MŰEGYETEM 1782

SZAKDOLGOZAT-FELADAT

Járó Áron (AU1V3G)
szigorló villamosmérnök hallgató részére

Threadalapú szenzorhálózatot vezérlő szoftver fejlesztése

Napjaink egy aktív kutatási és fejlesztési területe az otthonautomatizálás, intelligens otthon. A koncepció egyik alapja az, hogy az egyes otthoni eszközökbe épülő intelligens eszközök egymással kommunikálni tudnak, lehetőleg vezeték nélkül. Az iparban már sokféle megoldás létezik, ezek közül ez egyik legújabb és legbiztatóbb kommunikációs protokoll az IPv6 alapú Thread. A Thread protokoll kifejezetten azzal a céllal jött létre hogy az otthoni, lakóházi környezetben tudjon összekapcsolni az eszközöket vezeték nélkül.

Jelenleg dolgozat célja egy Threadalapú hálózat implementálása. Ehhez a hardver elemek egyrészt készen kapható, NXP által gyártott kártyák, amelyek támogatják a Thread protokollt, másrészt egy Raspberry PI beágyazott számítógép, amely gateway szerepben van az IPv4 és 6LoWPAN között.

A feladat fő része a gateway Raspberry PI-n futó Network Management Modul (NMM) megtervezése és implementálása, amely a hálózat képességeinek demonstrálására alkalmas magas szintű funkciókat valósít meg. Feladat továbbá a Thread hálózat végpontjainak firmware-fejlesztése is, amely csatlakozik az NMM-hez illeszkedően.

A hallgató feladatának a következőkre kell kiterjednie:

- Thread vezeték nélküli szenzorhálózat megismerése – Irodalomkutatás
- IPv4 és 6LoWPAN közötti gateway Network Management Modul szoftver tervezése
- IPv4 és 6LoWPAN közötti gateway Network Management Modul szoftver implementációja
- Thread hálózat végponti egységek firmware fejlesztése
- A működés demonstrációja

Tanszéki konzulens: Sujbert László, egyetemi docens
Külső konzulens: Molnár Károly, ProDSP Technologies Kft.

Budapest, 2017. október 6.

……
Dr. Dabóczi Tamás
tanszékvezető

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Threadalapú szenzorhálózatot vezérlő szoftver fejlesztése

Készítette

Járó Áron

Tanszéki konzulens

Sujbert László

Külső konzulens

Molnár Károly

2017

Tartalom

Hallgatói nyilatkozat

Alulírott Járó Áron, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulensek neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik.

Kelt: Budapest, 2017. 12. 15.

.....

Járó Áron

Kivonat

Az IoT az évtized egyik jelentős technológiai vívmánya – a hálózatba kapcsolt beágyazott eszközök száma gyorsuló ütemben növekszik világszerte. A koncepció jellemző alkalmazási területe, az épületautomatizálás egyik új és igéretes megoldása a Thread hálózati protokoll. Jelen dolgozatban bemutatott munka egy, a Threadre épülő hálózat képességeit bemutató világításvezérlő rendszer fejlesztésének egyik részfeladata. A Thread fejlesztői a protokoll szerinti működést megvalósító hardverkomponenseket, és ezeken futó firmware mintákat tettek elérhetővé, ezek a rendszerfejlesztés kiinduló pontjai voltak.

A világításvezérlő projekt megvalósításához, és így a Threadben rejlő lehetőségek demonstrálásához egy hálózati vezérlő szoftverkomponens fejlesztése volt szükséges, amely a hálózat állapotába egy felhasználói felületen keresztül beavatkozást tesz lehetővé. A vezérlés megvalósításában a Thread fejlesztői által rendelkezésre bocsátott vezérlő interfész nagy szerepet játszott. Ezen felül azonban, mivel a hálózat működésének, és a felhasználói interakcióknak a folyamatai különböző rendszereken zajlanak és egymástól függetlenül időzítettek, a feladat megoldása során a kihívást elsősorban a szoftverfolyamatok szinkronizálása és a folyamatok közötti adatátvitel megvalósítása jelentette. A megoldáshoz a Linux kernel FIFO impelementációját, valamint a POSIX által definiált mutex és feltételváltozó mechanizmusokat használtam fel. A vezérlőrendszer implementálásán túl a modellhálózat hálózati csomópontjainak firmware fejlesztése is szükséges volt, ennek alapjául a Thread fejlesztői által elérhetővé tett mintakódok szolgáltak.

A felállított modellhálózat a specifikált világításvezérlő rendszer funkciót ellátja, ezen keresztül a Thread lehetőségeit bemutatni képes. A dolgozatban bemutatom, hogy a féléves munkám során a feladatkiírás által specifikált összes részfeladat megoldását elvégeztem. Az ennek során megvalósított vezérlő modul a világítási rendszer specifikumaitól függetlenül, más kontextusú Threadalapú hálózatok vezérlésére újrafelhasználható.

Abstract

The Internet of Things is one of the major technological achievements of the decade – the number of connected embedded devices is rising worldwide at an accelerating rate. The Thread networking protocol is a new and promising solution for one of the application uses of the IoT concept, home automation. The work presented in this thesis is a subtask of development of a lighting control system. Hardware components and firmware examples capable of Thread-based operation were made availible by the delevopers of Thread, these were the starting points of development.

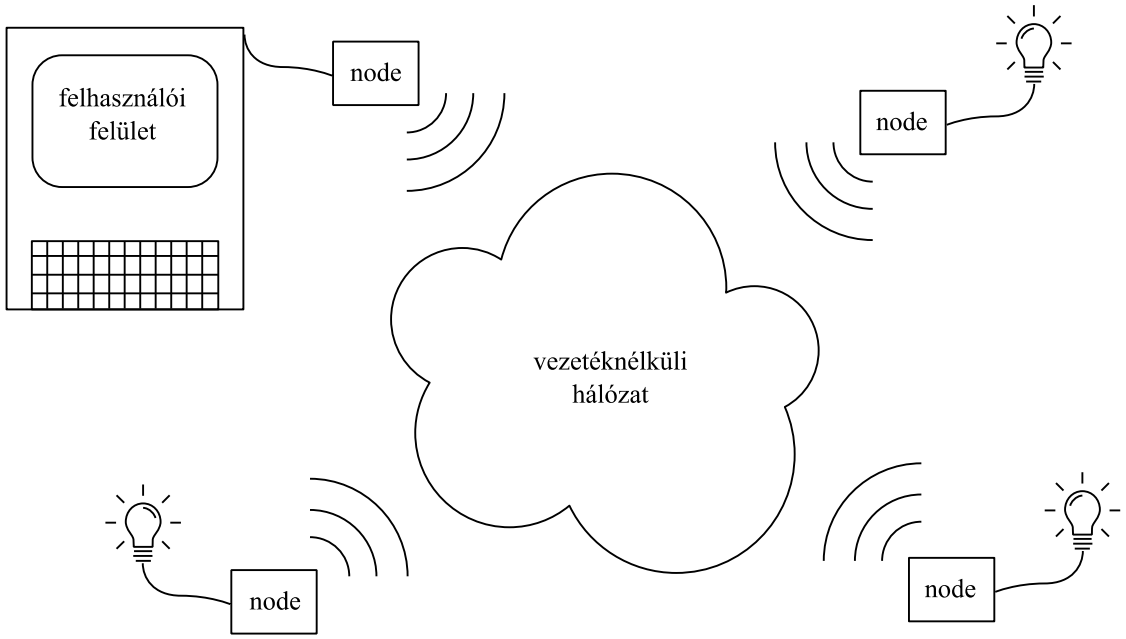
In order to implement the lighting control project, and therefore to demonstrate the potential of Thread, the development of a network management software component was needed, which allowed the mutation of the state of the network through a human user interface. In implementation of network management, usage of the Thread Host Control Interface played a major role. Nevertheless, since the processes of the network and user interactions are spread out on several machines, and are independent in timing, the main challange of development lied in synchronizing the independent software processes, and implementing communication between them. In solving these problems I used the FIFO implementation availible in the Linux kernel, as well as mutex and condition variable mechanisms, as defined by POSIX. In addition to creating a network management software, the firmware development of the network nodes was also necessary, the firmware versions were based on the examples projects made availible by the developers of Thread.

The established model network performs the specified functions of the lighting control system, through this, it is capable of demonstrating the power of Thread-based networks. In the thesis I show that during the semester I have accomplished all goals specified in the assignment. The created network management module is reusable to control Thread-based networks deployed in contexts that are independent of the specifics of the lighting control project.

1. Bevezetés

A 2010-es évek egyik jelentős technológiai hívószava az IoT – Internet of Things, azaz a Dolgok Internete. A koncepció mérnöki megoldásokat nyújthat az automatizált adatgyűjtés, illetve flottakövetés területén, szolgáltatók számára a felhasználóik viselkedése vizsgálatának, és felhasználói visszajelzések gyűjtésének eszközét jelenti, lehetővé teszi távoli rendszerek vezérlését, épületek gépészeti automatizálását, ipari automatizálást. A koncepció sok piaci területen nagy érdeklődésre tart számot, egész piaci szegmensek átalakulását indíthatja meg [1] . Az IoT eszközök megvalósításának alapja a hálózatba kapcsolt beágyazott rendszerek együttese. Ezen beágyazott rendszerek érzékelő, illetve szabályozó feladatokat látnak el üzembe helyezésük területén, és hálózati kapcsolódásuknak köszönhetően autonóm módon adatátviteli és vezérlési tranzakciókat intézhetnek a hálózaton belül.

Az ebben a dolgozatban bemutatott feladat egy piaci szereplő által megrendelt fejlesztési projekt egy részfeladata. A megrendelő egy tetszőlegesen skálázható, központilag vezérelt világítási rendszer fejlesztését jelölte ki. A specifikáció szerint a világítási rendszert alkotó világítótestek hálózatba kapcsolható vezérlőegységekkel vannak ellátva. A vezérlőegységek, és így a lámpák működését emberi felhasználó irányíthatja egy központi kezelői felületen keresztül. Az 1. ábrán a világításvezérlő rendszer vázlata látható. Az alkalmazandó technológiákat, így a hálózati protokollt, illetve a vezérlőegységek hálózati hardverét a megrendelő a specifikációban rögzítette.



1. ábra: A világítási rendszer koncepciója

A feladat specifikációja egy alacsony adatátviteli sebességű, és alacsony fogyasztású szenzorhálózat felépítését teszi szükségessé. Az így definiált problémára több, jelenleg egymással versenyben lévő hálózati technológia is megoldást nyújthat, a projekt gazdája ezek közül a viszonylag fiatal (2014-ben bejelentett) Thread protokollt jelölte ki. A protokollt számos, IoT technológiában érdekelt világvezető vállalat (például a Samsung, vagy a Google tulajdonában álló Nest) támogatja. A protokoll, ígérete szerint egy biztonságos, alacsony fogyasztású, robusztus rácsháló topológiájú hálózat létrehozását teszi lehetővé, amelyben egy hálózati csomópont (a továbbiakban node) meghibásodása nem okozhatja [2] a teljes hálózat működésképtelenné válását. A protokoll fizikai rétegeként a széles körben elterjedt IEEE 802.15.4 rádiós adatátviteli szabványt alkalmazza, valamint a hálózat node-jai IPv6 protokoll szerint címezhetőek, ezért a fejlesztői a Thread-et egy „jövőbiztos” IoT megoldásnak tekintik.

A specifikáció szerint egy ilyen protokollra épülő hálózat vezérlőrendszerének fejlesztése a feladat, amely a világításvezérlő rendszer magas szintű funkcióit valósítja meg. Azonban a projekt elsődleges célját tekintve a világításvezérlésnek nincs különösebb jelentősége, az a piaci igények szerint a későbbiekben módosulhat. Ami elsődleges szempont, hogy a fejlesztett rendszer a Thread protokollra épülő hálózat működését demonstrálja, illetve, hogy tetszőleges IoT probléma megoldására bevethető legyen. A fejlesztett vezérlőrendszer működése ezért a világítási rendszer által meghatározott funkciókhoz nem szorosan csatolt, a későbbiekben bármilyen Threadalapú hálózat vezérlése végett üzembe helyezhető, újrafelhasználható.

A vezérlőrendszer a Thread hálózat vezérlésének alapfunkcióit valósítja meg, a világításvezérlés sajátosságai csak a rendszer felhasználói felületén, illetve az üzembe helyezett node-ok működésének alacsony szintjén jelennek meg. A felhasználói felület a specifikáció szerint egy weblap, amelyet egy webszerver hagyományos HTTP kapcsolat segítségével, IPv4 hálózaton keresztül szolgál ki. Így a megvalósított vezérlőrendszer a felhasználói felületével együtt a Thread hálózat belseje (IPv6 címezhető node-ok) és az Internet (IPv4) közötti átjárót valósítja meg.

2. Irodalomkutatás

2.1. A Thread protokoll bemutatása

A Thread hálózathoz illesztett vezérlőrendszer fejlesztése előtt a Thread hálózati protokoll sajátosságaival kellett megismerkednem. A protokoll számomra, azaz a feladat megoldása szempontjából releváns részeit a Thread hálózat hivatalos specifikációjának [3] tanulmányozása útján ismertem meg.

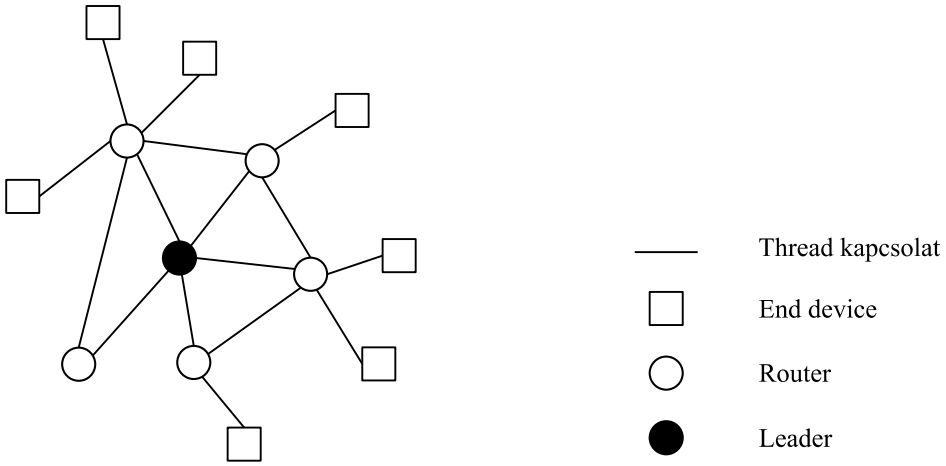
2.1.1. Fizikai réteg és 6LoWPAN

A Thread hálózat fizikai rétegét IEEE 802.15.4 szabvány szerinti rádiós kommunikáció valósítja meg. A szabvány alacsony adatforgalmú vezetéknélküli adatátvitelt ír le, széles körben elterjedt, számos vezetéknélküli technológia fizikai rétegeként használja. A már meglévő, bejáratott technológia Threadben való alkalmazása azzal az előnnyel jár, hogy a hálózati protokoll fejlesztése nem jár új rádiós technológia fejlesztésének igényével.

A Thread protokoll fejlesztése során a node-ok IPv6 hálózati réteg alkalmazása mellett döntöttek, így elősegítve a hálózat elemeinek Interneten keresztül történő elérését. A hálózat node-jai több IPv6-címet is regisztrálhatnak, kontextustól függően többféle címen is elérhetőek. Az IPv6 protokoll jellemzői a 128 bit hosszú hálózati címek, és az 1280 bájtos maximális csomagméret. Ezek egyértelműen az Internetes szolgáltatásokkal szemben támasztott növekvő adatmennyiségbeli elvárásoknak tudhatók be. Azonban az IEEE 802.15.4 szabvány alacsony adatforgalmú és kis teljesítményű hálózatok megvalósítására nyújt megoldást, ennek megfelelően a maximális kerethossz alacsonyabb, 127 bájt. A Thread mind az Interneten keresztüli címezhetőség, mind az alacsony teljesítményű vezetéknélküli hálózat elvárásainak meg kíván felelni, így a hálózati réteg (IPv6) és fizikai réteg (IEEE 802.15.4) között a 6LoWPAN (IPv6 over Low Power Wireless Personal Area Networks) [4] adaptációs réteget valósítja meg. Az előzőekhez hasonlóan a 6LoWPAN is egy önálló, már széles körben elterjedt szabvány, amely az IPv6 fejlécek tömörítését, valamint a datagrammok feldarabolását és összeillesztését valósítja meg.

2.1.2. A hálózat felépítése, szerepek

A Thread egy rácstopológiájú, úgynevezett mesh hálózat felépítését teszi lehetővé. Az így felállított hálózat node-jai között többszörös kapcsolat áll fenn, így, ha egy útválasztóként működő (router) node meghibásodik, a feladatát egy szomszédos node átveheti. A mesh topológia vázlatát a 2. ábra mutatja. A Thread hálózatot felépítő node-ok hálózatban betöltött szerepük szerint különbözőek lehetnek.



2. ábra: A Thread mesh topológiájának szemléltetése

A mesh topológia végpontjaiként működő (end device) node-ok csak a hozzájuk tartozó szülőn (parent node) keresztül fogadnak illetve küldenek üzeneteket. Ezen node-ok számára csak a szülőjük IP-címének ismerete szükséges, és a működésük a teljes hálózati kommunikációnak csak egy kis részére terjed ki. Ezért az ilyen szerepben lévő node-ok energiaigénye, fogyasztása könnyen minimalizálható. Az end device típus speciális esete a sleepy (aluszékony) end device, amely eszközön a mikrokontroller és egyéb hardverkomponensek csak periodikusan, rövid időre ébrednek fel, a köztes időben a fogyasztásuk minimális. A hálózati kommunikációban a köztes időben nyilván nem vesznek részt. Ébredés után bejelentkeznek a szülőjükhöz, aki a nekik címzett üzeneteket az ébredésükig eltárolja, és csak a periodikusan ismételt bejelentkezés után továbbítja azokat. Az ilyen működésű node-ok használatával, a Thread igérete szerint elérhető, hogy hálózatba kapcsolt eszközök akár évekig működhessenek telepről, külső energiaforrás nélkül.

A router node-ok több hálózati node-dal is kapcsolatot tarthatnak fenn, illetve end device típusú node-ok szülőjeként funkcionálnak. A velük közvetlen kapcsolatban lévő node-ok címén kívül egy teljes router-táblát is fenntartanak, amelyben az összes, a hálózaton éppen router szerepben működő node teljes elérési útja rögzítve van. A hálózaton vett üzenet esetén a router-tábla alapján továbbítják azt a címzetthez legközelebbi router irányába. A hálózaton egy pillanatban legfeljebb 32 router lehet aktív, ezek le kell hogy fedjék a hálózat teljes területét. A korlátozás célja többek között a minden router számára tárolni szükséges router-tábla memóriaigényének minimalizálása. A hálózat router képességű node-jai ezért router-eligible, azaz routerré választható állapotban is lehetnek. Ebben az esetben ezek a node-ok valójában end device-ként működnek, de szükség esetén routerré válhatnak.

A Thread hálózatban router szerepben működő node-ok egyike kitüntetett fontosságú, vezér node (leader). A routerek közül a leader kiválasztása autonóm módon történik, ha

a leader a hálózatról valami okból lecsatlakozik, meghibásodik, a szerepét egy másik router veszi át. A leader engedélyezi a router-eligible eszközök routerré válását, ő osztja ki a routerek IP-címeit. A hálózatot leíró paraméterek naprakészen tartásáért és kérésre továbbításáért is a leader felelős.

2.1.3. CoAP kommunikáció

A Thread a Constrained Application Protocolt (kis erőforrásigényű alkalmazási protokoll, röviden CoAP) [5] alkalmazza hálózaton belüli adatátvitel alkalmazási rétegbeli megoldásaként. A CoAP webalapú adatátviteli módszer, amelynek alkalmazása kis memóriaigénnyel és számításigénnyel jár, ezáltal ideális a kis teljesítményű beágyazott rendszerek közötti kommunikáció megvalósítására. A protokollt elsősorban gépközi adatátvitel és épületautomatizálás céljaira fejlesztették.

A CoAP az elérni kívánt erőforrásokat karaktersorozatokkal (Uniform Resource Identifier, egységes erőforrás-azonosító, röviden URI) azonosítja. A karaktersorozat elején a CoAP sémát jegyzi, majd az elérni kívánt erőforrást kiszolgáló gazdarendszer azonosítása következik (a Thread hálózat kontextusában ez a megszólítani kívánt hálózati node által regisztrált egyik IP-cím és port). Az elérni kívánt gép azonosítása után a karaktersorozat URI-path (elérési út) mezője jön, amely az elérni kívánt erőforrást azonosítja. Az URI ezek után opcionálisan tartalmazhat URI-query mezőt, amelyben paramétereket adhat át a megcímzett erőforrásnak (ebben a kontextusban ez az átvitt, feldolgozandó adattartalom).

2.2. Thread vezérlő interface – THCI

A Thread Host Control Interface (továbbiakban THCI), egy Thread hálózatba kapcsolható node és egy PC közötti kapcsolódási felület kiépítését teszi lehetővé, soros porton keresztül. Az interfész definiálja a soros porton továbbítandó üzenetkereteket, amelyek a vezérlést és megfigyelést lehetővé teszik.

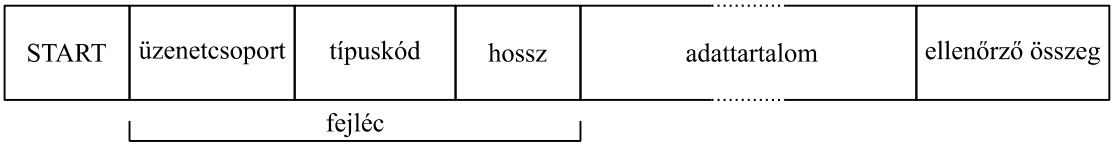
2.2.1. Üzenetkeretek csoportjai

A kommunikációt leíró üzenetkeretek három csoportba sorolhatók: kérés, válasz és jelzés. A vezérlő rendszer tranzakciót kezdeményezhet, ekkor egy kérés csoportba tartozó keretet továbbít a felcsatolt node felé. A node erre válasz üzenetet küld vissza, amelyben jelzi, hogy a kapott kérést vette, feldolgozta, szükséges esetben továbbította. Ha a kérés keretben olyan tranzakció került meghívásra, amelyet a felcsatolt node azonnal végre tud hajtani (pl. a felcsatolt node saját regisztrált IP-címének lekérése), a kérésre küldött válaszkeretben a tranzakció eredményét is szolgáltatja. Más esetben, azaz, ha a tranzakciónak a hálózat valamely más komponense a címzettje (pl. CoAP üzenet továbbítása),

vagy előre nem becsülhető idő alatt megy csak végbe (pl. a felcsatolt node gyári alaphelyzetbe állítása), a válasz keret csupán a kérés sikeres feldolgozását, illetve továbbítását jelzi. A hálózat egy távoli komponense felől érkező üzenetet, illetve a hosszú ideig tartó tartó hálózati események eredményét a felcsatolt node jelzés csoportba tartozó üzenetkeretben továbbítja a vezérlő rendszer felé.

2.2.2. Üzenetkeretek felépítése

A THCI által meghatározott üzenetkeretek általános felépítése a 3. ábrán látható.



3. ábra: THCI üzenetkeretek általános felépítése

A vezérlést szolgáló üzenetkeretek felépítését a THCI rendszer fejlesztői definiálták, és egy kézikönyvben [6] elérhetővé tették. A soros porton továbbítandó keretek szerkezete a következő. A keretek mindegyike startmezővel kezdődik, amelyet az üzenet fejléce követ. A fejléc első mezője az üzenetcsoport, amely ebben az esetben a kimenő (vezérlő rendszer felől a felcsatolt node felé tartó), illetve bejövő üzeneteket különbözteti meg. Az üzenetcsoport után az üzenet típuskódja következik. A rendszer fejlesztői a különböző hálózati tranzakcióknak különböző üzenettípust feleltetnek meg, így a tranzakciók a típuskód alapján azonosíthatók. Saját típuskóddal rendelkeznek például a CoAP üzenet továbbítására, a felcsatolt node újraindítására, vagy a felcsatolt node egy hálózati attribútumának lekérdezésére irányuló tranzakciók. A fejléc az üzenetkeret teljes hosszát hordozó mezővel zárul. Ezután a keret átviendő adattartalma következik, amelynek hosszát, illetve felépítését ugyancsak a THCI kézikönyv definiálja, ez az üzenettípustól függően változó. A keret egy hibaészlelést segítő ellenőrző összeggel zárul.

3. Specifikáció

3.1. Hardverkomponensek

A feladat specifikációja a Rigado által gyártott R41Z típusú rádiós mikrokontroller modult [7] jelölte ki a projekt során alkalmazandó célhardverként. A modul az NXP KW41Z típusú termékére [8] épül, amely a Thread fizikai rétegének megfelelő rádiós meghajtót integrálja egy Cortex-M0 típusú mikrokontrellerrel, órajelgenerátorral, illetve a beágyazott rendszerekben szokásos kommunikációs eszközökkel, és egyéb perifériákkal. A Rigado termék ezt egészíti ki a rádiós kommunikációhoz szükséges antennával és tápegységgel, így egy teljes értékű rádiós megoldást kínál, amely lehetővé teszi a gyorsabb piacra kerülési időt, csökkentett fejlesztési költségek mellett. A gyártó az eszközt specifikusan a Thread alapú rendszerek implementálására ajánlja, és a hozzá elérhető fejlesztői eszközök, elsősorban az NXP által a mikrokontrollerhez nyújtott fejlesztői környezet, és a modulhoz tartozó firmware példák a fejlesztést nagyban elősegítik.

A fejlesztés során a vizsgált modellhálózat a Rigado említett hálózati eszközzel szerelt próbapaneléből, illetve egyedi gyártású node-okból állt. A Rigado próbapanele a fejlesztett rendszerben a később tárgyalt menedzser node funkciót látja el, az egyedi gyártású node-okra pedig a későbbiekben távoli node-ként fogok hivatkozni. Ez utóbbin a hálózathoz való csatlakozás eszközén túl a világításvezérlő projekt által támasztott funkcionális elvárások hardver elemei is megvalósításra kerültek, így az elkészült modellhálózat segítségével a világításvezérlő rendszer működése bemutatható.

3.2. Felhasználói felület

A specifikáció a rendszer vezérlésének felhasználói felületeként egy weblapot határozott meg, melynek elemeit, tehát a vezérlő rendszer felhasználó számára elérhető funkcióit rögzítette. A weblapon a hálózat elérhető komponensei (ebben a kontextusban lámpák) listázva vannak, és egy adott komponens kiválasztása után a felhasználó projektspecifikus tranzakciót kezdeményezhet (pl. fényerő beállítása, illetve lekérdezése). Jelen dolgozat azon hálózatvezérlő rendszer fejlesztésére terjed ki, amely az így, a weblapon kezdeményezett tranzakciót véghez viszi a hálózaton, a felhasználó és a hálózatot felállító node-ok közötti kommunikációt lehetővé teszi. A vezérlő rendszert futtató beágyazott számítógépnek webszerver komponenst is futtatnia kell, amely hagyományos HTTP kapcsolat útján elérhető, és az említett weblapot kiszolgálja.

3.3. A fejlesztés munkakörnyezete

A hálózatot vezérlő hardver elem egy Raspberry PI beágyazott számítógép, amelyen hivatalosan a Raspbian operációs rendszer futtatása ajánlott, ez a Debian Linux disztribúció egy változata. A vezérlő rendszer elemei tehát Linux operációs rendszer fölött futtatható szoftverkomponensek kell, hogy legyenek. Ugyan a kijelölt hardver a Raspberry PI, a rendszer fejlesztése egy hagyományos, Linux operációs rendszert futtató PC-n zajlott. A vezérlő rendszer fejlesztése során, a későbbi, Raspberry rendszerre való átültetés miatt azonban nem használtam olyan komponenseket amelyek a Raspbian rendszerre nem elérhetőek.

A kijelölt rádiós modul gyártója számos erőforrást rendelkezésre bocsát a Thread protokollra épülő fejlesztés elősegítésére. Ezeket, a gyártó által adott szoftverfejlesztői környezetet (integrated development environment, IDE), a rádiós modulokhoz elérhető firmware példaprojekteket, illetve a THCI által definiált vezérlést elősegítő szoftvercsomagot a fejlesztés során kiindulási alapként használtam fel. Munkám során ugyancsak felhasználtam a felhasználói felületet megvalósító weblapot, amely a feladat specifikációjával együtt adott volt. A dolgozatban tárgyalt, de itt nem felsorolt szoftverkomponensek az ezekre az erőforrásokra épülő saját munkámat képezik.

4. Rendszerterv

4.1. Menedzser node

A menedzser node a Thread hálózat szemszögéből egy közönséges node, a hálózatvezérlés szempontjából azonban különleges fontosságú. A vezérlőrendszer ezen a node-on keresztül indíthat tranzakciókat a hálózaton.

4.1.1. Firmware mintaprojektek

Az MCUXpresso IDE az NXP Eclipse alapú fejlesztőkörnyezete, amelyben a célhardverhez tartozó firmware mintakódok elérhetőek. Ezek a példaprojektek lefedik a Thread hálózatba kapcsolt beágyazott rendszerek több alapesetét, így megtalálhatóak a hálózatban routerként működő, illetve a soros porton keresztül vezérelhető node mintakódjai. Ez utóbbi a fent tárgyalt Thread vezérlő interfész által definiált üzenetkereteket fogad soros porton, így a rajta futó Thread hálózati komponens PC-ről való megfigyelésének és vezérlésének lehetőségét nyújtja. Az ilyen firmware verziót futtató eszköz (a gyártó megnevezése szerint host-controlled device) a Thread hálózat felől nézve egy közönséges node. Az itt leírt rendszer szempontjából azonban különleges fontosságú, úgynevezett menedzser node. Ez a hálózati szereplő a hálózatot létrehozó node, a felhasználó ezen keresztül tud kommunikációt kezdeményezni a hálózaton, illetve engedélyezni a hálózathoz való csatlakozást a távoli node-ok számára.

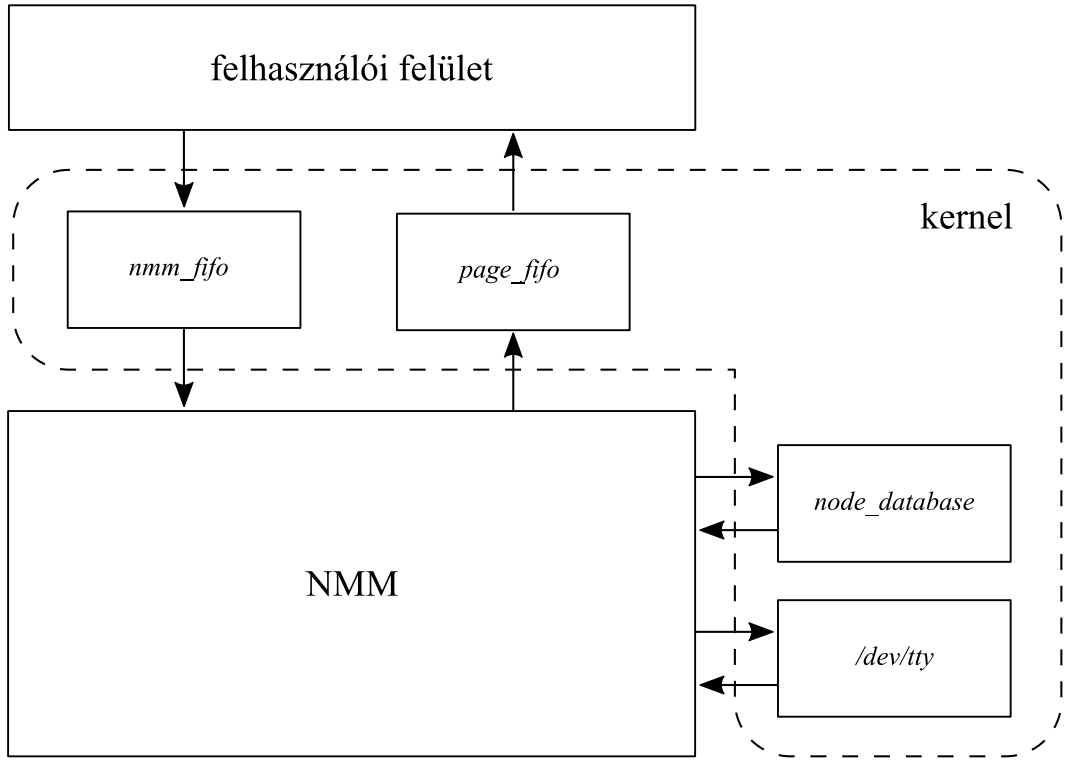
4.1.2. A menedzser node vezérlési felülete

A Kinetis Host-SDK (továbbiakban HSDK), egy PC-n, vagy egyéb, nagyobb teljesítményű gazdarendszeren futtatható fejlesztői csomag, amely az előbb említett, soros porton keresztül történő vezérlés felhasználói oldalát implementálja. Csatolva hozzá megtalálható a Thread vezérlő interfész dokumentációja, amelyben a vezérlő üzenetkeretek pontos felépítése olvasható.

A HSDK alapja egy C nyelven írt függvénykönyvtár, amely a felhasználó számára elfedi a közvetlen soros porti kommunikációt. A könyvtár modulokat szolgáltat az operációs rendszer által regisztrált soros porti eszközök megnyitására, majd a megfelelően összeállított THCI üzenetkeretek továbbítására, illetve a menedzser node felől érkező keretek fogadására, és feldolgozására. Ez a könyvtár egy Python nyelven írt, objektumorientált réteggel egészül ki. Ebben a THCI üzenetkeretek definiálásra kerültek objektumokként, így a keretek összeállítása egy jól struktúrált módon hajtható végre.

4.2. Hálózatvezérlő modul (NMM)

A hálózatvezérlő modul a Thread hálózat vezérlését megvalósító szoftverkomponens, amely a felhasználói felület felől érkező parancsokat átveszi, és a menedzser node-on keresztül a szükséges hálózati tranzakciót lebonyolítja. A hálózatvezérlő rendszer szoftveres felépítése a 4. ábra szerinti.



4. ábra: A hálózatvezérlő beágyazott számítógépen futó szoftverarchitektúra vázlata. A *node_database*-ként jelölt fájl az adatbázis nem felejtő memóriában történő tárolására szolgáló szöveges állomány, a */dev/tty* elérési úton pedig a menedzser node-hoz tartozó soros porti eszköz érhető el.

4.2.1. A hálózatvezérlés szoftveres kontextusa

A Python a webes szolgáltatások egyik hatékony háttérszkript-nyelve, illetve a HSDK fejlesztői a csomag Python interfészének használatát javasolják, az előre definiált üzenetkeretek miatt. Ezek miatt kézenfekvő megoldásnak tűnt a menedzser node-dal, illetve azon keresztül a távoli node-okkal való kommunikációt közvetlenül a felhasználói felületként szolgáló weblap háttérszkriptjéből végezni. Ebben az esetben a weblapon kezdeményezett tranzakció nyomán egy, a HSDK Python rétegével felépített szkript fut le. Ez továbbítja a megfelelő keretet a menedzser node felé, majd visszatér a weblaphoz az általa szolgáltatott válaszkeret tartalmával.

A megoldás fő hátulütője, hogy a Python-szkript kontextusa annak visszatérésekor elvész, így a rendszer nem tárol információt a hálózat és komponensei állapotáról. A specifikált feladat nem csupán a kommunikáció kezelését, de a hálózat monitorozását is magában foglalja. Ezért egy háttérben folytonosan futó szoftverkomponens fejlesztése mellett döntöttem, amely valamilyen folyamatok közti kommunikációs módszerrel átveszi a parancsokat a weblaptól, a HSDK nyújtotta szolgáltatások segítségével kommunikál a Thread hálózat szereplőivel, illetve adatot gyűjt és szolgáltat a hálózat állapotáról. Ez a háttérben futó szoftver valósítja meg a feladat által definiált hálózatvezérlési funkciókat, így ezt nevezem a továbbiakban NMM-nek (network management module, azaz hálózatvezérlő modul).

Az NMM fejlesztését C/C++ nyelven, Qt fejlesztői környezet segítségével végeztem. Ez a fejlesztői környezet a C/C++ nyelv számos, a fejlesztést könnyítő kiegészítését tartalmazza, azonban a fejlesztés során csak olyan, sztenderd nyelvi eszközöket használtam amelyeket a Raspbian rendszeren is elérhető GCC fordító csomag is ismer. A programozási nyelv kiválasztásában döntő szerepet játszott, hogy a C az egyik legelterjedtebb rendszerimplementációs programozási nyelv, és így a fejlesztés során alkalmazott kernelközeli mechanizmusokhoz kiváló dokumentáció érhető el a Linux beépítette kézikönyve formájában. Emellett nem utolsó szempont volt, hogy minden előzetes programozási tapasztalatom a C nyelvhez köthető.

4.2.2. NMM és weblap közötti kommunikáció

A felhasználói felület és a háttérben futó vezérlő szoftver közötti kommunikáció módszeréül (interprocess communication, röviden IPC) a Linux kernel karakter szervezésű FIFO implementációját választottam, a használatának egyszerűsége miatt. A vezérlő szoftver indulása után két FIFO-t hoz létre, olyan jogosultságokkal, hogy a weblapot kiszolgáló webszerver által az egyik írható, a másik pedig olvasható legyen. A felhasználó által a weblapon kiválasztott funkció alapján a weblap háttérszkriptje összeállítja a tranzakciót leíró karaktersorozatot, majd beleírja azt a kimenő FIFO-ba. A szkript ezután a bejövő FIFO-ból kiolvassa a kimenő üzenetre érkező választ. A kimenőt ezentúl nmm_fifónak, a bejövőt page_fifónak hívom, a FIFO-k címzett oldala után. Az NMM beolvassa a felhasználói felület felől érkező parancsot, végrehajtja a tranzakciót, majd annak eredményéről választ szolgáltat.

A rendszer jelen implementációjában a weblap háttérszkriptje a visszatérő válaszra blokkolva vár. Ezért, a FIFO működési elvének megfelelően, a page_fifóba írt válaszok sorrendje az nmm_fifóba kerülő kimenő üzeneteknek megfelelő kell, hogy legyen. Így az NMM működése soros, azaz a következő parancsot csak akkor olvassa be az nmm_fifóból, ha az azt megelőző tranzakció már végbement, és az ahhoz tartozó választ

továbbította a weblap felé. A kezelői felület felől érkező parancsok feldolgozásának párhuzamossá tételére a rendszer későbbi továbbfejlesztése során lehetőség nyílik, de jelen feladat megoldásakor nem láttam szükségesnek, mivel ez a rendszer alapfunkcióinak megvalósítására irányul.

4.2.3. Node nyilvántartás

Mivel a feladat által leírt hálózat helyükben, és funkciójukban rögzített node-okból áll, azok nyilvántartása, elérhetőségük, állapotuk követése, és ezen adatokból egy központi adatbázis fenntartása triviális elvárás a rendszerrel szemben. Ezen adatbázisban nyilvántartásra kerül a hálózatot felépítő összes node, azok IPv6 címeivel. Azonban ha egy node valamilyen okból – például karbantartás esetén – a hálózatról lecsatlakozik, újracsatlakozásakor új IP-címet kap. Ezért, ugyan az IP-cím szükséges a node eléréséhez, nem azonosítja azt. A node-okhoz tartozó adatbázis bejegyzések elsődleges kulcsa lehet a rajtuk szerelt rádiós modul gyári sorozatszáma, amely egy másik, a projekt által definiált funkcionalitás miatt a node firmware-ében rendelkezésre áll.

Egy távoli node esetleges újracsatlakozása esetén, annak új IP-címét a vezérlő rendszernek adatbázisában regisztrálnia kell. Ezért a node-ok szoftverét ki kell egészíteni a csatlakozás utáni bejelentkezés végrehajtásával. Ekkor a node, sikeres csatlakozása után kommunikációt kezdeményez a menedzser node-dal, amelyben közli vele új IP-címét, és gyári sorozatszámát.

4.2.4. A távoli node-ok elérhetőségének vizsgálata

A specifikált feladat része a hálózaton elérhető távoli node-ok felhasználói felületen való megjelenítése. Mivel a node-ok csatlakozás, és az azt követő bejelentkezés után számos hibaokból elérhetetlenné válhatnak (például áramkimaradás, vagy egyéb hardveres probléma miatt), a rendszerben regisztrált távoli node-ok felhasználói felületen való felsorolása nem kielégítő. A távoli node-ok elérhetőségét ezért periodikusan ismételve vizsgálni kell. A vizsgálat során a menedzser node a regisztrált távoli node-ok IP-címére ping üzenetet küld. Ha a kérésre megadott időn belül nem érkezik válasz, a rendszer az adott node-ot elérhetetlennek minősíti. A node-ok elérhetőségét az adatbázisban rögzíti, amely adatbázis teljes tartalmát továbbítja a felhasználói felületről érkező lekérés esetén.

A rendszer által fenntartott adatbázisnak tehát tartalmaznia szükséges a regisztrált node-hoz tartozó gyári sorozatszámot, aktuális IPv6 címet, illetve a node elérhetőségi állapotát. Ugyan a hálózatvezérlő szoftver a terv szerint a hálózat teljes életciklusa alatt futó állapotban van, a rendszer esetleges leállására, és újraindulására készülni kell. A rendszer ezért az adatbázist fájlba menti, és induláskor a fájlból olvassa be. Induláskor a rendszer az összes távoli node-ot elérhetetlennek tekinti, és elérhetőségi állapotukat a periodikusan

végzett vizsgálat során frissíti. Ezért az adatbázis mentett változatában nem szükséges a node-ok elérhetőségét rögzíteni, a fájlt csak új sorozatszám, vagy meglévő sorozatszámhoz új IP-cím regisztrálása esetén kell újra kiírni.

5. Hálózatvezérlő szoftver fejlesztése

5.1. Szoftver szálak és a közöttük zajló kommunikáció

5.1.1. Kommunikáció a felhasználói felület és a vezérlő modul között

A rendszer felhasználói felülete és az NMM konkurrens szoftver folyamatok, a közöttük zajló adatátvitel a Linux kernel rendszerhívásain alapul. A kommunikáció eszköze a Linux FIFO [9] implementációja, amely egy, a fájlrendszer részeként elérhető pipe. A fájlrendszerben való bejegyzés a FIFO névvel való azonosítását szolgálja, így a név, azaz az elérési út ismeretében több párhuzamos folyamat is megnyithatja azt.

Az NMM az mkfifo() rendszerhívás segítségével hozza létre a két FIFO-t. Ezek a fájlrendszer részeként, fájlokként elérhetőek, így a program az open() rendszerhívással nyitja meg őket. Az open() függvény visszatérési értéke a kerneltől kapott, fájlhoz tartozó leíró (pozitív egész szám), vagy hiba esetén -1. Ezen a megnyitás során visszaadott fájl leírón keresztül a FIFO-t azonosító fájlba írt karakterek a read() rendszerhívás [10] segítségével olvashatók ki.

Mivel a felhasználói felület és az NMM folyamatai függetlenek, az NMM számára nem ismert, hogy az nmm_fifóba mikor kerül a kiolvasandó karaktersorozat. Ezt a szinkronizációs problémát a blokkoló olvasással át lehet hidalni. Ekkor a fájl megnyitása során az open() függvény egyik argumentumában a program jelzi a kernel felé, hogy a fájlon blokkoló műveletet kíván végrehajtani. Az ezután indított read() rendszerhívások mindaddig nem térnek vissza, amíg az olvasandó fájlban nincs olvasni való tartalom.

5.1.2. A FIFO-k tartalmának vizsgálata

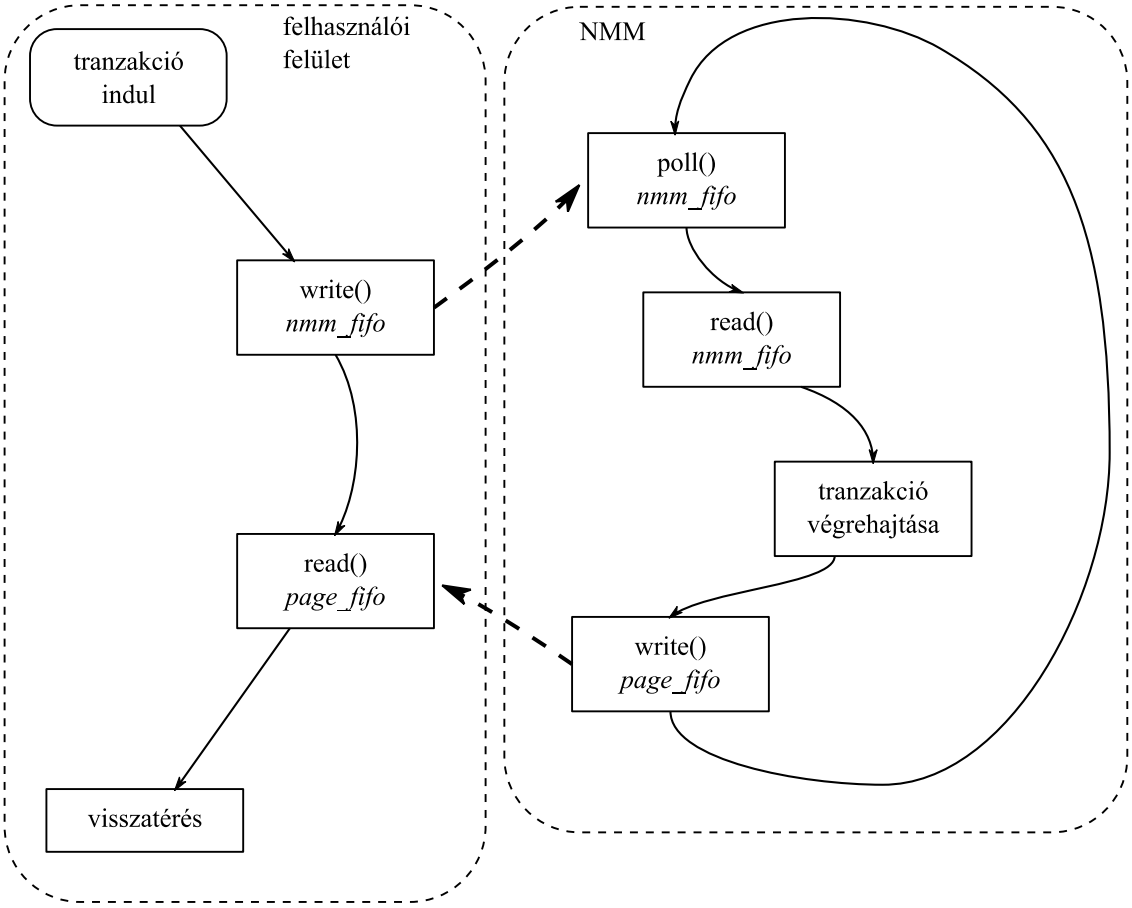
A FIFO impelementáció sajátossága, hogy a blokkoló módban, olvasásra történő megnyitása csak akkor tér vissza sikeresen, ha azt egy másik folyamat írásra már megnyitotta (azaz a pipe írási oldala is nyitva van). Az nmm_fifót a vezérlő rendszer annak inicializációs fázisában, a kezelői felület működésétől függetlenül nyitja meg, és ez a megnyitás a többi inicializációs lépést nem késleltetheti, ezért az NMM az nmm_fifót nem blokkoló üzemmódban nyitja meg. FIFO esetén ez a megnyitás akkor is végbemegy, ha a pipe írási oldalát még egy folyamat sem nyitotta meg.

Ilyen üzemmódban az üres nmm_fifóra hívott read() rendszerhívás azonnal visszatér, méghozzá egy EOF karakterrel. A FIFO-ba írt tartalom vizsgálata így busy wait, azaz foglalva várakozás állapotához vezet. A felesleges műveletek végzésének elkerülése érdekében a poll() rendszerhívást [11] alkalmaztam az nmm_fifo tartalmának vizsgálatára. Ennek használata során a program egy fájl leírókból, és ki- és bemeneti műveleteket azonosító flagekből összeállított struktúrával jelzi a kernel felé, hogy mely fájlokon

kíván, és milyen műveleteket véghez vinni. A poll() hívást intéző folyamatot a kernel várakozó állapotba teszi. Amikor a vizsgált fájlok valamelyikén elvégezhetővé válik a specifikált művelet, a kernel a hívó folyamatot újra futásra ütemezi, és a poll() visszatérési struktúrájában jelzi, hogy melyik fájlokon és milyen műveletet lehet végezni.

Az NMM inicializációs fázisában tehát létrehozza a két FIFO-t, az nmm_fifót megnyitja. A parancsok feldolgozása egy végtelen ciklusban történik. A ciklus törzsében a poll() rendszerhívással vizsgálja a FIFO olvashatóságát. Mikor az visszatér – a hibát jelző flagek ellenőrzése után – a read() hívással kiolvassa az nmm_fifo tartalmát. A kiolvasott parancsot feldolgozza, az annak megfelelő hálózati tranzakciót lebonyolítja a menedzser node-on keresztül. A tranzakció végbemenetelét követően az NMM megnyitja a page_fifót, beleírja a menedzser node-tól kapott választ, majd bezárja a pipe-ot. Ezután a ciklus újra kezdődik, így a program fő szálja újra várakozó állapotba megy, amíg az nmm_fifo olvashatóvá nem válik. Az itt leírt működés vázlata a 5. ábrán látható.

A kezelői felületként szolgáló weblapon tanzakció kezdeményezése során a háttérszkript írásra megnyitja az nmm_fifót, majd beleírja a tranzakciót leíró karaktersorozatot. Az nmm_fifót ezután bezárja, megnyitja a page_fifót olvasásra, és blokkoló olvasást végez, azaz várakozik amíg abban a válasz elérhetővé válik. Ennek teljesülésekor kiolvassa a FIFO tartalmát, bezárja azt és visszaadja a weblap felületének, így a tranzakció eredménye a felhasználó számára olvashatóvá válik.



5. ábra: A felhasználói felület és az NMM független kontextusai közötti kommunikáció vázlata

Az nmm_fifo nem blokkoló módban, olvasásra történő megnyitása sikerrel jár akkor is, ha a FIFO írási oldalát nem tartja nyitva egyik folyamat sem, és a poll() rendszerhívás is a fent leírt működést valósítja meg. Azonban, ha egy folyamat ezután megnyitja a FIFO-t írásra, majd végül be is zárja azt, a kernel a következő poll() hívás esetén a vizsgált fájl leíróra a HUP (hang-up, vonal elengedve) állapotot fogja jelezni, vagyis hibával tér vissza. A kívánt működés elérése érdekében ezért az NMM az nmm_fifót írás és olvasás műveletekre nyitja meg. Ekkor, függetlenül attól, hogy a weblap háttérszkriptje bezárja a pipe-ot, azt írásra nyitva tartja egy folyamat (az NMM maga), így a kernel a HUP feltétel teljesülését nem állapítja meg. Ez a kerülő megoldás történetesen feleslegessé teszi a poll() mechanizmus használatát, mert a FIFO írásra és olvasásra történő megnyitása mind blokkoló, mind nem blokkoló módban sikerrel jár, így azon lehetne blokkoló olvasást végezni. Ez a következetlenség a fejlesztés során elkerülte a figyelmemet, a rendszer későbbi optimalizálása során, mint felesleges komplexitás feloldható lesz. Az egyszerűbb, következetesebb megoldás szerint az NMM blokkoló módban nyitja meg az nmm_pipe-ot, majd a folyamat szükséges ütemezését a read() rendszerhívással történő blokkoló olvasás valósítja meg.

5.1.3. Kommunikáció a vezérlő modul és a menedzser node között

A Kinetis HSDK egy host-controlled firmware verziót futtató Thread hálózati node és egy vezérlő PC közötti kapcsolódási felület felállítását teszi lehetővé. Jelen projekt esetében ez a PC-hez csatolt node a hálózatvezérlés szempontjából kiemelt fontosságú menedzser node. A kommunikáció soros porti kapcsolat révén zajlik, előre definiált üzenetkeretekkel. A kereteket, illetve az elvárt működést a Thread vezérlő interfész (THCI) definiálja.

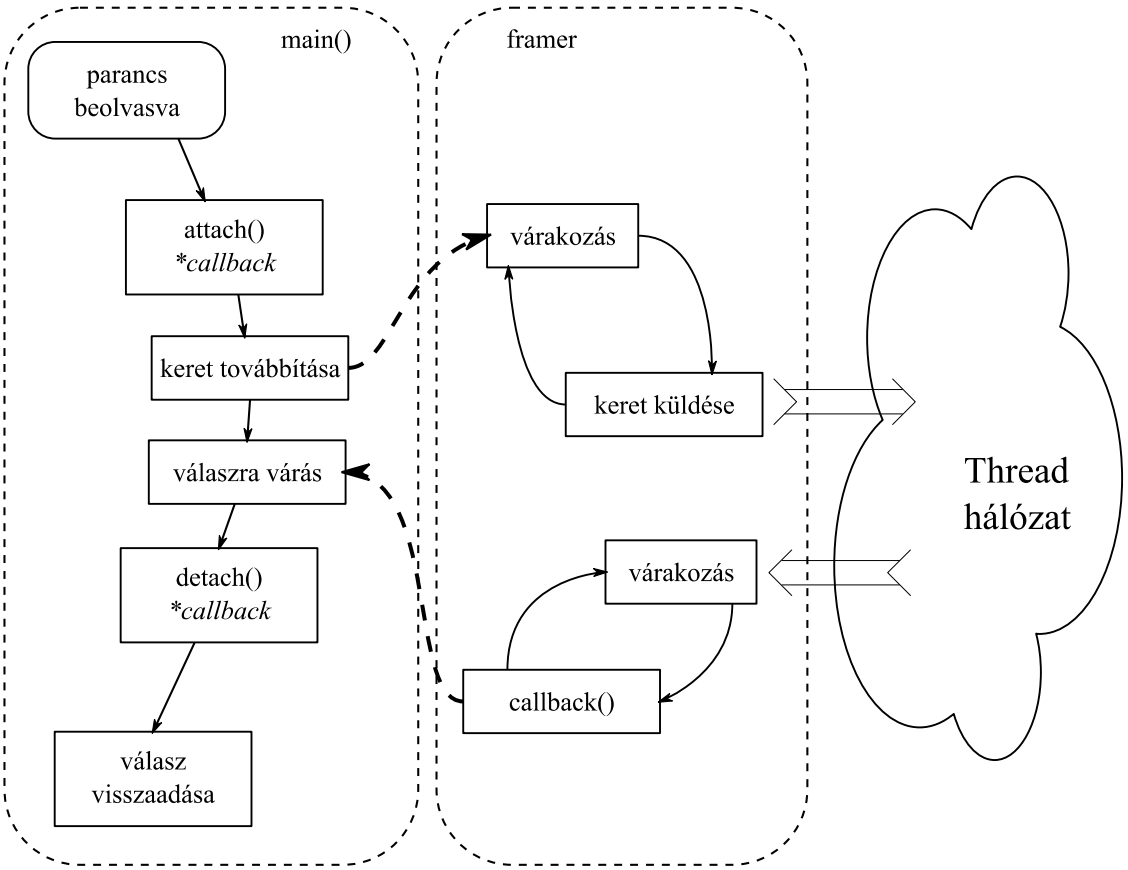
A HSDK egy szoftveres fejlesztői csomag, amely a felcsatolt node vezérlési felületének gazdarendszeren való fejlesztését teszi lehetővé. Egy olyan függvénykönyvtár, amely a szükséges soros porti kommunikáció hardver-, illetve kernelközeli rétegét valósítja meg. A HSDK komponenseinek használata során a gazdagép operációs rendszere által regisztrált soros porti eszközöket észleli, megnyitásuk esetén konfigurálja, és a megnyitott eszközön zajló kommunikáció végrehajtására egy új szoftver szálat indít el, amely szálban elvégzi a soros porton, független, aszinkron módon elküldeni kívánt, illetve beérkező üzenetkeretek sorba állítását, szinkronizálását.

A kimenő üzenetkeretek kezelése egy úgynevezett framer szál segítségével történik. Az összeállított kereteket a framer továbbítja a hardvert vezérlő szál felé, illetve a beérkező üzenetkereteket is a framer szál dolgozza fel. A HSDK-ra épülő szoftver fejlesztője saját callback függvényeket definiálhat, amelyeket a framer szál beérkező keret érkezésekor meghív, és a beérkező keret adattartalmát feldolgozásra átadja. A HSDK-val épített program futási idejében a framerhez tetszőleges számú callback függvényt adhat (attach) hozzá, illetve csatolhat le (detach), ahogyan a kívánt kommunikációhoz szükséges.

5.1.4. Callback függvények a válaszkeretek feldolgozására

Az NMM fejlesztése során, a fejlesztés követhetősége, a kódban történő navigálás könnyítése érdekében minden indított tranzakcióhoz külön callback függvényt definiáltam. Az NMM-ben tehát a kívánt hálózati tranzakció lebonyolítása a következő lépésekből áll. A felhasználói felület felől érkező parancsot a program főciklusából hívott függvény dolgozza fel. A függvény a kimenő keretet összeállítja, majd a framerhez a keret típusának megfelelő callback függvényt hozzáadja. Ezután a keretet a frameren keresztül továbbítja, és a válaszra blokkolva vár. A soros porton beérkező válasz, vagy jelzés keretet az eszközkezelő szál a framernek átadja, ezután a framer szálban az összes, aktuálisan regisztrált callback függvény egymás után meghívásra kerül. Az adott üzenet feldolgozása érdekében hozzáadott callback függvény a fő programciklus száljának egy IPC módszer segítségével jelzi a várt üzenet beérkezését, illetve szükség esetén továbbítja a beérkező üzenet adattartalmát. A fő ciklusban blokkolva várakozó függvény ekkor az

üzenettípushoz tartozó callback függvényt lecsatlakoztatja a framerről, majd visszatér, a tranzakció eredményét pedig az NMM továbbítja a felhasználói felület felé. Az itt leírt működés vázlata az 6. ábrán látható.



6. ábra: Az NMM fő szálja (main()) és framer szálja közötti kommunikáció vázlata

Mivel egy üzenetkeret beérkezése esetén az összes regisztrált callback meghívódik, azok első lépésként a beérkező üzenet típuskódját vizsgálják. Ez alapján eldől, hogy az adott keret feldolgozása melyik aktuálisan regisztrált függvény feladata. A kezdeményezett tranzakciókra érkező válaszok a vezérlő rendszer működéséhez teljesen szinkronizálatlanul érkeznek, illetve előfordulhat, hogy egy üzenet beérkezésének pillanatában több callback függvény is regisztrálva van. Ebben az esetben a válaszra váró főciklusbeli függvény számára explicit módon ismert kell, hogy legyen, hogy a hozzá eljutó beérkező adattartalom melyik callback függvényen keresztül érkezik. Ezért a beérkező keret feldolgozása után a framerhez tartozó szál, és a program főciklusának szálja közötti adatátvitelnek nemcsak a keret adattartalmának átadására, hanem az üzenetkeret azonosítására (típuskódjának azonosítására) is ki kell terjednie.

A fent leírt szinkronizációs probléma megoldására a POSIX által definiált, úgynevezett feltételváltozó (condition variable) mechanizmust [12] választottam. Az ezáltal elérhető

szinkronizációs struktúrának három eleme van: egy (egész) változó, amelyben az üzenetkeretek típuskódjának átadása történik, egy mutex, amely ezt a változót védi, valamint egy speciális, feltételváltozó típusú változó. A mechanizmus a feltételváltozó által jelzett feltételre való várakozást teszi lehetővé, illetve a feltételváltozón keresztül a feltétel teljesülésének jelzését. A feltételre váráshoz, és a jelzéshez a feltételváltozó, és a mutex együttes használata szükséges.

A kérés és válasz üzenetek szinkronizációjának lépései a következők. Az NMM főciklusában a kimenő üzenetkeretet összeállítja, majd annak a framerhez való továbbítása előtt lefoglalja a mutexet, majd a keret továbbítása után a feltételváltozó és a mutex megjelölésével feltételre várakozó állapotba megy, egy rendszerhívás segítségével. A beérkező válaszkeretet feldolgozó callback ezután lefoglalja ugyanazt a mutexet, az üzenet típuskódját hordozó változót beállítja, majd jelzi a kernel felé a feltétel teljesülését, a feltételváltozó és a mutex megjelölésével. A callback ezután elengedi a mutexet, ami után a kernel az NMM főciklusának szálját újra futásra ütemezheti. Ekkor a mutex újra a főciklus kontextusának birtokában van, ott a típuskódot tartalmazó változó kiértékelhető, a kölcsönös kizárási feltételek teljesülése mellett.

Ha a főciklus kontextusa felé a kernel jelezte a feltétel teljesülését, és a típuskódot vizsgálva egyezés állapítható meg, akkor biztos, hogy a menedzser node felől érkező visszatérő üzenet a végrehajtani kívánt tranzakció válaszkeretét tartalmazza. Az NMM jelen állapotában nem fordul elő, hogy több, független tranzakcióhoz tartozó callback függvény is regisztrálva legyen, így a feltételváltozós mechanizmus egy felesleges komplexitást jelent. A szoftver későbbi optimalizálásakor ez szükséges esetben feloldható, viszont a fejlesztés során szándékosan meghagyni kívántam a beérkező parancsok párhuzamosításának lehetőségét.

5.1.5. Kölcsönös kizárás biztosítása a HSDK-ban

A HSDK implementációjában a framer szálhoz regisztrált callback függvényekre mutató pointerek tárolása egy láncolt lista jellegű struktúrában történik. Új tranzakció indításakor a callback láncolt listára való felfűzése a főciklus száljában történik, ahogyan a visszatérést megelőzően a callback lecsatolása is. Beérkező keret feldolgozásakor a láncolt listában tárolt pointerek által kijelölt függvényeket a framer szál saját kontextusában, egy ciklusban hívja végig. A függvények regisztrálása és meghívása tehát egymáshoz szinkronizálatlanul megy végbe, ami potenciálisan kölcsönös kizárási problémához vezethet.

A fejlesztői csomag forráskódja nyílt, így a láncolt lista struktúrán, illetve az azon végzett műveletekben a kölcsönös kizárás feltételeit egyszerűen biztosíthattam, úgy, hogy a kódot

a mutex mechanizmusával egészítettem ki, majd a HSDK könyvtárait újrafordítottam. A láncolt listát leíró típusdefiníciót egy mutex típusú taggal egészítettem ki, amelyet a láncolt listát létrehozó függvény kiegészítéseként inicializáltam. Az ezen a láncolt listán bármilyen műveletet végző függvények mindegyikét kiegészítettem a mutex érdemi működés előtti lefoglalásával, majd az utána következő felszabadításával.

Ezután, ha a framer éppen a beérkező üzenet regisztrált callbackekkel való feldolgozását végzi, és a főciklus egy függvénye új callback regisztrálását indítja meg, a regisztrálás csak akkor megy végbe, ha az éppen feldolgozandó üzenetre az összes aktuális feldolgozó függvény már meghívódott és visszatért. Ugyanígy, ha egy új callback regisztrálása közben feldolgozandó üzenet érkezik, biztosított, hogy a framer csak akkor kezd hozzá a láncolt listában rögzített függvények meghívásához, ha az új callback regisztrálása már végbement.

5.2. A hálózat komponenseinek adatbázisa

A rendszertervben megfogalmazottak alapján a hálózatvezérlő rendszernek a hálózatba kapcsolt eszközöket nyilván kell tartania. A hálózaton regisztrált node-okat a felhasználói felület számára listázza, azok címét rögzíti, valamint az elérhetőségüket periodikusan vizsgálja. Miután egy node a hálózathoz sikeresen csatlakozott, az NMM felé egy bejelentkező üzenetet küld, az új IP-címével és a sorozatszámával. Az adatbázisban az adott sorozatszámhoz tartozó bejegyzés IP-címet tartalmazó mezője ekkor frissül. Ha a bejelentkezés egy addig ismeretlen node felől érkezik, vagyis az adatbázisban nem található meg a node sorozatszáma, akkor az NMM új bejegyzést hoz létre. A megvalósított működés tehát olyan, hogy az adatbázis az új IP-címekkel, illetve az új node-okhoz tartozó bejegyzésekkel automatikusan frissül, de egy adott node törlésére a rendszer nem nyújt lehetőséget (például, ha az eszköz működésképtelenné vált, és új rádiós modul kerül a helyére). A bejegyzések adatbázisból való törlése a későbbi fejlesztési lépések során kerül implementálásra.

Az NMM az adatbázist futási idejében a C++ sztenderd könyvtárában elérhető egyik asszociatív tároló struktúrában, std::map-ben [13] rögzíti. A tároló rekeszei két elemből, az adattartalmat hordó struktúrából, és a hozzá tartozó egyedi kulcsból állnak. Az adatbázis esetén az előbbi a sorozatszámot, IP-címet és elérhetőségi állapotot tartalmazza (online mező), utóbbi a sorozatszám. A C++ map használata lehetővé teszi a bejegyzésekből ily módon felépített tömb egyedi kulcs, azaz gyári sorozatszám szerinti indexelését. Ha a program a map egy olyan elemének próbál értéket adni, amely egyedi kulcsa nem létezik, az értékadás előtt az új bejegyzés automatikusan létrehozásra kerül. A map implementáció tehát mind az IP-cím frissítését, mind az új sorozatszámhoz tartozó bejegyzés létrehozását egyszerűvé teszi.

Mivel az NMM-et futtató beágyazott számítógép leállása előfordulhat, például karbantartás esetén, ezért az adatbázis tartalmát nem felejtő memóriában menteni kell. Új IP-cím, vagy új bejegyzés felvétele után a mapben tárolt tartalmat az NMM egy szöveges állományba kiírja. A rendszer indulásakor, az adatbázist leíró map ezen fájl tartalmával kerül inicializálásra. A rendszer első induláskor az adatbázist leíró szöveges fájlt az NMM hozza létre.

5.2.1. Hálózati eszközök elérhetőségének ciklikus vizsgálata

A vezérlő rendszernek a felhasználói felület felé nemcsak a regisztrált komponensek listáját kell szolgáltatnia, hanem a node-ok elérhetőségi állapotát is, ez a hálózat karbantartói számára fontos információt hordoz. Az NMM-nek ezért minden regisztrált node elérhetőségét periodikusan ismételve vizsgálnia kell.

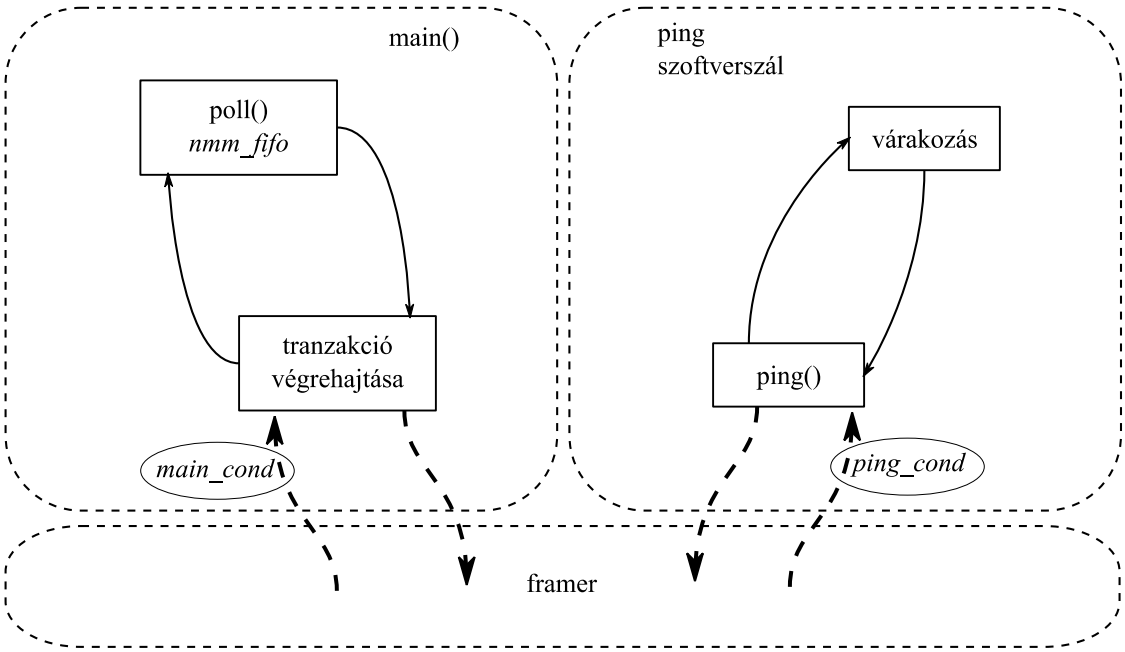
A vizsgálat módja a mapben tárolt IP-címekre ping üzenet küldésén alapul, amely üzenetre ha előre specifikált időtartamon belül válasz érkezik az adott címről, a node elérhető állapotúnak minősül. Ekkor az NMM a node-hoz tartozó bejegyzés online mezejét igaz értékűre, ellenkező esetben hamisra állítja be.

A ping funkcionalitást a THCI definiálja, vagyis a menedzser node felé a ping típusú üzenetkeretet továbbítva, az a hálózati komponens elérhetőségének vizsgálatát végre-hajtja. A kérés kerete tartalmazza a vizsgálni kívánt, illetve a küldő node IP-címét, valamint az időtúllépés (timeout) időtartamát ezredmásodpercben. A menedzser node kezdeményezi a szükséges kommunikációt a hálózaton, és legkésőbb timeout idő elteltével a válaszkeretben visszaadja, hogy a megcímzett hálózati komponens válaszolt-e, illetve azt, hogy mekkora késleltetéssel.

Az NMM indulása során, a FIFO-k megnyitása, az adatbázist leíró fájl beolvasása, és a HSDK eszközeinek inicializálása után elindít egy új szoftverszálat. A hálózaton elérhető eszközök vizsgálatát ebből a szálból kezdeményezi. A szál indulása után lekérdezi a menedzser node saját IP-címét, amelynek ismerete a pingelés indításához szükséges, majd végtelen ciklusban végzi a node-ok elérhetőségének vizsgálatát. Ez a vizsgálat a következő lépések szerint zajlik. Az adatbázisból kiolvassa egy node regisztrált IP-címét, majd ping tranzakciót indít a címre. A tranzakció legfeljebb timeout idő elteltével befejeződik, ekkor annak eredménye szerint beállítja az adott adatbázis-bejegyzés online mezejét. Ezután felolvassa az adatbázisban következő bejegyzéshez tartozó IP-t, és így az adatbázis összes bejegyzésén, egy számlálós ciklus által szervezve végiglépdel. A ciklusváltozó az adatbázis map struktúrájának egy iterator objektuma.

Az NMM fő szálja és a pingelésért felelős szál a framernek egymástól függetlenül, szinkronizáció nélkül továbbítja az üzenetkereteket, ahogyan azt a 7. ábra mutatja. A

HSDK implementációja a soros porti eszköz kezeléséért felelős szálban a kimenő keretek hozzáadását kritikus szakaszként kezeli, azon a kölcsönös kizárás feltételeit biztosítja, így a framer párhuzamos megszólítása megengedett. Mind a pingelés, mind a fő szálból indított tranzakciók esetén a kezdeményező szálak a korábban bemutatott feltételváltozós mechanizmusnak megfelelően várnak a menedzser node-tól érkező válaszkeretre. Azonban a fő szálból indított kérésekhez, és a pingelési szálhoz külön feltételváltozó tartozik, a visszatérő kereteket feldolgozó callback függvények különböző feltétel teljesülését jelzik vissza. Ezáltal biztosított a két szálból indított tranzakciókhoz tartozó válaszkeretek elkülönítése.



7. ábra: A fő szál és a pingelésért felelős szál a framert egymástól függetlenül szólítják meg. A *main_cond* a fő szálhoz tartozó feltételváltozó, *ping_cond* a pingeléshez tartozó.

Ha a pingelés az adatbázis minden bejegyzésén megtörtént, a szál külső ciklusa végére ér, majd az újrakezdés előtt egy előre meghatározott időpontig várakozhat. Ennek a késleltetésnek a hálózaton egységnyi idő alatt végbemenő tranzakciók mennyiségi mérséklése a célja, ami a hálózati forgalom aktuális szintjétől függően előnyös lehet. A hálózati komponensek elérhetőségi állapotában beálló változás tehát a felhasználói felületen nem jut azonnal érvényre. Ha a rendszer továbbfejlesztése során erre a működésre nézve rövidebb válaszidő igénye fogalmazódna meg, úgy a fent leírt késleltetés mértékét csökkentése, illetve a node-ok vizsgálatának párhuzamosítása szükséges.

5.2.2. Adatbázis frissítése a felhasználói felületen

A weblap felületén a regisztrált hálózati komponensek listája felhasználói kérésre (gombnyomásra) frissül. Ekkor a weblap háttérszkriptje a discover parancsot írja az nmm_fifóba. Ennek feldolgozásakor az NMM az adatbázist tároló map minden elemén végig iterál, annak teljes tartalmát egy karaktersorozattá összefűzi. Az így kapott sztringet adja vissza a felhasználói felületnek a page_fifón keresztül.

A visszaadott karaktersorozat formája JSON (JavaScript Object Notation) szabványnak megfelelő. Az ilyen szöveges állomány a tárolt mezőkhöz asszociatívan címkéket társít, amely címkék szerint a weblap JSON értelmezője a weblap objektumait a mezőkhöz társítja. A felhasználói felület a karaktersorozatként visszaadott adatbázist ezen mechanizmus szerint jeleníti meg táblázatos formában.

Mivel az adatbázis map struktúráját mind az NMM fő szálja, mind a pingelésért felelős szál elérheti, és ezek egymáshoz semmilyen formában nem szinkronizáltak, az adatbázison a kölcsönös kizárás feltételeit biztosítani kell. Ennek érdekében az NMM egy mutexet használ az adatbázis védelmére. A fájlból való beolvasás, a fájlba való kiírás, a weblap részére történő felolvasás, és a pingelés során szükséges módosítás műveletek mindegyike először lefoglalja, majd ha a szükséges lépések végbementek, újra elengedi a mutexet.

5.3. Hálózati hibák kezelése

5.3.1. Távoli node nem válaszol

Az NMM az általa indított tranzakcióra érkező válaszra, azaz specifikusan az adott üzenettípushoz tartozó callback függvény lefutására blokkolva várakozik. Ha a tranzakció során megcímzett hálózati komponens – legyen az a menedzser node, vagy valamelyik távoli node – bármilyen hiba folytán a kérésre nem válaszol, az NMM az eddig leírtak alapján, a felhasználói felületről érkező parancsokra nézve teljesen működésképtelenné válik. Ugyanis, annak korábban tárgyalt soros működése miatt a tranzakció eredményének visszaadása előtt nem olvassa be a következő parancsot.

Ha a hálózat egy szereplője az NMM által indított kérésre nem válaszol, a jelenséget okozó hiba kijavítására a vezérlőrendszernek nincs lehetősége, hiszen az minden funkcionalitását a hálózaton továbbított üzenetek révén valósítja meg. Ezért az NMM az elmaradó válasz esetére egy időtúllépési mechanizmust alkalmaz. Ha nem érkezik válasz egy meghatározott időkereten belül, a főciklusbeli függvény visszatér, és az időtúllépés tényéről értesíti a felhasználói felületet a page_fifón keresztül. A POSIX által definiált

programozói interfész lehetővé teszi a feltételváltozóra várás időzített verzióját, amely az időtúllépéses mechanizmus implementálását jelentősen leegyszerűsíti.

5.3.2. Időtúllépési mechanizmus

A kimenő üzenet esetén az összeállított keret framernek való átadása, és a feltételre való várakozás a következő lépésekkel egészül ki. A típuskódot védő mutex lefoglalása, és a kimenő keret továbbítása után az NMM a kerneltől lekéri az aktuális rendszeridőt. Az idő lekérésére szolgáló clock_gettime() rendszerhívás visszatérésekor kapott struktúra értékét módosítja az időtúllépést jelentő időtartammal, így egy abszolút időpontot jelöl ki a jövőben. A feltételváltozóra való időzített várakozás rendszerhívása ezt, az abszolút időpontot jelölő struktúrát is fogadja argumentumaként. A hívás mind a feltétel teljesülésekor, mind a kijelölt időpont elérése után visszatér, az okot a visszatérési értékével jelzi.

A feltételre várakozás egy feltételes ciklusban történik. A ciklus feltétele teljesül, ha várakozást megvalósító rendszerhívás időtúllépés miatt tér vissza (hálózati hiba), vagy visszatérésekor az üzenet típuskódját vizsgálva egyezés állapítható meg (sikeres tranzakció). Ellenkező esetben a feltételt nem a vizsgált üzenettípushoz tartozó callback jelezte, ekkor a főciklus függvénye a kijelölt időpontig még tovább várakozhat a válaszkeret beérkezésére.

5.3.3. Menedzser node nem elérhető

Abban az esetben, ha a menedzser node működése bármilyen hardveres, vagy szoftveres problémából fakadóan akadályozott, a vezérlőrendszer semmilyen feladatát nem képes ellátni. Ugyan az NMM, az előző alfejezetben leírtakhoz hasonlóan nem képes az ilyen hálózati problémákat kijavítani, a probléma tényét szükséges a felhasználói felület felé jelezni. Ha a menedzser node nem elérhető, bármilyen hálózati tranzakció indítására tett próbálkozás nyilvánvalóan felesleges.

Ha az NMM-et futtató beágyazott számítógépet a menedzser node-dal összekötő soros porti kapcsolat megszűnik (például a menedzser node szerepét betöltő próbapanel soros porti vezérlő hardvere meghibásodása okán, vagy azért mert a hardver tápellátása megszűnik), a beágyazott számítógépen futó kernelben az eszközhöz rendelt fájl megszűnik, így a HSDK hardvert kezelő szálja a keretek továbbítását nem viheti véghez. A HSDK implementáció sajátossága, hogy a hardvert kezelő szálban a soros porthoz tartozó leírón keresztül, annak olvashatóságára a poll() rendszerhívás segítségével várakozik. Ha ez a fájl a kernelben megszűnik, a fájl leíróra meghívott poll() rendszerhívás azonnal, hibával tér vissza. A HSDK által megvalósított hardverkezelő szál ilyen esetben busy wait állapotba kerül, mert a fejlesztői ennek az, egyébként igen egzotikus hibának a

kezelését nem implementálták. Az itt leírt eseményt ezért az NMM szintjén észlelni kell, és a hardvert kezelő szálat kívülről kell leállítani.

Ha a menedzser node egy ilyen esemény után újra megfelelő tápellátást kap, vagy a rajta működő mikrokontroller bármely egyéb okból újraindul, a soros porti eszköz újra elérhetővé válik a kernel számára, így a keretek továbbítása is lehetséges lesz. Azonban a menedzser node a beérkező kereteket csak az inicializáció szükséges lépéseit követően dolgozza fel, vagyis a nem tervezett újraindulást követően a menedzser node a THCI által definiált feladatainak nem tesz eleget. Ezért az újraindulás tényét is észlelni kell, az újraindult menedzser node felé tranzakció kezdeményezésének nincs értelme.

5.3.4. A menedzser node elérhetőségi vizsgálatának lépései

A fent leírt hibajelenségek észlelése a távoli node-ok pingelését végző szálban történik. Ezen szál végtelen ciklusának elején az NMM az access() rendszerhívás [14] segítségével vizsgálja, hogy a menedzser node soros porti eszközének elérési útja egy létező fájlra mutat-e. Ha az elérés sikeres, az NMM vizsgálja, hogy a menedzser node újraindult-e, oly módon, hogy egy érvényes THCI kérést továbbít felé, és arra választ vár, a korábban bemutatott időtúllépési mechanizmus alkalmazásával. Az elküldött keret konkrét típusának nincs jelentősége, a vizsgálat módja azt a tényt használja ki, hogy a helyesen inicializált menedzser node minden érvényes THCI kérésre legalább egy visszatérő válaszkerettel válaszol. A vizsgálat során, a késleltetés minimalizálása érdekében érdemes olyan adatlekérési tranzakciót választani, amely adat minden körülmények között a menedzser node birtokában van. A konkrét megvalósítást tekintve ez a tranzakció a menedzser node aktuális időbélyegét kérdezi le. Ezek a vizsgálatok tehát minden pingelési ciklus előtt lezajlanak, az eredményük pedig tárolásra kerül.

Az NMM a felhasználói felület felől érkező parancsnak megfelelő függvény meghívása előtt a pingelésért felelős száltól lekérdezi az előbb leírt vizsgálatok eredményét. Ha azok hibát jeleznek, az NMM a tranzakciót nem kezdeményezi a menedzser node-on keresztül, a felhasználói felület felé a „menedzser node nem elérhető” üzenetet továbbítja. A rendszer jelen állapotában a hibás állapotból való visszatérést nem kísérli meg. A hibajelenségek kezelése a rendszer továbbfejlesztése során kerül implementálásra. A tervek szerint az NMM egy monitorozást végző szkript kontextusából fog elindulni. Hiba esetén az NMM a szkripthez visszatér, visszatérési értékével jelzi a hiba okát. A monitorozó szkript feladata lesz az NMM-et a szükséges feltételek teljesülésekor újraindítani (amikor a soros porti eszköz újra elérhető).

6. Távoli node firmware fejlesztése

6.1. Firmware példaprojektek kiegészítése

A rádiós modul gyártója által szolgáltatott firmware mintapéldák a FreeRTOS (Free Real-Time Operating System) beágyazott operációs rendszeren alapulnak. Az operációs rendszer lehetővé teszi a firmware által megvalósított működés szoftverszálakba szervezését, a szálak közötti kommunikációhoz IPC módszereket biztosít, a kölcsönös kizárási feltételek biztosítására elérhetővé teszi többek között a mutex mechanizmust, illetve szoftverszálak időzítésétére is lehetőséget ad. A firmware példák fejlesztői a FreeRTOS programozói interfészére egy úgynevezett OSA-t (OS Abstraction Layer, operációs rendszer absztrakciós réteg) illesztettek, amely a mintaprojektek kiegészítése során a FreeRTOS működésének komplexitását jelentős mértékben elfedi.

A node-ok által futtatott firmware indulása után a rádiós modul hardverének inicializációs lépéseit végzi el, majd a hálózati működést megvalósító folyamatok elindítása következik. A node a Thread hálózathoz tartozó beállításait, attribútumait nem felejtő memóriában tárolja. Ezért, ha a node leállásakor egy Thread hálózathoz csatlakozva volt, újraindulásakor a hálózatban betöltött szerepének megfelelően „ott folytatja ahol abbahagyta”, újracsatlakozásra nincs szükség. A node firmware-e szükség esetén factory reset, azaz gyári alaphelyzet állapotba vihető, ekkor a Thread hálózathoz tartozó beállításai törlődnek. A node ebben az állapotban a felhasználás igényeinek megfelelően különböző módokon járhat el, például megkísérelheti az engedélyezett csatornákon a Thread hálózat keresését, és a felderített Thread hálózathoz való csatlakozást.

6.2. Jelentés a node indulása után, kommunikáció kezdeményezése

6.2.1. Jelentés CoAP kommunikáció útján

A dolgozat tárgyát képező vezérlőrendszer és modellhálózat fejlesztése a távoli node firmware kiegészítését tette szükségessé. A távoli node, hálózathoz való csatlakozása után a vezérlőrendszer NMM komponensével kommunikációt kell, hogy kezdeményezzen, amely során az újonnan kapott IP-címét és gyári sorozatszámát közli. A hálózathoz való csatlakozás után tehát a távoli node a menedzser node felé az említett adatokat tartalmazó CoAP üzenetkeretet továbbítja. A CoAP üzenetkeretben az elérési út (URI-path) mezővel azonosítja a menedzser node-on elérni kívánt erőforrás. Az URI-path jelen esetben „/report”, az üzenetkeret adattartalma a távoli node gyári sorozatszáma. Az így indított tranzakció végbemenetele érdekében a menedzser node firmware verzióját ki kell egészíteni a „/report” URI-path által kijelölt működési mechanizmussal.

A CoAP kommunikáció során használt elérési utakhoz a node-ok firmware-e callback függvényeket társít. Induláskor a node memóriájában létrehoz egy tömböt, amely tömb elemei az összetartozó URI-path karaktersorozatokat és a hozzájuk rendelt callback függvényekre mutató pointereket tartalmazó struktúrák. A firmware kiegészítésekor tehát definiálni kell az új (ebben az esetben „/report”) URI-path karaktersorozatot, megírni az ilyen üzenetkeretet feldolgozó függvényt, és az említett tömböt kiegészíteni az URI-path és a callback összerendelésével. Beérkező CoAP üzenet esetén a node firmware-e vizsgálja, hogy az üzenetben küldött URI-path egyezik-e a node által regisztrált URI-path-ok valamelyikével, és egyezés esetén a hozzá tartozó callback függvényt meghívja. A menedzser node a „/report” típusú CoAP üzenethez tartozó callback függvényben a beérkező keret THCI-n történő továbbítását végzi el, tehát a távoli node által küldött adattartalmat az NMM felé továbbítja.

6.2.2. A jelentés folyamata

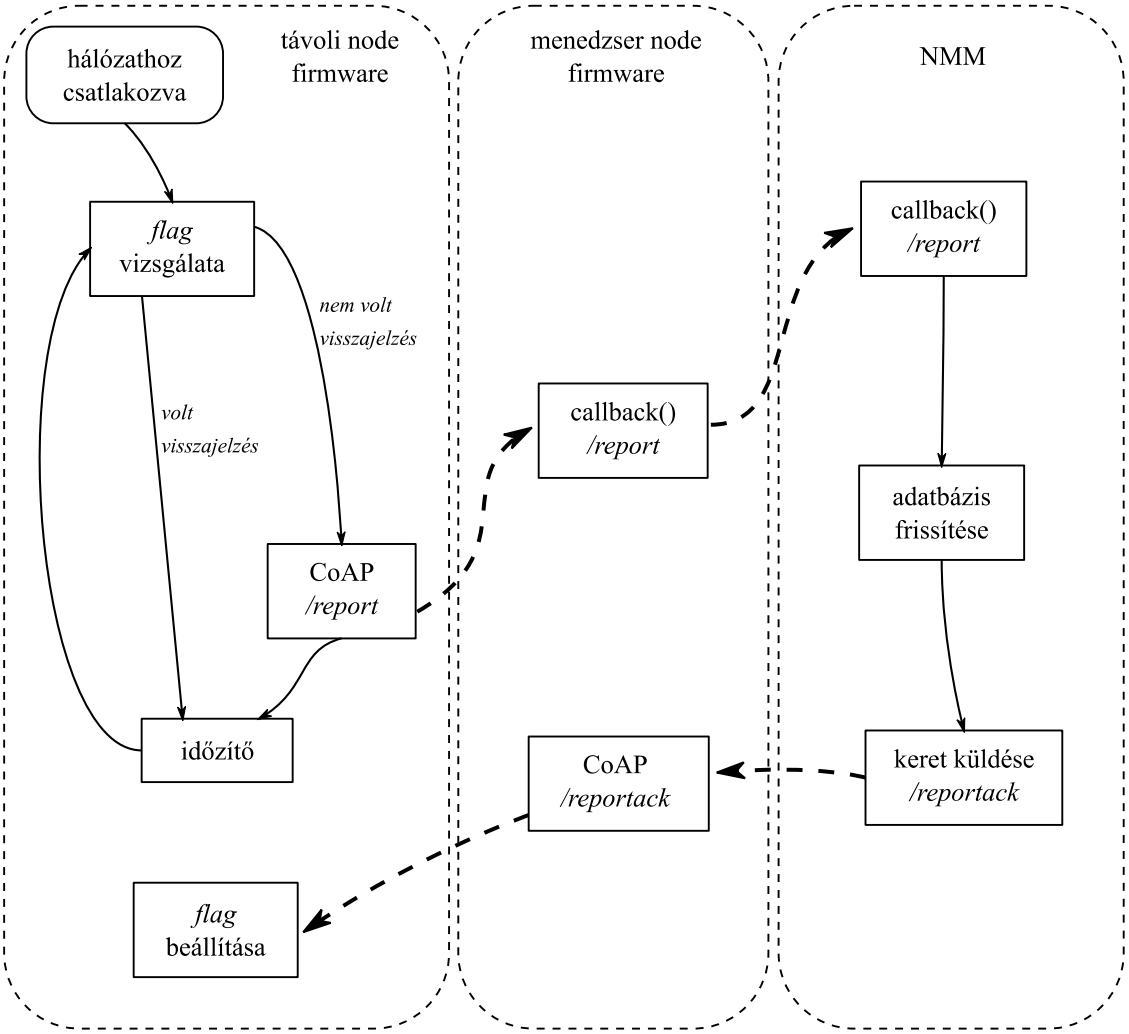
A távoli node firmware indulása után vizsgálja a Thread hálózathoz tartozó tárolt attribútumait. Amennyiben a node nincs csatlakozva egy Thread hálózathoz sem, megkísérli a csatlakozást. Ha a csatlakozás sikertelen, egy időzítőt indít el, amely lefutásakor a csatlakozást újrapróbálja. A távoli node tehát gyári alaphelyzetbe állítása után a hálózathoz való csatlakozásra ismételve kísérletet tesz, mindaddig, amíg a hálózathoz való csatlakozást sikeresen végre nem hajtja.

Sikeres csatlakozást követően a távoli node a menedzser node felé az előzőekben leírt CoAP kommunikációt kezdeményezi. A rendszer működéséhez elengedhetetlen, hogy az így az NMM felé indított adattartalom sikeresen célba érjen, ezért a távoli node a „/report” üzenetkeret befogadásáról visszajelzést vár. A visszajelzés tényét a távoli node egy flag állapotával rögzíti. A CoAP kommunikáció kezdeményezése előtt a flag állapotát vizsgálja, ha a visszajelzés megtörtént, a CoAP keret továbbítására nincs szükség. Ha a flag azt mutatja, hogy a visszajelzés még nem történt meg, a távoli node CoAP keretet továbbítja, majd elindít egy időzítőt. Az időzítő lefutásakor a flag állapotát újra vizsgálja, és az annak megfelelő módon jár el. Vagyis a távoli node a fenti CoAP kommunikáció kezdeményezését addig ismétli, amíg az NMM a bejelentkezést vissza nem igazolja.

A távoli node a bejelentkezéskor a menedzser node-ot az „összes node” IP-címen keresztül éri el (all-nodes multicast address), azaz azt minden, a hálózaton elérhető node megkapja. Ugyanakkor a hálózat node-jai közül egyedül a menedzser node regisztrálja a „/report” URI-path-ot, ezért a többi node ezt a beérkező keretet figyelmen kívül hagyja. A menedzser node a beérkező „/report” CoAP keretet A THCI szolgáltatásai segítségével az NMM-et futtató beágyazott számítógép felé továbbítja. Az NMM a beérkező CoAP keret alapján a távoli node-okról fenntartott adatbázisa megfelelő bejegyzését frissíti,

vagy új bejegyzést hoz létre. Ezután az újonnan regisztrált node immáron ismert IP-címére CoAP üzenetet továbbít a „/reportack” URI-path megjelölésével. Az itt leírt jelentés folyamatát a 8. ábra mutatja.

A távoli node-ok firmware-ében a „/reportack” URI-path az előzőekben tárgyalt módon regisztrálva van, a hozzá tartozó callback függvény a bejelentkezés visszajelzését rögzítő flag értékét állítja be. Mivel a flag beállítása és vizsgálata két egymástól függetlenül időzített szoftverszálban történik, a flagre a kölcsönös kizárás feltételeit biztosítani kell, ennek érdekében azt egy mutex védi.



8. ábra: A távoli node bejelentkezésének folyamata

7. Működés bemutatása, tesztelés

Az előzőekben bemutatott vezérlőrendszer és modellhálózat funkcionális tesztelését manuálisan végeztem. Először az NMM-et futtató számítógép terminálján keresztül ellenőriztem az NMM-nek a kernellel kialakított kapcsolódási felületét, ahogyan az a 9. ábrán látható, majd a Thread hálózat node-jainak elindítása után, az NMM-et futtató számítógép webszerverét helyi hálózaton keresztül elérve, a felhasználói felület által megvalósított funkciókon kersztül vizsgáltam a hálózat működését.

7.1. NMM és kernel kapcsolatának vizsgálata

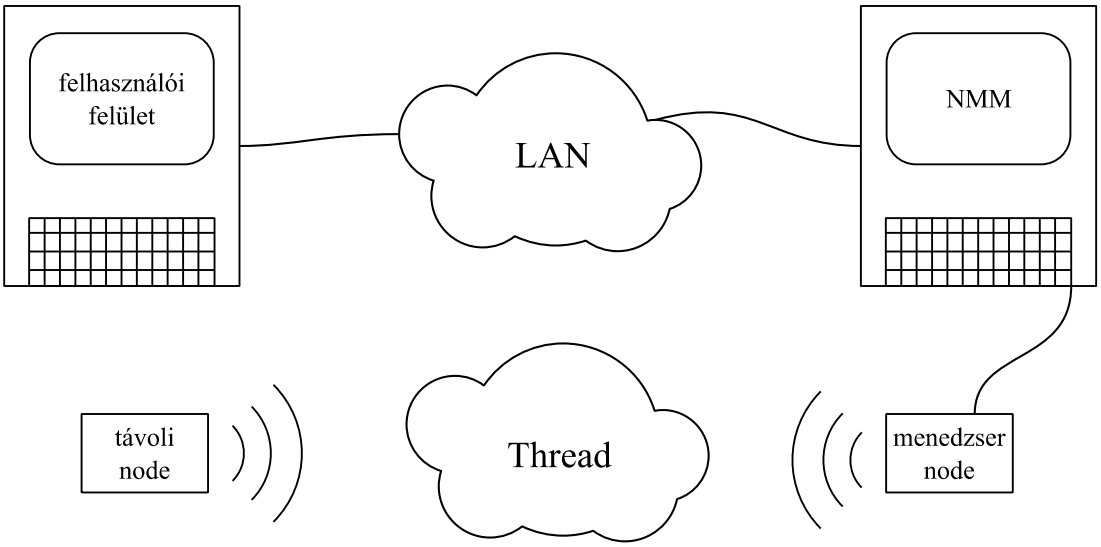
Az NMM indulása után az őt futtató számítógép termináljából ellenőrizhető, hogy az NMM létrehozza a két FIFO-t. A terminálon keresztül az ls parancs segítségével listázható a /tmp könyvtár tartalma, az NMM ebben a könyvtárban hozza létre az nmm_fifo és page_fifo fájlokat. A rendszer indítása után a terminálon keresztül a létrehozott FIFO-k írhatók, illetve olvashatók, így az NMM működése helyileg vizsgálható. Az echo paranccsal az nmm_fifóba a tranzakciót leíró karakter sorozat beírása után a cat paranccsal a page_fifóból olvasható a tranzakció eredménye.

9. ábra: Az NMM és a kernel kapcsolatának vizsgálata terminálon keresztül

7.2. Hálózat vizsgálata a felhasználói felületen keresztül

7.2.1. A modellhálózat

Az NMM-et futtató számítógép webszerver komponense a felhasználói felületként funkcionáló weblapot kiszolgálja, így a vezérlőrendszer elérését biztosítja IPv4 hálózaton keresztül. A tesztelés során a helyi hálózaton keresztül, egy másik számítógépen futó böngészőből indított tranzakciók segítségével vizsgáltam a rendszer működését. A rendszer fejlesztése során, illetve a dolgozat írásakor a Rigado által gyártott próbapanelen kívül csak egy távoli node hardvere állt rendelkezésemre, így a vizsgált modellhálózat két node-ból épül fel. A próbapanel által megvalósított node a menedzser node funkciót látja el, így a weblapon keresztül elérhető világításvezérlő hálózat egy távoli node-ból áll. A vizsgált modellhálózat vázlata a 10. ábrán látható.

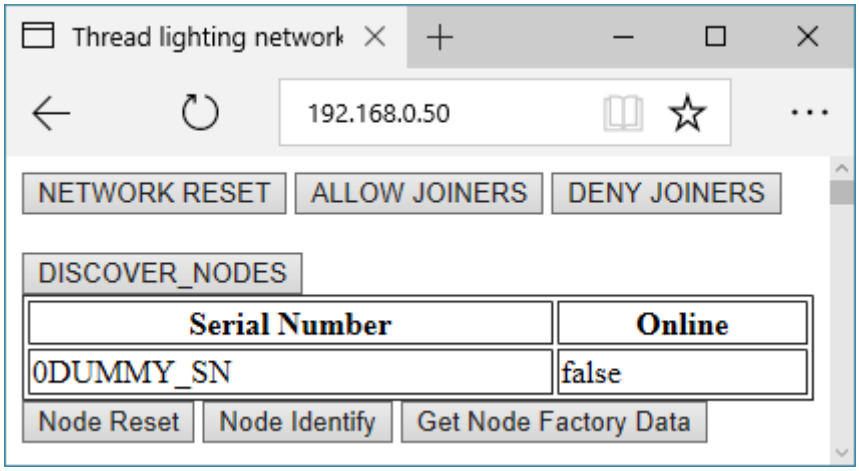


10. ábra: A vizsgált modellhálózat vázlata

7.2.2. A felhasználói felület funkciói

A böngésző által megjelenített felhasználói felületen lehetőség nyílik a hálózat alaphelyzetbe állítására (pontosabban új hálózat létrehozására), a távoli node-ok csatlakozásának engedélyezésére, illetve tiltására, valamint az NMM memóriájában tárolt node adatbázis lekérésére. A weblapon táblázatos formában felsorolásra kerül a világításvezérlő rendszer összes távoli node-ja, amelyek felé a felhasználó (a rendszer karbantartója) táblázatból való kiválasztásuk után projekt-specifikus tranzakciókat indíthat el. A kiválasztott távoli node lámpavezérlő fényerő-beállítása lekérdezhető, illetve beállítható, azonban a rendszer működésének ellenőrzése során – a vezérelt lámpák hiányában – a távoli node hardveres világítási funkcióit nem vizsgáltam. A világítási paraméterek beállítása és újra lekérdezése után, ha egyezés állapítható meg, akkor a működés a hálózat

szempontjából nézve helyes. Ezen felül a weblapon elérhetőek egyéb funkciók, amik a távoli node elérését igazolják. A node azonosítás (node identify) funkció indítása során a távoli node egy beépített ledje villogtatásával jelzi, hogy az üzenetet megkapta, ugyanakkor a távoli node-okat felsoroló táblázatban a node adatbázisban rögzített online mező is megjelenítésre kerül, amely node-dal fennálló kommunikációs kapcsolatot hasonlóképpen igazolja. A weblap lehetőséget nyújt a távoli node gyári alaphelyzetbe állítására, illetve a gyári adatok (többek között a sorozatszám) lekérésére. A vezérlési funkciók felhasználói felületét megvalósító weblap fő kezelőszerveri a 11. ábrán láthatóak.

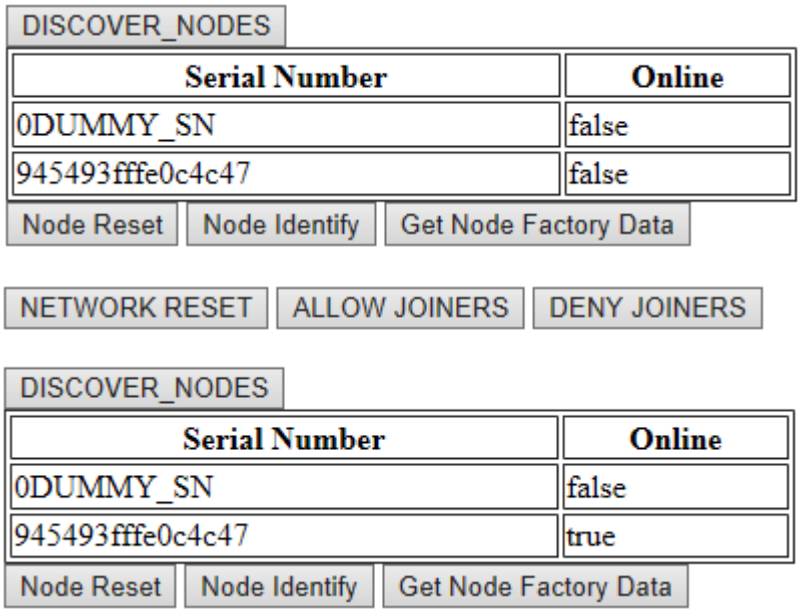


11. ábra: A felhasználói felületnek a rendszer alapfunkcióira vonatkozó részlete

7.2.3. Működés ellenőrzése az elérhetőség vizsgálatán keresztül

A fájlt, amelyben az NMM a node-ok adatbázisát tárolja, indulás előtt egy hamis bejegyzéssel láttam el. A bejegyzéshez tartozó sorozatszám (0DUMMY_SN), és IP-cím a hálózaton nem létező node-ot azonosít, így az erre az IP-címre indított elérhetőségi vizsgálat (ping) nyilván minden esetben időtúllépéssel fog visszatérni. Ez az elérhetőségi vizsgálat fals pozitív eredményeinek kiszűrését szolgálja.

A távoli node gyári alaphelyzetbe állítása után a Thread hálózathoz való csatlakozást kísérli meg. A csatlakozás weblapon keresztül történő engedélyezése után a távoli node sikeresen csatlakozik, majd a menedzser node felé bejelentkező üzenetet továbbít. Ennek hatására az NMM adatbázisában létrejön a távoli node-hoz tartozó új bejegyzés, amely bejegyzés online mezője kezdetben hamis értéket tartalmaz. Ennek az online mezőnek a tartalmát az NMM a következő pingelési ciklus során fogja igaz értékűre állítani, amikor az újonnan bejegyzett IP-címre indított ping tranzakció sikerrel lezajlik. Az adatbázis ennek a működésnek megfelelő állapotait a 12. ábra mutatja.



12. ábra: A node adatbázist megjelenítő táblázat a felhasználói felületen, az új node csatlakozása után, majd a következő pingelési ciklust követően.

7.3. A rendszer automatizált tesztelése

A fejlesztés alatt álló vezérlőrendszer, illetve az általa vezérelt modellhálózat az elvárások szerint üzembe helyezésekor mintegy kétszáz távoli node-ot fog tartalmazni. A távoli node-ok hardverének legyártását követően az üzembe helyezés előtt természetesen a hálózati tesztelés is szükséges lesz a teljes hálózaton. Ahhoz, hogy a rendszer helyes működését, üzembe helyezhetőségét kijelenthessük, a tesztelésnek alaposnak kell lennie, a távoli node-ok elérhetőségi vizsgálatán túl az összes világítási paraméter beállíthatóságát, visszaolvashatóságát is vizsgálni kell. Az említett magas példányszám miatt a manuális tesztelés nem volna gazdaságos, a szoftveres tesztelést valamilyen módon automatizálni kell.

Az automatizált tesztelés tervezése és implementálása a fejlesztés jövőbeli lépései közé tartozik. Az automatizált vizsgálatot olyan tranzakciók útján kell végezni, amelyek a normál használati eseteknek megfelelő összes lépést tartalmazzák, így biztosítandó, hogy a tesztelés eredménye az üzembe helyezés utáni működési jelenségeket egyértelműen előre jelzi. Ezért a tesztrendszernek a tranzakciókat a felhasználói felületen kell indítania, és az eredményüket a felhasználói felületen keresztül kell ellenőriznie, a felhasználói felületet pedig IPv4 hálózaton keresztül kell elérnie.

8. Összefoglalás

Az ebben a dolgozatban tárgyalt feladat specifikációja egy tetszőlegesen skálázható, központilag vezérelt világítási rendszer fejlesztését jelölte ki. A világítási rendszer hálózati protokollja a Thread, amely az IoT piac egy új, igéretes megoldása.

A féléves munkám során létrehoztam egy, a Thread hálózathoz illeszkedő hálózatvezérlő szoftvert, amely lehetővé teszi a hálózat állapotába való beavatkozást egy humán felhasználói felületen keresztül. A hálózatvezérlési funkciók implementálásában a Thread fejlesztői által nyújtott vezérlő interfésznek fontos szerepe volt, a feladat megoldását megelőzően azzal meg kellett ismerkednem. A vezérlő interfész jelenleg elérhető implementációjában a párhuzamos folyamatok kölcsönös kizárásának biztosítása szükséges volt.

A hálózatvezérlő szoftver fejlesztése C/C++ nyelven, Linux operációs rendszert futtató számítógépen történt, szem előtt tartva, hogy a világításvezérlési projekt céljai szerint a hálózatvezérlő funkciót egy Raspberry PI beágyazott számítógépen futó szoftvernek kell megvalósítania. A fejlesztés elsődleges kihívásai az egymástól független szoftverfolyamatok szinkronizálása, és a közöttük zajló adatátvitel megvalósítása voltak. Ezen problémák megoldására a Linux kernel rendszerhívásait és folyamatközi kommunikációs eszközeit használtam fel. A vezérlőrendszer a folyamatok közti kommunikációt a Linux FIFO implementációja segítségével végzi, a folyamatok szinkronizálására pedig a POSIX által definiált mutex és feltételváltozó mechanizmusokat használja. A hálózat állapotának monitorozása érdekében a vezérlőrendszer a hálózat csomópontjairól adatbázist tart fenn, mely adatbázis egyszerű kezelésének implementálásában a C++ sztenderd könyvtárában elérhető asszociatív tároló megoldás volt a segítségemre.

A hálózat működésének demonstrálása érdekében a hálózati csomópontok beágyazott firmware fejlesztése is szükséges volt. A firmware kiindulási alapjául a Thread fejlesztőinek C nyelvű mintakódjai szolgáltak, ezeknek a hálózatvezérlő rendszerhez illeszkedő módon való testreszabását, átalakítását végeztem el.

Az így felállított modellhálózat működését egy távoli node segítségével, a rendszer felhasználói felületén keresztül mutattam be. Jelen dolgozat így teljesíti a feladatkiírásban foglalt feladatokat, ugyanakkor a megvalósított hálózatvezérlő modul a feladatban specifikált világítási rendszertől függetlenül, más célú Threadalapú hálózatok vezérlésére is felhasználható.

9. Irodalomjegyzék

[1] *Internet of Things: Science Fiction or Business Fact?* – Harvard Business Review, 2014, online: https://hbr.org/resources/pdfs/comm/verizon/18980_HBR_Verizon_IoT_Nov_14.pdf

[2] *Why We Made Thread*, Thread protokoll weblap, online: https://www.threadgroup.org/About#OurMembers

[3] *Thread Specification, Ver 1.1.1* – hivatalos specifikáció, 2017

[4] *6LoWPAN demystified* – Texas Instruments white paper, 2014 http://www.ti.com/lit/wp/swry013/swry013.pdf

[5] *The Constrained Application Protocol (CoAP)* - RFC7252, 2014, online: https://tools.ietf.org/html/rfc7252

[6] Kinetis Thread Host Control Interface Reference Manual – NXP Semiconductors, 2017

[7] NXP KW41Z termékleírás, online: https://www.nxp.com/products/wireless-connectivity/zigbee/kinetis-kw41z-2.4-ghz-dual-mode-ble-and-802.15.4-wireless-radio-microcontroller-mcu-based-on-arm-cortex-m0-plus-core:KW41Z

[8] Rigado R41Z termékleírás, online: https://www.rigado.com/products/modules/r41z/

[9] *FIFO(7)*, Linux kézikönyv, online: http://man7.org/linux/man-pages/man7/fifo.7.html

[10] *READ(2)*, Linux kézikönyv, online: http://man7.org/linux/man-pages/man2/read.2.html

[11] *POLL(2)*, Linux kézikönyv, online: http://man7.org/linux/man-pages/man2/poll.2.html

[12] *PTHREAD_COND_TIMEDWAIT(3P)*, Linux kézikönyv, online: http://man7.org/linux/man-pages/man3/pthread_cond_timedwait.3p.html

[13] *std::map*, C++ kézikönyv, online: http://en.cppreference.com/w/cpp/container/map

[14] *ACCESS(2)*, Linux kézikönyv, online: http://man7.org/linux/man-pages/man2/access.2.html