Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Szeles Nikolett

LOKÁLIS IOT FELHŐK KÖZÖTTI BIZTONSÁGOS, SZOLGÁLTATÁSALAPÚ ADATKOMMUNIKÁCIÓ

TDK dolgozat

KONZULENS

Dr. Varga Pál

BUDAPEST, 2017

Tartalomjegyzék

Összefoglaló

Az infokommunikációs technológiák fejlődésének, valamint az összekapcsolható szenzorok és beavatkozók széles körű elérhetőségének köszönhetően egyre elterjedtebbek az IoT, azaz Internet of Things keretrendszerek. Ezek segítségével egyre összetettebb rendszereket építünk, melyekben rengeteg szenzor és különböző – valamilyen formában intelligensnek tekinthető – komponens működik együtt. Az IoT eszközöket az iparban is széleskörűen alkalmazzák, ilyen kontextusban IIoT, Industrial Internet of Thingsnek nevezzük őket.

Az IoT eszközök gyakran korlátozott erőforrásokkal rendelkeznek, ezért biztonságos kommunikációjuk nagyobb technológiai kihívást jelent, mint fejlettebb eszközök esetében. Ugyanakkor a biztonság kérdése itt is ugyanolyan fontos, nem hagyhatjuk őket könnyű célpontként a támadók számára.

Dolgozatomban szeretném bemutatni az Arrowhead keretrendszert, mely az ipari automatizálásra fókuszál és egy kidolgozott, biztonságos keretet szeretne nyújtani a beágyazott rendszerek együttműködéséhez. Alapjául a lokális felhők szolgálnak; ezek felelnek az adatbiztonságért, a valós idejű követelmények teljesítéséért, vagy a folyamatok és konfigurációk követhetőségéért is.

A keretrendszer működése során gyakran adódik olyan helyzet, hogy egy lokális felhő képtelen kiszolgálni az összes hozzá érkező kérést. Ezért is fontos, hogy létre tudjunk hozni biztonságos kapcsolatot olyan lokális felhők között, amelyeknek funkcionálisan egyébként megvan a lehetőségük szolgáltatást nyújtani egymás számára.

A megoldás részeként szeretném az Arrowhead keretrendszert egy új modullal bővíteni, amely együttműködik a keretrendszer már meglévő központi rendszereivel. Az új modul arra hivatott, hogy biztonságos adatutat hozzon létre a két lokális felhő között. A koncepció helyességének bizonyításaként ezt socketek és egy AMQP (Advanced Message Queuing Protocol) protokollon alapuló RabbitMQ szerver segítségével valósítottam meg, Java technológiával.

Abstract

Due to the great evolution of infocommunication and the prevalence of connectable sensors and actuators, IoT (Internet of Things) frameworks have become very popular and widely available. We are building more and more complex IoT applications, where a lot of sensors and other intelligent components work together. These concepts and devices - that are related to Internet of Things - are widely used in the industry as well. In this context, they are then called IIoT (Industrial Internet of Things).

IoT devices have constrained resources and communication capabilities. Therefore, the implementation of secure communication between them is a major technological challenge for the engineers. The issue of security is an essential aspect: we should not leave them an easy target for the attackers. The concept of local automation clouds helps tackling the issues of real-time restrictions, ergonomic engineering and handling the various levels of security. Still, some services should be able to reach over inbetween such closed, local networks (in here called “inter-cloud servicing”). Naturally, this has to be aided by proper security mechanisms as well.

In my paper, I provide a brief overview of the Arrowhead framework that focuses on industrial automation – for which the utilization of such closed, local automation-related networks (clouds) are inevitable. The aim of this framework is to provide a comprehensive, secure skeleton for the collaboration of embedded systems.

However, in order to provide secure data communication for inter-cloud services, new methods need to be introduced for Arrowhead. As part of my solution, I would like to expand the framework with a new module, which collaborates with the existing core systems. This new module shall establish the secure data path between two local clouds with the help of TCP sockets and a RabbitMQ broker (based on AMQP - Advanced Message Queueing Protocol). The implementation for the proof of concept is written in Java. This paper describes the planning, implementation and verification steps of this solution.

1 Bevezetés

Az informatika egy rendkívül sokrétű tudományággá nőtte ki magát az évek során. Számos területre tagolható és ennek egy viszonylag fiatal, de annál ígéretesebb eleme az IoT (Internet of Things, Dolgok internete) eszközök világa. A szakértők nagy jövőt, illetve a gőzgépek, szerelőszalagok és az automatizálás utáni negyedik ipari forradalom elősegítését jósolják az összekapcsolt, intelligens információs rendszerek kialakulása révén [1]. Az alábbi dolgozatban egy olyan problémát szeretnék kifejteni és megoldani, amely szoros összefüggésben áll ezzel a térhódító technológiai perspektívával.

A Gartner Inc. egy vezető kutatási és tanácsadó vállalat, mely a legkülönbözőbb iparágakban tevékenykedő vállalatok számára nyújt segítséget abban, hogy minden szempontból optimális döntést tudjanak hozni. Egy általuk publikált felmérés azt mutatta ki, hogy 2017-ben 8.4 milliárd világhálóra kapcsolt IoT eszköz létezik. Ez a szám 2020-ban várhatóan már eléri a 20.4 milliárdot [2]. Ez azt jelenti, hogy belátható időn belül a fejlett országok lakosságának legtöbb tagja rendelkezni fog legalább három világhálóra kapcsolt intelligens eszközzel. Ugyanakkor az IoT eszközök nem csak a háztartások szerves részeivé válnak, de az iparban is több milliárd található belőlük.

Az ipari IoT eszközöket szokás IIoT [3], azaz Industrial Internet of Things néven is emlegetni. Az együttműködésük révén változatos feladatokat tudnak ellátni, például vízminőség ellenőrzése, kritikus munkaeszközök állapotának figyelése, parkolórendszer kialakítása. Bonyolult feladat megteremteni egy olyan keretrendszert, mely szabványos keretek közé rendezi ezen készülékek kommunikációját és használatát, miközben figyelembe kell venni azt is, hogy milyen korlátozott erőforrásokkal rendelkező és sokszor végtelenül egyszerű működésen alapuló készülékekről beszélünk.

Az IIoT szenzorok közötti kommunikáció sok támadó figyelmét kelti fel. Ebből adódóan úgy gondolom, hogy kardinális kérdés a biztonság, ezért is ez az egyik fő aspektusa a dolgozatomnak. Az elmúlt években jellemző volt, hogy sok mérnök azzal a hozzáállással tervezett rendszereket, hogy az működjön minél hatékonyabban és minél gyorsabban, viszont teljes mértékben figyelmen kívül hagyták a biztonsági kérdéseket. Más esetekben a megrendelőket nem érdekelte, hogy sebezhető a rendszer, nem voltak hajlandóak a biztonsági megoldások miatt további összeget szánni a fejlesztésre, vagy

tovább várni a beüzemeléssel. Ez azért volt így, mert a legtöbben nem éreztek valós fenyegetést; úgy gondolták, csak másokat támadhat meg egy rosszindulatú harmadik fél. A világ pozitív irányba halad a biztonság fontosságának megítélését illetően, ezért lett alapkövetelmény az IoT keretrendszerek számára is ez napjainkban.

A következő fejezetben bemutatom az Arrowhead keretrendszert, mely fejlesztése során naprakész kérdőívek segítségével mérték fel az ipari partnerek igényeit, amelyekből egyértelműen kiderült, hogy fontosnak tartják a biztonságot. Az általános bemutatás után kifejtem a lokális felhők koncepciójának alapjait és tárgyalásra kerülnek azon problémák, amelyek felmerülnek az együttműködés során. A 3. fejezet részeként bemutatásra kerül a tervezés szempontjából kulcsfontosságú Orchestrator és Gatekeeper központi rendszer. Ezek mellett a fejezetben nagy hangsúlyt kap azon Gateway modul, mely implementálás alatt áll általam, és amely magába foglalja mindazon funkcionalitást, ami szükséges a biztonságos adatút kialakításához két szeparált lokális felhő együttműködése során. A 4. fejezet a megvalósítás részleteit tárgyalja, illetve szekvencia diagramokon és osztály diagramokon keresztül támasztja alá a Gateway modul megvalósíthatóságát és hatékonyságát.

Tehát összefoglalva és tömören, a dolgozatom fókuszában azon probléma kifejtése és megoldása áll, hogy a folyamatos fejlesztés alatt álló Arrowhead keretrendszer hogyan kezelje a felhők közötti kommunikációt biztonságos alapokon.

Az ebben a dolgozatban bemutatott munkám lényege a Gateway nevű központi modul folyamatainak és működésének tervezése és implementációja, beleértve az akörüli biztonságtechnikai kérdések lefedését is.

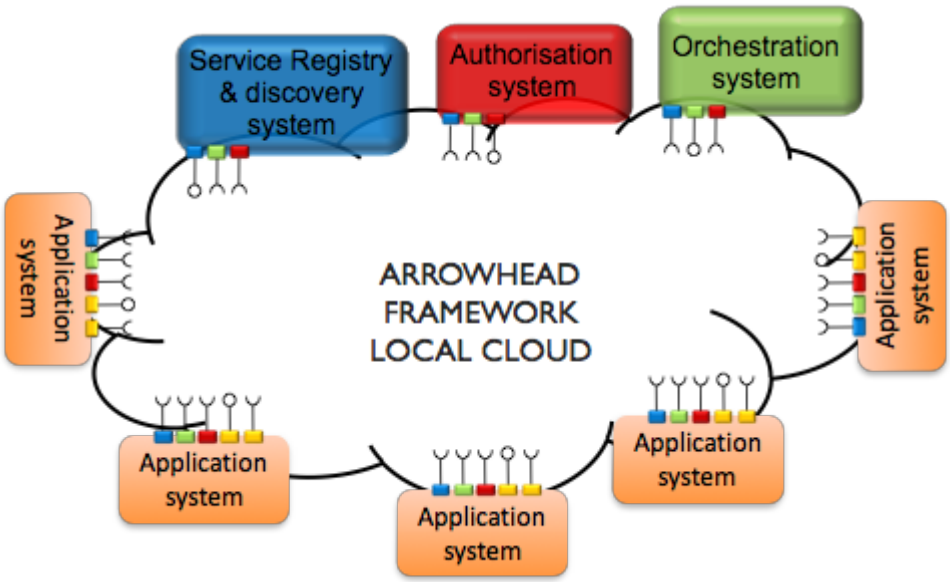
2 Lokális IoT felhők

2.1 Az Arrowhead keretrendszer

Az Arrowhead keretrendszer [4] fejlesztése egy Európai Unió által pártfogolt nemzetközi projekt keretein belül kezdődött. Az Arrowhead projekt ipari automatizálás területére fókuszál és egy kidolgozott, biztonságos keretet szeretne nyújtani a beágyazott rendszerek gördülékeny együttműködéséhez. Öt, viszonylag jól körülhatárolható területre összpontosít: ipari termelés és feldolgozóipar, okos városok, elektromobilitás, energiatermelés és a villamosenergia-piac automatizálása.

2.1.1 A központi rendszerek fogalma és helye az Arrowhead keretrendszerben

A keretrendszer alapjául az ún. „Core System”-ek, azaz központi rendszerek szolgálnak. Minden egyes központi rendszernek jól meghatározott feladata van és ajánlatos, hogy mindegyikből egy megtalálható legyen minden lokális felhőben. Az ajánlás ellenére mindössze három modul megléte kötelező és teljes mértékben nélkülözhetetlen, ezek név szerint a Service Registry, Orchestrator és az Authorization. Az 1. ábra szemlélteti az imént felvázolt architektúrát.



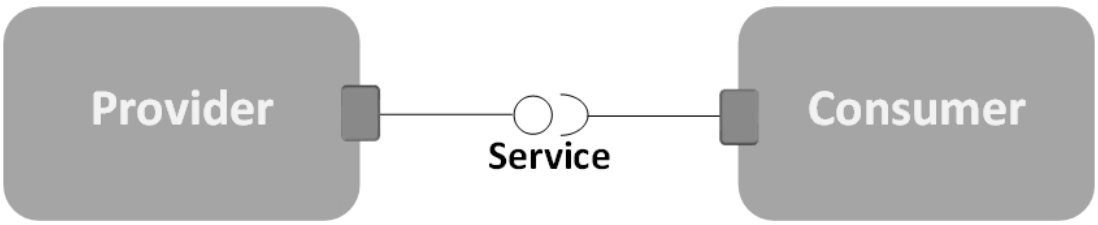
1. ábra: Az Arrowhead keretrendszer architektúrája[5]

Van néhány alapfogalom, amely gyakran elő fog fordulni a továbbiakban, így

célszerű definiálni őket még a központi rendszerek leírása előtt [6].

- Az egyik legfontosabb az, hogy mit is értünk *szolgáltatás (service)* alatt, amikor az Arrowhead koncepció keretein belül említjük. Minden szolgáltatásnak tekinthető, amikor információ cserélődik egy szolgáltató (providing) és egy fogyasztó (consuming) rendszer között
- A *rendszer (system)* maga az, ami kínál és/vagy felhasznál egy-egy szolgáltatást. Egy rendszer kínálhat egy vagy több szolgáltatást és használhat közben más szolgáltatásokat is. A rendszer szoftveresen van implementálva és egy eszközön fut.
- Az Arrowheaddel kompatibilis *eszköznek* nevezünk minden berendezést, gépet, hardvert, amely rendelkezik számítási és kommunikációs képességgel, illetve memóriával és fut rajta egy vagy több Arrowhead rendszer.
- „*System of Systems*”*(SoS)* alatt rendszerek egy olyan halmazát értjük, amelyek adminisztrációjáért a kötelező központi rendszerek felelnek, és ahol az információcsere szolgáltatások által valósul meg. Magyar megfelelője a „*rendszerek rendszere”* kifejezés lehet, de a magyar szaknyelvben is az angol változat az elterjedt és az elfogadott.
- A *Provider* olyan rendszer, amely egy vagy több szolgáltatást nyújt.
- A *Consumer* olyan rendszer, amely egy vagy több szolgáltatást „fogyaszt”.

Az Arrowhead kommunikációja SOA (Service-Oriented Architecture, szolgáltatásorientált architektúra) alapokon nyugszik, ami elősegíti a különböző rendszerek együttműködését és az integrációt magas szinten teszi láthatóvá, elfedve az egyes szolgáltatások megvalósítását [7]. Egy Arrowhead szolgáltatás különböző, a SOA elveit követő protokollt használhat, többek között REST, COAP, XMPP, MQTT vagy OPC-UA protokollt. A Provider és a Consumer közötti SOA alapú szolgáltatásnyújtás a 2. ábrán figyelhető meg.



2. ábra: A Consumer és a Provider közötti szolgáltatásalapú kommunikáció

A fent említett központi rendszerek részletesebben a következő bekezdésekben kerülnek tárgyalásra.

A *Service Registry* kezeli az aktuális lokális felhőben lévő Providerek által nyújtott szolgáltatások nyilvántartását. Egy Provider dinamikusan regisztrálhat egy-egy szolgáltatást, és ugyanilyen könnyen vissza is vonhatja azt.

Az *Orchestrator* feladata az, hogy támogassa az ugyanazon szolgáltatást nyújtó és igénylő felek egymásra találását. Nevezhetjük ezt a modult az Arrowhead framework lelkének is, ha úgy tetszik.

A történelmi okokból *Authorization* nevű rendszer mára már az AAA, azaz az Authorization, Authentication és az Accounting biztonsági szempontból szent hármasát kezeli. Ezen aspektusokat szeretném részletesebben is kibontani.

Az **Authentication (azonosítás)** célja az, hogy a kommunikációban résztvevő felekről egyértelműen el tudjunk dönteni, hogy azok-e, akiknek mondják magukat. Tehát valamilyen formában, akár felhasználónév-jelszó páros segítségével vagy token használatával, esetleg SSL tanúsítvány formájában ellenőrizni tudjuk kilétüket. Ezzel el tudjuk kerülni azt a típusú támadást, amikor a támadó valaki másnak próbálja meg kiadni magát.

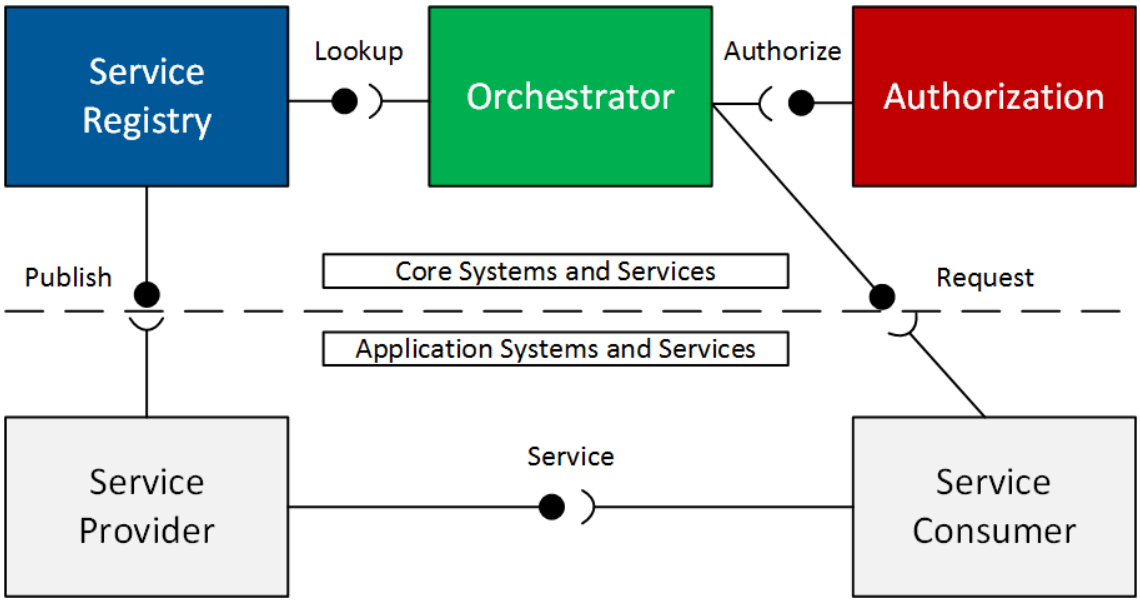
Az **Authorization (jogosultságkezelés)** dolga az, hogy az előzetesen azonosított résztvevők, azaz a hitelesített „személyek” esetében megállapítsa, hogy milyen jogosultságokkal rendelkeznek. Egyszerűbben kifejezve van-e joguk valamilyen szolgáltatás kéréséhez, valamilyen adathoz való hozzáféréshez. Az informatikában jól bevált és követendő gyakorlat az, hogy minden résztvevő a lehető legkevesebb jogot kapja, ami még elegendő ahhoz, hogy véghez vigye azt, amiért létrejött. Tehát bármilyen kényelmesnek is tűnik elsőre, nem tanácsos, sőt kifejezetten kerülendő mindenki számára a lehető legmagasabb jogosultság megadása. Bár elsőre körülményesnek és időigényesnek tűnhet az ilyen – elsődlegesen lustaságból vagy feledékenységből elkövetett – biztonsági rések elkerülése, napjainkban egy magára valamit is adó mérnök nem engedhet meg magának olyan szintű könnyelműséget, hogy nem fordít erre figyelmet.

Az AAA utolsó résztvevője az **Accounting (könyvelés)**, mely fontos szerepet tölt be többek között a telekommunikáció területén is. Bizonyos szolgáltatások csak díj

ellenében vehetők igénybe és az ezekkel kapcsolatban felmerülő feladatokat látja el az Accounting, adatokat gyűjt a felhasználó által igénybe vett hálózati erőforrásokról.

Ezeken a kötelezően jelenlévő modulokon kívül a keretrendszer több olyan kiegészítő rendszert kínál, melyek esszenciális szolgáltatásokat nyújtanak az ipari vállalatok számára. Ilyen többek között a Quality of Service (QoS) [8] szolgáltatás, vagy a jelenleg implementálás alatt álló Gateway és Gatekeeper modul.

Az előzőekben megismert központi rendszerek egymás számára nyújtott szolgáltatásai a 3. ábrán kerülnek bemutatásra.



3. ábra: A központi szolgáltatások az Arrowhead keretrendszerben [9]

2.1.2 A lokális felhők koncepciója

Ezen alfejezetben a lokális felhők fogalmát szeretném ismertetni és kifejteni. A lokális felhő alatt lényegében rendszerek egy halmazát tekintjük, azaz egy az előző alfejezetben már definiált „System of Systems" komponenst.

A lokális felhők fogalma azért vált az Arrowhead működésének alapjául, mert a keretrendszer fejlesztői globálisan gondolkoznak és felismerték azt, hogy ebben a sikerorientált és iparilag is felgyorsult társadalomban jellemzően olyan vállalatokkal kerülünk kapcsolatba, akik nemzetközileg elosztottan működnek. Abban az esetben, amikor egy logikailag összetartozó, de földrajzilag szétdarabolt üzem szeretne biztonságos és hatékony kooperációt lebonyolítani, fontos lehet az, hogy a cég által használt eszközök struktúrája logikailag jól legyen kialakítva. Erre nyújt ideális

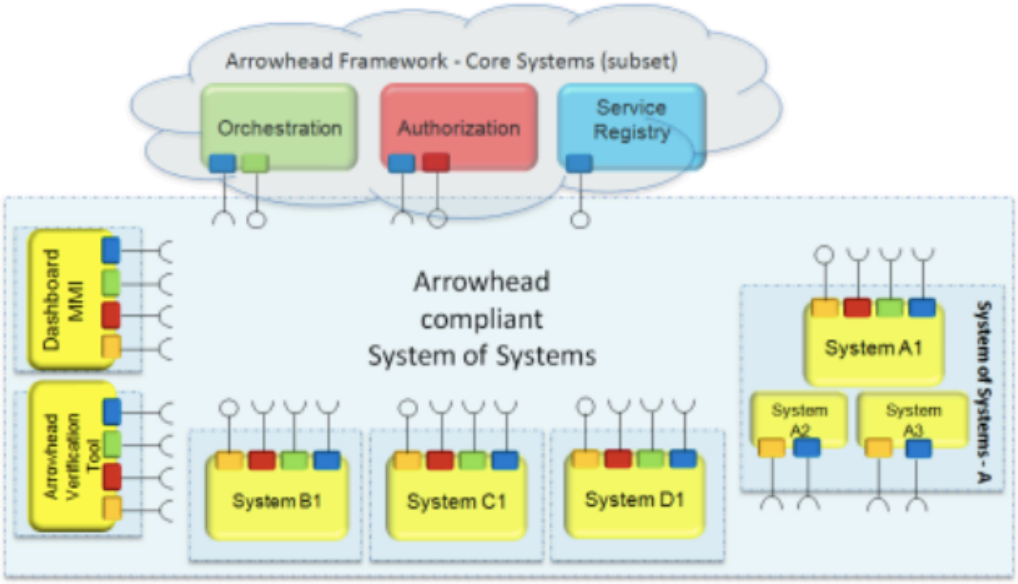
megoldást a helyi felhők alkalmazása.

Az Arrowhead lokális felhőjétől elvárt fő tulajdonságok a [10] alapján:

- Önállóság
- Automatizálás támogatása
    - Automatizált rendszerek tervezésének, konfigurálásának, telepítésének, működtetésének és karbantartásának támogatása
    - Lehetőség biztosítása eseményalapú információcserére
    - Lehetőség biztosítása az információcsere felülvizsgálatára
    - A Quality of Service (QoS) kommunikáció támogatása
- Védelem a külső hálózatoktól
- Biztonságos rendszerbetöltés („*bootstrapping*”) és szoftverfrissítés
- Eszköz, rendszer és szolgáltatásszintű metaadatok támogatása
- Transzparens protokollok és szemantika
- A biztonságos adminisztráció és adatcsere támogatása külső hálózatokkal

A 2.1.1. alfejezetben már említésre került az, hogy ajánlott, hogy minden központi rendszerből egy megtalálható legyen minden lokális felhőben. Ez azért fontos, mert a központi rendszerek kínálják azon központi szolgáltatásokat, melyek biztosítják az együttműködési követelményeket a felhők között: szolgáltatások regisztrálása, szolgáltatások felderítése, hitelesítés és orchesztrálás [11].

A koncepció egyik legfontosabb előnye az, hogy nagyon egyszerűen bővíthető, és a használatával egyre komplexebb architektúrák építhetőek. Ez abból adódik, hogy Arrowhead kompatibilis rendszerek együttműködésük révén System of Systems-é minősülnek és az így kialakult rendszerhalmazok szintén együtt tudnak működni. A 4. ábrán jól látszik, hogy az *A1, A2* és *A3* rendszerek halmaza adja az *A SoS*-t. Az *A* SoS könnyen együtt tud működni a *B1, C1* és *D1* rendszerekkel is újabb System of Systemst hozva létre. A *Dashboard MMI* egy interfészt nyújt a rendszer üzemeltetői számára, amelyen keresztül menedzselni tudják azt. Az *Arrowhead Verification Tool* komponens arra nyújt lehetőséget, hogy tesztelhető legyen egy-egy rendszer kompatibilitása az Arrowheaddel.



4. ábra: System of Systems komponensek az Arrowhead keretrendszerben

2.2 IoT felhők közötti együttműködés

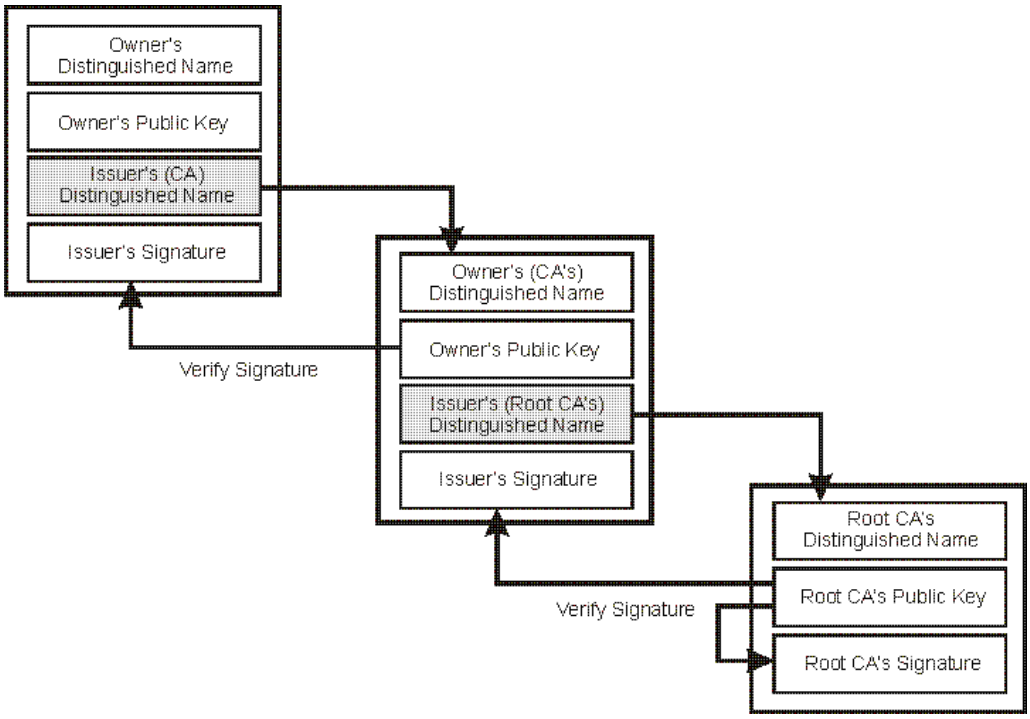
2.2.1 A tanúsítványok és használatuk általános megközelítése

A tanúsítvány alapú hitelesítés megismeréséhez szükség van néhány alapfogalom megértéséhez.

A kiinduló pont a PKI (Public Key Infrastructure, Nyilvános kulcsú infrastruktúra). A [12] definíciója alapján a PKI egy olyan keretrendszer, amely olyan protokollok használatán alapul, melyek biztosítják a hitelesítés, titkosítás, integritás, letagadhatatlanság és hozzáférés korlátozás megvalósítását digitális adatok kezelése során. Nyílt hálózatokban akkor garantálható ezen tulajdonságok megléte, ha minden fél identitása egyértelműen meghatározható és szavatolt. A biztonság területén laikus olvasó számára a PKI absztrakt fogalomnak tűnhet első hallásra, de bizonyára minden internetes felhasználó találkozik egyes implementációival nap mint nap, hiszen ide tartozik a böngészésnél használt SSL (Secure Socket Layer), illetve a biztonságos távoli elérésre használt SSH (Secure Shell). [13] A PKI működésének alapja a publikus (aszimmetrikus) titkosítási algoritmusok használata. Az aszimmetrikus titkosítási eljárás két logikailag és matematikailag összefüggő kulcsot kezel. Egy védendő digitális adatot a titokban tartott privát kulcsunk segítségével lehet aláírni. Eme aláírt adat hitelessége a publikus kulcs segítségével ellenőrizhetővé válik. A publikus kulcsú kriptográfia matematikai hátteréről bővebben is tájékozódhatunk a [14] forrásból.

A digitális tanúsítvány tehát olyan hitelesítési adatok összessége, melyek garantálják a nyilvános hálózatok résztvevőinek identitását egy a való életben is előforduló útlevélhez hasonlóan az előző bekezdésben részletezett PKI-ra alapozva. Egy digitális tanúsítványban akkor bízhatunk meg, ha azt egy erre szakosodott, megbízható szervezet adja ki. A tanúsítványokat kibocsátó entitásokat a szaknyelvben *Certificate Authoritynek (Hitelesítési szolgáltatónak)* nevezik, és gyakran hivatkoznak rá a CA rövidítést használva.

A tanúsítványok hierarchikusan léteznek és a gyökér mindig egy-egy CA, esetleg valamilyen *saját kezűleg aláírt tanúsítvány*. A saját kezűleg aláírt tanúsítványokat (self-signed certificate) ugyanaz az entitás írta alá, akinek a nevére szól. Tesztkörnyezetekben gyakran használják, bár biztonságtechnikai szempontból vitatható, hiszen az entitás bármikor generálhat egy új tanúsítványt, ahol más adatokat használ, kedvére változtathatja az érvényességi időt.

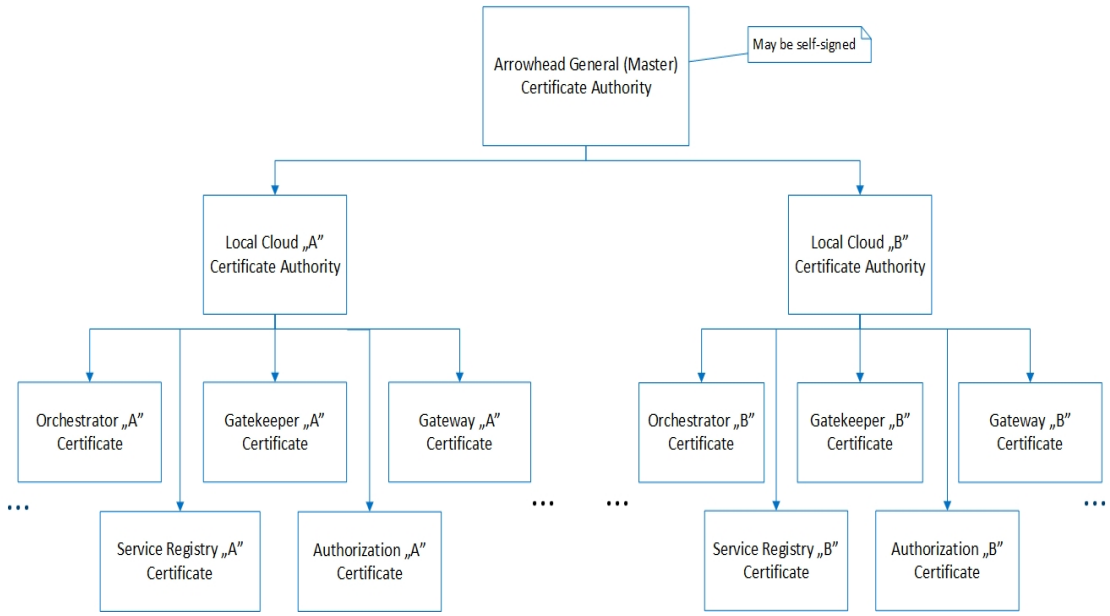


5. ábra: A tanúsítványok közötti bizalmi lánc (chain of trust) és összefüggések

A hierarchiából adódik tehát, hogy amennyiben megbízunk egy magasabb szintű csomópontban, akkor az alárendelt csomópontokban is bízhatunk, így alakul ki a *chain of trust*. A chain of trust egy rendezett listája a tanúsítványoknak, amely listának minden tagja megbízható. A gyökér tanúsítvány egy speciális saját kezűleg aláírt tanúsítvány, melyet a CA a saját kulcsának használatával írt alá. A lista úgy épül fel, hogy a gyökér

tanúsítvány aláír egy másik tanúsítványt, így az is megbízhatóvá válik, aki aláír egy következőt és így tovább. Az 5. ábra szemléletesen mutatja be a chain of trust listát és a tagjai között fennálló összefüggéseket.

2.2.2 A tanúsítványok szerepe az Arrowhead keretrendszerben

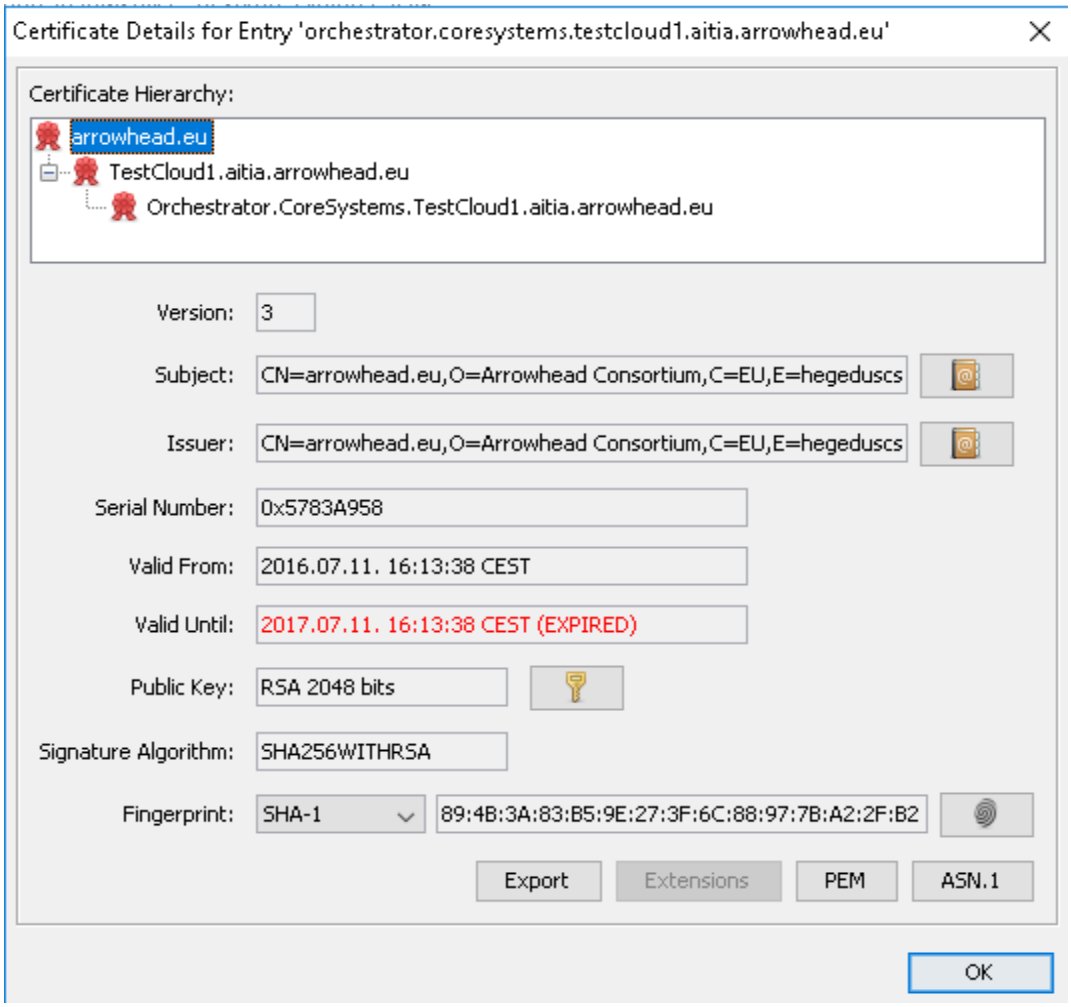


6. ábra: Tanúsítványok az Arrowhead keretrendszerben

Az előző alfejezetet áttekintve érezhető, hogy a tanúsítványok használata egy jól bevált és széleskörűen elfogadott módja a hitelesítésnek már hosszú évek óta. Az Arrowhead keretrendszer sem kerülte el a hagyományt és hasonló alapokra fektette a rendszerei hitelesítését. Az Arrowhead tanúsítványai egy hierarchikus architektúrát alkotnak, ahogyan az a 6. ábrán is látható. Minden tanúsítvány az *Arrowhead General (Master) Certificate* alá tartozik. A lokális felhők rendelkeznek saját tanúsítvánnyal és ezek alá vannak rendelve a központi rendszerek tanúsítványai. A hierarchikus felépítés elősegíti a 2.2.1. alfejezetben tárgyalt chain of trust kialakulását.

Minden központi rendszerhez, illetve lokális felhőhöz tartozik egy kulcstár, melyben tanúsítványok és kulcsok tárolhatók. A kulcstáraknak számos implementációja létezik, az Arrowhead jelenleg Java KeyStore (JKS) [15] fájlokat alkalmaz. A JKS fájlok megnyitásához több alkalmazás is rendelkezésre áll. Az egyik legelterjedtebb ezek közül a KeyStore Explorer [17], ami lehetőséget nyújt arra, hogy megtekinthetővé váljanak a kulcstárban található tanúsítványok, illetve privát és publikus kulcsok egyaránt.

Az 7. ábrán látható az Orchestrator központi rendszer tanúsítványa a KeyStore Explorer alkalmazáson belül.



7. ábra: A tanúsítványok tartalmának megjelenítése a KeyStore Explorer alkalmazásban az Orchestrator példáján keresztül

Az ábra felső negyedében vizuálisan is megjelenik az előző bekezdésben vázolt hierarchia. Alatta általános információk és hitelesítési adatok figyelhetőek meg. Néhány fontosabb kiemelve:

- a szabvány verziója;
- azon entitás adatai, akinek identitását igazolja a tanúsítvány;
- a kiállító CA adatai;
- a kiállítás dátuma;
- az érvényességi idő;
- publikus kulcs;
- az aláíráshoz használt hash algoritmus.

A digitális tanúsítvány tartalmi követelményeit az X.509 szabvány [18] definiálja. A tanúsítvány exportálható többek között DER bináris formátumba, PEM vagy PKCS#12 formátumba is [19].

2.3 Biztonsági kérdések

Ebben az alfejezetben szeretném kifejteni először az általános biztonsági elvárásokat a megismert keretrendszert tekintve. Az Arrowheaddel szemben támasztott fő biztonsági követelmények a **CIAAA** gyűjtőnevet kapták [24]:

- **Titkosítás** (Confidentiality) – Az adatokhoz csak az arra jogosultak férhessenek hozzá. A hamisítás és az információ közzététel a legnagyobb fenyegetés ebben az esetben.
- **Integritás** (Integrity) - Az adatok és az erőforrások csak arra alkalmas módon változhassanak meg arra jogosult személyek által. A szabotázs jelent veszélyt.
- **Hitelesség** (Authenticity) – a kommunikációban résztvevő felek azonosítani tudják egymást és csak olyan rendszerek számára legyen engedélyezve szolgáltatások igénybevétele, amelyek identitása bizonyított. A hamisítás jelenthet gondot.
- **Elérhetőség** (Availability) – a keretrendszernek mindig rendelkezésre kell állnia, amikor igény van rá. A túlterheléses támadások (Denial of Service, DoS) és az elosztott túlterheléses támadások (Distributed DoS, DDoS) támadások jelentik a legnagyobb veszélyt. Az erőforrások előrelátó és redundáns szervezése segíthet az elérhetőség biztosításában.
- **Elszámoltathatóság** (Accountability) – egy felhasználó se legyen képes letagadni olyan tevékenységeket, amelyeket ő vitt véghez. Ezt folyamatos naplózással lehet biztosítani.

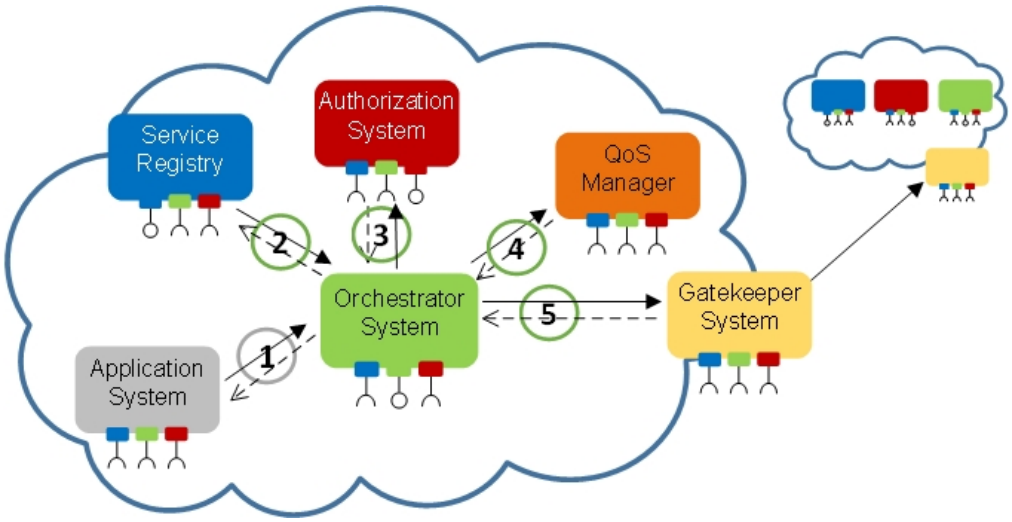
A tervezés során az imént felsorolt elvárásokat tartottam szem előtt és egy olyan egyszerű elven működő megoldást próbáltam találni, ami maradéktalanul kielégíti eme kritériumokat. A megoldás keresése alatt számos biztonságtechnikai kérdés merült fel. Az egyik legalapvetőbb kérdés*: hogyan döntsük el egy idegen lokális felhőről azt, hogy megbízható-e?* Erre több megoldás is létezhet. Az Arrowhead keretrendszernek jelenleg három ajánlása van, melyek közül egy még csak gondolati szinten él.

- Előre definiált, „megbízható” társfelhők – Ezen megoldás során előzetesen meghatározunk felhőket, amelyekről valamilyen megegyezés alapján úgy véljük, hogy megbízhatóak ránk nézve. A megbízható felhők listáját nyilvántartjuk (jelenleg egy könnyedén karbantartható MySQL adatbázisban [21]) és inter-cloud orchesztrálás esetén csak ezen felhőkkel lépünk kapcsolatba. Számos előnye mellett egy igazán hangsúlyos hátrány az, hogy nagyon korlátozott.
- Certificate (tanúsítvány) alapú – A 2.2.1. alfejezetben található részletezve.
- Megszavazott bizalom – Ezen elképzelés még nem része a keretrendszernek, csupán kilátásba van helyezve a jövőre nézve. A megoldás arra alapozna, hogy ha egy felhőről elég sok másik felhő véli úgy, hogy megbízható a tapasztalatai alapján, akkor mi is megbízunk benne.

3 IoT felhők közötti biztonságos adatkommunikáció

3.1 Miért fontos az Orchestrator, a Gatekeeper és a Gateway központi rendszer?

Ebben az alfejezetben szeretném egy példán keresztül szemléltetni azt, hogy miért elengedhetetlen az Arrowhead működése során az a 3.2., 3.3. és 3.4. fejezetekben bemutatott Orchestrator, Gatekeeper, illetve Gateway modul.



8. ábra: A központi rendszerek együttműködésének ábrázolása az orchesztrálás folyamata alatt [22]

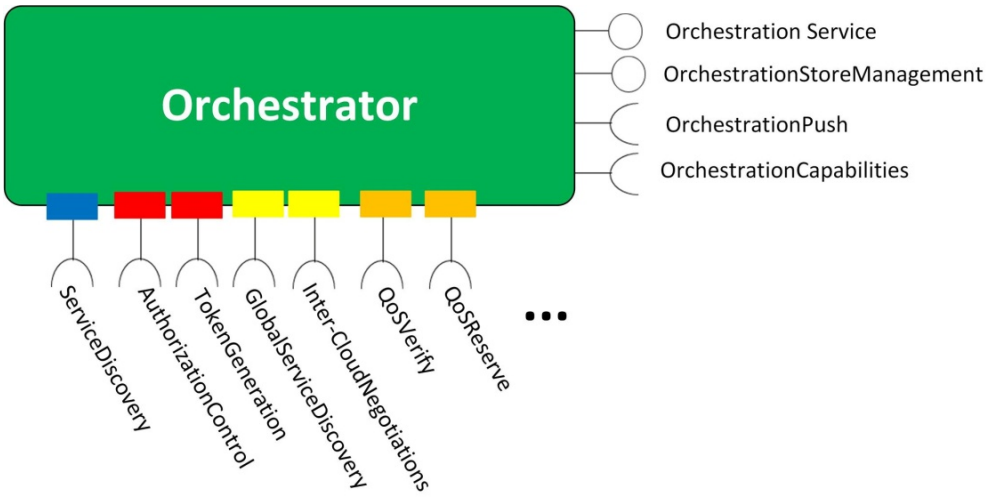
A 8. ábrán látható Application System egy tetszőleges rendszer egy helyi felhőn belül, ami valamilyen *X* szolgáltatást szeretne igényelni, ezért az Orchestratorhoz fordul, hogy megtudja ki alkalmas arra, hogy az adott szolgáltatást biztosítsa számára. Az Orchestrator a Service Registry Service Discovery szolgáltatásán keresztül kideríti, hogy milyen rendszerek regisztrálták magukat úgy, hogy képesek az *X* szolgáltatást nyújtani. Minden rendszer köteles regisztrálni az általa nyújtott szolgáltatásokat a helyi felhőjében található Service Registryben. Ajánlott, hogy egy bizonyos idő elteltével a modul metaadatokon keresztül ellenőrizni tudja a rendszer elérhetőségét, hogy biztosan csak használatra kész rendszerek szerepeljenek a Service Registry nyilvántartásában. A kötelező regisztráció által garantálható, hogy amennyiben a helyi felhőben van olyan rendszer, ami ki tudja szolgálni a kérést, arról az Orchestrator értesülni tudjon. Az Orchestrator az Authorization központi rendszerrel együttműködve elvégzi a

hitelesítéssel kapcsolatos teendőket és ezután amennyiben igény van a QoS szolgáltatásra a QoS Managerhez fordul. Az Authorization központi rendszer tokeneket generál az eddigi szűrések után megmaradt potenciális Providerek eléréséhez a Consumer számára és a tokenek listáját továbbítja az Orchestratornak. Ezen a ponton küldi el az Orchestrator egy üzenet formájában a Consumer számára azon rendszerek listáját, amelyek alkalmasak arra, hogy kiszolgálják a kérését.

Jól érzékelhető, hogy milyen megkerülhetetlen szerepe van az Orchestratornak a keretrendszer működésében. A példa egy lokális felhőn belüli szolgáltatásigénylést mutatott be; amennyiben az inter-cloud orchesztrálás is meg kell oldani, a feladat még összetettebbé válik. Ekkor válnak elengedhetetlenné a Gatekeeper és Gateway központi rendszerek, melyek megteremtik a lehetőséget a felhők közötti kapcsolatok létrehozásához.

3.2 Az Orchestrator működése és folyamatai

Az 9. ábrán látható Orchestrator modul [23] arra szolgál, hogy futási időben összeköttetéseket hozzon létre rendszerek között, így ez által az egyik szolgáltatást tudjon nyújtani a másik számára. Célja az, hogy információt biztosítson a rendszerek számára arról, hogyan és hová csatlakozhatnak az pillanatnyilag szükséges szolgáltatás igénybevételéért. Céljának teljesítése során lényegében SoS-eket jönnek létre a rendszerek összekapcsolódása révén.



9. ábra: Az Orchestrator modul és szolgáltatásai

Az Orchestrator két központi szolgáltatást nyújt, ezek az *Orchestration Service*, illetve az *Orchestration Store Management*. A 9. ábrán látható, hogy számos

szolgáltatást tud hasznosítani a modul attól függően, hogy mire van igény a keretrendszer felhasználóinak részéről. Képes rá, hogy beépítse az orchesztrálás folyamatába a tokengenerálást, QoS szolgáltatást vagy épp a 3.3.1. és 3.3.2. alfejezetben tárgyalt Global Service Discoveryt, illetve InterCloud Negotiationst.

3.2.1 Orchestration Service

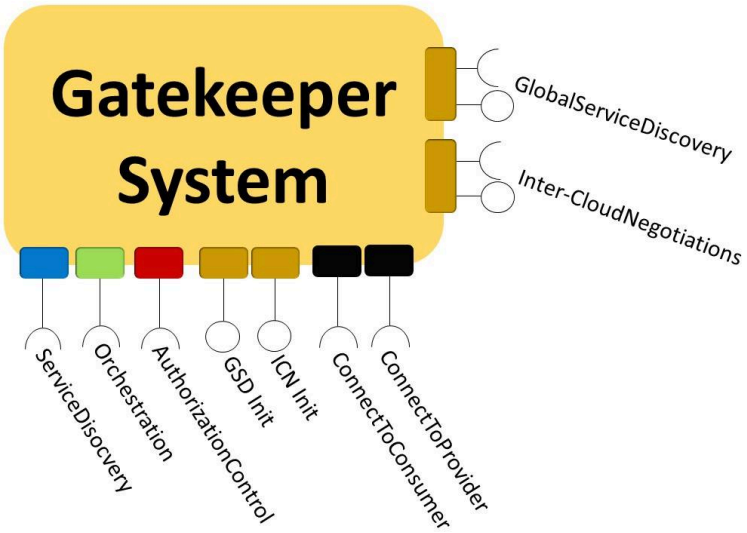
Az *Orchestration Service* szolgáltatás célja az, hogy kielégítse a felébredése után vagy működése közben valamilyen szolgáltatást igénylő rendszerek kérését, megfelelő kiszolgáló rendszert találva számukra. Ez a folyamat a 3.1.-ben leírtaknak megfelelően megy végbe.

3.2.2 Orchestration Store Management

Az Orchestrator központi rendszeren belül található egy adatbázis, mely az *Orchestration Store* nevet viseli. Ezen adatbázisban a rendszerek között előre megtervezett és definiált összefüggések tárolódnak. Amennyiben *X* rendszer *S* szolgáltatást szeretné igénybe venni és az adatbázisban található egy olyan bejegyzés, amely az *X* és *Y* rendszert, illetve az *S* szolgáltatást összerendeli, akkor *X* kérését lehetőleg az *Y* rendszer fogja kiszolgálni. Az *Orchestration Store Management* ezen bejegyzések kezelésével foglalkozik.

3.3 A Gatekeeper működése és folyamatai

Az 10. ábrán bemutatott Gatekeeper egy az Arrowhead keretrendszer alapjául szolgáló „Core System”-ek közül. Amikor egy Consumer igénybe szeretne venni egy szolgáltatást, amely egy másik lokális felhő Provideréhez tartozik, a szokott orchesztráláson kívül más feladatok is felmerülnek. Ezen feladatokat látják el a Gatekeeper által nyújtott szolgáltatások: a Global Service Discovery, illetve az Inter-Cloud Negotiations.



10. ábra: A Gatekeeper modul és szolgáltatásai [25]

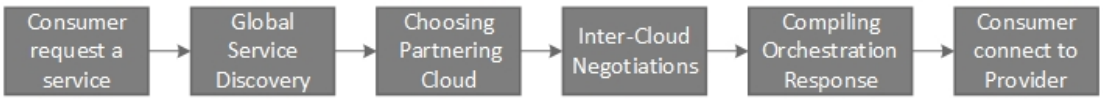
Könnyen belátható, hogy szükség van arra, hogy legyen egy modul, ami kezeli a felhők közötti szolgáltatás-felfedezésből és összekapcsolás-támogatásból (a későbbiekben: inter-cloud orchesztrálásból) adódó helyzeteket. Elsősorban lehetőséget kell biztosítanunk, hogy egy idegen lokális felhőben is meg tudjuk találni a számunkra megfelelő Providert, azaz kiszolgáló rendszert. Vegyünk egy olyan példát, amelyen keresztül egy, a témában laikus olvasó számára is érthetővé válik ez az igény. Elképzelhető egy olyan helyzet, hogy egy nagyobb nemzetközi projekt részeként több cég rendszere is együttműködik. Ekkor hamar elemi szükségletként jelentkezik az, hogy adatokat cseréljenek egymással. Ilyen adat lehet szemléltetésként egy hőmérsékletadat a másik cég lokális felhőjében elhelyezkedő szenzortól vagy egy páratartalmat leíró adat. Az is elképzelhető, hogy egy cég rendszere olyan méreteket ölt, hogy több lokális felhő fedi le a területét, például minden városban vagy országban elhelyezkedő központját egy-egy. Ekkor még égetőbb igény az, hogy biztonságosan adatot tudjanak cserélni a földrajzilag szétszórt székhelyek.

Ilyenformán a felhők közötti kiszolgálásra olyan esetekben lehet igény, amikor

a. Nem lehetséges a lokális felhőn belül a kiszolgálás, mivel nincs olyan Provider, akinél megvan a megfelelő adat
b. Provider van ugyan, aki képes lenne szolgáltatni a megfelelő adatot, de éppen nem elérhető valamilyen okból kifolyólag
c. A kiszolgálás nem lehetséges, mivel a Provider nem rendelkezik szabad erőforrással vagy a QoS követelményeknek nem felel meg
d. A kiszolgálás nem lenne optimális adott lokális felhőn belül (pl. létezik földrajzilag közelebb is alkalmas lokális felhő) [26]

3.3.1 A Global Service Discovery szolgáltatás

A 11. ábra egy sematikus folyamatábra, amely segíthet abban, hogy átláthatóbbá váljon mely fontosabb momentumok közé ékelődnek be a Gatekeeper szolgáltatásai. A későbbiekben ismertetett Gateway működéséhez tartozó lépések ezen a ponton még nincsenek kifejtve a folyamatábrán.



11. ábra: A felhők közötti kapcsolatteremtés fontosabb lépéseit bemutató folyamatábra

A *Global Service Discovery (GSD)* során idegen Arrowhead lokális felhőket keresünk, amelyek képesek kiszolgálni az adott kérést. A Gatekeeperrel REST [28] interfészen keresztül kommunikálhatunk. A GSD eljárás funkcionálisan két részre bontható és mindkettő külön erőforráson keresztül érhető el. A folyamat első része a consumer-oldali inicializáció. Ennek a keretein belül a Gatekeeper kap egy kérést az Orchestratortól, amely tartalmaz minden olyan információt, amely szükséges az alkalmas felhők megtalálásához. A folyamat második része az, amikor a kapott információt továbbítja a modul az összes olyan idegen felhő Gatekeepere számára, akivel kapcsolatban áll.

3.3.2 Inter-Cloud Negotiations szolgáltatás

A 7. ábrán is megjelenő *Inter-Cloud Negotiations (ICN)* során a kölcsönös hitelesítés után a kialakult együttműködés paramétereit rögzítik és határozzák meg a kommunikációban résztvevő felek, megpróbálva közös nevezőre jutni.

Az ICN folyamat a következő feladatok elvégzését foglalja magába a [27] alapján:

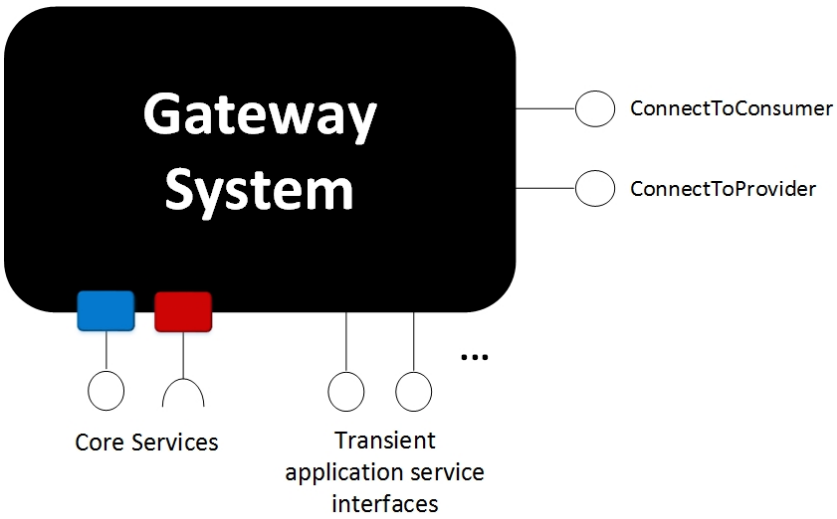
- verzió és protokoll egyeztetése
- kölcsönös hitelesítés és a személyazonosság ellenőrzése
- engedélyek kezelése és erőforrások allokálása a felhőkben a tranzakció végrehajtásához
- biztonságos kapcsolat kiépítése a Gatekeeperek között
- a helyi Service Discovery adatok megosztása a partnerfelhővel
- ideiglenes engedélyekkel kapcsolatos információk tárolása a partnerfelhő Authorization központi rendszerében
- a partnerfelhő létrehoz és megoszt egy ideiglenes tokent a kezdeményező felhő Gatekeeperével
- a kezdeményező helyi felhő továbbítja a kapott tokent a kezdeményező rendszer számára
- a rendszer csatlakozik a kiszolgáló félhez a partnerfelhő központi rendszereit használva

3.4 A Gateway működése és folyamatai

Munkám legfontosabb része a Gateway folyamatainak tervezése és megvalósítása volt.

Az ipari rendszereknél különösen fontos az, hogy az adatok biztonságban legyenek, hiszen rengeteg a konkurens vállalat, akik ugyanolyan területen tevékenykednek. Manapság elengedhetetlen az, hogy megvédjük az üzleti titkokat, a rendszerünk működésének részleteit. Könnyen belátható, hogy amint kilépünk a lokális felhő keretei közül, sebezhetővé válunk egy harmadik, rosszindulatú fél számára. A kapcsolat lebonyolítása közben beékelődhet a szolgáltatás igénylő és a kiszolgáló közé egy támadó, aki lehallgatja, vagy rosszabb esetben megváltoztatja az üzenetváltás menetét.

Mindezek miatt a lokális felhők közötti kommunikáció megvalósítása során az egyik legfontosabb elem a 12. ábrán bemutatott Gateway. A tervek szerint az Arrowhead keretrendszer ezzel az új központi rendszerrel bővül. Ez a modul az adatútban van, és annak felépítésével és biztonságossá tételével foglalkozik a kezdeményező Consumer és a kiszolgáló Provider között. A felhők közötti, inter-cloud kommunikáció egyik legnagyobb problémája az, hogy hogyan legyen a létrejövő adatút biztonsági szempontból megfelelő.



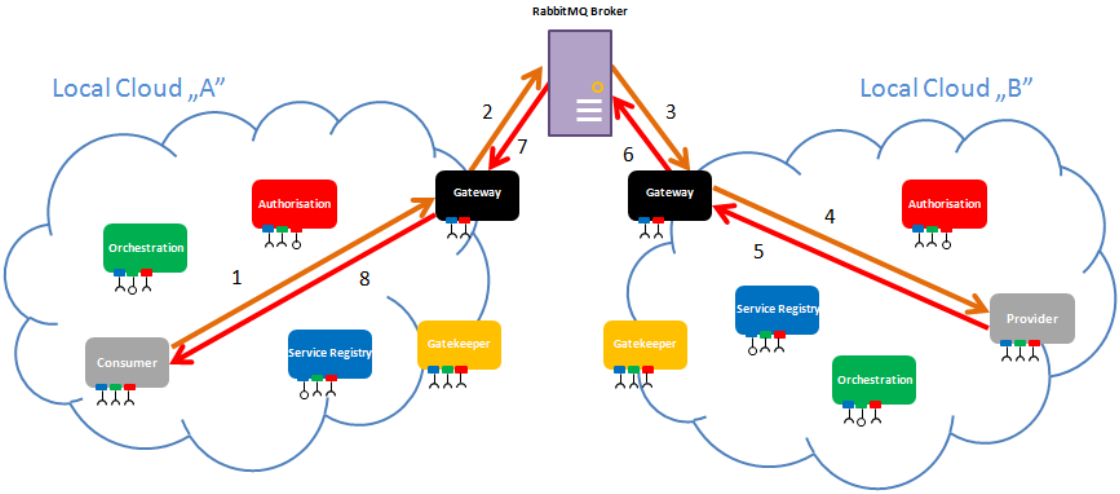
12.ábra: A Gateway modul és szolgáltatásai

Egy lokális felhő zárt informatikai rendszernek tekintendő, a felhőben elhelyezkedő rendszerek (legyen szó központi rendszerről vagy egy szolgáltatást nyújtó

rendszerről) a felhő határain kívül eső entitásokat nem érzékelik. Amikor két ilyen zárt felhő kapcsolatba szeretne lépni egymással, akkor szükség van felhőnként egy-egy olyan modulra, amelyet még érzékelnek a helyi rendszerek és megbízhatónak tekintik. Ez a modul a Gateway, amely tehát a lokális felhő határán helyezkedik el egy utolsó biztos és megbízható pontot nyújtva a központi rendszerek számára. A Gateway szorosan együttműködik a Gatekeeperrel, ami biztosítja az alkalmas szolgáltatások felderítését az idegen felhőben, illetve elősegíti a hitelesítést és az együttműködés gördülékenységét. A Gatekeeper veszi fel a kapcsolatot a Gateway-el REST interfészen keresztül. A két modul együttműködésének technikai részletei a 4.1. alfejezetben találhatóak.

A Gateway két szolgáltatást nyújt, melyek a *Connect To Consumer*, illetve a *Connect To Provider* nevet viselik. Azért került különválasztásra a szolgáltatást igénylő és biztosító rendszer oldalán található Gatewayhez történő csatlakozás, mert mindkét esetben más feladatokat kell megvalósítani.

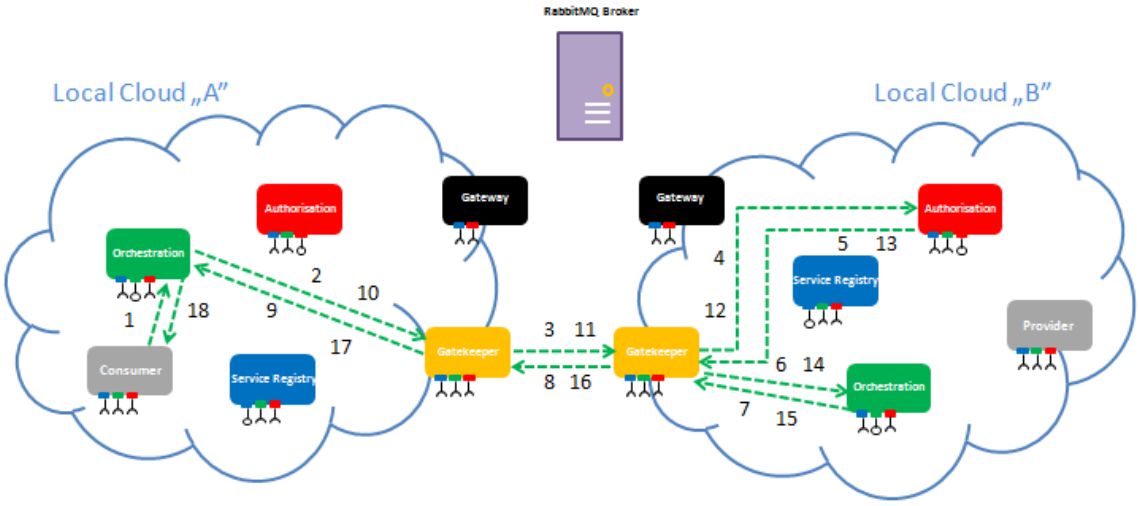
3.4.1 Orchesztrálás és az adatút kiépítése inter-cloud környezetben



13. ábra: Az adatok által bejárt út az inter-cloud orchesztrálás során

A 13. ábra szemlélteti az adat tervezett útját az inter-cloud orchesztrálás során. A narancssárga szín jelöli a Consumer kérésének, míg a piros szín a Provider válaszának az útját. A Gateway transzparens a Consumer és Provider számára. A tervezési fázis során úgy állapítottuk meg, hogy az egyik legfontosabb követelmény a modullal szemben az, hogy a Consumer és Provider működésén ne kelljen változtatnunk. A Consumer és a Provider is kölcsönösen úgy érzékelik, hogy közvetlenül egymáshoz

csatlakoznak és nincs tudomásuk arról, hogy a Gateway-ek is az adatútban ékelődtek. A két Gateway-t esetünkben a RabbitMQ Broker (lásd a következő fejezetben) köti össze, de más AMQP vagy MQTT implementációk is működőképesek lehetnek, mint például Apache Qpid, ActiveMQ, Apache Apollo.



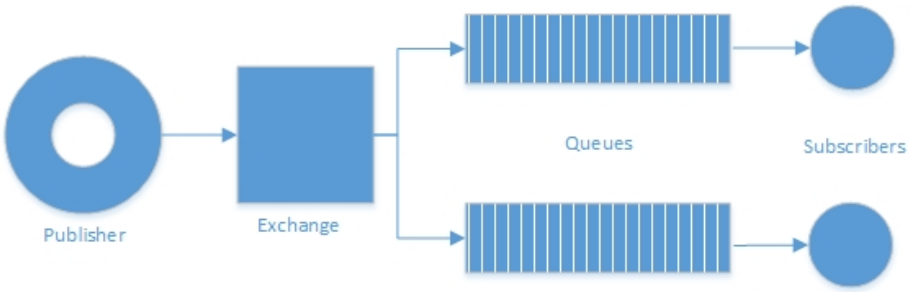
14. ábra: Az orchesztrálás útvonala inter-cloud környezetben

Érzékeltetésképpen a 14. ábra az orchesztrálás útvonalát mutatja be, mely teljesen eltérő az adatúttól. Ebben a Gateway nem vesz részt, a Gatekeepereken keresztül cserélődik minden orchesztráláshoz szükséges információ a két helyi felhő között az InterCloud Negotiations folyamat során.

3.4.2 Alkalmazott technológiák

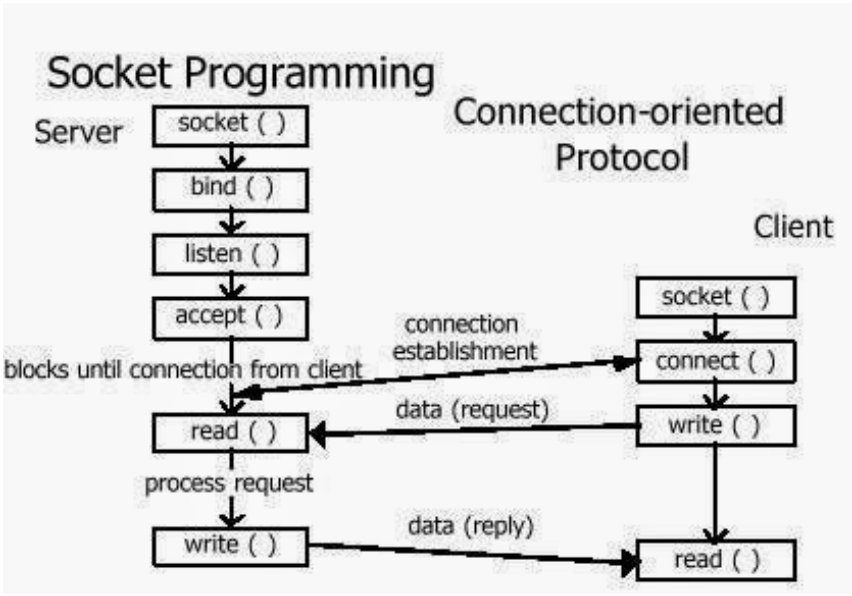
A két Gateway, azaz a szolgáltatást igénylő és a szolgáltatást biztosító rendszer oldalán elhelyezkedő Gateway-ek közötti kommunikáció egy RabbitMQ szerver installálásával és konfigurálásával képzeltük el. A RabbitMQ [28] az AMQP [29] egy megvalósítása, míg az AMQP, azaz Advanced Message Queuing Protocol, az MQTT széleskörűen elterjedt technológia egy továbbfejlesztett változata. A protokoll működését a 15. ábra mutatja be.

A RabbitMQ implementáció a Publish-subscribe modellt [30] követi. Úgy nevezett „*queue*”-kat definiálhatunk, majd ezeken keresztül üzeneteket küldhetünk. A kliens feliratkozhat az általunk definiált queuera és így azonnal megkapja az ide publikált üzeneteket. A RabbitMQ céljainknak megfelelő konfigurálása részletesen tárgyalásra kerül a 4.2. fejezetben.



15. ábra: Az AMQP protokoll

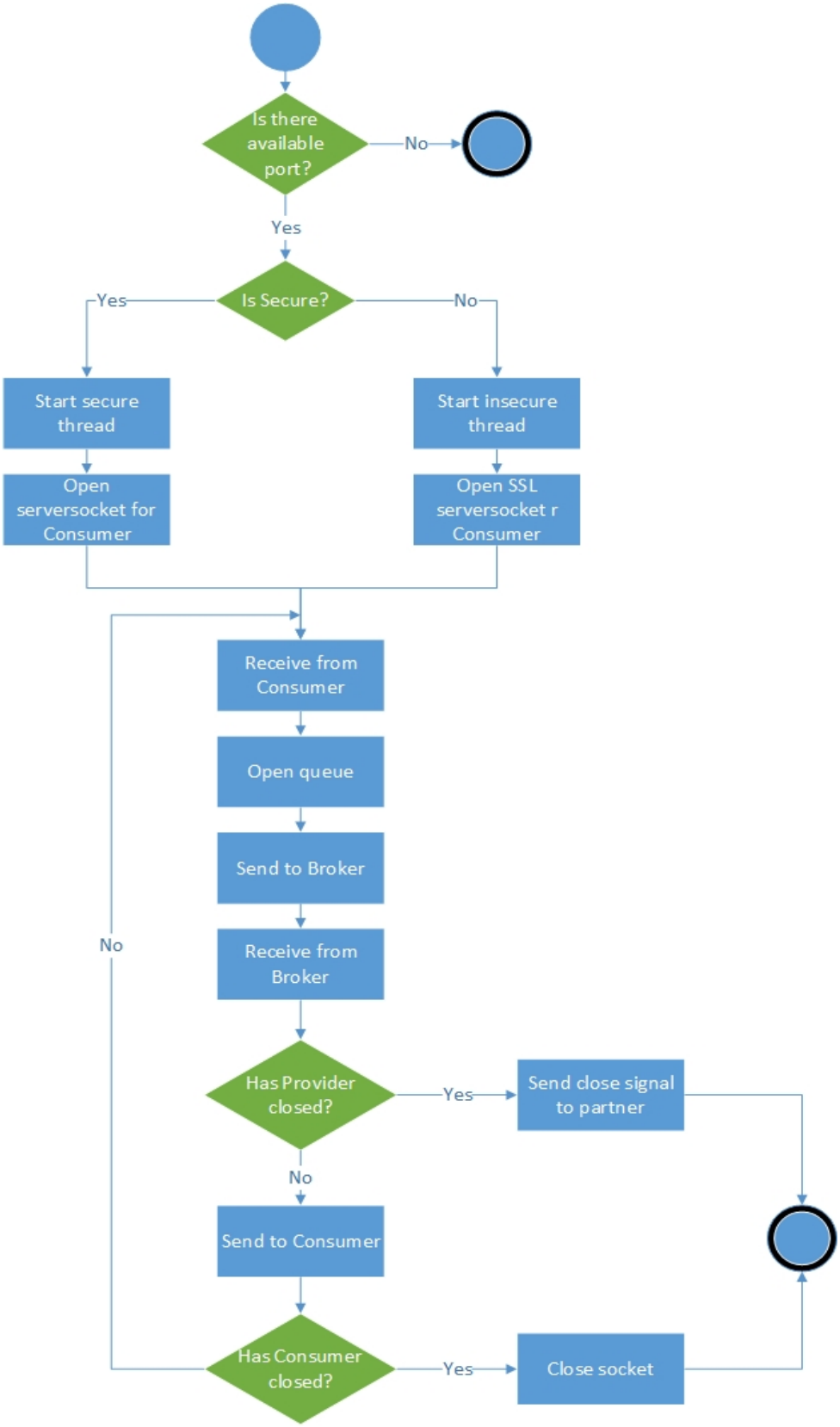
Az adatút teljes kiépítéséhez a RabbitMQ mellett a socketek [31] nyújtanak segítséget. A *socket* definíciója szerint egy olyan absztrakció, mely Internet Protocol alapú számítógépes hálózatokban valamely kétirányú kommunikációs hálózati folyam végpontját jelenti; használata általánosan a 16. ábrán van szemléltetve. Megkülönböztetünk kliens és szerver oldali socketeket. A szerver oldali socket egy kapcsolódási kísérletre vár valamely kliens részéről. A kliens oldali socket úgy tud kapcsolódni a szerver oldali sockethez, ha ismeri a hozzárendelt IP címet és portszámot.



16. ábra: Java socketek használata [32]

A továbbiakban szeretném külön alfejezetekben részletezni a Gateway által nyújtott szolgáltatások működését, melyek használják mind a RabbitMQ Broker, mind a socketek nyújtotta lehetőségeket.

3.4.3 Connect To Consumer szolgáltatás



17. ábra: Folyamatábra arról, hogyan csatlakozik a Gateway modul a Consumerhez

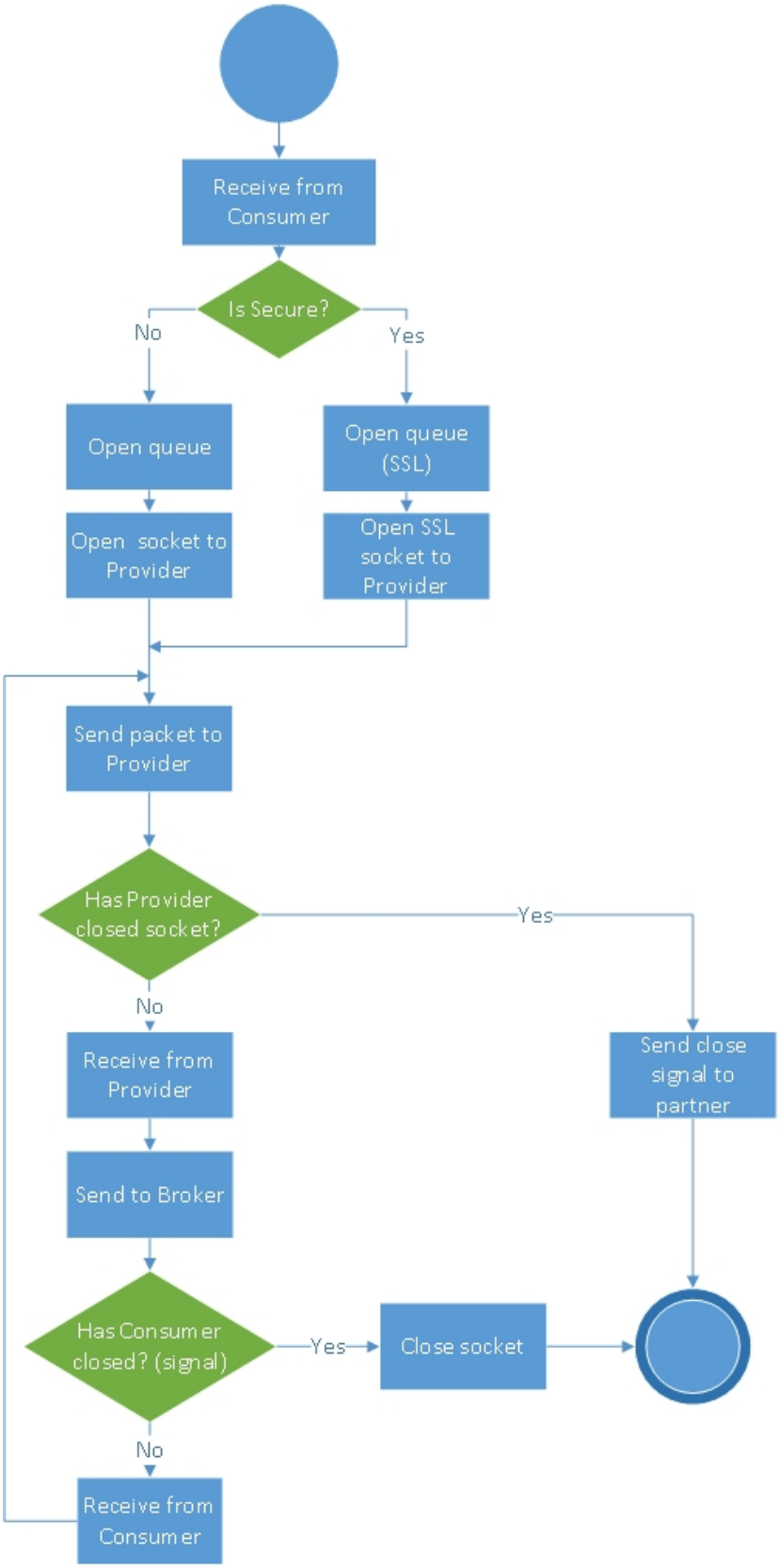
A 17. ábrán látható folyamatábra a Connect To Consumer szolgáltatás működését szemlélteti. Minden inter-cloud szolgáltatásigénylő rendszer külön szerver oldali socketet kap, melyhez kapcsolódhat, így az első elágazás során megvizsgáljuk, hogy rendelkezésre áll-e még szabad port, amelyhez csatlakozni tud új Consumer. Amennyiben van ilyen port, eldöntésre kerül, hogy az adatokat titkosítva továbbítjuk-e. A döntés mindkét kimenetele esetén elindul egy új szál, amely kezeli a kérést, illetve létrejön egy szerver oldali socket, amely vár a Consumer csatlakozására. A titkosítás esetén SSL alapú socketek vesznek részt a kommunikációban. A socketen keresztül érkezik meg a Consumer kérése a Gateway modulhoz. A kérés továbbításához először szükség van egy *queue* definiálására azért, hogy képes legyen a Gateway a RabbitMQ Brokerhez kapcsolódni. Miután létrejött a queue, rajta keresztül továbbítjuk a Consumer kérését a Broker felé, illetve a használatával kapjuk meg a Provider válaszát is.

Amennyiben a Provider lezárta a socketet, a kommunikációs partnernek küldünk egy jelzést erről és véget ér a folyamat.

Abban az esetben, ha nem zárta le a socketet, a Consumer felé nyitott socketen keresztül elküldjük számára a Provider válaszát.

Fontos, hogy ilyenkor a Consumer lezárja-e a socketet. Amennyiben igen, a folyamat véget ér. Ellenkező esetben visszatérünk a jelzett ponthoz a folyamatábrán és újra végrehajtódnak a feltüntetett lépések.

3.4.4 Connect To Provider szolgáltatás



18. ábra: Folyamatábra arról, hogyan csatlakozik a Gateway modul a Providerhez

A 18. ábrán látható folyamatábra segít vizualizálni a *Connect To Provider* szolgáltatás működését. Miután kaptunk egy kérést a Consumertől, a folyamatábra tetején látható egy elágazás, amely azt határozza meg, hogy szeretnénk-e titkosítást használni az adatok küldése során. Az elágazás mindkét kimenetele esetén először egy RabbitMQ csatornát és egy már említett *queuet* definiálunk a RabbitMQ Broker elérése érdekében, majd egy socketet nyitunk a Provider felé, hogy azon keresztül továbbítani tudjuk a beérkezett kérést. A különbség abban mutatható ki, hogy a biztonságos mód esetén SSL használatával kapcsolódunk a Brokerhez, illetve SSL-t használó socketet hozunk létre. Ezzel biztosítjuk azt, hogy a kommunikáció során elküldött adatok titkosított csatornán haladjanak keresztül.

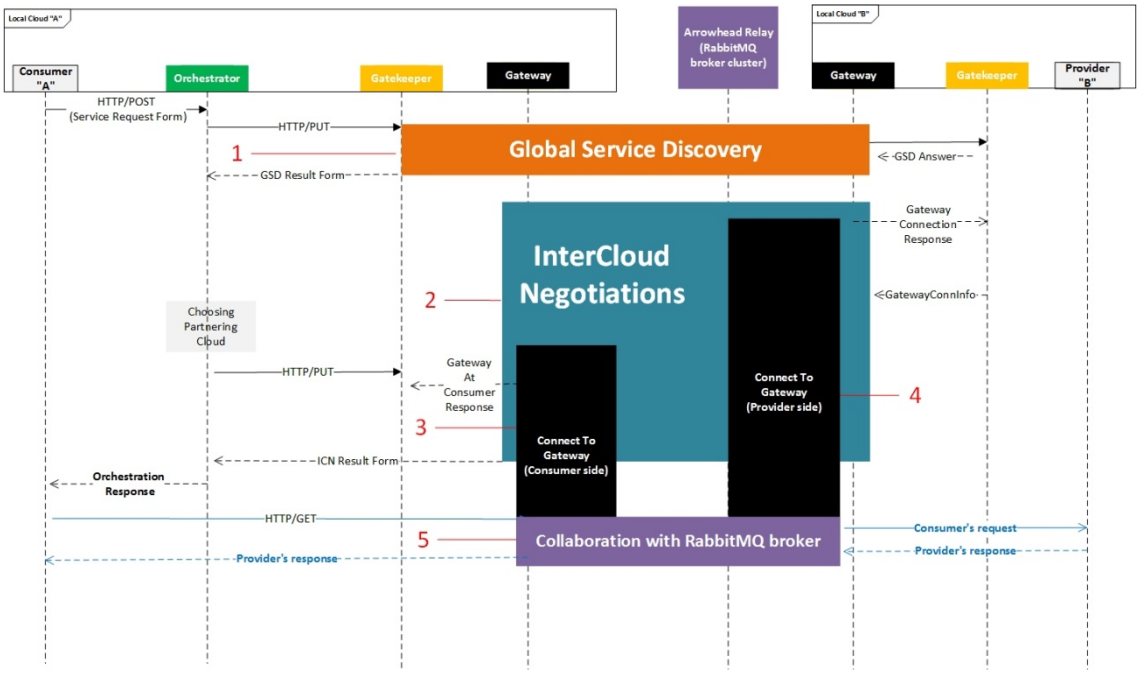
A következő lépésben a szolgáltatást igénylő rendszertől kapott kérést továbbítjuk a Provider felé felhasználva az irányába nyitott socketet.

Amennyiben a Provider lezárta a socketet, a kommunikációs partnernek küldünk egy jelzést erről és véget ér a folyamat.

Abban az esetben, ha nem zárta le a socketet, rajta keresztül megkapjuk a választ a kérésre és továbbítjuk a kapott információt a Brokerhez.

A továbbiakról az dönt, hogy a Consumer lezárta-e a socketet. Amennyiben igen, a socket lezárása után véget ér a folyamat. Ellenkező esetben megkapjuk az új kérést a Consumertől és ciklikusan végrehajtjuk a lépéseket újra az ábrán jelölt ponttól kezdődően.

3.5 Egy tipikus példa a rendszer együttes működésére

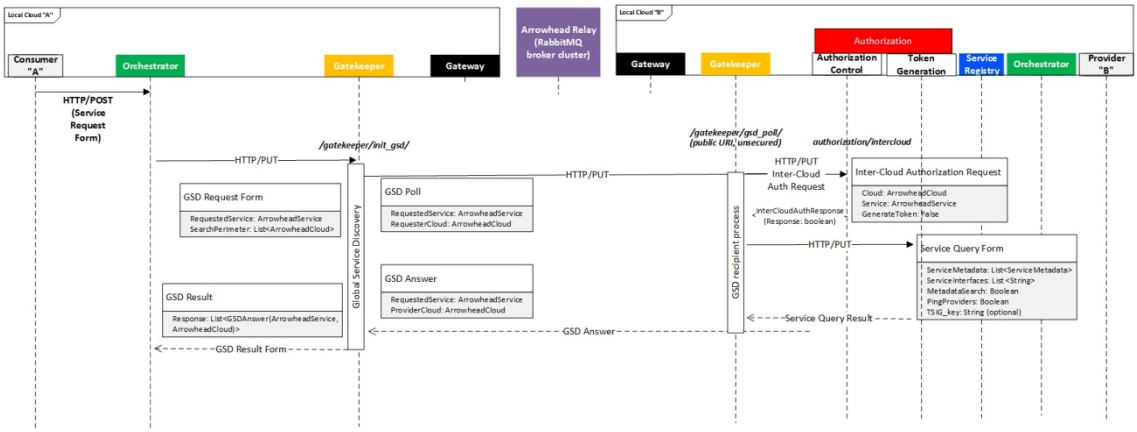


19. ábra: A központi rendszerek együttműködése vázlatosan bemutatva

Ebben az alfejezetben szeretnék demonstrálni egy tipikus példát, amely szemlélteti hogyan működnek együtt a központi rendszerek effektíven a lokális felhők közötti kommunikáció megteremtésében. A 19. ábra a központi rendszerek együttműködését mutatja be. Az ábra célja nem a részletes üzenetváltások megjelenítése, hanem az, hogy egy leegyszerűsített vázlatot prezentáljon a könnyebb érthetőség kedvéért.

Az együttműködés során végbemenő részletes üzenetváltás az 1. függeléken látható egy átfogó szekvencia diagram keretében. A kiinduló helyzet az, hogy a Consumer szeretne igénybe venni egy szolgáltatást. Ilyen esetben az első lépést mindig az jelenti, hogy a Consumer a saját lokális felhőjében található Orchestratorhoz fordul REST interfészen keresztül. Már ennél a pontnál jelzi a modul számára a kérésében foglalt metaadat segítségével, hogy engedélyezi az inter-cloud orchesztrálást. Amennyiben a kérés nem teljesíthető lokálisan, az Orchestrator felveszi a kapcsolatot az ugyanazon lokális felhőben elhelyezkedő Gatekeeperrel és kezdeményezi a 3.3.1 alfejezetben már említett *Global Service Discovery* folyamatot. A GSD eljárás a 19. ábrán 1-es sorszámmal van jelölve. A 20. ábra prezentálja a folyamat alatt végbemenő üzenetváltásokat. Ezen folyamat részeként a Gatekeeper próbál találni egy vagy több alkalmas idegen lokális felhőt, amely biztosítani tudja a kért szolgáltatást. A modul a

kérés teljesítésére megfelelőnek ítélt felhők listáját válaszként visszaküldi az Orchestrator számára, aki ekkor kiválaszthat különféle szempontokat figyelembe véve egy partner felhőt, akivel a Gatekeeperen keresztül megkezdi a *Inter-Cloud Negotiations* műveletet (lásd 3.3.2 alfejezet)



20. ábra: A Global Service Discovery folyamat üzenetváltásai

Az 19. Ábrán 2-es sorszámmal jelölt *Inter-Cloud Negotiations* során több kulcsfontosságú lépés megy végbe. A Gatekeeper kapcsolatot teremt a választott lokális felhőben helyet foglaló Gatekeeperrel és megtörténik a token-alapú hitelesítés az Authorizációért felelős modul segítségével. A következő lépésben az idegen lokális felhő Gatekeepere átveszi a kiszolgáló Providerrel és a kért szolgáltatással kapcsolatos releváns információkat az Orchestratorától. Ezt követően csatlakozik a Gatewayhez.

A 19. Ábra 3., 4., illetve 5. sorszámú folyamatai részletezésre kerültek már az előző fejezetben.

4 Megvalósítás

4.1 A Gatekeeper

A Gatekeeper központi rendszer megvalósításáról részletes tájékoztatást ad a [34], mivel ennek az implementációját nem én készítettem el, csupán néhány kisebb helyen módosítottam annak érdekében, hogy együtt tudjon működni a Gateway modullal.

A *Global Service Discovery* szolgáltatás nem változott. Az *InterCloud Negotiations* szolgáltatás inicializáló részéhez került a Gatewayhez való csatlakozás REST interfészen keresztül. A Gatekeeper és a Gateway szabványos kommunikációjának megteremtéséhez négy új üzenet típus hoztam létre. A Consumer felhőjében és a Provider felhőjében található Gatewayhez két különböző üzenet segítségével lehet csatlakozni.

A Provider lokális felhőjében elhelyezkedő Gateway válasza szolgáltatja a RabbitMQ Broker használatához szükséges információkat a Consumer felhőjének a részére.

A Consumer lokális felhőjében elhelyezkedő Gateway válasza pedig azt a serverSocketPort-ot tartalmazza, amelyhez a Consumernek csatlakoznia kell. Ezt az információt a Gatekeeper továbbítja a Consumer számára, aki ezután csatlakozni tud a Gatewayhez úgy, hogy közben nem érzékeli azt, hogy nem közvetlenül a Providerhez kapcsolódott.

4.2 A Gateway

4.2.1 A RabbitMQ Broker konfigurálása

A 3.4 fejezetben említett RabbitMQ Broker/Szerver telepítése viszonylag könnyen megoldható. A BME-TMIT laborjában működő, Unix alapú mantis2 és mantis3 szerverekre telepítettem a RabbitMQ Szervereket.

A telepítés után az első lépés a RabbitMQ-hoz tartozó konfigurációs fájl szerkesztése volt. Az alapbeállítások szerint az alapértelmezett *„guestuser”* csak a helyi hálózaton keresztül, azaz localhoston csatlakozhat a szerverhez, a konfigurációs fájlhoz egy megfelelő sort adva azonban elérhetjük azt, hogy távolról is használható legyen a szerver. Fontos, hogy megfelelő jelszót válasszunk és ne becsüljük alá ezt a kérdést, mert ez is komoly biztonsági rést eredményezhet. A szervert szeretném biztonságos módban, azaz SSL tanúsítványok igénybevételével is használni, ez szintén beállítható a konfigurációs fájlban. Többek között ki kell választani az SSL portot, amelyen hallgatni fog a szerver, tehát azt a portot, ahol várja a kapcsolatokat, illetve szükség van PEM formátumban a KeyStore és a TrustStore elérési útjára. A konfigurációs fájl tartalma a 21. ábrán látható.

```
[
  {rabbit, [
        {loopback_users, []},
        {ssl_listeners, [5671]},
        {ssl_options, [{cacertfile,"/etc/rabbitmq/ssl/testcloud1_cert.pem"},
                       {certfile,"/etc/rabbitmq/ssl/gatekeeper.testcloud1.crt.pem"},
                       {keyfile,"/etc/rabbitmq/ssl/gatekeeper.testcloud1.key.pem"},
                       {verify,verify_peer},
                       {fail_if_no_peer_cert,true}]}
]
  }
].
```

21.ábra: A RabbitMQ konfigurációs fájl tartalma

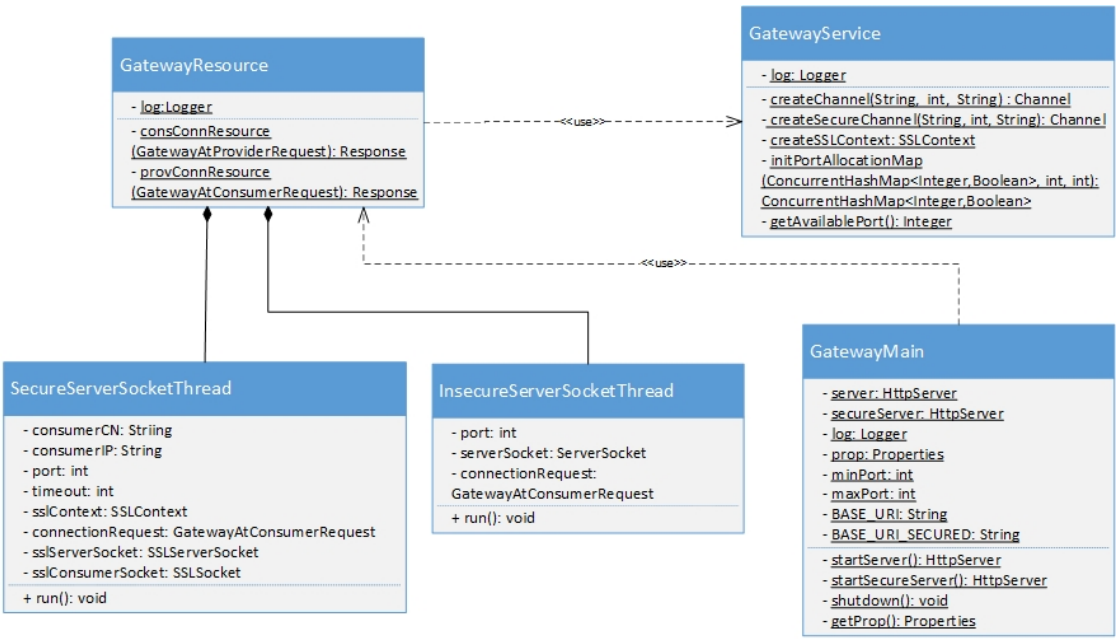
A mantis2 és mantis3, pontosabban a mantis2.tmit.bme.hu és a mantis3.tmit.bme.hu szerverekre telepített RabbitMQ szervereket *clusterbe* fűztem [35], majd a „*messagemirroring*” opciót is alkalmaztam, így amennyiben az éppen a kliens által használt szerver valamilyen okból kifolyólag elérhetetlenné válik, az elküldött, de még nem továbbított üzenetek nem vesznek el, mivel egy lemásolt példányuk a másik szerveren még elérhető ilyenkor is. Amennyiben az üzenetküldés közben áll le az a RabbitMQ szerver, amelyhez csatlakoztunk, a kapcsolat megszakad. Erre nem nyújt

megoldást a RabbitMQ implementáció. Meggyőződésük szerint ez a probléma több RabbitMQ Broker, vagy ahogyan az előzőekben hivatkoztam rá, szerver felsorolásával a kliens számára megoldandó probléma.

4.2.2 A socketek használata kétirányú kommunikáció megteremtéséhez

Az 3.4. alfejezetben említett socket típusokat a Java nyelv *serverSocket* és *Socket* néven implementálja alapesetben, illetve létezik ezek biztonságos párja, az *SSLServerSocket* és az *SSLSocket* is. Az én implementációm Java nyelven készült, de más programozási nyelvekben is megtalálhatóak ezek megfelelői, így eltérő környezetben is akadálymentesen implementálható a bemutatott elv.

A kérő és a fogadó oldali interfész különböző viselkedést valósít meg. Minden Consumer által indított kérés saját szálat kap, emiatt két ServerSocketThread osztály is implementálásra kerül a Gateway modulon belül a biztonságos, illetve a nem SSL alapú kommunikáció számára. Többek között az említett két osztály is megtalálható a 22. ábrán látható osztálydiagramon, amely a modulhoz tartozó osztályok viszonyát hivatott ábrázolni.



22. ábra: A Gateway modulhoz tartozó osztálydiagram

Egy szálbiztos ConcurrentHashMap segítségével tartom nyilván a még elérhető portokat, amelyek egy olyan tartományból kerülnek ki, amelynek felső és alsó korlátját a properties fájlban definiált minPort és maxPort határozza meg. Minden Consumer

számára visszaküldünk tehát egy severSocketPortot, amelyet így zavartalanul használhat a kérése kiszolgálására.

A megvalósításhoz szükséges elvek, információk és folyamatábrák a 3.4.3., illetve 3.4.4. alfejezetekben tárgyalásra kerültek. Ennek a dolgozatnak nem célja még részletesebben foglalkozni a fejlesztés során felmerült Java specifikus kérdésekkel, mivel az említett fejezetek alapján implementálható a tervezet más programozási nyelvek használatával is.

4.3 A Service Provider és Consumer

A tervezés első fázisában felmerült, hogy változtassunk a Provider, illetve a Consumer működésén, de a későbbiekben a Gateway funkcionalitásának köszönhetően erre nem volt szükség. Ez azt eredményezi, hogy a rendszer visszafelé is kompatibilis marad, így régebbi Consumer és Provider rendszerek is zavartalanul használni tudják az új modult és az általa nyújtott biztonságos együttműködési lehetőséget.

5 Modultesztek

A Gateway működését kisebb funkcionális egységekre bontottam és egységenként külön tesztprogramot implementáltam, amely alátámasztja a helyességét. Ez a fejlesztés során előnyös és időtakarékos megoldás, mivel a modul együttes tesztelésénél jelentősen csökkenthető vele a felmerülő hibakeresések ideje.

5.1 A Gateway és a RabbitMQ Broker kommunikációjának tesztelése

A fejlesztés során a projekt menedzseléséhez és a build folyamat automatizálásához a széleskörűen ismert Apache Maven [36] eszközt használtam. A Maven egyik legnagyobb előnye az, hogy a segítségével egyszerűen és időtakarékosan tölthetőek le a különböző függőségek. A tesztprogram egyetlen különleges függősége a RabbitMQ által kiadott AMQP klienskönyvtár, amely használatával fel tudjuk venni a kapcsolatot a korábban telepített RabbitMQ Brokerrel. A RabbitMQ Broker telepítésének és konfigurálásának lépései a 4.2. alfejezetben kerültek tárgyalásra.

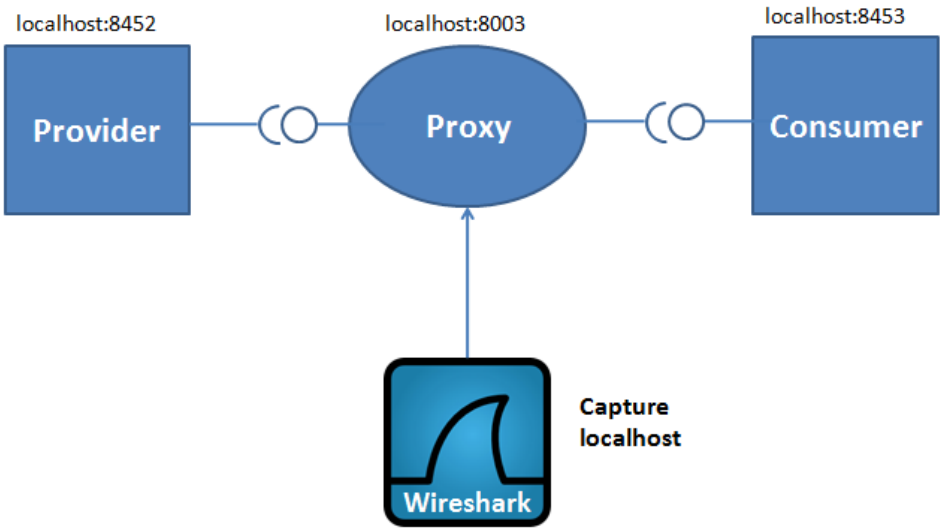
A tesztalkalmazást a RabbitMQ hivatalos dokumentációja alapján készítettem el minimális módosítással. A cél az volt csupán, hogy teszteljem a mantis3.tmit.bme.hu szerverre telepített Broker elérhetőségét és azt, hogy sikerül-e létrehozni a tanúsítvány alapú titkos kapcsolatokat. A 23. ábrán megfigyelhető a kapott üzenetek egy része. A küldött üzenet mindig az aktuális idő volt milliszekundumban mérve. Látható, hogy milyen gyorsan képes üzenetet továbbítani a Broker.

```
[*] Waiting for messages. To exit press CTRL+C
0. message: 1509029826566 ms
1. message: 1509029826581 ms
2. message: 1509029826597 ms
3. message: 1509029826597 ms
4. message: 1509029826597 ms
5. message: 1509029826597 ms
6. message: 1509029826597 ms
7. message: 1509029826597 ms
8. message: 1509029826597 ms
9. message: 1509029826597 ms
10. message: 1509029826597 ms
11. message: 1509029826597 ms
12. message: 1509029826597 ms
13. message: 1509029826597 ms
14. message: 1509029826597 ms
15. message: 1509029826597 ms
16. message: 1509029826597 ms
17. message: 1509029826597 ms
18. message: 1509029826597 ms
19. message: 1509029826597 ms
20. message: 1509029826597 ms
```

23. ábra: A RabbitMQ Brokeren keresztül kapott üzenetek egy része

5.2 A nem SSL alapú socketek működésének tesztelése

Az implementált tesztprogram egy proxyként működő egyszerű kis alkalmazás. Kezdetben nyit egy serverSocketPortot, majd várja a csatlakozni kívánó Consumert, aki számára a kapcsolat létrejöttekor létrehoz egy socketet és azon keresztül kapja meg a kérését. A következő lépésben a Provider számára is létrehoz egy socketet és ezen keresztül a Consumer kérését változtatás nélkül továbbítja a Provider részére. A Provider válasza szintén ezen a socketen keresztül jut el hozzánk és ezt továbbítjuk a Consumer részére. A Consumer, a Provider és a Proxy is helyi hálózaton, azaz localhoston kapcsolódik egymáshoz. A kommunikáció során továbbított csomagokat a Wireshark [37] program segítségével figyeltem meg. A 24. ábra a kialakított tesztelrendezést mutatja.



24. ábra: A socketek működésének teszteléséhez összeállított tesztkörnyezet

Az előzőleg tárgyalt program működése során küldött csomagok a 2. függeléken figyelhetőek meg. Szeretném kiemelni azt, hogy két HTTP GET kérés ment végbe, mivel a Consumer kérése először a proxyhoz került, majd a proxy továbbította azt a Provider felé nyitott socket számára. Ennek megfelelően két HTTP OK választ láthatunk, mivel ez is két lépésben jutott el a Consumerhez. Ezeken kívül a TCP kapcsolatok felépítéséhez és lezárásához szükséges üzenetek láthatóak még.

```
Time before sending:Tue Oct 24 17:11:27 CEST 2017
The indoor temperature is 21 degrees celsius.
Time after sending:Tue Oct 24 17:11:28 CEST 2017
Elapsed millisecs:493
```

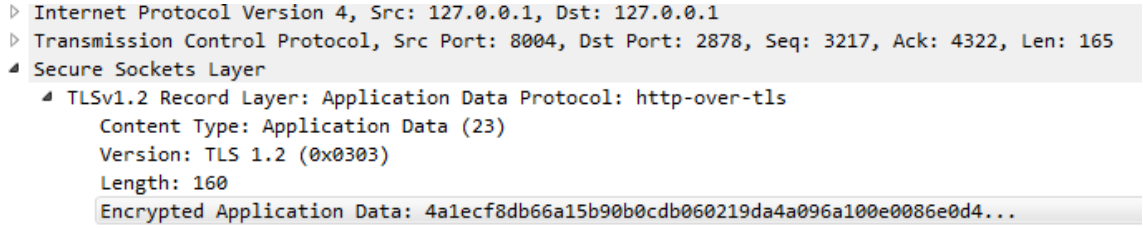
25. ábra: A kérés teljesítése alatt eltelt idő mérése

A kérés elküldése és a válasz megérkezése között 493 milliszekundum telt el, ezt mutatja a 25. ábra.

5.3 Az SSL alapú socketek működésének tesztelése

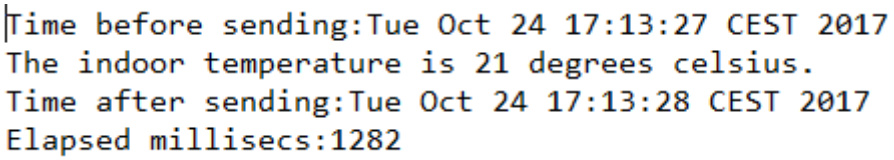
Az SSL alapú socketek működésének teszteléséhez szintén az előző alfejezetben található 24. ábrán látható tesztkörnyezetet használtam. A tesztprogram működése is teljesen azonos, a különbség abból adódik, hogy ebben az alkalmazásban minden előforduló socket SSL alapú. Az 3. függeléken látható a tesztelés során Wiresharkkal megfigyelt csomagok egy részlete. A kommunikáció titkosítására a szállítási réteg felett a TLSv1.2 protokollt használtam, ami a TLS (Transport Layer Security) legfrissebb

szabványos változata. Az 3. függelék bemutatja a tanúsítványok hitelesítéséhez szükséges üzeneteket is. A küldött adatok titkosítva továbbítódtak, ez látható a 26. ábrán, ahol az egyik csomag részletei kerülnek részletezésre, ahogyan éppen az SSL proxy küldi a Provider felé nyitott SSL alapú socket számára.



26. ábra: A titkosított adat, melyet az SSL proxy küld a Provider irányába (Wireshark)

A kérés elküldése és a válasz megérkezése között 1282 milliszekundum telt el, ezt mutatja a 27. ábra. Az 5.2. alfejezetben részletezett teszthez képest kétszer annyi időre volt szükség ebben az esetben a kérés teljesítéséhez, mivel a tanúsítványokkal és a titkosítással kapcsolatos feladatok is hozzáadódtak.



27. ábra: A kérés teljesítése alatt eltelt idő mérése (SSL)

6 Összefoglalás

Az első fejezet egy általános áttekintést és bevezetést nyújtott a biztonság és az IoT, illetve IIoT rendszerek világába.

Az második fejezet az Arrowhead keretrendszer felépítését és működését tárgyalta. Bemutattam a lokális felhők koncepcióját és a bennük elhelyezkedő rendszerek hitelesítésének folyamatát. Megvitattam a biztonságtechnikai kérdéseket.

A harmadik fejezet a lokális IoT felhők együttműködésével foglalkozott. Kifejtettem az együttműködés során kulcsfontosságú Orchestrator és Gatekeeper modul jelentőségét és részleteztem az általuk nyújtott szolgáltatások működését. Megterveztem az Arrowhead csapatának közreműködésével az új központi rendszer, a **Gateway** folyamatait és működését, biztosítva a nyílt információs hálózaton áthaladó adatok biztonságos továbbításának alapfeltételeit. Illusztráltam egy példán keresztül az Arrowhead keretrendszer már meglévő központi rendszer és az új Gateway modul együttműködését.

A negyedik fejezetben fókuszában az általam implementált Gateway megvalósítási részletei álltak, kiemelve az AMQP technológia és a socketek hasznosítási módját. Ezen kívül a Gatekeeper modul módosításai is bemutatásra kerültek. A fejezet alátámasztja az előző fejezetekben felvezetett biztonságos adatkommunikáció megvalósíthatóságát és megmutatja egy konkrét példáját.

Az ötödik fejezet a tervek és implementáció részfeladatokra bontott verifikációját tartalmazta.

Irodalomjegyzék

[1] Turzó Ádám Pál, *A ma ismert világot totálisan elsöpri a negyedik ipari forradalom*, 2016
(Letöltés időpontja: 2017 szept.)
http://www.portfolio.hu/vallalatok/it/a-ma-ismert-vilagot-totalisan-elsopri-a-negyedik-ipari-forradalom.237125-3.html

[2] Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016
(Letöltés időpontja: 2017 szept.)
http://www.gartner.com/newsroom/id/3598917

[3] Inductive Automation*: The Industrial Internet of Things (IIoT)*
(Letöltés időpontja: 2017 szept.)
https://inductiveautomation.com/what-is-iiot

[4] Az Arrowhead projekt weboldala
(Letöltés időpontja: 2017 szept.)
http://www.arrowhead.eu/

[5] J. Delsing, *The Arrowhead Framework vision and objective*
(Letöltés időpontja: 2017 szept.)
https://forge.soa4d.org/plugins/mediawiki/wiki/arrowhead-f/index.php/Why_%26_How

[6] J. Delsing, P.Varga, L. Ferreira, M. Albano, P. Punal Pereira, J. Eliasson, O. Carlsson, H. Derhamy: *IoT Automation: Arrowhead Framework, Chapter 3 – The Arrowhead Framework architecture,* 1 edition, ISBN 1498756751, CRC Press, 2017

[7] Middleware blog, *SOA alapok,* 2011
(Letöltés időpontja: 2017 szept.)
http://middleware.blog.hu/2011/07/25/soa_alapok

[8] A. Froehlich, *The Basics of QoS*
(Letöltés időpontja: 2017 szept.)
http://www.networkcomputing.com/networking/basics-qos/402199215

[9] Hegedüs Cs., Plósz S., *AAA in Arrowhead,* 2017, p. 8

[10] J. Delsing, P. Varga: *IoT Automation: Arrowhead Framework, Chapter 2 – Local automation clouds*, 1 edition, ISBN 1498756751, CRC Press, 2017

[11] P. Varga, F. Blomstedt, L. L. Ferreira, J. Eliasson, M. Johansson, J. Delsing: *Making System of Systems Interoperable – The Core Components of The Arrowhead Technology Framework*
Journal of Network and Computer Applications, 2016

[12] R. Donato, *PKI - Chain of Trust*, (Letöltés időpontja: 2017 okt.)
https://www.fir3net.com/Security/Concepts-and-Terminology/pki-chain-of-trust.html

[13] SSH Communications Security, *SSH (Secure Shell)*
(Letöltés időpontja: 2017 okt.)
https://www.ssh.com/

[14] Csabai Cs., *PKI – avagy mit takar a publikus kulcsú infrastruktúra?*, 2002
(Letöltés időpontja: 2017 okt.)
https://www.hwsw.hu/hirek/40563/pki----avagy-mit-takar-a-publikus-kulcsu-infrastruktura-.html

[15] IBM, *Certificate Authorities and trust hierarchies*
(Letöltés időpontja: 2017 okt.)
http://publib.boulder.ibm.com/tividd/td/ITAME/SC32-1363-00/en_US/HTML/ss7aumst14.htm

[16] Sharpened Productions, FileInfo: *.JKS File Extension*
(Letöltés időpontja: 2017 okt.)
https://fileinfo.com/extension/jks

[17] W. Grant, K. Kramer: *KeyStore Explorer*
(Letöltés időpontja: 2017 okt.)
http://keystore-explorer.org/index.html

[18] M. Rouse, S. Shea: *X.509 certificate,* TechTarget, SearchSecurity
(Letöltés időpontja: 2017 okt.)
http://searchsecurity.techtarget.com/definition/X509-certificate

[19] L. Grove: *DER vs. CRT vs. CER cs PEM Certificates and How To Convert Them*, SSL Installation Support, 2011
(Letöltés időpontja: 2017 okt.)
https://support.ssl.com/Knowledgebase/Article/View/19/0/der-vs-crt-vs-cer-vs-pem-certificates-and-how-to-convert-them

[20] A. Woland, *Simply put: How does certificate based authentication work?*
(Letöltés időpontja: 2017 okt.)
https://www.networkworld.com/article/2226498/infrastructure-management/simply-put-how-does-certificate-based-authentication-work.html

[21] T. King, G. Reese, Randy J. Yarger, *A MySQL kezelése és használata,*
ISBN9630943867, Kossuth Kiadó Zrt, 2002

[22] Cs. Hegedus, *Orchestrator System Description*
The Arrowhead Framework Documentation, 2017, p. 6

[23] Cs. Hegedus, *Orchestrator System Description*
The Arrowhead Framework Documentation, 2017

[24] Végh D. A., *IoT rendszerek biztonságtechnikai megoldásai, pp 51-52*
Szakdolgozat, Budapesti Műszaki és Gazdaságtudományi Egyetem,
Távközlési és Médiainformatikai Tanszék

[25] P. Varga, Cs. Hegedus, *Inter-Cloud Communication through Gatekeepers to Support IoT Service Interaction in the Arrowhead Framework*

[26] J. Delsing, J. Eliasson, M. Albano, P. Varga, L. Ferreira, H. Derhamy, Cs. Hegedűs, P. Punal Pereira, O. Carlsson: *IoT Automation: Arrowhead Framework, Chapter 4 – ArrowheadFramework core systems and services*, 1 edition, ISBN 1498756751, CRC Press, 2017

[27] P. Varga, Cs. Hegedus, *Service Interaction through Gateways for Inter-Cloud Collaboration within the Arrowhead*
Conference paper, December 2015, pp 5-6

[28] Pivotal: *RabbitMQ*, (Letöltés időpontja: 2017 okt.)
https://www.rabbitmq.com/

[29] R.Mason*, AMQPand the future of web messaging,* (Letöltés időpontja: 2017 okt.)
https://blogs.mulesoft.com/dev/news-dev/amqp-and-the-future-of-web-messaging/

[30] A. Papoola, *Design Patterns: PubSub Explained*, March 12, 2013
(Letöltés időpontja: 2017 okt.)
https://abdulapopoola.com/2013/03/12/design-patterns-pub-sub-explained/

[31] Oracle Java Documentation, *Lesson: All About Sockets*
(Letöltés időpontja: 2017 okt.)
https://docs.oracle.com/javase/tutorial/networking/sockets/index.html

[32] M. Shaikh, *Simple Chat Application using Java Socket programming*
(Letöltés időpontja: 2017 okt.)
http://www.codingdevil.com/2015/04/simple-chat-application-using-java-socket-programming.html

[33] T. Frederich*: What is Rest?* (Letöltés időpontja: 2017 okt.)
http://www.restapitutorial.com/lessons/whatisrest.html

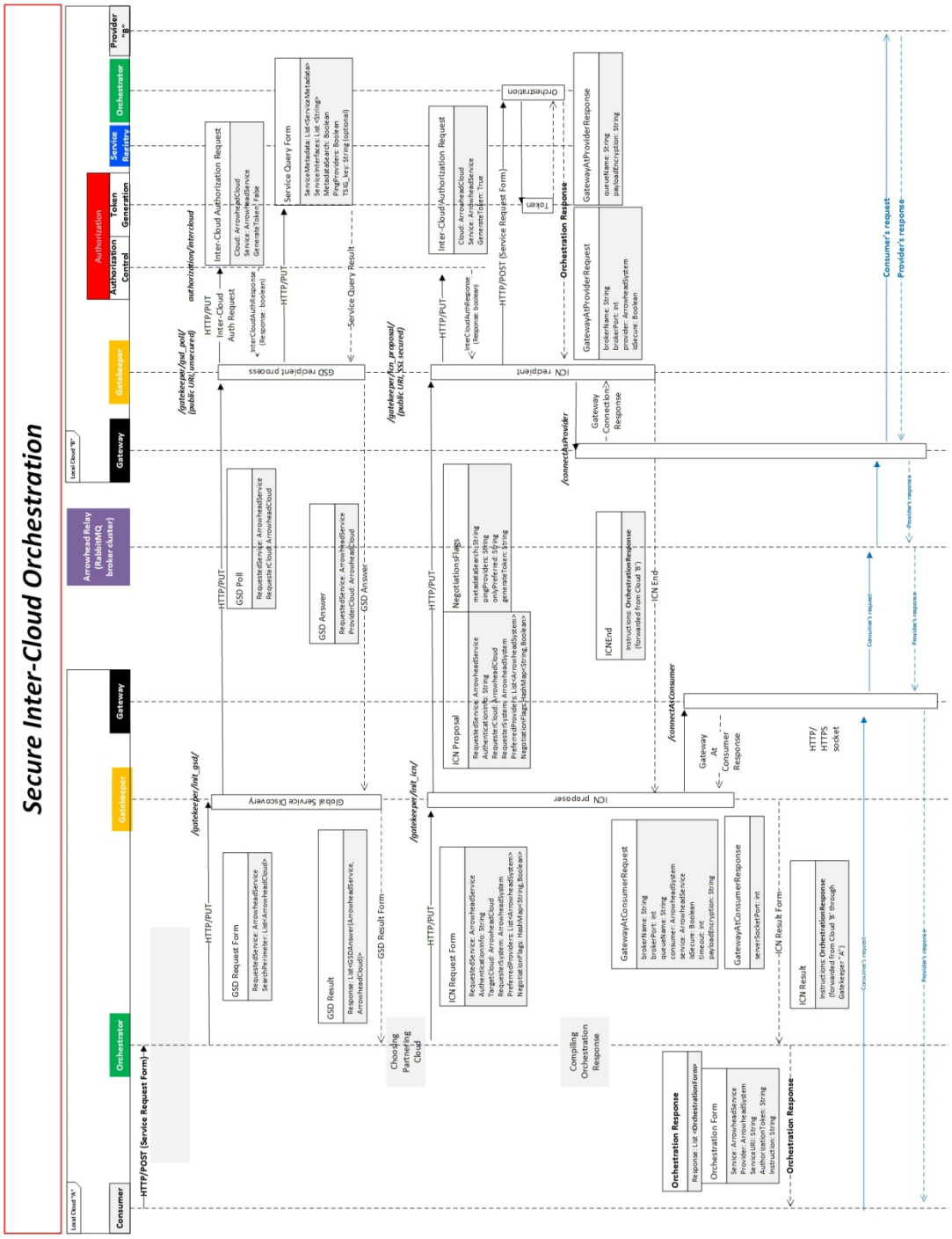
[34] Cs. Hegedus, *Making things collaborate – Interworking of Collaborative System-of-Systems in the Arrowhead Framework*
TDK-dolgozat, 2015. ősz

[35] Pivotal: *RabbitMQ Clustering Guide*, (Letöltés időpontja: 2017 okt.)
https://www.rabbitmq.com/clustering.html

[36] The Apache Software Foundation, *Apache Maven Project*
(Letöltés időpontja: 2017 okt.)
https://maven.apache.org/

[37] Wireshark Foundation, *Wireshark,* (Letöltés időpontja: 2017 okt.)
https://www.wireshark.org/

Függelék



1. függelék: Felhők közötti kommunikáció üzenetváltásait ábrázoló szekvencia diagram

2. függelék: A socketek tesztelése során megfigyelt csomagok Wiresharkban

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	0.0.0.0	255.255.255.255	DHCP	660	DHCP Discover - Transaction ID 0x7bf1ca9d
2	0.000061	0.0.0.0	255.255.255.255	DHCP	660	DHCP Discover - Transaction ID 0x7bf1ca9d
3	1.242407	127.0.0.1	127.0.0.1	TCP	108	2869 → 8003 [SYN] Seq=0 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM=1
4	1.242467	127.0.0.1	127.0.0.1	TCP	108	8003 → 2869 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM=1
5	1.242493	127.0.0.1	127.0.0.1	TCP	84	2869 → 8003 [ACK] Seq=1 Ack=1 Win=8192 Len=0
6	1.243790	127.0.0.1	127.0.0.1	HTTP	534	GET /temperature HTTP/1.1
7	1.243813	127.0.0.1	127.0.0.1	TCP	84	8003 → 2869 [ACK] Seq=1 Ack=226 Win=7936 Len=0
8	1.251009	127.0.0.1	127.0.0.1	TCP	108	2870 → 8453 [SYN] Seq=0 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM=1
9	1.251091	127.0.0.1	127.0.0.1	TCP	108	8453 → 2870 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM=1
10	1.251122	127.0.0.1	127.0.0.1	TCP	84	2870 → 8453 [ACK] Seq=1 Ack=1 Win=8192 Len=0
11	1.251402	127.0.0.1	127.0.0.1	TCP	86	2822 → 2823 [PSH, ACK] Seq=1 Ack=1 Win=31 Len=1
12	1.251425	127.0.0.1	127.0.0.1	HTTP	534	GET /temperature HTTP/1.1
13	1.251430	127.0.0.1	127.0.0.1	TCP	84	2823 → 2822 [ACK] Seq=1 Ack=2 Win=31 Len=0
14	1.251460	127.0.0.1	127.0.0.1	TCP	84	8453 → 2870 [ACK] Seq=1 Ack=226 Win=7936 Len=0
15	1.254328	127.0.0.1	127.0.0.1	HTTP	290	HTTP/1.1 200 OK (text/plain)
16	1.254357	127.0.0.1	127.0.0.1	TCP	84	2870 → 8453 [ACK] Seq=226 Ack=104 Win=7936 Len=0
17	1.254381	127.0.0.1	127.0.0.1	HTTP	290	HTTP/1.1 200 OK (text/plain)
18	1.254401	127.0.0.1	127.0.0.1	TCP	84	2869 → 8003 [ACK] Seq=226 Ack=104 Win=7936 Len=0
19	1.254586	127.0.0.1	127.0.0.1	TCP	84	8003 → 2869 [FIN, ACK] Seq=104 Ack=226 Win=7936 Len=0
20	1.254603	127.0.0.1	127.0.0.1	TCP	84	2869 → 8003 [ACK] Seq=226 Ack=105 Win=7936 Len=0
21	1.254656	127.0.0.1	127.0.0.1	TCP	84	2870 → 8453 [FIN, ACK] Seq=226 Ack=104 Win=7936 Len=0
22	1.254668	127.0.0.1	127.0.0.1	TCP	84	8453 → 2870 [ACK] Seq=104 Ack=227 Win=7936 Len=0
23	1.254739	127.0.0.1	127.0.0.1	TCP	86	2822 → 2823 [PSH, ACK] Seq=2 Ack=1 Win=31 Len=1
24	1.254758	127.0.0.1	127.0.0.1	TCP	84	2823 → 2822 [ACK] Seq=1 Ack=3 Win=31 Len=0
25	1.255024	127.0.0.1	127.0.0.1	TCP	84	8453 → 2870 [FIN, ACK] Seq=104 Ack=227 Win=7936 Len=0
26	1.255053	127.0.0.1	127.0.0.1	TCP	84	2870 → 8453 [ACK] Seq=227 Ack=105 Win=7936 Len=0
27	1.276690	127.0.0.1	127.0.0.1	TCP	84	2869 → 8003 [RST, ACK] Seq=226 Ack=105 Win=0 Len=0

1	0.000000	127.0.0.1	127.0.0.1	TCP	108 2878 → 8004 [SYN] Seq=0 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM=1
2	0.000085	127.0.0.1	127.0.0.1	TCP	108 8004 → 2878 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM=1
3	0.000118	127.0.0.1	127.0.0.1	TCP	84 2878 → 8004 [ACK] Seq=1 Ack=1 Win=8192 Len=0
4	0.032562	127.0.0.1	127.0.0.1	TLSv1.2	508 Client Hello
5	0.032596	127.0.0.1	127.0.0.1	TCP	84 8004 → 2878 [ACK] Seq=1 Ack=213 Win=7936 Len=0
6	0.118185	127.0.0.1	127.0.0.1	TCP	3004 8004 → 2878 [ACK] Seq=1 Ack=213 Win=7936 Len=1460 [TCP segment of a reassembled PDU]
7	0.118193	127.0.0.1	127.0.0.1	TCP	3004 8004 → 2878 [ACK] Seq=1461 Ack=213 Win=7936 Len=1460 [TCP segment of a reassembled PDU]
8	0.118222	127.0.0.1	127.0.0.1	TCP	84 2878 → 8004 [ACK] Seq=213 Ack=2921 Win=8192 Len=0
9	0.118238	127.0.0.1	127.0.0.1	TLSv1.2	494 Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done
10	0.118251	127.0.0.1	127.0.0.1	TCP	84 2878 → 8004 [ACK] Seq=213 Ack=3126 Win=7936 Len=0
11	0.167918	127.0.0.1	127.0.0.1	TCP	3004 2878 → 8004 [ACK] Seq=213 Ack=3126 Win=7936 Len=1460 [TCP segment of a reassembled PDU]
12	0.167926	127.0.0.1	127.0.0.1	TLSv1.2	2572 Certificate, Client Key Exchange
13	0.167980	127.0.0.1	127.0.0.1	TCP	84 8004 → 2878 [ACK] Seq=3126 Ack=2917 Win=8192 Len=0
14	0.224890	127.0.0.1	127.0.0.1	TLSv1.2	622 Certificate Verify
15	0.224916	127.0.0.1	127.0.0.1	TCP	84 8004 → 2878 [ACK] Seq=3126 Ack=3186 Win=7680 Len=0
16	0.224965	127.0.0.1	127.0.0.1	TLSv1.2	96 Change Cipher Spec
17	0.224973	127.0.0.1	127.0.0.1	TCP	84 8004 → 2878 [ACK] Seq=3126 Ack=3192 Win=7680 Len=0
18	0.226961	127.0.0.1	127.0.0.1	TLSv1.2	254 Encrypted Handshake Message
19	0.226985	127.0.0.1	127.0.0.1	TCP	84 8004 → 2878 [ACK] Seq=3126 Ack=3277 Win=7680 Len=0
20	0.232034	127.0.0.1	127.0.0.1	TLSv1.2	96 Change Cipher Spec
21	0.232056	127.0.0.1	127.0.0.1	TCP	84 2878 → 8004 [ACK] Seq=3277 Ack=3132 Win=7936 Len=0
22	0.232497	127.0.0.1	127.0.0.1	TLSv1.2	254 Encrypted Handshake Message
23	0.232513	127.0.0.1	127.0.0.1	TCP	84 2878 → 8004 [ACK] Seq=3277 Ack=3217 Win=7680 Len=0
24	0.235481	127.0.0.1	127.0.0.1	TLSv1.2	2174 Application Data
25	0.235500	127.0.0.1	127.0.0.1	TCP	84 8004 → 2878 [ACK] Seq=3217 Ack=4322 Win=6656 Len=0
26	0.248997	127.0.0.1	127.0.0.1	TCP	108 2879 → 8452 [SYN] Seq=0 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM=1

3. függelék: Az SSL-t használó socketek tesztelése során megfigyelt csomagok részlete Wiresharkban