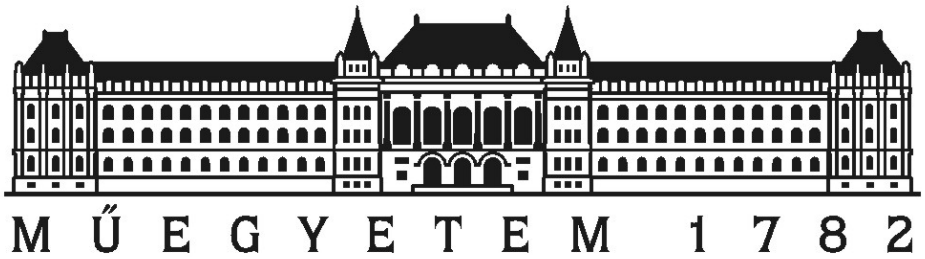


Hatékony link-hiba detekciós megoldások vizsgálata és implementálása alsó kategóriás hálózati útválasztókon

Készítette:	**Lévai Tamás**
Neptun-kód:	**BUIHKR**
E-mail cím:	**levai.tamas@t-online.hu**
Konzulens:	**Németh Felicián**
E-mail cím:	**nemethf@tmit.bme.hu**

Budapest, 2013.

Tartalomjegyzék

1. Bevezetés

A távközlés kialakulásának kezdete óta komoly problémát jelent a közvetítő közeg megváltozásából adódó információvesztés. Ennek speciális változata, ha a fizikai közvetítő közeg, azaz információ továbbításra használt csatorna alappillére szűnik meg. Ezt a jelenséget a távközlés jellegéből adódóan nem lehet elkerülni. Emiatt, függetlenül az átviteli közegtől, minden távközlési rendszernek számolnia kell ezzel a problémával. Ezért, akárcsak a régi távíró rendszerekben, úgy a mai számítógép-hálózatokban is különböző megoldások léteznek e probléma kezelésére, azaz a közvetítő közeg meglétének ellenőrzésére. Napjaink számítógéphálózataira jellemző továbbá az olyan eszközök elterjedése, melyek az alacsony árukhoz képest komoly teljesítménnyel rendelkeznek, ugyanakkor alsó kategóriás, otthoni felhasználásra szánt hálózati útvonalválasztók. Mára ezek az eszközök rendelkeznek akkora teljesítménnyel, hogy a tervezett otthoni felhasználáson túl akár különböző kutatások, fejlesztések prototípusaként is helyt állnak, bizonyos esetekben kiváltva a drágább teszteszközöket. Viszont ezen alsó kategóriás útvonalválasztók ilyen célú felhasználáshoz gyenge link-hiba detekciós képességekkel rendelkeznek, ugyanis a linkállapot változásokat az eszközben levő hálózati kapcsoló jelzés nélkül kezeli. Ezáltal megnehezítik például az egyedi routing szolgáltatások alkalmazását. Dolgozatomban a létező link-hiba detekciós megoldások ismertetésén túl bemutatok egy általam készített, ezen eszközöket megcélzó, azok hiányosságait áthidaló, az eddig alkalmazottak megoldásoknál kisebb erőforrásigényű megoldást. Hatékonyságát egyszerű OpenFlow teszthálózaton végzett méréssel igazolom, így az eljárás gyakorlati környezetben nyújtott teljesítménye kiértékelhető.

2. A linkvesztés

2.1. A linkvesztés folyamata

Mint ahogy azt a bevezetőben már említettem, a távközlési rendszereknek és így a számítógép-hálózatoknak is alappillére a közvetítő közeg. Éppen ezért szinte mindegyik hálózati modellnek, így a jól ismert hét rétegű OSI modellnek is a legalsó, úgynevezett fizikai rétege a közvetítő közeget írja le. Ennek a rétegnek a feladata a felette levő rétegekből érkező, bitekkel reprezentált adatok szimbólumokká alakítása, majd ezen szimbólumok a fizikailag is létező kommunikációs csatornára történő helyezése, vagy épp ennek fordítottja. Tehát ezen a szinten történik az eszközök fizikai leírása. Például az eszközök elektromos specifikációja, érintkezők méretei, frekvenciasávok vezeték nélküli kapcsolathoz és kábelspecifikációk a vezetékes kapcsolathoz. Szervesen kapcsolódik ehhez a réteghez a rá közvetlenül épülő adatkapcsolati réteg, amelynek feladata a fizikai réteg elfedése, azaz kezelni végpontok közötti összeköttetést, észlelni és lehetőségekhez mérten javítani a fizikai rétegben keletkező hibákat. Itt jelenik meg a *link* fogalma, ami két egymással szomszédos végpont összeköttetését jelenti.

Amikor link-hiba áll fenn, akkor ennek az összeköttetésnek a hibájáról beszélünk. Ez az összeköttetés jellemzően két végberendezésből és az őket összekötő közvetítő kötegből áll. A közvetítő közeg jellemzően rádiócsatorna, optikai kábel, vezeték. Mivel mindhárom közvetítő közeg jelentősen eltér működési elvében, karakterisztikájában, így érdemes megvizsgálni, hogy az egyes közegekben milyen események okozhatnak link-hibát és mi történik ilyenkor.

Rádiócsatorna esetén a link minőségét befolyásoló tényezők a környezetében jelentkező elektromágneses zajok és zavarok, illetve a végpontok egymás közti rálátása. A linkvesztés jellemzően a rálátás megszűnése miatt történik, azaz a két végpont egymás hatóterén kívülre kerül. Ugyanakkor nagy mennyiségű zaj is ellehetetlenítheti a kapcsolat kiépülését.

Az optikai kábel működése a fény terjedésének törvényszerűségeire épül. A küldő oldalon kibocsátott fénynyaláb a kábel belsejében, vagy annak faláról visszaverődve vagy visszaverődés nélkül, ugyanis egymódusú szál esetén a kábel úgy van kialakítva, hogy az hullámvezetőként viselkedjen, jut el a fogadó oldalra, ahol egy szenzor érzékeli. Ilyenkor akkor beszélünk linkvesztésről, ha a nyaláb nem ért célba. Ennek oka lehet a visszaverődések tökéletlensége, azaz a nyaláb nem tökéletesen verődik vissza, egy része elnyelődik. Emiatt minden visszaverődéssel veszít energiájából. A nyaláb eltérülése, illetve kioltódása adódhat a kábelt érő külső hatásoktól is, például erős fizikai hatástól megsérülhet és működésképtelenné válhat. Természetesen a kábel lecsatlakoztatása is linkvesztéssel jár.

Fémvezetékekben kettő vagy több fém ér fut, köztük dielektrikum. Az erekre kódolásnak megfelelően engednek feszültséget. Ezt az átviteli sebesség függvé-

nyében váltogatják és mivel a változó elektromos áram mágneses mezőt generál, ami viszont elektromos áramot indukál, és így tovább. Tehát az erek egymásra és a környezetükre is hatással vannak. Ezt a hatást az erek egymásra csavarásával hatékonyan lehet csökkenteni, ennek részletezésétől eltekintenék. A linkvesztés jellemzően külső fizikai behatástól történik, adódhat a kábel sérüléséből vagy lecsatlakoztatásából. Ilyenkor, mivel megszűnik az ér folytonossága, a két végpont különböző feszültségértéket fog mérni.

2.2. A linkvesztés detektálása

Sajnos egyik esetben sem egyértelmű a linkvesztés detektálása. Ugyanis míg a tényleges szakadásról értesülnek a hálózati modell fentebbi rétegei, addig rengeteg, az alsóbb rétegben eldobandó csomagot állítanak elő. Nyilván minél nagyobb a link adatátviteli sebessége – ami persze a technológiai fejlődés hatására folyamatosan nő –, annál több csomag megy veszendőbe egy link-hiba miatt azért, mert a csomag fizikai közegre történő helyezése annak hibája miatt lehetetlen. Éppen ezért kiemelten fontos a linkvesztés mihamarabbi észlelése.

Különböző ötletekre alapozott megoldások születtek e problémára. Legoptimálisabb persze az, ha már tervezési időben gondoltak erre és az átviteli közeghez hatékony link-hiba detekciós képességet társítottak. Ilyen tulajdonsággal rendelkeznek például a SONET [1], illetve SDH [2] hálózatok. Ezek szinkron optikai hálózatok, ami azt jelenti, a végpontok LED-jei által adott órajel alapján periodikusan kibocsátott fénysugár utazik az optikai kábelen. Működési elvéből adódóan következik, hogy könnyen detektálható, ha megszűnik a link. Egy linkvesztést viszonylag gyorsan, jellemzően 50 milliszekundum alatt fel tud dolgozni a rendszer [1]. Ennek hatékonysága tovább növelhető az úgynevezett monitoring trail (M-trail) [2, 3] alkalmazásával. Ugyanis a transzparens optikai hálózatokban a link-hiba pontos helyének detektálása nehézkes, mivel optoelektronikai jelfrissítők hiányában nehéz elektronikusan behatárolni a problémás link(ek)et. Hullámhossz-osztású multiplexálás esetén egy optikai kábelen több link található, így kábelszakadás esetén a ráépülő linkek mind megszakadnak. Így link-hiba jelzések sokasága jön létre. Továbbá az alacsony, optikai szinten mozgó hibajelzés mellé társulnak a fentebbi hálózati rétegek által generált hibajelzések, például a routing szolgáltatások hibái is. Emiatt egy hiba több másikat idéz elő. Az alsóbb réteg hibái jellemzően hamarabb keletkeznek, így hamarabb feldolgozhatóak. Tehát ezen hibák gyors feldolgozásával visszaszoríthatóak a redundáns hibajelzések. Erre nyújt pontos és erőforrás hatékony megoldást a monitoring trail. Mivel az optikai hálózati berendezések ára magas, így az alsó kategóriás hálózati eszközökben

[1] SONET: Synchronous Optical Networking

[2] SDH: Synchronous Digital Hierarchy

nem alkalmazzák őket. Ezért az optikai linkek és az m-trail bővebb ismertetésétől eltekintenék.

Alsó kategóriás hálózati eszközökön jellemzően Ethernet protokollt használnak általában rézkábelen. Az Ethernet alkalmazását alacsony költségének és egyszerűségének köszönheti. Ennek viszont ára van, az Ethernet nem rendelkezik a SONET-hez és SDH-hoz hasonló, hatékony link-hiba detekciós képességekkel. Így ilyenkor más megközelítést kell alkalmazni.

Adódik, hogy a hálózati elemek tárolják és megosztják a szomszédaikról szerzett ismereteiket. Ebből a végpont következtetéseket tud levonni a linkjeivel kapcsolatban. Ezen elv mentén működik az Etherneten alkalmazható LLDP [3] protokoll [4]. Egy hálózatban levő, ezen protokollt használó eszközök fix időközönként meghirdetik főbb tulajdonságaikat. Ilyen tulajdonság például az eszköz azonosítója, képességei továbbá szomszédjai. Ezeket az információkat TLV (Typle-Length-Value) struktúrákban tárolja. A TLV-ket az LLDP Data Unit fogja össze. Az LLDP Data Unitokat Ethernet keretbe ágyazva küldi el az eszközök minden szomszédjuknak. Címzés során speciális multicast MAC címeket használ, melyek jelentése a legközelebbi IEEE802.1D kompatibilis hálózati híd megcímzése. A fogadott üzeneteket a hálózati csomópontok a saját menedzsment információs adatbázisukba (MIB) gyűjtik. Ebben megtalálható az eszköz neve, leírása, port információk, VLAN-ok nevei, MAC/PHY információk, stb. Link-hiba akkor történik, ha link váratlanul megszakad azelőtt, hogy a link másik végén levő eszköz jelezte volna a link lekapcsolását. Ilyenkor a helyi eszköz nem törli rögtön az adott linkről szóló bejegyzéseit, hanem az rxInfoTTL időzítő lejártáig vár a link helyreállására. Az LLDP-ben használt időzítők átlagosan egy másodpercenként ketyegnek viszonylag nagy, akár 0,4 másodperces csúszásokkal. A várakozásból adódóan több másodperc is eltelik, mire a link-hiba jelentése, majd feldolgozása megtörténik.

LLDP-hez hasonlóan használható az STP [4] protokoll, melynek elsődleges feladata egy körmentes topológia kialakítása. Mivel e feladat miatt szükséges számára a linkvesztések kezelése, ezért rendelkezik nem túl hatékony linkvesztés detekcióval. Nem túl hatékony alatt értendő, hogy körülbelül tíz másodperces nagyságrendű idő alatt képes a link-hibát detektálni. Némiképp fényesebb képet fest a továbbfejlesztése, az RSTP [5], mely a hagyományos STP-nél hamarabb tudja detektálni a link-hibát [1].

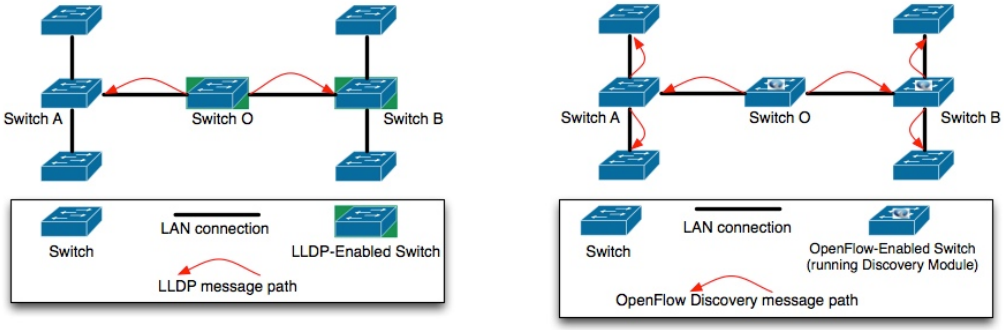
Az OpenFlow hálózatokon használt OFDP (OpenFlow Discovery Protocol) is az LLDP-re épül. Működését OpenFlow kontrollerben implementálják. Tanulmányozható például a discovery.py POX controlleren keresztül. Ez a mechanizmus lehetőséget nyújt a topológiában történő változások jelzésére is. Az 1. ábrán látha-

[3] LLDP: Link Layer Discovery Protocol

[4] STP: Spanning Tree Protocol

[5] RSTP: Rapid STP

tó, hogy az OFDP nagy hátulütője az, hogy míg az LLDP üzenetek csak a szomszédos LLDP csomópontig jutnak el, addig az OFDP csomagok a kontrollertől indulva az egész hálózatot bejárják [5]. A csomag küldésből adódó késleltetések miatt a link-hiba detektálás sok időbe telik.



1. ábra. Az LLDP és OFDP csomagok terjedése a hálózatban [5]

Összegezve az LLDP-re épülő megoldásokat, kijelenthető, hogy gyors link-hiba detekcióra alkalmatlanok a lassú időzítők, illetve az eltérő, topológiafeltáró szemlélet miatt.

Teljesen más szemléleten alapul az a megközelítés, melynek lényege, hogy a link végén elhelyezkedő végpontok egymásnak olyan speciális csomagokat küldenek, amelyeknek az a funkciójuk, hogy célba érésükkel igazolják a link működőképességét. Emiatt egyéb információt nem kell hordozniuk, mivel a célba éréshez a szükséges fejléceken túl nem kötelező mást tartalmaznia egy csomagnak, így ezen csomagok továbbítása minimális sávszélességet fog el a tényleges adattovábbításból. Funkciójukból adódóan ezen csomagokat gyakran *Hello* csomagoknak hívjuk. Ezen eljárás előnye, hogy bármilyen csomagkapcsolt hálózaton működhet, függetlenül az átviteli közegtől. Erre alapoznak például különböző routing protokollokban, lásd OSPF [6] hello packet. Viszont a routing protokollok jellemzően magasabb rétegekben működnek. Hatékonyabb lenne ezt az eljárást külön, pehelysúlyúbb protokollként alkalmazni.

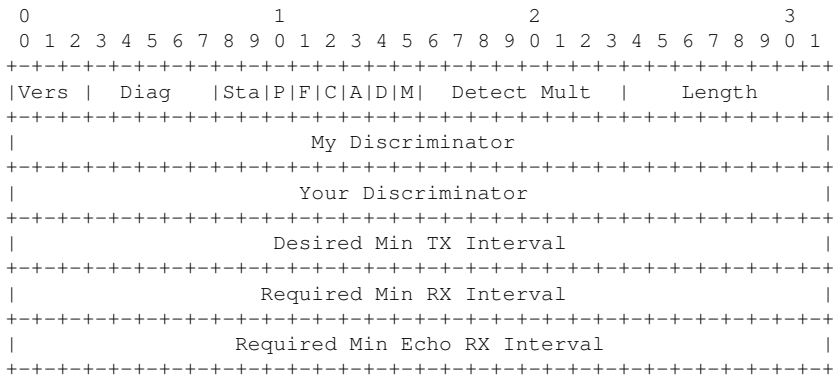
Ezen gondolat mentén jött létre a Bidiractional Forwarding Detection (BFD) [6, 7] is. E protokoll két végpont között épít fel linkenként egy-egy kapcsolatot három-utas kézfogással. Háromféle működési módot támogat. Első az aszinkron mód (Asynchronous mode), melynek során a két végpont periodikusan küld egymásnak BFD csomagokat és ha a kiküldött csomagok és a megérkező csomagok száma nem egyezik meg, akkor lezárja a munkamenetet, mert valószínűleg link-hiba történt.

[6]OSPF v2: Open Shortest Path First v2 (RFC 2328)

Második az igény-vezérelt mód (Demand mode). Ebben az esetben a BFD-n kívül is létezik egy link-hiba észlelő eljárás, amire a BFD támaszkodhat, így nem kell folyamatosan csomagokat küldeni. Elég csak akkor, ha úgy határoz az egyik fél, hogy külön igényli a kapcsolat meglétének ellenőrzését. Ennek a módnak nagy előnye, hogy jóval kevésbé terheli a linket.

Harmadik „mód" az előző kettő keveréke, az echó funkció (Echo function). Ilyenkor az egyik fél küld egy sornyi speciális, úgynevezett BFD echó csomagot, amiket a másik fél a csomagok megérkezése után változatlanul visszaküld a feladónak. Lehetséges a szerepek cseréje a két végpont között, így szabályozható a küldési irány. Amennyiben a visszaküldött sor mérete eltér, úgy azt a rendszer link-hibaként értelmezi.

Mielőtt mélyebben részletezném a három működési módot és a hozzákapcsolódó műveleteket, fontos, hogy bemutassam a BFD vezérlő csomagok felépítését. A 2. ábrán láthatóak a csomag kötelező mezői.

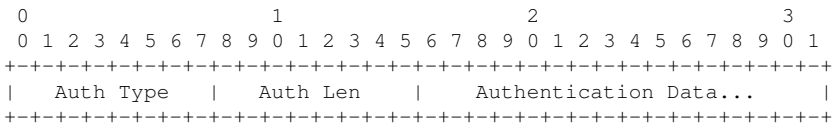


2. ábra. BFD Control Packet kötelező mezői [6]

Ezek közül az első mező célszerűen a csomag verziószámát jelöli. Ezt követi az ötbites Diag mező, amely a helyi rendszer legutóbbi munkamenet változtatásának okát jelöli. Például 0– nem történt változás, 2 – echó hiba, 5 – megszakadt az összeköttetés. A kétbites Sta mező a munkamenet állapotát írja le, 0 – adminisztratívan leállt, 1 – leállt, 2 – inicializál, 3 – működik. Bebillentett P (Poll) bittel lehet kérvényezni a a kapcsolat vagy munkamenet-állapotváltozás ellenőrzését. Erre egy bebillentett F (Final) bittel rendelkező csomaggal válaszol a másik fél. A C (Control Plane Independent) bit nevéből eredően a vezérlési síktól való függetlenedést hívatott jelölni. Az A (Authentication Present) bit jelöli, hogy a csomag tartalmaz autentikációs részt is. Ennek felépítését lásd a 3. ábrán.A D (Demand) bittel lehet aktiválni az igény-vezérelt módot. Az M (Multipoint) még nem használt, a pont-multipont összeköttetések támogatására fog szolgálni. A nyolcbites Detect Mult a meghatározott átviteli időtartam szorzója, továbbá aszinkron mód

esetén a vevő oldal észlelési idejét is jelenti. Azaz csak ezen az időtartamon belül beérkező csomagok érvényesek. A Length mező a BFD csomag hosszát adja meg. A 32 bites {My,Your} Discriminator mező egyedi, jellemzően nem nulla értékű azonosító, mely segítségével a BFD csomagok multiplexálhatók. A Your Discriminator mező – melynek szerepe a küldő visszakereshetősége – értéke lehet nulla abban az esetben, ha a küldő ismeretlen. Ellenkező esetben megegyezik a küldő My Discriminator értékével. Szintén 32 bites a Desired Min TX Interval mező, mely mikroszekundumokban adja meg a rendszer csomagküldésre szánt minimális időtartamát. Párja a Required Min RX Interval mező, mely szintén mikroszekundumokban adja meg a rendszer csomagfogadásra szánt minimális időtartamát. Továbbá a Required Min Echo RX Interval mező, mely az echó csomagok minimális fogadási időtartamát határozza meg mikroszekundumokban.

A kötelező mezőkön túl tartozhat autentikációs kiegészítés a BFD csomagokhoz. Ennek meglétét a kötelező mezők között az Authentication Present bit jelöli. A hitelesítési szekciót a 3. ábra szemlélteti. Látható, hogy háromféle mezőt tartalmaz. Ezek közül első a nyolcbites Auth Type, mely a használt hitelesítési eljárást jelöli. Például 1 – sima jelszó, 4 – SHA1 kulcs. Auth Len mező az autentikációs szekció hosszát tartalmazza. Míg az Authentication Data összetett mező, értéke a használt hitelesítési eljárástól függően vesz fel értéket.



3. ábra. BFD Control Packet autentikációs mezői [6]

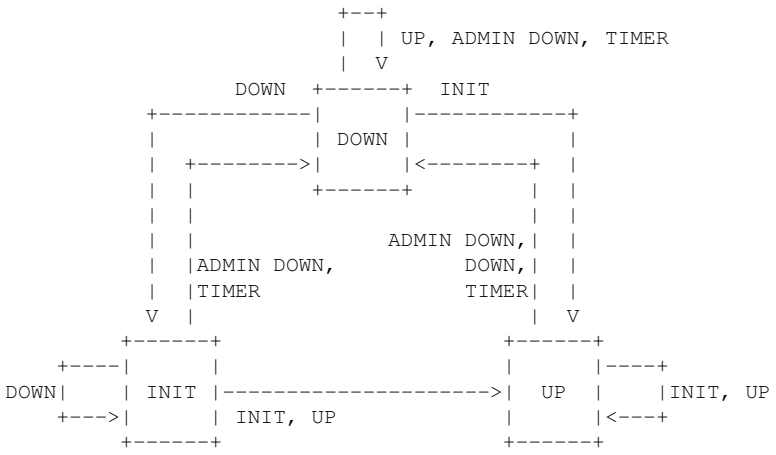
A csomag felépítésén kívül a BFD működésének megértéséhez fontos a hozzá tartozó állapotgép ismerete is. Ezt a csomagban a Sta mező tartalmazza. Ennek három értéke lehet: adminisztratívan leállt (AdminDown), leállt (Down), inicializál (Init), működik (Up). Az állapotgépet a 4. ábra szemlélteti.

Az inicializáló (Init) állapotban az egyik fél kezdeményezi a BFD munkamenet felépítését. Ez az állapot vagy addig tart, amíg a másik fél Init vagy Up Sta mezős BFD vezérlő csomagot nem küld, így felépülhet a BFD munkamenet. Vagy ha az észlelési időn belül nem érkezik ilyen csomag, akkor a munkamenet felépülése meghiúsul, mert a másik fél nem elérhető, a rendszer állapota így Down.

Működő (Up) állapotban a munkamenet felépült, a link működik. Egészen addig marad ebben az állapotban, amíg a kapcsolat meg nem szakad vagy lekapcsolják. A kapcsolatmegszakadás akkor beszélünk, ha az észlelési időn belül nem érkezik válaszüzenet vagy a másik fél válaszában a Sta mező értéke Down. Ilyen esetben a rendszer Down módba lép.

Leállt (Down) állapot jelzi, hogy a munkamenet még nem épült fel vagy linkhiba miatt megszakadt. Tehát ez az állapot jelzi, hogy a link nem létezik. Ha ilyen állapotban érkezik egy szintén Down üzenet, akkor Init módba lép, hiszen a linkhiba valószínűleg megszűnt,mert a másik féltől érkezett üzenet. Ha pedig rögtön Init üzenet érkezik, akkor Up módba lép, hiszen a kapcsolat kiépítése megkezdődött.

Adminisztratívan leállt (AdminDown) móddal lehet a rendszert kényszeríteni Down állapotra. Ilyen állapottal megjelölt csomag hatására a fogadó fél Down módba lép és mindaddig abban marad, amíg a küldő fel nem oldja az AdminDown állapotot. A link állapotáról nem hordoz információt.



4. ábra. BFD állapotgép [6]

Ezen BFD-s témák tisztázása után vissza lehet térni a három működési mód részletesebb bemutatásához. Kezdve az aszinkron móddal, ahol a két fél között egy- vagy kétirányú periodikus csomagküldés zajlik. Amennyiben a csomagokra érkezési időn belül nem érkezik válasz, úgy a rendszer Down állapotba kerül. Az aszinkron mód nagy előnye, hogy megadott észlelési időn belül fele annyi csomagot használ, mint az echó funkció. Az aszinkron mód észlelési ideje megegyezik a számított észlelési idővel, ami megjelenik csomagban, ugyanis ez a Detect Mult mező értéke.

Folytatva az igény-vezérelt (Demand) móddal. Már említésre került, hogy ez a mód épít arra, hogy a link állapotának ellenőrzését a BFD-n kívül más eljárás is végzi. Ilyen téren rugalmas, a két irányban akár használható eltérő eljárás is. Az igény-vezérelt mód aktiválásához be kell billenteni a BFD csomag D (Demand) bitjét. Amennyiben ez csak egy irányban történik meg, úgy periodikusan kell BFD vezérlő csomagokat küldeni hasonlóan az aszinkron módhoz. Ha mindkét végpont igény-vezérelt módban operál, akkor el lehet hagyni a periodikus csomagküldést

és elég csak kérésre végrehajtani a link ellenőrzését. Ezáltal jelentősen csökken a BFD hálózati forgalma. A link ellenőrzésére úgynevezett poll szekvenciákat küldenek észlelési idő végéig. Ha addig nem érkezik válasz, a küldő átlép Down módba. A poll szekvencia olyan egymásután kiküldött csomagokat fog össze, melyekben a P (Poll) bit be van billentve. Ezen csomagok küldése addig tart, amíg a másik féltől nem érkezik rá válaszul F (Final) bittel ellátott csomag. Jellemzően ezek a szekvenciák csak egy-egy csomagból állnak mindkét irányban, de nagy késleltetésű, illetve zajos csatornán elképzelhető, hogy több ilyet kell küldeni, hogy válasz érkezzen rá. Mivel az észlelési idő véges, így az felülről korlátozza a kiküldhető szekvenciák számát. Továbbá poll szekvenciák segítségével lehet a Demand módot tetszőleges időben ki- illetve bekapcsolni. Eddig nem esett szó az alkalmazott észlelési időről. Ez nem véletlen, mivel nem egyértelmű Demand mód esetén ennek meghatározása. Ugyanis e mód más, BFD által ismeretlen link figyelő eljárásokra építkezik. Ezen eljárások ráadásul a két végponton el is térhetnek, így a pontos észlelési idő meghatározása nehéz feladat. Ezért az összesített észlelési időt egy adott linkre a kalkulált észlelési idő és az első poll szekvencia inicializálása előtti időösszege jelenti.

Előbbi kettő módra épül az echó funkció. Ilyenkor a küldő fél BFD echó csomagokat küld, amiket a vevő megérkezés után visszaküld. Ez persze a két végpont között bármilyen irányban, akár egyszerre mindkét irányban működhet. Ennek megvalósulásához előnyös, ha jelzik az echózni vágyó felek, hogy hajlandóak echózni a beérkező csomagokat. Ehhez érdemes meghirdetniük a Required Min RX Interval és Required Min Echo RX Interval értékeiket is, hiszen így a másik fél ennek ismeretében meghatározhatja az ő észlelési idejét, illetve csomagküldési rátáját. Előbbi adja meg a rendszer csomagfogadásra szánt minimális időtartamát, míg utóbbi a rendszer echó csomagok minimális fogadási időtartamát adja meg. Ha a visszaérkező csomagok számossága eltér a kiküldöttektől, akkor valószínűleg link-hiba történt. Látható az is, hogy az echó funkció viszonylag jól támadható. Közbeékelődő harmadik fél – akihez hamarabb érnek el a csomagok, mint a valódi echózó félhez – is echózhat csomagokat, ezáltal például maga fele terelhet forgalmat. Éppen ezért tartalmaz a BFD viszonylag részletes hitelesítési lehetőségeket, ezek alapjait már említésre kerültek, mélyebb részletezésüktől eltekintenék.

Az eddig ismertett eljárások általánosan a BFD protokollra érvényesek. Megjelenése után hamar készült hozzá kiegészítés IPv4 és IPv6 támogatásra [7]. Ez kiemelten fontos, ugyanis az alsó kategóriás hálózati útvonalválasztók jellemzően IP-ra építenek. Leszögezendő, hogy ezen BFD kiegészítés csak egy ugrást támogat mind IPv4, mind IPv6 esetén. Ez előny, hiszen így az echó funkció használatánál egyszerűen vissza lehet irányítani a csomagokat. A BFD csomagok UDP-be csomagolandóak, munkamenetenként küldő oldalon egy darab, a $[49152, 65535]$ porttartományból választott portról küldendőek a fogadó oldali 3784-es portra. Az echó csomagokat kötelezően úgy kell küldeni, hogy a fogadó fél tudja fogadni,

így például osztott közeget (lásd Ethernet) használó többes hozzáférésű hálózaton az adatkapcsolati rétegbeli címmel kell címezni az echózó felet. A vezérlő csomagoknak pedig többes hozzáférésű hálózaton tartalmazniuk kell mind a küldő, mind a fogadó fél – kötelezően egy alhálózaton belüli – címét. Fontos továbbá, hogy az IP fejléc TTL [7] vagy Hop Limit mezőjének értékét 255-re kell állítani. Ennek hitelesítési szerepe van a GTSM [8]-re alapozva. Viszont ez ennek a dolgozatnak nem témája, így a részletezésétől eltekintek. A nem 255-ös TTL vagy Hop Limit értékű csomagok eldobása és a BFD által támogatott hitelesítési eljárások hatékony autentikációt tesznek lehetővé.

Összeségében tehát elmondható a BFD-ről, hogy egy viszonylag jól skálázódó, pehelysúlyú link-hiba detekciós megoldás. Jól-skálázódó, hiszen minden csomagkapcsolt hálózat támogatása mellett többféle hitelesítési eljárást is támogat. Ugyanakkor pehelysúlyú, hiszen kis méretű és kevés csomagtípussal dolgozik, így csomagjai egyrészt könnyen feldolgozhatók, másrészt kisebb sávszélességet igényel elküldésük, tehát viszonylag kicsi pluszterheléssel jár az alkalmazása. Továbbá hangsúlyozandó, hogy kellőképpen kifinomult eljárás, így támogat hitelesítést is különböző eljárások segítségével.

Teljesen más szemléleten alapul az a megközelítés,hogy a link hibáit a végpontok saját magukon belül kezeljék le. Ez amiatt korrekt megoldás, mert megfelelő implementációval minden végpont a képességeihez mérten tudja kezelni a link-hibákat. Egy link két végpontja tehát egymástól függetlenül, akár teljesen különböző módszerrel is kezelheti a link állapotát. Ezt az elvet követi a 3. fejezetben részletezett, alsó kategóriás útvonalválasztókat megcélzó, hatékony link-hiba detekciós eljárás is.

2.3. A linkvesztés detektáló eljárások összefoglalása

A bemutatott linkvesztés detektálásra használható eljárások többféle szempontok alapján csoportosíthatók. Léteznek topológia centrikus és topológia független eljárások. Topológia centrikus az LLDP, illetve az összes ráépülő eljárás, mint például az OFDP vagy az STP. Ezeknek elsődleges feladatuk a hálózat felépítéséről információt szolgáltatni. Mivel a hálózat linkek sokaságából áll, így ezene eljárások feladatkörébe beletartozik a link-hibák detektálása is. Viszont ezek az eljárások a működésükkel járó hálózatbejárás miatt lomhák, több másodpercbe telik egy linkvesztés detektálása. Mivel a teljes hálózatot bejárják, így a hálózat méretétől is függ a linkhiba detekciós idejük. Ezzel szemben a topológiától függetlenül működő eljárások link szinten működnek. Mivel ezen eljárások feladatköre kisebb és működési tere behatárolt, ezáltal nagyságrendekkel kisebb válaszidőt tudnak

[7] IPv4 fejlécben található mező a TTL (Time to Live).

[8] GTSM: The Generalized TTL Security Mechanism (RFC 5081)

produkálni link-hiba detekció terén. Ebbe a kategóriába tartozik például a BFD, illetve a végpontok által történő közvelten link-hiba feldolgozás.

Csoportosíthatók az eljárások hálózat használat alapján is. Legnagyobb forgalmat az egész hálózatot lefedő LLDP, OFDP, STP generálja, hiszen ezeknél az eljárásoknál minden végpont minden szomszédjának periódikusan küld üzenetet. Ezzel szemben viszonylag kevés, kis méretű csomagokat használ a BFD, így kevésbé terheli a hálózatot. Végpontok által történő közvelten link-hiba feldolgozása működési jellegéből adódóan nem használja a hálózatot.

Összegezve tehát gyors, hatékony linkhiba detektálásra alkalmazható a BFD protokoll, illetve a végpontok által megvalósított linkhiba detekció.

3. Hatékony linkvesztés detektálás alsó kategóriás útvonalválasztókon

3.1. Motiváció

A bevezető, 1. fejezetben már említésre került, hogy nem csak a háztartásokban örvendenek viszonylag nagy népszerűségnek az alsó kategóriás útvonalválasztók, hanem különböző kutatások vagy fejlesztések kísérleti, illetve demonstrációs platformjaként is. Hiszen ezen eszközök árukhoz képest meggyőző teljesítményt és szolgáltatásokat nyújtanak.

Ezen eszközök tipikus használati esete az OpenFlow alapú hálózat kiépítése. Ilyenkor jellemzően ezen berendezések OpenFlow kapcsolóként üzemelnek és külön PC-n fut az OpenFlow kontroller. Ezen a hálózaton lehetőség van különböző útvonalválasztási eljárások tesztelésére, bemutatására. Ehhez viszont hasznos lenne, ha létezne az OpenFlow által is használt, a dolgozatban bemutatott OFDP-nél, illetve LLDP-nél hatékonyabb link-hiba detektálás.

Viszont alapértelmezetten se ezek az eszközök, se a szoftveres OpenFlow kapcsoló sem támogat OFDP-re és LLDP-re épülő link-hiba detekciós eljáráson kívül mást. Azonban erre az alapvető fontosságú problémára már több megoldás is született, melyek az OFDP/LLDP-nél hatékonyabbak, de sajnos jellemzően eszköz specifikusak, illetve az implementációjuk nem a leghatékonyabb. A dolgozatomban ismertetett megoldás ezeket a problémákat próbálja meg áthidalni. Viszont a részletes ismertetést megelőzően szükséges részletesebben bemutatnom ezen eszközöket.

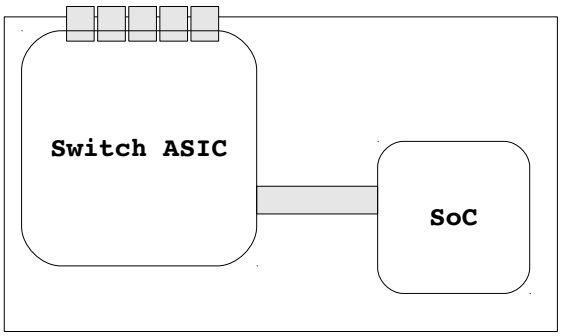
3.2. Alsó kategóriás hálózati útvonalválasztók felépítése

Az alsó kegóriás hálózati útvonalválasztók célhardverek, beágyazott rendszerek. A rendszer magja egy olyan integrált áramkör, melyben megtalálható a processzor, memóriavezérlő, stb. Angolul ezt a megoldást System on a chipnek (SoC) hívják. E köré csatlakozik az RAM modul, flashmemória, kommunikációért felelős chipek, továbbá a hálózati feladatspecifikus integrált áramkörök. Ilyen például a kapcsolót megvalósító chip vagy a rádiós kommunikációért felelős chip.

Magasabb szinten tekintről tekintve ezeket az eszközöket elmondhatjuk, hogy tartalmaznak valamilyen redukált utasítás készletű (RISC), jellemzően 200–800 MHz-es MIPS vagy ARM processzort, melyhez csatlakozik 8–256 MB közvetlen hozzáférésű memória, továbbá 4–128 MB flashmemória. Külső periféria illesztésére különböző technológiákat támogatnak. Ilyen például a soros port, JTAG, általános célú bemenet/kimenet (GPIO), USB. Ezek használatához elképzelhető, hogy az eszközt módosítani kell, hiszen nem mindegyik van kivezetve. Számí-

tógép hálózatok közül a vezeték nélküli kapcsolatokhoz a IEEE802.11-es szabványcsalád részhalmazát, vezetékes kapcsolatokhoz pedig a gyors, illetve gigabites Ethernetet támogatják.

A dolgozat szempontjából kiemelten fontos az eszközökön található hálózati kapcsoló feladatspecifikus integrált áramköre. Ugyanis az útvonalválasztón található Ethernet aljzatok a kapcsoló áramkörhöz kapcsolódnak. Továbbá ezeket az Ethernet portokat a kapcsoló áramkör a rendszertől függetlenül kezeli. A kapcsoló a rendszerhez külön porton kapcsolódik, mint ahogyan az látható az 5. ábrán.



5. ábra. A kapcsolót megvalósító IC elhelyezkedése az alaplapon

Ennek az architektúrának kétségen kívül előnye, hogy a rendszert tehermentesíti a csomagtovábbítástól. Ezáltal a gyártó spórolni tud az egyébként költségesebb SoC áramkörön, így a végtermék, azaz a bedobozolt hálózati útvonalválasztó árát le tudja szorítani. Az alsó kategóriás, otthoni útvonalválasztók pedig árérzékeny szegmensbe tartoznak.

Ugyanakkor hátránya ennek az architektúrának, hogy ezt a kapcsolatot a kapcsoló többi portjainak forgalmához képest kisebb sávszélességűre méretezték, mivel nem a csomagok kis részét kell csak a SoC áramkörben megvalósított hálózati rétegekbe eljutattni. Éppen ezért, ha a kapcsolót OpenFlow-val útvonalválasztóként használjuk, ez az összeköttetés jelentősen terhelődik, hiszen ilyenkor minden csomagnak át kell haladni az szoftveres útvonalválasztón, ami a kapcsoló feletti hálózati rétegben (Hálózati réteg) működik, ami viszont a SoC-on valósul meg.

3.3. Alsó kategóriás hálózati útvonalválasztók szoftveres felépítése

Ezen eszközök a gyártótól valamilyen előretelepített firmware-rel érkeznek, amellyel egy egyszerű webes felületen keresztül lehet kommunikálni. Ez otthoni felhasználásra tökéletes, viszont komolyabb módosításokra nem ad lehetőséget.

Viszont lehetőség van a gyári operációs rendszert lecserélni olyan szabad, nyílt forráskódú szoftverre, mellyel az eszközön mindenhez hozzáférést biztosít.

Ilyen operációs rendszer az OpenWrt, ami egy beágyazott rendszerekre szánt ingyenes Linux disztribúció, ahol a kiadások mind egy-egy koktélról kapják a nevüket. A legutolsó stabil kiadás neve Attitude Adjustment, 2013. áprilisában jelent meg és a 3.3.8-as verziószámú Linux kernelre épül. Az OpenWrt jellemzően hálózati eszközökön futtatható, de létezik portja más platformokra, többek között PC-re is. Különféle perifériákat támogat és olyan szolgáltatásokat nyújthat, amiket a gyári firmware nem. Ilyen szolgáltatás például az VPN szerver/kliens, CUPS, SFTP, Samba, Asterisk, PulseAudio. Külön kiemelendő tulajdonság az opkg nevű csomagkezelő és a uci nevű egységes konfigurációs felület. A rendszer ezenkívül további hatékony eszközökkel is fel van szerelve. Ezek közül a dolgozat szempontjából kiemelendő az swconfig nevű program, amellyel bizonyos hálózati kapcsolók menedzselhetők. Továbbá swconfig segítségével lehetőség van részletes információt szerezni a kapcsoló aktuális állapotáról. Így például az swconfig használatával lekérdezhető a portok állapota, azaz, hogy van-e link az adott porton.

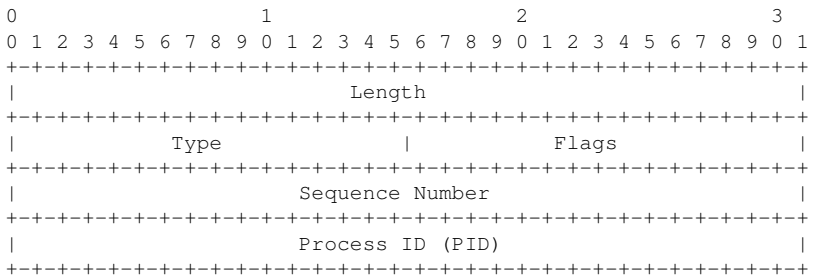
Az swconfig szerkezetileg két részből áll. Egyik fele egy kernelmodul. Feladata a rendszerben található kapcsolókhoz hozzáférést biztosítani, azokat menedzselni. Az swconfigban a hálózati kapcsolókat a switch_dev struktúra írja le, mely tartalmazza a nevüket, sorszámukat, portjaiknak számát, VLAN-ok számát, stb. Ahhoz, hogy az swconfig egy kapcsolót el tudjon érni, a kapcsolónak az eszközmeghajtójában swconfig specifikus függvényeket kell megvalósítania. A megvalósítandó függvényeket a switch_dev_ops struktúra írja le, ami megtalálható minden kapcsoló switch_dev struktúrájában. Ezen függvények feladata a kapcsoló adatainak lekérhetését, illetve azok értékeinek módosítását megvalósítaniuk. Konkrét példa erre a port-státusz lekérdezése, illetve virtuális helyi hálózatok felkonfigurálása. A különböző hálózati kapcsolókat leíró sturktúrákat az swconfig egy swdevs nevű láncolt listába fűzi össze. Ahhoz, hogy egy hálózati kapcsoló bekerülhessen ebbe a listába, eszközmeghajtójában be kell regisztrálnia magát az swconfig register_switch() nevű függvényének meghívásával. Ennek hatására az swconfig hozzáadja a kapcsolót leíró struktúrát a a kapcsolókat összefogó listájához, ha még az nem szerepel benne. Természetesen arra is van lehetőség, hogy egy kapcsoló eszközmeghajtója kivetesse magát a listából. Ehhez az unregister_switch() nevű swconfig függvényt kell meghívnia. Tehát a hálózati kapcsolók eszközmeghajtóinak szerepe a kapcsoló és a rendszer közötti kommunikáció biztosítása, illetve erre építve az swconfig függvényeinek eszközspecifikus implementálása.

Az swconfig másik egysége felhasználói módból érhető el. Feladata pedig a saját kernelmoduljának függvényein keresztül hozzáférést biztosítani a hálózati kapcsolókhoz. Ehhez a C-ben írt API-n túl egy parancssori alkalmazást is nyújt.

Ennek neve szintén swconfig, az API-é pedig swlib. A kernelmodul és a felhasználói módban futó alkalmazás között a netlink teremt kapcsolatot.

A netlink egy linuxos socket család, mely felhasználói módú alkalmazásoknak biztosít folyamatok közti kommunikációt, azaz Inter-Process Communicationt (IPC-t). Full-duplex kapcsolatot teremt a kernel módban futó és a felhasználó módban futó programok között hálózati információk továbbítására. Szolgáltatásaival hatékonyabban lehet kommunikálni felhasználói mód és kernel mód között, mintha azokat egyszerű rendszerhívásokon, vagy proc fájlrendszeren keresztül oldanánk meg [9]. A rendszerhívásokat mindig csak felhasználói módból lehet kezdeményezni, ezzel szemben egy netlink üzenet küldése kezdeményezhető kernel módban is. Mivel a netlink aszinkron, multicast címzésre képes a socket API révén, ilyenkor az üzenet bekerül a fogadó fél vagy felek netlink várakozási sorába és meghívja a fogadó kezelőfüggvényét. Ez dönthet arról, hogy az üzenetet rögtön, még abban kontextusban feldolgozza vagy későbbre, azaz egy másik kontextusra hagyja. Ezáltal hatékonyabban tud alkalmazkodni a rendszer ütemezéséhez, mivel nő az üzenő, illetve feldolgozó feladat granualitása.

A netlink fejlécformátumát mutatja be a 6. ábra. Az első mező, a 32 bites Length a teljes üzenet hosszát tartalmazza byte-okban. Ezt követi a 16 bites Type mező, amelyben az üzenet tartalmát adhatjuk meg (pl.: ignorálandó, többrészes üzenet vége, hibaüzenet,stb.). A Flag mezővel – ami szintén 16 bit hosszú – lehet jelezni, ha az üzenet több részből áll, kérés, echo igény van, stb. A Sequence Number az üzenethez rendelt 32 bites szekvencia szám. A Process ID (PID) a küldő folyamat pidjét tartalmazza 32 biten. Ennek segítségével tudja a kernel multiplexálni a socketeket. Értéke nulla, ha felhasználói módból küldünk üzenetet kernel módba [8].



6. ábra. Netlink üzenetek fejléce [8]

A netlink sockethez (AF_NETLINK) többféle protokoll tartozik, ilyen például a NETLINK_FIREWALL, ami a tűzfal által kezelt csomagok felhasználói módba történő továbbítására szolgál, a NETLINK_ARPD, amivel felhasználói módból lehet az arp táblát módosítani, a NETLINK_ROUTE, ami útvonal-választási és

link információkat nyújt. Továbbá lehetőség van egyedi netlink üzenetformátum definiálására is. Mint látható, a dolgozat szempontjából a NETLINK_ROUTE a fontos, mert az foglalkozik a link információk továbbításával.

Az NETLINK_ROUTE IPv4-es környezetben, a NETLINK_ROUTE6 protkoll pedig IPv6-os környezetben működik. Az RTNETLINK ezeket fogja össze [10]. Az RTNETLINK változatos üzenettípusokat nyújt, lefedve különböző hálózati rétegek műveleteit. Emiatt három üzenetes csportok vannak, melyek egy-egy feladat köré szerveződnek és üzeneteikben egységes formátumot igényelnek. Ezen üzenetek nevei egy sémát követnek, létezik RTM_NEW*, RTM_DEL*, RTM_GET* üzenet. Ezek egységes fejlécformátumot használnak, így üzenetcsoportonként eltér az RTNETLINK üzenetek felépítése.

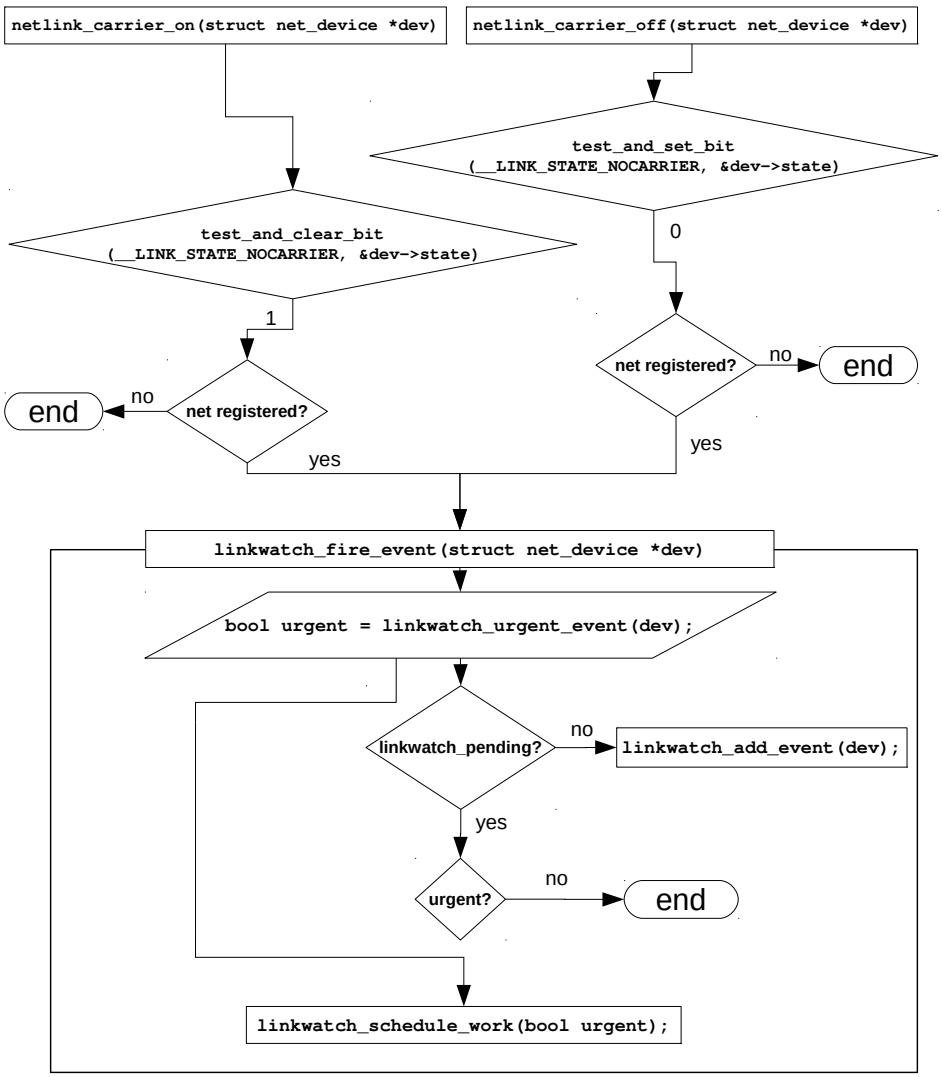
Így a link paramétereinek kezelésére is háromféle üzenet használható. Funkciójukat tekintve az RTM_NEWLINK új link létrejöttének jelzésére szolgál, míg az RTM_DELLINK-kel jelezhető egy link megszűnése. Az RTM_GETLINK üzenettel pedig a link paramétereit lehet lekérdezni. Ezen üzenettípushoz szükséges mezőket az ifinfomsg struktúra írja le. Ebben található az üzenetben említett hálózati eszköz típusa, neve, egyedi azonosítója, állapotátjelzői, illetve az állapotjelző változást jelölő bitmaszk. Az így létrejött mezők, illetve a csomag típusának a netlink fejlécbe való beszúrására szolgál a __nlmsg_put() függvény.

Az üzenet küldése előtt létre kell hozni egy socketet. Kernel módban erre a netlink_kernel_create() függvény szolgál. Ez paraméterként megkapja használandó protokoll azonosítóját. RTNETLINK esetén ez NETLINK_ROUTE*. Továbbá meg kell adni a socketet használó modul nevét, illetve a használandó hálózati névteret leíró struct_net struktúrát. Alapértelmezetten a kernel nyújt névteret init_net néven, mely tartalmazza a rendszerben található hálózati eszközöket. Az üzenetek küldésére kernel módból az rtnetlink_broadcast() függvény szolgál. Melynek első paramétere szintén a hálózati névteret leíró struct_net struktúra. Második paramétere az rtnetlink üzenetet tartalmazó puffer. Ezenfelül megadható port azonosító, illetve célcsoport.

Mivel a linkállapot változások kezelése a hálózati eszközök feladata, ezért az eszközmeghajtóikban használható olyan függvény, mely a linkállapot változás jelzésére szolgál. Nevezetesen ez a netif_carrier_on(struct net_device *dev) és a netif_carrier_off(struct net_device *dev), ahol a net_device a hálózati eszközt reprezentáló struktúra. Ezek a függvények pont a portstátusz-változás jelzésére használhatóak. A netif_carrier_on()-nal lehet jelezni, ha él a link. A netif_carrier_off()-fal pedig azt, ha megszűnt [11].

A háttérben lejátszódó folyamatokat a 7. ábra szemlélteti. Látható, hogy a netif_carrier_on() lekéri, majd törli az eszköz állapotát tároló bitek (dev.state) közül a __LINK_STATE_NOCARRIER-t, majd amennyiben az 1 volt, azaz eddig nem volt vivő az eszközön folytatja futását. Ezzel szemben a netif_carrier_off() lekéri, majd egybe állítja a __LINK_STATE_NOCARRIER-t és amennyiben az

0 volt, azaz volt vivő az eszközön, folytatja futását. Ezek után mindkét függvény a linkwatch_fire_eventet hívja meg, ha az eszköz nem regisztrálatlan. Ez a függvény adja hozzá a létrejött értesítést a hálózati eszközt leíró struktúra link_watch_list láncolt listájához, illetve kezdeményezi a feladat lista feldolgozását a linkwatch_schedule_work() függvény segítségével, ami további alacsonyabb szintű kernel függvények meghívásával eljuttatja az arra érdemes feladatot az ütemezőnek, ahol feldolgozás során megtörténik a jelzés emittálása, hiszen az eszközhöz tartozó flagek eltérnek a __LINK_STATE_NOCARRIER-ben .



7. ábra. A netif_carrier_{on,off} függvényhívás egyszerűsített folyamatábrája

Tehát OpenWRT és egyéb GNU/Linux rendszerek alatt a link-hibát a felhasználói módban futó alkalmazásoknak RTNETLINK üzenet segítségével lehet egyszerűen és hatékonyan jelezni. Ennek a jelzésnek az emittálása viszont a hálózati eszközök eszközmeghajtóinak feladata.

A gond viszont az, hogy a már említett hálózati kapcsolókat leíró switch_dev struktúra nem feltétlenül tartalmazza a hálózati eszközöket leíró net_device struktúrát. A kapcsolók hálózati interfésszé történő megfeleltetése eszközspecifikus, nem terjed ki minden port külön leképezésére. Ellenben a hálózati kapcsolón létrehozott virtuális helyi hálózatokhoz tartozik net_device struktúra. Tehát rögtön több lehetőség is adódik e probléma megoldására, melyhez viszont szükség van a meglévő OpenWrt módosításához.

OpenWrt-re való fejlesztésre szolgál az OpenWrt Buildroot nevű fejlesztői eszköz. Ez a buildroot nevű szoftverre épül, magában foglalja kereszt-fordításhoz szükséges eszköztárat (fordító, linker, autotools, stb.), a uClibc standard C könyvtárat, illetve a csomagok előállításához szükséges Makefile-okat és még sok egyebet. Menüs beállítóeszközt is nyújt a célhardver, illetve a létrejövő firmware paramétereinek beállítására [16]. Végtermékként egy eszközre tölthető képfájlt gyárt, illetve a csomagokat. A képfájl az eszközre történő másolása után az mtd paranccsal telepíthető.

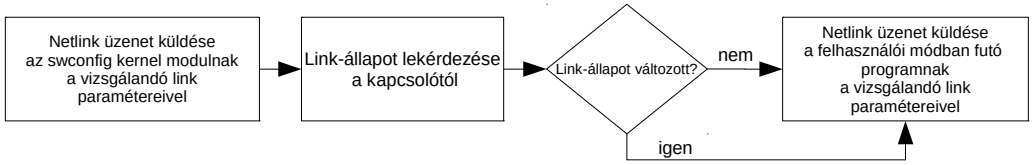
3.4. Hatékony linkvesztés detektáló eljárás megvalósítása

Sajnálatos módon használatra érdemes BFD implementáció nem létezik e platformra. Hatékony megvalósítása kétséges is, hiszen alsó kategóriás útvonalválasztók bemutatásából kiderült, hogy az eszközön futó operációs rendszer külön eszközként kezeli a hálózati kapcsolót. A Linux kernelben az ilyen eszközöket platform_device-oknak hívják és a hozzájuk tartozó eszközmeghajtók speciálisak, betöltésükre csak a operációs rendszer indulásakor van lehetőség [12]. Éppen ezért a hálózati kapcsolókhoz tartozó eszközmeghajtók szerepköre a hálózati kapcsoló és a rendszer közötti kommunikáció biztosítása. Ennek elfedését végzi az swconfig.

A linkállapot információkat a kapcsoló saját regisztereiben menedzseli. Tehát a hálózati kapcsoló képes detektálni a link jelenlétét a csatlakoztatt rézkábel fizikai tulajdonságai alapján.

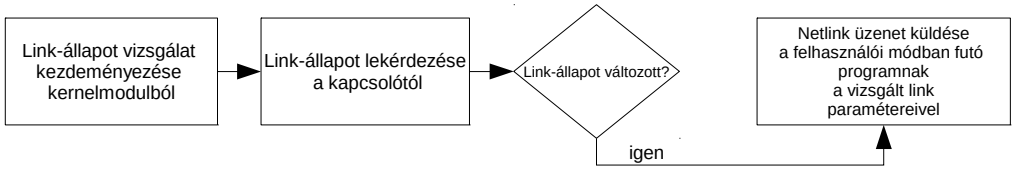
Mivel a hálózati kapcsoló nem tud interruptot küldeni a rendszernek linkvesztésről, ezért annak feltárását a rendszernek kell kezdeményeznie. Ehhez szükséges, hogy a rendszer periodikusan lekérdezze a hálózati kapcsoló portjainak linkállapotát. A lekérdezésre több lehetőség is adódik. Legkézenfekvőbb az az eset, amikor felhasználói módból az swconfig API-jának (swlib) segítségével történik a periodikus linkállapot lekérdezés. Ilyenkor minden lekérdezéshez az swconfignak saját formátumú netlink üzenettel kell jeleznie a kernelmoduljának, hogy szolgál-

tasson linkállapot információt. Ezt követően a kernelmodulnak a hálózati kapcsolótól lekéri az információt, majd azt szintén egy netlink üzenetben küldi vissza a felhasználói módban futó alkalmazásnak. Ez látható a 8. ábrán.



8. ábra. Swlibbel megvalósított linkállapot lekérdezés folyamatábrája

Látható, hogy a felhasználói módból történő lekérdezés nagy hátránya, hogy sok netlink üzenetet használ. Egy periódus alatt a portok számával megegyező lekérdezés zajlik. Viszont a netlink üzenetek elhagyásával növelhető lenne a linkállapot lekérdezés hatékonysága. Ilyenkor, ahogy a 9. ábra mutatja, a kernelmodulként futó programunk az swconfigon keresztül fér hozzá a hálózati kapcsolóhoz. Ezt követően csak linkállapot változás esetén küld netlink üzenetet. Tehát láthatóan érdemes a link-hiba detekciós eljárást kernelmodulként implementálni.



9. ábra. Kernelmodulból megvalósított linkállapot lekérdezés folyamatábrája

A linkállapot lekérdezés eredményét ezt követően össze kell hasonlítani az előző állapottal. Amennyiben az előző lekérdezéskor még volt vivő, az azt követőben pedig nem, linkvesztés történt. Viszont ha az előző lekérdezés során nem észlelt vivőt, az aktuális lekérdezéssel viszont igen, úgy vagy új link jött létre vagy egy régebbi működik újra. Ennek jelzése nem tartozik szervesen a link-hiba detektáláshoz, de érdemes ezt is jelezni. A korábbi linkállapotok tárolásához módosítottam a switch_dev struktúrát, felvettem egy, a portok számával megegyező méretű port_state nevű tömböt. Ennek allokálása és inicializálása az swconfig register_switch() függvényében történik meg. Eltávolítása pedig az unregister_switch() függvényben.

A Linux kernelben a modulok, akárcsak a felhasználói módú programok, nem férnek hozzá egymás struktúráihoz, függvényeihez. Ahhoz, hogy egy modul a másik modul adatszerkezetét vagy metódusát használni lehessen, elő-

zetesen a másik modulnak exportálnia kell azt a kernel névtérbe az EXPORT_SYMBOL(szimbólum_név) makróval [11]. Az swconfig modul csak a hálózati kapcsoló be- illetve kiregisztráló függvényeit teszi ily módon más modulok számára elérhetővé. Ez komoly probléma, mert így nem lehet hozzáférni a hálózati kapcsolókat leíró adatstruktúrákhoz. Emiatt szükséges az swconfig módosítása.

A periodikus lekérdezés biztosítása időzítővel célszerű. A Linux kernel többféle időzítőt támogat. Ezek közül a legalapvetőbb a Timer API [14] által nyújtott időzítő. Ennek felbontása a rendszer legkisebb időzítő egységével, a periodikus processzor ébredések közti időszelettel egyezik meg. Ez pár milliszekundumos nagyságrendű. Pontos értéke a kernel HZ makrójának reciprokával egyezik meg. Mivel a feladathoz nem eléggé pontos a hagyományos timer, ezért ismertetésétől eltekintenék. A hagyományos timernél nagyobb pontosságot nyújt a High Resolution Timer (hrtimer) [14], azaz a nagy felbontású időzítő. Felbontása nanoszekundumos nagyságrendű. Használata a High Resolution Timer API-n keresztül történik. Mint a legtöbb időzítőnél, úgy a hrtimer esetén is egy struktúra tárolja az időzítő adatait (struct hrtimer). Ez tartalmazza az időzítőről a felhasználói szempontból fontos adatokat (elsüléskor meghívandó függvény, meddig számoljon az időzítő, stb.) és a menedzseléshez szükséges információkat (hol található az időzítőket tároló piros-fekete fában, továbbá opcionálisan statisztikai adatok). Egyszerű periodikusan elsülő időzítőhöz ezek közül csak a felhasználói szempontok fontosak. Maga a folyamat az hrtimer_init inicializáló függvénnyel kezdődik, amiben megadandó a használandó óraforrás illetve, hogy ismétlődő időzítőt jöjjön létre. Fontos, hogy ismétlődő időzítő esetén a callback függvényben gondoskodnunk kell a következő időzítés felparaméterezéséről is. A lejárat esetén hívandó callback függvényt az inicializáló függvényen kívül kell a struktúra function elemének megadni. Az időzítő elindítására a hrtimer_start függvény szolgál, amely segítségével beállítható, hogy mennyi ideig ketyegjen, és hogy ismétlődjön e az időzítő futása. A hrtimer ezen felül további szolgáltatásokat is nyújt, de azokra jelen esetben nincs szükség, így bemutatásuktól eltekintenék.

Skálázhatósági és kényelmi szempontból érdemes a link-detektáló eljárás paramétereit könnyen módosíthatóvá tenni. Ehhez a paramétereket egy-egy globális változónak kell megfeleltetni a modulon belül. Ezt követően a module_param() makró szolgál a változó típusának, illetve a hozzátartozó hozzáférési jogoknak a beállítására. Ezek után a változó a modul betöltésekor megadható, továbbá elérhető a /sys/module/<modul_név>/parameters/<változó_név> elérési út alól is.

Az eddig ismertetett eljárásokra alapozva a link-hiba detekciós, sőt linkstátusz jelző eljárást az swconfig kernelmodulban implementáltam. Alapvetően az swconfig a felhasználói módból érkező kérések feldolgozására való. Így új függvények definiálása mellett pár létezőt is módosítottam. A módosítások lokálisan három helyen történtek. Elsőként a bevezetett új változókat ismertetem. Ehhez tartozó kódrészlet látható a 10. ábrán. A változók két csoportra oszlanak. Első cso-

portba tartoznak az általános globális változók. Ezek rendre a nagyfelbontású időzítőhöz tartozó poll_timer nevű struktúrája, illetve az ahhoz szükséges időtartamot leíró ktime_t típusú poll_timer_interval struktúra. Ezeket követi két segédváltozó, melyek közül a first_poll szerepe jelezni, hogy az aktuális linkállapot lekérés előtt történt-e közvetlenül linkállapot lekérés. A base_interval a pollozási időköz változásának lekezeléséhez kell. A második csoportba a kernel paraméterek tartoznak. Ezek mind módosíthatóak a már ismertetett /sys/module/... struktúrán keresztül. Ezek közül az interval adja meg a pollozás periódusidejét milliszekundumban. A poll_enable változóval lehet a linkállapot pollozást kikapcsolni. A port_if_id tömb pedig azokat a hálózati interfész indexeket fogja össze, melyekre egy-egy olyan VLAN-t húztak ki, melyben csak a hálózati kapcsoló egy hálózati portja szerepel. Így lehetővé válik a portok külön hálózati eszközként történő kezelése.

```
static struct hrtimer poll_timer;
static ktime_t poll_timer_interval;
static unsigned long base_interval;
static bool first_poll = 1;

static unsigned long interval = 40;
module_param(interval, ulong, S_IRUGO | S_IWUSR);
MODULE_PARM_DESC(interval,"Polling interval in ms for port state
    scanning.");

static bool poll_enable = 0;
module_param(poll_enable, bool, S_IRUGO | S_IWUSR);
MODULE_PARM_DESC(poll_enable,"Grant port state polling.");

static int port_if_id[] = {8,4,5,6,7};
module_param_array(port_if_id, int, NULL, S_IRUGO | S_IWUSR);
MODULE_PARM_DESC(port_if_id,"Preconfigured one port per vlan
   interface indexes.");
```

10. ábra. Kódrészlet: az implementációban szereplő változók

A modul betöltésekor az swconfig_init() modul-inicializáló függvény fut le. A link-hiba detekció implementálás szempontjából e függvény feladata a változók értékeinek inicializálása, illetve az időzítő létrehozása. A 11. ábrán látható, hogy a base_interval értéke az pollozási periódusidővel lesz egyenlő. Ennek a változónak a callback függvényben van szerepe, így annak bemutatása során fejtem ki e értékadás szerepét. A következő sorok az időzítő létrehozását végzi. Mivel a ktime_set() függvény második paramétere az időzítő intervalluma nanoszekundumban, ezért az interval változó milliszekundumot reprezentáló értékét át kell váltani. Ezt követően megtörténik az időzítő inicializálása, majd elsütése. Callback függvénye a poll_timer_callback().

```
static int __init swconfig_init(void) {
  int i, err;

  base_interval = interval;
  poll_timer_interval = ktime_set(0, interval*1000000);
  hrtimer_init(&poll_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
  poll_timer.function = &poll_timer_callback;

  hrtimer_start(&poll_timer, poll_timer_interval,
     HRTIMER_MODE_REL);

  [...]
```

11. ábra. Kódrészlet: a modul inicializálófüggvénye

Ezen callback függvény két segédfüggvényt is használ. Ezek közül egyik a hálózati kapcsoló adott portjának link-státusz lekérdezésére szolgál. Ennek kódja látható a 12. ábrán. Ebben a függvényben történik a hálózati kapcsolóval a kommunikáció. Ennek alacsony szintű részleteit az swconfig get_port_link() függvénye, illetve a switch_port_link struktúra elfedi. Tehát a függvény a dev változóban megadott hálózati kapcsoló port sorszámú portjának linkállapotát kéri le. Hiba esetén hibás értékkel (-EINVAL) tér vissza. Továbbá az adott port linkállapotának függvényében eggyel vagy nullával tér vissza.

```
static int get_link_state(struct switch_dev *dev, int port)
{
     struct switch_port_link link;
     link.link=0;

     if(dev->ops->get_port_link(dev, port, &link))
          return -EINVAL;

     if(link.link)
          return 1;

     return 0;
}
```

12. ábra. Kódrészlet: adott port linkállapotát lekérő függvény

A másik segédfüggvény feladata a linkállapot változás jelzése. Mint ahogyan a 13. ábrán is látható, a modify_if_carrier függvény a rendszerben található hálózati interfészek közül a függvényparaméterként megadott indexűt kikeresi. Majd ezen interfészen meghívja a netif_carrier_off()-t, ha linkvesztés tör-

tént. Ellenkező esetben, új link létrejöttekor a netif_carrier_on()-nal jelez. Ezen függvények részletes ismertetése megtalálható a 3.3. fejezetben. A megvalósítás szempontjából nagy jelentőséggel bír, hogy ezen függvények szabványos netlink, vagyis rtnetlink üzeneteket használnak. Emiatt felhasználói módban kompatibilisek az összes meglévő netlink könyvtárakkal, továbbá nem igényelnek egyedi feldolgozási mechanizmusokat.

```
static void modify_if_carrier(int port, int ifidx, int state)
{
     struct net_device *dev;
     list_for_each_entry(dev, &init_net.dev_base_head, dev_list)
          {
          if(dev->ifindex == ifidx) {
               if(state == 1)
                    netif_carrier_on(dev);
               else
                    netif_carrier_off(dev);
          }
     }
}
```

13. ábra. Kódrészlet: linkállapot megváltozását jelző függvény

Ezen segédfüggvényeket felhasználva jön létre a poll_timer_callback() függvény. Tulajdonképpen ez a lelke a linkvesztés detektáló eljárásnak. Működését tekintve két nagy esetre oszlik attól függően, hogy a poll_enable változóval engedélyezett-e a pollozás. Vegyük a 14. ábrán látható első esetet, amikor engedélyezett a pollozás. Ilyenkor az swconfig kapcsolókat tároló listájából (sw_devs) kiveszi az első elemet. Látható, hogy a megoldás így minden swconfigot támogató hálózati kapcsolóval működik. Hiszen nincs konkrétan megadva a használni kívánt eszköz típusa, azt az swconfig elfedi. Azért nem szükséges a teljes listát bejárni, mert ezekben az eszközökben nem található több hálózati kapcsoló. Így kevesebb műveletet kell elvégezni, ezáltal nő a hatékonyság. Ugyanakkor ha szükséges a későbbiekben egyszerre több hálózati kapcsoló támogatása, akkor az minimális erőbefektetéssel implementálható. Ezt követően ciklikusan minden portra elvégzi a linkállapot változás vizsgálatot. Ehhez először lekéri az aktuális állapotot. Majd összeveti ezt az előző állapottal és ha történt eltérés a linkállapotában, akkor meghívja a modify_if_carrier() segédfüggvényt, amely kezdeményezi a linkállapot változás hirdetését a rendszerben. Ezek után a kapcsolót leíró struktúrában is elkönyveli a változást, azaz a kapcsoló port-állapotát tároló tömb megfelelő elemének értékébe beírja az aktuális linkállapotot. Első pollolás esetén viszont nincs mihez viszonyítani, így ilyenkor az aktuálisan kiolvasott érték a mérvadó, annak megfelelően történik linkállapot jelzés és az állapotbeállítás. Vé-

gül nullázza a first_poll értékét, hiszen a következő periódus már az aktuális után következik, annak eredményére épít.

Ha nincs engedélyezve az időzítő, akkor a függvény a második, a 15. ábrán is látható felében levő utasításokra ugrik. linkállapot ellenőrzés helyett csak az időzítő menedzselése zajlik. Szándékosan az intervaltól eltérő értékűvé kell tenni a base_intervalt. Erre azért van szükség, mert ha a felhasználó később visszakapcsolja a pollolást, akkor az így újra fel tudja venni az intervalban megadott pollolási periódusidőt. Ugyanis a hrtimer-t nem szabad leállítani. Leállítást követően nem tudnánk hozzáférni, ezért ha nem zajlik linkállapot lekérdezés, akkor is fut a paramétereken történő változások lekövetése miatt. 100ms-os időközönként vizsgálja, hogy történt e változás. Ez az időtartam elég nagy ahhoz, hogy lényegében ne terhelje a rendszert, ugyanakkor elég kicsi ahhoz, hogy a felhasználó ne vegye észre.

Az első ág zárásaként még megtörténik az esetleges interval, illetve base_interval módosítás lekezelése. Az interval értékét csak a felhasználó módosíthatja, míg a base_interval értékét csak a program módosíthatja, például abban az esetben, ha lekapcsolták a pollolást. Az interval értékének beállítását követően a hrtimer újraindítása zajlik a hrtimer_forward függvénnyel, illetve a HRTIMER_RESTART értékkel történő visszatéréssel.

Az implementálás részletein túl megemlítendő a link-hiba detekciós eljárás alkalmazása is. Mivel az egész detekciós eljárás az swconfig kernelmoduljára épül, ami viszont a kernel szerves részét képezi, ezért ahhoz, hogy használni lehessen, a kernelt újra kell fordítani. Ezt követően az új kernel telepíthető az eszközökre. Minden swconfigot támogató hálózati kapcsolóval kompatibilis, használata előtt csupán minden porthoz egy dedikált VLAN interfészt kell létrehozni.

```
static enum hrtimer_restart poll_timer_callback(struct hrtimer *timer)
{
    struct switch_dev *dev;
    int i,cur_port_state;

    if(poll_enable) {
        dev = list_entry((&swdevs)->next, typeof(*dev), dev_list);
        for(i=0; i < dev->ports-1; i++) {
            cur_port_state = get_link_state(dev,i);
            if(dev->port_state[i] == 0 && cur_port_state == 1) {
                modify_if_carrier(i,port_if_id[i],1);
                dev->port_state[i] = 1;
            }
            else if(dev->port_state[i] == 1 && cur_port_state == 0) {
                modify_if_carrier(i,port_if_id[i],0);
                dev->port_state[i] = 0;
            }
            else if(first_poll) {
                if(cur_port_state){
                    modify_if_carrier(i,port_if_id[i],1);
                    dev->port_state[i] = 1;
                }
                else{
                    modify_if_carrier(i,port_if_id[i],0);
                    dev->port_state[i] = 0;
                }
            }
        }
        first_poll = 0;
    }
```

14. ábra. Kódrészlet: az időzítő callback függvény első fele

```
    else {
        if(base_interval == interval)
            base_interval = interval+1;
        first_poll = 1;
        poll_timer_interval = ktime_set(0, 100*1000000);
        hrtimer_forward(&poll_timer, ktime_get(), poll_timer_interval);
        return HRTIMER_RESTART;
    }

    if(interval != base_interval) {
        if(interval < 1)
            interval = 1;
        poll_timer_interval = ktime_set(0, interval*1000000);
        base_interval = interval;
    }

    hrtimer_forward(&poll_timer, ktime_get(), poll_timer_interval);
    return HRTIMER_RESTART;
}
```

15. ábra. Kódrészlet: az időzítő callback függvény második fele

3.5. A megvalósított linkvesztés detekció kiértékelése

A 3.4. fejezetben ismertetett linkvesztés detektáló eljárás implementálása TP-Link TL-WR1043ND típusú hálózati útvonalválasztón történt. Ez az eszköz Atheros AR9132 rendszerchipre épül, így 400MHz-es MIPS processzor és 32MB RAM található benne [15]. Az eszközben található hálózati kapcsoló Realtek RTL8366RB típusú. Öt darab gigabites Ethernet porttal és a CPU porttal rendelkezik. Viszonylag fejlett chip, támogat többek közt VLAN-t, QoS-t [13].

Az útvonalválasztón OpenWrt 12.09.1-es kiadása futott.

A link-hiba detektáló eljárás mérése OpenFlow hálózaton történt. A mérés célja az volt, hogy linkvesztés esetén az OpenFlow útvonalválasztó szolgáltatás másik, előre megadott portra terelje a forgalmat. A végpontokon futó forgalomrögzítés alapján pedig meghatározhatóvá válik így az eldobott csomagok száma, amelyből következtetni lehet a link-hiba detektálás idejére.

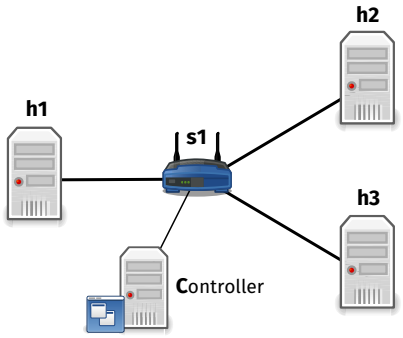
A méréshez megkezdéséhez viszont szükséges az OpenFlow hálózat felépítése. Ezért az útvonalválasztón futott az OpenFlow szoftveres kapcsoló, egy külön PC-n pedig az OpenFlow kontroller. A méréshez szükséges volt a létező programok módosítására, így az OpenFlow kapcsolóéra is. Az OpenFlow kapcsoló két részből áll, egyik az ofdatapath nevű program. Ennek feladata a hálózati eszközök menedzselése. Másik részt az ofprotocol nevű program alkotja, melynek feladata a kontrollerrel történő kommunikáció megvalósítása és az ofdatapath irányítása. Csomagot dolgoz fel, folyamtáblákat tart karban, illetve az ismeretlen folyamokat továbbítja az ofprotocolnak. Ahhoz, hogy az OpenFlow rendszer értesülni tudjon a netlinken meghirdetett portstátusz-változásokról, módosítani kell azt. Erre több lehetőség adódik. Ezek közül egyik az, hogy külön program figyeli a netlink üzeneteket, és ha egy kapott üzenet portstátusz-változásról szól, szignált küld az ofdatapath-nak. Másik lehetőség az, hogy az ofprotocol figyeli a netlink üzeneteket és OpenFlow üzenettel értesíti az ofdatapath-t a portstátusz-változásról. A mérések az utóbbi változattal zajlottak.

Az ismertetett link-hiba detektáló eljárást egy létező, hasonló linkállapot polloló programmal vetettem össze, melynek neve portd. A portd egy felhasználói módban futó, swlib API-t használó port-státusz polloló program küld szignált az ofdatapath-nak a 3.4. fejezetben ismertetett módon. Ehhez szintén módosítani kellett az ofdatapath-t.

Az OpenFlow kontroller egy lepke elrendezésben működő link-hiba áthidaló bemutatóból származott. Ezen minimális módosításokat kellett eszközölni, hogy a mérésben felhasználható legyen. Ezek elenyésző méretűek, így részletezésükre nem térnék ki.

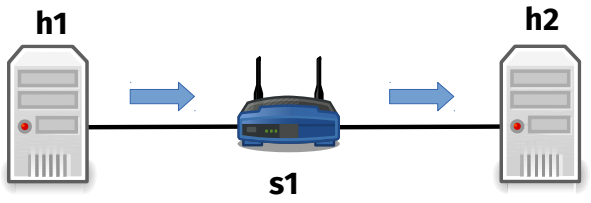
A mérési elrendezés látható a 16. ábrán. Három átlagos PC (h1,h2,h3) és az OpenFlow kontroller (C) csatlakozik az útvonalválasztóhoz (s1).

A mérés során a gyakorlatba bevett portd, illetve a 3.4. fejezetben ismertetett



16. ábra. A mérési elrendezés

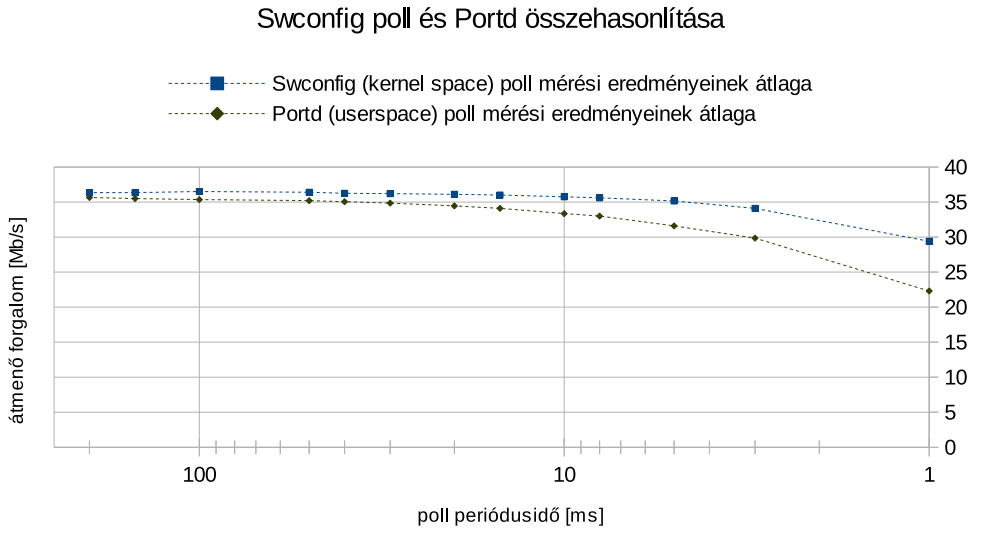
link-hiba detekciós eljárás szerepelt. A mérések a két végpont közötti maximális átviteli sebesség meghatározásával kezdődtek. Ennek meghatározására a 17. ábrán látható elrendezésben a (h2) számítógépen futó iperf szerverre csatlakozott a (h1) iperf kliens az OpenFlow kapcsolóként működő (s1)-en keresztül. Több egyperces TCP forgalom generálás eredményeinek átlagaként 36.4Mbits/sec átviteli sebesség adódott.



17. ábra. A maximális átviteli sebességet meghatározó mérés

Bekapcsolt port-státusz pollozással elvégezve ezt a mérést láthatóvá válik, hogy az egyes link-hiba detekció megvalósítások mennyire erőforrás igényesek. Mint ahogyan a 18. ábrán látható, mind a portd mind az swconfig alapú pollozási eljárás egészen 20 milliszekundumig a maximális átviteli sebességet kevesebb, mint öt százalékos hibával közelíti. Ez alatti érték alatt a portd teljesítménye lassan ereszkedik, majd 3 milliszekundum alatt meredeken esik. Ezzel szemben az swconfigos pollozás 5 milliszekundumig öt százaléknál alacsonyabb veszteséget okoz, viszont 3 milliszekundum alatt szintén meredeken esik. Viszont minden mérési pontban jobban teljesít a portdnél. Ez az eredmény várható is volt, hiszen a portdnél kevesebb erőforrást használ. Ez a mérés eredményein szépen látszódik, hiszen a mérés során a rendszer terhelése végig magas volt, így az erre rakodó kisebb többletterhelés nagyobb átviteli sebességet tesz lehetővé.

A maximális átviteli sebesség ismeretére szükség van linkvesztés detektálá-



18. ábra. Portállapot pollozási eljárások hatása az átviteli sebességre

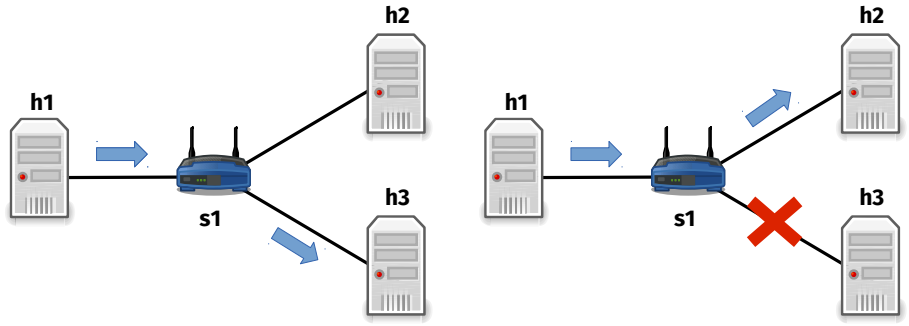
si idő méréséhez. Ugyanis ilyenkor ha ennél a sebességnél gyorsabban küldünk csomagokat, úgy azok a link-hibától függetlenül eldobásra kerülnek. Viszont ha nagyon alacsony rátával érkeznek a csomagok, a mérés pontatlanná válik. A linkvesztési idő az (1) képlet alapján adódik. Az eldobott csomagok száma a csomag sorszámozásból adódik. A sávszélesség megadható konstans. A kiszámolás nehézségét a csomagméret pontos meghatározása jelenti. Ugyanis a mérés során használt iperf programban az UDP datagram mérete és az átvitel sávszélessége adható meg. Viszont az átvitel során az UDP csomag többször becsomagolódik, így mérete megváltozik. Ha túl nagy az UDP datagram mérete, akkor előfordulhat, hogy az átvitel során több részre darabolódik. Ezért fontos, hogy a mérés során megfelelő méretű UDP csomagot használjunk. Alapértelmezetten az iperf 1470 byte méretű UDP csomagokat küld. Ez egy alsó korlátot jelent a csomagoknak. Felső korlátot a legalsó hálózati réteg maximális keretmérete jelent. Ethernet esetén a megengedett maximális kerethossz 1518 byte [9]. Ha link-hiba során elveszett fél csomagok elenyésző hányada miatt nem foglalkozunk, úgy a csomagok valós mérete az iperfnek megadott datagram méret és a maximális Ethernet kerethossz közé esik, ha a datagram belefér egy keretbe.

$$T_{link-vesztesdetektalasi} = N_{eldobottcsomagok}/(B_{savszelesseg}/S_{csomag}) \qquad (1)$$

A 19. ábrán látható a linkvesztés detektálási idejét meghatározó mérés folya-

[9] Az IEEE802.3-2012 1. szekció 4.2.7.1-es bekezdése alapján.

mata az OpenFlow kontroller ábrázolása nélkül. A mérés során a (h1) jelű végpont küld 25 Mbits/sec sebességgel 1470 byte-os UDP datagramokat először a (h3) jelű végpontnak, majd egy véletlen időpontban megszüntetjük az (s1)–(h3) összeköttetést. Mivel az OpenFlow kapcsolónak van ilyen helyzetre folyamtábla bejegyzése, így a kontrollerrel való kommunikáció nélkül azonnal az (s1)–(h2) előre beállított, úgynevezett failover linken folytatódik a csomagok küldése.



19. ábra. A linkvesztés detektálási idő mérése

A mérés során a (h2),(h3) jelű végpont tcpdumppal rögzíti a beérkező csomagok első 64 byte-ját. Azért nincs többre szükség, mert ebben található a csomagok sorszáma. A rögzített forgalom feldolgozását az alábbi parancssori parancs végzi el:

```
(echo 'ibase=16'; tcpdump -s 64 -r /tmp/meres.cap -x -n port
5001|grep 0x0010|cut -f10 -d" "|tr a-z A-Z)| bc > /tmp/meres.csv.
```

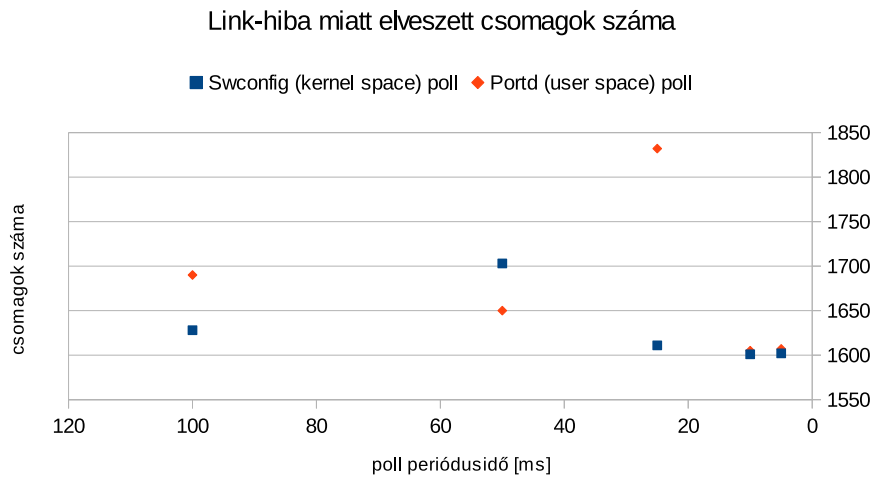
Működésének lényege, hogy kigyűjti a rögzített forgalomból az iperf által használt porton érkező csomagokat, majd azok sorszámmezőjét tízes számrendszerbe konvertálva egy csv fájlba írja.

Ezt követően az adatok feldolgozásával előállnak az egyes esetekben az elveszett csomagok száma. Különböző pollozási időközökhöz tartozó néhány véletlen kísérlet eredményeként előálló link-hiba miatt elveszett csomagok számát jeleníti meg a 20. ábra. Látható, hogy 1600 és 1850 darab között mozgott az elvesztett csomagok száma, függetlenül az alkalmazott link-hiba detekciós eljárástól. Ebből következik, hogy a kapcsoló nyújtotta gyakorlati link-hiba detekciós idő minimuma e tartományban található, mely alulról korlátozza a csomagvesztések számát.

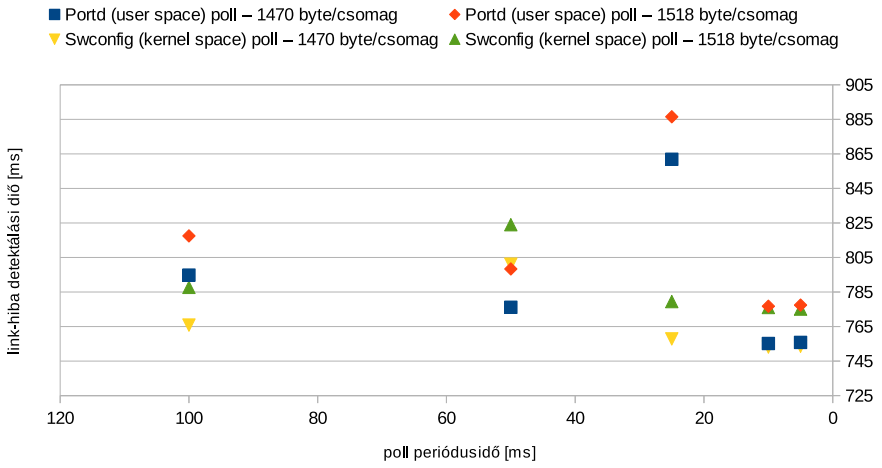
Mivel a linkvesztés detektálási időt pontosan nehéz megadni, de alsó- illetve felső becsléssel közelíthető. Az UDP datagram mérete 1470 byte volt, így ez a csomagméret alsó korlátjaként megfelelő. Felső korlát az Ethernet keret maximális mérete, 1518 byte. A linkvesztési idő számítása az eldobott csomagok számának alapján az (1) képletből következik. Ebbe a képletbe a véletlen kísérletek eredményeivel történő behelyettesítésével áll elő a 21. ábra. Látható, hogy a linkvesztési

idő függetlenül a pollozási eljárástól 750–900 milliszekundum között mozog. A két korlátból következik, hogy a szórás maximális mértéke 22 milliszekundum.

Tehát a mérések alapján kijelenthető, hogy a hálózati kapcsoló linkhiba detekciós képességére épülő eljárások az optikai hálózatokon bevett 50 milliszekundumos határtól messze elmaradnak, ugyanakkor az LLDP-nél, illetve OFDP-nél nagyságrendekkel hatékonyabbak, hiszen azok időzítőinek granualitása alatt működnek.



20. ábra. Elveszett csomagok száma különböző pollozási időközzel



21. ábra. Linkvesztés detektálási idők különböző pollozási időközökben

4. Összegzés

A dolgozatban bemutatott link-hiba detekciós eljárásról mérésekkel alátámasztottam, hogy hatékonyabb a meglévő LLDP/OFDP alapú megoldásoknál, azok átlagos időzítőfelbontása alatt megtörténik a link-hiba esemény feldolgozása. Ugyanis még a mérés során használt, a 16. ábrán látható, egyszerű topológiában is legalább két LLDP ciklus alatt történik meg a link-hiba detektálása. Ugyanakkor bebizonyosodott, hogy a bemutatott link-hiba detekciós eljárás a létező, hasonló elvet követő, port-státusz lekérdező megoldásokhoz képest link-hiba detekciós időben nem hoz jelentős változást, ugyanúgy a hardver nyújtotta minimum közelében mozog. Viszont azoknál hatékonyabb, így alkalmazása nagyobb áteresztő képességet tesz lehetővé. Továbbá hangsúlyoznám, hogy szabványos netlink üzenetek használata előnyt jelenet felhasználói szemszögből, ugyanis ezáltal könnyebben illeszthető meglévő és fejlesztés alatt álló programokhoz.

Összegezve tehát dolgozatommal azt próbáltam bizonyítani, hogy az ismertetett link-hiba detekciós eljárás nem csak hatékonysága, de könnyű alkalmazhatósága miatt is előnyös. Viszont a link-hiba detekciós ideje valószínűleg elmarad a BFD által elérhetőtől. Ennek bizonyítása további kutatási lehetőségként felveti a BFD protokoll implementálását, illetve alkalmazását az alsó kategóriás hálózati útvonalválasztókon.

5. Irodalom, és csatlakozó dokumentumok jegyzéke

5.1. A tanulmányozott irodalom jegyzéke

[1] János Farkas, Csaba Antal, Lars Westberg, Alberto Paradisi, Tania Regina Tronco, Vinicius Garcia de Oliveira,
Fast Failure Handling in Ethernet Networks,
Communications, 2006. ICC '06. IEEE International Conference, 2006.

[2] Bin Wu, Pin-Han Ho, Kwan L. Yeung,
Monitoring Trail: On Fast Link Failure Localization in All-Optical WDM Mesh Networks,
JOURNAL OF LIGHTWAVE TECHNOLOGY, VOL. 27, NO. 18, SEPTEMBER 15, 2009

[3] János Tapolcai, Bin Wu, Pin-Han Ho,
On Monitoring and Failure Localization in Mesh All-Optical Networks,
INFOCOM 2009, IEEE, 2009.

[4] LAN/MAN Standards Committee of the IEEE Computer Society,
IEEE Std 802.1AB-2009 (Revision of IEEE Std 802.1AB-2005), IEEE Standard for Local and metropolitan area networks–Station and Media Access Control Connectivity Discovery,
`http://standards.ieee.org/getieee802/download/802.1AB-2009.pdf`,
2013. 10. 18.

[5] J. Williams,
OpenFlow Discovery Protocol and Link Layer Discovery Protocol,
`http://groups.geni.net/geni/wiki/OpenFlowDiscoveryProtocol`,
2013. 10. 18.

[6] D. Katz, D. Ward, Juniper Networks,
RFC 5880 - Bidirectional Forwarding Detection (BFD),
`http://tools.ietf.org/html/rfc5880`,
2013. 10. 03.

[7] D. Katz, D. Ward, Juniper Networks,
RFC 5881 - Bidirectional Forwarding Detection (BFD) for IPv4 and IPv6 (Single Hop),
`http://tools.ietf.org/html/rfc5881`,
2013. 10. 03

[8] J. Salim, H. Khosravi, A. Kleen, A. Kuznetsov,
RFC 3549 - Linux Netlink as an IP Services Protocol,
http://tools.ietf.org/html/rfc3549,
2013. 05. 01.

[9] Kevin Kaichuan He,
Kernel Korner - Why and How to Use Netlink Socket,
http://www.linuxjournal.com/article/7356?page=0,0,
2013. 04. 30.

[10] Asanga Udugama,
Manipulating the Networking Environment Using RTNETLINK
http://www.linuxjournal.com/article/8498,
2013. 10. 05.

[11] Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini,
Linux Device Drivers, 3rd Edition,
O'Reilly, 2005. február.

[12] *Linux Kernel Documentation - Platform Devices and Drivers*,
https://www.kernel.org/doc/Documentation/
driver-model/platform.txt,
2013. 10. 11.

[13] *RTL8366/RTL8369 6/9-PORT 10/100/1000MBPS SWITCH CONTROLLER DATASHEET*,
http://realtek.info/pdf/rtl8366_8369_datasheet_
1-1.pdf,
2013. 04. 30.

[14] M. Tim Jones,
Kernel APIs, Part 3: Timers and lists in the 2.6 kernel,
http://www.ibm.com/developerworks/linux/library/
l-timers-list/,
2013. 11. 20.

[15] *TP-Link TL-WR1043ND - OpenWrt Wiki*,
http://wiki.openwrt.org/toh/tp-link/tl-wr1043nd,
2013. 04. 30.

[16] *OpenWrt Buildroot*,
http://wiki.openwrt.org/about/toolchain,
2013. 10. 11.

5.2. A csatlakozó dokumentumok jegyzéke

Az OpenWrt-s swconfigon illetve hálózati kapcsolókat leíró fejlécállományon eszközölt módosításaimat az alábbi patchek tartalmazzák:

- `http://centaur.sch.bme.hu/~leait/projects/openwrt/patches/swconfig_poll.patch`
- `http://centaur.sch.bme.hu/~leait/projects/openwrt/patches/switch_pollstate.patch`

Ezenfelül elérhetővé tettem a fejlesztés során használt firmware image-t létrehozó OpenWrt Buildroot környezetet. Ebben megtalálhatóak az előbb említett patchek, továbbá beállítások, tesztelésre használt alkalmazások. Ez elérhető a következő helyeken:

- `http://centaur.sch.bme.hu/~leait/projects/openwrt/OpenWRT_buildroot.tar.gz`