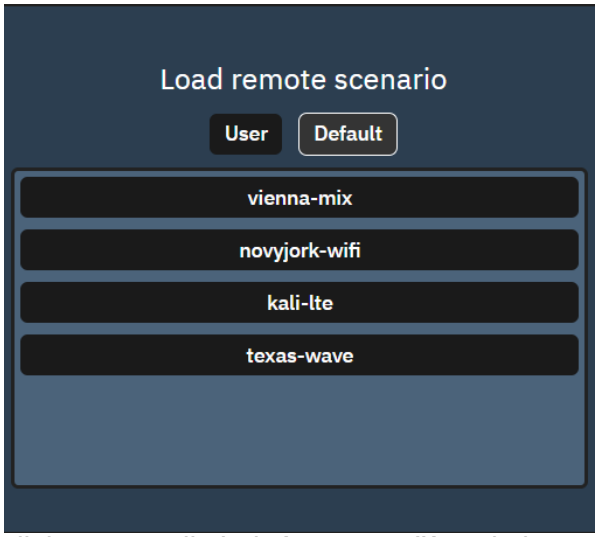
Load local JSON



Možnosť load local JSON slúži na načítanie scenára, ktorý bol vytvorený pomocou nášho nástroja. Takto vytvorený scenár je možné uložiť v konfiguračnom súbore JSON, ktorý je potom možné použiť na lokálne načítanie.

Load remote scenario





Možnosť na načítanie scenára uloženého na vzdialenom stroji. Jedná sa o používatelmi vytvorené scenáre a predefinované scenáre. Používateľské scenáre je možné vymazať ak máme zadaný správny API kľúč. Po kliknutí na scenár sa nám otvorí editor s konkrétnou konfiguráciou.

About

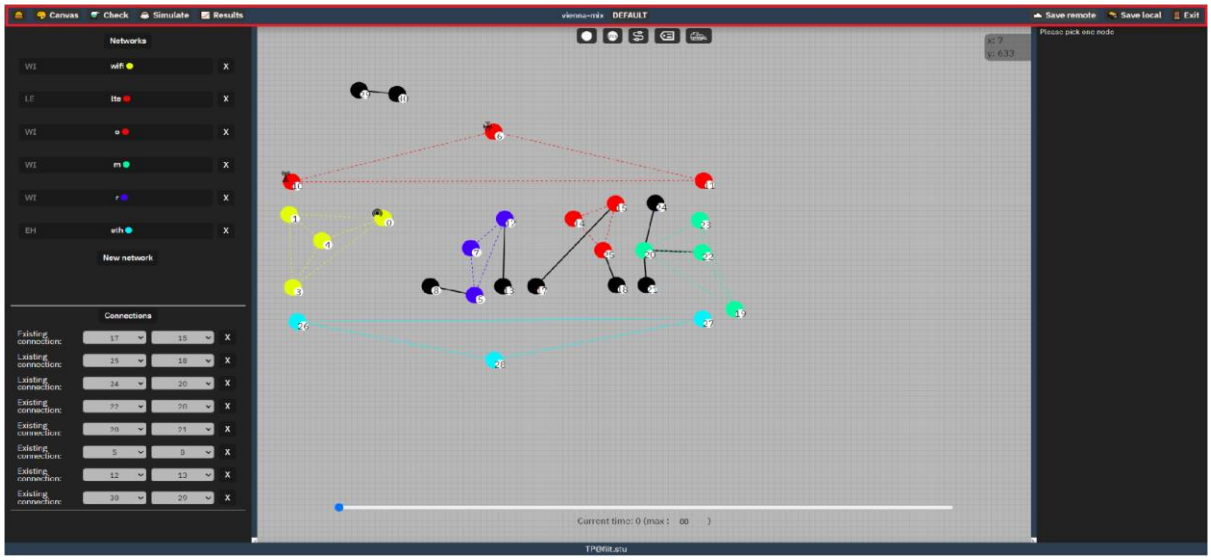
Sekcia opisujúca projekt, použité technológie a jeho vlastnosti.

Api key

Simulácia scenárov vyžaduje veľa systémových prostriedkov. Preto nechceme, aby hocikto mohol simulovať čo len chce, aplikácia by totiž bola náchylná na DoS útoky. Na to aby mohol používateľ simulovať alebo ukladať scenáre vzdialene, je potrebné aby mal nastavený API kľúč. Bez tohto kľúča je možná iba interakcia so scenármi s možnosťou lokálneho uloženia. Defaultný kľúč je: [redacted].

Simulátor

Top bar



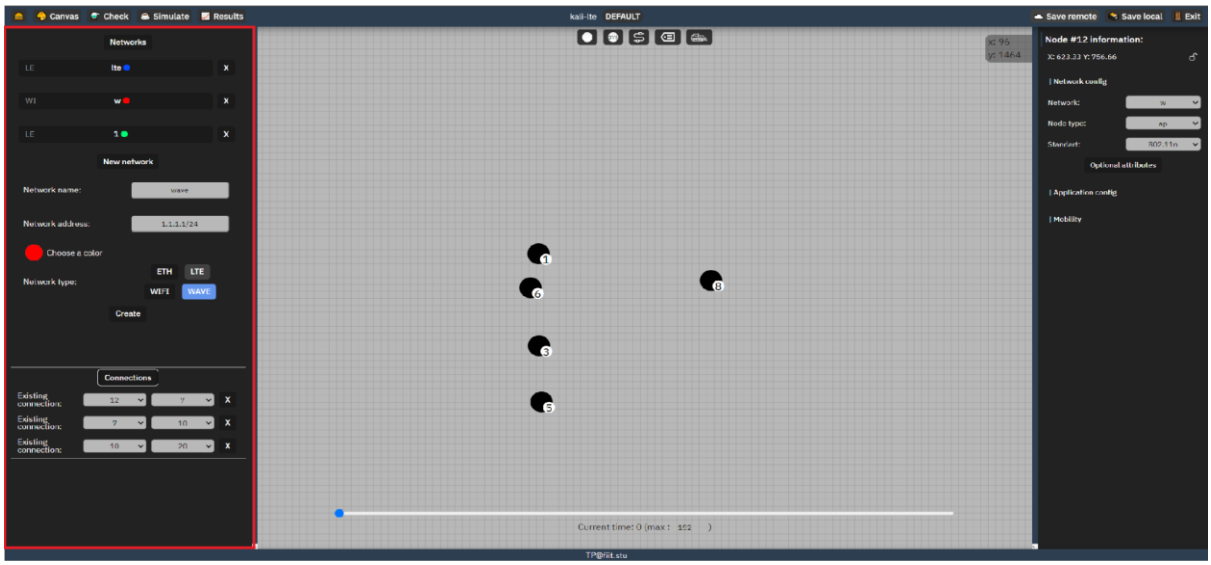
Nachádzajú sa tu ovládacie prvky.

Canvas nás prepne na zobrazenie plátna. Check nám skontroluje či je naša konfigurácia správna. Simulate spustí simuláciu(je potrebné mať pridaný validný api kľúč). V sekcií results uvidíme výsledky simulácie.

V strese sa nachádza názov simulačného scenáru. Ak je scenár preddefinovaný tak sa za názvom objaví DEFAULT značka.

Save remote uloží scenár na vzdialenom serveri, možnosť je dostupná iba s API kľúčom a ak scenár nie je default. Save local nám stiahne konfiguračný súbor. Exitom sa vrátime na úvodnú obrazovku.

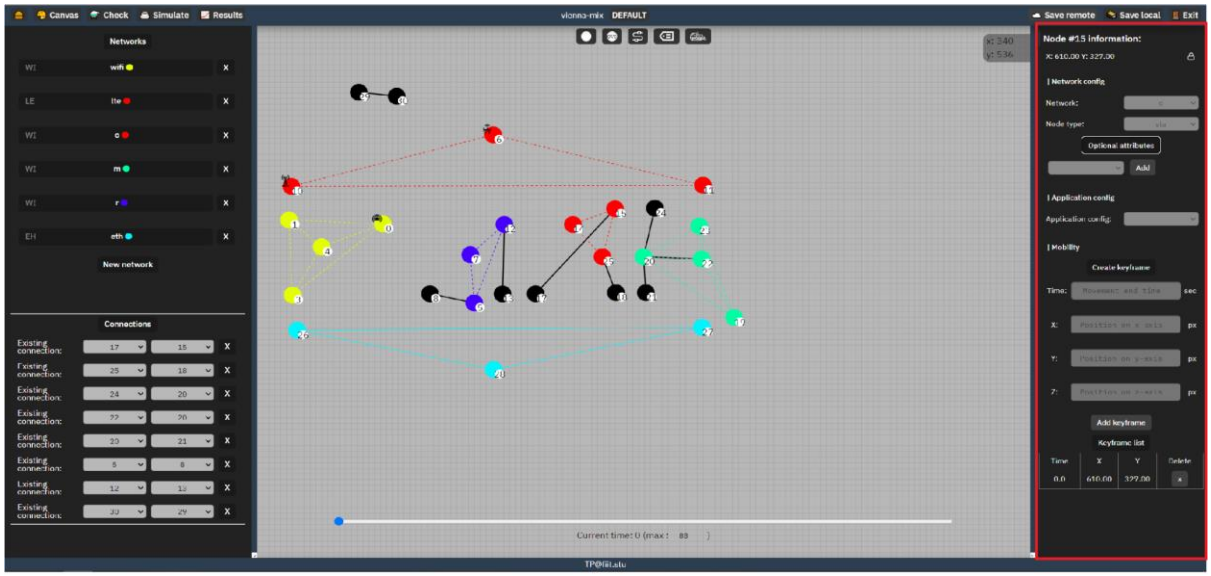
Left drawer



Ľavé menu je zodpovedné za vytváranie nových sietí. Každá sieť nosí mať unikátny názov a

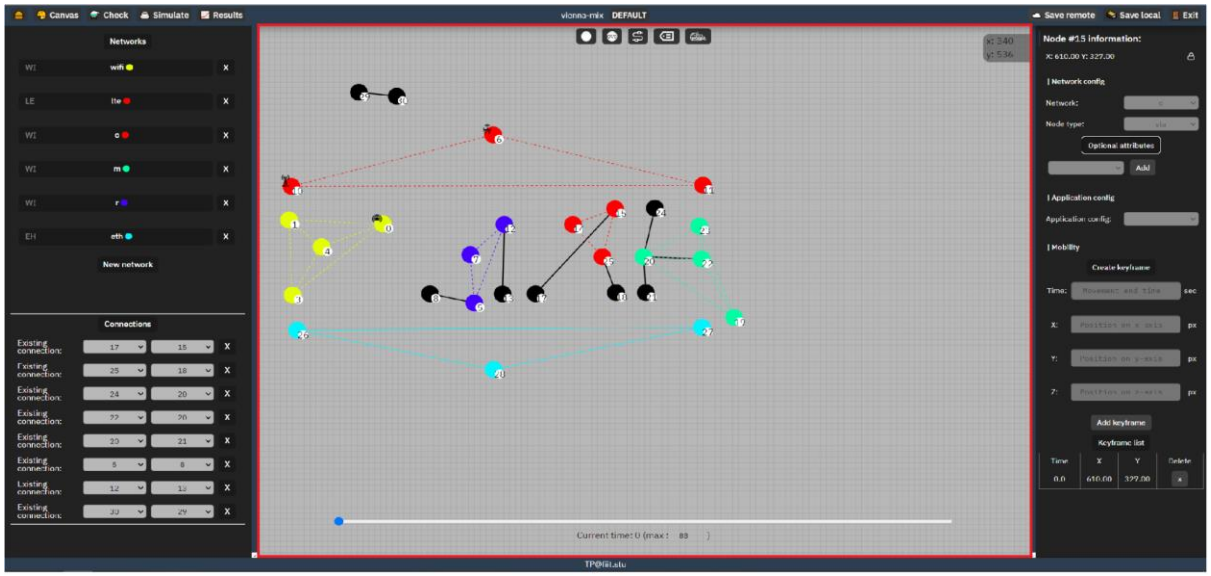
ip adresu v CIDR notácii. Simulátor je limitovaný pre maximálne jednu LTE kvôli limitáciam simulátora NS3. Každej sieti je možné nastaviť farbu uzlov, aby boli jasne odlíšiteľné. Pod sieťami sa nachádza zoznam p2p konekcií. Zoznam má dva stĺpce, odkial a kam ide prepojenie. Prepojenia je taktiež možné zo zoznamu mazať pomocou kliknutia na tlačidlo X.

Right drawer



Pravé menu zodpovedá za konfiguráciu uzlov. Po kliknutí na uzol sa v tomto menu zobrazia jeho príslušné informácie. Na zmenu parametrov je potrebné najprv kliknúť na tlačidlo zámku pre odomknutie menu v pravom hornom rohu. Po odomknutí je možné konfigurovať L2(nastavenie prislúchajúcej siete,...), L3(nastavenie komunikácie,...) vrstvy a mobilitu. Vrstvy sú závislé na type uzla. Mobilitu majú všetky uzly a zodpovedá za pohyb uzla v čase.

Canvas

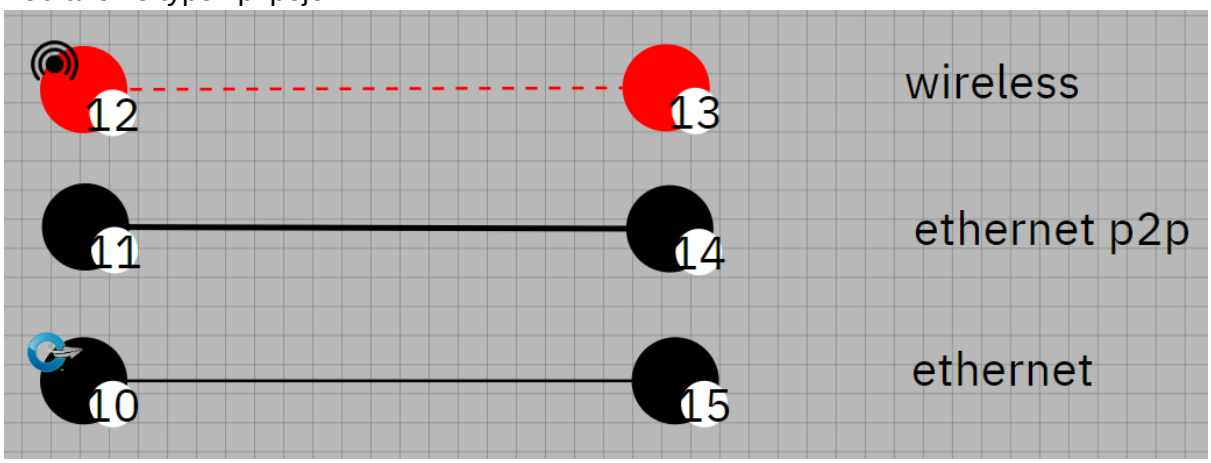


Jedná sa o samotné simulačné prostredie. V strede na vrchnej časti sa nachádzajú prvky plátna. Pridávanie uzlov, pridanie openVSwitchu, pridanie p2p konekcií, pridanie lebelu a buldozér.

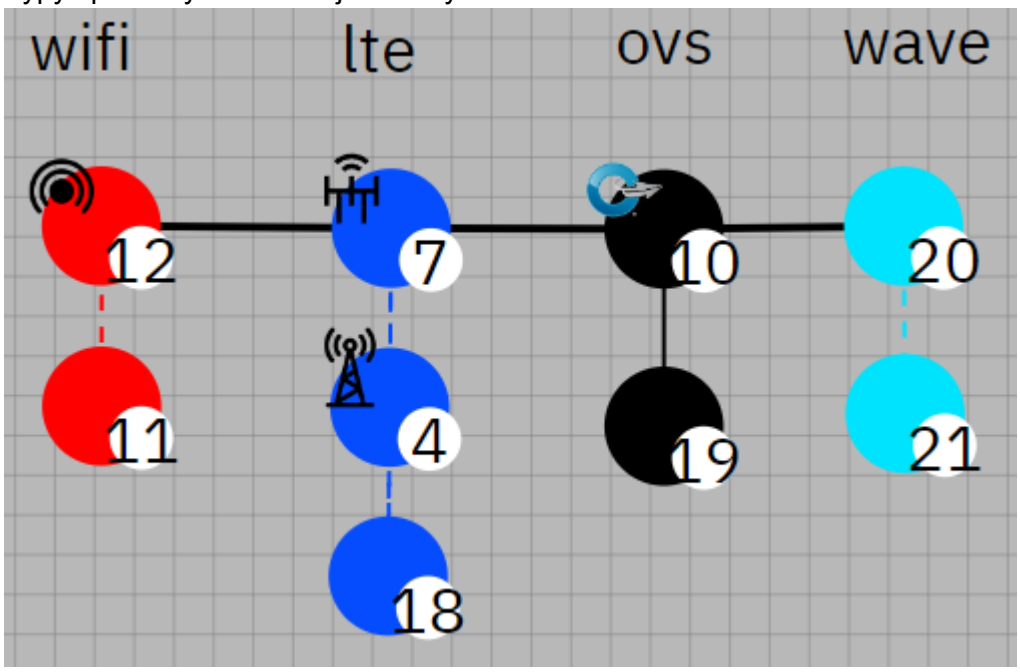
- Node, switch a label pridáme tak, že vyberieme možnosť a klikneme lubovoľne na plátno. Po kliknutí sa nám zobrazí nový prvok na plátne.
- P2p konekcie fungujú interaktívne, klikneme na prvy a následne druhý node, ktorý chceme prepojiť. Toto spojenie je fyzické pomocou ethernetového káble.
- Buldozérom mažeme nody, switche a labely.

Plátnom je možné pohybovať myšou, v pravom hornom roku sú zobrazené aktuálne súradnice kurzora. V spodnej časti je posuvný prvok na posúvanie času simulácie. Zoom reaguje na koliesko na myši. Prvky na plátne sa posúvajú potiahnutím.

Zobrazenie typov pripojení

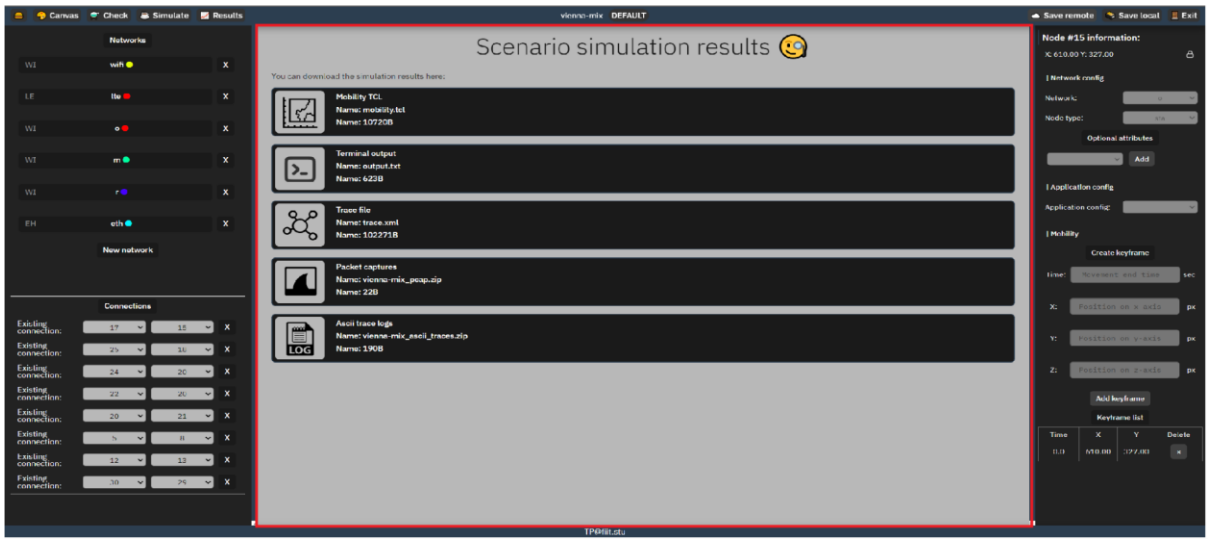


Typy špeciálnych uzlov v jednotlivých sieťach



Wifi -> ap(access point), obyčajný uzol(sta-stanica)
Lte-> pgw(packet gateway) node, enb(evolved node B) node, obyčajný uzol(ue-user equipment)
Ovs-> ovs(openVSwitch), obyčajný uzol(ordinary node)
Wave-> obyčajný node, obyčajný uzol(ordinary node)

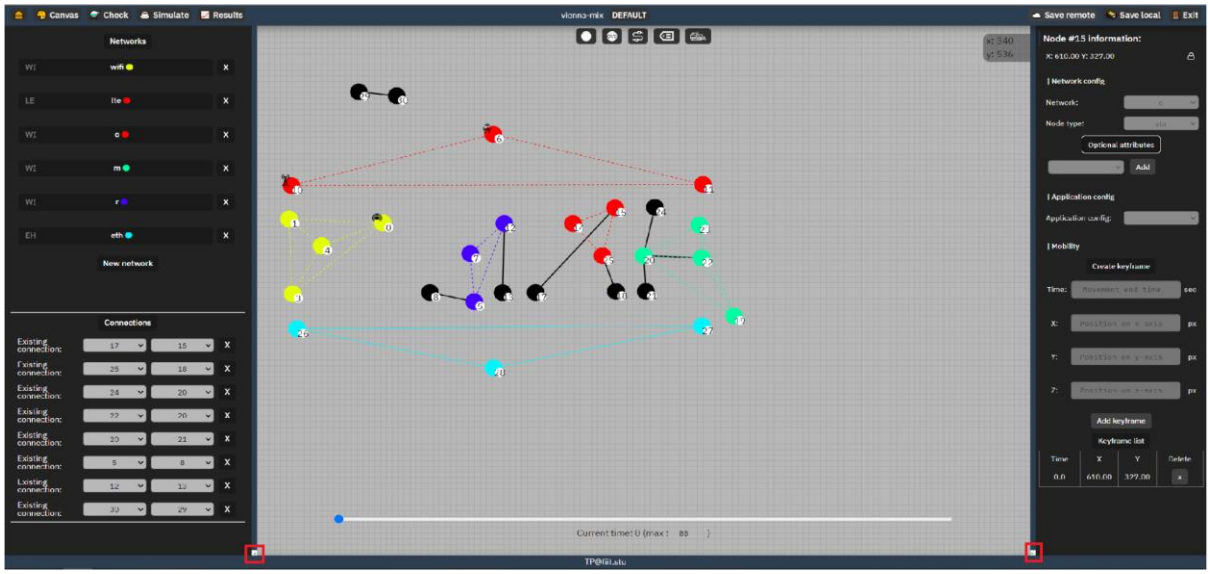
Results



Sekcia results nám ponúka stiahnutie výsledkov odsimulovaného scenára. K dispozícií sú súbory mobility uzlov vo formáte ns2 tcl, terminálový output vo forme txt, trace v xml pre otvorenie a analýzu v programe NetAnim, pcapy jednotlivých komunikácií v zipe pre analýzu vo wiresharku a ASCII logy, pre otvorenie v PacketExploreri. Súbory sú stiahnuté po kliknutí.

Simulátor si uchováva výstup z predchádzajúcej simulácie, preto je možné stiahnuť výstup aj bez simulácie ak už scenár niekedy v minulosti zbehol úspešne. Táto možnosť sa nachádza v taktiež v results ak nebola spustená simulácia.

Zmena veľkosti



Pravý a ľavý panel je možné skryť potiahnutím, čím získame väčšie plátno.