Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

Böősy Pál

PROTOKOLLKIEGÉSZÍTÉSI JAVASLAT SRv6 ALAPÚ TECHNOLÓGIÁK SKÁLÁZÓDÁSÁNAK JAVÍTÁSÁRA

KONZULENSEK

Dr. Bokor László; Leiter Ákos

BUDAPEST, 2023

Tartalomjegyzék

Összefoglaló

A felhőszolgáltatások és a rajtuk megvalósítható mikroszolgáltatás-architektúra új kihívások elé állítják a hálózatok tervezőit. A felhasználók által a publikus felhőkben futtatott virtuális gépek és konténerek hálózati összeköttetése komoly tervezést igényel a mérnökök részéről. Az ilyen rendszerek komplexitásából adódóan a szolgáltatásokat kiszolgáló hálózatnak is összetettnek kell lennie, mely az adatközponton belüli és kívüli útválasztás esetében is egyre több csomópontot tartalmazó útvonalat jelent. Ezen útvonalak esetében hasznos lenne különböző forgalomirányítási stratégia (Traffic Engineering) módszerek alkalmazása a hálózati réteg szintjén. Azonban az ilyen lehetőségeket csak több, különböző hálózati protokoll összehangolásával lehetne megvalósítani. A többféle protokoll alkalmazása magasabb üzemeltetési költségeket és a hálózat nehézkesebb karbantartását jelentené.

Ezen problémák kiküszöbölésére egy jó opció lehet az IPv6 protokoll szegmens alapú útválasztást megvalósító kiegészítése (Segment Routing over IPv6, SRv6) [1]. Az SRv6-ban (és más Source Routing-ot megvalósító protokollokban) a hálózat útválasztóitól elvesszük a döntési lehetőséget, és a hálózat belépési pontján ellátjuk a csomagokat az általuk bejárandó csomópontok listájával. Ezáltal explicit módon tudjuk a csomagot olyan útvonalon továbbítani, melyről megbízható információkkal rendelkezünk például a sávszélességi, késleltetési és egyéb hálózati paraméterekről. Mivel az SRv6 az IPv6 fejléc részét képezi szabványos kiegészítő fejlécként (IPv6 Extension Header), ezért az általa nyújtott lehetőségek miatt mind transzport-, mind maghálózatokban is jól használható, pl. SR-MPLS [2] helyett. Ennek köszönhetően akár a teljes hálózati folyamhoz (adatközpont hálózattól a transzport hálózatig beleértve a mobil maghálózatot is) elegendő lenne egyetlen protokollt használni, ezáltal elősegítve a protokoll heterogenitás csökkentését.

Azonban az SRv6 megvalósításából adódóan az egyes hálózati pontok (szegmensek) IPv6-os címekként vannak reprezentálva, mely szegmensenként is jelentős növekményt jelent egy csomagban. Nem nehéz elképzelni, hogy nagyobb számú szegmens esetén (10-es nagyságrend) a csomagok mérete jelentősen megnőhet. Ez pedig olyan problémákat idézhet elő, mint a lassabb feldolgozási idő, vagy a túl nagy csomagok eldobása.

Ezen probléma megoldására születtek már megoldások (pl. SRv6 Micro SID [3]), azonban bizonyos esetekben, ahol szükség lehet a nagyobb címzési tartomány megtartására, ezek nem biztos, hogy hatékonyak lehetnek. Ennek a megoldására egy újszerű megközelítést mutatok be, az SRv6 Policy Translation (a továbbiakban SRv6-PT) kiegészítést, mely a későbbiekben akár az SRv6 protokoll kiegészítése is lehet. Az SRv6-PT működésének alapját a hálózati folyamonként dinamikusan kinevezett speciális csomópontok jelentik. Ezek oly módon cserélik a Routing Header szegmenseit, hogy mindig az aktuálisan következő útvonal-részlet kerüljön a csomagba, és egy meghatározott számú szegmensnél több ne legyen egyszerre. Tekintve, hogy a Linux kernel részeként elérhető, nyílt forráskódú SRv6 implementáció még nem eléggé kiforrott, így ezt kiegészítendő, az általam kidolgozott új eljárást egy Linux kernelmodulként implementálom. A saját SRv6-PT megoldást és implementációt teljesítménytesztnek vetem alá a meglévő megvalósításokkal szemben, és a mérési eredmények függvényében körvonalazom a használhatóságot befolyásoló hálózati és egyéb paraméterek jellemzőit, valamint a feltárt előnyök és potenciális korlátok mentén javaslatot teszek a gyakorlati alkalmazási lehetőségeire.

Abstract

Implementing microservices architecture in a cloud-native environment could pose novel challenges for network designers. The network management of the virtual machines and containers that users run in public cloud environments requires a lot of planning and effort from engineers. Given the complexity of such systems, the network must also be complex, which means routing a growing number of nodes inside and outside the data center. Applying different traffic engineering techniques at the network layer level would be useful for these routes. However, such options could only be implemented by coordinating several different network protocols. Using multiple protocols would mean higher operational costs and more difficult network maintenance.

To overcome these problems, a better option could be the Segment Routing over IPv6 (SRv6) protocol [1]. In SRv6 (and other protocols implementing the Source Routing paradigm), the routing decision is taken away from the network routers, and packets are provided with a list of nodes they should traverse from the ingress of the SRv6 network domain. This allows us to explicitly route the packet along a route with reliable information about bandwidth, latency, and other network parameters. Since SRv6 is part of IPv6 as a standard IPv6 Extension Header, its capabilities make it a good choice for both transport and core networks, e.g., instead of SR-MPLS [2]. This would make it sufficient to use a single protocol for the entire network flow (from the data center network to the transport network, including the case of mobile core networks), thus helping to reduce protocol heterogeneity.

However, due to the implementation of SRv6, each node in the network (segment) is represented as an IPv6 address, which also means a significant increment per segment in the packet size. It is not difficult to imagine that the size of packets can increase significantly with a larger number of segments. This can cause problems such as increased processing delay or dropping overly large packets.

There are already existing implementations for the solution of this problem (e.g., SRv6 Micro SID [3]), but in some cases where it may be necessary to maintain a larger addressing range, they may not be effective. To solve this problem, I present a novel technique called SRv6-PT that could complement the SRv6 protocol in the future. SRv6-PT is based on special nodes that are dynamically assigned per network flow. They swap

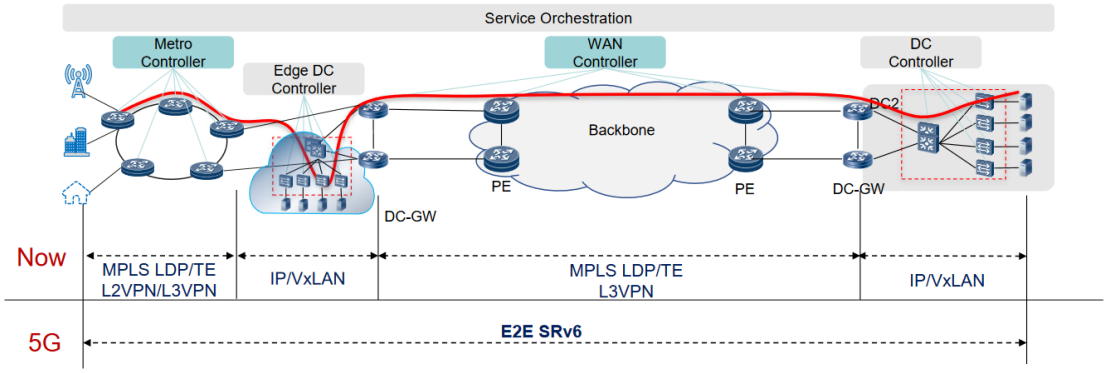
the segments of the Routing Header so that the next part of the path to be traversed is always included in the packet, and no more than a certain number of segments are included at the same time. Given that the open-source SRv6 implementation available as part of the Linux kernel is not yet mature enough, I implemented my new method as a Linux kernel module. I subject my SRv6-PT solution to extensive performance tests against existing implementations and, depending on the measurement results, outline the network and other parameters that affect its usability and suggest practical applications based on the advantages and potential limitations identified.

1 Bevezetés

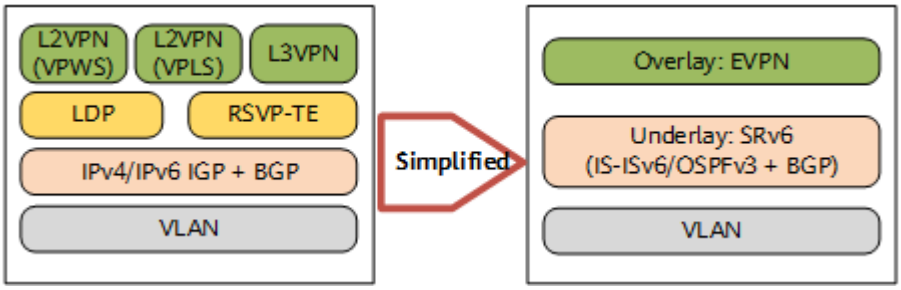
Az olyan innovatív területek, mint az ötödik generációs mobilhálózatok és felhő alapú szolgáltatások, komoly változtatások eszközölésére sarkallhatják az IP alapú hálózatokat. Az előbbi esetében főként a különböző felhasználási területeken átívelő, szigorú szolgáltatási paraméterek biztosítása jelenti a kihívást. Ez egyúttal a hálózat teljesítményének és rugalmasságának, jelenlegi állapotához képest exponenciális szintű növekedését kívánná meg. Megfelelő példa erre az 'Ultra Reliable Low Latency Communication' (URLLC) és az 'enhanced Mobile Broadband' (eMBB) hálózati szeletek által meghatározott követelmények biztosítása, illetve önmagában a hálózati szeletelés (network slicing) biztosítása is. Emellett a felhőszolgáltatások fejlődésével a szolgáltatásfeldolgozás lokációi is rugalmasabbá váltak. Egyes felhőszolgáltatások (például a távközlési felhő) tovább törik a fizikai és virtuális hálózati eszközök közötti határt, ezáltal integrálva a hálózati szolgáltatásokat és a transzport hálózatokat. [4] Látható tehát, hogy az olyan tényezők, mint a szolgáltatásminőség fenntartása (QoS), a hálózati forgalomirányítás (Traffic Engineering) és a hálózatok programozhatóságának (Network Programming) biztosítása kritikus feladata a jövő hálózati mérnökeinek.

A fejlődés egyik elősegítője az SRv6 alkalmazása lehet, mely egyszerre nyújt lehetőséget a hálózati forgalom jól definiált terelésére és a hálózat programozhatóságának megvalósítására is. Az SRv6 egy meglehetősen új koncepció, melynek igazi potenciáljai jelenleg kezdenek átértékelődni. A technológia alapját az adja, hogy a csomagokba kódolva, explicit módon tudjuk meghatározni egy csomag útját egy hálózati tartományon belül. Mindezek mellett a természetéből adódóan nem csak bejárandó csomópontokként kezelhetjük az egyes szegmenseket, hanem különböző hálózati funkciókat is azonosíthatunk velük. Az SRv6 egyik fő szerepe a modern hálózatok fejlődésében az lenne, hogy kiváltson egyéb, körülményesebben használható hálózati protokollokat és forgalomirányítási eljárásokat. Így egy egységes, közös alap biztosításával (mely jelen esetben az IPv6 alapú adatsíkot jelenti) változatos szcenáriókban is alkalmazásra kerülhet, a hálózatok protokoll-heterogenitásának csökkentésével együtt. Többek között ezáltal az is elképzelhetővé válhat, hogy adatközpontokban [5], mobile-coreban [6] és a transzport hálózatban [7] is SRv6 alapon működjön a csomagtovábbítás, és csak erre az egy közös protokollra lenne szükség. Mobilhálózatok esetében a GTP-U kiváltására [8], [9] különösen is ígéretesnek tűnik, így bevezetve ugyanazt a protokollt mind az IETF,

mind a 3GPP által standardizált hálózati területeken. Az SRv6 végponttól végpontig történő és a különböző hálózatokon átívelő használatát az 1.1 ábra kiválóan szemlélteti. Az 1.2 ábra pedig az SRv6-al elérhető protokoll heterogenitást szemlélteti a hálózati protokoll-stack csökkentésével.



1.1 ábra: Meglévő hálózati protokollok kiváltásának lehetősége SRv6-al [7]



1.2 ábra: A hálózati protokoll-stack egyszerűsítése az SRv6-al [10]

Az SRv6-ot használó csomagok esetében a bejárandó útvonal egy rendezett listaként kerül kódolásra a csomagban, melynek elemei a szegmens azonosítók (SID-ek). Ezek reprezentálása 128 biten történik, azaz szabványos IPv6 címekként foghatóak fel. A széles címzési tartomány azonban míg az IPv6 egyik nagy előnye, addig az SRv6-os szegmensek ilyen módon történő reprezentálása hátrányt is tud jelenteni. Nem nehéz elképzelni, hogy egy komplexebb hálózati útvonal/feldolgozás esetén a bejárandó csomópontok/végrehajtandó műveletek számának növekedésével a csomagokba kódolt szegmensek száma is jelentősen megnövekedik, mivel a szegmensenkénti növekmény is jelentősnek mondható. Ez a megnövekedett csomagméret pedig az SRv6 eredeti célkitűzésével messzemenően ellentétes állapotot idézhet elő a hálózatban: az ilyen csomagok feldolgozása jelentősen lassulhat, illetve fennáll a veszélye, hogy egyes hálózati csomópontok egyszerűen eldobják a túlzott méretük miatt ezeket a csomagokat.

A probléma tehát adott, melyre a szegmens lista hozzáadásával járó növekmény kordában tartása lenne a megoldás. Ennek egyik lehetséges megoldása az SRv6 microSID-ek [3] alkalmazása. Ezen eljárás szerint a 128 bites SID-ek feloszthatóak több kisebb részre, és egyetlen instrukció helyett több is elhelyezhetővé válna egy SRv6 SID-en belül. Ezzel szemben egy másik lehetőség a szegmens lista méretének karbantartására, ha nem a kódolt eljárások méretét akarjuk csökkenteni, hanem a lista elemeinek számát akarjuk egy meghatározott határon belül tartani, a csomag teljes útvonalán át. Az általam javasolt SRv6-PT eljárás ebből a szemszögből közelítené meg a problémát.

1.1 Motiváció

A motivációt egy új eljárás tervezésére és a dolgozat megírására az adta, hogy megvizsgáljam, az SRv6 csomagok méretének növekedésével keletkező problémát az irodalomban fellelhető sémáktól eltérő úton is lehet-e hatékonyan kezelni. A dolgozat készítésénél a microSID implementáció tesztelésére nem lesz lehetőségem, többek között a nyílt forráskódú microSID implementációk hiánya miatt, így a saját metódussal történő teljesítménybeli összehasonlítást nem fogom tudni elvégezni. Azonban, ha a tervezett eljárást sikeresen implementálom, és a hagyományos SRv6 alapú feldolgozással megközelítőleg azonos teljesítményűre sikerül verifikálni, azzal egy újabb lehetőséget lehetne nyújtani a szegmens lista növekedéssel járó állapottér-robbanás megoldására.

2 A Segment Routing technológia ismertetése

2.1 Segment Routing architektúra

A Segment Routing-ot (SR) mint fogalmat először az RFC 8402-ben [11] szabványosították, melyben a szükséges fogalmak definiálásán felül javaslatokat is tettek meglévő hálózati protokollokban történő imlpementálhatóságukra. Ilyenek például az SR-MPLS és az SRv6 is, melyek közül utóbbira a következő alfejezetben részleteiben is kitérek. A szabvány teljes egészének részletezése a dolgozat szempontjából irreleváns, azonban szükséges a fontosabb koncepciók tisztázása.

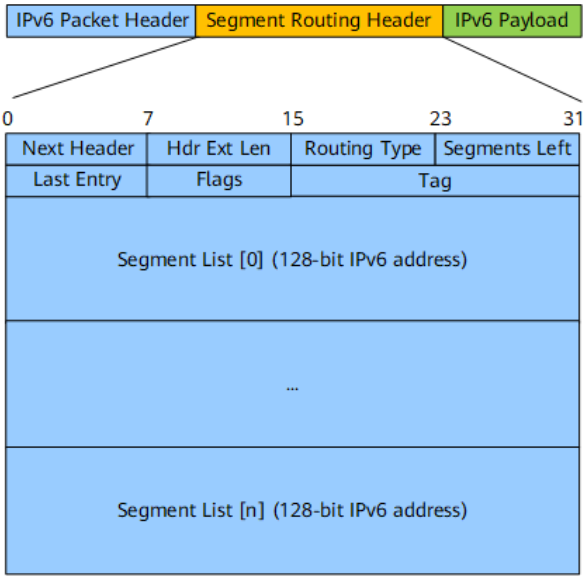
Ez a fajta útválasztás a forrás alapú útválasztási paradigmát (source routing) követi. Definíció szerint a csomagokat instrukciók egy rendezett listájával látjuk el, és a protokollt támogató csomópontok ezen rendezett lista elemein keresztül viszik végig a csomagot. Ezeket az instrukciókat, mint ahogy a technológia neve is sejteti, *szegmenseknek* (segment) nevezik. „Az SR olyan továbbítási mechanizmust biztosít, amely lehetővé teszi, hogy a hálózati folyamot egy jól definiált topológiai útvonalra korlátozza, miközben a folyamonkénti állapotot csak az SR-tartományba belépő csomópont(ok)on kell fentartani.” [11]

A szegmenseket a hálózaton belül a szegmens azonosítójuk (Segment Identifier, SID) azonosítja. Fontos, hogy egy szegmens a szokásos útvonalválasztási protokollokkal ellentétben nem feltétlenül csak egy hálózati csomópontot jelenthet. Jelölheti persze egy hálózati eszköz interfészét is, ugyanakkor egyfajta szolgáltatást, instrukciót is azonosíthat. Ezt úgy kell érteni, hogy a szegmens valamilyen speciális eljárást (SRv6 hálózati programozás [1]), QoS specifikus csomagkezelést, vagy akár egy általunk definiált csomagfeldolgozási metódust is jelölhet (a dolgozatban a későbbiekben ismertetésre kerülő, saját SRv6-PT feldolgozási eljárást fel lehet akár így is tüntetni, habár az implementáció jelenlegi formájában nem használja ki ezt a lehetőséget). A szegmensek hálózaton belüli elterjesztése történhet elosztott és centralizált módon is. Előbbi különböző útválasztási protokollokat felhasználva (pl. IS-IS, OSPF), míg utóbbi egy központi, SR vezérlő segítségével tenné ezt meg.

2.2 Segment Routing over IPv6

A Segment Routing architektúra egyik lehetséges megvalósítása az IPv6 protokollt használja ki. A megvalósítást egy újfajta IPv6 kiegészítő fejléc (Extension Header), a Segment Routing Header (SRH) segítségével lehet megtenni. Az SRH tulajdonképpen az IPv6 szabványos Routing Header fejlécének az egyik lehetséges változata.

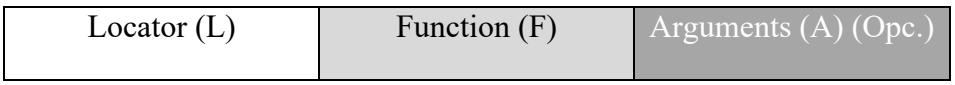
Az IPv6 egyik újítása az IPv4-el szemben a kiegészítő fejlécek. Szemben az IPv4-el, a 6-os verzióban a szabványos IPv6 fejléc mérete állandó (40 byte), és az opcionális, további információk ilyen kiegészítő fejlécek formájában kerülnek megadásra a csomagban. Ezek a csomagban az IPv6 fejléc és a felsőbb hálózati réteg fejlécei között helyezkednek el. Ezen kiegészítő fejlécek a saját, szabványos 'Next Header' értékük alapján kerülnek azonosításra. Többek között az IPv6 fejléc is rendelkezik 'Next Header' mezővel, az itt szereplő érték alapján kerül azonosításra a közvetlen felette elhelyezkedő, felsőbb rétegbeli protokoll, vagy egy ilyen kiegészítő fejléc is. A kiegészítő fejlécek számozása az IANA IP Protocol Numbers (IANA-PN) alapján történik, mely értékeket az IPv4 esetében is használják. A csomagban lévő 'Next Header' értékek sorozatának feldolgozása során az első olyan érték, amely nem kiegészítő fejlécet jelöl, az a megfelelő felső rétegbeli protokoll fejlécét jelöli (pl. transzport réteg protokollja, TCP, UDP stb.). Abban az esetben, ha nem szerepelne a csomagban más, felsőbb rétegbeli protokoll, egy speciális "No Next Header" értéket kell használni.



2.1 ábra: A Segment Routing Header felépítése és elhelyezkedése [12]

Az IPv6 Routing Header különböző típusainak első 4 byte-ja megegyezik, ezek név szerint a Next Header (a következő kiegészítő fejléc vagy réteg), Hdr Ext Len (a fejléc kiterjedése bájtban, leszámítva az első bájtját a felécnek), Routing Type (a fejléc által megvalósítandó útválasztási módszer azonosítója) és Segments Left (a még bejárandó útvonal szegmensek száma, nem feltétlenül SR szegmens) mezők. Az SRv6 esetében a Routing Type mező 4-es értéke jelöli, hogy Segment Routing Header-ről van szó. Az SRH további mezői rendre a Last Entry (az utolsó szegmens indexe, ebből is lehet a lista méretére következtetni), Flags, Tag, (utóbbi kettő opcionális információk tárolására alkalmas, alapértelmezett értékük 0) illetve végezetül a szegmensek rendezett listája. Utóbbit Segment List, vagy Segment Policy néven is szokták nevezni. A létrehozandó eljárásom neve innen származtatható (SRv6 Policy Translation). Egy SRH csomagon belüli struktúráját szemlélteti a 2.1-es ábra.

SRv6-ban a szegmensek szabványos (128 bites) IPv6-os címekként kerülnek reprezentálásra. Az RFC 8986-ban a következő szemantikus felbontást definiálják az SRv6 SID-kre: Locator (L), Function (F) és egy opcionális Arguments (A) részekre oszthatóak fel. Ezt szemlélteti a 2.2-es ábra is. A Locator mező a hálózati csomópontok helyét azonosítja más csomópontok számára, és a csomagok ezen azonosított csomóponthoz való továbbítására szolgál. A Function mező a végrehajtandó továbbítási viselkedést határozza meg, azaz, hogy az azonosított csomóponton az SRv6 fejlécet milyen módon dolgozza fel. Az Arguments mező opcionális. Az utasítás végrehajtásához szükséges paraméterek meghatározására szolgál, és tartalmazhat folyam-, szolgáltatási és más kapcsolódó információt. Ezen mezők hosszának értéke változhat, a következő feltételekkel: F és A bármilyen érték lehet, amíg L+F+A <= 128. Ha L+F+A kisebb, mint 128, akkor a SID fennmaradó bitjeinek nullának kell lenniük.



2.2 ábra: Egy SRv6 SID szemantikus felosztása

Az RFC 8986-ban rögzített, szabványos SRv6 hálózatprogramozási eljárások mindegyikéhez saját opkód tartozik. Ezekhez az IANA létrehozott egy új aljegyzéket "SRv6 Endpoint Behaviors" néven a "Segment Routing" nyilvántartás alatt. Ez az alregiszter 16 bites azonosítókat tart fenn ezekhez az SRv6 instrukciókhoz. A nyilvántartás azért jött létre, hogy konzisztens alapot nyújtson azon vezérlési protokollok számára, amelyeknek hivatkozniuk kell ezekre a viselkedésekre. [1] Ugyanakkor ezek az

értékek nincsenek belekódolva a SID-en belüli Function bitekben, tehát sokkal inkább úgy kell ezekre az értékekre gondolni, mint konvencionális hozzárendelések. Ezt a Linux SRv6 implementációjával is alá lehet támasztani, ugyanis egy SID beregisztrálása és egy adott eljárás hozzárendelése között nincs összefüggés, tetszőleges SID-hez lehet tetszőleges eljárást rendelni (ezt a 2.2.1 alfejezetben részletesen be is mutatom).

A következőkben egy példa SRv6 folyamot mutatok be, mely az általam rajzolt 2.3 ábrán kerül részletezésre. Ebben a példában egy SRv6 VPN megoldást mutatok be, ahol a meglévő IPv6-os csomagot egy további, külső IPv6 fejléccel és Segment Routing Header-rel enkapszuláljuk. Ezáltal egyfajta alagutazást létesítünk A és B csomópontok között. Ez persze nem az egyetlen módja az SRv6 alkalmazásának, ugyanis, ha feltételezzük, hogy egyazon SRv6 domainen belüli csomópontok kommunikálnak egymással, akkor csak a Segment Routing Header kerül beillesztésre az IPv6 fejléc alá.

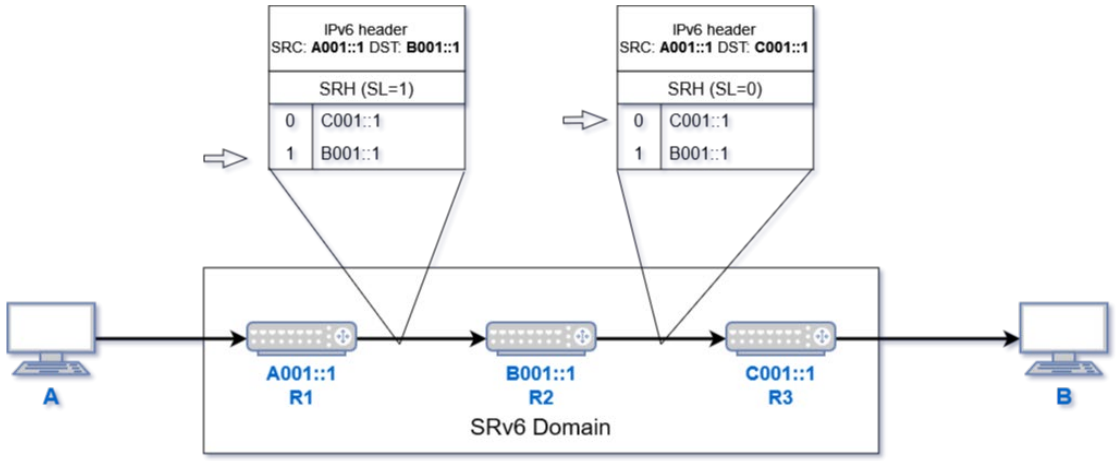
A kiindulási állapot a következő: adottak A és B jelű végpontok, akik kommunikálni szeretnének egymással. Közöttük egy SRv6-os hálózati tartomány húzódik, melyeket R1, R2 és R3 SRv6-képes routerek testesítenek meg. Ezen routerek mindegyike rendelkezik egy-egy saját SID-vel, melyek rendre A001::1, B001::1 és C001::1. A szegmensek első 4 bájtja jelen esetben tekinthető a lokátornak, a hátralevő összes bit pedig az eljárás azonosításához tartozik. Utóbbinak az értéke 1, amely egyébként az ’End’ nevű SRv6 eljárást jelöli. Ez lényegében egy SRv6 endpoint node viselkedését kódolja, mely egy alapvető feldolgozást jelent a Segment Routing Header szempontjából: eggyel csökkenti a Segment Left mező értékét, ezáltal a lista következő elemére helyezve a mutatót, és átírja az IPv6 fejléc célcím mezőjét a következő szegmensre, majd a módosított csomagot továbbítja a következő szegmens irányába.

A folyamat a következőképpen fog kinézni:

- A kiküldi a csomagot B-nek. Az ő továbbítási táblájában az R1 router fog illeszkedni, mint továbbítási csomópont a B irányába, így aztán R1-nek adja tovább a csomagot.
- R1 észlelni fogja, hogy B irányába kell csomagot továbbítani. Ennek hatására a csomagot enkapszulálja egy további IPv6-os fejléccel, illetve egy Segment Routing Headerrel is. Az IPv6 fejlécben forrásként megjelöli a saját SID-jét, célként a következő routernek (R2-nek) a SID-jét, feltölti az SRH-t R2 és R3 SID-

jeivel, a Segment Left mezőt 1-re állítja, mely jelen esetben a lista utolsó elemét fogja indexelni, majd kiküldi R2 felé.

- R2 megkapja az R1 által küldött csomagot, szintén csökkenti eggyel a Segment Left mező értékét, most már 0-ra, átírja a célcímet R3 SID-jére, és továbbítja R3-nak.
- R3 megkapja az R2-től jött csomagot. Látja, hogy a Segment Left mező értéke 0, azaz nincs már SRv6-os feldolgozó csomópont, akinek továbbítania kellene. Ekkor az R1 által felhelyezett, külső IPv6 és SRH fejlécet dekapszulálja, majd az eredeti, belső IPv6 fejléc alapján látni fogja, hogy B felé kell továbbítani a csomagot, melyet ezennel meg is tesz.



2.3 ábra: Példa SRv6 folyam

2.2.1 SRv6 implementáció Linuxon

Az SRv6 protokollban lakozó potenciált az ipar nagyobb nevei is felismerték, és többen létre is hozták saját implementációikat. Így többek között Huawei [13] és Nokia [14] gyártmányú routerek beállításai között is szerepelnek Segment Routing lehetőségek. Azonban nyílt forráskódú változatokból meglehetősen kevés opció ismert. Ezek közül talán a legismertebb és legfejlettebb a Linux kernel részeként megvalósított implementáció [15], mely a 4.10 számú verzió óta képezi a kernel szerves részét. Habár nem tekinthető egészen kiforrottnak a meglévő konfigurációs lehetőségek terén, az általa kínált funkcionalitás stabilnak és kielégítőnek mondható. A dolgozatban később részletezésre kerülő mérési környezet kiépítésében is ez az implementáció játszott fő szerepet, és a mérések esetében viszonyítási alapot is, így szükségesnek tartom bemutatni az általa nyújtott lehetőségeket.

Mint említettem, az implementáció a kernel részeként már adott, azonban a használatához szükséges bizonyos kernel változók engedélyezése (a legtöbb disztribúción alapértelmezetten be vannak kapcsolva), ezen lépések szintén fellelhetőek az implementációt dokumentáló weboldalon [15]. Ezen felül szükség lehet némely sysctl paraméterek engedélyezésére is, név szerint a '`net.ipv6.conf.*.seg6_enabled`'-re (a * helyére 'all' vagy egy meghatározott interfész neve kerül). Ezt követően a következőképp lehet SRv6 funkcionalitást konfigurálni: ha egy köztes SRv6 End csomópontról van szó, akkor az előbb említett `sysctl` paraméterek engedélyezése elegendő, és az ezeken bejövő, SRH-val rendelkező csomagok feldolgozása és továbbítása meg fog történni. Ha az SRv6 domain bemenő routeréről van szó, amely beilleszti a csomagba az SRH-t, azt a következő, az iproute2 csomag részeként meglévő paranccsal tudjuk felkonfigurálni:

```
ip -6 route add <prefix> encap seg6 mode <encapmode> segs <segments>
[hmac <keyid>] dev <device>
```

Az egyes helykitöltő értékek magyarázata:

- `prefix`: IPv6-os prefix, melyre illeszkedve végrehajtódik ez az útvonal-bejegyzés
- `encapmode`: értéke lehet 'inline' vagy 'encap'. Előbbi a meglévő IPv6 fejléc után illeszti be az SRH-t, utóbbi pedig a már korábban említett VPN megoldást teszi lehetővé.
- `segments`: az SRH-ba beillesztendő SID-ek vesszővel elválasztott listája (a 2.2-es ábra példája esetében 'B001::1,C001::1'). A megadási sorrend egyben a szegmensek bejárási sorrendje is lesz, azaz az SRH-ban a szegmens lista aljára az elsőként megadott SID kerül.
- `device`: azon interfész neve, melyen az SRH-val ellátott csomagot juttatja el a cél felé.

Ezen konfigurációs lépésekkel egy működőképes SRv6 alapú továbbítást lehet megvalósítani, azonban megfigyelhető, hogy itt még nem esett szó speciális instrukciók SID-ekhez való rendeléséről. Ebben az esetben a 'seg6' eljárást alkalmazzuk az iproute2 csomagban, mely abban az esetben teszi lehetővé SR csomagok feldolgozását, ha a csomag célcíme (beérkezéskor az aktív szegmens) egy lokális interfészen van. Az ilyen csomagok átmennek a bemeneti feldolgozó útvonalon és az IPv6 kiegészítő fejlécek feldolgozási funkcióin. Az ipv6_srh_rcv() SRH-feldolgozó függvény csak két esetet

kezel. Ha a csomópont nem az utolsó szegmens, akkor a csomagot a következő szegmens felé továbbítja. Ha a csomópont az utolsó szegmens, akkor a csomag tovább halad a bemeneti útvonalon (pl. elküldi egy helyi alkalmazásnak vagy a belső, enkapszulált csomagot továbbítja). [15]

Arra az esetre, ha szeretnénk az SRH-val rendelkező csomagok lokális feldolgozásához különböző eljárásokat is rendelni, a 'seg6' helyett a 'seg6local' módot lehet használni. Fontos azonban, hogy ebben az esetben a feldolgozandó SID-ek nem lehetnek lokális interfészekhez rendelve. Ilyenkor egy külön útválasztási táblát hozunk létre a kernelen belül, és ehhez rendeljük az egyes szegmensek feldolgozását. Ezt a következőképp lehet megtenni:

```
# echo 100 localsid >> /etc/iproute2/rt_tables
# ip -6 rule add to fc00::/64 lookup localsid
# ip -6 route add blackhole default table localsid
```

Létrehozzuk a 100-as azonosítóval rendelkező 'localsid' táblát, megadjuk, hogy az fc00::/64 prefixű szegmensek esetén ez a routing tábla legyen a mérvadó, illetve minden további szegmensre, amelyre nem definiálunk végrehajtást, azon csomagok kerüljenek eldobásra. Ha létrehoztuk ezt a táblát, a következőképpen tudjuk a SID-ekhez rendelni a feldolgozási funkciókat:

```
ip -6 route add <segment> encap seg6local action <action> <params>
dev <device> table localsid
```

Az egyes helykitöltők a következőket jelölik:

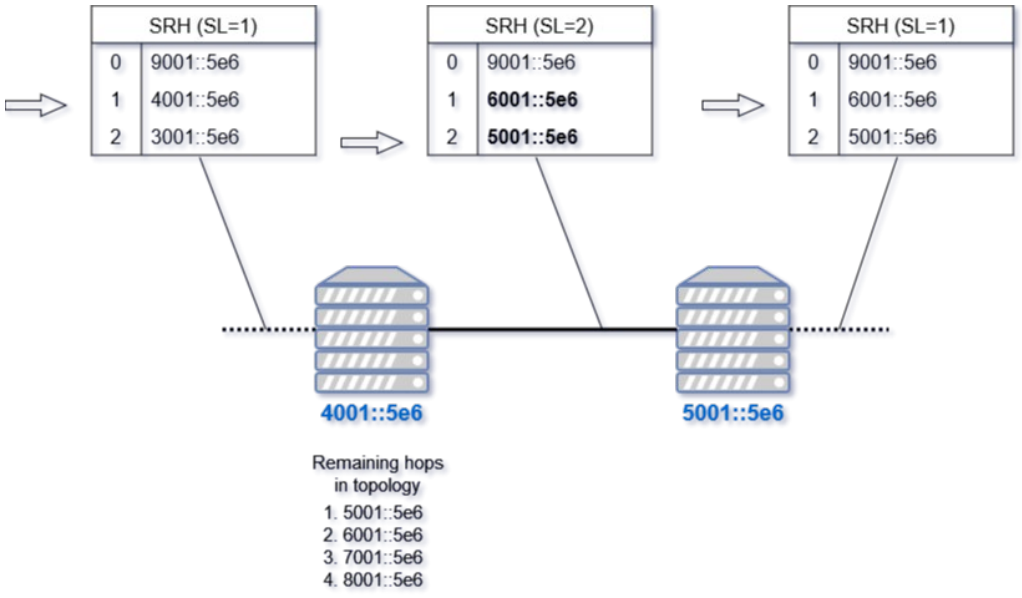
- `segment`: IPv6-os prefix, vagy teljes cím, mely magát a SID-et jelöli
- `action`: az illeszkedő szegmensen végrehajtandó funkció. Az itt megadható lehetőségek listája meglehetősen széles, így ezek részletezésére itt nem térek ki, az implementáció honlapján ezek megfelelően ki vannak fejtve [15].
- `params`: bizonyos funkciókhoz szükséges paraméterek megadásának helye
- `device`: egy lokális, nem loopback interfész neve (a működés szempontjából lényegtelen mely interfészt adjuk meg).

A Függelékben a felhasznált tesztkörnyezet konfigurációjához szükséges parancsokon keresztül részletesebben is bemutatom a fentebb részletezett parancsokat.

3 A saját SRv6-PT eljárás ismertetése

Ebben a fejezetben az általam javasolt, SRv6-PT névre keresztelt szegmens lista módosítási eljárást fejtem ki részletesebben. Kitérek a mögöttes logikára, melynek működését több ábrával is igyekszem illusztrálni, megemlítem a jelenleg még nem tárgyalt hiányosságokat, implementációs kérdéseket. Röviden kifejtem, hogy elméletben mennyivel csökkenthető így egy csomag mérete, illetve, hogy a viszonylag körülményes implementálhatósága ellenére miért érdemes mégis foglalkozni vele, majd zárásként összevetem még a vele járó potenciális előnyöket és hátrányokat.

Az általam javasolt SRv6-PT eljárás a következőket valósítaná meg. Folyamonként felbontanánk a hálózatot egy-egy lánc(rész)topológiára, melynek tagjai tisztában vannak a teljes lánc felépítéséről, a szegmens lista a kiinduláskor a teljes útvonalnak csak az első néhány szegmensét tartalmazná, viszont a legutolsó szegmensként a végső cél-szegmenst adnánk meg. Majd mikor elérkezik a csomag ahhoz az elemhez, ahol elfogynak a szegmensek, a feldolgozó ezt észleli, és úgy módosítja a szegmens listát, hogy a következő útvonal-részlet szerepeljen a listában. A továbbiakban ezen műveletet megvalósító csomópontra Central Node-ként fogok hivatkozni. Ez a művelet addig ismétlődne, amíg el nem jut a csomag a rendeltetési helyére, vagy nem kerül le róla az SR fejléc. Ezt a műveletet hivatott ábrázolni a 3.1-es ábra.



3.1 ábra: A szegmens lista csere ábrázolása egy SRv6-PT hálózat részletén

Bár maga az ötlet nem tűnik bonyolultnak, bizonyos implementációs kérdések óhatatlanul felmerülhetnek. Többek között az, hogy a hálózati elemek milyen logika mentén választják ki egymás közül, hogy kik lesznek a Central Node-ok.

Az első elképzelés az volt, hogy hasonlóan egy routing protokollhoz, a hálózat elemein valamely ágensek futnak, melyek egy közös konszenzus útján jelölik ki egymás között, hogy a teljes hálózati topológiában kik lesznek a Central Node-ok, ezzel kvázi egy speciális lefogó ponthalmazt definiálva a hálózati gráfban. Ezzel több gond is felmerül: első sorban, hogy a külön ágensek az SRv6 protokolltól függetlenül működnének. Így aztán a működés megvalósításához nem lenne elég az SRv6 alkalmazása, hanem szükség lenne kiegészítő eljárásra is. Másrészt a teljes topológiára vonatkozó Central Node kijelölés problémája NP nehéz problémának minősül, ugyanis visszavezethető a minimális lefogó ponthalmaz problémára. Továbbá az ilyen lefogásokkal nem lehetne garantálni, hogy a szegmens lista mérete minden útvonal esetén közel azonos maradna.

Azonban nem feltétlenül van szükség globális Central Node allokációra, ugyanis, ha feltételezzük, hogy a hálózat minden csomópontja implementálja ezt a működést, akkor az egyes csomópontok az egyes hálózati folyamoknak megfelelően végeznék el a szükséges szegmens lista fordítást. Azt pedig, hogy egy adott csomagon végre kell-e hajtani a műveletet, arra a csomag tulajdonságaiból és az aktuális routing konfigurációból tudna következtetni az feldolgozási pont, hasonlóképpen, mint ahogy ez az IPv4-es címfordítás esetében történik. Ehhez viszont szükséges, hogy minden csomópont ismerje a teljes hálózati topológiát, melyet legegyszerűbben valamilyen linkállapot routing protokollal lehetne megvalósítani, pl. OSPF-fel vagy IS-IS-el. Ezek jelenléte a hálózatban viszont alapvetőnek tekinthető, ugyanis az egyes szegmensek hirdetésére a teljes hálózatban, továbbá az egyes csomópontok kapcsolati információinak terjesztéséhez szükség van dinamikus routing protokollokra is. Mindezek mellett az OSPFv3 verziója külön SRv6 támogatással is rendelkezik [16], így még inkább kézenfekvővé téve ezt a fajta működést.

Ez az ötlet viszont kompatibilitási kérdéseket is feszeget. Mivel minden hálózati csomópontnak implementálnia kellene ezt az SRv6 kiegészítést, felveti annak az eshetőségét, hogy ezt az eljárást az SRv6 szabványos kiegészítő működéseként kelljen implementálni. A dolgozat egyik nem titkolt célja, hogy az említett eljárás proof-of-concept implementációjának verifikálásával akár ilyen irányba is mozdítsa a protokoll fejlesztését. Természetesen ez jelen pillanatban még távlati célnak mondható, ugyanis a

fentebb részletezett implementációs kérdések sincsenek véglegesítve, illetve verifikálva, hogy valóban működőképes lenne-e teljes valójában.

A szegmens lista ilyen módon történő cseréje mindezek mellett biztonsági kockázatot is jelenthet, ugyanis a szegmenslista illetéktelen módosítása akár a csomag útvonalának eltérítéséhez is vezethet. Ennek megoldására az SRv6-ban jelenleg is alkalmazható, HMAC TLV-hez hasonló verifikáció [17] is elképzelhető, bár ennek rögzítése egyelőre nem sürget.

Végül, de nem utolsó sorban fontos kérdés, hogy az által, hogy a csomagméret egy részét feldolgozási időre cseréljük, mekkora késleltetést viszünk be a rendszerbe, illetve mekkora ütemben romolhat a hálózat átviteli karakterisztikája? Ennek szemléltetésére tekintsük az alábbi példát: adott 10 darab hálózati csomópont, melyeken keresztül halad egy SRv6 folyam. A topológia két szélső eleme fogja végezni az SRv6-os enkapszulációt és dekapszulációt a folyam irányától függően. Ekkor, ha az enkapszulálóhoz beérkezik egy csomag, amit az SRv6 domainen keresztül kell átszállítania, a csomagra ráhelyezett SRH esetében ez 9 db szegmenst fog jelenteni. Ha ilyenkor úgy döntünk, hogy szeretnénk használni a listacsökkentő eljárást, előbb meg kell határoznunk egy határértéket, melynél nagyobb SL méretet nem engedünk meg. Ha a listaméretet 5-ben maximáljuk, az egy listacserét fog igényelni a hálózatban, a szegmens listát pedig majdnem a felére sikerült csökkenteni. Ha ezt lecsökkentjük 3-ra, a célbajutásig 4 listacserére is szükség lesz, azonban ennek következtében meg tudtuk harmadolni a lista méretét. Abban az esetben pedig, ha akár 20 hálózati csomópontról is szó lenne, a listaméret ilyen korlátokon belül tartásával még jelentősebb csökkenést könyvelhetünk el, pl. 3-as listaméret esetén 3/19 ~ 15%-a lenne az eredetinek. Azonban itt azt is figyelembe kell venni, hogy ami késleltetést a csomagméret csökkentésével megspórolunk, azt a túl sok listacseréből adódó késleltetés ne haladja meg. Ebből a szemszögből érdemes lehet majd a későbbiekben úgy implementálni az eljárást, hogy saját maga kalkulálja ki a topológia hossza és a szegmens lista cseréből adódó átlagos késleltetésből azt a maximált listaméretet, melynél a csökkentett méretből adódó nyereség és a listacsere feldolgozásából adódó hátrány kiegyensúlyozott legyen. Erre vonatkozóan a későbbiekben érdemes lehet további méréseket végezni, melyekből empirikus úton meg tudnánk határozni ezt a töréspontot.

4 A kitűzött feladat

Miután bemutattam az Segment Routing technológia és az SRv6 alapvető tulajdonságait, kifejtettem a szegmens lista méretével járó potenciális problémát, és ismertettem részletesen az ennek megoldására alkalmas eljárásomat, itt az ideje, hogy meghatározzam a dolgozat célkitűzését. A cél tehát egy proof-of-concept implementáció elkészítése, mellyel a szegmens lista konverzió sikerességét és működését lehet tesztelni a Linux csomagfeldolgozási alrendszer keretein belül.

A feladat komplexitásából adódóan az olyan megvalósítási részletek, mint a dinamikus routing protokollal történő integrálás és ezek mentén a dinamikus szegmens lista konverzió megvalósítása egyelőre nem kerülnek bele az implementációba. A megvalósított program jelenleg (nem túl elegánsan) a tesztkörnyezet jellegére lesz szabva, azaz a program azon része, mely megmondja, mire kell kicserélni a szegmens lista elemeit, statikusan megadott értékekből fog dolgozni. Természetesen az ezen a modulon végzett mérések nem lesznek teljeskörűek, hiszen feltételezve, hogy a dinamikus szegmens lista csere logikája is részét képezni a programnak, a várhatóan több és komplexebb végrehajtásnak köszönhetően tovább csökkentené a hálózat átlagos teljesítményét. Azonban az ebből adódó eredmények is már előre tudják vetíteni a végleges implementáció várható sikerességét.

5 A feladat megvalósítása

5.1 Implementációs kérdések

Miután a szegmens lista cserélési instrukció lényeges mozzanatait és a feladat fontosabb kérdéseit sikerült tisztázni, a következő kérdés az volt, hogy a proof-of-concept implementációt milyen formában készítsem el.

A cél az volt, hogy olyan megvalósítást lehessen megalkotni, melyet aztán a beépített Linux kernel implementációkkal is össze lehet vetni teljesítményben. A kernel részeként implementált (kernel-space) programok esetében a magas overheaddel járó kernel megszakításkezeléseket (interruptokat) és a processzor kontextusváltásait is meg lehet spórolni, mely lépések kiiktatásával nagyobb feldolgozási teljesítményt igénylő programokat lehet implementálni. Emellett azonban a fejlesztés során különösen is körültekintőnek kell lenni, ugyanis a nem megfelelő memóriakezelés egy kernel program esetében végzetes is lehet (hiszen nincs ott a kernel számunkra, hogy pl. a nem megengedett memóriahozzáférés mellékhatásaitól megvédjen). Végérvényben tehát kernel-space eljárások logikáját és azok hatékonyságát lehetne összehasonlítani, ezáltal pedig arra fényt deríteni, hogy a saját eljárás komplexebb működése mekkora overhead-et vinne a csomagfeldolgozásba (és nem kellene a user és kernel space közötti teljesítménybeli különbségeket figyelembe venni). Ezen logika mentén tehát kernel programot csak kernel programmal lenne érdemes összehasonlítani, így aztán a legkézenfekvőbb megoldás egy kernel modul írása lett.

A megvalósítandó modulnak a következő elemi lépéseket kell majd tudnia megvalósítani: el kell tudni ’kapnia’ a gazda rendszerbe beérkező csomagokat, ellenőriznie kell, hogy a csomag rendelkezik-e SRH-val, a fejléc olyan paraméterekkel rendelkezik, melyek hatására ki kellene váltódnia a csomagfeldolgozásnak, illetve, ha mindezek teljesülnek, akkor keresse ki a csomagban az SRH helyét és módosítsa a tartalmát, ha pedig szükséges, a csomag méretét is alakítsa ehhez.

5.2 Kernel-space implementáció

Tekintettel arra, hogy a kernel modul fejlesztés nem a legnépszerűbb műfaj, így viszonylag kevés használható forrást sikerült találnom a folyamat sajátságainak megismeréséhez. Szerencsére az elinduláshoz szükséges alapokat a „The Linux Kernel Module Programming Guide” [18] segítségével el tudtam sajátítani. További inspirációt jelentett számomra egy szintén SRv6 funkcionalitást nyújtó, nyílt forráskódú kernel modul, a „Segment Routing EXTension kernel module” (SREXT) [19], melyből az olyan implementációs részleteket, mint a csomagok Netfilter Hook-okkal történő elkapását, illetve a Socket Buffer struktúrák módosítását is sikerült mélyrehatóan áttanulmányoznom.

5.2.1 Linux kernel modul készítése

Ebben az alfejezetben röviden ismertetem a kernel modul fogalmát, bemutatom, hogy mi kell ahhoz, hogy saját modult írjunk, fordítsunk, majd ezt követően be is töltsük a kernelbe. Mielőtt belemerülnénk a modul elkészítésébe, először tisztázzuk, mit is értünk kernel modul alatt. A kernel modul egy olyan kód, mely igény szerint tölthető be a kernelbe (vagy csatolható le), és anélkül tudja kiterjeszteni a kernel funkcionalitását, hogy a rendszer újraindítására, vagy akár újrafordítására lenne szükség. Modulok nélkül a kernelfejlesztők kénytelenek lennének monolitikus kerneleket gyártani, és az új funkcionalitást közvetlenül a kernel image-be fordítani. Fontos, hogy egy modul fejlesztésekor különösen nagy körültekintéssel legyünk, ugyanis a legkisebb bug vagy rosszul végzett memóriakezelés is végzetes hibát jelenthet a rendszer szempontjából.

Most egy példán keresztül bemutatom, mely elemeket szükséges a C kódba írni ahhoz, hogy kernel modult kapjunk. A kódot egy az egyben a Linux Kernel Module Programming Guide-ból kölcsönöztem, viszont ezen lehet a legegyszerűbben bemutatni egy ilyen program fontosabb elemeit.

```
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

int init_module(void)
{
printk(KERN_INFO „Hello world.\n”);
/*
* A non 0 return means init_module failed; module can’t be loaded.
*/
return 0;
}
void cleanup_module(void)
```

```
{
printk(KERN_INFO „Goodbye world.\n”);
}
```

A ’module.h’ könyvtárat minden kernel modulhoz szükséges hozzáadni, az alapvető működés, az inicializáló és cleanup metódusok ebből a könyvtárból származnak. Egy modul C kódja a szokásos ’main’ függvény helyett két másikkal rendelkezik: egy inicializáló és egy cleanup metódussal. Előbbi, az ’init_module’ függvény, akkor fut le, mikor a modult betöltjük a kernelbe, ekkor történik az erőforrások foglalása, vagy regisztrál egy eseménykezelőt a kernelben, vagy a kernel egyik függvényét helyettesíti a saját kódjával. Utóbbi, a ’cleanup_module’ metódus pedig akkor fut le, mikor a modul eltávolításra kerül a kernelből. Itt értelemszerűen az esetlegesen foglalt erőforrások felszabadításának kell megtörténnie. A 2.3.13-as kernelverzió óta azonban nem feltétlenül kell ezt a nevet adni a két metódusnak, tetszőleges névvel is elláthatjuk őket. Ekkor a kód végén a module_init() és module_exit() függvényekkel tudjuk beregisztrálni, hogy melyik saját metódus az inicializáló és melyik a kilépési függvény. Jelen példánkban a modul betöltésekor a kernel logba ír egy „Hello, world.” -öt, majd a lecsatoláskor pedig egy „Goodbye world.” -öt.

5.2.2 Netfilter hooks

A modulunk első és legfontosabb része az lenne, hogy a hosztra beérkező csomagokat interceptálni tudja és akár módosítani is. Ennek megvalósítására lehet segítségünkre a Linux kernel részeként jelenlevő Netfilter keretrendszer [20]. A segítségével alapvetően csomagszűrést, a hálózati cím- és port fordítást (NAT), csomagnaplózást, a csomagok user-space programokba történő sorolását (Queueing) és egyéb csomagmódosítást is lehetővé tesz.

A keretrendszer számunkra érdekes funkcionalitását a Netfilter Hooks jelenti. A netfilter hooks a Linux kernelen belül lehetővé teszi a kernelmodulok számára, hogy a Linux networking-stack különböző pontjain callback hívásokat regisztráljanak. A regisztrált függvényt ezután minden olyan csomag esetén visszahívja a rendszer, amely áthalad a Linux networking-stack megfelelő hook-ján. Ezek a hook-ok név szerint a következők:

- PRE_ROUTING: mikor egy csomag frissen érkezik be a hálókártyán, és semmilyen hálózati feldolgozás nem kezdődött még el

- LOCAL_IN: akkor hívódik, mikor a csomagról kiderült, hogy ennek a hosztnak került címzésre
- FORWARD: akkor hívódik, mikor egy bejövő csomagnak egy másik interfészen keresztül el kell hagynia a hosztot
- LOCAL_OUT: mikor egy lokálisan létrejött csomagnak el kell hagynia a hosztot
- POST_ROUTING: akkor hívódik meg, mikor már eldőlt, hogy a csomag kifelé halad és végbe mentek a feldolgozási lépések, és már csak a hálókártyán történő kiküldés van hátra

Esetünkben a PRE_ROUTING hookot használjuk. Mindezt azért, hogy ha van illeszkedő csomag, amin végre kellene hajtanunk a szegmens lista cserét, azt még azelőtt tudjuk megtenni, hogy a network-stack beépített SRv6 feldolgozása végbemenne, és ezáltal meg tudjuk kerülni ezt a fajta feldolgozást.

Egy kernel modulon belül a követezőképpen tudunk egy hookpointra callback hívást regisztrálni. Létre kell hoznunk egy ’nf_hook_ops’ struktúrát, melyben a következő paraméterek találhatóak:

```
struct nf_hook_ops {
     /* User fills in from here down. */
     nf_hookfn           *hook;
     struct net_device   *dev;
     void                *priv;
     u_int8_t            pf;
     unsigned int        hooknum;
     /* Hooks are ordered in ascending priority. */
     int            priority;
};
```

Nekünk ezek közül most a következő négy elem lesz fontos: a ’hook’ jelöli azt a callback függvényt, amit be szeretnénk regisztrálni, és ami majd esetünkben a csomagok feldolgozását, elfogadását vagy eldobását fogja végezni. A modul kódjában ez a ’modify_packet’ nevű függvény lesz, melyből fontosabb részleteket a későbbi alfejezetekben lehet majd látni. A ’pf’ tag a packet family-t jelöli, vagyis, hogy milyen protokollt alkalmazó csomagokat akarunk kiszűrni (esetünkben PF_INET6-ot, azaz IPv6-os csomagokat). A ’hooknum’ azonosítja azt a hookpointot, amire a callbacket regisztrálni szeretnénk, ez nekünk az NF_IP_PRE_PROUTING lesz. Illetve a ’priority’ a hook prioritása (abban az esetben, ha más hook is regisztrálva lenne a rendszerben). Ez a prioritás a netfilter_ipv4.h fájlban definiált nf_ip_hook_priorities enumban meghatározott prioritások egyike lehet (mi most a NF_IP_PRI_FIRST-öt fogjuk

használni). Ezt a feltöltött struktúrát aztán az nf_register_net_hook() függvénnyel tudjuk beregisztrálni.

A ’modify_packet’ függvényünk regisztrálása az init_module függvényben a következőképpen fog kinézni:

```
hook_function = (struct nf_hook_ops*)kcalloc(1, sizeof(struct nf_hook_ops),
GFP_KERNEL);
hook_function->hook  = (nf_hookfn*)modify_packet;   /* hook function */
hook_function->hooknum    = NF_INET_PRE_ROUTING;    /* received packets */
hook_function->pf    = PF_INET6;                    /* IPv6 */
hook_function->priority   = NF_IP6_PRI_FIRST;  /* max hook priority */

int reghook = nf_register_net_hook(&init_net, hook_function);
```

A beregisztrált függvényünket a cleanup_module függvényünkben fel kell majd szabadítani, ezt az nf_unregister_net_hook() függvénnyel tudjuk megtenni.

A Netfilter hookpointra beregisztrálásnál látszik, hogy egy függvénypointert adunk át az nf_hook_ops struktúrában. Ez azt fogja jelenteni, hogy a callback függvényünknek ugyanolyan függvény szignatúrát kell megvalósítania, mint az nf_hookfn függvénynek:

```
unsigned int nf_hookfn(void *priv, struct sk_buff *skb,
                                    const struct nf_hook_state *state);
```

A függvény lehetséges visszatérési értékei közül néhány fontosabb:

- NF_ACCEPT: a csomag elfogadása, és a feldolgozás folytatása
- NF_DROP: a csomag eldobása
- NF_STOLEN: a csomag eltérítése, jelezzük a kernel számára, hogy a csomag feldolgozását és az esetleges kiküldését teljes mértékben mi kezeljük innentől.
- NF_QUEUE: egy meghatározott netfilter queue-ba továbbítja a csomagot, melyhez user-space alkalmazások tudnak majd hozzáférni

Ezen felül három különböző paramétert kap az így regisztrált függvényünk, melyek közül számunkra a legfontosabb paraméter most az skb lesz, mely a netfilter hook által elkapott csomagok struktúrája. Ez a Socket Buffer, pontosabban az sk_buff struktúra lesz.

5.2.3 Az sk_buff struktúra és módosítása

Az sk_buff struktúra adja a Linux kernel számára a különböző kommunikációs protokollok által közvetített csomagok egységes felépítését. Ez azt jelenti, hogy nem csak

az IP csomagok, de pl. a Bluetooth adatátviteli adategységek is ebben a formában kerülnek reprezentálásra a kernel szolgáltatásai számára. A struktúra felépítését tekintve meglehetősen összetett, rengeteg metaadat és egyéb segéd struktúra tárolását valósítja meg. A teljes struktúra bemutatása a nagy kiterjedése, és az egyes kernelverziók közötti apróbb eltérések miatt nem lenne indokolt, illetve a feladat megvalósítása szempontjából számunkra csak a buffer rész lesz érdekes.

Amikor a hálózati eszközön beérkezik egy csomag, a hálókártya drivere a netdev_alloc_skb() metódushívással memóriát allokál egy új sk_buff-nak. A buffert négy fontosabb pointer osztja fel: head, data, tail és end. A head-data és a tail-end mutatók közötti memóriaterület üres, előbbit headroomnak, utóbbit tailroomnak is szokás nevezni. A csomaginformáció a data és tail közé kerül. Az üres memóriaterületek lehetővé teszik, hogy a csomag feldolgozási folyamata során a csomag elejéhez és végéhez szabadon lehessen további információt hozzáfűzni. Ez a kialakítás teszi lehetővé, hogy egy sk_buff-ot rétegről rétegre lehessen építeni, és hogy a csomag tartalmának módosítása esetén ne kelljen további memóriát foglalni. Például, ha lokálisan generálódik egy csomag, akkor az egyes hálózati rétegek felhelyezése a feldolgozó metódusok sorrendjében kerüljenek a csomagba beillesztésre. Így a hasznos adat fölé kerülnek közvetlenül a transzport réteg, majd a hálózati és az adatkapcsolati réteg fejlécei.

A saját kernel modul esetében a használata a buffer tartalmának, és bizonyos estekben a buffer méretének módosítására fog koncentrálódni. Miután a korábban már mutatott ’modify_packet’ függvényünk meghívódott, paraméterként megkapva egy sk_buff-ot, nekiláthatunk a csomag tartalmának elemzéséhez. Ennek megkönnyítésére a kernel részeként léteznek különböző előre definiált struktúrák, melyek gyakorlatilag pointereket tárolnak csak az sk_buff megfelelő pontjaira. Így, ha létrehozunk egy ilyen struktúrát és azt megfelelően feltöltjük, akkor ezen struktúra mezőin végzett változtatások az eredeti sk_buff esetében is végbe fognak menni. Ilyen speciális struktúrák az ’ipv6hdr’ és az ’ipv6_sr_hdr’ is. Ezen struktúrák inicializálása a következőképpen történik.

```
int srh_offset = 0, srh_proto;
struct ipv6hdr* iph;
struct ipv6_sr_hdr* srh;
…
iph = ipv6_hdr(skb); //ipv6 fejléc
…
srh_proto = ipv6_find_hdr(skb, &srh_offset, NEXTHDR_ROUTING, NULL, NULL);
…
srh = (struct ipv6_sr_hdr*) (skb->data + srh_offset); //srh fejléc
```

Először ellenőrizzük, hogy a csomagok tartalma szemantikailag helyes-e, pl. az IPv6 hop limit nagyobb-e mint 0, az SRH segment left mezője nagyobb vagy egyenlő-e 1-nél, illetve nem nagyobb, mint a last segment mező értéke stb. Ha itt valami rendellenességet tapasztalunk, akkor azokat a csomagokat egyszerűen eldobjuk. Ezt követően ellenőrizzük, hogy az SRH segment left mezőjének értéke micsoda. Ha ez 1, és az utolsó szegmens a listában már egy hop-ra van tőlünk, akkor normális SRv6 feldolgozást követően tovább engedjük a csomagot. Ha viszont nem érhető el egy hopon belül az utolsó szegmens, akkor szükség lesz a Central Node viselkedés megvalósítására.

```
if(srh->segments_left==1){
        if(isdirectlyconnected(&srh->segments[0])==1){
             goto acc; //csomag elfogadása, sima srv6 feldolgozás
        } else {
             central_node_behavior(skb,iph,srh); //central node viselkedés
             goto acc; //módosítást követően a csomag elfogadása
        }
}
```

A Central Node viselkedést egy további függvény, a ’central_node_behavior’ valósítja meg, melynek feladata, hogy a paraméterként megkapott skb-t megfelelően módosítsa. A függvény elején a következő értékek kerülnek meghatározásra:

```
int currentsize_ofseglist = srh->first_segment + 1;
int desiredsize_ofseglist = MAX_SEG_SIZE; // desired implementation:
//(REMAINING_HOP_ON_PATH >= MAX_SEG_SIZE) ? MAX_SEG_SIZE :
REMAINING_HOP_ON_PATH
```

A ’currentsize_ofseglist’ a kapott skb szegmens listájának méretét rögzíti, a ’desiredsize_ofseglist’ pedig ami a módosításra váró csomagban az elvárt szegmens lista méret lenne. Ez a jelenlegi implementációban mindig a maximált lista méret lesz (MAX_SEG_SIZE), azonban a későbbi implementációkban ez az érték a topológia aktuális állapotától is függhet. Például, ha a listacserére olyan hoszton kerülne sor, amely MAX_SEG_SIZE -1 -nél kevesebb lépésre lenne a célcímtől, akkor természetesen kevesebb szegmensnek kellene belekerülnie a listába. Ennek a logikája látható a ’desiredsize_ofseglist’ értékadásának sorában látható kommentben.

A kód innentől 3 féle kiértékelés felé mehet. Ha a jelenlegi listaméret nagyobb, mint az elvárt, akkor az skb méretének csökkentésére, ha kisebb, akkor bővítésre, ha pedig megegyezik a két érték, akkor a méret módosítására nem kerül sor. A következő kódsorok az skb méretének csökkentését mutatják be.

```
unsigned short seglist_diffbytes =
                        (currentsize_ofseglist-desiredsize_ofseglist)*16;
unsigned short seglist_diffbytes_sw_endian =
```

```
                        (seglist_diffbytes>>8) | (seglist_diffbytes<<8);
int new_srh_len = desiredsize_ofseglist*16 + 8;
new_ipv6_hdr = (struct ipv6hdr *) skb_pull(skb, seglist_diffbytes);
new_srh =
(struct ipv6_sr_hdr *)((char *) new_ipv6_hdr + sizeof(struct ipv6hdr));
new_ipv6_hdr->payload_len   = iph->payload_len – seglist_diffbytes_sw_endian;
```

A ’seglist_diffbytes’ jelöli, hogy hány bájttal kell csökkenteni a csomag méretét. Amikor értéket adunk az új IPv6 fejlécet kódoló struktúrának, abban egy speciális, sk_buff struktúrák módosítására alkalmas függvényt is hívunk. Ez az ’skb_pull’ függvény, mely paraméterként kapja, hogy hány bájttal kell előrébb mozgatni a ’data’ pointert, és a visszatérési értéke egy mutató a ’data’ pointer új helyére. Ennél a pontnál lehet kezdeni újraépíteni az IPv6 fejlécet. Az ezt követő SRH kezdőpontját pedig 40 bájttal hátrébb, azaz az ipv6hdr struktúra méretével előrébb adtam meg. (Megjegyzés: az előrébb szó jelen esetben a ’data’ pointertől a ’tail’ pointer felé menő irányt jelöli.) Az egyes fejlécek elhelyezkedései most már adottak, a struktúrák mezői későbbi lépések során kerülnek feltöltésre. Egy feladatunk van még, hogy az skb fontosabb mutatóit, melyek a hálózati és a transzport réteg kezdeteire mutatnak, a ’data’ pointer eltolásának függvényében ugyanakkora értékkel toljuk előrébb. Az adatkapcsolati réteg esetében más lesz a helyzet. Az skb ’data’ pointere a kernel modulba kerüléskor a hálózati réteg kezdetére mutatott, így, ha itt csomagméret módosításra kerülne sor, a data pointer fölötti rész, azaz az adatkapcsolati réteg integritása sérülhet. Ilyenkor az ’skb_mac_header_rebuild’ függvényhívással helyesen újra tudjuk építeni ezt a réteget is.

```
skb_mac_header_rebuild(skb);
skb_set_network_header(skb, seglist_diffbytes);
skb_set_transport_header(skb, seglist_diffbytes);
```

A csomagméret módosítás logikája a növelő esetben is nagyon hasonló, ott az ’skb_pull’ helyett az ’skb_push’-t használjuk, mely értelem szerint visszafelé mozgatja a ’data’ pointert.

Ezt követően pedig már az új IPv6 és SR fejlécek tartalmának feltöltése következik. A legtöbb esetben itt az eredeti csomag fejléceinek tartalma kerül másolásra, kivétel persze az IPv6 ’payload length’, a ’destination address’, az SR fejléc esetében a ’segment left’, a ’header length’, a ’first segment’ és persze a szegmenseket tároló tömb elemeinek átírása. A szegmens lista csere például a következőképpen néz ki: ’in6_addr’ struktúrákban definiálunk IPv6 formátumú címeket, melyek jelen esetben a SID-eket fogják reprezentálni, majd az új SRH-t alkotó, ’new_srh’ struktúra szegmenseit tartalmazó tömb elemei kerülnek megadásra.

```
struct in6_addr topo_first = {.in6_u.u6_addr8={0x20, 0x01, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x05, 0xe6}};
struct in6_addr topo_last  = {.in6_u.u6_addr8={0x90, 0x01, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x05, 0xe6}};

if( ipv6_addr_equal(&srh->segments[0], &topo_first) ) {
    struct in6_addr s1 = {.in6_u.u6_addr8={0x30, 0x01, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x05, 0xe6}};
    struct in6_addr s2 = {.in6_u.u6_addr8={0x40, 0x01, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x05, 0xe6}};
    struct in6_addr s3 = {.in6_u.u6_addr8={0x50, 0x01, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x05, 0xe6}};

    new_srh->segments[0]=topo_first;
    new_srh->segments[1]=s1;
    new_srh->segments[2]=s2;
    new_srh->segments[3]=s3;
    new_ipv6_hdr->daddr = s3;
}
```

Ezzel a kernel modul csomagmódosítását megvalósító kódrészletek végére, és a teljes kernel modul ismertetésének végére értünk. A következő lépés a modul kódjának lefordítása, a kernelbe történő hozzáadása, és a használatának rövid bemutatása van hátra.

5.2.4 A kész modul telepítése és használata

Egy modul fordításához ’gcc’ fordítóra és a ’make’ eszköz meglétére lesz szükségünk. A könyvtárban, ahol létrehoztuk a C kódunkat, létrehozunk egy Makefile-t is. Ez a következőképpen kell, hogy kinézzen:

```
obj-m += srv6mod.o
all:
make -C /lib/modules/$(shell uname -r)/build M=$(pwd) modules
clean:
make -C /lib/modules/$(shell uname -r)/build M=$(pwd) clean
```

Az obj-m adja meg a cél objektum fájlt, amit a Makefile buildelni fog. Az ’all’ az alapértelmezett belépési pont a Makefileban. A /lib/modules/$(shell uname -r)/build elérési útvonal a megfelelő kernel forrás- és build fájlok felkeresésére szolgál a kernel aktuálisan futó verziója alapján (uname -r). Az M=$(pwd) az aktuális munkakönyvtárat állítja be a modul forráskódjának helyeként. Ez biztosítja, hogy a kernel build rendszere tudja, hol találja meg a modul forrásfájljait. A make parancs végén a ’modules’ arra utasítja a kernelépítő rendszert, hogy kernel modult buildeljen, és ne a teljes kernelt. A ’clean’ blokkban pedig arra utasítjuk a make-et, hogy a buildelés során keletkező ideiglenes fájlokat távolítsa el.

Ezt követően a ’make’ parancsot kiadva lefordul a modulunk, és egy srv6mod.ko kiterjesztésű fájlt fogunk kapni, melyet most már be is tölthetünk a kernelbe. Ezt a legegyszerűbben az ’insmod’ paranccsal tudjuk megtenni: `sudo insmod srv6mod.ko`

Ha minden jól ment, az ’lsmod’ parancsot kiadva meg kell jelennie a modulunknak a listában. Ha a kernel logjaira is vetünk egy pillantást (`dmesg` parancs), akkor észrevehetjük a kernel modulunk által hagyott üzenetet, melyet a modult inicializáló függvény lefutásakor hajtunk végre („SRv6 Policy Translator Module loaded.”).

Példának a 6.1-es ábrán látható mérési környezet telepítését szeretném említeni. A működés megvalósításához az S5 és S6 jelű virtuális gépek esetében kellett a kernel modult fordítani és telepíteni, előbbi az S1→S10 irányba, utóbbi az S10→S1 irányba menő forgalom esetén végezte a listacserét (hogy a lista mérete mindkét irány mindkét szakaszában 4 maradhasson). A modulok telepítését követően az S2 és S9 VM-eken, melyek az SRv6 domain be- és kilépési pontjai voltak, külön be kellett állapítani, hogy a csomagenkapszulációt hány szegmens beillesztésével tegyék meg. Mivel ez a feladat a mérések során meglehetősen sokszor előfordult, így erre a feladatra egy bash scriptet hívtam segítségül a redundáns konfigurációk érvényre juttatására. Ennek tartalma az ’A’ Függelékben található meg. Ezután minden készen áll arra, hogy leteszteljük, áthalad-e a forgalom sikeresen az SRv6 domainen. Az SRv6-PT-t használó topológia esetében az 5.1 és 5.2 ábrákon látható, hogy sikeresen végzi a modul a feladatát, és mindkét irányba megfelelően megérkeznek a csomagok.

```
ubuntu@srv6-1:~$ ping f009::20
PING f009::20(f009::20) 56 data bytes
64 bytes from f009::20: icmp_seq=1 ttl=63 time=7.52 ms
64 bytes from f009::20: icmp_seq=2 ttl=63 time=6.71 ms
64 bytes from f009::20: icmp_seq=3 ttl=63 time=7.31 ms
64 bytes from f009::20: icmp_seq=4 ttl=63 time=7.30 ms
64 bytes from f009::20: icmp_seq=5 ttl=63 time=6.42 ms
^C
--- f009::20 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 6.419/7.050/7.523/0.416 ms
```

5.1 ábra: Sikeres ICMPv6 Echo Request/Reply forgalom a szegmens lista csere alkalmazásával

```
ubuntu@srv6-1:~$ iperf3 -c f009::20
Connecting to host f009::20, port 5201
[  5] local f001::10 port 58246 connected to f009::20 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   181 MBytes  1.52 Gbits/sec    6    743 KBytes
[  5]   1.00-2.00   sec   116 MBytes   975 Mbits/sec    3    859 KBytes
[  5]   2.00-3.00   sec   204 MBytes  1.71 Gbits/sec    1   1.20 MBytes
[  5]   3.00-4.00   sec   221 MBytes  1.86 Gbits/sec    2   1.53 MBytes
[  5]   4.00-5.00   sec   248 MBytes  2.08 Gbits/sec   12    968 KBytes
[  5]   5.00-6.00   sec   228 MBytes  1.91 Gbits/sec    3   1.20 MBytes
[  5]   6.00-7.00   sec   220 MBytes  1.85 Gbits/sec   11    480 KBytes
[  5]   7.00-8.00   sec   180 MBytes  1.51 Gbits/sec    7    472 KBytes
[  5]   8.00-9.00   sec  71.2 MBytes   598 Mbits/sec   13    348 KBytes
[  5]   9.00-10.00  sec  60.0 MBytes   503 Mbits/sec   11    457 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  1.69 GBytes  1.45 Gbits/sec   69             sender
[  5]   0.00-10.01  sec  1.68 GBytes  1.45 Gbits/sec                  receiver

iperf Done.
```

5.2 ábra: TCP forgalom a szegmens lista csere alkalmazásával

Az egyik virtuális gépen (S5) a ICMPv6 Echo Request/Reply forgalommal párhuzamosan tcpdump-pal kaptam el a csomagokat. Wireshark-kal megnyitva a capture fájlt, kiválóan látszik a szegmens lista csere hatása (5.4 és 5.5 ábrák).

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	f001::10	f009::20	ICMPv6	232	Echo (ping) request id=0x0082, seq=1, hop limit=64 (no response found!)
2	0.000060	f001::10	f009::20	ICMPv6	232	Echo (ping) request id=0x0082, seq=1, hop limit=64 (reply in 3)
3	0.008467	f009::20	f001::10	ICMPv6	232	Echo (ping) reply id=0x0082, seq=1, hop limit=64 (request in 2)
4	0.008498	f009::20	f001::10	ICMPv6	232	Echo (ping) reply id=0x0082, seq=1, hop limit=64

5.3 ábra: ICMPv6 Echo Request/Reply üzenetváltás 4 csomagja a szegmens lista csere alkalmazásával

```
▾ Routing Header for IPv6 (Segment Routing)
    Next Header: IPv6 (41)
    Length: 8
    [Length: 72 bytes]
    Type: Segment Routing (4)
    Segments Left: 1
    Last Entry: 3
    Flags: 0x00
    Tag: 0000
    Address[0]: 9001::5e6
    Address[1]: 5001::5e6
    Address[2]: 4001::5e6
    Address[3]: 3001::5e6
```

5.4 ábra: ICMPv6 echo request üzenet SRH-ja a szegmens lista csere előtt

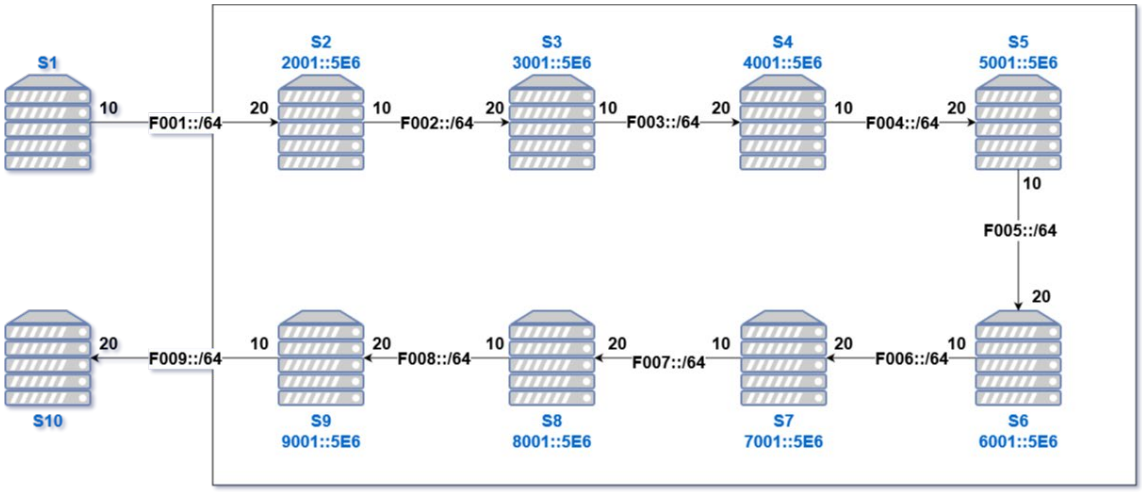
```
▾ Routing Header for IPv6 (Segment Routing)
    Next Header: IPv6 (41)
    Length: 8
    [Length: 72 bytes]
    Type: Segment Routing (4)
    Segments Left: 3
    Last Entry: 3
    Flags: 0x00
    Tag: 0000
    Address[0]: 9001::5e6
    Address[1]: 8001::5e6
    Address[2]: 7001::5e6
    Address[3]: 6001::5e6
```

5.5 ábra: ICMPv6 echo request üzenet SRH-ja a szegmens lista csere után

6 Teljesítményelemzés

6.1 Mérési környezet és módszerek

A teljesítményelemzésekhez szükséges mérések alapját egy OpenStack klaszterben telepített virtuális gépek adták. A várhatóan kevésbé intenzív kihasználtság miatt a VM-eknek elegendő volt allokálni 2 VCPU-t, 4GB RAM-ot és 40GB-nyi diszket, továbbá mindegyikükön Ubuntu 20.02 operációs rendszer futott. Szám szerint 10 VM-et hoztam létre, melyeket nemes egyszerűséggel lánc-topológiában kapcsoltam össze, ezzel egyetlen csomagfeldolgozási útvonalat emulálva. A különböző hálózati konfigurációk és mérési esetek könnyebb kezelhetősége miatt két párhuzamos láncot hoztam létre: egyiken a sima IPv6-os továbbítás, másikon pedig az SRv6-ot használó megoldások forgalmai haladtak át. Utóbbiak esetében az egyes mérések között többször is kellett állítani az egyes hálózati konfigurációkon, melyet bash szkriptek futtatásai tettek könnyebben megvalósíthatóvá. Ezek tartalma az 'A' Függelékben található. A lánc két végén elhelyezkedő VM-ek folytattak egymással kommunikációt, melyeken nem került SRv6 funkcionalitás konfigurálásra. A többi 8 csomópont kvázi egy SRv6 hálózati domaint alkotott, melyen keresztül, alagutazással haladt át a két szélső VM közötti hálózati forgalom. Ez a topológia látható a 6.1-es ábrán is, feltüntetve az egyes VM-ek között húzódó hálózati kapcsolatokat, illetve a bekeretezett VM-ek jelölik az SRv6 domaint.



6.1 ábra: A mérési környezet felépítése OpenStackben

A méréseket a következőképpen terveztem meg. Ahhoz, hogy megfelelő képet alkothassunk a SRv6-PT modulom általános hálózati teljesítményéről, több paraméter

szerinti méréseket is végezni kellett. A végponttól végpontig terjedő késleltetésre ICMPv6 echo request-reply üzenetpárosok küldését használtam, melyek esetén a körülfordulási idejüket (round-trip time) vizsgáltam. Emellett a két leggyakoribb, transzport rétegbeli protokoll esetében is végeztem teljesítményelemzést, TCP esetében a hálózati áteresztőképességet, UDP esetében pedig egy fixen megadott adatátviteli sebesség (1 Gbps) melletti csomagvesztési arányokat vizsgáltam. Ezeken felül egy további protokoll esetében is végeztünk méréseket, mely az SCTP volt. Ennek a protokollnak a tesztelésére többek között az adta az ötletet, hogy 5G core-ban több helyen is javasolják a használatát [21], másrészt így a modul működését egy nem szokványos protokollal is verifikálni tudtam.

A fentebb említett paramétereket 3(+1) különböző továbbítási módszer esetében vizsgáltam: a teljes topológián, végponttól végpontig terjedő, klasszikus IPv6 alapú útválasztás esetében (statikus route bejegyzések megadásával), egy normál SRv6 alapú adatsíkon, mely esetben a csomagok a teljes topológia szegmens listáját tartalmazták (összesen 7 db szegmenst, és minden köztes SRv6 csomóponton az ’End’ funkció kerül végrehajtásra a csomagok feldolgozásakor), illetve az SRv6-PT modult alkalmazó adatsík esetében. Utóbbit két további szcenárióra bontottam: az egyik esetben a maximális szegmens lista méret 4, a másik esetében 3 volt. Ez a hálózat szempontjából azt jelentette, hogy listacserére az előbbi esetben egyszer, a második esetben kétszer volt szükség. Ennek oka az volt, hogy megnézzem, mekkora feldolgozási overheaddel járhat a kernel modulon belüli csomagfeldolgozás, többszöri szegmens lista módosítás esetén is. A mérési környezet méretbeli korlátjai miatt a szegmens lista cserék számát nem tudtam tovább növelni, azonban a kétszeri feldolgozásból származó eredmények esetében is következtethetünk a modullal járó teljesítményváltozás nagyságára.

Felhasznált protokollok	ICMPv6	TCP	UDP	SCTP
Vizsgált paraméterek	Echo request/reply RTT (ms)	Adatátviteli sebesség (Mbit/s)	csomagvesztési arány (%)	Adatátviteli sebesség (Mbit/s)

6.1 táblázat: A mérések során vizsgált protokollok és paramétereik

Mérési forgatókönyvek	Normál **IPv6** alapú útválasztás	**Normál SRv6** útválasztás (7 szegmens)	**SRv6-PT (1 lista cserével**, 4 szegmens)	**SRv6-PT (2 lista cserével**, 3 szegmens)

6.2 táblázat: A mérési forgatókönyvek összefoglalva

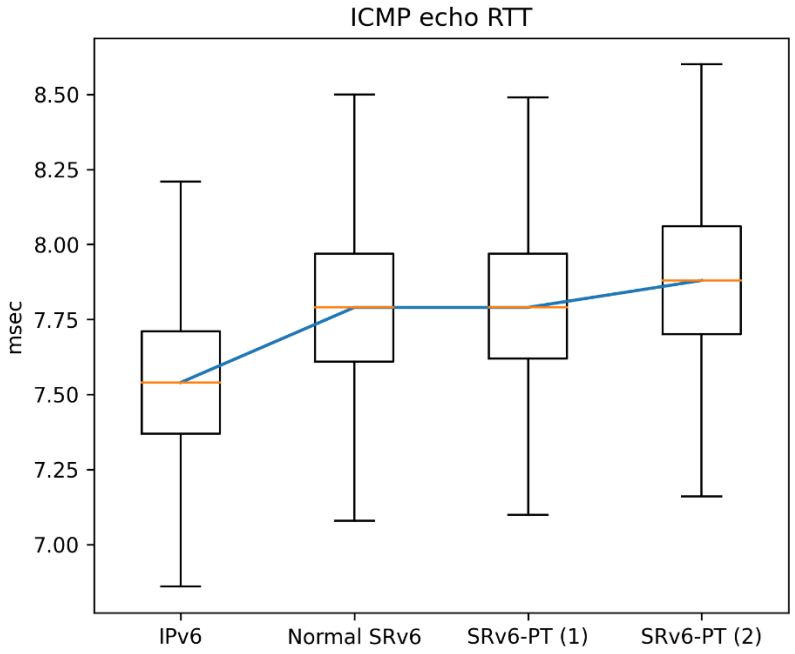
A mérések tehát a következők voltak a fenti 3 eset mindegyikében:

- 10000 db ICMPv6 Echo üzenet küldése és a körülfordulási idő mérése
- 100 db 1 perces TCP folyam generálása és az áteresztőképesség mérése
- 100 db 1 perces UDP folyam generálása és a csomagvesztési arány mérése
- 100 db 1 perces SCTP folyam generálása és az áteresztőképesség mérése

Az ICMPv6 Echo késleltetés méréséhez a ’ping’ parancsot, míg a többi méréshez az ’iperf3’ nevű eszközt használtam, melynél a tesztek többszöri ismétlésének megvalósításához kiegészítő bash szkripteket használtam. A mérések során alkalmazott parancsok és szkriptek tartalma a ’B’ Függelékben található.

6.2 Eredmények kiértékelése

A fentebb említett mérésekből generált eredmények statisztikái a következő ábrákon és táblázatokban láthatóak. A mérések pontosságát illetően a következőt tudom állítani: a dolgozatban feltüntetett eredmények kiértékelései, a diagramok és a részletes statisztikák mindegyike az utoljára végzett, konzisztensnek vett hálózati működés során keletkezett mérésekből vannak. Bár bizonyos anomáliák előfordultak a különböző alkalommal végzett mérések esetében, a teljesítménybeli megoszlás azonos volt minden alkalommal. Rögzítettem például, hogy a körülfordulási idő átlagosan 1 ms-al kisebb volt minden forgatókönyvben, és az is, hogy az átlagos TCP átviteli sebesség akár 1000 Mbit/sec-el is nagyobb volt. Tekintettel arra, hogy nem csak egyetlen, jól definiált környezetben lehetne alkalmazni a megoldásomat, és a mérések célja is az volt, hogy a saját implementáció teljesítményét arányaiban vessem össze a létező implementációkkal, így a mérési eredményeket konzisztensnek lehet tekinteni.

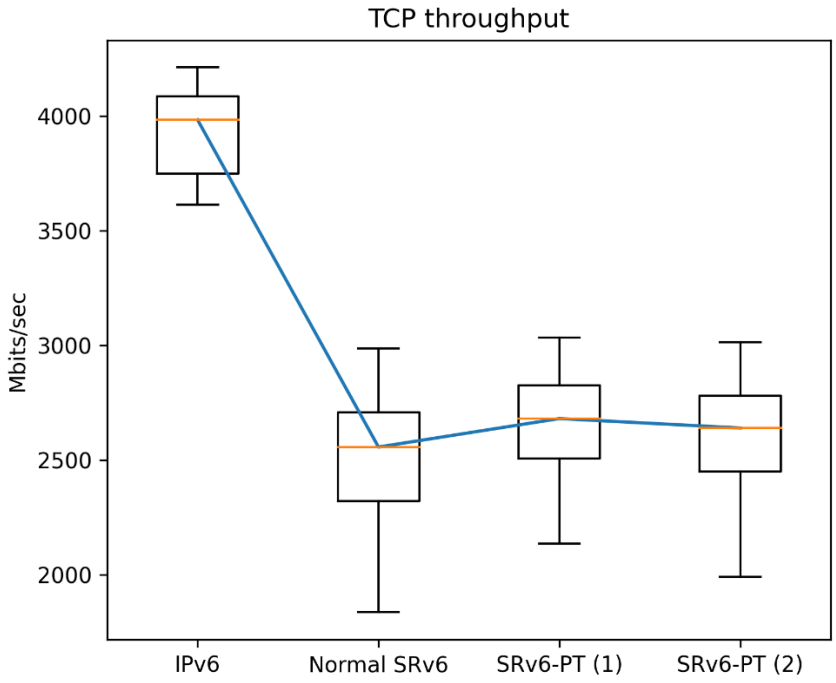


6.2 ábra: A körülfordulási idő (RTT) méréseinek diagramja

	Min	Max	Átlag	Medián	Szórás
IPv6	6,48	17,8	7,565	7,54	0,408
Normal SRv6	6,78	16,8	7,817	7,79	0,405
SRv6-PT (1)	6,75	17,6	7,825	7,79	0,42
SRv6-PT (2)	6,54	15,2	7,907	7,88	0,41

6.3 táblázat: A körülfordulási idő (RTT) méréseinek statisztikája (milliszekundumban)

Itt az volt a feltételezés, hogy az SRv6 a komplexebb feldolgozása miatt általánosságban is kicsit több késleltetést visz a rendszerbe, a lista cserét alkalmazó forgatókönyvek pedig még tovább fogják emelni ezt. Látható, hogy az IPv6-os eset adja mind közül a legkisebb késleltetést, azonban ez csak alig 3-4%-nyi eltérést mutat az egyes protokollok átlagos késleltetéséhez képest. A normál SRv6 és az egyszeri lista cserét megvalósító SRv6-PT eset között szinte nincs különbség, míg a kétszeri listacserét megvalósító esetben 1% növekedés tapasztalható az átlagos késleltetésben az egyszerihez képest.

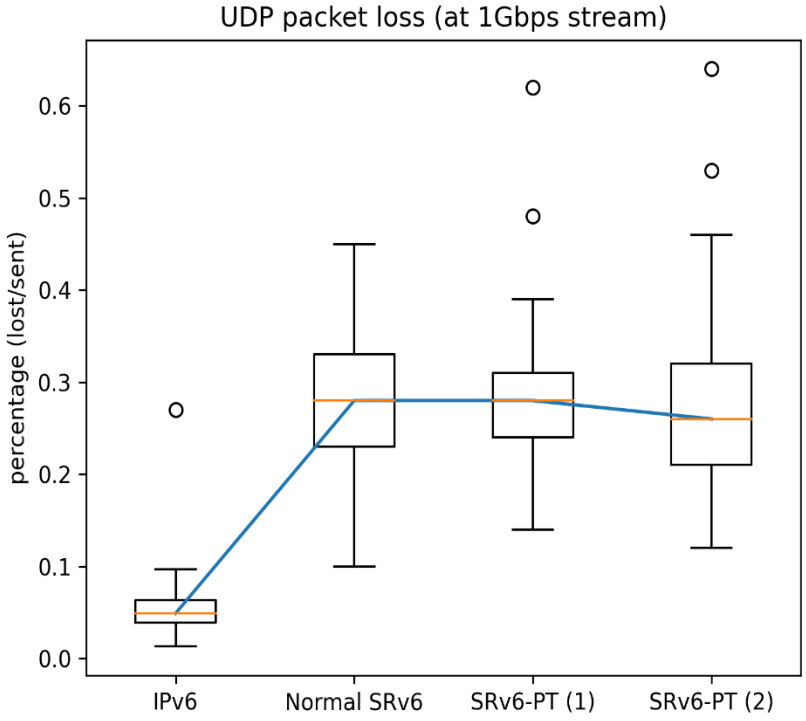


6.3 ábra: A TCP áteresztőképesség méréseinek diagramja

	Min	Max	Átlag	Medián	Szórás
IPv6	2856	4211	3820,6	3982,5	388,04
Normal SRv6	685	2986	2432,86	2557	449,63
SRv6-PT (1)	1928	3034	2653,86	2681	218,77
SRv6-PT (2)	1916	3013	2598,2	2640	268,98

6.4. táblázat: A TCP áteresztőképesség méréseinek statisztikája (Mbit/sec)

A TCP esetében is folytatódott a papírforma, viszont itt már szignifikánsabb különbséget lehetett kimutatni a normál IPv6-os és az SRv6-ot használó esetek között. A normál SRv6 közel 40%-nyi teljesítménycsökkenést mutatott a TCP átvitel esetében az IPv6-hoz képest. Az egyszeri listacsere esetében viszont már minimális javulás tapasztalható, mely jelenséget akár a kisebb csomagméretből adódó gyorsabb feldolgozás is magyarázhatja. A kétszeri csere esetében alig 2%-ot visszaesett az átvitel, viszont ez egyáltalán nem nevezhető szignifikánsnak, sőt, még így is jobban teljesít, mint a teljes szegmens listát használó megoldás.



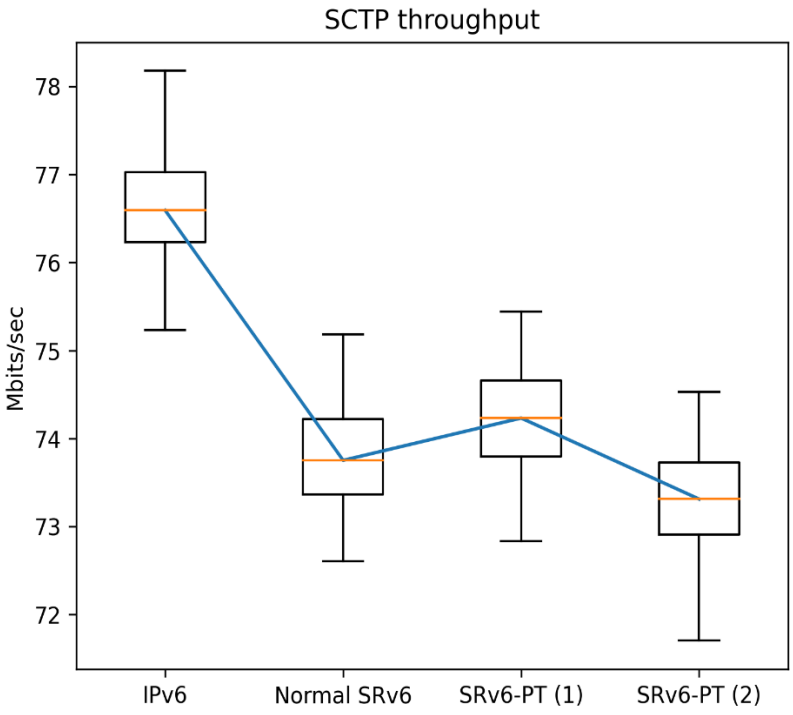
6.4 ábra: Az UDP csomagvesztés méréseinek diagramja

	Min	Max	Átlag	Medián	Szórás
IPv6	0,013	0,27	0,053	0,049	0,028
Normal SRv6	0,1	0,45	0,2756	0,28	0,076
SRv6-PT (1)	0,14	0,62	0,276	0,28	0,067
SRv6-PT (2)	0,12	0,64	0,266	0,26	0,085

6.5. táblázat: Az UDP csomagvesztés méréseinek statisztikája (százalékban)

Az UDP csomagvesztés mérési paramétereinek megválasztása kissé ad-hoc volt, a csomagküldés átviteli sebességének megválasztása terén legalábbis. Itt először kisebb inkrementumokban emeltem a sebességet és figyeltem, hogy melyik esetnél lesz a csomagvesztés aránya konzisztensen kimutatható, viszont még nem túl nagy. Ezt 1 Gbps-nél értem el. Ismét elmondható itt is, hogy a normál IPv6 esetében stabilabb a csomagfeldolgozás. Ami érdekes lehet, hogy az SRv6-os esetek közül pont a kétszeri listacserénél volt tapasztalható kisebb csomagvesztés. Tekintettel arra, hogy ez a különbség sem tekinthető jelentősnek a többi SRv6-os esethez képest, illetve a szórása

még nagyobb is, elképzelhető, hogy nagyobb számú mérés esetén ez a differencia tovább csökkenhetett volna.



6.5 ábra: Az SCTP adatátviteli sebesség méréseinek diagramja

	Min	Max	Átlag	Medián	Szórás
IPv6	75,2	79,1	76,64	76,6	0,64
Normal SRv6	72,61	75,2	73,78	73,75	0,58
SRv6-PT (1)	72,83	76,6	74,26	74,232	0,62
SRv6-PT (2)	71,45	75,26	73,29	73,31	0,64

6.6 táblázat: Az SCTP adatátviteli sebesség mérések statisztikája (Mbit/sec)

Az SCTP esetében is tapasztalható volt a TCP méréseknél látott karakterisztika, azonban itt kicsit élesebb volt a különbség az egyes esetek között. Míg a TCP esetében gigabites nagyságrendben lehetett beszélni a átviteli teljesítményről, addig az SCTP esetében ez két nagyságrenddel kisebbnek bizonyult. Ez főként amiatt van, mert az SCTP nem ugyanazt a célt szolgálja, mint a TCP, és a torlódáskezelési (congestion control) algoritmusa is más módon van implementálva. Ami számunkra most a lényeges, hogy más protokoll esetében is ugyanazt az sebességbeli megoszlást tapasztalhattuk, mely még tovább erősíti a méréseink konzisztenciáját.

Az eredményekről a következőket lehet kollektíven megállapítani. Jól látható, hogy a négy mérési szcenárió közül minden esetben a sima IPv6 alapú csomagtovábbítás teljesített a legjobban. Ez várható volt abban az értelemben, hogy abban az esetben a csomagoknak nem kell további feldolgozási lépéseken átmenniük, nem úgy, mint az SRv6-os mérések esetében. A sima IPv6 alapú mérést tekinthetjük a viszonyítási alapnak is. Ami viszonylag érdekes, hogy a kernel modult használó esetekben a teljesítmény azonos, vagy akár jobb is volt, mint a teljes szegmens listát tartalmazó SRv6-os esetben. Ezt a jelenséget jelenleg azzal tudom magyarázni, hogy mivel a kernel modulban a teljes SRv6 feldolgozást is elvégeztem a csomagokon (a listacserén kívül átírtam az IPv6-os célcímet), így azokat még egyszer nem kellett átküldenie a kernelbe épített SRv6 feldolgozáson, ezáltal minimális késleltetést megspórolva. Közrejátszhatott továbbá az is, hogy a kernel modul implementáció statikus konfiguráció alapján végezte a szegmens lista cseréket. Azonban ez változni fog egy teljes körű implementáció esetén, ahol dinamikus konfigurációk alapján fog dönteni a kernel modul arról, hogy egy adott szegmens lista tartalmát mire cserélje ki. Ez pedig nagy valószínűséggel meg fogja dobni a jelenlegi késleltetést. Ugyanakkor kifizetődőnek bizonyult, hogy a kernel modul implementációt kétféle szegmens lista méretezéssel is lemértem, ugyanis szépen kirajzolódott amire számítottam. A többszöri szegmens lista cserének köszönhetően minimális növekedés tapasztalható a késleltetésben, illetve az átviteli képességekben is.

Összességében elmondható, hogy a mérések során kapott eredmények beváltották a feléjük támasztott elképzeléseimet. Elmondható továbbá, hogy bár van minimális különbség az SRv6-ot használó esetekben, azok nem nevezhetőek statisztikailag szignifikáns különbségeknek. Emiatt aztán nyugodt szívvel állíthatom, hogy az általam javasolt SRv6-PT eljárás esetében nincs jelentősebb teljesítmény degradáció, alkalmazásának minimális a műszaki költsége, mellyel viszont az SL állapottér robbanását meg lehet előzni. Ez azt sugallja, hogy érdemes lehet a továbbiakban is fejleszteni és tesztelni az eljárást, a korábban meghatározott követelmények teljesítése mellett.

6.3 Felhasználási javaslatok és fejlődési lehetőségek

Ahogy a mérésekből kiderült, az általam javasolt SRv6-PT eljárás egy működőképes koncepciónak bizonyult. Az implementáció sikerességét több különböző hálózati protokollal történő alkalmazhatósága is bizonyítja. Azonban, mint azt a dolgozat elején is már jeleztem, és a jelenlegi implementáció korlátjaiból is adódik, ez még nem számít egy teljeskörű megoldásnak, a modul jelenleg még statikus kódrészeletei miatt a megoldás egyelőre még nem alkalmazható tetszőleges hálózati topológia esetében. Azonban mivel jelen dolgozattal sikerült verifikálni a koncepció helyességét és a meglévő SRv6 implementációval közel azonos teljesítményt tudott produkálni, ez lényegében zöld utat jelent számomra, hogy a későbbiekben egy olyan modult tudjak kifejleszteni, melyet tetszőleges hálózati topológián is lehet majd alkalmazni.

Ahhoz, hogy ezt meg lehessen valósítani, még rengeteg munkára lesz szükség a későbbiekben. A következő lépés ennek irányában a dinamikus linkállapot routing protokollokkal történő együttműködés vizsgálata lesz, mely esetben arra az implementációs kérdésre is keresem a választ, hogy kernel programként hogyan lehetne egy user-space-ben futó routing deamon adatbázisához hozzáférni. Ezt a specifikus problémát az adja, hogy ahhoz, hogy az egyes SRv6-PT végpontok is ismerjék a teljes hálózati topológiát, szükség van a linkállapot protokollok által felépített adatbázisra is, melyet azonban közvetlenül nem tesznek elérhetővé, és csak a topológia adottságai mentén kiválasztott útvonal bejegyzéseket teszik közzé. A kérdés tehát jobban leszűkítve az, hogy mi alapján fog az SRv6-PT globális topológia információkhoz jutni.

Mindazonáltal, ha ezeket a kérdéseket sikerül tisztázni, vagy akár egy kisebb architekturális változtatás mentén sikerül egy teljesértékű implementációt létrehozni, az nagy előrelépést jelenthet a hatékonyabban skálázódó szegmens alapú útválasztás irányában.

7 Összefoglalás

Ahogy azt már korábban is jeleztem, a dolgozat megírásának alapvető motivációját az jelentette, hogy megvizsgáljam, az SRv6 alapú útválasztást érintő skálázódási problémák kezelhetőek-e valamilyen alternatív módon is. A meglévő microSID implementáció ezt úgy kezeli, hogy a 128 bites SID-ekbe több instrukció kerül kódolásra. Az általam javasolt ötlet szerint nem a SID-ek struktúrájának változtatásával, hanem a szegmens lista méretének maximálásával próbálnánk a csomagméret növekedéssel járó skálázhatósági problémát kezelni. Ennek a megvalósítására terveztem meg az SRv6-PT megoldást, mely bár egyelőre még fejlesztési fázisban van, egy új SRv6 feldolgozási módszer implementálásával akár a microSID működés kihívója is lehet. Hogy koncepció szintjén bizonyítani tudjam az eljárásom működésének megvalósíthatóságát és fel tudjam deríteni a teljesítménybeli korlátjait, egy Linux kernel modul formájában implementáltam a működést. A modul hálózati teljesítményének felderítéséhez összehasonlítottam azt klasszikus IPv6 és normál SRv6 alapú csomagtovábbítással is, és bár az IPv6-hoz képest elmaradást lehetett tapasztalni, a normál SRv6 működéssel szinte megegyező, esetenként hajszállal jobb paramétereket is produkált. Ez bizonyítja, hogy a tervezett végleges implementáció is jó eséllyel összemérhető lesz vele teljesítményben, mely lehetővé tenné, hogy meglévő SRv6 hálózatokban is jelentős teljesítménycsökkenés nélkül lehessen alkalmazni az eljárásomat.

Irodalomjegyzék

[1] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, és Z. Li, „Segment Routing over IPv6 (SRv6) Network Programming”, sz. 8986. in Request for Comments. RFC Editor, 2021. [Online]. Elérhető: https://www.rfc-editor.org/info/rfc8986

[2] A. Bashandy, C. Filsfils, S. Previdi, B. Decraene, S. Litkowski, és R. Shakir, „Segment Routing with the MPLS Data Plane”, sz. 8660. in Request for Comments. RFC Editor, 2019. [Online]. Elérhető: https://www.rfc-editor.org/info/rfc8660

[3] C. Filsfils *és mtsai.*, „Network Programming extension: SRv6 uSID instruction”, Internet Engineering Task Force, Internet-Draft draft-filsfils-spring-net-pgm-extension-srv6-usid-15, 2023. [Online]. Elérhető: https://datatracker.ietf.org/doc/draft-filsfils-spring-net-pgm-extension-srv6-usid/15/

[4] Li, Z., Hu, Z., és Li, C., *SRv6 Network Programming: Ushering in a New Era of IP Networks*, 1. kiad. CRC Press, 2021. [Online]. Elérhető: https://doi.org/10.1201/9781003179399

[5] „Telcos and Data Center operators - Cilium”. Elérés: 2023. október 25. [Online]. Elérhető: https://cilium.io/industries/telcos-datacenters/

[6] S. Matsushima, C. Filsfils, M. Kohno, P. Camarillo, D. Voyer, és C. E. Perkins, „Segment Routing IPv6 for Mobile User Plane”, Internet Engineering Task Force, Internet-Draft draft-ietf-dmm-srv6-mobile-uplane-21, 2022. [Online]. Elérhető: https://datatracker.ietf.org/doc/draft-ietf-dmm-srv6-mobile-uplane/21/

[7] „SRv6 for Network Slicing - SRv6-based-Network-Slicing_MPLS-Congress-2019.pdf”. Elérés: 2023. október 30. [Online]. Elérhető: https://www.ipv6plus.net/resources/VTN/SRv6-based-Network-Slicing_MPLS-Congress-2019.pdf

[8] C. Lee, K. Ebisawa, H. Kuwata, M. Kohno, és S. Matsushima, „Performance Evaluation of GTP-U and SRv6 Stateless Translation”, in *2019 15th International Conference on Network and Service Management (CNSM)*, 2019, o. 1–6. doi: 10.23919/CNSM46954.2019.9012725.

[9] M. Gramaglia, V. Sciancalepore, F. J. Fernandez-Maestro, R. Perez, P. Serrano, és A. Banchs, „Experimenting with SRv6: a Tunneling Protocol supporting Network Slicing in 5G and beyond”, in *2020 IEEE 25th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, 2020, o. 1–6. doi: 10.1109/CAMAD50429.2020.9209260.

[10] „What is SRv6? How does SRv6 work? - Huawei”. Elérés: 2023. október 30. [Online]. Elérhető: https://info.support.huawei.com/info-finder/encyclopedia/en/SRv6.html

[11] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, és R. Shakir, „Segment Routing Architecture”, sz. 8402. in Request for Comments. RFC Editor, 2018. [Online]. Elérhető: https://www.rfc-editor.org/info/rfc8402

[12] „Understanding Segment Routing IPv6 - New IP technologies - Huawei”. [Online]. Elérhető: https://support.huawei.com/enterprise/tr/doc/EDOC1000173015/d169625f/understanding-segment-routing-ipv6

[13] „Segment Routing IPv6 Configuration - NetEngine 8100 M14, M8, 8000E M14, M8, 8000 M14K, M14, M8K, M8 V800R021C10SPC600 Configuration Guide - Huawei”. Elérés: 2023. október 19. [Online]. Elérhető:

https://support.huawei.com/enterprise/en/doc/EDOC1100269507/f1ef5696/segment-routing-ipv6-configuration
[14] „Segment Routing and PCE User Guide - Nokia”, Nokia Help. Elérés: 2023. október 19. [Online]. Elérhető: https://infocenter.nokia.com/public/7750SR217R1A/index.jsp?topic=%2Fcom.nokia.Segment_Routing_and_PCE_User_Guide_21.7.R1%2FFeature_configuration.html
[15] „SRv6 - Linux Kernel implementation”. Elérés: 2023. október 19. [Online]. Elérhető: https://segment-routing.org/
[16] Z. Li, Z. Hu, K. Talaulikar, és P. Psenak, „OSPFv3 Extensions for SRv6”, Internet Engineering Task Force, Internet-Draft draft-ietf-lsr-ospfv3-srv6-extensions-12, 2023. [Online]. Elérhető: https://datatracker.ietf.org/doc/draft-ietf-lsr-ospfv3-srv6-extensions/12/
[17] C. Filsfils, D. Dukes, S. Previdi, J. Leddy, S. Matsushima, és D. Voyer, „IPv6 Segment Routing Header (SRH)”, sz. 8754. in Request for Comments. RFC Editor, 2020. [Online]. Elérhető: https://www.rfc-editor.org/info/rfc8754
[18] „The Linux Kernel Module Programming Guide - lkpmg.pdf”. Elérés: 2023. október 26. [Online]. Elérhető: https://tldp.org/LDP/lkmpg/2.6/lkmpg.pdf
[19] „SREXT”. Elérés: 2023. október 26. [Online]. Elérhető: https://netgroup.github.io/SRv6-net-prog/
[20] „netfilter/iptables project homepage - The netfilter.org project”. Elérés: 2023. október 26. [Online]. Elérhető: https://www.netfilter.org/
[21] „SCTP Use Cases”. Elérés: 2023. október 30. [Online]. Elérhető: https://docs.paloaltonetworks.com/service-providers/10-1/mobile-network-infrastructure-getting-started/sctp/sctp-use-cases

Ábra- és táblázatjegyzék

Rövidítések jegyzéke

3GPP – 3rd Generation Partnership Program

CLI – Command Line Interface

eMBB – enhanced Mobile BroadBand

GTP – GPRS Tunneling Protocol

IANA – Internet Assigned Numbers Authority

IETF – Internet Engineering Task Force

IPv6 – Internet Protocol version 6

MPLS – Multi Protocol Label Switching

OSPF – Open Shortest Path First

PT – Policy Translation

RFC – Request for Comments

RTT – Round-Trip Time

SCTP – Stream Control Transmission Protocol

SID – Segment IDentifier

SL – Segment List

SR – Segment Routing

SREXT – Segment Routing EXTension module

SRH – Segment Routing Header

SRv6 – Segment Routing over IPv6

SRv6-PT – Segment Routing over IPv6 with Policy Translation

TCP – Transmission Control Protocol

UDP – User Datagram Protocol

VCPU – Virtual Central Processing Unit

VM – Virtual Machine

Függelék

Függelék A

A következőkben a 6. fejezetben bemutatott mérési környezetre vonatkozó konfiguráció beállításokat mutatom be részletesen, az egyes virtuális gépekre levetítve. A könnyebb vizualizálás érdekében a 6.1 ábrát érdemes újra átvizsgálni.

A mérési környezet létrehozásához és konfigurálásához a Terraform nevű ’Infrastacture as Code’ eszközt használtam, és az egyes konfigokat, mint cloud-init szkripteket adtam át az OpenStack számára végrehajtásnak. Az alább részletezésre kerülő bash szkriptek egy része így Terraform specifikus elemeket is tartalmazhat, pontosabban Hashicorp Configuration Language nyelvi elmeket.

Az egyes VM-ek interfészeinek nevei, melyek a lácntopológia részét képezték, mindegyik esetben ’ens4’ és ’ens5’ voltak. Előbbi a láncban visszafele, utóbbi a láncban előbbre levő szomszédjával volt összekötve. Az első lépés tehát a ’forwarding változók 1-be billentése volt, majd a statikus útvonalbejegyzések megadása, és az S1 és S10 esetében az iperf3 letöltése következett. Az alábbi kivágás az S1 konfigjából való.

```
sudo sysctl -w net.ipv6.conf.all.forwarding=1
sudo sysctl -w net.ipv6.conf.ens4.forwarding=1

sudo ip -6 route add f009::/64 via f001::20 dev ens4
sudo apt update && sudo apt install -y iperf3
```

Az S10 esetében is nagyon hasonló volt a konfig, ott a route bejegyzés a következőképpen nézett ki: `sudo ip -6 route add f001::/64 via f009::10 dev ens4`

A köztes csomópontok konfigurációja kissé komplikáltabb volt már, ugyanis ezek már az SRv6 domain részét képezték. Először a szükséges sysctl változókat kellett engedélyezni:

```
#---------------sysctl enables----------------
sudo sysctl -w net.ipv6.conf.all.seg6_enabled=1
sudo sysctl -w net.ipv6.conf.ens5.seg6_enabled=1
sudo sysctl -w net.ipv6.conf.ens4.seg6_enabled=1
sudo sysctl -w net.ipv6.conf.all.forwarding=1
sudo sysctl -w net.ipv6.conf.ens5.forwarding=1
sudo sysctl -w net.ipv6.conf.ens4.forwarding=1
```

Következőnek a statikus route bejegyzéseket kell megadni. Ezek azt adják meg, hogy egy adott SID lokátora mely hosztot azonosítja. Pl. az S5 hoszt esetében annak SID lokátora

5001::/64, a szomszédjainak, S4-nek és S6-nak rendre 4001::/64 és 6001::/64. Az S4-et az f004::10, az S6-ot az f005::20 címeken lehetett elérni.

```
#-------------SRv6 seg6 encap route table input---------------
sudo ip -6 route add ${count.index+1}001::/64 via
f00${count.index+1}::10 dev ens4
sudo ip -6 route add ${count.index+3}001::/64 via
f00${count.index+2}::20 dev ens5
```

A seg6local féle SRv6 feldolgozást az alábbi parancsokkal lehetett megtenni. Létrehoztunk egy új route táblát az iproute2 számára, és definiáltuk, hogy az adott hosztnak szánt lokátorú szegmenseket ezen tábla bejegyzései szerint dolgozza fel.

```
#localsid
echo 100 localsid >> /etc/iproute2/rt_tables
sudo ip -6 rule add to ${count.index+2}001::/64 lookup localsid
sudo ip -6 route add blackhole default table localsid
```

Az SRv6 domain két szélső elemét kivéve az End viselkedést valósítsák meg,

```
#----------------------
${count.index == 0 || count.index == 7 ? "" : "sudo ip -6 route add
${count.index+2}001::5e6/64 encap seg6local action End dev ens4
table localsid"}
```

Majd a két szélső elem beállításai a következők voltak: S9 esetében a 9001::/64-es SID, S2 esetében a 2001::64-es SID-re az End.DT6-os műveletet adtuk meg, azaz ha ezekre a SID-ekre van illeszkedés, a külső IPv6 és SRv6 fejlécek dekapszulálódnak és a belső IPv6-os fejléc mentén, a main tábla bejegyzése szerint történik a csomagtovábbítás a két, nem SRv6 domain tag hálózati elem, az S1 és S10 számára. Illetve az `ip -6 sr tunsrc` paranccsal beállításra kerül az is, hogy az enkapszulálandó csomagok esetén mi szerepeljen a külső IPv6 fejléc forráscímeként.

```
${count.index == 0 ? "sudo ip -6 route add 2001::5e6/64 encap
seg6local action End.DT6 table main dev ens4 table localsid" : ""}

${count.index == 0 ? "sudo ip -6 route add f009::/64 encap seg6 mode
encap segs 3001::5e6,4001::5e6,5001::5e6,9001::5e6 dev ens5" : ""}

${count.index == 0 ? "sudo ip sr tunsrc set 2001::5e6" : ""}

${count.index == 7 ? "sudo ip -6 route add 9001::5e6/64 encap
seg6local action End.DT6 table main dev ens4 table localsid" : ""}

${count.index == 7 ? "sudo ip -6 route add f001::/64 encap seg6 mode
encap segs 8001::5e6,7001::5e6,6001::5e6,2001::5e6 dev ens4" : ""}

${count.index == 7 ? "sudo ip sr tunsrc set 9001::5e6" : ""}
```

Ezen a ponton az egyes közbülső hosztok készenálltak az SRv6 feldolgozásra, azonban még egy dolog hiányzott: nem került megadásra, hogy az S2 és S9 SRv6 domainbe bemenő forgalma esetén milyen szegmens listával kerüljön enkapszulálásra a csomag. Mivel ezen route bejegyzéseknek a módosítására az egyes mérési forgatókönyvek esetén gyakran sor került, ezért erre a feladatra egy külön bash szkriptet készítettem, mellyel a sok, redundáns parancs kiadását meglehetősen le lehetett rövidíteni. A szkript két argumentumot kap, a hoszt nevét (ez ’s2’ vagy ’s9’ lehet), illetve az adott mérési forgatókönyv módját (’nomod’, ’mod’ vagy ’mod3’).

```
#!/bin/bash

MODE=$2
HOST=$1

case "$HOST" in
s2)
     case "$MODE" in
     mod)
          sudo ip -6 sr tunsrc set 2001::5e6
          sudo ip -6 route del f009::/64 encap seg6 mode encap segs
3001::5e6,4001::5e6,5001::5e6,6001::5e6,7001::5e6,8001::5e6,9001::5e
6 dev ens5
          sudo ip -6 route del f009::/64 encap seg6 mode encap segs
3001::5e6,4001::5e6,9001::5e6 dev ens5
          sudo ip -6 route add f009::/64 encap seg6 mode encap segs
3001::5e6,4001::5e6,5001::5e6,9001::5e6 dev ens5
          ;;
     mod3)
          sudo ip -6 sr tunsrc set 2001::5e6
          sudo ip -6 route del f009::/64 encap seg6 mode encap segs
3001::5e6,4001::5e6,5001::5e6,6001::5e6,7001::5e6,8001::5e6,9001::5e
6 dev ens5
          sudo ip -6 route del f009::/64 encap seg6 mode encap segs
3001::5e6,4001::5e6,5001::5e6,9001::5e6 dev ens5
          sudo ip -6 route add f009::/64 encap seg6 mode encap segs
3001::5e6,4001::5e6,9001::5e6 dev ens5
          ;;
     nomod)
          sudo ip -6 sr tunsrc set 2001::5e6
          sudo ip -6 route del f009::/64 encap seg6 mode encap segs
3001::5e6,4001::5e6,5001::5e6,9001::5e6 dev ens5
          sudo ip -6 route del f009::/64 encap seg6 mode encap segs
3001::5e6,4001::5e6,9001::5e6 dev ens5
          sudo ip -6 route add f009::/64 encap seg6 mode encap segs
3001::5e6,4001::5e6,5001::5e6,6001::5e6,7001::5e6,8001::5e6,9001::5e
6 dev ens5
          ;;
     *)
          echo "Incorrect input! Try: ./envinit.sh <node> <mode>"
          ;;
     esac
     ;;
s9)
     case "$MODE" in
     mod)
```

```
            sudo ip -6 sr tunsrc set 9001::5e6
            sudo ip -6 route del f001::/64 encap seg6 mode encap segs
8001::5e6,7001::5e6,6001::5e6,5001::5e6,4001::5e6,3001::5e6,2001::5e
6 dev ens4
            sudo ip -6 route del f001::/64 encap seg6 mode encap segs
8001::5e6,7001::5e6,2001::5e6 dev ens4
            sudo ip -6 route add f001::/64 encap seg6 mode encap segs
8001::5e6,7001::5e6,6001::5e6,2001::5e6 dev ens4
            ;;
        mod3)
            sudo ip -6 sr tunsrc set 9001::5e6
            sudo ip -6 route del f001::/64 encap seg6 mode encap segs
8001::5e6,7001::5e6,6001::5e6,5001::5e6,4001::5e6,3001::5e6,2001::5e
6 dev ens4
            sudo ip -6 route del f001::/64 encap seg6 mode encap segs
8001::5e6,7001::5e6,6001::5e6,2001::5e6 dev ens4
            sudo ip -6 route add f001::/64 encap seg6 mode encap segs
8001::5e6,7001::5e6,2001::5e6 dev ens4
            ;;
        nomod)
            sudo ip -6 sr tunsrc set 9001::5e6
            sudo ip -6 route del f001::/64 encap seg6 mode encap segs
8001::5e6,7001::5e6,6001::5e6,2001::5e6 dev ens4
            sudo ip -6 route del f001::/64 encap seg6 mode encap segs
8001::5e6,7001::5e6,2001::5e6 dev ens4
            sudo ip -6 route add f001::/64 encap seg6 mode encap segs
8001::5e6,7001::5e6,6001::5e6,5001::5e6,4001::5e6,3001::5e6,2001::5e
6 dev ens4
            ;;
        *)
            echo "Incorrect input! Try: ./envinit.sh <node> <mode>"
            ;;
        esac
        ;;
*)
        echo "Incorrect input! Try: ./envinit.sh <node> <mode>"
        ;;
esac
```

Ha például az SRv6 konfigurációból a nomod módot szeretnénk használni, akkor a következő parancsokat kell csak kiadni: S2-n '`./envinit.sh s2 nomod`', S9-en pedig '`./envinit.sh s9 nomod`'.

Függelék B

Ebben a részben a mérési forgatókönyvekhez használt szkripteket mutatom be részletesen.

Az ICMPv6 echo request/reply RTT méréséhez egész egyszerűen a beépített ping parancsot használtam, melynek a kimenetét egy-egy szöveges fájlba irányítottam, és azokból nyertem ki az RTT értékeket. Egy parancs a következőképpen nézett ki:

```
ping -c 10000 -i 0.5 -s 8000 f009::20 >> srv6_nomod_ping.txt
```

A pinggel ilyenkor 10000 db Echo requestet küldünk ki, melyre várjuk az Echo reply-t, az üzenetek küldésének intervalluma fél másodperc, és a kiküldendő csomag mérete 8000 bájt legyen. Azért állítottam a mérésekkor ilyen arbitrálisan nagyra a csomagméretet, hogy a nagyobb csomagmérettel járó terheléssel jobban kirajzolódjanak az egyes mérési forgatókönyvek közötti különbségek.

A TCP, UDP és SCTP mérései esetében a következő volt a helyzet: mivel a használt iperf3 eszköz nem támogatja, hogy egymás után több mérési folyamot is lehessen indítani, ezért ennek a megvalósítására egy burkoló (wrapper) szkriptet hoztam létre. Ezen szkriptek nevei perf.sh, perf_udp.sh és perf_sctp.sh voltak. A perf.sh felépítése a következő volt:

```
#!/bin/sh

for i in `seq 1 $3`
do
  if [ "$#" -eq 0 ]
  then
    echo "params"
    echo "perf.sh <ServerIP> <Server port> <num of iterations> <file
output name>"
    break
  fi

  iperf3 -c "$1" -p "$2" -t 60 -V -f m -i 60 >> "$4".txt
done
```

Egy for ciklusban iterált végig annyiszor, ahány mérést paraméterként adtam meg, és futtatta az iperf3 parancsot. A szkript paraméterezése a következő, az egyes helykitöltők leírása magáért beszél:

```
perf.sh <ServerIP> <Server port> <num of iterations> <file output name>
```

Egy futtatás például a következőképp nézett ki:

```
./perf.sh f009::20 5201 100 srv6_nomod_tcp
```

Az UDP és SCTP mérési szkriptek szinte ugyanígy néztek ki, annyi különbséggel, hogy a beágyazott iperf3 parancs hogy nézett ki. A protokolloknak megfelelő specifikus kapcsolókat félkövérrel jeleztem.

UDP: `iperf3 -c "$1" -p "$2" -V -f K -t 60 -i 60` **`-u -b 1G`** `>> "$4".txt`

SCTP: `iperf3 -c "$1" -p "$2" -t 60 -V -f K -i 60` **`--sctp`** `>> "$4".txt`