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

A DDS kommunikációs szabvány memóriamodelljeinek formalizálása

Készítette

Rádai Ronald

Konzulens

Huszerl Gábor
BME MIT, mesteroktató

2023

Tartalomjegyzék

Összefoglaló

A kritikus rendszerekben gyakran elosztottan futó, beágyazott komponensek együttműködése biztosítja a megfelelő működést. A helyes működés szempontjából kiemelten fontos a hálózati kommunikáció, a közösen használt adatok elosztott írásának és olvasásának helyes és kiszámítható működése. A szabványosító szervezetek többféle megoldást is kidolgoztak a gyors és megbízható adatcserére, ezek közül az egyik fontos szabvány a Data Distribution Service (DDS), melynek használata több iparágban is igen elterjedt, többek között a repülőgépekben, az autókban, nagyméretű ipari rendszerek vezérlésében. A szabvány jellemzője, hogy gyors adatmegosztást tesz lehetővé, melynek során különböző QoS paraméterek beállításán keresztül képesek vagyunk megszabni az adatok továbbításának tulajdonságait. Ezen beállítások különböző garanciákat adnak a kommunikáló komponensek számára, akik ezekre a garanciákra építve tudják a saját helyes működésüket garantálni.

Ezeket a DDS szabvány által nyújtott garanciákat derítettem fel a szabvány tanulmányozásával és kontrollált környezetben automatizáltan futtatott mintaalkalmazások viselkedésének vizsgálatával, majd a formális módszerek eszköztárát felhasználva a matematika nyelvén definiáltam őket. Erre a célra egy memóriamodell-alapú leíró nyelvet választottam, amely alkalmas elosztott rendszerekben az adatműveletek lehetséges végrehajtásainak precíz modellezésére, ábrázolására. Az általam készített memóriamodellek az első ilyen jellegű formalizálásai a DDS szabványnak. A memóriamodellek felhasználásával lehetőségünk van a DDS protokoll viselkedésének részletes leírására, hogy az azt használó elosztott alkalmazások rendszer szintű formális analízisét is elvégezhessük.

Munkám során az adatvesztésre és az adatok átrendeződésére fókuszáltam, mivel ezek azok a problémák, amelyek leginkább veszélyeztetik az adatkonzisztenciát nagyobb, kritikus rendszerekben. Az egyetlen komponens által végzett írások átrendeződésének elkerülését a DDS minden esetben biztosítani fogja, viszont probléma lehet a sorrendhelyesség akkor, ha több komponens is írja ugyanazt az adatot. Ezen túl készítettem egy valódi, fizikai demonstrátort is, amely jól ábrázolja, hogy miért is érdemes foglalkozni a QoS beállításokkal egy DDS-t használó rendszerben, és milyen katasztrofális meghibásodásokhoz vezethet egy hibásan beállított QoS profil.

Abstract

In critical systems, it is the interoperation of embedded, distributed components that ensures proper operation. The correct and predictable operation of network communication, distributed read and write of shared data, is of paramount importance for correct operation. Standardisation organisations have developed several solutions for fast and reliable data exchange, one of the most important of them is the Data Distribution Service (DDS), which is widely used in many industries, including aerospace, automotive and control of large industrial systems. The standard is characterised by its ability to enable fast data sharing, whereby we can specify the properties of the data transmission through the setting of various QoS parameters. These settings provide various guarantees to the communicating components, which can build on these guarantees to ensure their own correct operation.

I explored these guarantees provided by the DDS standard by studying the standard and examining the behaviour of sample applications run automatically in a controlled environment, and then using formal methods to define them in the language of mathematics. For this purpose, I chose a memory model-based description language, which is suitable for accurately modelling and representing the possible executions of data operations in distributed systems. My memory models are the first formalizations of the DDS standard from this aspect. Using the memory models, I managed to describe in detail the behaviour of the DDS protocol, so that a system-level formal analysis of the distributed applications can be performed.

My work has focused on data loss and data reordering, as these are the problems that most threaten data consistency in large, critical systems. Avoiding reordering of writes by a single component will always be ensured by DDS, but reordering can be a problem when multiple components write the same data. In addition, I have also created a real, physical demonstrator that illustrates why it is important to care for QoS settings in a system using DDS, and what catastrophic failures can result from an incorrectly configured QoS profile.

1. Bevezetés

Munkám során a memóriamodell-alapú megközelítésnek az adatmegosztást használó elosztott alkalmazásokra való kiterjesztésével foglalkoztam. A memóriamodell [1] egy formális specifikáció, amely leírja a számítógépen futó processzek írási és olvasási műveleteinek lehetséges lefutási sorrendjeit, ezáltal az adatok lehetséges terjedési útvonalait. A DDS (Data Distribution Service) [2] (melyet általában az elosztott alkalmazások használnak) egy elosztott adatmegosztó szolgáltatás, amely middleware szinten megoldja, hogy minden komponens egy rendszerben ugyanazt az adatot tudja egy adott időpillanatban olvasni, legalábbis bizonyos beállítások mellett. Ezeket a beállításokat, melyeknek segítségével a számunkra fontos garanciákat tudjuk beállítani, QoS beállításoknak hívjuk. Fontos észrevenni, hogy a DDS alapú kommunikáció és az egy processzoron belüli írási és olvasási műveletek között nagy a hasonlóság, hiszen ugyanúgy elosztottan helyezkednek el (értsd több végrehajtó egység esetén hasonló kommunikációs problémák fordulhatnak elő, mintha több komponensünk lenne, amik bizonyos változók értékét szeretnék megosztott módon írni-olvasni). A munkámban kitérek arra, hogy a DDS és néhány QoS beállítása mit biztosít számunkra, illetve rendszerkomponenseink számára különböző hálózati komplikációk esetén. A garanciák bemutatására a CAT nyelvet használtam, hiszen a CAT [3] a leggyakrabban használt nyelv memóriamodellek leírására.

1.1. Célom

A célom a DDS kommunikáció memóriamodelljeinek leírása CAT nyelven, illetve vizsgáltam néhány QoS beállítás hatását ezekre a modellekre. Az eredményekhez egy tesztkörnyezetben minialkalmazásokat is készítettem, hogy bemutassam a beállítások hatását, és teszteljem a működést. A bemutatáshoz a RTI cég DDS implementációját [4] használtam, a QoS beállítások közül kiemelten foglalkoztam a RELIABILITY, HISTORY és a DESTINATION_ORDER beállításokkal, illetve ezek CAT [3] nyelvű leírásával. A CAT leírás alapján generálhatók lesznek a kommunikációra jellemző memóriamodellek, ezekből worst case elemzéssel a kommunikációs hibák feltárhatóak, elkerülhetőek lesznek.

1.2. Motiváció

Az adatvesztésre, illetve az adat írások átrendeződésére fókuszáltam a munkám során, mivel ezek azok a problémák, amelyek leginkább veszélyeztetik az adatkonzisztenciát nagyobb, kritikus rendszerekben. Ezek közül az adatvesztés megakadályozását a RELIABILITY, HISTORY QoS-sek együttes beállításával és néhány egyéb erőforráskorlát beállítás átállításával lehet elérni, az átrendeződés elkerülését egyetlen adatmegosztó komponens estén pedig a middleware minden esetben biztosítani fogja. A problémát bővebben ki fogom fejteni a munkám során. Ezen kívül érdekes probléma a sorrendhelyesség több adatmegosztó esetén, ugyanis ha a hálózat nagy késleltetéssel továbbítja csak a csomagokat, akkor előfordulhatnak átrendeződések az üzenetek között, vagy ha az írók órája nincs megfelelően beállítva, akkor az időbélyeg is lehet helytelen.

Sajnos a DDS programokat egy fizikai gépről futtatva az alkalmazások optimalizálási okokból osztott memórián keresztül a hálózatot kihagyva kommunikálnak, így hálózati kommunikáció nem történik. Egy másik lehetőség lett volna a fizikai gépeken való demonstráció, viszont ebben az esetben a tesztkörnyezet nem lett volna „hordozható” és automatizálható, a sikeresség az emberi gyorsaságon múlt volna, ezért ezt az ötletet is elvetettem. A megoldást végül a konténer alapú virtualizáció jelentette, hiszen ezzel egyetlen PC segítségével tudunk több adat író és adat olvasó komponenst futtatni, melyek szeparáltan futnak egymástól, és virtuális hálózatokon kommunikálnak, így ezek kommunikációjába kívülről is bele tudunk nyúlni. Egy másik előnye a konténerizációnak, hogy a gazdagépről nagyon egyszerűen, parancsokkal vagy akár scriptekkel is bele tudunk nyúlni a konténerek kapcsolataiba, működésébe, és így egy hasonló demonstrációknál könnyen biztosítható a konzisztens működés, és csökkenthető futtatás közben az emberi hiba kockázata.

1.3. A dolgozat felépítése

A dolgozatom során a 2. fejezetben bemutatom, hogy hogyan lehet hasznosítani a kutatásom eredményét, a 3. fejezetben ismertetem a felhasznált technológiákat. A 4. fejezetben részletezem az egyes QoS-ek memóriamodelljeinek formalizálását, majd ezt követően az 5. és a 6. fejezetben bemutatom a teszteléshez használt scriptek és a tesztelési környezet létrehozásának folyamatát. Majd a 7. fejezetben bemutatok egy fizikai demonstrátort is, melynek segítségével látható az eddig szemléltetett problémák egy valós, fizikai esete is. A **Hiba! A hivatkozási forrás nem található.**. és 0. fejezetet a dolgozat összefoglalására és továbbfejlesztési lehetőségeinek ismertetésére szánom.

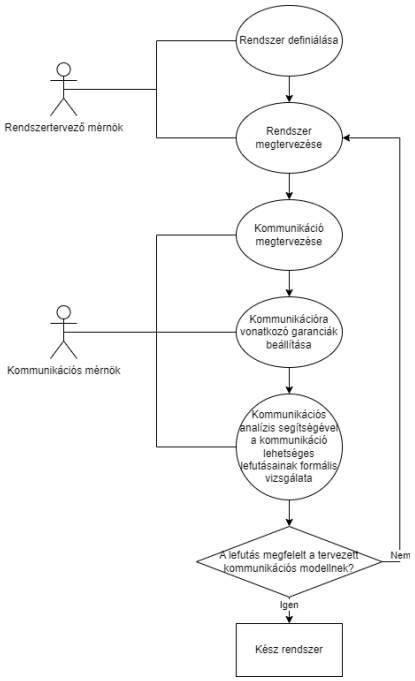
2 A munkám elhelyezése a rendszertervezési folyamatban

Célom, hogy a nagy rendszerek tervezéséért felelős mérnökök munkájának könnyítése érdekében, illetve a meglévő kritikus környezetben használt DDS alapú kommunikáció hibáinak feltárása érdekében formalizáljam a lehetséges kommunikáció memóriamodelljeit, hogy azok összes lehetséges lefutása generálható és ez által ellenőrizhető legyen.

A 2.1-es fejezetben a munkám eredményének egy lehetséges gyakorlati felhasználását mutatom be.

2.1 Az eredmények lehetséges felhasználása a gyakorlatban

Az eredményeimet főként nagyobb beágyazott rendszerek tervezésénél, illetve meglévő rendszerek biztonságának felülvizsgálatánál lehet felhasználni. A DDS szabványban minden szabványos DDS kommunikációs garanciára vonatkozó beállítás leírása jelen van, elérhető, viszont ez sokszor nem elég pontos, nem teljeskörű, vagy megértése nagyon nehézkes. Éppen ezért dolgoztam ki egy olyan tesztkörnyezetet, melynek segítségével ezeknek a beállításoknak a hatása tesztelhető, és az alapján értelmezhető. Én ezeket a teszteket futtatva és megfigyelve formalizáltam a garanciákat. A formalizált leírást egy olyan programba beillesztve, amely a formális leírásokból le tudja generálni az összes lehetséges leírást, könnyedén meg lehet kapni az összes olyan lefutást, amelyek ilyen beállítások mellett lehetségesek. A rendszertervezést az általam megadott formális leírásokkal az 1. ábra szemlélteti.



1. ábra Rendszertervezés a megalkotott formális leírásokkal

Egy rendszer megtervezésénél kezdetben egy erre szakosodott mérnök definiálja a rendszert, megtervezi az architektúrát. A folyamat során magas szintű nem funkcionális követelményeket támaszt a rendszer különböző komponenseinek kommunikációjával szem-

ben. Ezek a magas szintű követelmények azok, amiket a Kommunikációért felelős mérnök megkap, és az alapján tervezi meg a komponensek kommunikációját. Itt jön a munkám a képbe, ugyanis még a tervezés közben a mérnököt segíteni lehet abban, hogy mielőtt bármit is megvalósítana, a tervezett kommunikációs garanciákra vonatkozó beállítások formalizált modelljéből megkapja, hogy milyen tulajdonságok garantálhatóak. A lehetséges lefutásoknak a grafikus ábrázolásával könnyen látható, hogy a tervezett beállítások megfelnek-e a kritériumként adott nem funkcionális követelményeknek. Ennek azért lehet nagy szerepe az ezzel a területtel foglalkozó szakemberek körében, mivel a bonyolult rendszereknél ezeknek a beállításoknak a megfogalmazása nagyon nehéz, és az ezekből adódó hibák nagy problémát okozhatnak kritkus rendszerekben való használat esetén.

3 Felhasznált technológiák

Ebben a fejezetben bemutatom a legfontosabb technológiákat, amelyeket a megoldás során használtam. Fontos, hogy megismerjük ezeket a technológiákat a további eredmények megértése érdekében.

A demonstrációhoz az RTI cég implementációját fogom használni, egész pontosan az RTI Connext DDS 6.1.1-es verzióját.

A továbbiakban ismertetem a DDS-t és az alapvető működését (3.1. fejezet), a 3.1.1-es fejezetben külön kitérek az RTI Connext DDS implementációra. Ezen fejezetek fontos ismereteket hordoznak, amelyek segítik a dokumentáció teljes megértését. A 3.2. fejezetben ismertetem a Dockert mint technológiát és azokat az eszközöket, amelyekkel a Docker képek közti kommunikációt befolyásoltam (nevezetesen a Pumbát (3.3. fejezet) és a TC-t (3.4. fejezet)). A 3.5.fejezetben bemutatom a CAT nyelv alapjait, ami segíti az elkészült memóriamodell leírások megértését.

3.1 DDS

A DDS (Data Distribution Service) egy olyan szolgáltatás, amely middleware szinten képes adatot megosztani egy rendszer különböző komponensei között. Minden komponens minden megosztott változópéldányra egy lokális cache-t tart fenn, amelyben lehetőség szerint a különböző változók legfrissebb értékét tárolja. Ezen lokális másolatok valós idejű konzisztenciáját biztosítja a DDS szolgáltatás, illetve az azt megvalósító middleware. Működése során kiemelkedő megbízhatóságot és alacsony késleltetést biztosít, ezért gyakran használják kritikus környezetekben és az ipari IoT világában.

A kommunikáció topicokra osztható, ezek olyan struktúrák, amelyeknek közös az adattípusa a kommunikáció során. Ezekben a topicokban adatokat osztanak meg a komponensek egymás között. Az ilyen topicokba való adat írást végzik a publisherek, akik a kommunikáció során az adatmegosztók szerepét töltik be. Ezeket az adatokat a subscriberek olvassák, akik valamilyen logika szerint feldolgozzák, tárolják ezeket az adatokat. Az adatok publisherektől subscriberekig jutásának viselkedését (késleltetés, biztonság, hibatolerancia, idő, erőforráshasználat) bizonyos QoS paraméterek segítségével tudjuk beállítani. Minden publisher az üzenetek küldése során sorszámozza az adatmódosításokat, így amennyiben a változó értékét beállító üzenetek átrendeződnek vagy elvesznek, azt a subscriber oldalon detektálni, vagy egyes beállítások esetén javítani tudjuk. Ebben segít a RELIABILITY QoS, amely azért felelős, hogy amennyiben ez a szekvenciaszám nem megfelelő, a middleware addig ne adja át az új értéket az alkalmazásunknak, ameddig a másik komponenstől meg nem érkeztek a hiányzó írások, ha kell, akkor újra elkéri a publishertől. Amennyiben ezt a beállítást nem aktiváljuk, és az írás üzenetek átrendeződnek vagy kiesnek, akkor az esetlegesen késve érkező üzeneteket a middleware eldobja, ezzel biztosítva, hogy ne történjen átrendeződés. DDS környezetben azt is lehet szabályozni, hogy hány DataWriter számára engedélyezett egy adott változó írása. Az `OWNERSHIP` policy segítségével tudjuk megmondani, hogy egy adott változót egyszerre több DataWriter tud-e írni, vagy csak egy, azaz van-e egy tulajdonosa.

Abban az esetben, ha a QoS beállítások nem egyeznek meg a különböző komponensek között, akkor, ha a beállításaik kompatibilisek, fognak tudni kommunikálni, ellenkező esetben adatküldés nem történik a komponensek között. Fontos megjegyezni, hogy a DDS aszinkron callback függvények segítségével is megvalósíthatja az olvasást, ami azt jelenti, hogy ha van új adat, akkor arról értesíti a middleware az alkalmazást, aki kiolvashatja azt. Ezzel szemben a CAT nyelvben az olvasás csak akkor történik meg, amikor explicit olvasunk egy tartalmat (LDR – LoaD Register utasítás Assembly-ben). Igaz, hogy a két olvasási mód között szignifikáns különbség van, de ez a jelen alkalmazásban nem okoz problémát, mivel nem az írás típusát vizsgáljuk, hanem az írások egymásra hatását. Jelen alkalmazásban ez a különbség elhanyagolható.

3.1.1 RTI Connext DDS

A következőkben néhány alapfogalmat fogok definiálni, melyek az RTI DDS implementációjával kapcsolatosak, ez a dolgozat érthetőségét megkönnyíti.

Platformok: Egy olyan környezetre jellemző leíró, amelynek alapján az adott rendszereket kategorizálni lehet az arra szükséges kód, fordítási segédfile-ok segítségével. Ilyen leírásokra példák:

- x64Linux4gcc7.3.0 – x64 architektúra, Linux kernel 4-es verzió[1], gcc fordító 7.3.0 verzió
- armv7Linux4gcc7.5.0 – armv7 architektúra, Linux kernel 4-es verzió, gcc fordító 7.5.0 verzió
- x64Win64VS2017 - x64 architektúra, Windows 64 bites verzió, Visual Studio 2017

Host package: Egy olyan telepítendő állomány, amely telepíti a fordító gépre a DDS implementációt, a fordításhoz szükséges függőségeket és könyvtárakat. Általában a formátuma `.run`, `.exe` vagy `.dmg` az operációs rendszer függvényében.

Target package: Egy olyan telepíthető állomány, amely az cél gép architektúrára és rendszerre a fordításhoz szükséges függőségeket, könyvtárakat tartalmazza. Általában a formátuma `.rtipkg`, `rtipkginstall` segítségével telepíthetőek. Ezek a csomagok azért fontosak, mert a komponenesek sokszor beágyazott környezetben futnak, ahol a fordítás nem feltétlen lehetséges, illetve a gazdagépen való munka sokkal kényelmesebb. Ebben az esetben a target package segítségével fordított állományt a cél gépre kell juttatni és ott futtatni.

Licensz file: Egy olyan file, amely szükséges a middleware futtatásához, a licenszadatokat tartalmazza. Az RTI DDS egy fizetős, zárt forráskódú megoldás, amely licenszköteles, de a BME a RTI University Program tagja, így a hallgatók oktatási-kutatási célra ingyenesen igénybe vehetik és használhatják az implementációt.

IDL file (Interface Definition Language): egy általában elosztott számítógépes környezetben használt interface leíró nyelv, mely a processzek közti kommunikációt se-

[1] A megadott verziószámnál magasabb verziójú kernellel is általában működik

gíti. A DDS-ben a kommunikáció definiálására a CORBA szabványban megfogalmazott IDL nyelv alapján készült file alkalmas, melyben definiált adatstruktúrába írjuk az adatokat, amelyeket a middleware-nek átadunk, hogy eljuttassa a publisherektől a subscriberekig (pl. egy teremhez tartozó hőmérsékletet tároló struktúra definíciója). A CORBA IDL szintaktikája nagyban hasonlít a C nyelvre, általában `.idl` a kiterjesztése. Az IDL mellett a tool bemeneti formátumként támogat még XML és XSD formátumot is, de a munkám során ezt nem fogom használni.

Példány: Az IDL-ben definiált interface-t megvalósító osztály egy példánya, amelyből egy kommunikáció során lehet több is (pl. különböző termek hőmérsékletét tároló változók)

Minta (Sample): Egy adott példány aktuális értéke, egy konkrét mérés eredménye, melyet a publisherek a subscribereknek küldenek. (pl. egy adott teremben a hőmérséklet aktuális értéke)

QoS profile: QoS beállítások egy gyűjteménye, amely meghatározza, hogy milyen követelményeknek kell megfelelnie egy kommunikációnak. Vannak beépített profileok (pl `BuiltinQosLib::Generic.StrictReliable` vagy `BuiltinQosLibExp::Generic.BestEffort`). Az egyszerűbb definiálás érdekében a beépített profilokból le tudjuk származtatni a sajátunkat. Ha nem szeretnénk sajátot létrehozni, akkor a beépített profilokat is használhatjuk a kommunikáció módjának jellemzésére.

A Connext DDS megfelelő működéséhez fontos még az `NDDSHOME` környezeti változó beállítása, ennek értékeként a DDS telepítési mappáját kell megadni.

Fontos megjegyezni, hogy az RTI DDS middleware bármilyen QoS beállítása esetén a duplikáció és átrendeződés nem fordulhat elő egy adatíró komponens írásai esetén, ez minden esetben biztosítva van (minden publisher sorszámozza az írásait, ha az olvasó azt látja, hogy ugyanolyan sorszámú vagy már megérkezett írás sorszáma előtti sorszámú üzenet jön, akkor azt az értéket eldobja).

3.2 Docker

A Docker [5] egy olyan nyílt forráskódú konténerizációs technológia, amely kernelszintű virtualizációt használ, vagyis a kernel a host géppel közös, viszont tőle el van szeparálva, nem tudnak a folyamataik egymásra hatni. Ezen technológia segítségével viszonylag kis plusz költséggel lehet konténereket futtatni. A konténer egy olyan, a host rendszertől és a többi konténertől elszeparált egység, amely általában egy program vagy folyamat futtatásáért felelős. A Docker környezetről még fontos megemlítenem, hogy a hálózatot is virtualizálja, minden konténernek saját virtuális interfésze van, és ezek az interfészek virtuális switchekhez vannak csatlakoztatva. A Docker azért fogja segíteni a munkámat, mert lehetővé teszi, hogy előre definiált, konzisztens környezetben futtassam a tesztjeimet. A konténerek a virtuális hálózaton keresztül kommunikálnak, így nem tudják kihasználni a memórián keresztüli kommunikációt. Ezen felül sokkal kisebb költséggel futnak, mintha szeparált virtuális gépek lennének, sőt a tesztek teljes mértékben automatizálhatóak, mivel a host gépről csatlakozhatunk a konténer termináljához, azon parancsokat

adhatunk ki, a virtuális hálózatot manipulálhatjuk. Ezek a kiadott parancsok ütemezhetőek egy script segítségével, így biztosítva a reprodukálhatóságot és a konzisztenciát.

3.3 Pumba

Az adatcsere befolyásolására egy Pumba [6] nevű tool-t kezdtem el használni, mivel ez az egyik leginkább használt „chaos engineering” eszköz Docker konténerekhez. A használata kifejezetten egyszerű, és számos paraméterezési lehetőséggel rendelkezik. Sajnos rövid időn belül kiderült, hogy ez nem felel meg az igényeimnek, így váltottam részben a TC nevű Linux program használatára, részben a Windows-os tesztek esetében a Docker képek virtuális hálózatról való lecsatlakoztatására.

3.4 TC

A TC [7] egy hasonlóan hálózati forgalom manipulációra használt eszköz, melyet a Pumba is használ, viszont nála alacsonyabb szintű. Szerencsére ettől függetlenül a közvetlen használata viszonylag egyszerű.

A TC-nek kiadható parancsok egyik leggyakrabban használt felépítése a következő:

`tc` [beállítások] `qdisc` [add | change | replace | link | delete] `dev` [eszköz] [parent qdisc-id | root] [qdisk class] [qdisc specifikus paraméterek]

A demonstrációk során a következő paraméterezésekkel is használtam például a parancsot:

> `tc qdisc add dev eth0 root netem loss 100%` -> A parancs az eth0 interfészre beállít egy új szabályt, amely az összes kimenő csomagot eldobja.
>
> `tc qdisc add dev eth0 root netem delay 1s 2s` -> A parancs az eth0 interfészre beállít egy új szabályt, amely az összes kimenő csomagot késlelteti véletlenszerűen egy 0 és 2 másodperc közti késleltetéssel. Tehát a szabály hozzáadásával elérhetjük, hogy a küldő által küldött csomagok átrendeződjenek.

Fontos megjegyezni, hogy a TC parancs segítségével csak kimenő irányba lehet a hálózati forgalmat manipulálni, a befelé irányulót nem, ezért a demonstráció működtetése érdekében a parancsot a publisheren érdemes futtatni.

Érdekes jelenség, hogy ha a subscriber (vagyis a kommunikáció során az olvasó oldal) kimenő üzeneteit teljesen eldobjuk, akkor egy ideig semmit nem tapasztalunk, viszont ~40mp késleltetéssel a publisher megszakítja a kommunikációt a subscriberrel. Ez azzal magyarázható, hogy a publisher és a subscriber egy külön csatornán (nem a topicon belül) kommunikál egymással, hogy ha a subscriber meghatározott időközönként nem küld életjelet magáról, akkor a publisher annak érdekében, hogy ne ”árassza” el feleslegesen a hálózatot, azt fogja feltételezni, hogy a subscriber nem olvassa az üzeneteit (leszakadt a hálózatról, meghibásodott, kikapcsolt), és így abbahagyja az üzenetek küldését.

3.5 CAT

A CAT absztrakt nyelv, melynek segítségével memóriamodellekkel írható le a többszálú programok lefutása. Ez azt jelenti, hogy segítségével egy program összes lehetséges lefutása közül ki lehessen szűrni azokat, amelyek nem lehetségesek egy adott szcenárióban. Egy program lehetséges lefutásait végrehajtási gráffal szoktuk jellemezni, amelyben G(V, E, l_v) irányított élek szerepelnek, ahol l_v egy élhez tartozó címke, V és E a memóriával előforduló két esemény (írás, olvasás), G pedig ezen két eseményt összekötő él. [8]

Ezek az élekhez tartozó címkék lehetnek:

- `po` – program order, vagyis a forráskódból eredő utasítássorrend
- `rf` – read from, vagyis olyan él, amely adatfolyamot jelenít meg az írások és az olvasások között. Az rf élek mindig az adatáramlás irányába mutatnak.
- `co` – coherence order, vagyis az azonos című írások globális rendezése
- `int` – ugyanazon szál belső eseményei
- `ext` – nem ugyan azon a szálon történő események

A végrehajtási gráf éleinek halmazán műveleteket tudunk definiálni, melyeknek segítségével kiszűrjük a nem megfelelő lefutásokat.

A CAT által definiált műveletek lehetnek:

- `|` – unió
- `++` – halmaz összeadás
- `;` – konkatenálás
- `&` – metszet
- `\` – különbség
- `*` – Descartes szorzat

Ezen halmazokra írhatunk elő kritériumokat, melyek lehetnek:

- `empty` – a halmaz üres
- `irreflexive` – a halmaz irreflexív
- `acyclic` – a halmaz körmentes

A kritériumoknak a tagadása is kifejezhető a tilde (~) szimbólummal (pl. `~empty`).

Nagy előnye a CAT nyelvnek, hogy matematikai leírással lehet ábrázolni a lehetséges lefutásokat, és azokat így tudjuk tanulmányozni, következtetéseket levonni vagy akár formálisan elemezni. Mindezt jelen munkámban DDS feletti adatmegosztást használó elosztott programok lehetséges lefutási sorrendjének megadására használom.

A CAT nagyon sok lehetőséget hordoz magában, hiszen nagyon kompaktan leírhatóak benne a bonyolult kommunikációs modellek, viszont vannak hátrányai is. Például CAT-ben a számosság nem írható le, csak különféle kerülő megoldások segítségével (a 2-vel ezelőtti üzenet megegyezik az előző előtti üzenettel). A megalkotott CAT leírások teszteléséhez a Herd7 simulator [9] nevű lefutásokat generáló online demo toolt használom, amely generálja az adott modell melletti összes lehetséges lefutást. A toolnak van egy teljes verziója is [10], de ez telepítést igényel.

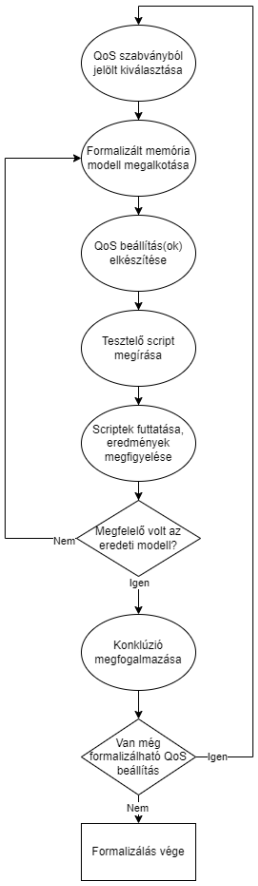
4 A DDS QoS beállításai működésének ellenőrzése és tesztelése

Ebben a fejezetben a különböző DDS QoS beállítások formalizálásával, bemutatásával, és működésének megfigyelésével fogok foglalkozni, illetve a QoS-ek hatásait fogom vizsgálni a különböző QoS-t használó alkalmazásokra.

A 4.1. fejezetben bemutatom, hogyan végeztem a memóriamodellek formalizálását, a 4.2. fejezetben ismertetem a modellek validálására használt tesztkörnyezet működését. A 4.3. fejezetben röviden összefoglalom, hogy milyen egységes logika mentén tervezem bemutatni az egyes tulajdonságokhoz tartozó modellek működését és formalizálását a 4.4., 4.5. és 4.6. fejezetekben.

4.1 A formalizálás folyamata

A formalizálási folyamatot az 2. ábra szemlélteti. A formalizálás során minden bemutatott QoS beállításnál a szabványban definiált működés szerint megalkottam egy memóriamodellt, majd ezt a modellt próbálom validálni a tesztkörnyezettel (ennek a megalkotása során szükség van a megfelelő beállításhoz specifikus QoS beállítás, tesztelő script és DDS alkalmazás megírására), majd ezeket a lépéseket addig ismétlem, amíg a mérés eredményei meg nem egyeznek a memóriamodell által definiált lépésekkel.

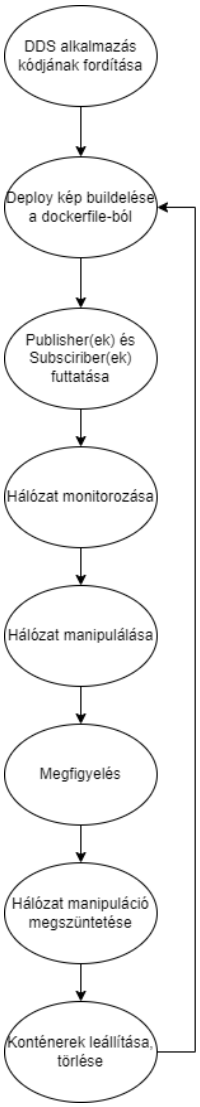


2. ábra Memóriamodell formalizálás folyamata

4.2 A tesztkörnyezet működési elve

A QoS-ek adatátvitelre gyakorolt hatásainak megvizsgálásához és az eredmények bemutatásának céljából összeraktam egy tesztkörnyezetet, amellyel elérhető, hogy több Docker kép között a csomagok átrendeződjenek, vagy teljesen elvesszenek. Ebben a környezetben egy olyan saját készítésű, DDS middleware-rel felszerelt programot használtam, melynek segítségével könnyen lehet detektálni a kiesett vagy átrendeződött üzeneteket.

A teszteket általában egymáshoz nagyon hasonló fő tesztelő scriptek futtatják. Ezeknek a scripteknek az általános modelljét a 3. ábra mutatja be.



3. ábra Tesztkörnyezet fő futtató script működése

A „google.com” pingelése lépés nem feltétlen lenne szükséges a tesztkörnyezet működtetése érdekében, ennek azért van jelentősége, mert így megfigyelhetjük, hogy a hálózat manipulálása hogyan befolyásolja a kommunikációt egy külső szerverrel.

A tesztek különböző változatait Windows és Linux alatt is működésre bírtam. Mivel Windows alatt a Docker WSL-t [11] használ, amiben a TC által használt funkciók kernel szinten nincsenek implementálva [12], ezért a tesztkörnyezet működés közben a Docker network emulált hálózatának beállításait módosítja, hogy elérje a csomagvesztést (le fogja csatlakoztatni a Docker képet a hálózatról teljesen, pont mintha kihúztuk volna egy fizikai gépből az Ethernet kábelt). Linux alatt a TC-vel két dolgot tudok bemutatni:

- az adatok nem rendeződnek át, ha a jitter értéke megnő, viszont változó írás üzenetek elvesznek
- ha a csomagok egy részét eldobjuk, akkor a változó írás üzenetek is elvesznek

Ezeket a problémákat (változó írás üzenetek elveszése) bizonyos QoS beállításokkal kompenzálni lehet, a tesztkörnyezet ezt is bemutatja.

A teszteket elvégeztem fizikai gépeken is, ahol a konklúzió ugyanaz lett, ebből feltételezhetjük, hogy a virtualizálás nem befolyásolta a tesztek eredményét.

4.3 Memória modell bemutatásának folyamata

Minden demonstrátor esetében (lásd 4.4, 4.5, 4.6) a következő struktúrában fogom bemutatni a tesztkörnyezet felépítését, működését:

1. A vizsgált tulajdonságot leíró memóriamodell
2. QoS paraméter jelölt
3. Mérési elrendezés
4. Szkriptek magyarázata
5. Eredmény, tapasztalat

4.4 Sorrendhelyesség

A sorrendhelyesség egy nagyon fontos tulajdonság a kommunikáció során, hiszen a hálózatok világában gyakran előfordul, hogy egy csomag extra késleltetést szed magára az útja során. Ha ez gyakran előfordul (hibás linkek, hálózati eszközök esetén gyakori lehet), akkor a kommunikációban a csomagok átrendeződhetnek, ennek pedig komoly következményei lehetnek egy kritikus rendszerben.

4.4.1 A sorrendhelyesség memóriamodellje

Annak, hogy az adatolvasások nem rendeződnek át, CAT-ben van egy speciális neve, ezt hívjuk Sequential Consistency-nek, vagy röviden `SC`-nek. Az SC a következő CAT nyelvű kóddal írható le:

```
1 include "cos.cat"
2 let fr = rf^-1; co
3 acyclic po | fr | rf | co as sc
```

Az első sor betölt egy másik CAT file-t, ami jelen esetben a `cos.cat` [13]. Erre azért van szükség, mert a `co`, vagyis coherence order nincs definiálva alapértelmezetten, ez ebben a file-ban található.

A második sorban definiáljuk a `fr`-t, vagyis a from-read-et. Ez egy olyan reláció, ami egy olvasó műveletet köt össze egy olyan írással, ami később történik, mint az az írás, amiből az olvasó művelet kapta az értékét. Vagyis az olvasó műveleteket köti össze az összes olyan író művelettel, amiknek eredményét (még) nem lát(hat)ta a komponens akkor.

A harmadik sorban a valódi kritérium van, vagyis nem lehet kör a `po` (program order) unió `fr` (from-read) unió `rf` (read-from) unió `co` (coherence order) élek között. Ez azért fontos, mert ha lenne egy ilyen kör, akkor az azt jelentené, hogy valamely írás-olvasás pár nem a program order szerinti sorrendben történt volna.

4.4.2 A sorrendhelyességhez használt QoS beállítások

A sorrendhelyesség beállításához a protokoll leírásban egyetlen QoS beállítást sem találunk, mivel ez a DDS implementációban minden esetben biztosítva van a komponensek

számára. Nincs olyan beállításhalmaz, melynek segítségével elérhető lenne, hogy egy Publisher által írt minták ne az írás sorrendjében érkezzenek meg az olvasókhoz. Az RTI DDS esetében, ha nem Reliable a kommunikáció, akkor a később megérkezett, ámbár korábbi minták egyszerűen eldobásra kerülnek. Tehát ebben az esetben a működés demonstrálásához egy üres QoS profile-t használtam, ami a beépített (és egyik legegyszerűbb, legkevésbé erőforrásigényes) `Generic.BestEffort` QoS profilból származik. A QoS profile a következőképpen nézett ki:

```
1 <qos_profile name="TestData_Profile"
  base_name="BuiltinQosLibExp::Generic.BestEffort">
2     <datawriter_qos>
3         <publication_name>
4             <name>TestDataDataWriter</name>
5         </publication_name>
6     </datawriter_qos>
7     <datareader_qos>
8         <subscription_name>
9             <name>TestDataDataReader</name>
10         </subscription_name>
11     </datareader_qos>
12     <participant_qos>
13         <participant_name>
14             <name>TestDataParticipant</name>
15             <role_name>TestDataParticipantRole</role_name>
16         </participant_name>
17     </participant_qos>
18 </qos_profile>
```

A fent látható QoS beállításban a következő információkat érdemes kiemelni:

- A QoS profil a beépített `Generic.BestEffort` QoS profilból származik (1.sor)
- A Publisherek QoS beállításai (2.-6.sor)
- A Subscriberek QoS beállításai (7.-11.sor)
- Az összes kommunikációban résztvevő QoS beállításai (12.-17.sor)

4.4.3 Mérési elrendezés

A mérés során egy Publisher és egy Subscriber kommunikál egymással, kettőjük között a hálózat üzeneteit átrendezzük, majd egy másik próbálkozás során teljesen eldobjuk néhány másodpercig.

A mérési ötlet a következő: A Publisher egy szekvenciaszámot növel folyamatosan, fix időközönként, másodpercenként többször. A Subscriber ezt a számot olvassa, és ha a jelenleg érkező szám kisebb, mint az előtte érkező szekvenciaszám, akkor érzékeli, hogy átrendeződtek az üzenetek.

A demonstrációhoz használt IDL a következő:

```
1 struct TestData
2 {
3     unsigned long num;
4 };
```

Az IDL file-ból láthatjuk, hogy egy darab előjel nélküli `long` típus van benne definiálva, amely annak a szekvenciaszámnak a típusa, amelyet a Publisher írni, a Subscriber pedig olvasni fog.

4.4.4 A futtató scriptek és programok

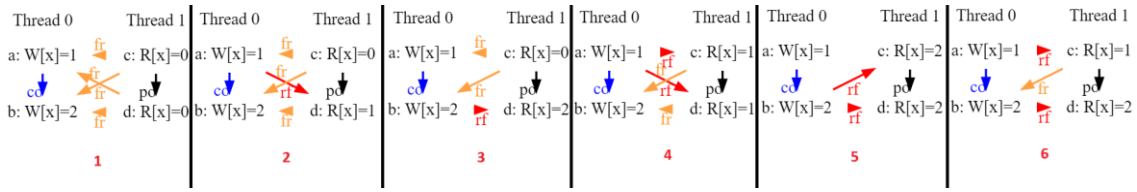
A tesztet futtató script az A. Függelékben olvasható. A script működése a következő:

- Megnézzük, hogy super user jogokkal fut-e a script, ha nem, akkor újraindítjuk, hogy azzal fusson (3-14. sor)
- A DDS alkalmazás kódjának fordítása (19. sor)
- Deploy kép buildelése a docker file-ból (22. sor)
- Jelenleg aktív ablak id-jának lekérése (25. sor)
- Subscriber futtatása (28. sor)
- Publisher futtatása (31. sor)
- Pingelés indítása a ’google.com’ felé, hogy figyelemmel tudjuk kísérni a hálózat manipulációját (35. sor)
- Fókusz visszaállítása az eredeti terminal ablakra (41. sor)
- Publisher kimenő csomagjainak átrendezése (46. sor)
- Várakozás billentyűzet eseményre (48. sor)
- Csomag átrendezése szabály törlése (51. sor)
- Várakozás billentyűzet eseményre (53. sor)
- Konténerek törlése (56-57. sor)
- Subscriber futtatása (63. sor)
- Publisher futtatása (66. sor)
- Pingelés indítása a ’google.com’ felé, hogy figyelemmel tudjuk kísérni a hálózat manipulációját (70. sor)
- Fókusz visszaállítása az eredeti terminal ablakra (75. sor)
- Publisher kimenő csomagjainak eldobása (80. sor)
- Billentyű eseményre várakozás (82. sor)
- Csomageldobás megszüntetése (85. sor)
- Billentyű eseményre várakozás (87. sor)
- Konténerek leállítása (90-91. sor)

4.4.5 Eredmény, tapasztalat / Konklúzió

A tesztek során kiderült, hogy az egy Publishertől származó írások csak sorrendben juthatnak el a Subscriber alkalmazásig. Olyan beállítás nem lehetséges, hogy a Subscriber egy adott írásnál előbb kapjon meg egy később írt adatot egy Publisher esetén. Tehát bármely QoS beállítások mellett a 4.4.1-es fejezetben tárgyalt leírás (a sorrendhelyesség, azaz Sequential Consistency, SC memóriamodelljéről) érvényes lesz. A memóriamodellből generált lefutásokat a 4. ábra szemlélteti egy olyan esetben, amikor 2 szál van, illetve

4 utasítás. A 0-s szál kétszer ír egymás után (1-es, majd 2-es értéket), a 1-es szál pedig kétszer olvas egymás után.

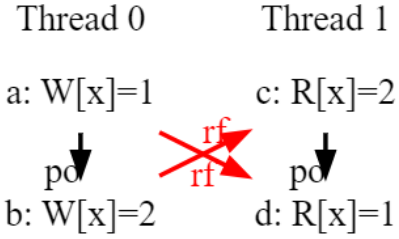


4. ábra SC memóriamodelljéből generált lefutások

A lefutásokból láthatjuk, hogy az 1. lefutásnál egyik olvasás sem olvassa az írások értékeit, tehát egyszerre nem volt aktív az író és az olvasó komponens. A 2. lefutás esetén azt figyelhetjük meg, hogy az olvasó komponens már aktív volt, mielőtt az író komponens első írása megtörtént volna, ezért az első olvasás az IW-ból (initial write) olvasott, a második pedig az a: műveletből. A 3. lefutás alkalmával szintén aktív volt az olvasó komponens, mikor az első írás megtörtént, ám a 0-s szál írása után nem történik átütemezés, hanem még egyszer ír a 0-s szál, majd utána fog a másik szál olvasni. A 4. lefutás egy olyan esetet mutat be, amikor az első írás után történik mindkét olvasás, majd a 2. írás. Az 5. esetben a 2. írás után történnek az olvasások, így azok ugyanazt az értéket olvassák. A 6. eset az „optimális” lefutása a programnak, vagyis minden írást egyszer olvasnak. Jól láthatóan olyan lefutás nincsen, ahol az olvasó komponens először a második írás eredményét olvasná, majd utána az elsőét.

Fontos megjegyezni, hogy mivel a `Reliablility` nem képezte részét a modellnek, ezért itt nincs garantálva, hogy minden írást olvasni fognak. Ezen kívül DDS esetén az on data avaible típusú kommunikációnál a lefutások nem feltétlen ebben a sorrendben történnek, de szemléltetni nagyon jól lehet rajta a működést.

Azt a lefutást, amit ez a modell kiszűr, az 5. ábra mutatja be. Ez az az eset, amikor előbb történik meg a második olvasás, mint az első, ezért a második olvasás még az első írás eredményét olvassa, majd megtörténik a második írás, és végül az első olvasás, amely a második írás eredményét fogja olvasni. Ez a lefutás a DDS middleware használatával nem történhet meg, ugyanis ellentmond az `acyclic po|fr|rf|co` szabálynak.



5. ábra Nem SC lefutás

4.5 Adatvesztés

Az adatok megérkezésére való garancia egy nagyon fontos tulajdonság beágyazott környezetben a különböző komponensek kommunikációja esetén. Ha egy ilyen kritikus környezetben egy adat írásról szóló üzenet kiesik, akkor elképzelhető, hogy a meghibásodás katasztrófához vezet. Szerencsére megfelelő QoS beállítások esetén az adatvesztés nem fordul elő (vagyis garantálható, hogy az nem fordulhat elő, hogy egy későbbi írást úgy olvasunk, hogy egy korábbi írást még nem láttunk), így ez a hiba megelőzhető.

4.5.1 Adatvesztés nélküli írás-olvasás memóriamodellje

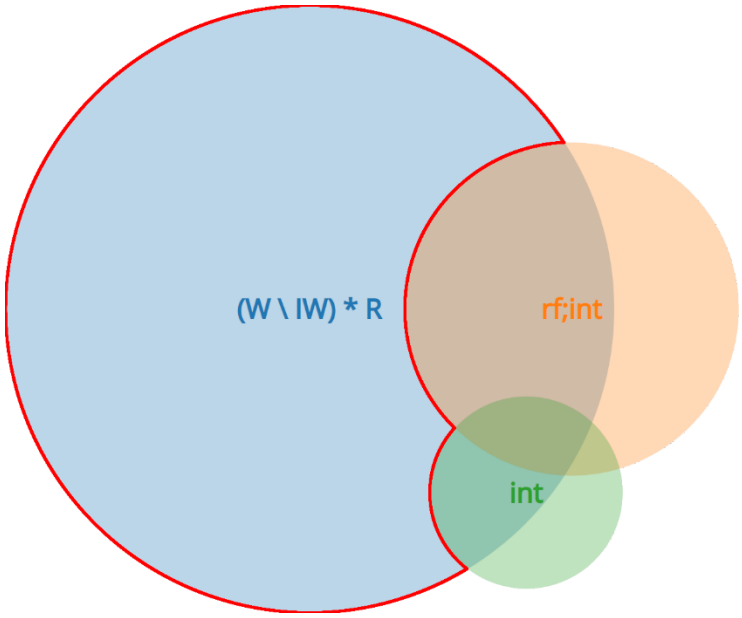
Az, hogy nem történik adatvesztés, ekvivalens azzal, hogy minden adatírást minden olvasó egyszer el fog olvasni. Erre a következő CAT leírást alkottam meg:

```
empty ((W \ IW) * R) \ (rf;int) \ int
```

Ez a modell a következőképpen zárja ki azokat a lefutásokat, amelyekben adatvesztés történik:

- (W \ IW) * R: Az összes adatírások közül azoknak, amelyek nem inicializáló írások voltak és az összes olvasásnak a Descartes-szorzata. Tehát a (W \ IW) * R kifejezés azon (w, r) él összefűzések halmaza, amelyben w eleme (W \ IW)-nek (nem inicializáló írások) és r eleme R-nek (olvasások).
- \ (rf;int): Kivonja az eddigiekből azt a halmazt, amely a rf (read from, vagyis azon élek egy halmaza, amely egy írás és a hozzá tartozó olvasás között megy, a valódi adatáramlást mutatja) után fűzött int (internal, vagyis azon élek egy halmaza, amelyek egy szálon belüli műveleteket végeznek) élekből áll.
- \ int: Kivonja azon élek halmazát, amelyek egy szálon belüli műveleteket reprezentálnak.
- empty: Azok a lefutások lesznek ennek a kritériumnak megfelelőek, melyben a sorban a további kritériumoknak megfelelő egyetlen él sem marad.

Vagyis ez egy olyan halmaz lesz, amelyben az összes nem inicializáló írásból induló és olvasásban végződő élek halmazából kivonjuk azokat, amelyek egy adott írásból egy olyan szálon lévő olvasásba mennek, amelyik szálon azt az írást már olvasták, illetve az egy szálon belüli éleket. Így ez a modell azt írja le, hogy minden írást olvasni fognak, hiszen ha lenne egy olyan írás, amit egy adott szálon nem olvasnak, akkor az ebből az írásból az adott szál felé menő élek megmaradnának, kivéve ha azon a szálon egyáltalán nem történt olvasás, de ilyenkor az nem lesz a W \ IW * R generált relációiban. A megértést a 6. ábra segíti.



6. ábra `(W \ IW) * R) \ (rf;int) \ int` szemléltetése

Ezen kívül az RTI DDS automatikusan garantálja a sorrendhelyességet, így azt is hozzá kell fűznünk a memóriamodellhez. A teljes modell a következőképpen néz ki:

```
1  include "cos.cat"
2  let fr = rf^-1; co
3  acyclic po | fr | rf | co as sc
4  empty ((W \ IW) * R) \ (rf;int) \ int
```

A feljebb olvasható kombináció egyszerre garantálja, hogy az írások időrendben érkeznek meg, illetve minden írás, amelyet valamelyik Publisher publikál, azt az összes aktív, kommunikációban jelen lévő Subscriber olvasni fogja.

4.5.2 Az adatvesztés kiküszöbölése érdekében használt QoS beállítások (Reliability QoS)

A DDS oldalról ahhoz, hogy ne tapasztaljunk adatvesztést, két QoS beállítást (illetve néhány erőforrás-korlátot) kell átállítani.

Itt fontos a Reliability QoS, ami abban segít, hogy ne tűnhessen el üzenet a kommunikáció során. Addig nem adja át a middleware az alkalmazásunknak az aktuális mintát, amíg az összes előzőt át nem adta. Ha ehhez szükséges, akkor az elveszetteket újrakéri. A hiba eliminálásához a Reliability QoS értékét `Reliable`-re kell állítanunk.

Egy másik fontos QoS beállítás a History Qos. Ezzel a QoS-sel azt lehet beállítani, hogy a Publisher milyen mélységig tartsa meg az elküldött mintákat. Ha a beállított mélység nem elég, akkor a Subsciber által újrakért adat lehet, hogy már nem található meg a Publisher lokális chace-ében, és ekkor el fogjuk veszteni az adatot. A History értékét `KEEP_ALL`-ra kell állítani, ezzel jelezve, hogy lehetőség szerint az összes írt mintát tartsa meg a Publisher.

Ezen kívül érdemes beállítani a `max_send_window_size` és a `min_send_window_size` értékét `DDS_LENGTH_UNLIMITED`-re, hogy az erőforráslimit ne írja felül a QoS-seket. A `send`

window egy olyan ablak az író oldalán, amely a még nem nyugtázott sample-ekből áll Reliable kommunikáció esetén. A max_send_window_size és a min_send_window_size egy olyan RTI DDS specifikus QoS beállítás, amellyel a send window méretét tudjuk befolyásolni. A minimumot szeretnénk végtelenre növelni, viszont a maximum nem lehet kisebb a minimumnál.

A demostrációhoz összesen 2 db QoS profile-t hoztam létre, ezek platformfüggetlenül ugyanazok.

Az egyik a 4.4.2 fejezetben is használt üres QoS profile. Ez a profile semmilyen garanciát nem vállal az adatokra, tehát azok a hálózat meghibásodása következtében el fognak veszni.

A másik egy olyan profile, amely a BuiltinQosLib::Generic.StrictReliable-ból származik (megtiltja az adatvesztést), és be van állítva rajta *Reliability* - RELIABLE_RELIABILITY_QOS, *History* - KEEP_ALL_HISTORY_QOS, ezen kívül a *max send window size* és *min send window size* DDS_LENGTH_UNLIMITED-re van állítva. A beállítás ide tartozó része a következő:

```
1 <qos_profile name="TestData_Profile"
  base_name="BuiltinQosLib::Generic.StrictReliable" is_default_qos="true">
2     <datawriter_qos>
3         <publication_name>
4             <name>TestDataDataWriter</name>
5         </publication_name>
6         <history>
7             <kind>KEEP_ALL_HISTORY_QOS</kind>
8         </history>
9         <reliability>
10             <kind>RELIABLE_RELIABILITY_QOS</kind>
11         </reliability>
12         <protocol>
13             <rtps_reliable_writer>
14                 <max_send_window_size>DDS_LENGTH_UNLIMITED
15                 </max_send_window_size>
16                 <min_send_window_size>DDS_LENGTH_UNLIMITED
17                 </min_send_window_size>
18             </rtps_reliable_writer>
19         </protocol>
20     </datawriter_qos>
21
22     <datareader_qos>
23         <subscription_name>
24             <name>TestDataDataReader</name>
25         </subscription_name>
26         <history>
27             <kind>KEEP_ALL_HISTORY_QOS</kind>
28         </history>
29         <reliability>
30             <kind>RELIABLE_RELIABILITY_QOS</kind>
31         </reliability>
32     </datareader_qos>
33     <participant_qos>
34         <participant_name>
35             <name>TestDataParticipant</name>
36             <role_name>TestDataParticipantRole</role_name>
37         </participant_name>
38     </participant_qos>
39 </qos_profile>
```

Látható, hogy a beállításban mind Publisher és Subscriber oldalról felül vannak definiálva a `Reliability` és a `History` beállítások, ezek értékei fontos, hogy megegyezzenek mindkét oldalt, mert ebben az esetben lesznek megfelelőek a kommunikáció QoS beállításai. A `max send window size` és a `min send window size` beállítások viszont csak Publisher oldalon szerepelnek, mivel csak ott van ennek a beállításnak szerepe. A beállítás hiánya esetén a Publisher nem tudja tárolni a még nem kézbesített üzeneteket (ha legalább néhány 100 ms-ig nincs hálózat), hiszen a Subscriber nem nyugtázza őket, és a QoS profile-ban, amiből leszármazunk, limitálva van a window size. Amennyiben a send window betelt, a Publisher ezeket az üzeneteket eldobja.

4.5.3 Mérési elrendezés

A mérés során egy Publisher és két Subscriber kommunikál egymással (a két Subscribernek nincs külön szerepe, azért vannak duplikálva, hogy lássuk, hogy a működés konzisztens, és több olvasó komponensnél is ugyanaz a probléma figyelhető meg).

Két tesztet is készítettem, amely a csomagvesztést igyekszik demonstrálni. Az egyik során a komponensek közti csomagokat átrendezzük, a másik során pedig teljesen eldobjuk. Mindkét futtatásra igaz, hogy két futtatást tartalmaznak: Az első futtatásnál az első teszt során is látható ''üres'' QoS profile-t használjuk, a második futtatás során pedig a biztonságosabb `RELIABILITY` beállítással ellátott QoS profile-t.

A mérési ötlet a következő: A Publisher egy szekvenciaszámot növel folyamatosan, fix, 100 milliszekundum időközönként, másodpercenként többször. A Subscriber ezt a számot olvassa, és ha az éppen érkező szám nagyobb, mint az előző szekvenciaszámot követő érték, akkor detektálja, hogy elveszett legalább egy változóírás.

A demonstrációhoz használt IDL a következő:

```
1  struct TestData
2  {
3      unsigned long num;
4  };
```

Az IDL file-ból láthatjuk, hogy egy darab előjel nélküli `long` típus van benne definiálva, amelyet szekvenciaszámként fog a Publisher írni a Subscriber pedig olvasni.

4.5.4 Futtató scriptek és programok

A futtatást az előzőekhez hasonlóan egy Bash script végzi. Ebben az esetben kettő Bash script is tartozik a tulajdonsághoz, mivel a problémát a hálózat megszakadása és a csomagok átrendeződése esetén is demonstráljuk. Ennek a teljes leírása megtalálható a B. Függelékben és a C. Függelékben, működési elve a következő:

- A DDS alkalmazás kódjának fordítása (4. sor)
- Deploy kép buildelése a docker file-ból (7.sor)
- Subscriber(ek) futtatása (10., 13. sorok)
- Publisher futtatása (16. sor)
- Google pingelése (20. sor)
- Publisher kimenő csomagjai 50 százalékának eldobása (25. sor)
- Billentyűlenyomásra várakozás (27. sor)
- Csomageldobási szabály törlése (30. sor)
- Billentyű eseményre várakozása (32. sor)
- Konténerek leállítása, törlése (35., 36., 37. sorok)
- Deploy kép buildelés a módosított QoS beállításokkal (42. sor)
- Subscriber(ek) futtatása (45., 48. sor)
- Publisher futtatása (51. sor)
- Google pingelése (55. sor)
- Publisher kimenő csomagjai 50 százalékának eldobása (60. sor)
- Billentyűlenyomásra várakozás (62 sor)

- Csomageldobási szabály törlése (65. sor)
- Billentyű eseményre várakozás (67. sor)
- Konténerek leállítása, törlése (70., 71., 72. sorok)

Ezen kívül fontos még bemutatni a Publisher és a Subscriber kódjának lényegi részét, mert ezeknek a komponenseknek az implementációja sem teljesen triviális.

A Publisher C nyelvű kódjának lényegi része:

```
1 for (count=0; (sample_count == 0) || (count < sample_count); ++count)
2 {
3     printf("Writing TestData, count %d\n", count);
4
5     instance->num = count;
6
7     /* Write data */
8     retcode = TestDataDataWriter_write(
9         TestData_writer, instance, &instance_handle);
10     if (retcode != DDS_RETCODE_OK)
11         fprintf(stderr, "write error %d\n", retcode);
12
13     NDDS_Utility_sleep(&send_period);
14 }
```

Itt láthatjuk, hogy a kódban egy cikluson iterálunk végig (sample_count == 0, tehát ez egy végtelen ciklus lesz), ahol minden körülforduláskor a következő események történnek:

- Konzolra írjuk a globális változó aktuális értékét (4. sor)
- Az elküldendő sample-be beleírjuk a jelenlegi értéket (6. sor)
- Megpróbáljuk végrehajtani a sample (globális változó jelenlegi értékének) írását (9. sor)
- Ellenőrizzük a visszatérési értéket, ha probléma volt, akkor kiírjuk a hibakódot (11., 12. sorok)
- Várakozunk 100 milliszekundumot (14. sor)

A Subscriber C nyelvű kódjának lényegi része:

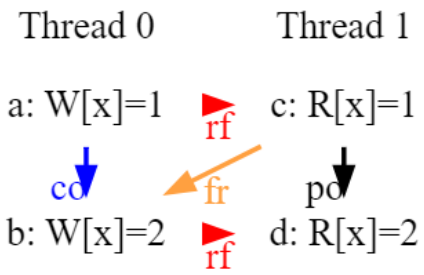
```
1 for (i = 0; i < TestDataSeq_get_length(&data_seq); ++i)
2 {
3     if (DDS_SampleInfoSeq_get_reference(&info_seq, i)->valid_data)
4     {
5         if(breaked == 0)
6         {
7             int jel_num = TestDataSeq_get_reference(&data_seq, i)->num;
8             printf("num: %d, last num was: %d\n", jel_num, last_num);
9             if(jel_num != last_num+1 && last_num != -1)
10             {
11                 breaked = 1;
12                 printf("Sequence number is incorrect!\n");
13                 break;
14             }
15             last_num = jel_num;
16         }
17     }
18 }
```

A Subscriber működése során olvassuk a globális változó értékét, a kódrészletben az adatok feldolgozása látható. Előfordulhat, hogy egyszerre több korábbi írás eredményét is látjuk (bizonyos esetekben nem feltétlenül csak a legutolsó írás eredményét látja a Subscriber), ezért kell egy ciklusban feldolgozni őket. A feldolgozás lépései a következők minden adatmintára:

- Megnézzük, hogy valid flag be van-e billentve, vagyis található-e adat a SampleInfo mellett (3. sor)
- Ellenőrizzük, hogy nem történt-e még leállító jelzés (5. sor)
- Kiolvassuk a sample-ben tárolt értéket (7. sor)
- Kiírjuk a kiolvasott értéket és az előző értéket a kijelzőre (8. sor)
- A hibás értéket onnan ismerjük fel, hogy korábban már olvastunk számot, és nem az azt követőt olvassuk. Ha ez történt, akkor kiírjuk, hogy hibás a kapott érték, és megszakítjuk a jelenlegi és az összes többi feldolgozást (9-14. sor)
- A legutóbb olvasott értéket tároló változóba beírjuk a most olvasottat (15. sor)

4.5.5 Eredmény, tapasztalat / Konklúzió

A tesztek során kiderült, hogy DDS-sel történő kommunikáció során adatok írása elveszhet, így erre a tulajdonságra ügyelnünk kell, amikor a komponensek kommunikációját tervezzük. Amennyiben ez nem jelent problémát az alkalmazás számára, akkor a `Reliability` és `History` QoS-ek értékének a beállítása nem feltétlen szükséges. Viszont, ha biztosítani szeretnénk, hogy minden írt adat megérkezik a másik oldalra, akkor a `Reliablility` QoS-nek *`Reliable`*, a `History` QoS-nek pedig *`KEEP_ALL_HISTORY`* értéket kell adnunk. Ebben az esetben a 4.5.1. fejezetben tárgyalt modellek lesznek jellemzőek a kommunikációra. Tehát a DDS feletti `Reliable` kommunikáció esetén minden írást kötelezően olvasnak, és ezek az írások-olvasások szálon (komponensen) belül csak a kódbeli sorrendben történhetnek meg. Ezt a kommunikációt a 7. ábra mutatja be, ahol láthatjuk, hogy az összes írást olvassák és az olvasások a programbeli utasítások sorrendjében történnek meg.



7. ábra Reliable kommunikáció lefutása

4.6 Adatkonzisztencia több író esetén

Több író esetén az adatkonzisztencia egy nagyon érdekes kérdés, hiszen esetleges hálózati kihagyások vagy kimaradások esetén nincs egyértelmű stratégia arra, hogy mi történjen az író komponens által már írt, de az olvasó komponensek által késve olvasott adatokkal.

- Historikus megközelítés: Ha valami 'log'-szerű adatunk van, és az egyik forrásból kis késéssel (pár száz milliszekundum, vagy akár pár másodperc) kapjuk meg, az még mindig sokkal jobb, mintha egyáltalán nem kapnánk meg az üzenetet, ezért jó stratégia lehet, ha az írt, de időben nem megérkezett adatokat elfogadjuk, és felhasználjuk később, amikor megérkeznek.
- Aktuális állapot megközelítés: Ez a megoldás akkor lehet kifizetődő, ha valaminek az aktuális állapotára vagyunk kíváncsiak. Ilyenkor annak, hogy eddig mi történt, nincs túl nagy jelentősége. Például, ha egy nagyobb rendszerben egy tartályon lévő szelep állapotát szeretnénk vezérelni két szenzor mérései alapján, akkor a szenzorok által jelenleg érzékelt értékek a fontosak, nem az, hogy 10 másodperccel ezelőtt mit érzékelt az egyikük. A megoldás lényege, hogy a késéssel megérkező üzeneteket eldobjuk, így azok nem kerülnek feldolgozásra, de cserébe nem „csúszik el" a két író üzenetei közötti szinkronitás.
- Egy további lehetséges megközelítés: Bizonyos esetekben szükségünk lehet arra a megoldásra, hogy az olvasó minden írást olvasson, de szigorúan abban a sorrendben, amilyenben az írások valójában megtörténtek. Ehhez egy adott érték olvasásakor az olvasónak tudnia kellene, hogy egy másik író írhatott-e olyat, amit még ő nem látott, de ennek kezelése csak nagy költségek árán lenne megoldható, és ezért DDS-sel nincs olyan QoS beállításkombináció, amivel ezt a működést el tudnánk érni.

4.6.1 Adatkonzisztencia memóriamodellje

Ez a QoS beállítás habár egy nagyon fontos beállítás, de a CAT nyelv hiányosságai miatt nem alkotható rá megfelelő formalizált memóriamodell. Sajnos CAT-ben – többek között – nem lehetséges késleltetés definiálása sem, így sehogyan sem tudnánk jellemezni azt a tulajdonságot, hogy hálózat megszakadás esetén az üzenetek nem érkeznek meg, de azok a hálózat visszakapcsolása után egyből kézbesítésre kerülnek. Ahhoz, hogy ez is megfogalmazható legyen, a CAT nyelvet ki kellene terjesztenünk, viszont ez jelen dolgozat keretein jóval túlmutatna.

4.6.2 Timestamp forrásának beállításához használt QoS beállítás (Destination_order QoS)

Ez a beállítás szabályozza, hogy az egyes Subscriberek hogyan értelmezik a több (akár különböző csomóponton futó) Publisher által írt adatpéldányok közül az utolsó értéket.

A QoS lehetséges értékei:

- A BY_RECEPTION_TIMESTAMP beállítás azt jelenti, hogy amennyiben az OWNERSHIP policy ezt lehetővé teszi, a legutolsó kapott érték az legyen, amelyet az adott Subscriber utoljára kapott meg. Ez az alapértelmezett érték.
- A BY_SOURCE_TIMESTAMP beállítás azt jelenti, hogy - feltéve, hogy az OWNERSHIP policy ezt lehetővé teszi – a Subscribernek a Publisherek által elhelyezett időbélyegeket kell használnia az utolsó üzenet meghatározása érdekében. Ez az egyetlen olyan beállítás, amely az azonos példányt frissítő egyidejű, azonos erősségű Publisherek esetében biztosítja, hogy több Subscriber esetén mindegyikük ugyanazt az aktuális értéket kapja egy adott változóhoz.

Fontos megjegyezni, hogy a szabvány részletesen nem tér ki az adatkonzisztenciára, illetve, hogy minden adat írás meg fog-e érkezni az olvasó komponensekhez!

4.6.3 Mérési elrendezés

A mérés során 2 db Publisher és 1 db Subscriber fog kommunikálni. A Publisherek egy `ID`-t (ez az adott Publisher `ID`-je) és egy szekvencia számot fognak publikálni, amely szekvenciaszámot 0-ról indítják, és külön-külön folyamatosan növelik.

Az előző IDL-ekhez képest bevezetésre került az `ID` mező, amely a Publishereket hivatott megkülönböztetni. Ez azért szükséges, mert az esetleges elcsúszások felderítése érdekében detektálni szeretnénk, hogy melyik szekvenciaszám honnan érkezett.

A kommunikáció során használt IDL file tartalma a következő:

```
1  struct TestData
2  {
3      int32 id;
4      unsigned long num;
5  };
```

Egy teszt abból áll, hogy két Publisher és egy Subscriber próbál DDS segítségével adatvesztésmentesen kommunikálni, ám a kommunikáció során az egyik Publisher csomagjait néhány másodpercig eldobjuk, majd ezt követően a csomagokat hiánytalanul továbbítjuk. Ez akár szimulálhat egy olyan kommunikációt, amelynek során a kapcsolat néhány másodpercre megszakad. A futtató script kétszer futtatja a tesztet, mindkét esetben különböző QoS beállításokkal. Az első esetben a Publisherek (írási) időbélyegeit vesszük alapul az üzenetek sorrendezéséhez, a másik esetben pedig a Subscriber (olvasási) időbélyegeit. A megbízható kommunikáció miatt a kapcsolat megszakadása alatt végzett írásokat újrakéri a Subscriber, de az első esetben érzékeli, hogy a másik Publisher már feldolgozott írásaihoz képest régi írásokról van szó. Ezért ezeket az alkalmazás részére már nem adja át. A második esetben a Subscriber oldalon nem érzékelhető a probléma, és minden írás feldolgozásra kerül. A két futtatás során egyszer a QoS beállítások miatt az

egyik Publishertől nem fog megérkezni az összes üzenet (csak a legfrissebb, a köztes írásokat eldobjuk), a másik esetben pedig meg fog érkezni az összes üzenet, ám azok a hálózat visszatérése után egyszerre érkeznek meg, így az üzenetek szinkronja egymástól elcsúszik.

4.6.4 Felhasznált QoS beállítások

Az ide tartozó QoS lényegi része a következő az első futtatásnál (Source timestamp):

```
1  <qos_profile name="TestData_Profile"
   base_name="BuiltinQosLib::Generic.StrictReliable" is_default_qos="true">
2      <datawriter_qos>
3          <publication_name>
4              <name>TestDataDataWriter</name>
5          </publication_name>
6          <history>
7              <kind>KEEP_ALL_HISTORY_QOS</kind>
8          </history>
9          <reliability>
10             <kind>RELIABLE_RELIABILITY_QOS</kind>
11         </reliability>
12         <protocol>
13             <rtps_reliable_writer>
14
   <max_send_window_size>DDS_LENGTH_UNLIMITED</max_send_window_size>
15
   <min_send_window_size>DDS_LENGTH_UNLIMITED</min_send_window_size>
16             </rtps_reliable_writer>
17         </protocol>
18         <destination_order>
19             <kind>DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS</kind>
20         </destination_order>
21     </datawriter_qos>
22
23     <datareader_qos>
24         <subscription_name>
25             <name>TestDataDataReader</name>
26         </subscription_name>
27         <history>
28             <kind>KEEP_ALL_HISTORY_QOS</kind>
29         </history>
30         <reliability>
31             <kind>RELIABLE_RELIABILITY_QOS</kind>
32         </reliability>
33         <destination_order>
34             <kind>DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS</kind>
35         </destination_order>
36     </datareader_qos>
37 </qos_profile>
```

Fontos megjegyezni, hogy ezekben az esetekben is fontos a `Reliability`, mivel ha nem Reliable lenne a kommunikáció, akkor a kapcsolat megszakadás alatti összes írás adatát elvesztenénk akár SOURCE_TIMESTAMP, akár RECEPTION_TIMESTAMP beállítást használnánk. Éppen ezért a fenti QoS profile-ban alkalmazzuk mindazt, amit a 0 fejezetben tárgyaltunk (az adatvesztés elleni beállítások, 6-16. sor és 27-32. sor), ezen

kívül definiálásra került egy eddigiekben nem látott QoS beállítás, a `DESTIONATION_ORDER` (18-20. sor és 33-35. sor). A QoS beállítás értéke az első futtatás során `DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS`, míg a második futtatás során pedig `DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS`.

4.6.5 Futtató script

A futtató script a D. Függelékben található, a működésének rövid összefoglalása:

- Megnézzük, hogy super user jogokkal fut-e a script, ha nem, akkor újraindítjuk, hogy azzal fusson (3-14. sor)
- Alkalmazás kódjának fordítása (19. sor)
- A Docker image Deploy képének elkészítése (22. sor)
- Jelenleg aktív ablak id-jának lekérése (25. sor)
- Subscriber futtatása (28. sor)
- Publisher(ek) futtatása (31., 34. sor)
- Pingelés indítása a ’google.com’ felé, hogy figyelemmel tudjuk kísérni a hálózat manipulációját (38. sor)
- Fókusz visszaállítása az eredeti terminal ablakra (43. sor)
- Publisher kimenő csomagjainak eldobása (48. sor)
- Néhány másodperces várakozás (50. sor)
- Csomageldobás megszüntetése (53. sor)
- Billentyű eseményre várakozás az első futtatás befejezéséhez (55. sor)
- Konténerek törlése (58-60. sor)
- A Docker image Deploy képének elkészítése a módosított QoS beállításokkal (65. sor)
- Subscriber futtatása (68. sor)
- Publisher(ek) futtatása (71.,74. sor)
- Pingelés indítása a ’google.com’ felé, hogy figyelemmel tudjuk kísérni a hálózat manipulációját (78. sor)
- Fókusz visszaállítása az eredeti terminal ablakra (83. sor)
- Publisher kimenő csomagjainak eldobása (88. sor)
- Néhány másodperces várakozás (90. sor)
- Csomageldobás megszüntetése (93. sor)
- Billentyű eseményre várakozás az első futtatás befejezéséhez (95. sor)
- Konténerek leállítása (98-100. sor)

A kódban van néhány kényelmi lépés, amely a futtatásnak nem kötelező része, viszont a tesztkörnyezetet sokkal felhasználóbarátabbá teszi. (Egy ilyen például a fókusz állítása a 43. sorban.)

4.6.6 Módosítások a programban

A tesztelésben használt program lényegi része ugyanaz, mint az eddig használt, apróbb módosítások lettek csak eszközölve, hogy meg tudjuk különböztetni a több író üzeneteit egymástól. A módosítások a következőek:

A Publisher oldalon az IDL módosítását átvezettem a kódba, az üzenet küldése előtt beírunk egy publisher `ID`-t is az üzenetbe.

Subscriber oldalon egy sokkal nagyobb módosításra került sor. Legfeljebb 10 Publishert feltételez a kód, ezért 10 Publisher utoljára megosztott értékét tároljuk az érkezett üzenetek közül. A program kód lényegesebb része megtalálható az E. Függelékben.

A Subscriber kódja a következőképpen épül fel:

- Változók deklarálása és inicializálása (1.-6. sor)
  - A Publisherektől utoljára olvasott értékeit tároló tömb (1. sor)
  - A jelenleg (valamelyik Publishertől) éppen olvasott érték változója (2. sor)
  - Éppen olvasott érték írója `ID`-jának változója (3. sor)
  - Flag, hogy megszakítottuk-e az adatok feldolgozását (4. sor)
  - Maximálisan megengedett adatelcsúszás két Publisher között (max_difference, 5. sor)
  - Két Publisher nem egyszerre történt indítása miatti kezdeti adatkülönbség (delta, 6. sor)
- Egy ciklussal végigiterálunk az összes feldolgozandó üzeneten (9-55. sor)
  - Megnézzük, hogy érvényes-e a kapott adat (11. sor)
  - Beállítjuk az Publisher `ID`-t és a jelenleg olvasott szekvenciaszámot (13-14. sor)
  - Megnézzük, hogy a Publisher ID beleesik-e a megengedett tartományba (17-18. sor)
  - Végigiterálunk a lehetséges Publishereken, keresve, hogy elcsúszott-e a szekvenciaszám (24-44. sor)
    - Ha az i. Publisher még nem írt üzenetet, vagy az i. Publisher a jelenlegi Publisher (nem az egy Publisher általi üzenetek kimaradását, átrendeződését vizsgáljuk), vagy az i. Publisher által írt üzenet kisebb szekvenciaszámú, mint a jelenlegi üzenet (előrefelé nem tud elcsúszni a szekvencia szám), akkor ugrunk a következő iterációra. (26-27. sor)
    - Ellenőrizzük, hogy a különbség a jelenlegi Publisher által publikált szekvenciaszám és az i. Publisher utolsó írása között meghaladja-e a max_difference deltával eltolt értékét. Ha igen, akkor valamilyen írást nem olvastunk. (29. sor)
    - Ha a jelenlegi Publisher még egyetlen adatot sem írt, akkor nem állunk le (31-36. sor)
      - Ilyenkor, ha a jelenlegi (új) Publisher és az i. Publisher által írt érték nagyobb, mint a delta, akkor beállítjuk a delta értékét erre (34. sor)
    - Kiírjuk, hogy a Publisherek szekvenciaszáma elcsúszott, és leállunk (37-39. sor)
    - Ellenkező esetben, hogyha a különbség mínusz a delta még kisebb a maximálisan megengedettnél, de a különbség nagyobb a deltánál, akkor a delta értékét ehhez a különbséghez igazítjuk. Ezzel

kezeljük a Publisherek sebességkülönbségéből adódó esetleges folyamatos csúszást (41-44. sor)
    - Ellenőrizzük, hogy a kapott szekvenciaszám eggyel nagyobb-e az előzőnél (48. sor)
    - Ha nem, akkor kiléptetjük a Subscribert, és jelezzük, hogy eltérést tapasztaltunk (50-52. sor)
    - A legutóbb olvasott értéket tároló tömbbe beírjuk a most olvasottat (54. sor)

4.6.7 Eredmény, tapasztalat / Konklúzió

Mivel a szabvány a `Destination_order` QoS pontos jelentését nem részletezi, ezért ez a mérés különösen érdekes. A tapasztalatokból az látszik, hogy az RTI cég megvalósítása úgy működik, hogy az adatot olvasó oldalon a middleware (a DDS subscriber) több adat író esetén kétféle logika szerint is sorrendezi a változó frissítéseket:

- Egyrészt az azonos író általi frissítéseket sorrendezi a már említett sorszámozás alapján, ez a sorszámozás az alapja a meg nem érkezett frissítések újrakérésének is.
- Másrészt a különböző írók általi írásokat sorrendezi az időbélyegek alapján. Az időbélyeg a `Destination_order` QoS értékétől függően lehet az adat írásának időbélyege (`BY_SOURCE_TIMESTAMP`) vagy a frissítésnek az olvasóhoz érkezésének az időbélyege (`BY_RECEPTION_TIMESTAMP`).

Hálózati hibák esetén fontos, hogy az újraküldött frissítések természetesen az eredeti sorszámmal és az esetleges eredeti forrásoldali időbélyeggel kerülnek elküldésre, de a valós megérkezésük céloldali időbélyegét kapják. Ha egy másik író későbbi frissítését olvasta már az alkalmazás, akkor az újraküldött (de időbélyege szerint régi) frissítést már nem fogja megkapni. A DDS middleware ezzel őrzi meg az adatok feldolgozásának sorrendhelyességét.

Ebből következik az, hogy forrásoldali időbélyegezés esetén a megbízható kommunikáció ellenére sem tud minden frissítést olvasni az alkalmazás olvasó komponense, míg céloldali időbélyegezés esetén tud, csak ott a különböző írók általi frissítések furcsán el tudnak csúszni egymástól. Ilyenkor a hálózat helyreállása után az egyik író által újraküldött több frissítés is egyszerre bukkan fel az olvasó számára, de esetleg egy másik írótól származó későbbi frissítések után.

Ennek a memóriamodellnek az időbélyegezés alapú megközelítését nem tudja a CAT nyelv a jelenlegi formájában kezelni.

5 Scriptek

Az egyszerűség és időzíthetőség kedvéért a tesztkörnyezet Docker képének buildelését, a konténer futtatását és a hálózati hibák szimulálását scriptek végzik. A scriptek segítségével az emberi lassúság és pontatlanság tényezőit zárhatom ki, és a megismételhetőséget biztosíthatom.

Kezdetben a tesztkörnyezetet Windows rendszer alatt szerettem volna megvalósítani, de a megfelelő hálózat manipulációs eszközök hiányában ez nem minden esetben volt lehetséges. A Windows alapú tesztkörnyezet működését is bemutatom, mivel ha valaki Windows alatt szeretné reprodukálni a teszteket, akkor az Adatvesztés kiküszöbölésén (4.5) alapuló tesztet reprodukálni tudja. Ennek módja, hogy a 4.5-es fejezetben meghatározottak szerint futtatja a 5.1-es fejezetben található scriptekkel.

A Windows alapú Adatvesztéses scripten kívül az összes demonstráló script Linuxon fut. Az előbb említett Adatvesztéses script továbbfejlesztett változata megtalálható az Adatvesztéses kommunikációt tárgyaló (4.5) fejezetben, ami ugyanazt az eredményt produkálja, viszont a használata jóval egyszerűbb.

5.1 Windowson futtatható teszt

Ebben a fejezetben a Windowson futtatható tesztet fogom bemutatni. Mint korábban írtam, Windowson csak a csomagvesztést demonstráló script működik, mivel a TC-hez szükséges kernel komponensek nincsenek a WSL-ben implementálva. Itt a csomagvesztést a Docker network-ből való lecsatlakoztatással érem el.

5.1.1 Docker deploy képet futtató script Windowson

A futtatás egyszerű, a Publisher futtatásához csak a `docker run --rm --name publisher --network=test -it myapp-deploy TestData_publisher` utasítást kell kiadni (itt a `test` egy docker virtuális hálózatot jelöl), de ezt sem szeretnénk kézzel elvégezni. Azért, hogy aszinkron tudjunk indítani konténereket, egy külön script kell, amely indítja őket. A következő scriptet írtam ezen célból (a Publisher és a Subscriber script felépítése ugyanaz, csak a futtatott bináris és a kép nevében különbözik):

```
1 @ECHO OFF
2
3 REM Starting Publisher
4 docker run --rm --name publisher --network=test -it myapp-deploy
   TestData_publisher
```

Ezeket a scripteket minden esetben a `batch` nyelven íródott főscriptek futtatják, mint például amit a 5.1.2-es fejezetben bemutatott script 10-es, 13-as, 16-os, 36-os, 39-es, és 42-es sora is bemutat.

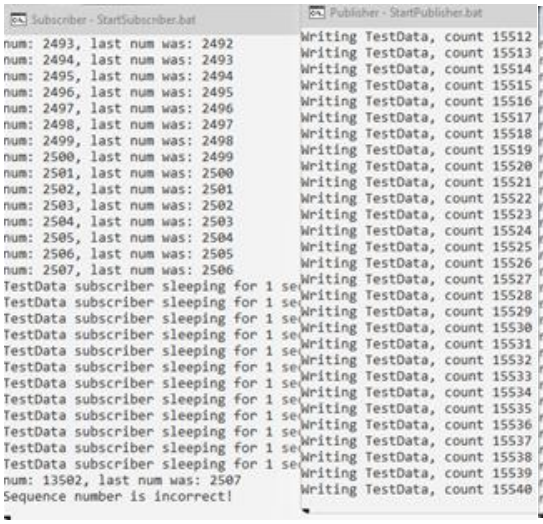
5.1.2 A csomagvesztést demonstráló script Windows alatt

A csomagvesztéses kommunikáció tesztelésére a következő scriptet írtam:

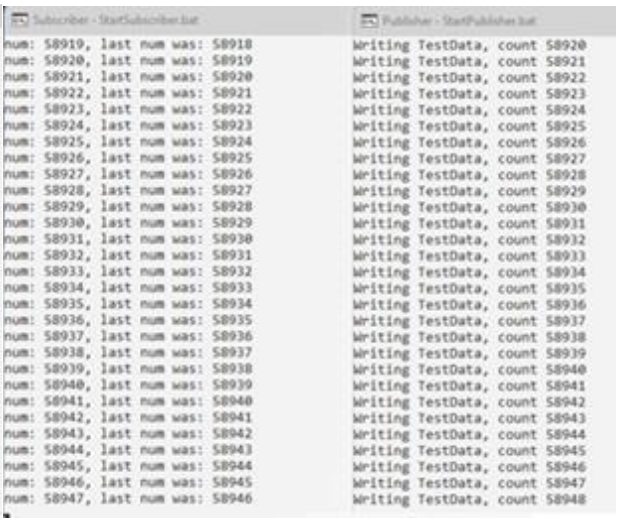
1. Kód fordítása (4. sor)

2. Deploy kép buildelés (7. sor)
3. Subscriber(ek) futtatása (10., 13. sor)
4. Publisher futtatása (16. sor)
5. Subscriber hálózati kapcsolatának lecsatlakoztatása (20. sor)
6. 10 másodperces várakozás (21. sor)
7. Subscriber visszacsatlakoztatása (24. sor)
8. Billentyű eseményre várakozás (25. sor)
9. Konténerek leállítása (28. sor)
10. Deploy kép buildelés a módosított QoS beállításokkal (33. sor)
11. Subscriber(ek) futtatása (36., 39. sor)
12. Publisher futtatása (42. sor)
13. Subscriber hálózati kapcsolatának lecsatlakoztatása (46. sor)
14. 10 másodperces várakozás (47. sor)
15. Subscriber visszacsatlakoztatása (50. sor)
16. Billentyű eseményre várakozás (51. sor)
17. Konténerek leállítása (54. sor)

A script kódja az M. Függelékben található.



8. ábra A teszt eredménye alap QoS beállításokkal



9. ábra A teszteredménye RELIABLE kommunikáció esetén

Ezen script segítségével bemutatható és könnyen belátható, hogy RELIABLE kommunikáció nélkül adatvesztés tapasztalható (8. ábra) hálózatszakadás esetén, míg RELIABLE kommunikáció esetén nincs adatvesztés (9. ábra).

5.2 Linuxon futtatható tesztek

A Linuxon nagy könnyebbséget jelentett a TC program használata, sokkal egyszerűbben és sokoldalúan lehet vele hálózati anomáliákat szimulálni.

5.2.1 Docker deploy képet futtató script Linuxon

A futtatás egyszerű, a Publisher futtatásához csak a `sudo docker run --rm --name publisher --cap-add=NET_ADMIN -it myapp-deploy TestData_publisher` utasítást kell kiadni, de ezt sem szeretnénk kézzel elvégezni. A `--cap-add=NET_ADMIN` paraméterre azért van szükség, mert a Docker konténerek alapból nem kapnak jogot hozzáférni a hálózati interfészükhöz, így a konténerből nem tudnánk a kapcsolatot befolyásolni. Ezen paraméter segítségével viszont a TC parancs hiba nélkül működik, a konténerek hozzáférnek az interfészhez. Azért, hogy aszinkron tudjunk indítani konténereket, egy külön script kell, amely indítja őket.

Ezeket a scripteket minden esetben a `bash` nyelven íródott főscriptek futtatják. Egy egyszerű példa egy ilyen teszt scriptre a 5.2.2-es fejezetben bemutatott script, amely demonstrálja, hogy bizonyos QoS beállítások mellett (`RELIABLE` kommunikáció) adatvesztés nem következhet be.

```
1 #! /bin/bash
2
3 # Starting Publisher
4 sudo docker run --rm --name publisher --cap-add=NET_ADMIN -it myapp-deploy
   TestData_publisher
```

A következő scriptet írtam ezen célból (a Publisher és a Subscriber script felépítése ugyanaz, csak a futtatott bináris és a kép nevében különbözik):

5.2.2 A csomagvesztést demonstráló script Linuxon

A csomagvesztéses kommunikáció tesztelésére a következő scriptet írtam:

1. Kód fordítása (4. sor)
2. Deploy kép buildelés (7. sor)
3. Subscriber(ek) futtatása (10., 13. sorok)
4. Publisher futtatása (16. sor)
5. Google pingelése (20. sor)
6. Publisher kimenő csomagjai 50 százalékának eldobása (25. sor)
7. Billentyű lenyomásra várakozás (27. sor)
8. Csomageldobási szabály törlése (30. sor)
9. Billentyű eseményre várakozás (32. sor)
10. Konténerek leállítása, törlése (35., 36., 37. sorok)
11. Deploy kép buildelés a módosított QoS beállításokkal (42. sor)
12. Subscriber(ek) futtatása (45., 48. sor)
13. Publisher futtatása (51. sor)
14. Google pingelése (55. sor)
15. Publisher kimenő csomagjai 50 százalékának eldobása (60. sor)
16. Billentyű lenyomásra várakozás (62 sor)
17. Csomageldobási szabály törlése (65. sor)
18. Billentyű eseményre várakozás (67. sor)
19. Konténerek leállítása, törlése (70., 71., 72. sorok)

A script kódja a N. Függelékben található.

5.3 Tesztkörnyezet scriptek konklúzió

A fejezetben láthattuk, hogy hogyan, milyen scriptekkel tudjuk futtatni a tesztkörnyezetet, és ezek a scriptek hogyan működnek. A teszteket érdemes Linuxon megfogalmazni és futtatni, mivel Linux alatt sokkal több eszköz áll rendelkezésre a hálózat manipulálásához.

6 Tesztkörnyezet létrehozása, működtetése

Ebben a fejezetben a tesztkörnyezet létrehozásának pontos lépéseit és működését fogom részletezni.

6.1 Docker konténerek készítése

Több lépés is szükséges a végleges tesztkörnyezet eléréséhez. Létre kell hoznunk egy Docker build képet, amely a DDS alkalmazások fordítását fogja végezni, majd egy Docker deploy képet, amely a lefordított binárisok futtatásáért felelős, és egy scriptet, amely futtatni fogja a teszteket. Ez a fejezet a Docker képek készítésének folyamatát, míg a következő a scriptek működését fogja bemutatni.

6.1.1 DDS Docker build kép létrehozása

Első lépésként egy DDS implementáció buildelésére alkalmas Docker képet kellett alkotnom, mely a Publisherek és Subscriberek kódjából egy targeten (jelen esetben egy másik Docker kép) futtatható binárist fordít. A Docker kép elkészítéséhez egy az RTI által készített tutorialt [14] használtam. Lényegében kellett egy x64 Linux host és target RTI Connext DDS telepítő file és egy license file, majd a következő Dockerfile-lal el is lehetett kezdeni az image buildelését:

A Dockerfile egy Ubuntu 20.04-es képből indul ki (4. sor), majd ezen telepít egy Linuxos host file-t és egy szintén Linuxos target file-t (10.-20. sor), amiket átmásol egy teljesen friss képre (25.-26. sor). Ezen a képen beállítja a szükséges környezeti változókat (29.-30. sor), hozzáadja a Licensz file-t (33. sor), és telepíti a build-essential csomagot (36. sor). Azért van szükség egy másik képbe másolni a dolgokat egy korábbiból, mert így csak a szükséges file-ok kerülnek be, a végleges kép mérete sokkal kisebb lehet.

```
# Ubuntu 20.04 is supported by RTI Connext DDS 6.1.1
# Use Docker's multi-stage build capability to create an image that
# contains only Ubuntu and the fully-installed RTI Connext distribution
FROM ubuntu:20.04 as dds_install

# /app will hold our installation files
WORKDIR /app

# Copy the installation files into the image
COPY rti_connext_dds-6.0.1-pro-host-x64Linux.run \
     rti_connext_dds-6.0.1-pro-target-x64Linux4gcc7.3.0.rtipkg \
     ./

# Run the host package installer
RUN chmod +x rti_connext_dds-6.0.1-pro-host-x64Linux.run && \
    ./rti_connext_dds-6.0.1-pro-host-x64Linux.run --mode unattended

# Run the target package installer
RUN /opt/rti_connext_dds-6.0.1/bin/rtipkginstall \
    -u rti_connext_dds-6.0.1-pro-target-x64Linux4gcc7.3.0.rtipkg

# Start from a fresh image
FROM ubuntu:20.04
# Copy the RTI Connext DDS installation from the image created above
COPY --from=dds_install \
     /opt/rti_connext_dds-6.0.1 /opt/rti_connext_dds-6.0.1

# Set up the DDS execution environment
ENV NDDSHOME /opt/rti_connext_dds-6.0.1
ENV PATH $PATH:$NDDSHOME/bin

# Install the license file
COPY rti_license.dat /opt/rti_connext_dds-6.0.1

# Install build tools
RUN apt-get update && apt-get -y install build-essential
```

Fontos, hogy a Dockerfile könyvtárába másoljuk be a host, a target és a license file-okat, majd a `docker build -t dds-build .` parancsot kiadva készíthetjük el a build képet.

6.1.2 DDS docker deploy kép készítése

A következőkben egy DDS Docker deploy képet készítettem, amely lehetővé teszi, hogy a rendszerbe egy újabb Publisher és Subscriber csatlakoztatása gyorsan, egy parancs futtatásával elérhető legyen.

Kezdetben létrehoztam egy IDL file-t, majd ebből le kellett generálnom a projektet, amivel a demonstrációt fogom végezni a C nyelvű API segítségével. A projektet a következő paranccsal generáltam le: `docker run -it --rm -v C:\...\container\myapp:/app -w="/app" dds-build rtiddsgen TestData.idl -ppDisable -language C -create typefiles -create examplefiles -create makefiles -platform x64Linux4gcc7.3.0`. Ez a parancs egy Docker konténert fog futtatni, amelyhez hozzálinkel egy mappát (`-v`) (és beállítja workdir-nek (`-w`)), majd egy `rtiddsgen` parancsot futtat, amely generálni fogja a projektet az összes generált DDS specifikus file-lal. Látható, hogy a `TestData.idl` file a struktúra leírónk, és x64 Linuxra dolgozunk (`-platform`) A parancs lefutása után a DDS

fájlok generálását végző konténer törlődik a `--rm` kapcsoló hatására. Ez azért fontos, mert minden fordításhoz egy új konténert szeretnénk létrehozni a Dockerfile alapján. Ha nem lenne ez a kapcsoló bekapcsolva, akkor a gépünk tele lenne felesleges, már nem használt konténerekkel.

A bemutatáshoz használt IDL file (amely a kommunikációra használt struktúra felépítését írja le) tartalma:

```
1 struct TestData
2 {
3     unsigned long num;
4 };
```

Látható, hogy a leírás egy struktúrát fog definiálni, amelyben egyetlen szám van. Ezen az egyszerű IDL file-on mutatom be a tesztkörnyezet elkészítését, mivel ezt néhány teszt során is használni fogjuk.

Ezután buildeltem a projektet a `docker run -it --rm -v C:\...\container\myapp:/app -w="/app" dds-build make -f makefile_TestData_x64Linux4gcc7.3.0` parancs segítségével. Ez a parancs szintén egy Docker konténert fog futtatni, amelyhez hozzálinkel egy mappát (és beállítja workdir-nek), majd egy `make` parancsot futtat, amely lebuildeli a projektet a `makefile_TestData_x64Linux4gcc7.3.0` file segítségével. A parancs lefutása után a buildelő konténer törlődik a `--rm` kapcsoló hatására.

A tesztkörnyezet működése során ezt a parancsot minden indításkor lefuttatom, hogy a kód legfrisebb verziójával dolgozzunk.

Létrehoztam 2db QoS profile file-t, az egyikben nincs semmi felülírva, a másikban pedig a RELIABILITY RELIABLE-re van állítva. Ezekhez a QoS profile-okhoz létrehoztam továbbá 1-1 Dockerfile-t, amellyel majd buildelni lehet a deploy képeket. (Erre azért volt szükség, mert más-más QoS fileokat használunk attól függően, hogy milyen viselkedést szeretnénk bemutatni. Nem minden demonstráció során pontosan ezeket fogom használni, ez csak egy példa, amivel ki lehet próbálni a tesztkörnyezet működését.)

Az egyik Dockerfile így néz ki:

```
1 FROM ubuntu:20.04
2
3 # Store the test apps in /app
4 WORKDIR /app
5
6 # Copy the apps and QoS file
7 COPY USER_QOS_PROFILES.xml \
8     objs/x64Linux4gcc7.3.0/TestData_publisher \
9     objs/x64Linux4gcc7.3.0/TestData_subscriber \
10     rti_license.dat \
11     ./
12
13 # Add the apps to the PATH
14 ENV PATH $PATH:/app
```

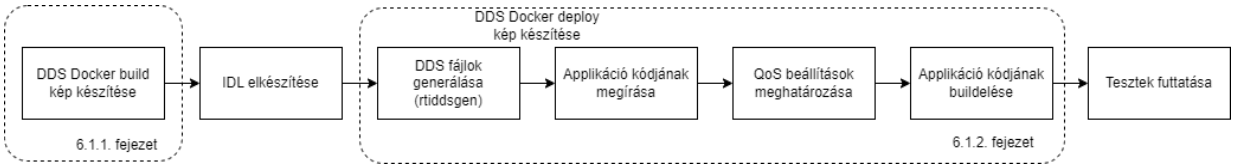
A fenti Dockerfile egy Ubuntu 20.04-es képből indul ki, majd ebben beállítja a `/app` könyvtárat munkakönyvtárként, majd ide bemásolja a futtatáshoz szükséges file-okat

(publisher binárisa, Subscriber binárisa, license file). A végén hozzáadja az elérési útvonalat a `PATH`-hoz.

Lényegében egy olyan kép fog létrejönni, amelybe bemásolja a legfrissebb fordított binárisokat, hogy az később a teszt közben futtatható legyen. A Dockerfile-okat a következő paranccsal lehet buildelni: „`docker build -t myapp-deploy .`”. Ez a parancs buildeli a `myapp-deploy` Docker képet a jelenlegi könyvtárban található Dockerfile segítségével.

6.2 A tesztkörnyezet felépítésének konklúziója, összefoglalása

Ebben a fejezetben láthattuk, hogy hogyan lehet DDS felett a tesztkörnyezetet felépíteni és működtetni. A felépítést a 10. ábra szemlélteti.



10. ábra Tesztkörnyezet felépítésének lépései

A lépések kötött sorrendben végzendőek el. Az egyes lépéseket röviden összefoglalva: Egy Build képet készítünk, amelyben a DDS applikációkat fogjuk tudni buildelni. Már ez is egy konténeren fog futni, amelyből minden indítás alkalmával újat hozunk létre. A build kép létrehozása után elkészítjük az IDL file-t, ami az átküldött adat struktúráját fogja leírni. A következőkben a deploy képet fogjuk elkészíteni, melynek során elsőnek legeneráltatjuk az `rtiddsgen` parancs segítségével a projektet, majd megírjuk az alkalmazás kódját, módosítjuk a QoS beállításokat, buildeljük az alkalmazást, és egy Docker image-t hozunk létre. A továbbiakban ezt a képet kell futtatnunk a tesztelés érdekében.

7 Kereszteződés, mint fizikai demonstrátor

Az eddigi "virtuális" tesztkörnyezetet követően elkészült egy "fizikai", kézzel fogható demonstrátor is, ami bemutatja, hogy csupán a rossz QoS beállítások is okozhatnak katasztrófát. A következőkben bemutatott kereszteződés modell ámbár egy valódi kereszteződés naív megvalósítása, viszont jól mintázza, hogy érdemes foglalkozni ezekkel a beállításokkal, hiszen DDS-t használnak több Airbus típusú repülőgépen is. Abban a kritikusabb környezetben egy ehhez hasonló kommunikációs hiba akár emberéleteket is követelhetne.

A demonstráció során egy valós kereszteződést modelljét fogom bemutatni, amely 4 darab közlekedési lámpából áll, amelyek páronként (egy párt képeznek az egymással szemközti lámpák) tartoznak egy zónába. Az egy zónába tartozó lámpák azonos színnel világítanak, a vezérlő hatására pedig színt válthatnak. Ez a rendszer azért egy nagyon jó példa a felhasználási területre, mert ez egy olyan biztonságkritikus rendszer, ahol hibának nincsen helye, és a működés helyességén emberéletek is múlhatnak akár.

A következőkben megpróbálom a fizikai demonstrátort is úgy bemutatni, mint a virtuális demonstrátorokat. A demonstrátor működésében nagyon sok a hasonlóság az adatvesztésről szóló (4.5) fejezethez képest, mivel itt is az adatvesztés lehetőségét „kihasználva” romolhat el a kommunikáció.

7.1 A kereszteződés kommunikációjára vonatkozó memóriamodell

A memóriamodell a két futtatás esetén nagyban hasonlít az „Adatvesztés mentes kommunikáció” esetére, mivel hasonló a megoldott probléma. Vagyis itt is az adatvesztést kell elkerülnünk, ami pedig ekvivalens azzal, hogy minden adatírást minden olvasó egyszer el fog olvasni. Erre a következő CAT leírás vonatkozik: `empty ((W \ IW) * R) \ (rf;int) \ int`.

Ezen kívül mint ahogy a 4.4. fejezetben láthattuk, az RTI DDS automatikusan garantálja a sorrendhelyességet, így azt is hozzá kell fűznünk a memóriamodellhez. A teljes modell a következőképpen néz ki:

```
1  include "cos.cat"
2  let fr = rf^-1; co
3  acyclic po | fr | rf | co as sc
4  empty ((W \ IW) * R) \ (rf;int) \ int
```

Mivel ugyanazt a tulajdonságot teszteljük a fizikai demonstrátor során, mint az „Adatvesztés mentes kommunikáció” esetén, ezért az itt bemutatott memóriamodell megegyezik a 4.5.1. fejezetben bemutatottakkal.

7.2 A kereszteződés során használt QoS beállítások

Mivel ugyanazt a tulajdonságot szeretnénk bemutatni, mint az „Adatvesztés mentes kommunikáció” során, ezért a QoS beállítások is meg fognak egyezni az ott használt QoS beállításokkal.

A QoS profile-ok gyors összefoglalása:

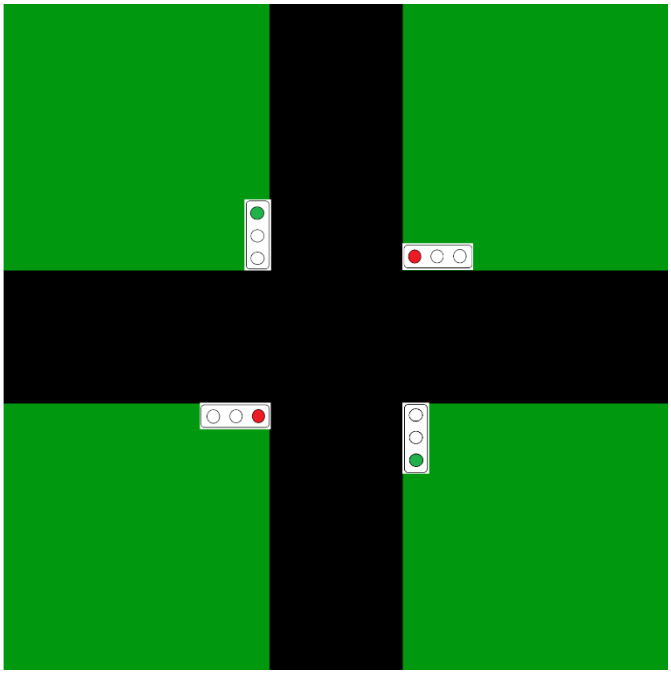
Az első tesztet egy üres QoS profile-lal végeztem, amely a `Generic.BestEffort` profile-ból származik, így nem lesz semmi garancia a színváltások megérkezésére. A QoS beállítással azt fogjuk demonstrálni, hogy megelőzhető, hogy a lámpa színének távoli állítása esetén a lámpa állapota nem feltétlen fog változni.

A második tesztet egy olyan QoS profile-al végeztem, amelyben felül van definiálva a `Reliability` és a `History` QoS beállítás. Ez az általunk készített QoS profile azért lesz érdekes, mert a csomagok elvesztését nem engedélyezi (ha egy változó írás elvész a hálózat kimaradása során, akkor azt a hálózat helyreállása után a middleware automatikusan újrakéri, így a Subscriberhez ugyan egy kis késleltetéssel, de el fog jutni).

A fentebb említett QoS beállítások megegyeznek a 0. fejezetben ismertetett QoS beállításokkal.

7.3 Mérési elrendezés

Az ötlet egy nagyon egyszerű kommunikáció megvalósítása volt, amely viszonylag könnyen meghibásodhat. Egy négy közlekedési lámpából álló kereszteződést szimulálunk, amelyben azt szeretnénk elérni, hogy bizonyos (hibás) QoS beállítások mellett lehessen egyszerre mind a négy lámpa zöld, más esetben viszont a kereszteződés kommunikációja más (helyes) QoS beállítások mellett ne tudjon meghibásodni, egyszerre ne lehessen két egymástól eltérő zónában lévő jelzőlámpa egyszerre zöld. A kereszteződés modelljét a 11. ábra mutatja be.



11. ábra Működő kereszteződés elképzelt modellje

Alapvetően 1 db vezérlő egység van, aki irányítja a lámpákat, ő lesz a Publisher és minden közlekedési lámpát egy darab végrehajtó egység képvisel, ez a végrehajtó egység a

Subscriber. Ha valamelyik lámpának váltania kell, akkor a vezérlő "utasítást küld" a lámpának (végrehajtó egységnek), hogy váltson az adott színre. A vezérlő 2 különböző "zónát" tud megkülönböztetni, amelyek ellentétes fázisban váltják a színeket (amikor az egyik zónában pirosak a lámpák, akkor a másik zónában zöldek és fordítva). Az egyes zónákban bárhány lámpa lehet, mivel a globális változókat akárhányan is olvashatják egyszerre. Tehát a lámpák "irányítását" a vezérlő végzi, a két zóna színeit bizonyos késleltetéssel váltogatja. A végrehajtó egységekben egy nagyon egyszerű logika van megvalósítva, miszerint ha a fogadó zóna száma megegyezik az ő számával, akkor feldolgozza az üzenetet, az üzenetben meghatározott értékre állítja a lámpát. Ha a fogadó zóna száma nem egyezik a végrehajtó egység id-jával, akkor az üzenetet nem fogja feldolgozni az alkalmazás.

A demonstráció során használt IDL a következőképpen néz ki:

```
1 enum lamp_state {
2  RED,
3  RED_AND_YELLOW,
4  GREEN,
5  YELLOW
6 };
7
8 struct CrossingData
9 {
10     unsigned long id;
11     unsigned long from;
12     unsigned long to;
13     lamp_state prompt;
14 };
```

Az IDL-ből kiderül, hogy ez a váz akár nagyobb rendszerekben is használható lenne, hiszen minden üzenet a rendszerben egyértelműen azonosítható a Publisher (tehát a küldő) `ID`-je (`from` mező) és azon belül a szekvencia szám (`ID` mező) alapján. Minden üzenetnek van egy címzett zónája (`to` mező). Ezen kívül a struct-ban még található egy `lamp_state` enum mező, amely az aktuális, kért állapotot fogja jelezni. Ez az enum C nyelven egy sima előjeles int-re fog lefordulni, amely biztosítja, hogy az ellenőrzése egyszerű legyen, és a kommunikáció során se jelentsen nagy overhead-et.

A lámpa váltásra való utasításnak 4 érték közül kell az egyiket felvennie:

- Piros – Piros jelzés, kezdetben ebből az állapotból indul az első zóna
- Piros és Sárga – A Piros utáni állapot, azt jelzi, hogy a lámpa hamarosan Zöld lesz
- Zöld – Zöld jelzés, kezdetben ebből az állapotból indul a második zóna
- Sárga – A Zöld jelzés, kezdetben ebből az állapotból indul a második zóna

A vezérlőtől érkező parancsok esetén a KRESZ-t követjük csak a normál működésre koncentrálva, viszont amennyiben elég rég nem küldött semmilyen üzenetet a vezérlő, abban az esetben villogó sárga módba váltanak a jelzőlámpák. A demonstrátor létrehozásánál ügyeltem rá, hogy az kívülről minél inkább hasonlítson egy valódi kereszteződés működésére.

Ez a modell lehetővé teszi, hogy nagyon egyszerűen tudjunk hibákat szimulálni a rendszeren. A demonstráció során egy rövid ideig tartó hálózati kimaradást szimulálunk. Ez a hálózati kimaradás akkor következhet be, amikor a Publisher az első zónának a "Sárga", majd a "Piros" jelzést küldi. A hálózati kimaradás valószínűsége 50% (ezt minden váltás előtt sorsoljuk, eldöntjük, hogy az üzeneteket szeretnénk-e manipulálni). Ha hálózati kimaradás történik, akkor mielőtt a központi logika (vezérlő) állítaná a lámpa színét, előtte beállítunk egy 100%-os csomageldobást, és amint az írás "megtörtént", ezt a beállítást visszavonjuk. A hiba injektálása azért így történik, hogy a hibák fennállása elég „hosszú” legyen, hogy megfigyelhető legyen és konzisztensen egymással szimuláljunk hibákat.

7.4 Fizikai megvalósítás

A megvalósításhoz felhasználtam egy korábbi projekt, a MODES3 [15] Smart City során készített lámpa modelleket (12. ábra), amelyeket 4 db Raspberry PI miniszámítógéppel hajtottam meg. A Raspberry-k vezeték nélküli (WiFi) hálózaton keresztül csatlakoznak egy routerhez, amely a csomagok elosztásáért felelős.

12. ábra Lámpa fizikai megvalósítása

Ezeken az eszközökön kívül csatlakozik a hálózathoz még egy vezérlő is (jelen esetben a laptopom), amely a lámpák átkapcsolásáért felelős. A megvalósítás során `Bash` scriptek juttatják el és indítják a binárisokat és a QoS profile file-okat.

A megvalósításnak egy fontos lépése – amit a futtatás során számos helyen kihasználok –, hogy a vezérlő IP címe a `192.168.1.100`, és az összes lámpát vezérlő számítógép a következő címeket kapja meg `192.168.1.{id + 100}` formátumban. Tehát 4 lámpát feltételezve a végpontok címei rendre: `192.168.1.101, 192.168.1.102, 192.168.1.103, 192.168.1.104.`

7.5 A kereszteződést futtató scriptek

A futtatás egy viszonylag nehéz feladatnak bizonyult, hiszen a lámpák a hardver kezelésének nehézségét is magukban hordozták, illetve a lámpákon futó alkalmazásokat 4 vezeték nélkül csatlakoztatott miniszámítógépre kell kitelepíteni. A futtatásnak több lépése is van, ezeken sorban fogok végig menni.

7.5.1 Kezdő beállítást végző script

Ez a script nagyon fontos, hiszen ő fogja beállítani a Raspberryket (ssh key beállítása, license fájl felmásolása, WiringPi telepítése, frissítés), ennek a scriptnek minden Raspberry-n le kell futnia, mielőtt bármit is futtatnánk rajta. A script kódja megtalálható az F. Függelékben.

A script működése:

- PI-k számának detektálása (7.-21. sor)
- PI-k egyesével történő beállítása (23.-51. sor)
    - Publikus SSH kulcs PI-re másolása (30. sor)
    - Licensz fájl feltöltése (33. sor)
    - WiringPI telepítése (36.-37. sor)
    - PI szoftverek verziójának frissítése a legújabbra (39.-43. sor)

7.5.2 Tesztelést futtató script

A script azért felelős, hogy a tesztet futtassa, a lámpákat irányítsa. A működése nagyban hasonlít az általános tesztkörnyezet scriptekre (mint amelyek a 4-es fejezetben is találhatóak). A tesztet a megszokottak szerint kétszer futtatjuk, a két futtatás során különböző QoS beállításokkal. A tesztet futtató script a G. Függelékben fellelhető. A Subscriberek futtatását egy külön script végzi, ennek a működését a 7.5.3. fejezetben tárgyaljuk.

Működés:

- Sudo jogok ellenőrzése (3.-14. sor)
- PI-k keresése a hálózaton (18.-33. sor)
- Tűzfal ideiglenes kikapcsolása a lámpa végrehajtó egységeivel való kommunikáció érdekében (36. sor)
- DDS alkalmazás kód fordítása (39. sor)
- Lefordított bináris konténerbe csomagolása a Dockerfile alapján (42. sor)
- Aktuális ablak ID-jának lekérdezése (csak kényelmi funkció, 45. sor)
- Esetlegesen futó Subscriberek leállítása az összes végrehajtó egységen (48.-52. sor)
- Binárisok kitelepítése a Deploy script segítségével (55. sor)
- Subscriberek elindítása a végrehajtó egységeken (58.-62. sor)
- Kereszteződést vezérlő Publisher elindítása (65. sor)
- Fókusz visszaállítása az fő ablakra (kényelmi funkció, 69. sor)
- Billentyű eseményre való várakozás (70. sor)
- Publisher leállítása, eltávolítása (73.sor)

- Bináris konténerbe csomagolása a Dockerfile alapján a megbízható QoS profilelal (78. sor)
- Subscriberek leállítása az összes végrehajtó egységen (81.-85. sor)
- Binárisok kitelepítése a Deploy script segítségével (88. sor)
- Subscriberek elindítása a végrehajtó egységeken (91.-95. sor)
- Kereszteződést vezérlő Publisher elindítása (98. sor)
- Fókusz visszaállítása az fő ablakra (kényelmi funkció, 102. sor)
- Billentyű eseményre való várakozás (103. sor)
- Publisher leállítása, eltávolítása (106. sor)
- Tűzfal visszakapcsolása (109. sor)
- Hálózati kommunikációt befolyásoló script által véletlenül aktívan maradt csomageldobás törlése (112. sor)

7.5.3 LampStart script

A lampStart scriptek felelősek a Subscriber alkalmazások indításáért. A script kódja a H. Függelékben olvasható. A működése a következő:

- Subscriber pingelése (3. sor)
- Ha a pingelés sikeres, akkor Subscriber leállítása, majd újra futtatása (4.-7.sor)

A pingelésre azért van szükség, mert ha a végrehajtó egység nem aktív, akkor ne is próbáljunk hozzá csatlakozni. Ha sikerült a pingelés, akkor indítsuk el a végrehajtó egységen az alkalmazás kódját.

7.5.4 Deploy script

A Deploy script felelős azért, hogy a futtatandó binárisok és a QoS profilok eljussanak a lámpa vezérlőkig. Ennek a scriptnek kettős célja van. Először is ha módosult az applikáció kódja, akkor abból az újonnan fordított változatot, illetve minden futtás során a megadott QoS profile-okat juttatja el a végrehajtó egységekhez. A script kódja megtalálható az I. Függelékben. A script működése a következő:

- Munkakönyvtárba való belépés (7. sor)
- DDS alkalmazás fordítása a megadott architektúrára crosscompile-lal (10. sor)
- Alkalmazás és QoS beállítás kitelepítése a paraméterként megadott számú célgépre (15.-23. sor)
    - IP cím összefűzése (17. sor)
    - Bináris másolása a célszámítógépre (19. sor)
    - Paraméterként kapott QoS beállítás másolása a célszámítógépre (21. sor)

7.5.5 Shutdown script

Ez a script felelős azért, hogy a Raspberry-k rendeltetésszerűen legyenek leállítva. Ha ezt a scriptet nem használnánk, akkor az esetleges hirtelen áramelvételtől a flash tartalma inkonzisztens állapotba kerülhetne, így meglenne az esélye, hogy a végrahajtó egységünk nem bootolna fel. Az ide tartozó kód megtalálható a J. Függelékben. A script a következőképpen működik:

- Leállítást számláló változó inicializálása (7. sor)
- Raspberry-k keresése ciklikusan, amelyik címen van elérhető, ott leállítjuk (9.-17. sor)
    - IP cím összefűzése (11. sor)
    - IP cím pingelése (12. sor)
    - Ha volt válasz, akkor leállítjuk a számítógépet, és növeljük a leállítást számláló változót (13.-16. sor)
- Leállított számítógépek számának kiírása (18. sor)

7.6 A kereszteződés demonstrátor működtető kódja

A következőkben a kereszteződés vezérlésére szolgáló applikáció kódját fogom bemutatni.

7.6.1 A végrehajtó egység kódja

A végrehajtó egység egy nagyon egyszerú logikát valósít meg. Ha kap egy olyan üzenetet, amiben a zóna száma megegyezik a saját zóna számával, akkor a benne található utasítást végrehajtja, ellenkező esetben az üzenetet eldobja. A végrehajtó egység alkalmazáskódjának itt releváns része megtalálható a K. Függelékben.

A végrehajtó egység kódja lényegi részének működése:

- Define-ok bevezetése a lámpák színének állítására, a GPIO lábak vezérlésére (1.-5. sor)
- Saját üzenetek feldolgozása (10.-31. sor)
- Összes új üzenet feldolgozása (39.-52. sor)
    - Üzenet megfelelőségének ellenőrzése (41. sor)
    - Saját üzenetek szűrése (44. sor)
    - Jelezzük, hogy a kontroller még aktív (47. sor)
    - Feldolgozzuk az üzenetet (49. sor)
- Feliratkozás az on_data_avaible eseményre (61.-62. sor)
- Main loop (70.-82. sor)
    - Ha a késleltetés véget ért, és van online controller, akkor a controller_online változót 0-ra állítjuk, majd várunk poll_period időt (72.-76. sor)
    - Ellenkező esetben a lámpát sárgán villogtatjuk (79. sor)
    - Várunk switch_period-nyi időt (81. sor)
- Leállítás, összes entitás törlése (85. sor)

A 72.-76.sor-ban található controller_online változó állítása arra szolgál, hogy detektáljuk, hogy van-e aktív publisher. Ha a controller_online változó értéke a várakkozás után 0, akkor a 79. sor fog végrehajtódni, és a végrehajtó egység villogó sárga jelzést fog küldeni a lámpának.

7.6.2 A vezérlő kódja

A vezérlő kódja a kereszteződés fő logikája, hiszen az itt lévő kód vezérli az összes lámpát, ő dönti el, hogy melyik lámpa milyen színt mutasson. A hálózat manipulációja is itt

található, itt szimuláljuk a hálózatszakadást a változó állítás előtti 100%-os csomageldobás segítségével. A vezérlő kódja a L. Függelékben található.

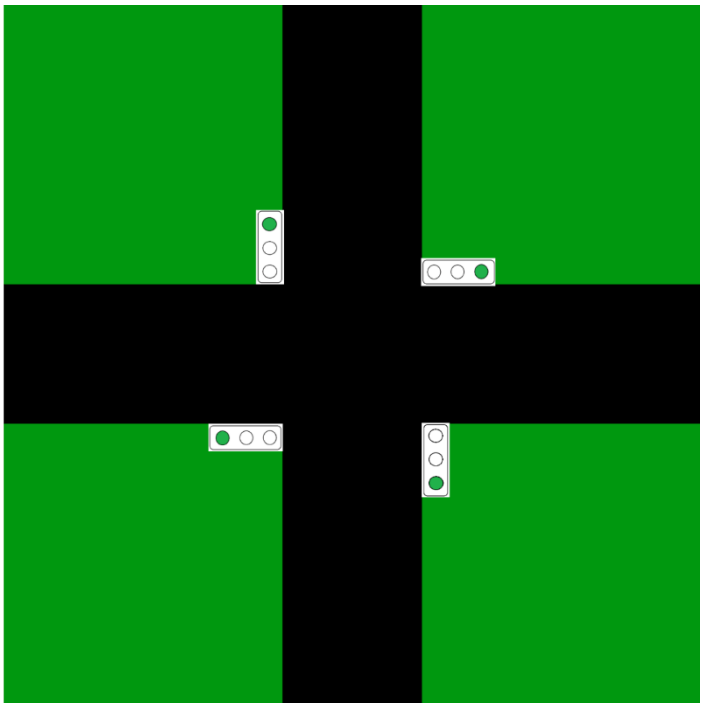
A vezérlő kódja lényegi részének működése a következő:

- Csomag eldobási szabály parancs definiálása (1. sor)
- Csomag eldobási szabályt törlő parancs definiálása (2. sor)
- Flag definiálás, az első zóna zöld színt mutat (4. sor)
- Lámpa zónák számának definiálása (5. sor)
- Flag definiálás, véletlen bit, hogy a jelenlegi ciklus futásában szeretnénk-e manipulálni a csomagok eljutását a Subscriberhez (6. sor)
- Flag definiálás, azt jelzi, hogy a jelenlegi ciklus futásában manipuláljuk-e a csomagok eljutását a Subscriberig (7. sor)
- Flag definiálás, azt jelzi, hogy a manipulálás sikerült-e (8. sor)
- Író szekvenciaszámának definiálása (9. sor)
- Publisher ID beállítása (10. sor)
- Késleltetések definiálása (11.-13. sor)
- Rövid (2mp-es) várakozás az indítás után a Subscriberek csatlakozása érdekében (16. sor)
- Fő ciklus, amiben a leállításig küldjük az üzeneteket (18.-97. sor)
    - Negáljuk a zónák színeit (20. sor)
    - Ciklus, melynek során minden zónának küldünk egy piros vagy egy zöld jelzést (22.-54. sor)
        - Eldöntjük, hogy a kimenő csomagokat kell-e manipulálni (25. sor)
        - Ha a manipulálás mellett döntöttünk, akkor beállítjuk a csomageldobási szabályt (26.-27. sor)
        - Kivárjuk a szabály aktivációs késleltetést (10ms, 28. sor)
        - Beállítjuk a kimenő struktúra zónaszámát és az üzenet ID-ját (30.-31. sor)
        - Beállítjuk a kiküldendő módosítás által kért színt (32.-35. sor)
        - Konzolra logoljuk a módosítás küldés kérést (37.-43. sor)
        - Elküldjük a színmódosítást (45.-49. sor)
        - Visszaállítjuk a hálózatot, ha manipulálva volt (52.-53. sor)
    - Növeljük az üzenet szekvenciaszámát (56. sor)
    - Kivárjuk a váltási késleltetést (20 mp, 57. sor)
    - Kisorsoljuk, hogy a következő menetben meg fog-e hibásodni a kommunikáció (59. sor)
    - Ciklus, melynek során az összes zónának küldünk egy sárga, vagy egy piros-sárga jelzést (61.-93. sor)
        - Eldöntjük, hogy a kimenő csomagokat kell-e manipulálni (64. sor)
        - Ha a manipulálás mellett döntöttünk, akkor beállítjuk a csomageldobási szabályt (65.-66. sor)
        - Kivárjuk a szabály aktivációs késleltetést (10ms, 67. sor)
        - Beállítjuk a kimenő struktúra zónaszámát és az üzenet ID-ját (30.-31. sor)
        - Beállítjuk a kiküldendő módosítás által kért színt (71.-74. sor)

        - Konzolra logoljuk a módosítás küldés kérést (76.-82. sor)
        - Elküldjük a színmódosítást (84.-88. sor)
        - Visszaállítjuk a hálózatot, ha manipulálva volt (91.-92. sor)
    - Növeljük az üzenet szekvenciaszámát (95. sor)
    - Kivárjuk a sárga utáni váltási késleltetést (2 mp, 96. sor)

7.7 Konklúzió

Azt tapasztalhatjuk, hogy abban az esetben, amikor a kommunikáció nem `Reliable`, akkor néhány színváltás írás "elvész", így a lámpa hibásan zöld marad. Azért maradhat a lámpa mindig zöld, mert ha a hálózat manipulációja mellett döntöttünk, akkor a sárga és a piros üzeneteket konzisztensen eldobjuk. Nem `Reliable` kommunikáció esetén ennek az állapotnak a fennállása legalább 20 másodperc lehet, hiszen addig nem történik állapotfrissítés, és a DDS nem vállal garanciát a kommunikációra. Ezt a jelenséget mutatja be a 13. ábra is. Ha `Reliable` QoS beállításokat használunk, akkor lehetséges úgy beállítani a kommunikációt, hogy az elveszett üzenetet újraküldje, és ezzel a hiba ne léphessen fel. Ebben az esetben a Publisherhez nem érkezik nyugta a Subscribertől (pont mint a TCP esetén), ezért az üzenetet újraküldi. Ebben az esetben az együttes zöld jelzés ideje a tényleges hálózati kimaradásnál csak milliszekudumokkal hosszabb, hiszen a hálózat helyreállása után a DDS middleware fog rájönni, hogy történt olyan írás, amelyet az olvasó még nem látott. Ennek a kiesett üzenet detekciójának az ideje elhanyagolható, így az inkonzisztens viselkedés intervalluma a hálózat kimaradási idővel jól közelíthető, ami jelen esetben néhány processzorutasítás ideje.



13. ábra Kereszteződés hibás kommunikáció esetén

8 Összefoglalás

Napjainkban elosztott rendszereket egyre elterjedtebben használnak akár kritikus területeken is, ahol a megbízhatóság és a helyes működés fontos szempontok. Dolgozatomban ezt a területet jártam körbe, és célom egy kritikus rendszerekben elterjedten használt kommunikációs protokoll, a DDS formális modelljének megalkotása volt. Ezt úgy valósítottam meg, hogy egy vizsgálati környezetben előre megtervezett teszteket végeztem, és a futtatás eredményeit megfigyelve, a hibákra adott viselkedéseket monitorozva elkészítettem a deklaratív formális reprezentációját a protokollnak.

A dolgozatom során elért elméleti eredményeim az alábbiak:

- Megvizsgáltam, hogy miként lehet az elosztott rendszerekben használt DDS kommunikációs köztesréteg/protokoll memóriamodell-alapú formális reprezentációját megalkotni.
- Formalizáltam a sorrendhelyesség, adatvesztés memóriamodelljét és körüljártam az adatkonzisztencia kérdését.

A dolgozatom során elért gyakorlati eredményeim a következők:

- Létrehoztam egy demonstrációs környezetet, melynek segítségével bárki ki tudja próbálni egy példakódon, hogy a megfelelő QoS beállítások mellett hogyan működik a komponensek kommunikációja.
- Készítettem egy fizikai demonstrátort, amin bemutatható, hogy miért fontos foglalkozni a QoS beállításokkal biztonságkritikus rendszerek esetén.
- Segítettem a mérnökök munkáját, hogy a formalizált modelleimet felhasználva kigenerálhassák a QoS beállítások szerinti írások-olvasások összes lehetséges egymásra hatását.

Fontos megjegyeznem, hogy habár a CAT nyelv nagyban segít, hogy megértsük a DDS kommunikációját, sajnos vannak korlátjai, és nem lehet a segítségével mindent leírni. Ez nagyban csökkentette a lehetséges QoS-sek CAT-beli megfogalmazásának lehetőségét.

Munkám eredményeképpen remélhetőleg a jövőben a vizsgált DDS implementációt használó mérnökök jobban előre fogják tudni jelezni az elosztott rendszereik viselkedéseit, továbbá képesek lesznek funkcionálisan helyes rendszerek megvalósítására.

9 Továbbfejlesztési lehetőségek

A munkám sok új irány előtt nyit utat és többféleképpen is felhasználható akár az ipar, akár az akadémia által. A lehetséges további irányok közül a fontosabbak:

- TC paranccsal több, más típusú hálózati anomália szimulálása
- Erőforráslimitek hatásának vizsgálata a rendszerre, akár ezek CAT-es leírása
- Automatizálás a kommunikáció memóriamodelljének leírására a tesztek alapján
- Tool támogatás a beállított garanciáknak megfelelő QoS profile generálására
- CAT nyelv kiterjesztése, hogy több QoS beállítás is leírható legyen

Függelék

A. Függelék Adat írás átrendeződés futtató shell script

Működés:

- Megnézzük, hogy super user jogokkal fut-e a script, ha nem, akkor újraindítjuk, hogy azzal fusson (3-14. sor)
- Kód fordítása (19. sor)
- Deploy kép buildelés (22. sor)
- Jelenleg aktív ablak id-jának lekérése (25. sor)
- Subscriber futtatása (28. sor)
- Publisher futtatása (31. sor)
- Pingelés indítása a ’google.com’ felé, hogy figyelemmel tudjuk kísérni a hálózat manipulációját (35. sor)
- Fókusz visszaállítása az eredeti terminal ablakra (41. sor)
- Publisher kimenő csomagjainak átrendezése (46. sor)
- Várakozás billentyűzet eseményre (48. sor)
- Csomagátrendezése szabály törlése (51. sor)
- Várakozás billentyűzet eseményre (53. sor)
- Konténerek törlése (56-57. sor)
- Subscriber futtatása (63. sor)
- Publisher futtatása (66. sor)
- Pingelés indítása a ’google.com’ felé, hogy figyelemmel tudjuk kísérni a hálózat manipulációját (70. sor)
- Fókusz visszaállítása az eredeti terminal ablakra (75. sor)
- Publisher kimenő csomagjainak eldobása (80. sor)
- Billentyű eseményre várakozása (82. sor)
- Csomageldobás megszüntetése (85. sor)
- Billentyű eseményre várakozása (87. sor)
- Konténerek leállítása (90-91. sor)

```
#! /bin/bash

# Check for sudo privileges
if [ "$USER" != "root" ]
then
    sudo -k                         # Ask for sudo password (Won't ask
again)
    if sudo true; then              # If has permission rerun the same
script with bash
        sudo bash $0
        exit                        # Exit if finished
    else                             # Else show error message
        echo "Please run this script as sudo!"
        exit 1
    fi
fi

BUILD_PATH=/home/ronald/Work/7.2023/TDK/student-radai-bsc

# Compile code
docker run -it --rm -v $BUILD_PATH/myapp:/app -w="/app" dds-build make -f
makefile_TestData_x64Linux4gcc7.3.0

# Build containers from code
docker build -t myapp-deploy $BUILD_PATH/myapp -f
$BUILD_PATH/myapp/Dockerfile

# Get the current terminal~s window id (require xdotool linux package)
main_terminal_window_id=$(xdotool getactivewindow)

# Start subscriber with xterm in the background
xterm -geometry 80x27+500+0 -e "$BUILD_PATH/LinuxTest/StartSubscriber.sh"
&

# Start publisher with xterm in the background
xterm -geometry 80x27+0+0 -e "$BUILD_PATH/LinuxTest/StartPublisher.sh" &
sleep 2

# Start pinging with xterm in the background to demonstrate network
emulation
xterm -geometry 80x27+1920+0 -e "docker exec -t publisher ping google.com"
&
echo Pinging google.com
sleep 5

# Set focus back to the main terminal window
sleep 1
xdotool windowactivate $main_terminal_window_id

sleep 4

# Reordering packets from publisher
docker exec -t publisher tc qdisc add dev eth0 root netem delay 1s 2s
echo Started reordering packets from publisher
read -p "Press Enter key to continue." < /dev/tty

# Stopping reordering packets from publisher
docker exec -t publisher tc qdisc del dev eth0 root netem delay 1s 2s
echo Stopped reordering packets from publisher
```

```
53 read -p "Press Enter key to continue." < /dev/tty
54
55 # Removing all containers
56 docker container rm -f publisher
57 docker container rm -f subscriber
58
59
60 # Running the test again with Data dropping profile
61
62 # Start subscriber with xterm in the background
63 xterm -geometry 80x27+500+0 -e "$BUILD_PATH/LinuxTest/StartSubscriber.sh"
   &
64
65 # Start publisher with xterm in the background
66 xterm -geometry 80x27+0+0 -e "$BUILD_PATH/LinuxTest/StartPublisher.sh" &
67 sleep 2
68
69 # Start pinging with xterm in the background to demonstrate network
   emulation
70 xterm -geometry 80x27+1920+0 -e "docker exec -t publisher ping google.com"
   &
71 echo Pinging google.com
72
73 # Set focus back to the main terminal window
74 sleep 1
75 xdotool windowactivate $main_terminal_window_id
76
77 sleep 4
78
79 # Disconnect publisher from network
80 docker exec -t publisher tc qdisc add dev eth0 root netem loss 50%
81 echo Started dropping some packets from publisher
82 read -p "Press Enter key to continue." < /dev/tty
83
84 # Reconnect publisher to network
85 docker exec -t publisher tc qdisc del dev eth0 root netem loss 50%
86 echo Stopped dropping some packets from publisher
87 read -p "Press Enter key to continue." < /dev/tty
88
89 # Removing all containers
90 docker container rm -f publisher
91 docker container rm -f subscriber
92
```

B. Függelék Adatvesztés tesztelését futtató script csomag eldobást használva

Az adatvesztéses kommunikáció tesztelésére írt csomag eldobást használó script:

- Kód fordítása (4. sor)
- Deploy kép buildelés (7. sor)
- Subscriber(ek) futtatása (10., 13. sorok)
- Publisher futtatása (16. sor)
- Google pingelése (20. sor)
- Publisher kimenő csomagjai 50 százalékának eldobása (25. sor)
- Billentyűlenyomásra várakozás (27. sor)

- Csomageldobási szabály törlése (30. sor)
- Billentyű eseményre várakozása (32. sor)
- Konténerek leállítása, törlése (35., 36., 37. sorok)
- Deploy kép buildelés a módosított QoS beállításokkal (42. sor)
- Subscriber(ek) futtatása (45., 48. sor)
- Publisher futtatása (51. sor)
- Google pingelése (55. sor)
- Publisher kimenő csomagjai 50 százalékának eldobása (60. sor)
- Billentyűlenyomásra várakozás (62 sor)
- Csomageldobási szabály törlése (65. sor)
- Billentyű eseményre várakozása (67. sor)
- Konténerek leállítása, törlése (70., 71., 72. sorok)

A script kódja:

```
#! /bin/bash

# Compile code
sudo docker run -it --rm -v ~/demo/myapp:/app -w="/app" dds-build make -f
  makefile_TestData_x64Linux4gcc7.3.0

# Build containers from code
sudo docker build -t myapp-deploy ~/demo/myapp -f $HO-
  ME/demo/myapp/Dockerfile

# Start subscriber with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartSubscriber.sh" &

# Start subscriber2 with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartSubscriber2.sh" &

# Start publisher with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartPublisher.sh" &
sleep 5

# Start pinging with xterm in the background to demonstrate network
  emulation
sudo xterm -e "docker exec -t subscriber ping google.com" &
echo Pinging google.com
sleep 5

# Disconnect publisher from network
sudo docker exec -t publisher tc qdisc add dev eth0 root netem loss 50%
echo Started dropping some packets from publisher
read -p "Press Enter key to continue." < /dev/tty

# Reconnect publisher to network
sudo docker exec -t publisher tc qdisc del dev eth0 root netem loss 50%
echo Stopped dropping some packets from publisher
read -p "Press Enter key to continue." < /dev/tty

# Removing all containers
sudo docker container rm -f publisher
sudo docker container rm -f subscriber
sudo docker container rm -f subscriber2

# Running the test again with qos profile

# Build containers from code
sudo docker build -t myapp-deploy ~/demo/myapp -f $HO-
  ME/demo/myapp/Dockerfileqos

# Start subscriber with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartSubscriber.sh" &

# Start subscriber2 with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartSubscriber2.sh" &

# Start publisher with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartPublisher.sh" &
sleep 5

# Start pinging with xterm in the background to demonstrate network
  emulation
```

```
55 sudo xterm -e "docker exec -t subscriber ping google.com" &
56 echo Pinging google.com
57 sleep 5
58
59 # Disconnect publisher from network
60 sudo docker exec -t publisher tc qdisc add dev eth0 root netem loss 50%
61 echo Dropping some packets from publisher
62 read -p "Press Enter key to continue." < /dev/tty
63
64 # Reconnect publisher to network
65 sudo docker exec -t publisher tc qdisc del dev eth0 root netem loss 50%
66 echo Stop dropping some packets from publisher
67 read -p "Press Enter key to continue." </dev/tty
68
69 # Removing all containers
70 sudo docker container rm -f publisher
71 sudo docker container rm -f subscriber
72 sudo docker container rm -f subscriber2
```

C. Függelék Adatvesztés tesztelését futtató script csomag átrendeződést használva

Az adatvesztéses kommunikáció tesztelésére írt csomag átrendeződést használó script:

- Kód fordítása (4. sor)
- Deploy kép buildelés (7. sor)
- Subscriber(ek) futtatása (10., 13. sorok)
- Publisher futtatása (16. sor)
- Google pingelése (20. sor)
- Publisher kimenő csomagjainak átrendezése (1s delay, 2s jitter)(25. sor)
- Billentyűlenyomásra várakozás (27. sor)
- Csomagátrendezési szabály törlése (30. sor)
- Billentyű eseményre várakozása (32. sor)
- Konténerek leállítása, törlése (35., 36., 37. sorok)
- Deploy kép buildelés a módosított QoS beállításokkal (42. sor)
- Subscriber(ek) futtatása (45., 48. sor)
- Publisher futtatása (51. sor)
- Google pingelése (55. sor)
- Publisher kimenő csomagjai nak átrendezése (1s delay, 2s jitter) (60. sor)
- Billentyűlenyomásra várakozás (62 sor)
- Csomagátrendezési szabály törlése (65. sor)
- Billentyű eseményre várakozása (67. sor)
- Konténerek leállítása, törlése (70., 71., 72. sorok)

A script kódja:

```
#! /bin/bash

# Compile code
sudo docker run -it --rm -v ~/demo/myapp:/app -w="/app" dds-build make -f
  makefile_TestData_x64Linux4gcc7.3.0

# Build containers from code
sudo docker build -t myapp-deploy ~/demo/myapp -f
  $HOME/demo/myapp/Dockerfile

# Start subscriber with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartSubscriber.sh" &

# Start subscriber2 with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartSubscriber2.sh" &

# Start publisher with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartPublisher.sh" &
sleep 5

# Start pinging with xterm in the background to demonstrate network
  emulation
sudo xterm -e "docker exec -t subscriber ping google.com" &
echo Pinging google.com
sleep 5

# Reordering packets from publisher
sudo docker exec -t publisher tc qdisc add dev eth0 root netem delay 1s 2s
echo Started reordering packets from publisher
read -p "Press Enter key to continue." < /dev/tty

# Stopping reordering packets from publisher
sudo docker exec -t publisher tc qdisc del dev eth0 root netem delay 1s 2s
echo Stopped reordering packets from publisher
read -p "Press Enter key to continue." < /dev/tty

# Removing all containers
sudo docker container rm -f publisher
sudo docker container rm -f subscriber
sudo docker container rm -f subscriber2

# Running the test again with qos profile

# Build containers from code
sudo docker build -t myapp-deploy ~/demo/myapp -f
  $HOME/demo/myapp/Dockerfileqos

# Start subscriber with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartSubscriber.sh" &

# Start subscriber2 with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartSubscriber2.sh" &

# Start publisher with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartPublisher.sh" &
sleep 5

# Start pinging with xterm in the background to demonstrate network
  emulation
```

```
55 sudo xterm -e "docker exec -t subscriber ping google.com" &
56 echo Pinging google.com
57 sleep 5
58
59 # Reordering packets from publisher
60 sudo docker exec -t publisher tc qdisc add dev eth0 root netem delay 1s 2s
61 echo Started reordering packets from publisher
62 read -p "Press Enter key to continue." < /dev/tty
63
64 # Stopping reordering packets from publisher
65 sudo docker exec -t publisher tc qdisc del dev eth0 root netem delay 1s 2s
66 echo Stopped reordering packets from publisher
67 read -p "Press Enter key to continue." < /dev/tty
68
69 # Removing all containers
70 sudo docker container rm -f publisher
71 sudo docker container rm -f subscriber
72 sudo docker container rm -f subscriber2
```

D. Függelék Adatkonzisztencia teszt script

- Megnézzük, hogy super user jogokkal fut-e a script, ha nem, akkor újraindítjuk, hogy azzal fusson (3-14. sor)
- Alkalmazás kódjának fordítása (19. sor)
- A Docker image Deploy képének elkészítése (22. sor)
- Jelenleg aktív ablak id-jának lekérése (25. sor)
- Subscriber futtatása (28. sor)
- Publisher(ek) futtatása (31., 34. sor)
- Pingelés indítása a ’google.com’ felé, hogy figyelemmel tudjuk kísérni a hálózat manipulációját (38. sor)
- Fókusz visszaállítása az eredeti terminal ablakra (43. sor)
- Publisher kimenő csomagjainak eldobása (48. sor)
- Néhány másodperces várakozás (50. sor)
- Csomageldobás megszüntetése (53. sor)
- Billentyű eseményre várakozás az első futtatás befejezéséhez (55. sor)
- Konténerek törlése (58-60. sor)
- A Docker image Deploy képének elkészítése a módosított QoS beállításokkal (65. sor)
- Subscriber futtatása (68. sor)
- Publisher(ek) futtatása (71.,74. sor)
- Pingelés indítása a ’google.com’ felé, hogy figyelemmel tudjuk kísérni a hálózat manipulációját (78. sor)
- Fókusz visszaállítása az eredeti terminal ablakra (83. sor)
- Publisher kimenő csomagjainak eldobása (88. sor)
- Néhány másodperces várakozás (90. sor)
- Csomageldobás megszüntetése (93. sor)
- Billentyű eseményre várakozás az első futtatás befejezéséhez (95. sor)
- Konténerek leállítása (98-100. sor)

```
#!/bin/bash

# Check for sudo privileges
if [ "$USER" != "root" ]
then
    sudo -k                             # Ask for sudo password (Won't ask again)
    if sudo true; then                  # If has permission rerun the same script with bash
        sudo bash $0
        exit                            # Exit if finished
    else                                # Else show error message
        echo "Please run this script as sudo!"
        exit 1
    fi
fi

BUILD_PATH=/home/ronald/Work/7.2023/TDK/student-radai-bsc

# Compile code
docker run -it --rm -v $BUILD_PATH/appreorder:/app -w="/app" dds-build make -f makefile_TestData_x64Linux4gcc7.3.0

# Build containers from code
docker build -t myapp-deploy $BUILD_PATH/appreorder -f $BUILD_PATH/appreorder/Dockerfile

# Get the current terminal~s window id (require xdotool linux package)
main_terminal_window_id=$(xdotool getactivewindow)

# Start subscriber with xterm in the background
xterm -geometry 80x27+1000+0 -e "$BUILD_PATH/LinuxTest/StartSubscriber.sh" &

# Start publisher with xterm in the background
xterm -geometry 80x27+0+0 -e "$BUILD_PATH/LinuxTest/StartPublisher.sh" 1 &

# Start publisher2 with xterm in the background
xterm -geometry 80x27+500+0 -e "$BUILD_PATH/LinuxTest/StartPublisher2.sh" 2 &
sleep 2

# Start pinging with xterm in the background to demonstrate network emulation
xterm -geometry 80x27+1920+0 -e "docker exec -t publisher ping google.com" &
echo Pinging google.com

# Set focus back to the main terminal window
sleep 1
xdotool windowactivate $main_terminal_window_id

sleep 4

# Disconnect publisher from network
```

```
48  docker exec -t publisher tc qdisc add dev eth0 root netem loss
  100%
49  echo Started dropping all packets from publisher
50  sleep 6
51
52  # Reconnect publisher to network
53  docker exec -t publisher tc qdisc del dev eth0 root netem loss
  100%
54  echo Stopped dropping all packets from publisher
55  read -p "Press ENTER key to continue." < /dev/tty
56
57  # Removing all containers
58  docker container rm -f publisher
59  docker container rm -f publisher2
60  docker container rm -f subscriber
61
62  # Running the test again with qos profile
63
64  # Build containers from code
65  docker build -t myapp-deploy $BUILD_PATH/appreorder -f
  $BUILD_PATH/appreorder/Dockerfileqos
66
67  # Start subscriber with xterm in the background
68  xterm -geometry 80x27+1000+0 -e
  "$BUILD_PATH/LinuxTest/StartSubscriber.sh" &
69
70  # Start publisher with xterm in the background
71  xterm -geometry 80x27+0+0  -e
  "$BUILD_PATH/LinuxTest/StartPublisher.sh" 1 &
72
73  # Start publisher2 with xterm in the background
74  xterm -geometry 80x27+500+0 -e
  "$BUILD_PATH/LinuxTest/StartPublisher2.sh" 2 &
75  sleep 2
76
77  # Start pinging with xterm in the background to demonstrate
  network emulation
78  xterm -geometry 80x27+1920+0 -e "docker exec -t publisher ping
  google.com" &
79  echo Pinging google.com
80
81  # Set focus back to the main terminal window
82  sleep 1
83  xdotool windowactivate $main_terminal_window_id
84
85  sleep 4
86
87  # Disconnect publisher from network
88  docker exec -t publisher tc qdisc add dev eth0 root netem loss
  100%
89  echo Started dropping all packets from publisher
90  sleep 6
91
92  # Reconnect publisher to network
93  docker exec -t publisher tc qdisc del dev eth0 root netem loss
  100%
94  echo Stopped dropping all packets from publisher
95  read -p "Press ENTER key to continue." < /dev/tty
```

```
96
97  # Removing all containers
98  docker container rm -f publisher
99  docker container rm -f publisher2
100 docker container rm -f subscriber
```

E. Függelék Adatkonzisztencia több író esetén program kód lényeges része

- Vátozók deklarálása és inicializálása (1.-6.sor)
  - Publisherek utolsó írt értékei (1.sor)
  - Aktuálisan jelenlegi iterációban Publishertől olvasott érték (2.sor)
  - Aktuális Publisher `ID`-ja (3.sor)
  - Flag, hogy megszakítottuk-e az adatok feldolgozását (4.sor)
  - Maximálisan megengedett adatelcsúszás két Publisher között (5.sor)
  - Két Publisher nem egyszerre történt indítása miatti maximális adatkülönbség (6.sor)
- Egy ciklussal végigiterálunk az összes feldolgozandó üzeneten (9.-55.sor)
  - Megnézzük, hogy érvényes-e a kapott adat (11.sor)
  - Beállítjuk az Publisher `ID`-t és a jelenleg küldött szekvencia számot (13-14.sor)
  - Megnézzük, hogy a Publisher ID beleesik-e a megengedett tartományba (17-18.sor)
  - Végigiterálunk a lehetséges Publishereken keresve, hogy elcsúszott-e a szekvencia szám (24-44.sor)
    - Ha az i. Publisher még nem írt üzenetet, vagy az i. Publisher a jelenlegi Publisher, vagy az i. Publisher által írt üzenet kisebb szekvenciaszámú, mint a jelenlegi üzenet (előrefelé nem tud elcsúszni a szekvencia szám), akkor ugrunk a következő iterációra (26-17.sor)
    - Ellenőrizzük, hogy a különbség a jelenlegi Publisher által publikált szekvenciaszám és az i. Publisher utolsó írása meghaladja a max_difference deltával eltolt értékét. (29.sor)
    - Ha a jelenlegi publisher még egyetlen adatot sem írt, akkor nem állunk le (31-36.sor)
      - Ha a jelenlegi publisher és az i.publisher által írt érték nagyobb, mint a delta, akkor beállítjuk a delta értékét erre (34.sor)
    - Kiírjuk, hogy a Publisherek szekvenciaszáma elcsúszott és leállunk (37-39.sor)
    - Ellenkező esetben megnézzük, hogy a detánál nagyobb-e a jelenlegi eltérés. Amennyiben igen, akkor frissítjük a delta értékét (41-44.sor)
  - Ellenőrizzük, hogy a kapott szekvenciaszám az előző szám eggyel inkrementált értéke-e (48.sor)
  - Ha nem megfelelő az érték, akkor kiléptetjük a Subscribert és jelezzük, hogy eltérést tapasztaltunk (50-52.sor)

- Beállítjuk a Publisher által küldött értéket a legfrisebb értékre (54.sor)

```
int last_num[10] = {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
int jel_num = -1;
int jel_publisher = -1;
int breaked = 0;
int max_difference = 20;
int delta = 0;


for (i = 0; i < TestDataSeq_get_length(&data_seq); ++i)
{
    if (DDS_SampleInfoSeq_get_reference(&info_seq, i)->valid_data)
    {
        int jel_num = TestDataSeq_get_reference(&data_seq, i)->num;
        int jel_publisher = TestDataSeq_get_reference(&data_seq, i)->id-1;

        // check if publisher has avaible id
        if(jel_publisher < 0 || jel_publisher >= 10)
            continue;

        if(last_num[jel_publisher] != -1)
            printf("Publisher%d\tnum: %d, last num was:
%d\n",jel_publisher+1, jel_num, last_num[jel_publisher]);


        for (int i = 0; i < 10; i++)
        {
            if(last_num[i] == -1 || i == jel_publisher || last_num[i] <
jel_num)
                continue;

            if (abs(last_num[i] - jel_num) - delta > max_difference)
            {
                if(last_num[jel_publisher] == -1)
                {
                    if(abs(last_num[i] - jel_num) > delta)
                        delta = abs(last_num[i] - jel_num);
                    continue;
                }
                breaked = 1;
                printf("Difference is too big between sequence numbers
(%d)!\n", abs(last_num[i] - jel_num));
                break;
            }
            else if (abs(last_num[i] - jel_num) > delta)
            {
                delta = abs(last_num[i] - jel_num);
            }
        }


        if(jel_num != last_num[jel_publisher]+1 && last_num[jel_publisher]
!= -1)
        {
            breaked = 1;
            printf("Sequence number is incorrect!\n");
            break;
        }
        last_num[jel_publisher] = jel_num;
    }
```

```
56 }
```

F. Függelék Kereszteződés Raspberry PI beállító script

Ez a script azért felelős, hogy a frissen telepített Raspberriket beállítja, hogy a tesztkörnyezet futtatható legyen rajtuk. Telepít egy olyan könyvtárat is, amely lehetővé teszi a GPIO pinek egyszerű, parancssorból történő állítását is.

Működése:

- PI-k számának detektálása (7.-21.sor)
- PI-k egyesével történő beállítása (23.-51.sor)
  - Publikus SSH kulcs PI-re másolása (30.sor)
  - Licensz fájl feltöltése (33.sor)
  - WiringPI telepítése (36.-37.sor)
  - PI szoftverek verziójának frissítése a legújabbra (39.-43.sor)

```
#!/bin/bash

# ------------------------------------------
# Setup Crossing Demo for raspberry pi
# ------------------------------------------

# Ask how many pi's are present in the Demo
echo "How many pi's do I need to setup?"
read piCount
clear

# Check if all pi's are connected to the network
for (( i=1; i <= $piCount; i++ ))
do
    ip_addr="192.168.1.$((i + 100))"
    ping -c 1 -W 1 $ip_addr > /dev/null 2>&1
    if [ $? -ne 0 ]; then
        echo "Lamp $i is not connected to the network! Please resolve the problem and try again."
        exit 1
    fi
done

# Iterate through all pi's in the network
for (( i=1; i <= $piCount; i++ ))
do
    echo -e "\nSettupping the $i. pi\n"
    ip_addr="192.168.1.$((i + 100))"

    # Adding public key to the pi's authorized_keys
    ssh-copy-id ronald@${ip_addr}

    # Uploading license file
    scp rti_license.dat ronald@${ip_addr}:.

    # Installing WiringPi and upgrading the pi
    ssh ronald@${ip_addr} "echo 'pass' | sudo -S apt-get update"
    ssh ronald@${ip_addr} "echo 'pass' | sudo -S apt-get install wiringpi -y"

    if [ $i -eq $piCount ]; then
        ssh ronald@${ip_addr} "echo 'pass' | sudo -S sud && echo 'pass' | sudo -S reboot now"
    else
        ssh ronald@${ip_addr} "echo 'pass' | sudo -S DEBIAN_FRONTEND=noninteractive apt-get -y upgrade && echo 'pass' | sudo -S reboot now" &
    fi

done
```

G. Függelék Kereszteződés demonstrációt futtató script

```
#!/bin/bash

# Check for sudo privileges
if [ "$USER" != "root" ]
then
```

```
6     sudo -k                          # Ask for sudo password (Won't ask
  again)
7     if sudo true; then               # If has permission rerun the same
  script with bash
8         sudo bash $0
9         exit                         # Exit if finished
10    else                             # Else show error message
11        echo "Please run this script as sudo!"
12        exit 1
13    fi
14 fi
15
16 BUILD_PATH=/home/ronald/Work/7.2023/TDK/student-radai-bsc
17
18 while :; do
19    for (( i=1; i <= 40; i++ ))
20    do
21        ip_addr="192.168.1.$((i + 100))"
22        ping -c 1 -W 1 $ip_addr > /dev/null 2>&1
23        if [ $? -ne 0 ]; then
24            PI=$((i-1))
25            break
26        fi
27    done
28    read -n1 -p "I detected $PI Lamps. Is it correct? [y,n]" ans
29    echo -e "\n"
30    if [[ $ans == "y" || $ans == "Y" ]]; then
31        break
32    fi
33 done
34
35 # Disable firewall to communicate outside of the host
36 ufw disable
37
38 # Compile code
39 docker run -it --rm -v $BUILD_PATH/crossing:/app -w="/app" dds-build make
  -f makefile_Crossing_x64Linux4gcc7.3.0
40
41 # Build containers from code
42 docker build -t myapp-deploy $BUILD_PATH/crossing -f
  $BUILD_PATH/crossing/Dockerfile
43
44 # Get the current terminal~s window id (require xdotool linux package)
45 main_terminal_window_id=$(xdotool getactivewindow)
46
47 # Stopping subscribers on remote lamps
48 for (( i=1; i <= $PI; i++ ))
49 do
50    ip_addr="192.168.1.$((i + 100))"
51    su -c "ssh ronald@$ip_addr \"pkill Crossing_subscr\"" ronald
52 done
53
54 # Deploying binaries with the QoS profiles
55 su -c "bash $BUILD_PATH/crossing/crosscompile/Deploy.sh
  $BUILD_PATH/crossing/USER_QOS_PROFILES.xml $PI" ronald
56
57 # Starting subscribers on remote lamps
58 for (( i=1; i <= $PI; i++ ))
59 do
```

```
60      ip_addr="192.168.1.$((i + 100))"
61      su -c "ssh ronald@$ip_addr \"./Crossing_subscriber $((i - 1))\" &"
    ronald
62  done
63
64  # Start publisher with xterm in the background
65  xterm -geometry 80x27+20+20 -e
    "$BUILD_PATH/LinuxTest/Crossing/StartCrossingPublisher.sh" 0 &
66
67  # Set focus back to the main terminal window
68  sleep 1
69  xdotool windowactivate $main_terminal_window_id
70  read -p "Press ENTER key to continue." < /dev/tty
71
72  # Removing all containers
73  docker container rm -f publisher
74
75  # Running the test again with qos profile
76
77  # Build containers from code
78  docker build -t myapp-deploy $BUILD_PATH/crossing -f
    $BUILD_PATH/crossing/Dockerfileqos
79
80  # Stopping subscribers on remote lamps
81  for (( i=1; i <= $PI; i++ ))
82  do
83      ip_addr="192.168.1.$((i + 100))"
84      su -c "ssh ronald@$ip_addr \"pkill Crossing_subscr\"" ronald
85  done
86
87  # Deploying binaries with the QoS profiles
88  su -c "bash $BUILD_PATH/crossing/crosscompile/Deploy.sh
    $BUILD_PATH/crossing/qos/USER_QOS_PROFILES.xml $PI" ronald
89
90  # Starting subscribers on remote lamps
91  for (( i=1; i <= $PI; i++ ))
92  do
93      ip_addr="192.168.1.$((i + 100))"
94      su -c "ssh ronald@$ip_addr \"./Crossing_subscriber $((i - 1))\" &"
    ronald
95  done
96
97  # Start publisher with xterm in the background
98  xterm -geometry 80x27+20+20 -e
    "$BUILD_PATH/LinuxTest/Crossing/StartCrossingPublisher.sh" 0 &
99
100 # Set focus back to the main terminal window
101 sleep 1
102 xdotool windowactivate $main_terminal_window_id
103 read -p "Press ENTER key to continue." < /dev/tty
104
105 # Removing all containers
106 docker container rm -f publisher
107
108 # Re-enable firewall
109 ufw enable
110
111 # remove accidental network rule
112 tc qdisc del dev enx00e04c6801bb root netem loss 100%
```

H. Függelék LampStart script

A lampStart scriptek felelősek a Subscriber alkalmazások futtatásáért. A működése a következő:

- Subscriber pingelése (3.sor)
- Ha a pingelés sikeres, akkor Subscriber leállítása, majd újra futtatása (4.-7.sor)

```
1 #!/bin/bash
2
3 ping -c 1 -W 1 192.168.1.101 > /dev/null 2>&1
4 if [ $? -ne 0 ]; then
5     exit 1
6 fi
7 ssh ronald@192.168.1.101 "pkill Crossing_subscr && ./Crossing_subscriber 1"
   &
```

I. Függelék Deploy script

A script működése a következő:

- Munkakönyvtárba való belépés (7.sor)
- DDS alkalmazás fordítása a megadott architektúrára crosscompile-lal (10.sor)
- Alkalmazás és QoS beállítás kitelepítése a paraméterként megadott számú célgépre (15.-23.sor)
    - IP cím összefűzése (17.sor)
    - Bináris másolása a célszámítógépre(19.sor)
    - Paraméterként kapott QoS beállítás másolása a célszámítógépre (21.sor)

```
1 #!/bin/bash
2
3 # ------------------------------------------------
4 # Settupping all Lamps connected to the network
5 # ------------------------------------------------
6
7 cd /home/ronald/7.2023/TDK/student-radai-bsc/crossing/crosscompile
8
9 # Compile code to target platform
10 make -f makefile_Crossing_armv8Linux4gcc7.3.0
11
12
13 turned_off=0
14 # Check if a pi is are connected to the network
15 for (( i=1; i <= $2; i++ ))
16 do
17     ip_addr="192.168.1.$((i + 100))"
18     echo "Sending subscriber to Lamp $i"
19     scp objs/armv8Linux4gcc7.3.0/Crossing_subscriber ronald@${ip_addr}:.
20     scp $1 ronald@${ip_addr}:.
21     turned_off=$((turned_off + 1))
22 done
23 echo "Deployed subscriber to $turned_off pi."
```

J. Függelék Shutdown script

A script felelős azért, hogy a Raspberry-k rendeltetésszerűen legyenek leállítva. A script a következőképpen működik:

- Leállítást számláló változó inicializálása (7.sor)
- Raspberry-k keresése ciklikusan, amelyik címen van elérhető, ott leállítjuk (9.-17.sor)
    - IP cím összefűzése (11.sor)
    - IP cím pingelése (12.sor)
    - Ha volt válasz, akkor leállítjuk a számítógépet és növeljük a leállítást számláló változót (13.-16.sor)
- Leállított számítógépek számának kiírása (18.sor)

```
1 #!/bin/bash
2
3 # ----------------------------------------------
4 # Shutting down all Lamps connected to the network
5 # ----------------------------------------------
6
7 turned_off=0
8 # Check if a pi is are connected to the network
9 for (( i=1; i <= 40; i++ ))
10 do
11     ip_addr="192.168.1.$((i + 100))"
12     ping -c 1 -W 1 $ip_addr > /dev/null 2>&1
13     if [ $? -eq 0 ]; then
14         ssh ronald@${ip_addr} "echo 'Ronald6.' | sudo -S shutdown now" &
15         turned_off=$((turned_off + 1))
16     fi
17 done
18 echo "Turned off $turned_off pi's"
```

K. Függelék Kereszteződés végrehajtó egységek applikáció kódja

A kód működése a következő:

- Csomag eldobási szabály parancs definiálása (1.sor)
- Csomag eldobási szabályt törlő parancs definiálása (2.sor)
- Flag definiálás, az első zóna zöld színt mutat (4.sor)
- Lámpa zónák számának definiálása (5.sor)
- Flag definiálás, véletlen bit, hogy a jelenlegi ciklus futásában szeretnénk-e manipulálni a csomagok eljutását a Subcriberhez (6.sor)
- Flag definiálás, azt jelzi, hogy a jelenlegi ciklus futásában manipuláljuk-e a csomagok eljutását a Subscriberig (7.sor)
- Flag definiálás, azt jelzi, hogy a manipulálás sikerült-e (8.sor)
- Író szekvenciaszámának definiálása (9.sor)
- Publisher ID beállítása (10.sor)
- Késleltetések definiálása (11.-13.sor)
- Rövid (2mp-es) várakozás az indítás után (16.sor)
- Fő ciklus, amiben a leállításig küldjük az üzeneteket (18.-97.sor)
    - Negáljuk a zónák színeit (20.sor)
    - Ciklus, melynek során minden zónának küldünk egy piros vagy egy zöld jelzést (22.-54.sor)
        - Eldöntjük, hogy a kimenő csomagokat kell-e manipulálni (25.sor)
        - Ha a manipulálás mellett döntöttünk, akkor beállítjuk a csomageldobási szabályt (26.-27.sor)
        - Kivárjuk a szabály aktivációs késleltetést (10ms, 28.sor)
        - Beállítjuk a kimenő struktúra zónaszámát és az üzenet ID-ját (30.-31.sor)
        - Beállítjuk a kiküldendő módosítás által kért színt (32.-35.sor)
        - Konzolra logoljuk a módosítás küldés kérést (37.-43.sor)
        - Elküldjük a színmódosítást (45.-49.sor)

        - Visszaállítjuk a hálózatot, ha manipulálva volt (52.-53.sor)
- Növeljük az üzenet szekvenciaszámát (56.sor)
- Kivárjuk a váltási késleltetést (20 mp, 57.sor)
- Kisorsoljuk, hogy a következő menetben meg fog-e hibásodni a kommunikáció (59.sor)
- Ciklus, melynek során az összes zónának küldünk egy sárga, vagy egy piros-sárga jelzést (61.-93.sor)
    - Eldöntjük, hogy a kimenő csomagokat kell-e manipulálni (64.sor)
    - Ha a manipulálás mellett döntöttünk, akkor beállítjuk a csomageldobási szabályt (65.-66.sor)
    - Kivárjuk a szabály aktivációs késleltetést (10ms, 67.sor)
    - Beállítjuk a kimenő struktúra zónaszámát és az üzenet ID-ját (30.-31.sor)
    - Beállítjuk a kiküldendő módosítás által kért színt (71.-74.sor)
    - Konzolra logoljuk a módosítás küldés kérést (76.-82.sor)
    - Elküldjük a színmódosítást (84.-88.sor)
    - Visszaállítjuk a hálózatot, ha manipulálva volt (91.-92.sor)
- Növeljük az üzenet szekvenciaszámát (95.sor)
- Kivárjuk a sárga utáni váltási késleltetést (2 mp, 96.sor)

```
1 #define SET_RED() { system("gpio -g write 7 0"); system("gpio -g write 8
  0"); system("gpio -g write 25 1"); }
2 #define SET_RED_AND_YELLOW() { system("gpio -g write 7 0"); system("gpio -g
  write 8 1"); system("gpio -g write 25 1"); }
3 #define SET_GREEN() { system("gpio -g write 7 1"); system("gpio -g write 8
  0"); system("gpio -g write 25 0"); }
4 #define SET_YELLOW() { system("gpio -g write 7 0"); system("gpio -g write 8
  1"); system("gpio -g write 25 0"); }
5 #define SET_YELLOW_BLINK() { system("gpio -g write 7 0"); system("gpio -g
  toggle 8"); system("gpio -g write 25 0"); }
6
7 void ProcessMessage(CrossingData* msg)
8 {
9     // Outputting the prompted state to the console
10     printf("%d. Message received from publisher %d. Set the lamp state to:
  %s\n", msg->id, msg->from+1, lamp_states[msg->prompt]);
11
12     // Setting the state according to the message
13     switch (msg->prompt)
14     {
15         case RED:
16             SET_RED();
17             break;
18         case RED_AND_YELLOW:
19             SET_RED_AND_YELLOW();
20             break;
21         case GREEN:
22             SET_GREEN();
23             break;
24         case YELLOW:
25             SET_YELLOW();
26             break;
27         default:
28             SET_YELLOW_BLINK();
29             break;
30     }
31 }
32
33 void CrossingDataListener_on_data_available(
34     void* listener_data,
35     DDS_DataReader* reader)
36 {
37     ...
38
39     for (i = 0; i < CrossingDataSeq_get_length(&data_seq); ++i)
40     {
41         if (DDS_SampleInfoSeq_get_reference(&info_seq, i)->valid_data)
42         {
43             CrossingData* current =
  CrossingDataSeq_get_reference(&data_seq, i);
44             if(current->to == zone_id)
45             {
46                 // Controller sent a message to us, so it is online
47                 controller_online = 1;
48                 // Process the received message
49                 ProcessMessage(current);
50             }
51         }
52     }
```

```
53
54      ...
55  }
56
57  int subscriber_main(int domainId, int sample_count)
58  {
59      ...
60
61      reader_listener.on_data_available =
62      CrossingDataListener_on_data_available;
63
64      ...
65
66      struct DDS_Duration_t poll_period = {60,0};
67      struct DDS_Duration_t switch_period = {2,0};
68
69      /* Main loop*/
70      for (count=0; (sample_count == 0) || (count < sample_count); ++count)
    {
71
72          if(controller_online)
73          {
74              controller_online = 0;
75              NDDS_Utility_sleep(&poll_period);
76          }
77          else
78          {
79              SET_YELLOW_BLINK();
80          }
81          NDDS_Utility_sleep(&switch_period);
82      }
83
84      /* Cleanup and delete all entities */
85      return subscriber_shutdown(participant);
86  }
```

L. Függelék Kereszteződés vezérlő applikáció kódja

A kereszteződést vezérlő Publisher kódjának lényegi részének működése a következő:

- Csomag eldobási szabály parancs definiálása (1.sor)
- Csomag eldobási szabályt törlő parancs definiálása (2.sor)
- Flag definiálás, az első zóna zöld színt mutat (4.sor)
- Zónák számának definiálása (5.sor)
- Flag definiálás, véletlen bit, hogy a jelenlegi ciklus futásában szeretnénk-e manipulálni a csomagok eljutását a Subcriberhez (6.sor)
- Flag definiálás, azt jelzi, hogy a jelenlegi ciklus futásában manipuláljuk-e a csomagok eljutását a Subscriberig (7.sor)
- Flag definiálás, azt jelzi, hogy a manipulálás sikerült-e (8.sor)
- Író szekvenciaszámának definiálása (9.sor)
- Publisher ID beállítása (10.sor)
- Késleltetések definiálása (11.-13.sor)
- Rövid (2mp-es) várakozás az indítás után (16.sor)
- Fő ciklus, amiben a leállításig küldjük az üzeneteket (18.-97.sor)

- Negáljuk a zónák színeit (20.sor)
- Ciklus, amely során minden lámpának küldünk egy piros, vagy egy zöld jelzést (22.-54.sor)
    - Eldöntjük, hogy a kimenő csomagokat kell-e manipulálni (25.sor)
    - Ha a manipulálás mellett döntöttünk, akkor beállítjuk a csomageldobási szabályt (26.-27.sor)
    - Kivárjuk a szabály aktivációs késleltetést. (10ms, 28.sor)
    - Beállítjuk a kimenő struktúra cél ID-ját és az üzenet ID-ját (30.-31.sor)
    - Beállítjuk a kiküldendő módosítás által kért színt. (32.-35.sor)
    - Konzolra logoljuk a módosítás küldés kérést (37.-43.sor)
    - Elküldjük a módosítást a Subscribereknek (45.-49.sor)
    - Visszaállítjuk a hálózatot, ha manipulálva volt (52.-53.sor)
- Növeljük az üzenet szekvenciaszámát (56.sor)
- Kivárjuk a váltási késleltetést (20 mp, 57.sor)
- Kisorsoljuk, hogy a következő menetben meg fog-e hibásodni a kommunikáció (59.sor)
- Ciklus, amely során az összes lámpának küldünk egy sárga, vagy egy piros-sárga jelzést (61.-93.sor)
    - Eldöntjük, hogy a kimenő csomagokat kell-e manipulálni (64.sor)
    - Ha a manipulálás mellett döntöttünk, akkor beállítjuk a csomageldobási szabályt (65.-66.sor)
    - Kivárjuk a szabály aktivációs késleltetést. (10ms, 67.sor)
    - Beállítjuk a kimenő struktúra cél ID-ját és az üzenet ID-ját (30.-31.sor)
    - Beállítjuk a kiküldendő módosítás által kért színt. (71.-74.sor)
    - Konzolra logoljuk a módosítás küldés kérést (76.-82.sor)
    - Elküldjük a módosítást a Subscribereknek (84.-88.sor)
    - Visszaállítjuk a hálózatot, ha manipulálva volt (91.-92.sor)
- Növeljük az üzenet szekvenciaszámát (95.sor)
- Kivárjuk a sárga utáni váltási késleltetést (2 mp, 96.sor)

```
1 static char* ruleActive = "tc qdisc add dev enx00e04c6801bb root netem loss
  100%";
2 static char* ruleInactive = "tc qdisc del dev enx00e04c6801bb root netem
  loss 100%";
3
4 int firstGreen = 0;
5 int zoneCount = 2;
6 int randManipulate = 0;
7 int manipulate = 0;
8 int manipulated = 0;
9 int writeID = 1;
10 instance->from = publisher_id;
11 struct DDS_Duration_t yellow_to_prompt_delay = {2,0};
12 struct DDS_Duration_t change_time = {20,0};
13 struct DDS_Duration_t ten_millisec = {0,10000000};
14
15 // Wait for initialization
16 NDDS_Utility_sleep(&yellow_to_prompt_delay);
17
18 for (count=0; (sample_count == 0) || (count < sample_count); ++count)
19 {
20     firstGreen = !firstGreen;
21
22     for(int i = 0; i < zoneCount; i++)
23     {
24         // Manipulate if needed
25         manipulate = i % 2 == 0 && !firstGreen && randManipulate;
26         if(manipulate)
27             manipulated = system(ruleActive);
28         NDDS_Utility_sleep(&ten_millisec);
29
30         instance->to = i;
31         instance->id = writeID;
32         if(i % 2 == 0)
33             instance->prompt = firstGreen ? GREEN : RED;
34         else
35             instance->prompt = firstGreen ? RED : GREEN;
36
37         if(manipulate && !manipulated)
38         {
39             printf("*%d. Sending prompt %s to subscriber %d\n", writeID,
  lamp_states[instance->prompt], i+1);
40             manipulated = 0;
41         }
42         else
43             printf("%d. Sending prompt %s to subscriber %d\n", writeID,
  lamp_states[instance->prompt], i+1);
44
45         retcode = CrossingDataDataWriter_write(
46             CrossingData_writer, instance, &instance_handle);
47         if (retcode != DDS_RETCODE_OK) {
48             fprintf(stderr, "write error %d\n", retcode);
49         }
50
51         // Restore network
52         if(manipulate)
53             system(ruleInactive);
54     }
55     printf("\n");
```

```
56      ++writeID;
57      NDDS_Utility_sleep(&change_time);
58
59      randManipulate = rand() % 2;
60
61      for(int i = 0; i < zoneCount; i++)
62      {
63          // Manipulate if needed
64          manipulate = i % 2 == 0 && firstGreen && randManipulate;
65          if(manipulate)
66              manipulated = system(ruleActive);
67          NDDS_Utility_sleep(&ten_millisec);
68
69          instance->to = i;
70          instance->id = writeID;
71          if(i % 2 == 0)
72              instance->prompt = firstGreen ? YELLOW : RED_AND_YELLOW;
73          else
74              instance->prompt = firstGreen ? RED_AND_YELLOW : YELLOW;
75
76          if(manipulate && !manipulated)
77          {
78              printf("*%d. Sending prompt %s to subscriber %d\n", writeID,
    lamp_states[instance->prompt], i+1);
79              manipulated = 0;
80          }
81          else
82              printf("%d. Sending prompt %s to subscriber %d\n", writeID,
    lamp_states[instance->prompt], i+1);
83
84          retcode = CrossingDataDataWriter_write(
85              CrossingData_writer, instance, &instance_handle);
86          if (retcode != DDS_RETCODE_OK) {
87              fprintf(stderr, "write error %d\n", retcode);
88          }
89
90          // Restore network
91          if(manipulate)
92              system(ruleInactive);
93      }
94      printf("\n");
95      ++writeID;
96      NDDS_Utility_sleep(&yellow_to_prompt_delay);
97  }
```

M. Függelék A tesztkörnyezet létrehozását bemutató, adatvesztést demonstráló script Windowson

A csomagvesztéses kommunikáció tesztelésére a következő scriptet írtam:

- Kód fordítása (4. sor)
- Deploy kép buildelés (7. sor)
- Subscriber(ek) futtatása (10., 13. sor)
- Publisher futtatása (16. sor)
- Subscriber hálózati kapcsolatának lecsatlakoztatása (20. sor)
- 10 másodperces várakozás (21. sor)

- Subscriber visszacsatlakoztatása (24. sor)
- Billentyű eseményre várakozása (25. sor)
- Konténerek leállítása (28. sor)
- Deploy kép buildelés a módosított QoS beállításokkal (33. sor)
- Subscriber(ek) futtatása (36., 39. sor)
- Publisher futtatása (42. sor)
- Subscriber hálózati kapcsolatának lecsatlakoztatása (46. sor)
- 10 másodperces várakozás (47. sor)
- Subscriber visszacsatlakoztatása (50. sor)
- Billentyű eseményre várakozása (51. sor)
- Konténerek leállítása (54. sor)

```
@ECHO OFF

REM Compile code
docker run -it --rm -v C:\...\container\myapp:/app -w="/app" dds-build make
 -f makefile_TestData_x64Linux4gcc7.3.0

REM Build containers from code
docker build -t myapp-deploy C:\...\container\myapp
-f C:\...\container\myapp\Dockerfile

REM Start subscriber
START "Subscriber" StartSubscriber.bat

REM Start subscriber2
START "Subscriber2" StartSubscriber2.bat

REM Start publisher
START "Publisher" StartPublisher.bat
TIMEOUT 5

REM Disconnect subscriber from network
docker network disconnect test subscriber
TIMEOUT 10

REM Reconnect subscriber to network
docker network connect test subscriber
PAUSE

REM Stopping all containers
FOR /f "tokens=*" %%i IN ('docker ps -q') DO docker stop %%i

REM Running the test again with qos profile

REM Build containers from code
docker build -t myapp-deploy C:\...\container\myapp
-f C:\...\container\myapp\Dockerfile2

REM Start subscriber
START "Subscriber" StartSubscriber.bat

REM Start subscriber2
START "Subscriber2" StartSubscriber2.bat

REM Start publisher
START "Publisher" StartPublisher.bat
TIMEOUT 5

REM Disconnect subscriber from network
docker network disconnect test subscriber
TIMEOUT 10

REM Reconnect subscriber to network
docker network connect test subscriber
PAUSE

REM Stopping all containers
FOR /f "tokens=*" %%i IN ('docker ps -q') DO docker stop %%i
```

N. Függelék A tesztkörnyezet létrehozását bemutató, adatvesztést demonstráló script Linuxon

A csomagvesztéses kommunikáció tesztelésére a következő scriptet írtam:

- Kód fordítása (4. sor)
- Deploy kép buildelés (7. sor)
- Subscriber(ek) futtatása (10., 13. sorok)
- Publisher futtatása (16. sor)
- Google pingelése (20. sor)
- Publisher kimenő csomagjai 50 százalékának eldobása (25. sor)
- Billentyűlenyomásra várakozás (27. sor)
- Csomageldobási szabály törlése (30. sor)
- Billentyű eseményre várakozása (32. sor)
- Konténerek leállítása, törlése (35., 36., 37. sorok)
- Deploy kép buildelés a módosított QoS beállításokkal (42. sor)
- Subscriber(ek) futtatása (45., 48. sor)
- Publisher futtatása (51. sor)
- Google pingelése (55. sor)
- Publisher kimenő csomagjai 50 százalékának eldobása (60. sor)
- Billentyűlenyomásra várakozás (62 sor)
- Csomageldobási szabály törlése (65. sor)
- Billentyű eseményre várakozása (67. sor)
- Konténerek leállítása, törlése (70., 71., 72. sorok)

```
#! /bin/bash

# Compile code
sudo docker run -it --rm -v ~/demo/myapp:/app -w="/app" dds-build make -f
  makefile_TestData_x64Linux4gcc7.3.0

# Build containers from code
sudo docker build -t myapp-deploy ~/demo/myapp -f $HO-
  ME/demo/myapp/Dockerfile

# Start subscriber with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartSubscriber.sh" &

# Start subscriber2 with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartSubscriber2.sh" &

# Start publisher with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartPublisher.sh" &
sleep 5


# Start pinging with xterm in the background to demonstrate network
  emulation
sudo xterm -e "docker exec -t subscriber ping google.com" &
echo Pinging google.com
sleep 5

# Disconnect publisher from network
sudo docker exec -t publisher tc qdisc add dev eth0 root netem loss 50%
echo Started dropping some packets from publisher
read -p "Press Enter key to continue." < /dev/tty

# Reconnect publisher to network
sudo docker exec -t publisher tc qdisc del dev eth0 root netem loss 50%
echo Stopped dropping some packets from publisher
read -p "Press Enter key to continue." < /dev/tty

# Removing all containers
sudo docker container rm -f publisher
sudo docker container rm -f subscriber
sudo docker container rm -f subscriber2

# Running the test again with qos profile

# Build containers from code
sudo docker build -t myapp-deploy ~/demo/myapp -f $HO-
  ME/demo/myapp/Dockerfileqos

# Start subscriber with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartSubscriber.sh" &

# Start subscriber2 with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartSubscriber2.sh" &

# Start publisher with xterm in the background
sudo xterm -e "$HOME/demo/LinuxTest/StartPublisher.sh" &
sleep 5

```

```
55 # Start pinging with xterm in the background to demonstrate network
   emulation
56 sudo xterm -e "docker exec -t subscriber ping google.com" &
57 echo Pinging google.com
58 sleep 5
59
60 # Disconnect publisher from network
61 sudo docker exec -t publisher tc qdisc add dev eth0 root netem loss 50%
62 echo Dropping some packets from publisher
63 read -p "Press Enter key to continue." < /dev/tty
64
65 # Reconnect publisher to network
66 sudo docker exec -t publisher tc qdisc del dev eth0 root netem loss 50%
67 echo Stop dropping some packets from publisher
68 read -p "Press Enter key to continue." </dev/tty
69
70 # Removing all containers
71 sudo docker container rm -f publisher
72 sudo docker container rm -f subscriber
73 sudo docker container rm -f subscriber2
```

Irodalomjegyzék

[1] H.-J. B. Sarita V. Adve, „Auburn Engineering,” augusztus 2010. [Online]. Available: https://www.eng.auburn.edu/~agrawvd/COURSE/READING/ARCH/memory_models.pdf. [Hozzáférés dátuma: 28 március 2023].

[2] DDS Foundation, „What is DDS?,” Object Management Group, Inc., [Online]. Available: https://www.dds-foundation.org/what-is-dds-3/. [Hozzáférés dátuma: 23 04 2023].

[3] „diy Release Seven,” [Online]. Available: https://diy.inria.fr/doc/herd.html.

[4] RTI, „DDS: An Open Standard for Real-Time Applications,” . [Online]. Available: https://www.rti.com/products/dds-standard.

[5] I. Docker, „Docker docs,” Docker, Inc., 2013. [Online]. Available: https://docs.docker.com/get-started/overview/.

[6] Alexei, „Pumba GitHub,” [Online]. Available: https://github.com/alexei-led/pumba. [Hozzáférés dátuma: 13 04 2023].

[7] M. Kerrisk, „tc(8) — Linux manual page,” Linux/UNIX system programming training courses, 16 December 2001. [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html.

[8] B. Levente, „Axiomatic Analysis of Distributed Systems,” 2022.

[9] J. A. a. L. Maranget., „diy7 tool suite,” Institut National de Recherche en Informatique et en Automatique, 2010. [Online]. Available: http://diy.inria.fr/www/.

[10] J. A. a. L. Maranget, „The Herd toolsuite to deal with .cat memory models,” Institut National de Recherche en Informatique et en Automatique, 2010. [Online]. Available: https://github.com/herd/herdtools7.

[11] 1. contributors, „What is the Windows Subsystem for Linux?,” Microsoft, 08 12 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/wsl/about.

[12] Yiyiyimu, „WSL2 seems not support traffic control,” [Online]. Available: https://github.com/microsoft/WSL/issues/6065.

[13] J. A. a. L. Maranget, „Simulating memory models with herd7,” [Online]. Available: http://diy.inria.fr/doc/cos.cat.

[14] R. C. users, „RTI Community,” RTI, [Online]. Available: https://community.rti.com/kb/creating-docker-image-rti-connext-dds. [Hozzáférés dátuma: 13 04 2023].

[15] B. M. FTSRG, „MODES3 - Model-based Demonstrator for Smart and Safe Systems,” Budapest University of Technology and Economics, Fault Tolerant Systems Research Group, 2017. [Online]. Available: https://modes3.inf.mit.bme.hu. [Hozzáférés dátuma: 24 10 2023].

[16] P. Koopman, „How to Write an Abstract,” október 1997. [Online]. Available: https://users.ece.cmu.edu/~koopman/essays/abstract.html. [Hozzáférés dátuma: 20 október 2015].

[17] W3C, „HTML, The Web’s Core Language,” [Online]. Available: http://www.w3.org/html/. [Hozzáférés dátuma: 20 október 2015].

[18] K. Nahtkasztlija, „Az idegen szavak toldalékolása,” június 2009. [Online]. Available: http://www.pcguru.hu/blog/kredenc/az-idegen-szavak-toldalekolasa/5062.

[19] Object Management Group (OMG), "What is DDS?," [Online]. Available: https://www.dds-foundation.org/what-is-dds-3/.

[20] RTI, „Data Distribution Service,” Real-Time Innovations, [Online]. Available: https://www.rti.com/products/dds-standard. [Hozzáférés dátuma: 23 04 2023].

[21] O. M. Group, „OMG Data Distribution Service,” 2015.