UNIVERSITATEA „SAPIENTIA” DIN CLUJ-NAPOCA

FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE,

TÎRGU-MUREȘ

SPECIALIZAREA CALCULATOARE

MODELAREA REȚELELOR DE SENZORI FĂRĂ FIR

PROIECT DE DIPLOMĂ

Coordonator științific:

Ș.l.dr.ing Turos László-Zsolt

Absolvent:

Czimbalmos Ákos

2023

UNIVERSITATEA „SAPIENTIA” din CLUJ-NAPOCA **Viza facultății:**
Facultatea de Științe Tehnice și Umaniste din Târgu Mureș
Specializarea: **Calculatoare**

LUCRARE DE DIPLOMĂ

Coordonator științific:	Candidat: **Czimbalmos Ákos**
ș.l. dr. ing. Turos László-Zsolt	Anul absolvirii: **2023**

a) Tema lucrării de licență:
MODELAREA REŢELELOR DE SENZORI FĂRĂ FIR

b) Problemele principale tratate:
- Studierea cadrului OMNeT++
- Modelarea şi simularea diferitelor topologii a reţelelor de senzori şi a protocoalelor de rutare Gossiping, LEACH, GAF

c) Desene obligatorii:
- Schema modulelor simple şi complexe în OMNet++
- Schema cu topologiile realizate în cadrul ONNet++

d) Softuri obligatorii:
- Realizarea aplicaţiei pentru modelarea şi simularea diferitelor topologii a reţelelor de senzori şi a protocoalelor de rutare

e) Bibliografia recomandată:
-Túrós László-Zsolt, Székely Gyula: Érzékelők és mérőhálózatok, Scientia kiadó, 2022
-W. Yen, C.-W. Chen és C.-h. Yang, „Single gossiping with directional flooding routing protocol in wireless sensor networks,” 2008
- Javaid, M., Haleem, A., Rab, S., Singh, R. P., & Suman, R. (2021b). Sensors for daily life: A review. Sensors International

f) Termene obligatorii de consultații: săptămânal
g) Locul și durata practicii: Universitatea „Sapientia” din Cluj-Napoca,
Facultatea de Științe Tehnice și Umaniste din Târgu Mureș
Primit tema la data de: 31.03.2022
Termen de predare: 27.06.2023

Semnătura Director Departament

Semnătura coordonatorului

Semnătura responsabilului programului de studiu

Semnătura candidatului

Declarație

Subsemnatul Czimbalmos Ákos, absolvent a specializării Calculatoare, promoția 2023 cunoscând prevederile Legii Educației Naționale 1/2011 și a Codului de etică și deontologie profesională a Universității Sapientia cu privire la furt intelectual declar pe propria răspundere că prezenta lucrare de licență/proiect de diplomă/disertație se bazează pe activitatea personală, cercetarea/proiectarea este efectuată de mine, informațiile și datele preluate din literatura de specialitate sunt citate în mod corespunzător.

Tg-Mureș

Data: 27.06.2023 Czimbalmos Ákos

Semnătura.....

Modelarea rețelelor de senzori fără fir

Extras

Scopul documentului "Modelarea rețelor de senzori fără fir" este de a demonstra utilizarea protocoalelor de rutare în proiectarea și funcționarea rețelelor. Protocoalele de rutare joacă un rol esențial în selectarea și rutarea pachetelor de date pentru a ajunge la destinație prin rețea. Protocoalele de rutare se bazează pe diferiți algoritmi, care au caracteristici și principii de funcționare diferite. Scopul acestei documentații este de a crea un model pentru a vizualiza aplicarea protocoalelor de rutare, propagarea pachetelor și evaluarea datelor. De asemenea, oferă un mijloc de a compara performanța și eficiența diferitelor protocoale de rutare. Protocoalele de rutare sunt esențiale pentru funcționarea rețelelor fără fir. O înțelegere temeinică a algoritmilor și a implementărilor acestora va permite o mai bună înțelegere a funcționării rețelelor, detectarea potențialelor defecțiuni și o planificare, optimizare și funcționare eficiente. Documentația ajută la înțelegerea funcționării, avantajelor și limitărilor diferitelor protocoale de rutare și ajută la simularea rețelelor prin furnizarea de coduri detaliate pentru simulări scrise în cadrul OMNeT++. Sunt descrise trei protocoale de rutare, unul centrat pe date (Gossiping), unul ierarhic (LEACH) și unul bazat pe locație (GAF), compararea lor, concluziile trase și performanța și eficiența protocoalelor de rutare studiate.

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR
SZÁMÍTÁSTECHNIKA SZAK

VEZETÉK NÉLKÜLI ÉRZÉKELŐHÁLÓZATOK MODELLEZÉSE

DIPLOMADOLGOZAT

Témavezető:
Turos László-Zsolt,
egyetemi adjunktus

Végzős hallgató:
Czimbalmos Ákos

2023

Kivonat

A Vezeték nélküli érzékelőhálózatok modellezése című dokumentáció célja, hogy bemutassa az útválasztó protokollok alkalmazását a hálózatok tervezésében és működtetésében. Az útválasztó protokollok kulcsfontosságú szerepet játszanak az adatcsomagok kiválasztásában és irányításában, hogy azok célba érjenek a hálózaton keresztül. Az útválasztó protokollok különböző algoritmusokon alapulnak, amelyeknek különböző jellemzői és működési elvei vannak. A dokumentáció célja egy modell létrehozása, amely segítségével vizualizálhatjuk az útválasztó protokollok alkalmazását, a csomagok terjedését és az adatok kiértékelését. Emellett lehetőség nyílik a különböző útválasztó protokollok teljesítményének és hatékonyságának összehasonlítására. Az útválasztó protokollok kritikus jelentőséggel bírnak a vezeték nélküli hálózatok működtetésében. Az algoritmusok és azok implementációinak alapos megértése lehetővé teszi a hálózatok működésének jobb átlátását, esetleges hibák felderítését, valamint hatékony tervezést, optimalizálást és üzemeltetést. A dokumentáció segítséget nyújt a különböző útválasztó protokollok működésének megértésében, előnyeinek és korlátainak feltárásában, valamint segítséget nyújt a hálózatok szimulációjának elkészítésében, mivel részletesen tárgyalja az OMNeT++ keretrendszerben íródott szimulációk kódját. Három útválasztó protokollról olvashatunk, egy adatközpontúról (Gossiping), egy hierarchikusról (LEACH) és egy helyalapúról (GAF), ezek összehasonlításáról, következtetések levonásáról és a vizsgált útválasztó protokollok teljesítményéről és hatékonyságáról.

Kulcsszavak: hálózat, protokoll, útválasztás

Abstract

The purpose of the document Wireless Sensor Networks Modelling is to demonstrate the use of routing protocols in the design and operation of networks. Routing protocols play a key role in the selection and routing of data packets to reach their destination through the network. Routing protocols are based on different algorithms, which have different characteristics and operating principles. The aim of this documentation is to create a model to visualise the application of routing protocols, the propagation of packets and the evaluation of data. It also provides a means to compare the performance and efficiency of different routing protocols. Routing protocols are critical to the operation of wireless networks. A thorough understanding of the algorithms and their implementations will allow for a better understanding of the operation of networks, the detection of potential failures, and efficient planning, optimisation and operation. The documentation helps to understand the operation, advantages and limitations of different routing protocols, and helps to simulate networks by providing detailed code for simulations written in the OMNeT++ framework. Three routing protocols are described, a data-centric one (Gossiping), a hierarchical one (LEACH) and a location-based one (GAF), their comparison, conclusions drawn and the performance and efficiency of the routing protocols under study.

Keywords: network, protocol, routing

1. Tartalom

2. Ábrajegyzék

3. Bevezető

A modern hálózatok tervezése és működtetése során az útválasztó protokollok alapvető szerepet játszanak. Az útválasztó protokollok meghatározzák, hogyan választják ki és irányítják az adatcsomagokat a hálózatban, hogy célba érjenek. Az útválasztó protokollok különböző algoritmusokon alapulnak, amelyeknek különböző jellemzőik és működési elveik vannak.

A cél egy modellen keresztül vizualizálni az útválasztó protokollok alkalmazását, a csomagok terjedését és az adatok kiértékelését. Ezenkívül összehasonlítani a különböző útválasztó protokollok teljesítményét és hatékonyságát.

Célkitűzések:

- Az útválasztó protokollok különböző algoritmusainak kivitelezése
- A csomagok terjedésének vizuális megjelenítése és elemzése
- Az adatok kiértékelése és az eredmények összehasonlítása a különböző útválasztó protokollok között
- Következtetések levonása a vizsgált útválasztó protokollok teljesítményéről és hatékonyságáról

Az útválasztó protokollok kritikus szerepet játszanak a vezeték nélküli hálózatok működtetésében. Az algoritmusok és az implementációik alapos megértése lehetővé teszi számunkra, hogy jobban átlássuk egy hálózat működését, feltárjuk azok esetleges hibáit, valamint hatékonyan tervezzünk, optimalizáljunk és üzemeltessünk hálózatokat. A dokumentáció segítséget nyújt a különböző útválasztó protokollok működésének megértésében, valamint azok előnyeinek és korlátjainak feltárásában.

4. Elméleti megalapozás és szakirodalmi tanulmány

4.1. Szakirodalmi tanulmány

Első sorban meg kellett értenem, hogy mit is jelent egy vezeték nélküli érzékelőhálózat (WSN), ehhez nagy segítséget nyújtott az [1] szakirodalmi tanulmány, amely alapjaiban tárgyalja egy WSN felépítését, működését, valamint tisztázza az alapfogalmakat. Továbbá megismertem a WSN típusokat, és a köztük lévő különbségeket és a fő alkotóelemeit, mint például az érzékelő csomópont, vagy a bázisállómás. A tanulmány a WSN alapismereteinek lefektetése után szót ejt az útválasztásról is a hálózaton, ebből megtudtam, hogy mit jelent egy útválasztó protokoll, milyen típusai vannak és a különböző típusok milyen logika szerint működnek. Viszont ez

önmagában nem volt elegendő arra, hogy elég információt szerezzek egy specifikus protokoll szimulálásához.

Ezt követően tudományos cikkeken keresztül tanulmányoztam a különböző útválasztó protokollok felépítését, elvét és gyakorlati működését. Ezek alapján mindegyikről kiderült, hogy az adott protokollt milyen közegben érdemes használni, hol működik a legnagyobb hatásfokkal, mik az esetleges hátrányai vagy előnyei.

Az első és legegyszerűbb tanulmányozott és leszimulált protokoll a Pelyka – Gossiping, ennek a megértésére a [2] és [9] tanulmáynok nyújtottak segítséget, valamint a szimuláció elkészítéséhez az OMNeT++ oldalán levő segédanyag [6].

A [1] és [2] tanulmányok a LEACH (Low Energy Adaptive Clustering Hierarchy) protokoll megértését tették lehetővé. Ez a protokoll a csomópontjait klasszterekbe szervezi, minden klaszterben van egy klaszter fej, aminek feladata a klaszterben levő csomópontok információjának a továbbítása a bázisállomás felé.

A hálózati protokollok szimulálására az OMNet++ nevű fejlesztői környezetet használtam, ami egy moduláris, komponens-alapú C++ szimulációs könyvtár és keretrendszer, és bármilyen hálózati szimuláció elkészítésére alkalmas akár vezetékes, akár vezeték nélküli (mint pl: szenzorhálózatok, vezeték nélküli ad hoc hálózatok, internet protokollok, teljesítmény modellezés, fotónikus hálózatok). Az OMNet++ dokumentációját [3] felhasználtam a komponensek, modulok, és beépített osztályok tanulmányozására, valamint ezek használati lehetőségeire, továbbá az osztályok metódusainak feltárására és használatára. Hálózatok létrehozására, szimulálására és konfigurálására az OMNet++ weboldalán fellelhető „Learn OMNeT++ with TicToc” [4] nevű oktató cikket használtam, amiben lépésről lépésre megtudtam, hogy ebben a fejlesztői környezetben milyen fájl típusok vannak, mire valók, hogyan lehet kezelni a hálózatokat, és magát a szimulációt miként lehet futtatni.

4.2. Elméleti alapok

4.2.1. Vezeték nélküli érzékelőhálózatok (WSN – Wireless Sensor Network)

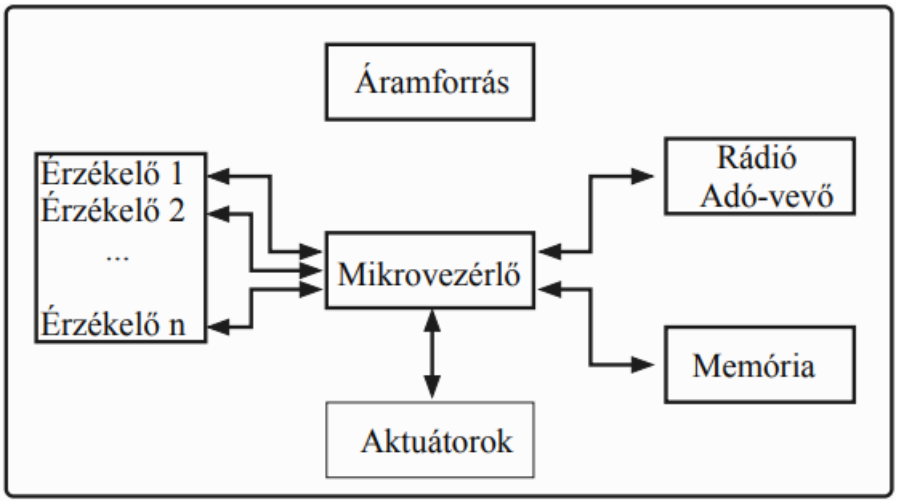
A WSN olyan érzékelő csomópontok gyűjteménye, amelyek egy hálózatba csoportosulnak és valamilyen információt közvetítenek egymás közt vagy egy bázisállomás felé. A bázisállomás egy olyan adó-vevő egység, amely kiemelt szerepet játszik a hálózatban. Általában egy központi csomópont, amely felelős a hálózat irányításáért és kommunikációjáért,

főbb feladata a hálózat többi csomópontja által begyűjtött adatok összegyűjtése. A bázisállomás általában nagyobb energiaellátást igényel, mint a többi érzékelő csomópont.

Az érzékelő csomópontok (sensor node) kis méretű eszközök, amelyek több részből állnak és jelentős szerepet játszanak a vezeték nélküli érzékelőhálózatokban. Ezek a csomópontok képesek különböző fizikai vagy környezeti jellemzőket érzékelni, mint például fényerősség, hőmérséklet, hang, páratartalom stb., és ezeket az adatok továbbítják a hálózaton keresztül egy másik csomópont vagy a bázisállomás felé [5]. A felépítésük összetett, több részből állnak:

- **mikrovezérlő**: egy mikroprocesszorból áll, és a lapkájára integrált perifériákból, feladata vezérelni a csomópontban levő érzékelőket és összehangolni a többi komponenssel a megfelelő működés elérése érdekében, valamint felelős az adatok feldolgozásáért és a döntéshozatalért
- **kommunikációs modul (adó-vevő egység)**: ez a modul lehetővé teszi a kommunikációt az érzékelő csomópontok és a bázisállomás között. Ez a modul lehet vezeték nélküli rádiómodul, amely segítségével adatokat küldhet és fogadhat a hálózatban.
- **érzékelők**: egy csomópont egy vagy több érzékelőt tartalmaz, amelyek segítségével különböző fizikai jelenségekkel kapcsolatos paraméterek elektromos jelekké alakíthatók. A mezőgazdaságban leggyakrabban a talajjal kapcsolatos mérések és megfigyelések elvégzésére alkalmazzák. Ezek az érzékelők mérhetik például a talaj nedvességtartalmát, a talaj víztartalmát, talajhőmérséklet, talaj elektromos vezetőképessége, a talaj sótartalmát és az éghajlati paramétereket, beleértve a napfényt, sugárzási szintet, környezeti hőmérsékletet, szélsebességet, csapadékmennyiséget [6].
- **aktuátorok**: az aktuátorok olyan eszközök, amelyek képesek környezeti változásokra reagálni vagy feladatokat végrehajtani a környezetben. Például motorok, szelepek, LED-ek stb. Az aktuátorokat vezérlő jelek alapján aktiválják, amelyeket a csomópontok kapnak más csomópontoktól vagy a hálózat vezérlő elemeitől.
- **energiaforrás**: mivel legtöbb esetben az érzékelő csomópontok olyan helyen vannak, amelyek nehezen elérhetők, ezért az energiaforrás általában elem vagy akkumulátor, amely biztosítja az érzékelő csomópontok működéséhez szükséges

energiaellátást. A WSN-ben az energiahatékonyság kiemelt fontossággal bír, mivel az érzékelő csomópontok általában korlátozott energiaforrásokkal rendelkeznek. Az energiaforrás hatékony kezelése és az energiafogyasztás minimalizálása kulcsfontosságú a hálózat hosszú élettartamának biztosításához [5].



***1. ábra**: Érzékelő csomópont felépítése [5]*

4.2.2. Útválasztó protokollok

A vezeték nélküli érzékelőhálózat nagyszámú érzékelőből áll, amelyeket olyan területeken helyeznek el, ahol vezeték nélküli közegen keresztül kommunikálnak egymással. Ez a fejlett technológia még mindig korlátozott az olyan problémák miatt, mint az útválasztási protokollok, az energiafogyasztás, a biztonság és az adatösszesítés. A hálózaton amikor egy esemény bekövetkezik, a szenzorok begyűjtik az információt, és továbbítják egy másik csomópont vagy a célállomás felé [7].

Az útválasztó protokoll a hálózati kommunikáció során alkalmazott mechanizmus, amely felelős az adatcsomagok útválasztásáért a hálózaton keresztül. Az útválasztó protokoll meghatározza, hogy hogyan választják ki és irányítják az adatcsomagokat a forrás- és a célcsomópont közötti optimális úton. Ehhez figyelembe veszi a hálózati topológiát, a rendelkezésre álló erőforrásokat, a forgalomterhelést és más tényezőket. A cél az, hogy minimalizálja a késleltetést, maximalizálja a sávszélességet, csökkentse a hálózati terhelést és optimalizálja a kommunikáció hatékonyságát.

Számos útválasztási protokollt találtak ki ennek a funkcióknak a biztosítására, és ezek három családba sorolhatók:

1. **Adatközpontú (DC):** Az adatközpontú útválasztás folyamatában a nyelő (vagy fogadó) lekérdezéseket küld bizonyos régiókba, és várja a kiválasztott régiókban található érzékelők adatait. A rendszer az adatok lekérdezésein keresztül szerzi meg az adatokat, ezért az adatok tulajdonságainak meghatározásához attribútumalapú azonosításra van szükség. Az adatközpont (Data Center, DC) fő célja a különböző forrásokból származó adatok kombinálása útvonalak mentén, miközben minimalizálja az adatok továbbításának redundanciáját és az átvitelek számát. Ennek eredményeként a hálózat energiafogyasztását csökkenti és meghosszabbítja annak élettartamát. Az adatközponti útválasztás módszerével hatékonyan lehet irányítani az adatok áramlását és elosztását a hálózatban, biztosítva ezzel az optimális adatgyűjtést és -feldolgozást. Ilyen adatközpontú protokollok a következők: Gossiping, SPIN, COUGAR, GBR, CADR. [5] [8]
2. **Hierarchikus:** A hierarchikus útválasztás egy olyan módszer a vezeték nélküli szenzorhálózatokban, amelynek fő célja az érzékelőcsomópontok energiafogyasztásának optimalizálása. A hierarchikus útválasztás segítségével a hálózatot klaszterekre osztják, ahol minden klaszternek van egy klaszterfeje (cluster head), amely a klaszter tagjainak kommunikációját felügyeli. A klaszterképzés általában az érzékelők energiatartalékára és a klaszterfőhöz való közelségére épül. Az erőforrásokban gazdagabb érzékelők lehetnek klaszterfejek, mivel képesek hosszabb ideig működni. Energiát takarítanak meg azáltal, hogy az alacsonyabb energiafogyasztású érzékelők a klaszterfejek közvetítésével kommunikálnak egymással, ahelyett hogy közvetlenül a távoli adatgyűjtőhöz küldenék az üzeneteket. Ilyen alapon működik a: LEACH, PEGASIS, TEEN. [7]
3. **Helyalapú:** Az ilyen típusú hálózati architektúrában az érzékelőcsomópontokat véletlenszerűen elhelyezik egy adott területen. A helymeghatározásuk gyakran GPS segítségével történik. A csomópontok

közötti távolságot a kapott jelerősség alapján becsülik meg, és a koordinátákat a szomszédos csomópontok közötti információcserével számítják ki. Azonban a helyzetmeghatározás támogatása speciális hardverkomponenseket igényel, és jelentős számítási erőforrást igényel az érzékelőcsomópontokon, ezért ez a megközelítés kihívást jelent erőforrás-korlátozott vezeték nélküli érzékelőhálózatokban. Például: GAF [5]

4.3. Ismert hasonló alkalmazások

4.3.1. Learn OMNeT++ with TicToc

Az OMNeT++ hivatalos oldalán található segédanyag által kivitelezett hálózati protokoll szimulációi hasonlítanak a Pletyka (Gossiping) nevű útválasztó protokoll felépítéséhez. A segédanyag alapjaiban tárgyalja egy hálózat felépítését ebben a fejlesztői környezetben, és több lépésben mutatja be annak létrehozását. Az itt megtalálható kódrészletek, fájlok mindegyike egy újabb funkciót vagy megvalósítást ad hozzá az előző példához. Az első fejezetben (Part 1 – Getting Started) egy projekt létrehozásáról van szó, valamint a hozzá tartozó fájlok (.ned, .cc, .ini tipusú) funkciójáról, és a projekthez való hozzáadásáról. A második részben (Part 2 – Running the Simulation) megtanulhatjuk, hogyan lehet futtatni és debuggolni a szimulációt, milyen alapvető hibákat ejthetünk ami miatt nem indulna el vagy hibásan működne a szimuláció. A következő fejezetekben arról olvashatunk, hogy hogyan tudunk nagyobb hálózatokat kialakítani és a szimuláció eredményét hogyan tudjuk elmenteni. [4]

4.4. Felhasznált technológiák

4.4.1. OMNeT++

Az OMNeT++ lehetővé teszi a felhasználók számára, hogy modelleket hozzanak létre, amelyek reprezentálják a hálózatot vagy rendszert, amelyet szimulálni szeretnének. Ezek a modellek komponensekből (modulokból) állnak, amelyek C++ nyelven vannak programozva. Az OMNeT++ magas szintű nyelvet, a NED-et használja, hogy ezekből a modulokból egy hálózatot tudjon kialakítani [9]. A szimuláció során az OMNeT++ szimulációs kernelét használjuk, amely végrehajtja a modellben definiált viselkedést és interakciókat. A szimuláció eredményeként az OMNeT++ lehetővé teszi a rendszer teljesítményének és más metrikáinak értékelését, valamint statisztikai ábrák létrehozását, szimuláció érédményének elmentését külön fájlba (.sca, .vci,

.vec). A szimuláció eredménye minden esetben releváns, mivel a szimulációban előre tudunk ugorni (fel tudjuk gyorsítani), így pillanatok alatt több száz vagy akár több ezer adatot is kinyerhetünk.

NED (Network Description Language) modulokat egyszerűen lehet létrehozni, van egy erre kitalált editor, amiből bármilyen modult ki tudunk választani. Persze ezeknek a moduloknak, hogy hasznát is vegyük meg kell írni a kódját, úgy tudunk nekik funkciót adni. Ezt a kódot egy C++ fájlban kell megírnunk, és fordítás után egy szimuláció segítségével tudjuk szemléltetni a hálózatunkat. A modulok egyszerűen is kommunikálhatnak egymással például, ha egy üzenetet küldünk egyik modulról a másiknak, de sokkal bonyolultabb funkciókat is tudunk írni nekik és nagyobb hálózatokat is ki tudunk építeni. A modulokat egymásba is lehet rakni, ilyenkor a program automatikusan létrehozza a külső modul kódjába az almoduljai class-ét.

A NED fájlon belül hozzuk létre magát a hálózatot is a „network” kulcsszóval, ezen belül hozhatunk létre paramétereket, mint például hogy mennyi legyen a csomópontok maximális száma, deklaráljuk a használandó modulokat, és létrehozhatjuk a kapcsolatokat is a csomópontok között.

Az OMNeT++ keretrendszerben a .msg fájlok az üzenetek formátumának leírására szolgálnak. Ezek a fájlok meghatározzák, hogy milyen adatokat és mezőket tartalmazhat egy adott üzenet. Egy .msg fájlban definiálhatunk üzeneteket, amelyeket a szimuláció során a komponensek között továbbítunk. Az üzenetek struktúráját mezők határozzák meg, amelyek tartalmazhatnak különböző adattípusokat, például egész számokat, valós számokat, stringeket stb. Ez lehetővé teszi a szimuláció során az adatok pontos és strukturált továbbítását a komponensek között.

A szimulációhoz még szükségünk van a .cc fájlokra is, amelyek a komponensek viselkedésének implementációját tartalmazzák. Ezek a fájlok C++ nyelven íródnak, és lehetővé teszik a komponensek működésének testre szabását és a szimulációhoz szükséges logika implementálását. A .cc fájlokban definiáljuk a komponens osztályokat, amelyeket a szimuláció során példányosítunk és használunk. Ezek az osztályok kiterjesztik az OMNeT++ keretrendszerben előre definiált osztályokat vagy interfészeket, például a cModule vagy a cSimpleModule osztályokat. Ezekben a fájlokban definiáljuk a komponens viselkedését, mint például az inicializációs lépéseket, az üzenetek fogadását és feldolgozását, az időzítők kezelését, a hálózati események reagálását stb. Ezenkívül itt hozzuk létre a kommunikációt más

komponensekkel, küldünk és fogadunk üzeneteket, valamint végrehajtjuk a szükséges számításokat és műveleteket a szimuláció során.

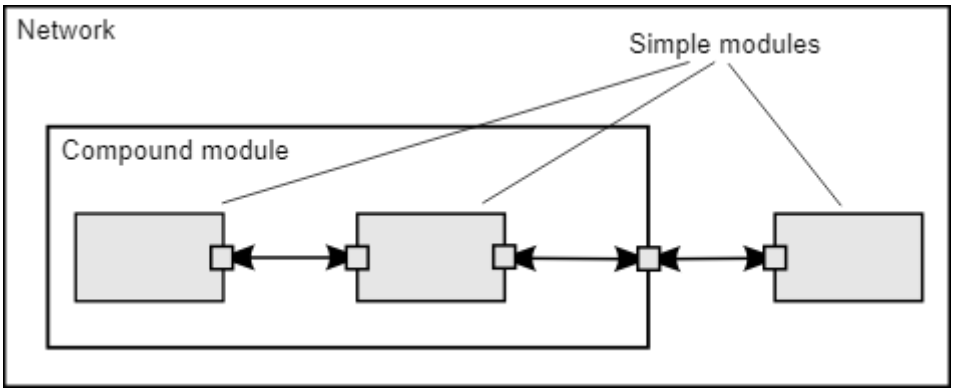
A szimuláció eredményét két féle fájlban kapjuk meg, mint a .sca (Scalar Output), és a .vec (Vector Output) típusú fájlok, amelyek egy automatikusan létrehozott ‘results’ mappába kerülnek a szimuláció lefutása után. Az .sca fájlokban skaláris adatok, például időbélyegek vagy statisztikák, míg a .vec fájlokban vektoradatok, például üzenetek vagy csomagok nyomon követhető adatai találhatóak. Ezeket az eredményfájlokat az OMNeT++ beépített eredménykezelője generálja a szimuláció során. [3]

5. A rendszer specifikációi és architektúrája

5.1. Modellek

Az OMNeT++ modell olyan modulokból áll, amelyek üzenettovábbítással kommunikálnak egymással. Az aktív modulokat egyszerű moduloknak nevezzük, C++ nyelven íródnak, a szimulációs osztálykönyvtár segítségével. Az egyszerű modulok összetett modulokká és így tovább csoportosíthatók, a hierarchiaszintek száma korlátlan. Az OMNeT++-ban hálózatnak nevezett teljes modell maga is egy összetett modul. Az üzeneteket vagy modulokon átívelő kapcsolatokon keresztül, vagy közvetlenül más modulokhoz lehet küldeni.

Az alábbi ábrán a dobozok az egyszerű modulokat és az összetett modulokat jelölik. A kis dobozokat összekötő nyilak kapcsolatokat és kapukat jelentenek.



***2. ábra**: Egyszerű és összetett modulok [3]*

Példa egyszerű modul létrehozására:

```
simple modelName
{
    parameters:
        int parameter1;
        @display("i=block/queue");
    gates:
        input in;
        output out;
}
```

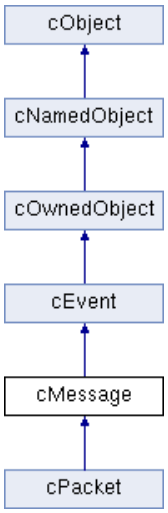
A modulok olyan üzenetekkel kommunikálnak, amelyek a szokásos attribútumok, például az időbélyegző mellett tetszőleges adatokat is tartalmazhatnak. Az egyszerű modulok jellemzően kapukon keresztül küldik az üzeneteket, de lehetőség van arra is, hogy közvetlenül a célmoduloknak küldjék azokat. A kapuk a modulok bemeneti és kimeneti interfészei: az üzeneteket a kimeneti kapukon keresztül küldik, és a bemeneti kapukon keresztül érkeznek. Egy bemeneti és egy kimeneti kapu összekapcsolható egy kapcsolattal. A modell hierarchikus felépítése miatt az üzenetek jellemzően kapcsolatok láncolatán keresztül haladnak, egyszerű modulokból indulva és oda érkezve. A kapcsolatokhoz olyan paraméterek rendelhetők, mint a terjedési késleltetés, az adatátviteli sebesség és a bithibaarány. Speciális tulajdonságokkal rendelkező (csatornáknak nevezett) kapcsolattípusokat is definiálhatunk, és ezeket több helyen újra felhasználhatjuk. A modulok rendelkezhetnek paraméterekkel. A paraméterek elsősorban arra szolgálnak, hogy konfigurációs adatokat adjanak át egyszerű moduloknak, és hogy segítsenek a modelltopológia meghatározásában. A paraméterek string, numerikus (int, double, stb) vagy bool értékeket vehetnek fel, amelyeket a NED fájlokban vagy az omnetpp.ini konfigurációs fájlban lehet hozzárendelni.

5.2. Üzenetek

A modulok üzenetek cseréjével kommunikálnak, amelyek tetszőlegesen összetett adatstruktúrákat tartalmazhatnak. Az egyszerű modulok az üzeneteket vagy közvetlenül a célállomásukra, vagy egy előre meghatározott útvonalon, kapukon és kapcsolatokon keresztül küldhetik el. Az üzenetek a cMessage osztályba tartoznak, egy új üzenetet a következő képpen tudunk létrehozni:

```
cMessage *msg = new cMessage;
```

Öröklődési diagram a cMessage osztályhoz:



***3. ábra**: cMessage öröklődés [13]*

A cMessage egy beépített osztály, de saját üzenet típust is tudunk létrehozni, erre szolgálnak a .msg típusú fájlok. Ezekben a fájlokban tetszés szerint határozhatunk meg új adatokat amelyeket majd tartalmazni fog ez a típusú üzenet. A következő üzenet típus három adattagot tartalmaz: source, destination és hopCount

```
message newMsgType {
    int source;
    int destination;
    int hopCount = 0;
}
```

Ilyen típusú üzenetet a következő képpen tudunk definiálni, és beállítani az adattagjait:

```
int src = source;
int dest = destination;

newMsgType *msg = new newMsgType();
msg->setSource(src);
msg->setDestination(dest);
```

Az üzenet típus létrehozásakor a paramétereihez tartozó set és get metódusok automatikusan generálódnak.

Az üzenet elküldését a ’send()’ paranccsal tudjuk megtenni, amelynek a következő paramétereket kell beállítani: magát az üzenetet, és a kapu azonosítóját, amelyen keresztül szeretnénk küldeni az üzenetet. Pl: `send(msg, gateID);`

5.3. C++ fájlok felépítése OMNeT++-ban

A diszkrét eseményszimuláció a jövőbeli események halmazát egy adatszerkezetben tartja nyilván, amelyet gyakran FES-nek (Future Event Set) vagy FEL-nek (Future Event List) neveznek. Az ilyen szimulátorok általában a következő pszeudokód szerint működnek [3]:

```
initialize -- ez magában foglalja a modell felépítését és
              a kezdeti események beillesztése a FES-be
while (FES not empty and simulation not yet complete)
{
    retrieve first event from FES
    t:= timestamp of this event
    process event
    (a feldolgozás új eseményeket illeszthet be a FES-be vagy törölheti a
meglévőket)
}
finish simulation (write statistical results, etc.)
```

Az inicializálási lépés általában felépíti a szimulációs modellt reprezentáló adatstruktúrákat, meghívja a felhasználó által definiált inicializálási kódot, és beilleszti a kezdeti eseményeket a FES-be, hogy a szimuláció elindulhasson.

A szimuláció akkor áll le, amikor már nincs több esemény (ez a gyakorlatban ritkán fordul elő), vagy amikor a szimuláció további futtatására nincs szükség, mert a modellidő vagy a CPU-idő elért egy adott határt, vagy mert a statisztikák elérték a kívánt pontosságot. Ilyenkor, a programból való kilépés előtt a felhasználó jellemzően a statisztikákat a kimeneti fájlokban kívánja rögzíteni.

Az OMNeT++-ban a .cc fájl a modulokhoz tartozó C++ osztályok implementációját tartalmazza. Ezek az osztályok kiterjesztik a szimulációs keretrendszer által biztosított alapvető osztályokat, például a cSimpleModule vagy a cMessage osztályokat. A .cc fájlban definiált osztályok több előre meghatározott metódust is tartalmazhatnak, amelyeket a szimulációs környezet hív meg a megfelelő időpontokban:

1. **’initialize()**’: Az initialize() azért létezik, mert általában nem lehet szimulációval kapcsolatos kódot tenni az egyszerű modul konstruktorába, mert a szimulációs modell még mindig a beállítás alatt van, amikor a konstruktor fut, és sok szükséges objektum még nem áll rendelkezésre. Ez a metódus azután hívódik

meg, hogy az OMNeT++ létrehozta a hálózatot (azaz létrehozta a modulokat és a definícióknak megfelelően összekapcsolta őket). Itt inicializálódnak a modulok, beállíthatóak a kezdeti állapotaik, konfigurációik, valamint ütemezni lehet az eseményeket és üzeneteket.

2. **'handleMessage(cMessage *msg)**': Ez a metódus akkor hívódik meg, amikor a modul egy üzenetet fogad. Itt kezeljük az érkező üzeneteket, feldolgozhatjuk az adatokat és válaszokat generálhatunk. Eben a metódusban implementálható a modulok által kívánt logika az üzenetek kezelésére. Ez lehet például az üzenet tartalmának feldolgozása, a válaszüzenetek generálása, további üzenetek küldése más moduloknak, vagy a modul állapotának módosítása az üzenet alapján.
3. '**finish()**': Ez a metódus a szimuláció végén fut le. Itt végezheted el a modul utómunkáit, például adatokat menthetünk le vagy statisztikákat generálhatunk.

Tehát az egyszerű modul nem más, mint egy C++ osztály, amelyet a cSimpleModule alosztályba kell sorolni, és egy vagy több virtuális tagfüggvényt újra kell definiálni a viselkedés meghatározásához. Az osztályt az OMNeT++-ban a Define_Module() makróval kell regisztrálni. A Define_Module() sort mindig .cc vagy .cpp fájlba kell tenni, nem pedig fejlécfájlba (.h), mert a fordító ebből generál kódot.

Példa egy modul osztály implementálására:

- NED modul létrehozása:

```
simple HelloModule
{
    gates:
        inout gate[];
}
```

- C++ osztály implementálása:

```
#include <omnetpp.h>
using namespace omnetpp;

class HelloModule : public cSimpleModule
{
  protected:
    virtual void initialize() override;
    virtual void handleMessage(cMessage *msg) override;
};
```

```
Define_Module(HelloModule);

void HelloModule::initialize()
{
    EV << "Hello World!" << endl;
}

void HelloModule::handleMessage(cMessage *msg)
{
    delete msg;
}
```

6. Részletes tervezés

A tervezés első fázisaként bele kellett ásnom magam a vezeték nélküli hálózatok működésébe, és mélyebb tanulmányozást kellett csinálnom a különböző útválasztó protokollokról. Mivel egy teljesen idegen környezetben kellett megvalósítanom a szimulációkat, ezért kezdetben egyszerűbb hálózatok kiépítésével próbálkoztam a programban, és próbáltam alapjaiban megérteni, hogyan kell felépíteni egy ilyen projektet, hogyan tudnak kommunikálni egymással a modulok és hogyan lenne lehetséges megvalósítani az általam választott protokollokat. A legnehezebb része ez volt, mivel nagyon kevés oktató- és segédanyagot találtam, amely OMNeT++-ban való programozásra tanított volna, ezért sok dologra magamtól kellett rájönnöm, és részletesen kellett böngésznem az OMNeT++ kézikönyvét [3], amiben minden fellelhető modul le van írva és annak metódusai.

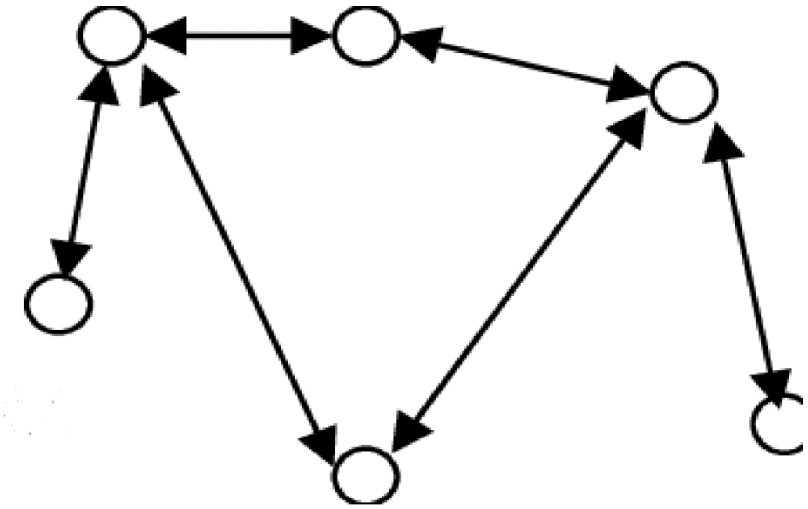
Miután kezdtem megérteni a program struktúráját, elkezdhettem megtervezni az első szimulálni kívánt protokoll felépítését, amely a Gossiping volt, mivel relatív ez a könnyebb a három protokoll közül amiket még szimulálni akartam. A LEACH és a GAF protokollok egészen más logikát követtek a Gossipinghez képest, ezért ezekhez nem tudtam felhasználni az előzőleg megírt protokoll kódját. Viszont miután jobban kezdtem feltárni az OMNeT++ által nyújtott lehetőségeket, rengeteg ötletem támadt, hogy miként lehetne kivitelezni a maradék két protokollt is, nyilván nem sikerül elsőre, de rengeteg debuggolás után azok is készen lettek.

6.1. Útválasztó protokollok szimulációja

6.1.1. Pletyka (Gossiping)

6.1.1.1 Elméleti tudnivalók

A Gossiping egy adatközpontú útválasztó protokoll, amely az Áradás (Flooding) továbbfejlesztett változata. Az Áradás működési elve szerint az érzékelő csomópont minden beérkező csomagot tovább sugároz az összes csomópontnak, amíg az el nem jut a célállomásig. Ezzel szemben a Gossiping úgy működik, hogy az aktuális csomópont a kapott csomagot egy véletlenszerűen kiválasztott szomszédos csomópontnak küldi, így előre kiszámíthatatlan, hogy a csomag mennyi idő alatt ér el a célállomásig, vagy hány csomóponton kell keresztül mennie [10] [5].



***4. ábra**: Gossiping protokoll [11]*

6.1.1.2 NED modul létrehozása

A NED fájlban két részt különböztetünk meg: a "simple GossipNode" és a "network Gossip" részeket. A "simple GossipNode" részben definiáljuk a GossipNode modelljét. Ez egy egyszerű modul, amely tartalmazza a paramétereket és a kapukat. A paraméterek között van egy "arrival" jel, és egy "hopCount" statisztika, amely arra szolgál, hogy tudjuk rögzíteni, hogy egy csomag hány csomóponton kellett keresztül mennie, mire célba ért. A modul megjelenítése a

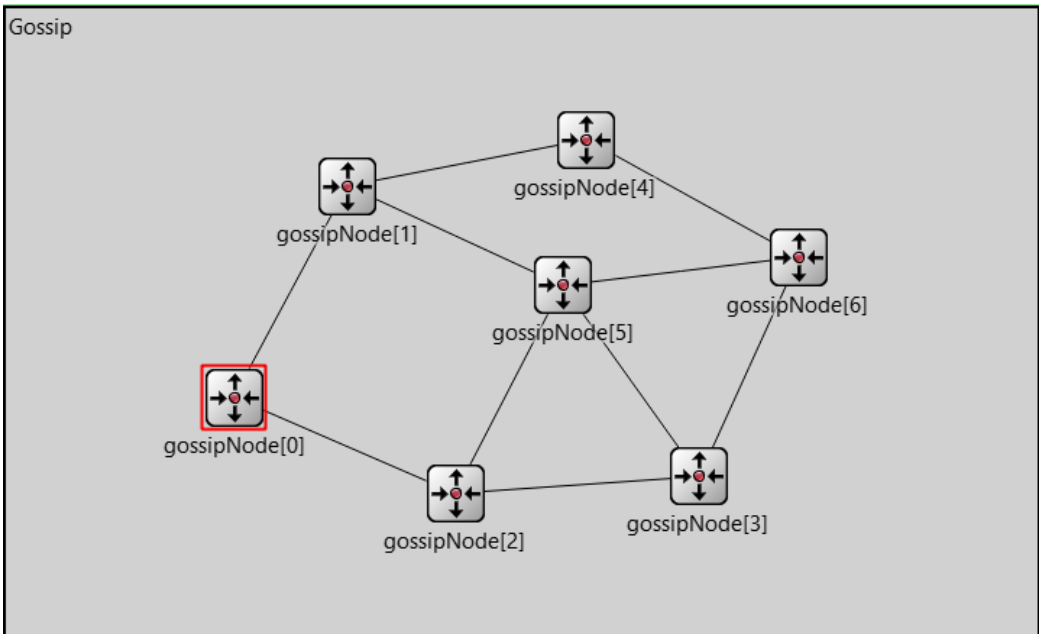
"display" utasítással van beállítva, ahol a "block/routing" érték a blokkdiagram típusú megjelenítést jelzi.
A modul létrehozása:

```
simple GossipNode
{
    parameters:
        @signal[arrival](type="long");
        @statistic[hopCount](title="hop count"; source="arrival";
record=vector,stats; interpolationmode=none);

        @display("i=block/routing");
    gates:
        inout gate[];
}
```

A "network Gossip" részben definiáljuk a Gossip hálózatot. Itt a GossipNode-ot definiáljuk almodulként, 7 példányban, ez azt jelenti, hogy 7 csomópontunk lesz a hálózatban. A modulok közötti kapcsolatokat a "connections" részben definiáljuk. Az itt felsorolt kapcsolatok meghatározzák, hogy melyik GossipNode kapui vannak összekötve egymással. Például a "gossipNode[0].gate++ <--> gossipNode[1].gate++" jelenti, hogy az első GossipNode modul egyik kapuja össze van kötve a második GossipNode modul egyik kapujával. Mivel a kapu típusa ’inout’, ezért nem kell külön meghatározni, hogy a kapcsolatban a kapu ki- vagy bemenetelét kötjük össze.

```
network Gossip
{
    @display("bgb=548,332");
    submodules:
        gossipNode[7]: GossipNode;

    connections:
        gossipNode[0].gate++ <--> gossipNode[1].gate++;
        gossipNode[0].gate++ <--> gossipNode[2].gate++;
        gossipNode[2].gate++ <--> gossipNode[3].gate++;
        gossipNode[2].gate++ <--> gossipNode[5].gate++;
        gossipNode[1].gate++ <--> gossipNode[4].gate++;
        gossipNode[1].gate++ <--> gossipNode[5].gate++;
        gossipNode[3].gate++ <--> gossipNode[5].gate++;
        gossipNode[3].gate++ <--> gossipNode[6].gate++;
        gossipNode[4].gate++ <--> gossipNode[6].gate++;
        gossipNode[5].gate++ <--> gossipNode[6].gate++;
}
```

Ez a gyakorlatban a következő hálózatot reprezentálja:



5. ábra: *Gossip hálózat*

6.1.1.3 Új üzenet típus meghatározása

Ha nem hozunk létre új üzenet típust, akkor alapértelmezetten ’cMessage’ típusú üzenetet fogunk küldeni, viszont szükség van arra, hogy tudjuk egy üzenetről, hogy melyik csomópont küldi, mi a célállomás, e mellett tudjuk számon tartani, hogy hány csomóponton haladt keresztül, mire célba ért. Ezek megvalósítására létrehoztam egy új ’.msg’ típusú fájlt Gosspi.msg névvel, amiben implementáltam a fent említett változókat:

```
message GossipMsg {
    int source;
    int destination;
    int hopCount = 0;
}
```

Ebben a kódban ezt az üzenet típust használtam, így könnyebben számon lehet tartani egy üzenet útvonalát és egyéb paramétereit.

6.1.1.4 A hálózati logika implementálása

A hálózati logika és a modulokhoz tartozó C++ osztály implementálása a .cc fájlban történik. Először is létre kell hoznunk az osztályt, és definiáljuk a tagfüggvényeket:

```
class GossipNode : public cSimpleModule
{
  private:
    simsignal_t arrivalSignal;
  protected:
    virtual GossipMsg *generateMessage();
    virtual void forwardMessage(GossipMsg *msg);
    virtual void initialize() override;
    virtual void handleMessage(cMessage *msg) override;
};

Define_Module(GossipNode);
```

A ’virtual’ kulcsszó használata lehetővé teszi, hogy a származtatott osztályok felülírják ezeket az előre definiált függvényeket a saját specifikus implementációjukkal. Ez nagyon hasznos az OMNeT++ modellezésében, mert lehetővé teszi a modulok viselkedésének testre szabását a konkrét igényeknek megfelelően.

A "signal" változó a szimulációban használt jelzés típusának regisztrálására szolgál. A jelzés egy adott esemény bekövetkeztét jelzi, és a modulok közötti kommunikáció, illetve információátvitel eszköze lehet. Ez a későbbiekben arra fog szolgálni, hogy tudjuk mikor kell rögzíteni a ’hopCount’ statisztikát.

Az ’`initialize()`’ függvényben annyi dolgunk van, hogy inicializáljuk a ’signal’ változónkkat a ’`registerSignal()`’ segítségével. Ez után meg kell vizsgálnunk, hogy a jelenlegi csomópontunk a 0-ás indexű-e, mert azt szeretnénk, hogy az üzenetküldést ez a csomópont kezdje. A ’`getIndex()`’ hívás visszatéríti a jelenlegi csomópont indexét, így ha a feltételünk igaz, akkor biztosra tudjuk, hogy az első csomópontnál vagyunk. Ekkor generálunk egy új üzenetet a ’`generateMessage()`’ függvénnyel és ütemezzük.

Az ’`generateMessage()`’ függvény a GossipNode osztályban egy új GossipMsg típusú üzenet létrehozásáért felelős. Az aktuális modul azonosítóját (src) lekéri a ’`getIndex()`’ függvénnyel. Ez azért fontos, mert az üzenet forrása a jelenlegi modul lesz. A cél modul azonosítóját (dest) véletlenszerűen választjuk ki a ’intuniform()’ függvénnyel. A választás során kizárjuk a saját modult, ezért az értéknek src-nél kisebbnek kell lennie, ha src a legnagyobb modul azonosító a hálózatban, majd beállítjuk az üzenet forrását (setSource(src)) és célját (setDestination(dest)) a korábban meghatározott értékek alapján és visszatérítjük az üzenetet.

A ’`handleMessage()`’ függvény kezeli az üzenetek fogadását és tovább küldését. Mivel ez a függvény alapvetően ’cMessage’ típusú üzenetet vár, ezért az elején a ’GossipMsg’ típusú üzenetünket castolnunk kell, hogy tudjunk tovább dolgozni vele. Ezt követően csak

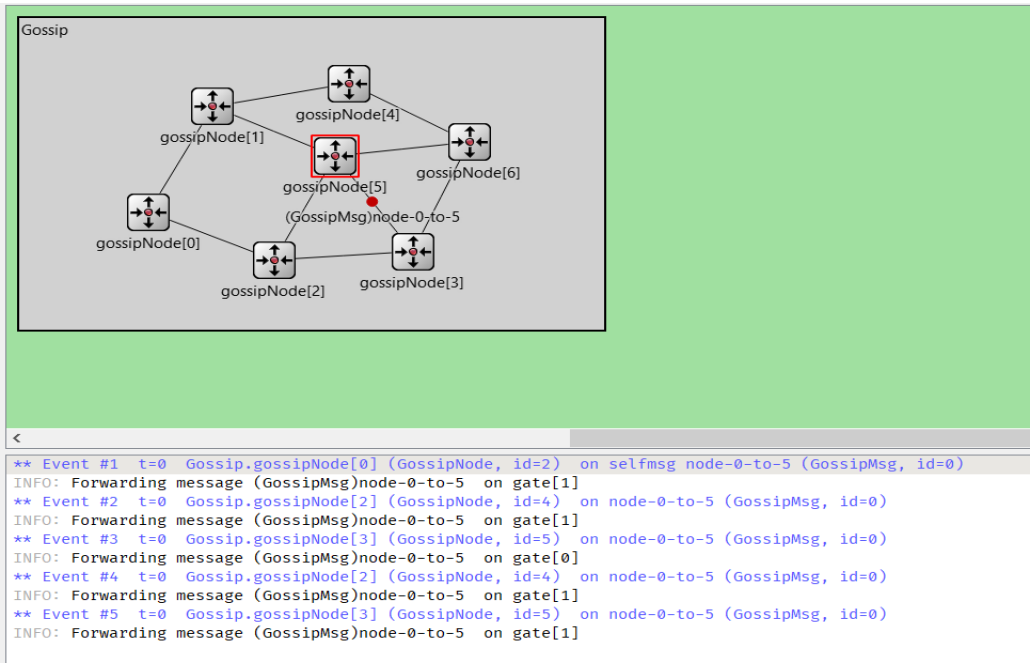
ellenőriznünk kell, hogy a jelenlegi csomópont indexe egyezik-e az üzenet célállomásával, ha igen akkor az azt jelenti, hogy célba ért az üzenet, ekkor annyi a dolgunk, hogy rögzítjük az üzenet ugrás számlálóját (hopCount) a signal segítségével, majd új üzenetet generálunk, amelynek a kiinduló csomópontja a jelenlegi csomópont lesz, célállómása pedig random választódik ki. Viszont, ha a feltétel hamis, akkor csak egyszerűen továbbítjuk az üzenetet egy következő csomópontnak.

Az üzenet továbbítását a '`forwardMessage()`' függvény intézi, ha ez meghívódott az azt jelenti, hogy az üzenet átkerült egy másik csomóponthoz, tehát az üzenet ugrás számát növelnünk kell, és ki kell választanunk egy véletlenszerű kaput amin keresztül a 'send()' metódussal tovább küldjük az üzenetet.

6.1.1.5 Összegzés

Összegzésképpen tehát a szimuláció úgy működik, hogy az első csomópont generál egy új üzenetet, amely tartalmazza a küldő csomópont azonosítóját és a célállomás azonosítását, valamint számolja, hogy az üzenet hány csomóponton haladt át mielőtt célba ért. Az üzenet továbbító függvény véletlenszerűen választja ki, hogy melyik szomszédos csomópontjának fogja küldeni az üzenetet.

A hálózat működés közben:



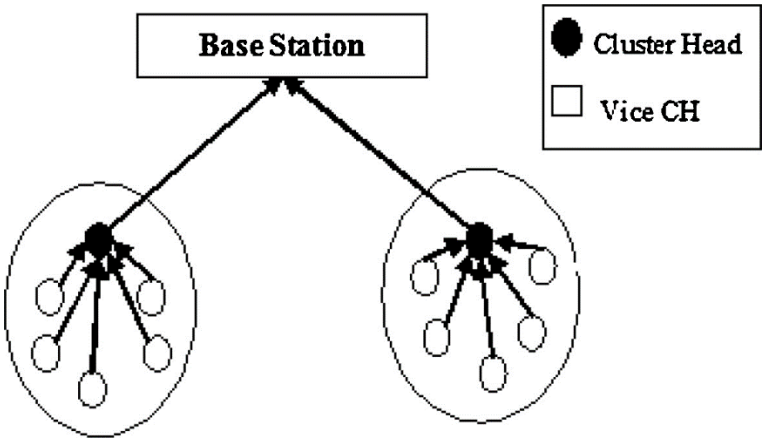
***6. ábra**: A hálózat működése*

Az ***6. ábra*** prezentálja az üzenet haladását a hálózaton egy adott pillanatban, látható, hogy az üzenet éppen a hármas és ötös csomópontok között halad az ötös felé. Az üzenet neve ’node-0-to-5’ azt jelenti, hogy a kezdő csomópontja a nullás csomópont, a célállomása pedig az ötös. A hálózat alatt láthatóak az információs üzenetek, amelyek nyomon követik a üzenet útját, és kiírják a képernyőre, hogy melyik csomóponton és kapunk haladt át, jelzik célba érést és az új üzenet generálását.

6.1.2. LEACH (Low Energy Adaptive Clustering Hierarchy)

6.1.2.1 Elméleti tudnivalók

Egy LEACH hálózat felépítése több lépésből áll: van egy kezdeti, inicializálási állapot, amikor a hálózat csomópontjai klaszterekbe szerveződnek, és kiválasztanak egy klaszter fejet. A klaszter azon csomópontok összeségét jelenti, amelyek azonos klaszter fejnek küldenek adatokat. Miután megtörtén a klaszterezés, a csomópontok adatokat kezdenek küldeni a klaszter fejnek, amely összegyűjti ezeket, és továbbítja a bázisállomásnak. Ez az érzékelőcsomópontok energiájának megtakarítását eredményezi, mivel a bázisállomás helyett kevesebb energiát kell fordítaniuk arra, hogy az adataikat a klaszterfejhez küldjék. Ezen túlmenően a klaszterfejek összesítik az összegyűjtött adatokat, hogy eltávolítsák a hasonló adatok közötti redundanciát, és ezáltal csökkentsék a bázisállomásra továbbított adatokat. Ez nagy mennyiségű energia megtakarítását eredményezi, mivel az összesített adatokat egyetlen ugrással továbbítják. [2] [1].



7. ábra: *LEACH hálózat szemléltetése [1]*

6.1.2.2 NED modul megvalósítása

A Gossip-hoz hasonlóan itt is egy egyszerű modult hoztam létre (LEACHNode), viszont ennek a modulnak később más-más szerepe lesz, attól függően, hogy klaszter fej, bázisállomás vagy egyszerű csomópont.

A hálózat létrehozásához paramétert használok, amelynek szerepe, hogy inicializáláskor meg tudjuk határozni, hát csomópontot szeretnénk a hálózatban, ezt az értéket a 'numNodes' változóban tároljuk.

```
parameters:
    int numNodes;
```

A '`submodules`' részben meghatározzuk a modulokat, ahogy fent említettem, ebből háromfajta lesz:

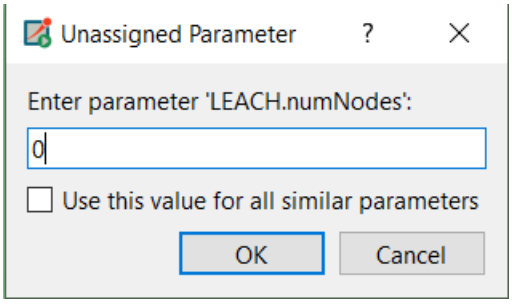
```
submodules:
        node[numNodes]: LEACHNode {

        }
        ClusterHead: LEACHNode {
            @display("p=352,227");
        }
        BaseStation: LEACHNode {
            @display("p=146,203");
        }
```

Mind a három a 'LEACHNode' modulból származik, ezért mindegyik rendelkezni fog az ebben a modulban meghatározott összes tulajdonsággal. A '`@display`' tulajdonsággal azt állíthatjuk be, hogy a csomópont hol helyezkedjen el a hálózatban.

A kapcsolatok csomópontok között úgy kell kialakítani, hogy a klaszter fej egy központi csomópontként viselkedjen, tehát mindennel kapcsolatban kell álljon. Ezek alapján kapcsolatot kell kialakítani a bázisállomás és a klaszterfej között, valamint a klaszter fej és az összes többi csomópont között, amelyeket levélcsomópontnak nevezünk. Így az üzenetek a következő útvonalon tudnak közlekedni: küldő csomópont `->` klaszter fej `->` bázisállomás. Ehhez szükséges egy ismétlődő ciklus, amely végig megy a csomópontokon és létrehozza a kapcsolatot a klaszter fejjel:

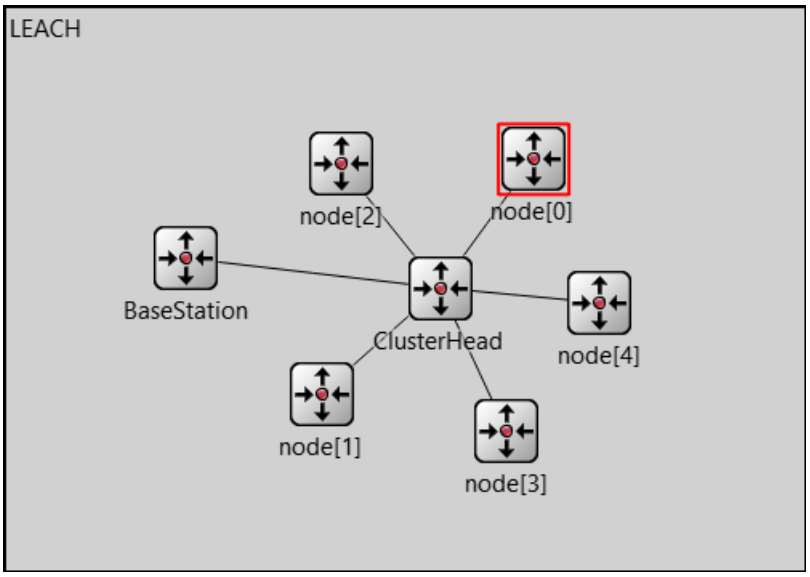
```
connections:
        ClusterHead.gate++ <--> BaseStation.gate++;
        for i=0..numNodes-1 {
            node[i].gate++ <--> ClusterHead.gate++;
        }
```

Mivel a hálózatban nem határozzuk meg előre a ’numNodes’ változó értékét, ezért a szimuláció elindításakor, még inicializálás előtt felugrik egy ablak, amiben be tudjuk állítani ezt az értéket, így egyszerűen különböző számú csomópontokkal rendelkező hálózatokat is tudunk készíteni:



***8. ábra**: Paraméter beállítása*

Miután beállítottuk a csomópontok számát, ettől függően a hálózat a következőképpen fog kinézni:



***9. ábra:** LEACH hálózat 5 levélcsomóponttal*

6.1.2.3 Hálózati logika implementálása

Hogy a megfelelő protokoll szerint működjön a hálózatunk, azt kell elérnünk, hogy kezdetben az egyszerű érzékelőcsomópontok generáljanak üzeneteket, azokat küldjék a klaszter fej felé, majd a klaszter fej továbbítja ezeket a bázisállomás felé, amely fogadja az üzeneteket.

A C++ osztály hasonló a Gossip osztályhoz, viszont itt bővült egy újabb tagfüggvénnyel amit fölülírunk, és ez a 'finish()', a szimuláció végén itt fogjuk kiiratni a begyűjtött statisztikákat:

```
class LEACHNode : public cSimpleModule
{
  private:
    int numNodes;
    simsignal_t arrivalSignal;
    long numSent;
    long numReceived;
    cHistogram hopCountStats;
    cOutVector hopCountVector;
  protected:
    virtual void initialize() override;
    virtual void handleMessage(cMessage *cmsg) override;
    virtual LEACHmsg *generateMessage();
    virtual bool forwardMessage(LEACHmsg *msg);
    virtual void finish() override;
};

Define_Module(LEACHNode);
```

Valamint számos új paraméter jelent meg, amelyek újabb statisztikák rögzítésére szolgálnak, mint például: 'numSent' – a csomópontok által küldött üzenetek számát rögzíti, 'numRecieved' – a fogadott üzenetek száma. Ez a két érték különbözhet egymástól, mivel ebben a hálózatban jelen van a lehetőség egy üzenet elvesztésére, ez azt jelenti, hogy a csomópont elküldte az üzenetet, viszont az nem érkezett meg a fogadó csomóponthoz, ebben az esetben újra kell küldeni az üzenetet, ennek biztosítására küldés előtt mindig csinálunk egy másolatot az üzenetről, hogy szükséges akárhányszor tudjuk majd újra küldeni. Az üzenet elvesztését úgy oldottam meg, hogy tovább küldés előtt generálok egy véletlen számot 0 és 1 között (egy tizedes pontossággal), és ha ez a szám kisebb mint 0.1, tehát ha 0.0, akkor töröljük az üzenetet, értelemszerűen erre 10% esély van. A 'numSent' paraméter mindig fog növekedni, ha egy üzenetet megpróbálunk elküldeni, beleértve az újraküldést is ha az üzenet elvesztődött, viszont a 'numRecieved' paraméter csak akkor növekszik, ha az üzenet sikeresen továbbítódott a célcsomópontnak.

```
numSent++;
```

```
while (!forwardMessage(msg))
{
    numSent++;
    EV << "Resending message.." << endl;
    msg = copyMsg;
}
 numReceived++;
```

A legfontosabb függvény, amely az üzenetek irányítását végzi, az a ’`handleMessage(cMessage *cmsg)`’. Ebben az esetben egy másik módszerrel tudjuk lekérni, hogy éppen melyik csomópontnál járunk, a Gossip-hoz képes, itt az ’`if (strcmp(getName(), "ClusterHead") == 0)`’ teljesülésekor tudjuk biztos, hogy (ebben az esetben) a jelenlegi csomópont a klaszter fej, a bázisállómás ellenőrzése is hasonlóképpen zajlik, csak a hasonlítandó szöveg részt kell kicserélnünk `"BaseStation"`-re. A levélcsomópontok azonosítására használhatjuk a ’`getIndex()`’ metódust, mivel ezek a NED fájlban tömbként vannak deklarálva, ezért van indexük, az előző csomópontokra pont ezért nem lehetett használni ezt a metódust, mert nem tömbök. Ha már tudjuk, hogy melyik típusú csomópontnál vagyunk, nincs más dolgunk, csak tovább küldeni az üzenetet a megfelelő kapunk.

Az üzenetek felépítéséhez is szükségünk van a jelenlegi csomópont azonosítására, mivel az üzenet ’source’ paraméterét be kell állítanunk a küldő csomópont nevére, ez lehet levélcsomópont vagy klaszter fej, a ’destination’ paraméterét pedig a célállomás nevére, ez pedig vagy a klaszter fej lesz, vagy a bázisállómas. A ’`LEACHmsg`’ típusú üzenetek hasonlóképpen vannak deklarálva, mint a Gossip hálózathoz tartozó üzenettípus:

```
message LEACHmsg
{
    string source;
    string destination;
    int hopCount = 0;
}
```

Az adattagok megfelelő beállítása a következőképpen történik:

```
std::string src, dest;
if (strcmp(getName(), "ClusterHead") == 0)
{
    src = "ClusterHead";
    dest = "BaseStation";
}
else {
```

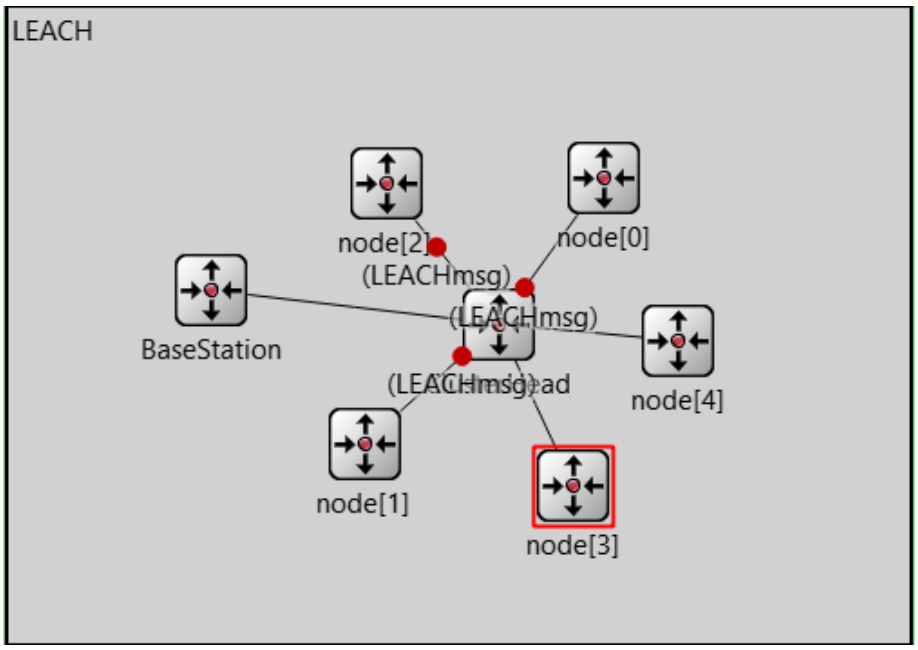
```
        if (strcmp(getName(), "BaseStation") == 0)
        {
            src = "BaseStation";
            dest = "null";
        }
        else {
            src = "Node " + std::to_string(getIndex());
            dest = "ClusterHead";
        }
    }

    LEACHmsg *msg = new LEACHmsg();
    msg->setSource(src);
    msg->setDestination(dest);
```

A statisztika megjelenítése szempontjából fontos függvény a ’`finish()`’, ami akkor hívódik meg, ha már nincs több esemény amit szimulálni kellene, vagy lejárt a szimulációra szánt idő, de ezt külön be kell állítani a .ini fájlban. A következő statisztikai adatokat jelenítem meg a szimuláció végén, ezek minden csomópontra külön érvényesek: elküldött üzenetek száma, fogadott üzenetek száma, az ugrás szám minimuma, maximuma, átlaga és szórása, valamint itt rögzítjük ezeket az adatok a kimeneti fájlokba.

6.1.2.4 Összegzés

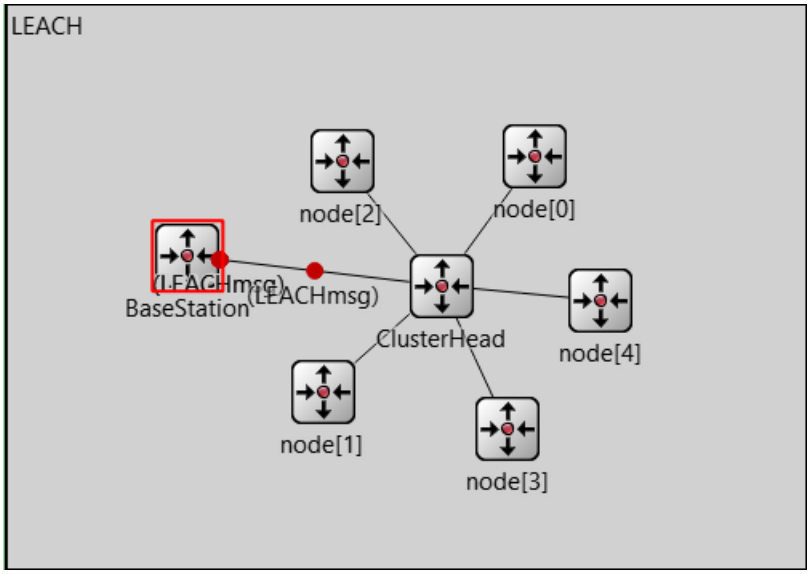
Összességében a szimuláció prezentálja egy LEACH típusú útválasztó protokoll működését egy klaszter fejjel, láthatjuk ahogyan a levélcsomópontok folyamatosan küldik az üzenetet a klaszter fej felé, amely miután összegyűjtötte ezeket, tovább küldi a bázisállomás felé.

Levélcsomópontok üzeneteinek összegyűjtése: itt láthatjuk, ahogyan a klaszter fej begyűjti az üzeneteket, a ’node0’ és ’node1’ már elküldte az üzenetet, a ’node2’ épp most küldi, a többi csomópont még hátra van:



10. ábra: *Üzenetek összegyűjtése*

A szimuláció következő fázisában pedig már a klaszter fej az utolsó üzenetet továbbítja a bázisállomás felé:



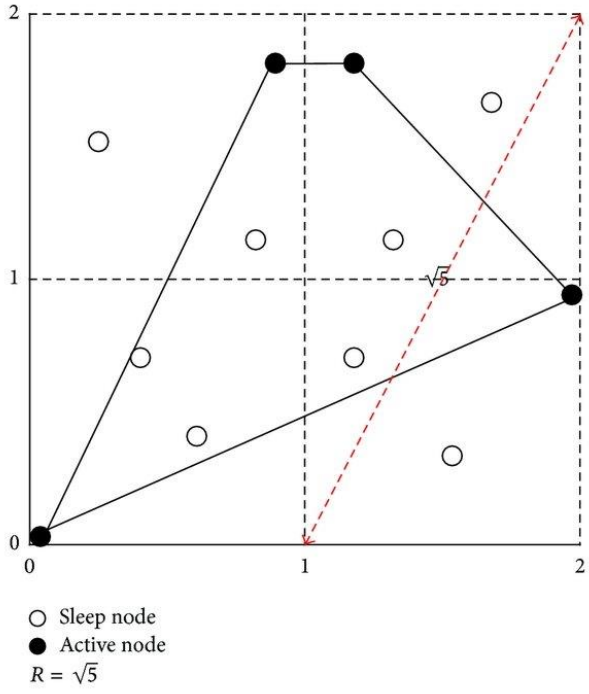
11. ***ábra:*** *Klaszter fej továbbítja az üzeneteket*

6.1.3. GAF (Geographic Adaptive Fidelity)

6.1.3.1 Elméleti tudnivalók

A Geographic Adaptive Fidelity (GAF) protokoll az útválasztási döntéseket a szenzorcsomópontok földrajzi helyzetének alapján hozza meg. Az útválasztás során a protokoll figyelembe veszi a csomópontok közötti távolságot, a jelminőséget és az energiafogyasztást. A földrajzi helyzettel kapcsolatos információk alapján a protokoll adaptív módon választja ki az optimális útvonalat a hálózaton keresztül. [5]

A GAF protokoll előnyei közé tartozik a hálózat energiahatékonyságának javítása, a hálózati terhelés kiegyensúlyozása és az adatátvitel megbízhatóságának növelése. A protokoll úgy tudja csökkenteni a hálózat energiafogyasztását, hogy a földrajzi területet, ahol a csomópontok helyezkednek el, szektorokra osztja, és az egy szektorban levő csomópontok közül csak az egyiknek fogja küldeni az adatcsomagot. Hogy melyik csomópontnak küldi, az függ a szektorban levő csomópontok energiakapacitásától, tehát mindig a legnagyobb energiával rendelkezőnek fogja küldeni. A csomagot fogadó csomópont folyamatosan változik, attól függően, hogy a szektoron belül épp melyik rendelkezik a legnagyobb energiatartalékkal, ilyenkor az a bizonyos csomópont aktív lesz, a többi pedig passzív (alvó módba kerül, nem fogyaszt energiát, vagy csak minimálisat). Ezen kívül a protokoll skálázható és alkalmazkodó képessége lehetővé teszi a változó környezeti feltételekhez való rugalmas alkalmazkodást. [12]



12. ábra: *GAF hálózat [12]*

6.1.3.2 NED modul megvalósítása

Az egyszerű ’GAFNode’ modulnak két paramétere van, egy, amivel a csomópont állapotát tartjuk számon, ez két értéket vehet fel: igaz vagy hamis, mivel ’`bool`’ típusú változóról van szó, és ezt alapértelmezetten mindegyik csomópontnál ’`true`’-ra állítjuk, majd az .ini fájlban kedvünk szerint állíthatjuk bármelyik csomópontra vonatkozó értéket. A másik változó pedig az energiaszintjét tárolja, ami kezdetben 5-re van beállítva, és minden üzenetküldéssel csökken, ez a szám csak egy példa, bármennyire állíthatjuk ezt az értéket, attól függően, hogy mennyire szeretnénk növelni vagy csökkenteni a hálózat élettartamát. Ezek mellett a modul még tartalmazza a szokásos ki- és bemeneti kapu-vektor deklarálást, amelyeken keresztül tudjuk küldeni a csomagokat.

```
simple GAFNode
{
    parameters:
        bool state = default(true);
        int energyLevel = default(5);
    gates:
        inout gate[];
}
```

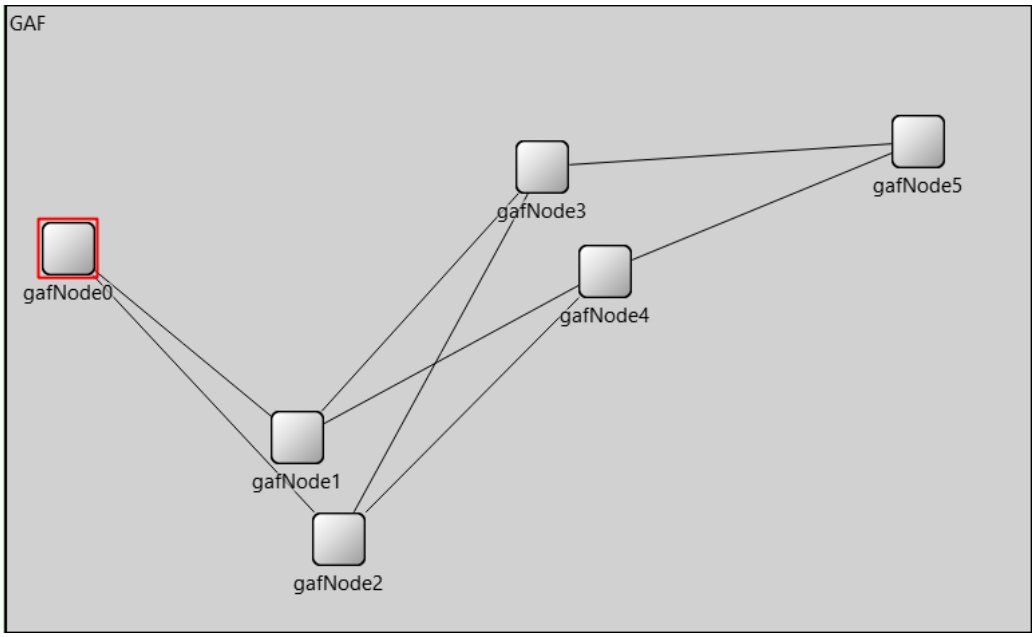
A hálózat deklarálásának a részében létrehozunk egy ’numNodes’, `int` típusú paramétert, ami ebben az esetben is a hálózatban levő csomópontok számát fogja jelölni, viszont most ezt beállítjuk egy alapértelmezett értékre, ami jelenleg 6. Ha szeretnénk növelni a csomópontok számát, a ’`submodules`’ részben hozzá kell adnunk az új csomópont meghatározást, valamint a ’`connections`’ részben be kell építenünk a hálózatba.

```
network GAF
{
    parameters:
        int numNodes = default(6);
        @display("bgb=588,359");
    submodules:

        gafNode0: GAFNode {
            @display("p=35,139");
        }
        gafNode1: GAFNode {
            @display("p=167,248");
        }
        gafNode2: GAFNode {
            @display("p=191,306");
        }
```

```
        gafNode3: GAFNode {
            @display("p=308,92");
        }
        gafNode4: GAFNode {
            @display("p=344,152");
        }
        gafNode5: GAFNode {
            @display("p=524,78");
        }
    connections:
        gafNode0.gate++ <--> gafNode2.gate++;
        gafNode0.gate++ <--> gafNode1.gate++;
        gafNode1.gate++ <--> gafNode3.gate++;
        gafNode1.gate++ <--> gafNode4.gate++;
        gafNode2.gate++ <--> gafNode3.gate++;
        gafNode2.gate++ <--> gafNode4.gate++;
        gafNode3.gate++ <--> gafNode5.gate++;
        gafNode4.gate++ <--> gafNode5.gate++;
}
```

Ebben az esetben nagy jelentőségük van a ’**@display**’ tulajdonságoknak, mivel ezek azt határozzák meg, hogy a csomópont hol helyezkedik el a hálózaton belül, úgymond ezek a csomópont koordinátái. Ezek az értékek szükségesek ahhoz, le tudjuk kérni a csomópont helyzetét, és ez alapján el tudjuk dönteni, hogy mely csomópontok helyezkednek el egy szektoron belül.

Ez a megvalósítás a következő hálózati modellt eredményezi:



13. ábra: *GAF hálózati modell*

Ebben a hálózatban a ’gafNode1’ és ’gafNode2’, valamint a ’gafNode3’ és ’gafNode4’ kerültek egy szektorba. Néhány csomópont között azért van több kapcsolat is, mert előre nem tudhatjuk, hogy az egy szektorban levők közül pillanatnyilag melyik lesz aktív és melyik lesz inaktív, tehát nem tudjuk előre, hogy melyiknek fog kelleni küldeni a csomagot. Hogy kezdetben melyik csomópont milyen állapotban van, az láthatjuk a szimuláción belüli információs üzenetekből, a ’true’ jelöli az aktív, a ’false’ pedig az inaktív csomópontokat:

```
Initializing module GAF.gafNode0, stage 0
INFO: Node 0: 35, 139, state: (omnetpp::cPar)state true
Initializing module GAF.gafNode1, stage 0
INFO: Node 1: 167, 248, state: (omnetpp::cPar)state false
Initializing module GAF.gafNode2, stage 0
INFO: Node 2: 191, 306, state: (omnetpp::cPar)state true
Initializing module GAF.gafNode3, stage 0
INFO: Node 3: 308, 92, state: (omnetpp::cPar)state false
Initializing module GAF.gafNode4, stage 0
INFO: Node 4: 344, 152, state: (omnetpp::cPar)state true
Initializing module GAF.gafNode5, stage 0
INFO: Node 5: 524, 78, state: (omnetpp::cPar)state true
```

14. *ábra:* *Kezdeti állapotok*

6.1.3.3 Hálózati logika implementálása

A ’GAFNode’ osztály deklarálásánál be kell hoznunk egy pár új paramétert, mint például az ’x’ és ’y’ koordináták, vagy a ’`maxEnergy`’, ami egy referencia, ebből tudjuk, hogy egy csomópontnak mennyi lehet a maximum energiája, és a két új statikus vektor változó az állapotok és az energiaszintek tárolására. Azért kell ezek statikusak legyenek, mert nem szeretnénk, hogy minden modul létrejötténél inicializálódjon egy új ilyen típusú vektor, hanem az lenne a cél, hogy az egész szimulációban egy ilyen változó létezzen, amit mindegyik modul el tud érni és használni tud, egy helyen legyen az összes szükséges adat. Ezekhez a vektorokhoz írtam egy-egy kis függvényt, ami különböző elemeket tud hozzáadni ezekhez a vektorokhoz. Mivel az egyik vektor ’`bool`’ a másik pedig ’`int`’ típusú ezért kellett két külön függvény:

```
void GAFNode::addToBoolVector(bool value, std::vector<bool>& vec)
{
    vec.push_back(value);
}

void GAFNode::addToIntVector(int value, std::vector<int>& vec)
{
```

```
        vec.push_back(value);
    }
```

Ezek működése nagyon hasonló, annyiban különböznek, hogy különböző típusú elemeket szúrnak be az adott vektorba. Erre azért volt szükség, mert a kurrens csomóponton kívül egy másik csomópont paramétereit nagyon nehéz elérni, ezek a statikus változók viszont megkönnyítik ezt.

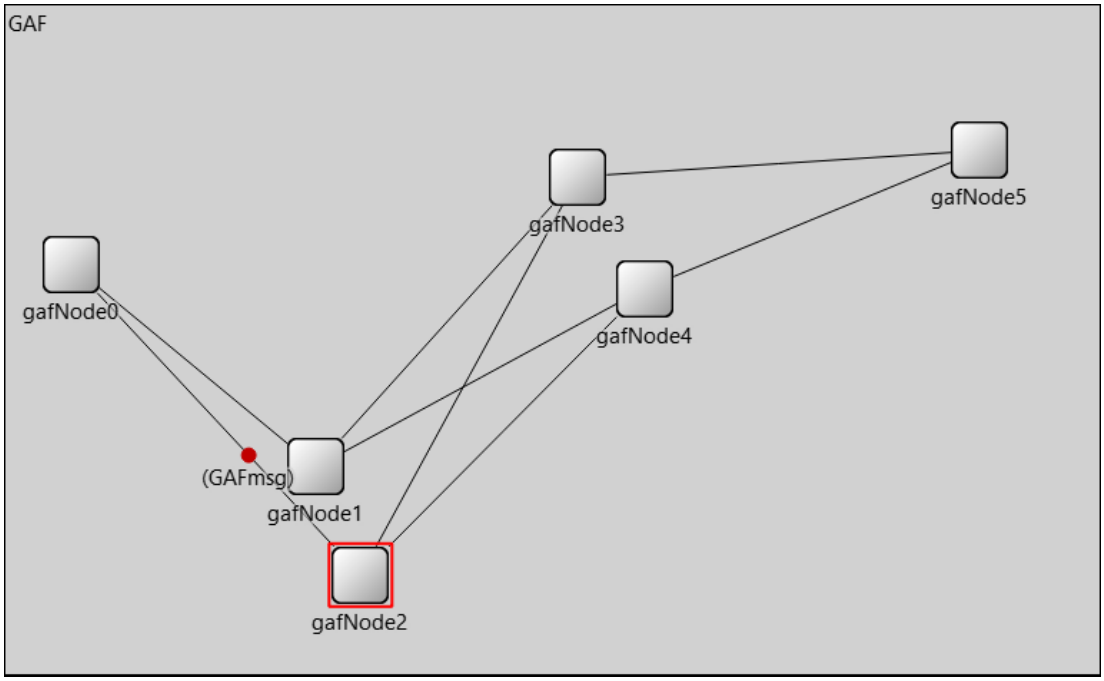
A következőkben a ’`handleMessage()`’ függvény intézi a számításokat, itt az történik, hogy ha a következő szektorban több csomópont is jelen van, akkor megnézzük, hogy melyikük az aktív csomópont, és azt választjuk ki a célállomásnak majd elküldjük az üzenetet. Ha a következő szektorban csak egy csomópont van, akkor értelemszerűen csak annak tudjuk küldeni a csomagot. Ha egy csomópont elküld egy csomagot, akkor annak energiája csökken, és ha eléri a nullát, akkor a csomópont inaktívvá válik, és a szektorában levő másik csomópont lesz aktív, így majd az tudja fogadni és tovább küldeni az adatokat. Viszont, ha egy csomópont inaktív, akkor ez azt is jelenti, hogy tud töltődni, tehát egy idő után visszakapja a maximum energiaszámát, és újra készen áll, hogy aktív legyen, ha a szektorában levő valamelyik csomópont inaktív lesz.

Az inaktív csomópontokat a konfigurációs fájlban előre be tudjuk állítani a következőképpen:

```
**.gafNode1.state = false
**.gafNode3.state = false
```

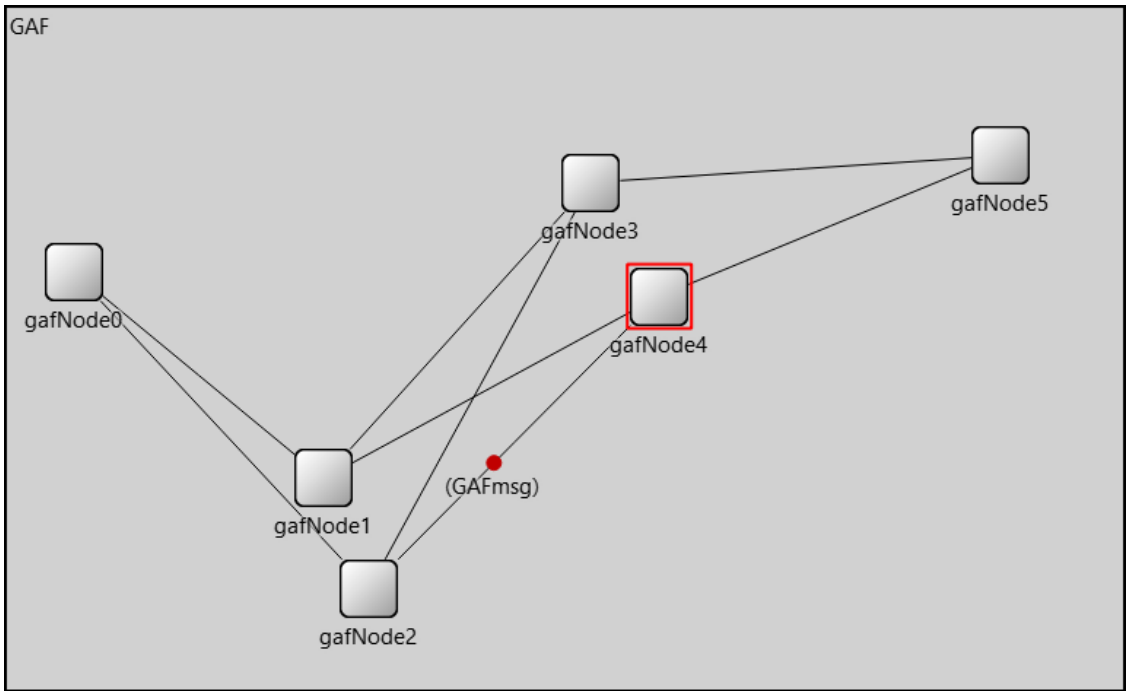
6.1.3.4 Összegzés

Összességében a rendszer működési elve szerint, a kurrens szektorban levő aktív csomópont megvizsgálja a következő szektorban levő csomópontokat, és lekéri, hogy melyik az aktív, ezt a ’states’ nevű vektorból tudja kiolvasni, majd a megfelelő csomópontnak továbbítja a csomagot. A következő ábrán látszik, ahogy a ’gafNode0’ először a ’gafNode2’-nek továbbítja a csomagot, mivel kezdetben a ’gafNode1’ inaktív:



15. ábra: *Adatcsere gafNode0 és gafNode2 között*

Majd a következő lépésben a ’gafNode2’ továbbítja a csomagot a ’gafNode4’ felé, mivel abban a szektorban a ’gafNode3’ az inaktív:



16. ábra: *Adatcsere gafNode2 és gafNode4 között*

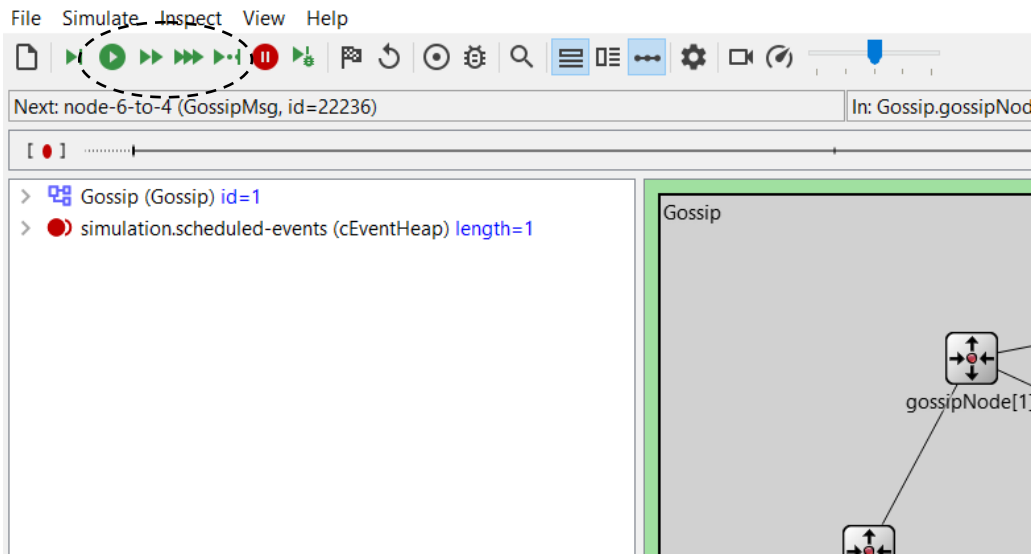
7. Mérési eredmények

7.1. Gossiping

Ebben a fejezetben láthatjuk a Gossiping útválasztó protokoll ugrás számához tartozó mérési eredményeket. Egy protokollban az ugrás szám azt jelenti, hogy az adott adatcsomag hány csomóponton kellett áthaladjon, a küldő csomóponttól indulva, mire elérte a célállomását. Mivel ez a protokoll véletlenszerűen választja ki a célcsomópontot (a kezdő csomópont pedig az előző célállomás), ezért előre nem tudhatjuk, hogy a kezdő- és a célcsomópont között mekkora lesz a távolság. Viszont azt sem tudhatjuk előre, hogy az adatcsomag milyen irányba fog terjedni. Elképzelhető az a szituáció is, hogy a kezdő- és a célcsomópont egyetlen ugrásra vannak egymástól, de a protokoll a csomagot a másik irányba fogja küldeni. Ez jelentősen növeli a csomaghoz tartozó ugrás számot, ami rontja egy protokoll hatékonyságát.

7.1.1. Szimuláció gyorsítása

A szimuláció futási ideje könnyen redukálható, nem kell megvárnunk amíg több száz- vagy több ezerszer lefut a szimulációnk, az indító gomb mellett találunk egy ’Fast Run’ és egy ’Express Run’ gombot ami a szimuláció idejének a felgyorsítására szolgál:



17. ábra: *Szimuláció gyorsítása*

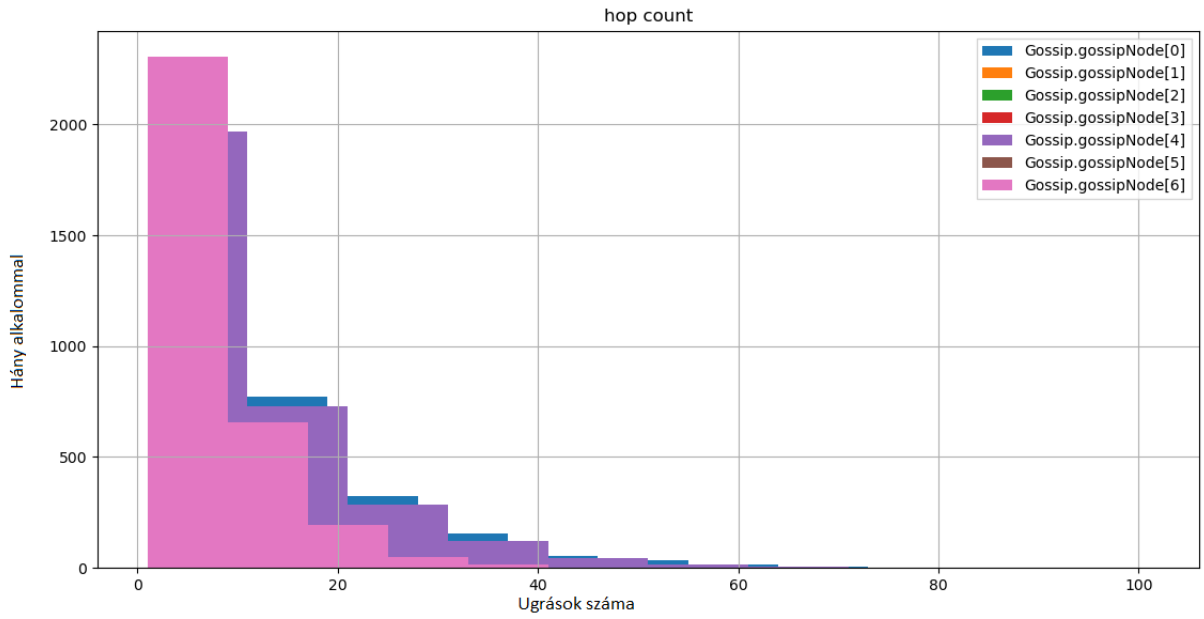
Ennek eredményeképp látható az információs üzeneteknél, hogy ebben az esetben a szimuláció már több mint 22 ezerszer eljuttatta az adatcsomagot a célállomásig, és az utolsó továbbítás már a 170 ezret is meghaladta.

```
INFO: Forwarding message (GossipMsg)node-4-to-6  on gate[1]
** Event #170365  t=0  Gossip.gossipNode[4] (GossipNode, id=6)  on node-4-to-6 (GossipMsg, id=22235)
INFO: Forwarding message (GossipMsg)node-4-to-6  on gate[1]
** Event #170366  t=0  Gossip.gossipNode[6] (GossipNode, id=8)  on node-4-to-6 (GossipMsg, id=22235)
INFO: Message (GossipMsg)node-4-to-6  arrived after 15 hops.
INFO: Generating another message: (GossipMsg)node-6-to-4 (new msg)
INFO: Forwarding message (GossipMsg)node-6-to-4 (new msg) on gate[2]
** Event #170367  t=0  Gossip.gossipNode[5] (GossipNode, id=7)  on node-6-to-4 (GossipMsg, id=22236)
INFO: Forwarding message (GossipMsg)node-6-to-4  on gate[1]
** Event #170368  t=0  Gossip.gossipNode[1] (GossipNode, id=3)  on node-6-to-4 (GossipMsg, id=22236)
INFO: Forwarding message (GossipMsg)node-6-to-4  on gate[0]
** Event #170369  t=0  Gossip.gossipNode[0] (GossipNode, id=2)  on node-6-to-4 (GossipMsg, id=22236)
INFO: Forwarding message (GossipMsg)node-6-to-4  on gate[0]
** Event #170370  t=0  Gossip.gossipNode[1] (GossipNode, id=3)  on node-6-to-4 (GossipMsg, id=22236)
INFO: Forwarding message (GossipMsg)node-6-to-4  on gate[1]
```

18. *ábra:* *Információs üzenetek – adattovábbítás*

7.1.2. Eredmények

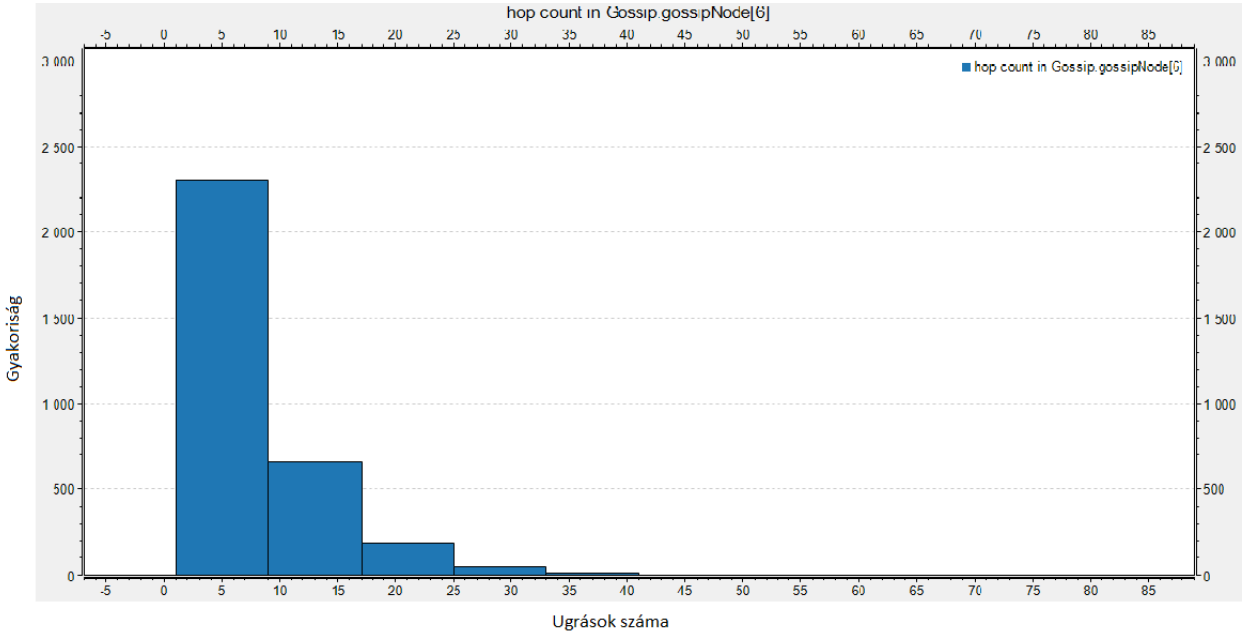
A következő ábrán láthatók a Gossiping protokoll ugrás számai.



19. ábra: *Ugrások száma gyakoriság szerint*

Az oszlopok azt jelentik, hogy egy bizonyos ugrás szám hány alkalommal történt meg, csomópontokra lebontva. Az ábrában rejlő legfontosabb információ, hogy a legtöbb esetben, az adatcsomag, már 10 ugrás alatt elérte a célállomást, de minimum 1 kellett hozzá, ez érthető is mert egy csomópont nem tud saját magának küldeni adatot. Viszont olyan esetek is előfordultak, hogy több mint 30 vagy 40 ugrás kellett, hogy az adat célba érjen. Ez egy ilyen méretű hálózatnál nagyon rossz érték, mivel összesen 7 csomópontot tartalmaz, ezért egy optimális úton, a hálózat felépítése szerint, maximum 3 ugrással célba lehetne juttatni az adatot bármelyik két csomópont között.

Csomópontonként a következőképpen néz ki ez az eredmény, a 6-os csomópont a következő gyakorisággal produkálta a különböző ugrás számokat:



20. ábra: *Ugrások - 6-os csomópont*

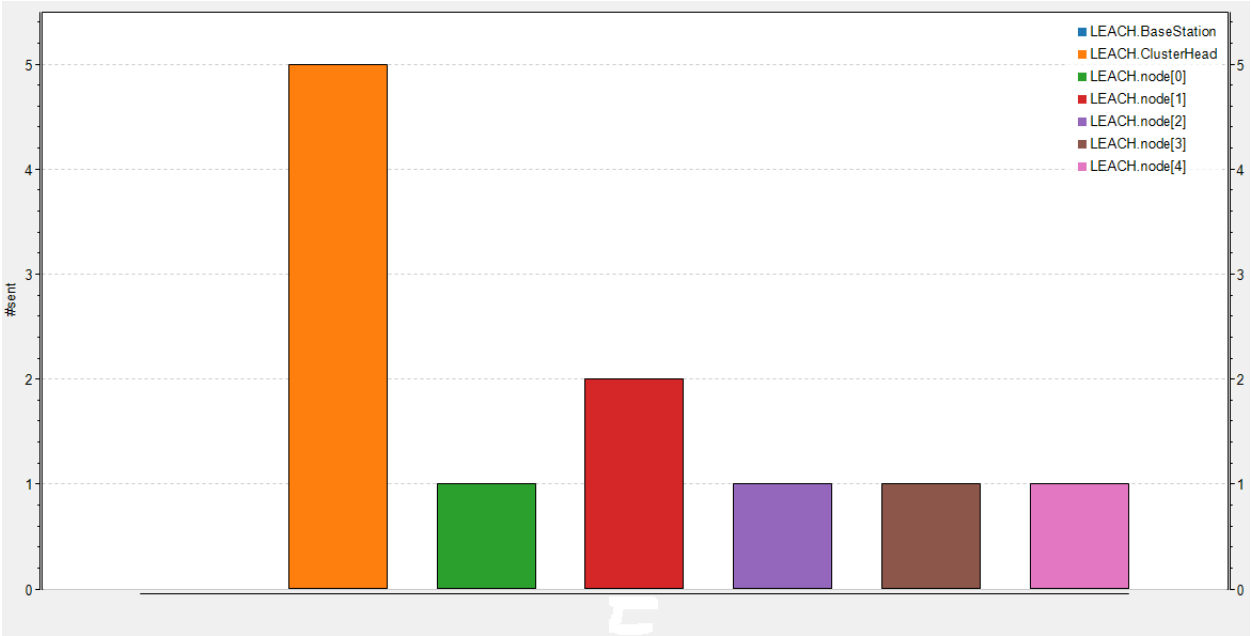
Itt az látható, hogy 2300-szor történt 1 és 9 közötti ugrás szám, ez volt a leggyakoribb, a legtöbb pedig 41 ugrás kellett az adat célba juttatásához.

7.2. LEACH,

A LEACH protokollnál közrejátszik egy adatvesztési tényező, amely megadja, hogy egy adatcsomag milyen eséllyel veszhet el egy továbbítás során, ennek az esélye 10%. Ebben az esetben az adatcsomag másolatát újra el kell küldeni a célcsomópontnak, ezt annyiszor tudjuk megtenni ahányszor szükséges (ha egymás után többször is elvesztődne az adatcsomag). Ennek a

statisztikának a mérésére bevezettem két változók, az egyik az elküldött csomagok számát rögzíti, a másik pedig a megérkezett csomagok számát, a végén pedig láthatunk valamennyi különbséget a kettő között, ha út közben történt csomagvesztés.

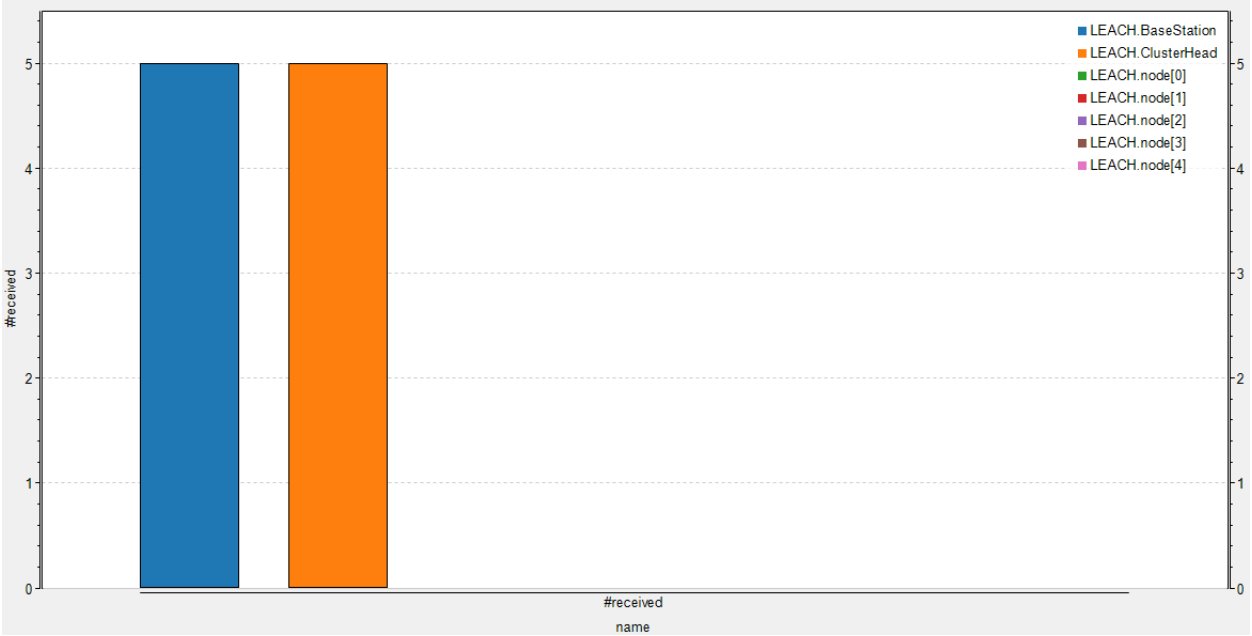
A következő ábrán láthatjuk az elküldött csomagok számát csomópontok szerint:



21. ábra: *Elküldött adatcsomagok száma*

Az ábrán látható első és kiemelkedő oszlop a klaszter fej továbbított adatcsomagainak a száma, ez az érték 5, mivel a klaszter fejen kívül 5 csomópont volt még a klaszterben, és ezek mindegyik üzenetét továbbította a bázisállomás felé anélkül, hogy adatvesztés történt volna. A bázisállomás ezen a grafikonon nem látszik, mivel neki 0 továbbított üzenete van, de neki nem is ez a feladata, hanem csak fogadja az üzeneteket. Az ábra többi részében láthatjuk az 5 csomópontot, ahol kiemelkedik az 1-es indexű csomópont, annál azt láthatjuk, hogy a többi csomóponthoz képest egyel több elküldött üzenete van, ez azt jelenti, hogy történt egy adatcsomag vesztés, amit újra kellett küldeni.

A következő ábra szemlélteti a fogadott adatcsomagok számát:

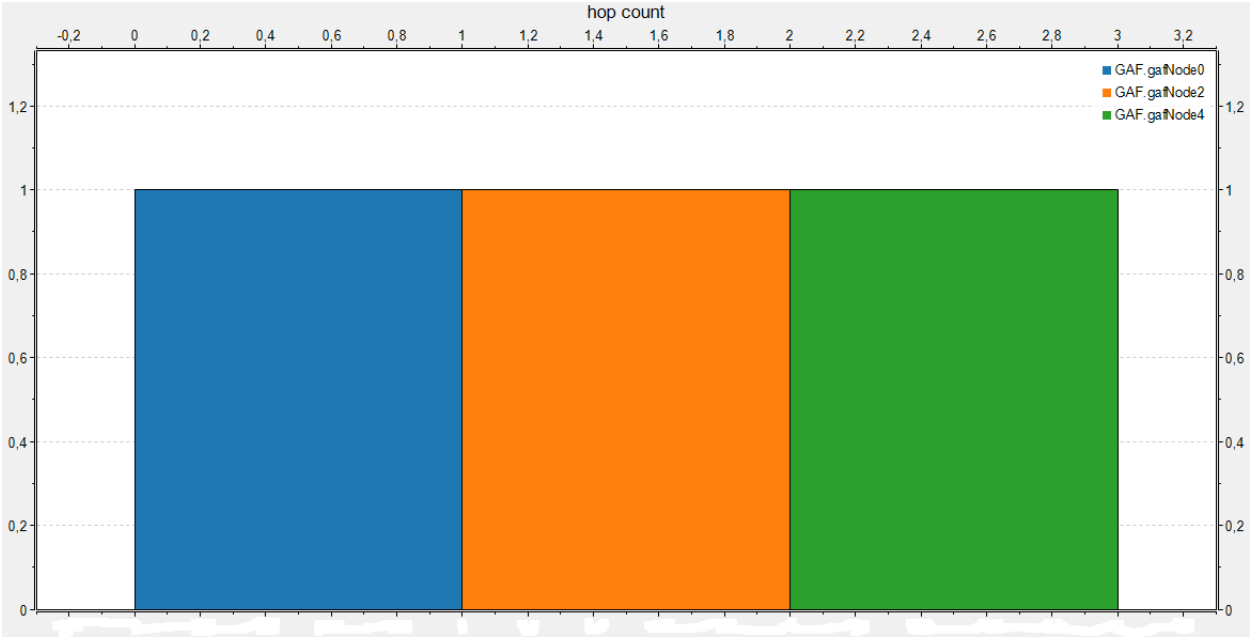


22. ábra: *Fogadott adatcsomagok száma*

Ezen az ábrán csak a bázisállomás és a klaszter fej látható, mivel csak ezek fogadtak adatot, látható, hogy mindkettőhöz sikeresen megérkezett mind az öt csomag, amiket kezdetben a klaszter fej összegyűjtött. A többi csomópont csak küldő, tehát ők nem kell fogadjanak semmilyen adatot.

7.3. GAF

A protokoll sajátossága, hogy a hálózatban nem használ fel minden csomópontot az adat továbbítására, szektoronként csak 1 aktív csomópont van, ezért a rögzített adatok is csak néhány csomópontra szólnak. Az alábbi ábrán látható a hálózatban levő aktív csomópontok ugrás számai, amelynek értéke mindegyiknél 1, mivel a hálózat veszteség nélkül továbbítja az adatokat, így minden adatcsomag csak egyszer halad át egy csomóponton. Ebben az esetben a csomag útvonala: gafNode0 -> gafNode2 -> gafNode4 -> gafNode5. A 0 indexű ponttól indult és a célállomása az 5-ös indexű, az utolsó csomópont nem szerepel az ábrán, mivel az nem hajtott végre küldést.



23. ábra: *GAF protokoll ugrás számai*

8. Következtetések

8.1. Megvalósítások

A projekt során sikeresen tanulmányoztam és megvalósítottam három típusú útválasztó protokollt, ezek a következőek: Gossiping, LEACH, GAF. Ezek szimulációját az OMNeT++ nevű keretrendszer és fejlesztői környezetben végeztem. A szimulációk során láthatjuk a hálózatban levő adatcsomagok terjedését a csomópontok között, az adott protokollnak megfelelően. Valamint a szimuláció futása során különböző statisztikai adatok kerülnek rögzítésre, amely lehet például az adatcsomagok ugrás száma, vagy az elküldött / fogadott csomagok száma.

8.2. Továbbfejlesztési lehetőségek

A szimulációk tartalmaznak néhány továbbfejlesztési lehetőséget, a kódot néhány helyen lehetne optimalizálni. A LEACH protokoll megvalósításánál a szimulációban csak egy klasztert alakítottam ki, ennek a számát esetleg lehetne növelni, például két vagy három klasztert is ki lehetne alakítani, viszont ez növelné a szimuláció futási idejét. Ezt viszont csökkenteni lehetne azzal, hogy megoldjuk az adatcsomagok egyszerre küldésének a problémáját, ez onnan ered, hogy OMNeT++-ban a hálózati logika csomópontonként külön-külön fut le, egymás után és nem

egyszerre. Erre nem találtam megoldást, így nem tudtam kivitelezni, hogy a csomópontok egyidőben küldjék az adatcsomagot a klaszter fejnek, de ez nincs hatással a szimuláció működésére, vagy hatékonyságára.

A GAF protokollnál a szektorok elkülönítése statikusan van meghatározva, tehát előre tudjuk, hogy a csomópontok hol vannak elhelyezve és melyek vannak egy szektoron belül és így előre tudjuk, hogy melyik csomópont melyik szektor csomópontjai közül kell kiválasztja a célállomást. Ezt lehetne dinamikussá tenni, hogy a csomópontok helyzete alapján döntse el magától, hogy melyek vannak egy szektorban, viszont ez több nehézséggel is járna. Ebben az esetben előre össze kellene kötni mindegyik csomópontot mindegyikkel, mivel előre nem fogjuk tudni, hogy egy adott csomópont kinek kell majd küldjön adatot, ez viszont nagyon bonyolulttá tenné az egész rendszert, és kevésbé lenne átláthatóbb.

9. Hivatkozások

[1] S. S. S. Deepak M. Birajdar, „LEACH: An energy efficient routing protocol using Omnet++ for Wireless Sensor Network," International Conference on Inventive Communication and Computational Technologies (ICICCT), 2017. [Online]. [Hozzáférés dátuma: 2023].

[2] V. S. M. S. Vishal Kumar Arora, „A survey on LEACH and other's routing protocols in wireless sensor network," Department of Computer Science & Engineering, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0030402616303199?via%3Dihub#sec0010. [Hozzáférés dátuma: 2023].

[3] „OMNeT++ Simulation Manual," [Online]. Available: https://doc.omnetpp.org/omnetpp/manual/. [Hozzáférés dátuma: 2023].

[4] A. Sekercioglu, „OMNeT++ Technical Articles," OpenSim Ltd., 2019. [Online]. Available: https://docs.omnetpp.org/tutorials/tictoc/. [Hozzáférés dátuma: 2023].

[5] S. G. Túrós László-Zsolt, „Érzékelők és mérőhálózatok," Scientia Kiadó, 2022. [Online]. Available: http://real.mtak.hu/143818/1/Erzekelok%20es%20merohalozatok_interior_REAL.pdf. [Hozzáférés dátuma: 2023].

[6] R. K. Kodali, N. Rawat és L. Boppana, „WSN sensors for precision agriculture," 2014. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6863114. [Hozzáférés dátuma: 2023].

[7] H. Aznaoui, S. Raghay, L. Aziz és A. Ait-Mlouk, „A comparative study of routing protocols in WSN," International Conference on Information & Communication Technology and Accessibility (ICTA), 2015. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7426884. [Hozzáférés dátuma: 2023].

[8] H. C. Dongsoo S. Kim, „Adaptive Duty-Cycling to Enhance Topology Control Schemes in Wireless Sensor Networks," International Journal of Distributed Sensor Networks, 2014. [Online]. Available: https://www.researchgate.net/figure/Geographical-adaptive-fidelity-GAF_fig11_275468645. [Hozzáférés dátuma: 2023].

[9] „What is OMNeT++?," 2019. [Online]. Available: https://omnetpp.org/intro/.

[10] W. Yen, C.-W. Chen és C.-h. Yang, „Single gossiping with directional flooding routing protocol in wireless sensor networks,” 2008. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4582790. [Hozzáférés dátuma: 2023].

[11] A. Tsapanoglou, „ResearchGate - Gossiping,” [Online]. Available: https://www.researchgate.net/figure/Gossip-protocol-response-for-a-static-6-nodes-network_fig1_224562599. [Hozzáférés dátuma: 2023].

[12] J. S. S. M. Grover, „Optimized GAF in Wireless Sensor Network,” Proceedings of 3rd International Conference on Reliability, Infocom Technologies and Optimization, 2014. [Online]. [Hozzáférés dátuma: 2023].

[13] „OMNeT++ Simulation Library,” doxygen, 2022. [Online]. Available: https://doc.omnetpp.org/omnetpp/api/index.html. [Hozzáférés dátuma: 2023].

UNIVERSITATEA SAPIENTIA DIN CLUJ-NAPOCA

FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE, TÎRGU-MUREȘ

SPECIALIZAREA CALCULATOARE

Vizat decan

Conf. dr. ing. Domokos József

Vizat director departament

Ș.l. dr. ing Szabó László Zsolt