Implementacija TCP-a u jezgri Linuxa i njene primjene

Tomasović, Ana

Sveučilište u Rijeci – Odjel za informatiku

Diplomski sveučilišni studij informacijski i komunikacijski sustavi

Ana Tomasović

Implementacija TCPa u Linux jezgri i njene primjene

Diplomski rad

Mentor: dr. sc. Vedran Miletić

Rijeka, 10. studenog 2018. godine

Sažetak

TCP je protokol transportne razine TCP/IP mrežnog modela. Zadatak protokola na transportnoj razini jest osigurati prijenos podataka između dviju aplikacija na udaljenim računalima. Zadatak svakog operacijskog sustava jest implementirati TCP stog po specifikaciji samog protokola, kako bi računalo bilo u mogućnosti komunicirati s ostalim umreženim računalima. U ovom radu sam se osvrnula na implementaciju TCP protokola u Linux jezgri.

Linux jezgra napisana je u C programskom jeziku. Implementacija mrežnih funkcija nalazi se velikom većinom unutar net biblioteke. U mrežnom kodu, često se koriste hash tablice (u koje se primjerice, pohranjuju podaci o susjednim protokolima), uvjetne naredbe kompajleru (kojima se provjerava prisutnost nekog uređaja u računalu) te pokazivači na funkcije.

Glavna programska struktura mrežnog koda jezgre jest socket buffer, odnosno, sk_buff. Ona sadrži mrežne pakete koji se trenutno procesiraju te njihove upravljačke informacije. Funkcija bilo kojeg mrežnog sloja u Linux jezgri radi sa sk_buff tipovima podataka.

Samu implementaciju TCP stoga možemo podijeliti na nekoliko glavnih dijelova: zaglavlja, socketi, trostruko rukovanje, upravljanje vezama, upravljanje istekom vremena, mehanizam pomičnih prozora te mehanizam kontrole zagušenja.

Naposlijetku, spomenute su najznačajnije primjene TCP stoga u Linux jezgri, koje većina korisnika susreće svakodnevno.

Ključne riječi

TCP, funkcija, mreža, Linux jezgra, socket (utičnica), zaglavlje, veza, tok podataka, zagušenje, prozor

Sadržaj

1 Uvod

Jedna od najistaknutijih karakteristika suvremenog društva jest potreba za povezanošću kroz tehnologiju. Osim svakodnevne komunikacije, sve se više aspekata poslovanja oslanja na dostupnost Internet veze. Poslovne aplikacije primarno zahtijevaju stabilnu komunikaciju, koja jamči slanje i primanje poruka, no istovremeno dostavlja podatke u realnom vremenu.

TCP je protokol koji ispunjava postavljene uvjete. Implementira razne algoritme i mehanizme koji osiguravaju stabilnost i maksimalnu iskorištenost veze uz izbjegavanje latencije i zagušenja.

Implementaciju TCP protokola i spomenutih mehanizama koje koristi, navesti ću u ovom radu te proučiti njihovu implementaciju u Linux jezgri. Linux jezgru koristi GNU/Linux operacijski sustav, poznat po širokoj upotrebi na serverima. Serveri su specifični po mrežnim zahtjevima stoga što procesiraju vrlo veliku količinu podataka paralelno, opslužujući često i više od nekoliko tisuća korisnika istovremeno. Kako bi se posluživanje tolike količine podataka moglo ostvariti, mrežni stog mora biti optimalno implementiran te konfigurabilan ovisno o zahtjevima i korisnicima.

Upravo implementaciju tog stoga istražiti ću u sljedećim poglavljima te navesti najčešće korištene algoritme, opcije i konfiguracije TCP-a unutar Linux jezgre. Naposlijetku, navesti ću primjere specifičnih aplikacija koje koriste implementirane mehanizme.

2 TCP/IP model

Mrežni model je skup dokumenata koji cjelovito utvrđuju i objašnjavaju sve elemente potrebne da bi računalna mreža funkcionirala. Dio dokumenata definira fizička svojstva mreže: računalno sklopovlje, kablove, žice te dedicirane mrežne elemente kao što su preklopnici i usmjerivači. Objašnjava se i algoritamski i softverski dio mreže: specifikacije protokola na svakom sloju modela, odnosno, logička pravila komunikacije računala. Cilj mrežnih modela jest izgraditi sustav pravila koje proizvođači računalne opreme moraju slijediti kako bi računalo bilo u mogućnosti komunicirati sa drugim računalima koja slijede istu specifikaciju. Prednost mrežnih modela jest u činjenici da definiraju skup pravila, no ne i implementaciju, stoga svaki proizvođač može izvesti implementaciju na svojstven način, dok god slijedi specifikaciju mrežnog modela. ISO OSI i TCP/IP modeli su definirani kao hijerarhija sekvencijalnih slojeva, od kojih svaki izvršava specifičan skup zadaća. Prednost slojevitog pristupa je modularnost: kompleksna mrežna komunikacija je podijeljena u manje, jednostavnije segmente. Svaki od slojeva je fokusiran na vlastitu zadaću te se oslanja na korektan rad slojeva o kojima njegova funkcionalnost ovisi. ISO OSI model definira sedam slojeva: fizički, podatkovni, mrežni, transportni, sloj sesije, prezentacijski i aplikacijski.

TCP/IP model definira četiri sloja: podatkovni, mrežni, transportni i aplikacijski. ISO OSI model je stariji, te uključuje slojeve koji se u suvremenim mrežnim implementacijama ne smatraju relevantnima, stoga će fokus ovog poglavlja biti na TCP/IP modelu. TCP/IP model je igrađen oko dva temeljna susjedna protokola: IP i TCP, koji su okosnica suvremene mrežne komunikacije putem Interneta. Komunikacija putem TCP/IP protokola je za samog korisnika transparentna, a za implementaciju je zadužen proizvođač, odnosno, programer operacijskog sustava. Krajnji cilj je postići da transportni sloj korektno komunicira s mrežnim slojem, te pouzdano izvršava naputke aplikacije.

2.1 Internet protokol

Internet protokol definira pravila slanja datagrama od točke do točke (eng. point-to-point), te ne obećava zajamčenu dostavu podataka. Datagram je sačinjen od podataka koje želimo poslati kroz mrežu primatelju. Primatelj je definiran u zaglavlju datagrama, skupa sa pošiljateljem, kako bi se znalo gdje slati odgovor. Datagrami se kroz mrežu usmjeravaju od jednog do drugog usmjerivača, sve dok adresa primatelja nije u lokalnoj mreži jednog od usmjerivača. Adresa koja se na ovom sloju koristi jest IP adresa - niz od 32 bita koji jedinstveno identificiraju primatelja u mrežnom sloju modela.

2.2 TCP protokol

TCP uspostavlju konekciju od pošiljatelja do primatelja (eng. end-to-end). Podatke dobivene od aplikacijskog sloja oblikuje u TCP segment koji ujedno sadrži i metapodatke u zaglavlju. S obzirom da se TCP protokol oslanja na mrežni sloj za dostavu podataka od točke do točke, pouzdanost se mora implementirati na transportnom sloju.

Mehanizmi pouzdanog prijenosa i njihova implementacija su opisani u poglavlju Implementacija TCP protokola . TCP komunikacija se uspostavlja trostrukim rukovanjem (eng. three-way handshake) u kojem se određuju osnovni parametri konekcije. TCP koristi port broj, osim IP adrese, kako bi se osim samog računala jedinstveno mogla identificirati i aplikacija kojoj je potrebno dostaviti podatke.

3 Programsko sučelje Linux jezgre

U ovom poglavlju ukratko su opisani programski koncepti često korišteni u implementaciji mrežnih funkcionalnosti Linux jezgre.

3.1 Predmemorija i hash tablice

Ukoliko se od neke strukture očekuje česta alokacija i dealokacija memorije, običaj je da inicijalizacijska rutina te strukture alocira dio memorije koji će se koristit kao predmemorija (eng. *cache*). Jednom kada je predmemorija alocirana, iz nje se rade daljnje alokacije te u nju vraćaju dealocirani, prethodno korišteni blokovi. Ukoliko inicijalizacijska rutina ne bi unaprijed alocirala dediciranu predmemoriju, za svaku novu strukturu bi bilo potrebno kontaktirati jezgru, što bi uzrokovalo značajnu latenciju. Za pojedine često korištene mrežne strukture, predmemoriju ne održava inicijalizacijska rutina već sama jezgra. Primjeri takvih struktura su sk_buff, koju koriste svi mrežni redovi (eng. *queue*) i međuspremnici (eng. *buffer*) te tablica mapiranja susjednih mrežnih protokola (eng. n*eighbouring table*, poznatija kao ARP cache za IPv4). Za održavanje tih predmemorija, jezgra pruža određene metode, kao što su kmem_cache_alloc ili kmem_cache_destroy. Te metode se pozivaju iz dediciranih podsustava, kao što je međuspremnik utičnica (eng. *socket buffer)* ili routing tablice, iz metoda nazvanih "omotači" (eng. *wrapper*). Primjerice, zahtjev za oslobađanjem instance sk_buff strukture izvršava se pozivanjem kfree_skb metode, koja u suštini poziva jezgrinu kmem_cache_free metodu. Međutim, metoda omotač služi i tome da se očiste i reference na objekt nad kojim je pozvana kako bi se osiguralo da jezgrina direktna metoda ne izazove neželjene posljedice oslobađanja referencirane memorije. Predmemorija se često implementira pomoću hash tablica. Za gradnju hash tablica se mogu koristiti strukture predefinirane u jezgri, primjerice, jednosmjerne ili dvosmjerne liste. Primjer upotrebe hash tablice je ARP predmemorija, odnosno, tablica koja sadržava adrese sa podatkovnog sloja mapirane na odgovarajuće adrese sa trećeg sloja.

3.2 Reference

Kako bi se izbjegle situacije u kojima proces pokušava pristupiti prethodno oslobođenom dijelu memorije, svaki podsustav mora implementirati brojač referenci. Brojač referenci se povećava i smanjuje sukladno sa instaciranjem struktura i njihovom dealokacijom. Brojanje referenci se implementira u svakom podsustavu, međutim i sama jezgra izlaže dvije funkcije koje se pozivaju prilikom držanja ili oslobađanja reference. Ovisno o podsustavu, te funkcije se najčešće zovu xxx_hold i xxx_release.

Prilikom oslobađanja ili držanja reference, potrebno je unutar podsustava pozvati odgovarajuću jezgrinu funkciju kako bi osim samog podsustava i jezgra imala način

saznati koji dijelovi memorije se aktivno koriste, a koji su oslobođeni. Osim što držači referenci obavještavaju jezgru o dealokaciji, u određenim situacijama jezgra mora dealocirati strukturu koju držači još nisu otpustili. Tada se držači kroz obavijesne lance (eng. *notification chain*) obavještavaju o dealokaciji kako bi bili u mogućnosti prisilno očistiti pokazivače na strukturu u pitanju.

3.3 Čišćenje memorije

Mrežni podsustavi implementiraju mehanizme čišćenja, odnosno, dealokacije memorije koja je zauzeta od strane nekorištenih ili neaktivnih struktura. Ukoliko se nekorišteni blokovi memorije ne oslobađaju, aktivni procesi imaju sve manje slobodnih blokova na raspolaganju. Susreću se dva glavna mehaniza čišćenja memorije: sinkroni i asinkroni. Proces asinkronog čišćenja ne ovisi o vanjskim događajima te se pokreće kao odgovor na periodičku aktivnost. Ta aktivnost je najčešće implementiran u vidu brojača, koji nakon isteka zadanog vremena poziva skener podsustava koji skenira strukture trenutno prisutne u memoriji i odlučuje koja od njih se može dealocirati. Parametri odluke o dealokciji uvelike ovise o podsustavu i svrsi strukture, no čest kriterij odluke je da niti jedan pokazivač ne pokazuje na strukturu u pitanju. Sinkrono čišćenje se aktivira u slučaju kada je dostupna memorija kritično niska te proces ne može čekati na istek asinkronog brojača. U tom slučaju se mehanizam čišćenja pokrene odmah, te su kriteriji čišćenja najčešće agresivniji nego prilikom asinkronog čišćenja.

3.4 Pokazivači na funkcije

Pokazivači na funkcije su preteča objektnog modela programiranja. Struktura odgovara konceptu klase, a pokazivač na funkciju odgovara metodama koje se koriste u isključivo objektnim jezicima. Jednom kada je pokazivač na funkciju definiran unutar strukture, manipulacija te strukture i njezinih instanci se odvija isključivo kroz pokazivač na funkciju. Kao i u objektnim jezicima kod preopterećivanja metoda, pokazivači na funkciju mogu biti inicijalizirani različito, ovisno o proslijeđenim parametrima. Čest primjer toga je rad sa upravljačkim uređajima: implementirano je jedno programsko sučelje za rad sa svim uređajima, no inicijalizira se različito ovisno o konkretnom uređaju u pitanju. Također, česta struktura u kodu jezgre jest virtualna tablica fukcija (eng. V*irtual Function Table, VFT*). VFT je u suštini grupirani skup pokazivača na funkcije[1].

3.5 Uvjetne naredbe kompajleru

Uvjetne naredbe kompajleru se često koriste kako bi se provjerilo je li određena funkcionalnost podržana u jezgri. Primjerice, podsustav Netfilter zahtijeva da u instancama

sk_buff strukture bude prisutno polje nf_debug, tipa unsigned int. Naredbom kompajleru je moguće provjeriti podržava li jezgra Netfilter, te ukoliko ne, iz sk_buff strukture je moguće izostaviti ovu varijablu te tako uštediti na zauzeću memorije.

3.6 Pretvorbe težinskog zapisa bitova između mreže i računala

Težinski zapis bitova u računalu (Little ili Big Endian) ovisi o proizvođaču procesora, međutim TCP/IP stog koristi Big Endian zapis. Prilikom svake operacije sa IP zaglavljem koje je veće od jednog bajta (s obzirom da se individualni bitovi u memoriji ne mogu adresirati) ili sa poljem bitova (eng. *bitfield*), jezgra mora izvršiti pretvorbu težinskog zapisa bitova ukoliko je potrebno. U ovu svrhu su u include/linux/byteorder/generic.h biblioteci definirani makroi koji se mogu koristiti za pretvorbu, primjerice: htons, nthos.

3.7 Organizacija mrežnog programskog koda u jezgri

Biblioteka net u kodu jezgre podijeljena je na deset logički izdvojenih cjelina. Biblioteka socket.c sadrži funkcije dedicirane upravljanju socketima. Core sadrži ključne mrežne funkcije koje objedinjavaju više podsustava, primjerice: sock.c, skbuff.c, datagram.c. Biblioteka ipv4 implementira cijeli mrežni model protokola vezan uz IPv4: ARP, ICMP, TCP, UDP. Također, unutar ipv6 biblioteke implementirane su specifične funkcije za rad sa IPv6 protokolom. Netlink sadrži funkcije potrebne za upravljanje netlink utičnicama, odnosno, utičnicama pomoću kojih jezgra komunicira sa korisničkim prostorom. Packet biblioteka implementira funkcije za kreiranje, zatvaranje i upravljanje raw utičnicama, koji se koriste za primanje, slanje i procesiranje paketa koji nisu direktno podržani od strane jezgre. Sched sadrži funkcije koje upravljaju IP ToS (Type of Service) zahtjevima. ToS je polje definirano u zaglavlju IP paketa, a koristi se kako bi se naznačilo kolika razina prioriteta, odnosno, kvalitete prijenosa se treba dodjeliti paketu. Primjerice, u ToS polju se može definirati zahtjev da se paket šalje rutom koja ima najmanju latenciju. Unix biblioteka objedinjava funkcije vezane za rad sa Unix utičnicama. Naposlijetku, ethernet biblioteka sadrži specifikacije svih funkcija i varijabli specifičnih za Ethernet protokol.

3.8 Kontrolne staze jezgre za TCP/IP stog

U trenutku kada aplikacija, odnosno, proces na višoj razini treba zapisati ili pročitati podatke iz TCP socketa, mora prebaciti kontrolu nad tim operacijama jezgri preko sistemskog poziva. U tom trenutku jezgra izvršava spomenute operacije umjesto procesa.

U prvoj kontrolnoj stazi, jezgra procesira dobivene podatke kroz cjelokupni TCP stog, te kontrolu vraća procesu tek u trenutku kada su podaci fizički poslani kroz mrežnu karticu.

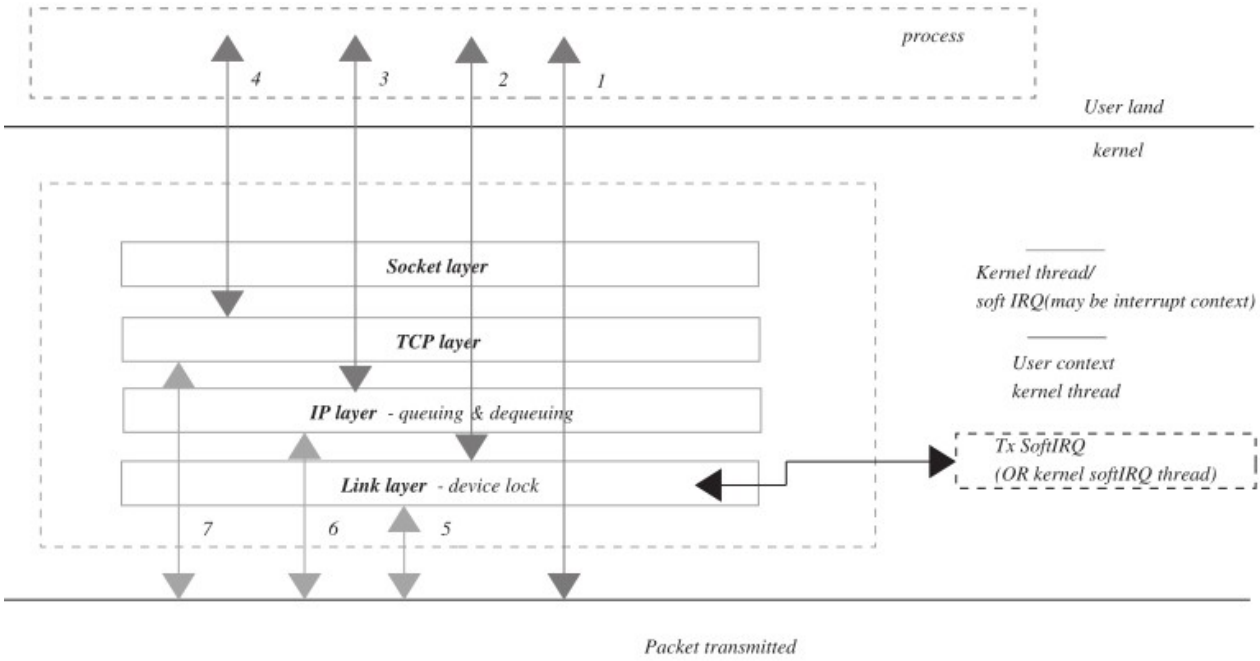
Druga kontrolna staza prolazi kroz sličan postupa - razlika je u tome što se kontrola procesu vraća prije nego su podaci fizički poslani, najčešće uzrokovano time što je mrežna kartica zauzeta i blokirana od strane drugog procesa u tom trenutku. Jezgra prije vraćanja kontrole označi Tx softIRQ, koji dolazi u red čekanja na procesiranje u kasnijem trenutku. U trećoj kontrolnoj stazi, jezgra kontrolu vraća još ranije nego u prethodne dvije - neposredno nakon što procesira TCP podatke u transportnom sloju. Procesiranje se ne može nastaviti zbog toga što određeni QoS (Quality of Service) mehanizam spriječava daljnju obradu.

U četvrtoj kontrolnoj stazi, jezgra vraća kontrolu nakon dolaska do transportnog sloja, odnosno, jezgra nije u mogućnosti procesirati paket dalje od aplikacijskog sloja. Najčešći razlog ovom jest blokiranje od strane TCP mehanizama za kontrolu zagušenja.

U petoj kontrolnoj stazi, podaci se procesiraju kao interrupt jezgre ili u regularnom kontekstu jezgre. U ovoj se stazi obrađuju podaci koji su proizašli iz softIRQ u drugoj kontrolnoj stazi, odnosno, oni podaci koji su u tom trenutku stavljeni u red čekanja. Slično stazi pet radi i šesta kontrolna staza, no razlika je u tome da šesta staza procesira podatke koji su u red čekanja stavljeni u trećoj kontrolnoj stazi. Naposlijetku, sedma kontrolna staza također procesira podatke iz reda čekanja, no u četvrtoj stazi.

Prethodno opisani koraci odnose se na slanje paketa. Prilikom primanja paketa, on se procesira u dva koraka. U prvom koraku, upravljač prekidima (eng. interrupt handler) uzima paket, stavlja ga u red čekanja te naznačuje softIRQ. U drugom koraku, softIRQ se obrađuje kasnije od strane Rx softIRQ servisa ili kao prekid jezgre.

Na slici 1 je prikazan prethodno opisan mehanizam kontrolnih staza [2].



Slika 1: Kontrolne staze za TCP u Linux jezgri

4 Socket buffer (sk_buff) struktura

Zaglavlja svakog paketa akoji se procesira spremljeni su unutar sk_buff strukture. Ona se u jezgri proslijeđuju između svih slojeva TCP/IP modela. Kao glavni zahtjev nameće se što veća brzina procesiranja. Kako bi se izbjeglo kopiranje paketa između slojeva, implementirana je solucija socket buffera. Socket buffer jest u suštini blok memorije alociran za jedan paket koji je primljen ili se treba poslati, a definiran je strukturom sk_buff iz datoteke include/linux/skbuff.h. Osnovna ideja jest da se za jedan paket alocira dedicirani blok memorije, te se na taj blok vežu pokazivači svakog sloja na njihova odgovarajuća zaglavlja.

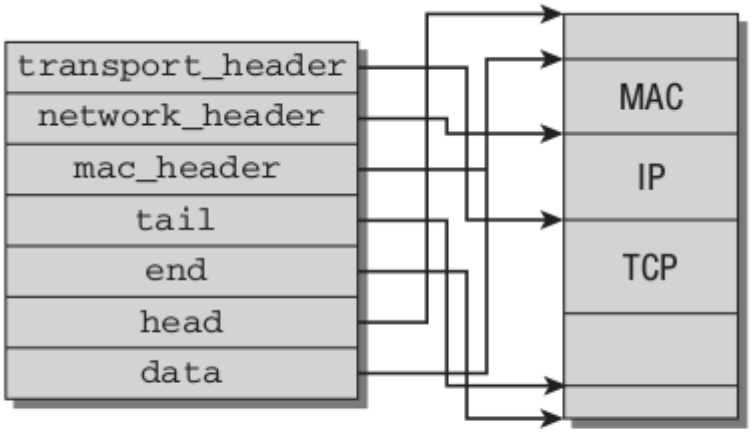


Figure 12-6: Link between socket buffer and network packet data.

Slika 2: Veza pokazivača strukture sk_buff i TCP/IP stoga

head i end pokazivači pokazuju na početak i kraj cjelokupnog paketa, dok data i tail pokazivači pokazuju na početak i kraj memorijskog bloka podataka rezerviranog za palyload određenog protokola. Mac_header, network_header i transport_header su pokazivači na odgovarajuća zaglavlja slojeva u paketu.

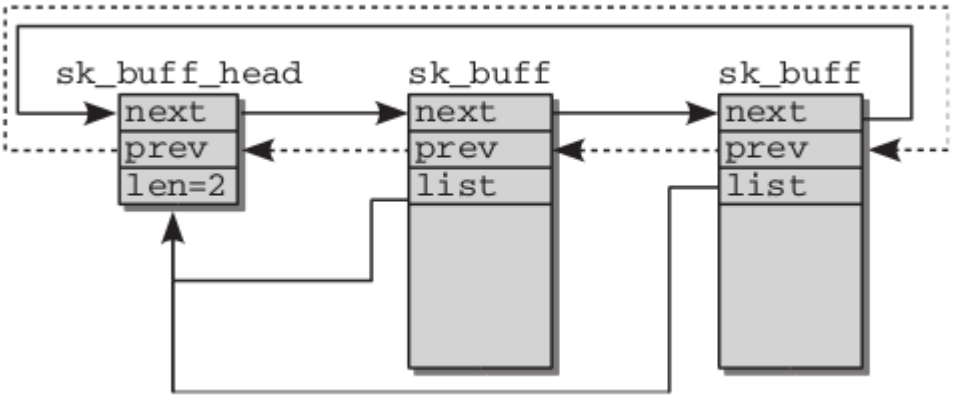
Kako sk_buff struktura može sadržavati podatke i zaglavlja bilo kojeg protokola, potrebne su funkcije koje mogu iščitati i konvertirati sadržaj za specifičan protokol. U ovu svrhu su definirane funkcije xxx_hdr (primjerice, tcp_hdr za ekstrahiranje podataka iz TCP zaglavlja). Takve pomoćne funkcije su prisutne za sve protokole koje jezgra podržava.
U trenutku kada je procesiranje na jednom sloju završeno, pokazivači data i tail se pomiču kako bi pokazivali na sljedeći sloj. Primjerice, ukoliko je u tijeku procesiranje izlaznog paketa, prvi sloj koji će obraditi paket će biti aplikacijski sloj. Prva radnja tog sloja je pozvati skb_reserve kako bi se rezervirala memorija potrebna za dodavanje zaglavlja. Nakon alokacije memorije, sloj dodaje potrebno zaglavlje. U trenutku kada je obrada na danom sloju završena, pokazivač data se pomiče iznad, na početak TCP ili UDP zaglavlja.

Ukoliko se procesiranje odvija obratno (obrada ulaznog paketa), pokazivač data se također pomiče, no u suprotnom smjeru, što je prikazano na slici 2.
Upotrebom pokazivača, uštađeno je vrijeme na operacijama kopiranja ili brisanja, što bi zahtjevalo veću količinu procesorskog vremena i ciklusa.

4.1 Upravljački podaci u sk_buff strukturi

Neki od upravljačkih podataka unutar sk_buff strukture su tstamp (vrijeme primitka paketa), dev (specificira trenutni mrežni uređaj na kojem se obrađuje paket) te sk (specificira koja instanca utičnice se koristi za obradu paketa).

U danom trenutku, na računalu se nalazi više od jedne sk_buff instance. Kako bi se svaki paket mogao korektno obraditi (posebice prilikom preslagivanja pristiglih paketa), potrebno je implementirati redove čekanja. Struktura podataka korištena u tu svrhu jest dvostruko povezana lista. Dodatan zahtjev za povezanu listu sk_buff instanci u jezgri jest da svaka od instanci može brzo pronaći glavu. U tu svrhu koristi se struktura sk_buff_head, koja se uvijek umeće na početak liste. Struktura sk_buff_head se koristi kao glava povezane liste, sadrži pokazivače na prethodnu i sljedeću sk_buff instancu te varijablu u koju je spremljena ukupna duljina liste.



Slika 3: Upravljački mehanizam strukture sk_buff

4.2 Alociranje memorije: alloc_skb i dev_alloc_skb

Kako sk_buff struktura u sebi sprema zaglavlja paketa, spremnik za teret (eng. *payload*) potrebno je alocirati zasebno, odnosno, za svaki primljeni paket, jezgra ivršava dvije alokacije memorije.

Funkcija alloc_skb uzima pripremljenu sk_buff strukturu iz dediciranog međuspremnika pozivajući kmem_cache_alloc u procesu. Spremnik tereta alocira se pozivom kmalloc funkcije, koja u tu svrhu također koristi memoriju međuspremnika, ukoliko je dostupna.

Funkcija dev_alloc_skb je omotač oko alloc_skb funkcije. Namjena je da se izvršava kao prekid trenutnog rada jezgre (eng. interrupt). Upotrebljavaju je upravljački programi raznih mrežnih uređaja. Dodatno, dev_alloc_skb zahtijeva garanciju atomične operacije upravo iz razloga što se koristi u prekidnom načinu rada.

4.3 Oslobađanje memorije: kfree_skb i dev_kfree_skb

Pozivom funkcije kfree_skb, oslobađa se alociranu strukturu sk_buff te nakon oslobađanja vraća taj dio memorije u dijeljeni međuspremnik za daljnje alokacije. Funkciju dev_kfree_skb također koriste upravljački uređaji, te je ona kao i u slučaju alokacije, samo omotač oko kfree_skb funkcije. Za razliku od dev_alloc_skv, dev_kfree_skb ne zahtijeva atomičnost te se ne izvršava kao prekid jezgre.

Funkcija kfree_skb će otpustiti memoriju isključivo ukoliko ona više nije referencirana i korištena od strane drugih struktura, odnosno, ukoliko je njezin brojač referenci jednak jedan (koristi se u trenutnom pozivu kfree_skb).

Ukoliko se kfree_skb pozove nad instancom sk_buff koja je još uvijek referencirana, smanjiti će njezin brojač referenci za jedan.

5 Implementacija TCP protokola

5.1 Upravljanje vezama

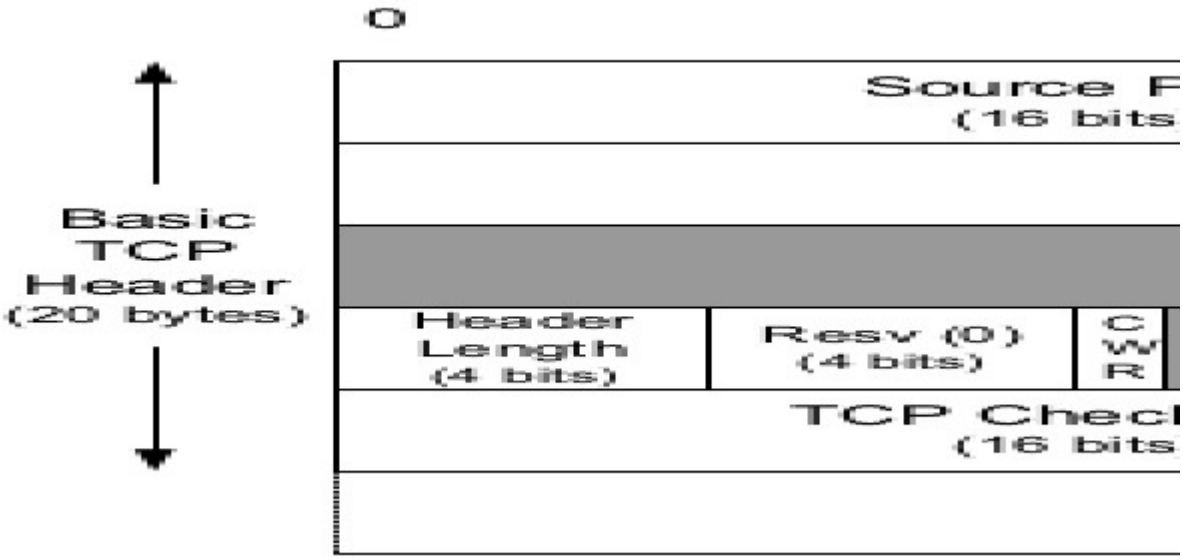
TCP protokol pruža pouzdan prijenos tokova podataka aplikacijama na dva različita računala. Za razliku od UDP-a, TCP je protokol orijentiran vezi, što označava kako prije slanja podataka, veza između dvije strane koje sudjeluju u komunikaciji mora biti eksplicitno uspostavljena. Komunikaciju putem TCP-a mogu ostvariti isključivo dva računala, direktno izmjenjujući podatke, koji će u daljnjem tekstu biti označeni kao pošiljatelj i primatelj. Koncept prijenosa podataka na više računala (eng. *multicast*) ili na sva računala u mreži (eng. *broadcast*) nije primjenjiv na TCP.

Uz TCP protokol se veže pojam tokovoa podataka, što je i ranije napomenuto. Taj izraz govori kako TCP protokol ne postavlja granice ili okvire na podatke koje šalje. U potpunosti je prepušteno aplikacijskom sloju i klijentskoj aplikaciji dainterpretira podatke. Primjerice, ukoliko aplikacija na strani pošiljatelja podatke zapisuje u slijedu od deset bajtova, dvadeset bajtova te potom pedeset bajtova, nema načina da aplikacija na strani primatelja sazna taj redoslijed, ona interpretira primljenih osamdeset bajtova neovisno o redoslijedu slanja.

Pouzdanost TCP protokola proizlazi iz mehanizama i algoritama koje koristi za prijenos podataka. Primjerice, TCP tokove podatak koje dolaze od aplikacije, pretvara u pakete (eng. packetization) označene rednim brojem i enkapsulirane unutar IP paketa. Upotreba rednih brojeva dopušta da se paketi šalju bilo kojim redoslijedom, a da primatelj i dalje može sastaviti točan slijed. TCP veličinu paketa određuje na način da se izbjegne fragmentacija na ruti do pošiljatelja. Dodatan mehanizam stabilnosti i autentikacije pošiljatelja jest CRC kontrolna suma (eng. *checksum*) koja se proslijeđuje u zaglavlju paketa. Ukoliko primatelj primi segment koji nema važeću sumu, odbacuje ga.

TCP protokol je dvostran i simetričan (eng. *full duplex*) što znači da pošiljatelj i primatelj istovremeno moglu slati i primati podatke jednakom kvalitetom. Jednom kada je veza uspostavljena, segmenti koji se šalju iz jednog smjera sadrže i ACK zastavice za pakete iz drugog smjera veze.

5.1.1 Zaglavlja



Slika 4: Zaglavlja TCP paketa

TCP paket je enkapsuliran unutar IP paketa, odnosno, TCP paket čini teret IP paketa. Svaki TCP paket sadrži zaglavlja u kojima se nalaze informacije potrebne za točno slanje paketa. Osim izvorišnog porta, odredišnog porta i tereta paketa, definiran je redni broj, pomak, rezervirani bajtovi, zastavice, prozor, kontrolni zbroj te opcije [3]. Redni broj označava koji je paket po redu od svih poslanih paketa. Ovo polje je važno kada se paketi izgube, ali i kako bi se na odredištu mogla složiti poslana poruka. Zastavice mogu poprimiti nekoliko vrijednosti, od kojih svaka signalizira ulazak u novo stanje veze između klijenta i servera: ACK, RST, FIN, SYN, PSH, CWR, ECE i URG. Primjerice, SYN implicira zahtjev za konekcijom, dok FIN završava konekciju. Pomak u zaglavlju označava ukupnu duljinu TCP zaglavlja. Prozor označava broj bajtova koje pošiljatelj može poslati bez da napuni međuspremnik klijenta. Jednom kada je buffer pun, svi nadolazeći paketi se ispuštaju. Kod sporih primatelja, ovaj mehanizam spriječava ponavljanje slanja velikog broja paketa.

TCP zaglavlje je implementirano kao tcphdr struktura podataka. U standardnoj GNU/Linux instalacija, datoteka tcp.h se nalazi na lokaciji: /usr/include/linux/tcp.h.

```
struct tcphdr {
        __be16  source;
        __be16  dest;
        __be32  seq;
        __be32  ack_seq;
#if defined(__LITTLE_ENDIAN_BITFIELD)
        __u16   res1:4,
```

```
                doff:4,
                fin:1,
                syn:1,
                rst:1,
                psh:1,
                ack:1,
                urg:1,
                ece:1,
                cwr:1;
#elif defined(__BIG_ENDIAN_BITFIELD)
        __u16   doff:4,
                res1:4,
                cwr:1,
                ece:1,
                urg:1,
                ack:1,
                psh:1,
                rst:1,
                syn:1,
                fin:1;
#else
#error  "Adjust your <asm/byteorder.h> defines"
#endif
        __be16  window;
        __sum16 check;
        __be16  urg_ptr;
};
```

5.1.2 TCP utičnice

Utičnice (eng. *socket*) je reprezentacija veze između dvije aplikacije. Može se odnositi na internu komunikaciju, na istom čvoru, ili preko mreže. Utičnica je definirana trima parametrima: portom, IP adresom i protokolom koji ga koristi. Svaki TCP socket je prisutan

u jednoj od sljedeće tri hash tablice: socketi koji trenutno održavaju vezu, socketi koji čekaju na vezu (u stanju slušanja, odnosno eng. listen), te socketi koji su trenutno u procesu uspostavljanja konekcije (prolaze proces trostrukog rukovanja).

Logika socketa definirana je u datoteci net/ipv4/tcp_input.c.

U trenutku kada jezgra primi paket, potrebno ga je upariti sa vezom kojoj pripada. U ovu svrhu se koriste hash tablice. Hash tablice su definirane u /include/net/inet_hashtables.h datoteci.

Primjerice, definicija veličine hash tablice za sockete u stanju slušanja jest:

```
/* This is for listening sockets, thus all sockets which possess wildcards. */
#define INET_LHTABLE_SIZE    32
```

Unutar datoteke /proc/net/tcp, moguće je vidjeti otvorene TCP sockete, te njihovo stanje. 0A označava socket u stanju slušanja, a 01 u stanju ostvarene veze.

5.1.3 Trostruko rukovanje

TCP je protokol koji zahtijeva eksplicitnu uspostavu konekcije prije nego što započne sa slanjem podataka (eng. *connection oriented*). TCP vezom se smatra uređena četvorka koja se sastoji od dvije IP adrese, pošiljatelja i primatelja, te dva broja koja označavaju sockete na koje je podatke potrebno slati. Veza može biti uspostavljena aktivno ili pasivno. Aktivnu uspostavu vežemo uz primatelja, odnosno, klijenta, dok pasivnu uspostavu veze odrađuje pošiljatelj, odnosno, server.

Sam proces uspostave konekcije uključuje razmjenu tri TCP paketa, a služi tome kako bi pošiljatelj i primatelj razmjenili detalje o vezi te eksplicitno naznačili da proces slanja podataka može započeti. Klijent šalje TCP paket sa oznakom SYN u polju zaglavlja dediciranom za kontrolne zastavice. Mijenja stanje socketa sa CLOSED na SYN_SENT. Također, prije slanja, klijent generira nasumičan broj koji se sprema u polje rednog broja te se koristi kao početni redni broj u vezi.

Generiranje rednog broja implementirano je u datoteci net/core/secure_seq.c.

Server prima zahtijev za konekcijom te klijentu šalje paket sa kontrolnim zastavicama SYN i ACK. Mijenja stanje socketa sa LISTEN na SYN_REC. Nakon primitka SYN/ACK paketa, socket klijenta mijenja stanje na ESTABLISHED (veza je uspostavljena) te šalje TCP paket sa kontrolnom zastavicom ACK serveru. Server popunjava polje rednog broja na način da veličina paketa doda jedan, te zbroji sa rednim brojem primljenim od klijenta. Pri primitku tog TCP paketa, server također mijenja stanje socketa u ESTABLISHED. Time je završena faza uspostave konekcije, nakon koje slijedi slanje podataka.

Proces uspostave trostrukog rukovanja definiran je u funkciji tcp_rcv_state_process u datoteci net/ipv4/tcp_input.c.

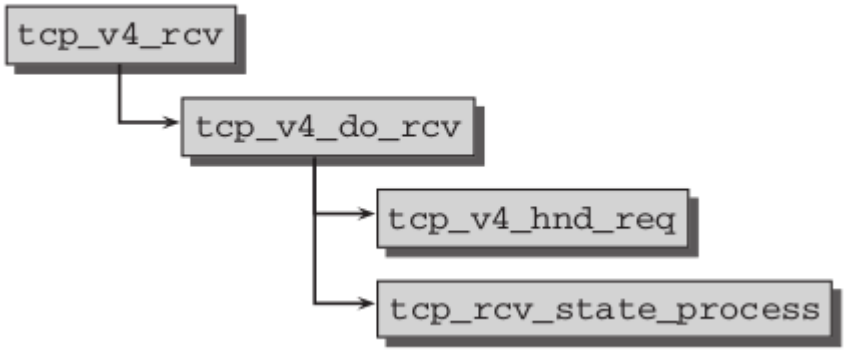
Primjerice, navedeni blok određuje slijed događaja nakon što je SYN paket sa strane klijenta poslan:

```
   case TCP_SYN_SENT:
         tp->rx_opt.saw_tstamp = 0;
         tcp_mstamp_refresh(tp);
         queued = tcp_rcv_synsent_state_process(sk, skb, th);
         if (queued >= 0)
             return queued;

         /* Do step6 onward by hand. */
         tcp_urg(sk, skb, th);
         __kfree_skb(skb);
         tcp_data_snd_check(sk);
         return 0;
}
```

5.1.4 Pasivna uspostava veze

Pasivnom uspostavom veze smatra se primitak SYN paketa od strane primatelja. Jezgra nema aktivnu ulogu, osim prilikom kreiranja socketa koji će moći prihvatiti klijentske konekcije. Ulazna točka jest funkcija tcp_v4_rcv čiji je zadatak pronaći socket u stanju slušanja za nadolazeći SYN paket. Kada je socket pronađen, kontrolu preuzima funkcija tcp_v4_do_rcv. Ona prvo poziva tcp_v4_hnd_req koja izvršava inicijalizaciju sustava potrebnu prilikom uspostave nove veze.

Prijelaz između stanja TCP veze odvija se unutar funkcije tcp_rcv_state_process. Unutar te funkcije, definirana su sva moguća stanja TCP veze te akcije koje se odvijaju ovisno o ulasku u određeno stanje.



Slika 5: Tijek pasivne uspostave veze

U datoteci include/net/tcp_states.h definirana su sva stanja u kojima se se veza, odnosno socket, može naći tijekom komunikacije:

```
enum {
    TCP_ESTABLISHED = 1,
    TCP_SYN_SENT,
    TCP_SYN_RECV,
    TCP_FIN_WAIT1,
    TCP_FIN_WAIT2,
    TCP_TIME_WAIT,
    TCP_CLOSE,
    TCP_CLOSE_WAIT,
    TCP_LAST_ACK,
    TCP_LISTEN,
    TCP_CLOSING,    /* Now a valid state */
    TCP_NEW_SYN_RECV,

    TCP_MAX_STATES    /* Leave at the end! */
};
```

Funkcija tcp_v4_conn_request se poziva ukoliko je socket u stanju TCP_LISTEN. Ona je definirana u datoteci net/ipv4/tcp_ipv4.c, a poziva se iz include/net/tcp.h, između ostalog. Definicija te funkcije glasi:

```
int tcp_v4_conn_request(struct sock *sk, struct sk_buff *skb)
{
    /* Never answer to SYNs send to broadcast or multicast */
    if (skb_rtable(skb)->rt_flags & (RTCF_BROADCAST |
RTCF_MULTICAST))
        goto drop;

    return tcp_conn_request(&tcp_request_sock_ops,
                &tcp_request_sock_ipv4_ops, sk, skb);

drop:
    tcp_listendrop(sk);
    return 0;
}
```

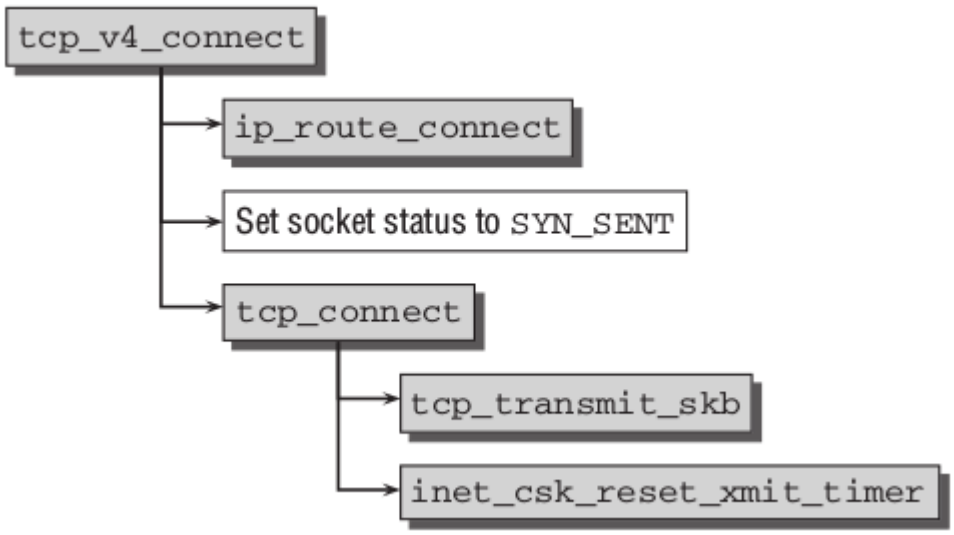
```
EXPORT_SYMBOL(tcp_v4_conn_request);
```

Iz priloženog koda možemo vidjeti kako tcp_v4_conn_request poziva tcp_conn_request, koja je definirana unutar net/ipv4/tcp_input.c.
Važno za pasivnu uspostavu konekcije jest da tcp_conn_request šalje SYN/ACK par kontrolnih zastavica. Također, unutar iste funkcije se generira redni broj na način propisan TCP protokolom i po zahtjevima trostrukog rukovanja.
Zadnje dvije radnje zaključuju pasivnu uspostavu veze sa strane servera: po primitku i procesiranju SYN paketa od klijenta, klijentu se vraća SYN/ACK.

5.1.5 Aktivna uspostava veze

Aktivna uspostava konekcije se promatra na strani klijenta. Prije svega, aplikacija u korisničkom prostoru operacijskog sustava poziva funkciju open. Funkcija open potom predaje kontrolu jezgri kako bi se izvršio sistemski poziv socketcall. Socketcall pristupa tcp_v4_connect funkciji, čiji je zadatak pozvati povezane funkcije koje pripremaju paket za slanje.

Dijagram tijeka aktivne uspostave konekcije priložen je na slici 6.



Slika 6: Tijek aktivne uspostave veze

Funkcija tcp_v4_connect prvo poziva ip_route_connect kako bi se odabrao put za usmjeravanje paketa do krajnje destinacije. U isto vrijeme se generiraju zaglavlja paketa te kopiraju u instancu sk_buff. U tom trenutku se stanje socketa promjeni u SYN_SENT. Funkcija tcp_connect šalje generirani SYN paket IP sloju, koji ga proslijeđuje klijentu. Također, u jezgri se kreira brojač koji vodi računa o tome koliko vremena je proteklo od zadnjeg primitka ACK paketa, te ukoliko se određena granica prijeđe, paket za koji nije primljen ACK se šalje iznova.

Nakon slanja SYN paketa, klijent čeka na SYN/ACK, čije slanje je opisano u prethodnom poglavlju, a za kojeg je odgovorna funkcija tcp_rcv_state_process. Ona poziva

tcp_rcv_synsent_state_process, koja postavlja status socketa na ESTABLISHED te poziva tcp_send_ack funkciju, ukoliko paket prođe sve prethodne provjere unutar funkcije: tcp_send_ack(sk);.
Funkcija tcp_rcv_synsent_state_process je definirana u datoteci include/net/ipv4/tcp_input.c.
Također, funkcija tcp_send_ack je definirana u net/ipv4/tcp_output.c. Ona se za samo slanje ACK paketa oslanja na __tcp_transmit_skb funkciju, definiranu u istoj datoteci.

5.1.6 Inicijalni redni broj

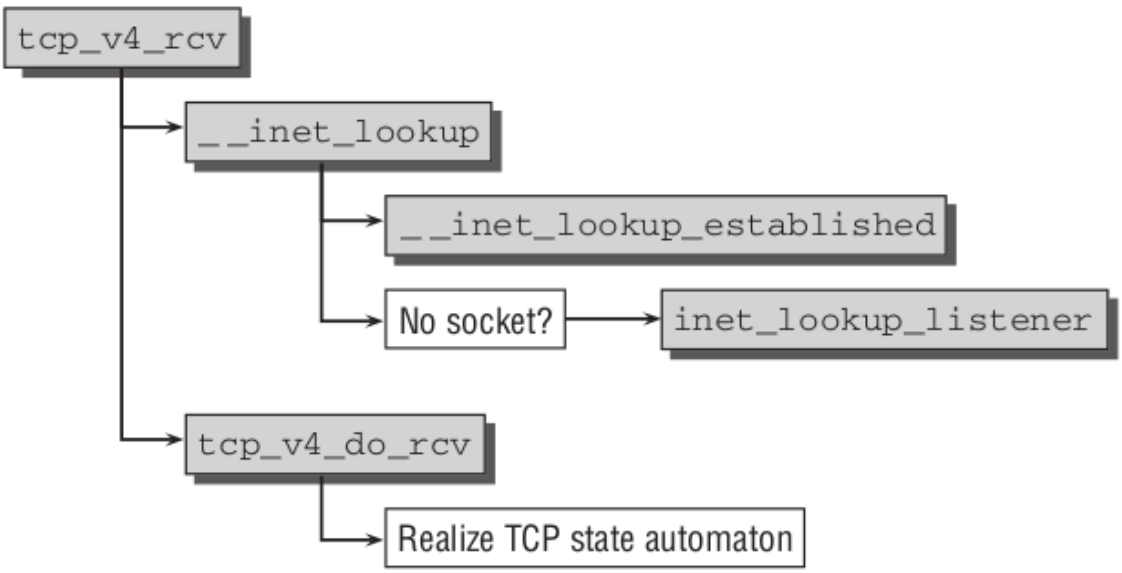
Prije nego što bilo koja strana pošalje SYN segment drugoj strani, odabire inicijalni redni broj (eng. ISN, initial sequence number) duljine 32 bita za segmente u toj vezi. Inicijalni redni broj se po specifikaciji treba mijenjati, tako da svaka nova veza dobije drugi, i to na način da se inkrementira za 1 svake 4 mikrosekunde. Za odabir inicijalnog rednog broja, Linux jezgra se oslanja na sistemski sat, uzimajući također pomak za svaku novu vezu. Pomak se računa kriptografskom funkcijom kojoj se kao argumenti proslijede identifikatori konekcije, odnosno, uređena četvorka IP adresa i socketa. Kriptografska funkcija također sadrži značku (eng. token) koja se mijenja svakih pet minuta, a služi za haširanje krajnjih rezultata. Od 32 bita u inicijalnom rednom broju, prvih osam predstavlja redni broj značke, dok je ostalih 24 bita generirano kriptografskom funkcijom.

Funkcija za generiranje inicijalnog rednog broja u Linux jezgri nazvana je tcp_v4_init_sequence, a definirana je u datoteci net/ipv4/tcp_ipv4.c. Ta funkcija poziva drugu funkciju za samo generiranje vrijednosti: secure_tcp_sequence_number.

5.1.7 Primanje TCP podataka

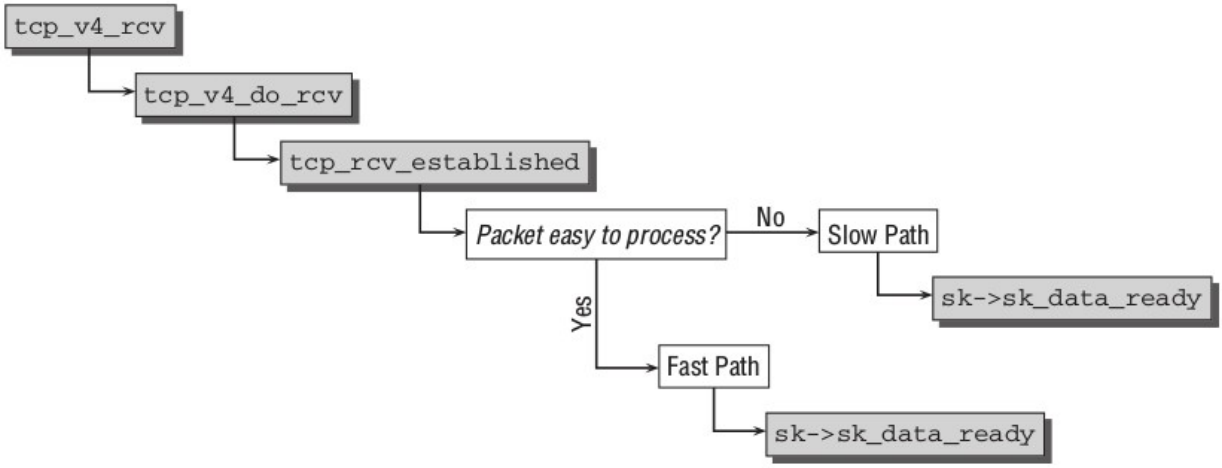
U trenutku kada je pristižući paket procesiran od strane IP sloja, šalje se prema TCP sloju. Ulazna točka za TCP sloj jest funkcija tcp_v4_rcv.

Na slici 7 je prikazan tijek paketa od ulaska do uspostave veze na socketu.



Slika 7: Tijek primanja TCP paketa

Prije dodjele socketa, jezgra nad paketom obavlja verifikacija zaglavlja, koje se naknadno kopira u instancu sk_buff. U tom trenutku, jezgra delegira kontrolu __inet_lookup funkciji čiji je zadatak pronaći socket u koji će se paket proslijediti [4]. __inet_lookup poziva druge dvije funkcije. __inet_lookup_established skenira hash tablicu socketa u stanju ostvarene konekcije, kako bi provjerila je li ovaj paket vezan za već postojeću konekciju. Ukoliko se ne nađe odgovarajući socket, __inet_lookup_established vraća negativan odgovor pozivajućoj funkciji, a __Inet_lookup potom poziva inet_lookup_listener za pretragu svih socketa koji mogu prihvatiti konekciju.
Obje funkcije koriste adresu klijenta, adresu servera, port broj i jezgrin indeks mrežnog sučelja kako bi našle odgovarajući socket za uspostavu komunikacije.
U trenutku kada je socket pronađen, tcp_v4_rcv dodjeljuje kontrolu funkciji tcp_v4_do_rcv, koja se može nazvati multiplekserom. Funckija tcp_v4_do_rcv odlučuje koji dio koda će se izvršiti ovisno o trenutnom stanju konekcije.
Ukoliko se paketi šalju unutar postojeće veze, odnosno, ukoliko je dodijeljeni TCP socket u ESTABLISHED stanju, tcp_v4_do_rcv poziva tcp_rcv_established. Ta funkcija odlučuje o programskoj putanji (spora ili brza) kroz koju će se procesiranje paketa odvijati. Brze programske putanje su implementirane za dijelove koda koji se moraju često ponavljati. Funkcija tcp_rcv_established postavlja dva neovisna zahtjeva za sve pakete kako bi bili procesirani brzom putanjom: paket mora ili sadržavati ACK kontrolnu zastavicu za podatke koji su zadnji poslani ili sadržavati podatke koji se očekuju sljedeći. Dodatno pravilo koje se primjenuje na prethodna dva jest da paket ne smije imati niti jednu od sljedećih kontrolnih zastavica: SYN, FIN, RST ili URG.S obzirom da su to najčešći paketi, smisleno je optimizirati njihovo procesiranje kroz brzu putanju, gdje se teži što brže proslijediti paket socketu. Svi ostali paketi su rjeđi te se oni procesiraju sporom putanjom. Spora putanja sadrži više dodatnih provjera, te implementira redove čekanja za pakete koji su stigli van redoslijeda.
Na slici 8 ukratko je prikazan opisani tijek.



Slika 8: Procesiranje pristiglih paketa brzom ili sporom putanjom

Brza putanja

U trenutku kada su paketi poslani na brzu putanju procesiranja, odvijaju se dodatne provjere koje će odbaciti paket i vratiti ga na sporu putanju ukoliko je kompleksniji od postavljenih zahtjeva. Ukoliko paket uspješno prođe provjere kompleksnosti, provjerava se njegova veličina. Kada je samo kontrolna zastavica postavljena, ukupna veličina paketa odgovara veličini zaglavlja. Nakon te provjere, daljnje procesiranje se delegira tcp_ack funkciji. Neki od glavnih dužnosti tcp_ack funkcije su analizirati stanje veze (primjerice, veličinu prozora) te očistiti pakete za koje je dostavljena potvrda primitka iz reda čekanja za ponovno slanje.

Ukoliko je u brzu putanju ušao paket koji sadrži podatke, a ne samo ACK kontrolnu zastavicu, on se proslijeđuje direktno socketu, odnosno, funkciji sk_data_ready, bez dodatnih provjera.

5.1.8 Slanje TCP podataka

Slanje paketa inicira aplikacijski sloj pozivom tcp_sendmsg funkcije. Kao što je opisano i u prethodnim poglavljima, stanje socketa mora biti ESTABLISHED kako bi transfer mogao započeti. Ukoliko to nije slučaj, jezgra poziva sk_stream_wait_connect.

Nakon što je dobiven socket, podaci dobiveni od aplikacijskog sloja se pakiraju u TCP paket koji se može proslijediti podređenim slojevima.

Nakon što je paket kreiran, jezgra poziva funkciju tcp_push_one, koja izvršava tri provjere nad paketom. Funkcija tcp_push_one poziva tcp_snd_test, koja testira može li trenutni paket biti poslan odmah. To često nije moguće zbog redova čekanja koji nastanu ukoliko je klijent spor ili nije poslao potvrde o primitku dovoljno brzo. Potom tcp_transmit_skb

proslijeđuje TCP paket mrežnom sloju, koristeći specifičnu funkciju ovisno o činjenici koristi li se IPv4 ili IPv6 adresa.

Naposlijetku, update_send_head inicijalizira brojač vremena koji vodi računa o tome treba li paket ponovno poslati. Važno je napomenuti kako se brojač ne održava za svaki paket, nego samo za prvi paket koji je poslan nakon zadnje primljene potvrde. U trenutku primitika paketa sa ACK kontrolnom zastavicom, red čekanja na ponovno slanje se prazni.

5.1.9 Mehanizmi odgode odgovora

U interaktivnim aplikacijama, primjerice, telnet sesijama, potvrda primitka paketa bi se u teoriji trebala slati nakon svakog primljenog slova. Međutim, to kreira veliki promet na mreži, a sami paketi ne nose podatke, već samo informaciju o stanju veze. Kako bi se izbjeglo zagušenje u takvim situacijama, primjenjuju se algoritmi odgode odgovora - odgoda slanja ACK paketa i Nagle-ov algoritam.

Odgađanje slanja ACK paketa pretpostavlja kako nije nužno da se ACK šalje za svaki paket, već se dopusti slanje nekoliko paketa bez ACK, te se ACK pošalje samo za zadnji od njih. Implementacija ovog mehanizma se oslanja na sistemski mjerač vremena, detaljnije opisan u poglavlju “TCP mjerači vremena”.

Nagle-ov algoritam se primjenjuje na paketima male veličine (eng. tinygrams). On prilagođava brzinu slanja odgovora kapacitetu veze, odnosno, kontrolira tijek podataka u vezi. U suštini, Nagle-ov algoritam se ne oslanja na brojač vremena operacijskog sustava, već na vremenski razmak paketa primatelja. Definiran je u datoteci tcp_output.c, kao statička funkcija tcp_nagle_check. Kontrola primjene Nagle-ovog algoritma vrši se postavljanjem opcije TCP_NODELAY prilikom kreacije socketa.

5.1.10 Zatvaranje TCP veze

Kao i uspostava veze, zatvaranje također uključuje razmjenu nekoliko specifičnih paketa propisanih protokolom. Za razliku od uspostave veze, zatvaranje veze može biti planirano ili iznenadno. Planirano zatvaranje (eng. graceful close) terminira vezu na zahtjev klijenta ili servera. Iznenadno zatvaranje (eng. abort) se u većini slučajeva dogodi ukoliko neki od protokola na višoj razini iskusi grešku, primjerice, neočekivana terminacije aplikacije. Za planirano zatvaranje veze, klijent i server trebaju izmjeniti 4 paketa.

Na računalu koje želi zatvoriti vezu, prvo se poziva funkcija close koja generira TCP sa označenom kontrolnom zastavicom FIN. Ujedno se i stanje socketa na tom računalu postavlja na FIN_WAIT_1. Drugo računalo prima FIN paket, na kojeg odgovara ACK paketom i postavlja stanje socketa sa svoje strane na CLOSE_WAIT. Po primitku ACK paketa, prvo računalo postavlja socket na FIN_WAIT_2. U to vrijeme aplikacija na drugom računalu također poziva funkciju close koja šalje paket sa zastavicom FIN prvom računalu. Socket na drugom računalu se postavlja na LAST_ACK stanje. Prvo računalo šalje

posljedni ACK paket drugom računalu te postavlja socket u TIME_WAIT. Nakon isteka određenog vremenskog perioda, stanje socketa se automatski mijenja u CLOSED. Drugo računalo prima posljednji ACK paket, što uzrokuje da socket automatski prijeđe u CLOSED stanje.

Promjene stanja socketa odvijaju se unutar tcp_rcv_state_process funkcije.

Polu-zatvorena TCP veza

Osim stanja potpunog zatvaranja, veza se može polu-zatvoriti (eng. Half close), odnosno, FIN segment se pošalje sa strane primatelja, ali ne i sa strane pošiljatelja, te primatelj nastavi primati podatke. Jednostrano slanje podataka završava u trenutku kada i druga strana pošalje FIN segment. Mehanizam razmjene paketa je sličan kao i kod potpunog zatvaranja: jedna strana pošalje FIN segment, kojeg druga strana potvrdi ACK segmentom, međutim, druga strana potom ne šalje FIN. To signalizira prvom računalu kako postoji još podataka koje je potrebno razmjeniti na vezi. U tom trenutku je primatelj u stanju FIN_WAIT_2, a pošiljatelj u stanju CLOSE_WAIT.

Polu-zatvaranje veza ne iziskuje dodatnu programsku implementaciju: obje strane veze su u stabilnom stanju TCP konačnog automata, te ne zahtijevaju dodatnu intervenciju.

5.1.11 Otkrivanje MTU-a na ruti

MTU je definiran kao maksimalna veličina paketa koja se može prenijeti određenom vezom između dva računala (eng. Maximum transfer size). Različita računala mogu imati različito podešene MTU vrijednosti, ali za pouzdan prijenos podataka, računala s većom vrijednošću poštuju računalo s najmanjom, kako bi se izbjegla fragmentacija paketa. Na starijim sustavima, otkrivanje MTU vrijednosti se izvršavalo na mrežnom sloju, međutim, s vremenom je prebačeno na kranja računala (eng. endpoints), odnosno, na transportni sloj mreže. Pretpostavka ovog pristupa je kako je efikasnije u početku generirati paket točne veličine, nego nositi se s fragmentacijom na prijenosnoj ruti. U implementaciji se koristi mehanizam povratnih informacija kako bi se došlo to točne veličine.

U trenutku uspostave veze, TCP kao MTU koristi ili MTU definiran na izlaznom sučelju računala ili maksimalnu veličinu segmenta (MSS) definiranu od strane primatelja. Jednom kada je odabran SMSS (eng. Send maximum segment size), odnosno, maksimalna veličina poslanog segmenta, svi paketi na toj vezi imaju označenu DF (eng. Don’t fragment) zastavicu. IPv6 implicira tu zastavicu, te se ona ne dodaje eksplicitno. Ukoliko pošiljatelj primi poruku kako je poslani segment prevelik, odnosno, poruku da je fragmentacija potrebna, smanjuje SMSS i ponovno šalje podatke. Poruka o činjenici da je poslani paket prevelik može ujedno sadržavati i sugestiju za postavljanje MTU vrijednosti za sljedeći skok u mreži. Ukoliko to nije slučaj, pošiljatelj mora odlučiti koju vrijednost koristiti, a za tu svrhu se često koristi binarno pretraživanje. Mrežne rute se s vremenom mijenjaju, stoga je preporučeno da algoritam svakih deset minuta pokuša povećati MTU vrijednost, maksimalno do broja inicijalne SMSS vrijednosti.

5.2 Istek vremena i ponovno slanje

5.2.1 TCP mjerači vremena

Prilikom slanja TCP podataka, paketi se stavljaju u red čekanja za ponovno slanje ukoliko se ne primi ACK u određenom vremenskom periodu. Taj vremenski period prati se od strane dediciranog brojača vremena. Ovo je samo od jedan od brojača koji se pojavljuju u TCP implementacijama, a najmanje šest je potrebno za pouzdano funkcioniranje protokola: brojač vremena za ponovno slanje (eng. Retransmit timer), brojač za zakašnjeli ACK paket (eng. Delayed ACK timer), brojač za nul-prozor (eng. Zero window probe timer) te keep-alive brojač (eng. Keep-alive timer), TIME_WAIT brojač te SYN-ACK brojač.

Prva tri brojača navedena u listi su definirana kao dio osnovnog TCP koda u jezgri, odnosno, oni su dio osnovne specifikacije protokola. Keep-alive brojač se koristi u upravljanju vezama koje su u ESTABLISHED stanju socketa. TIME_WAIT brojač se na isti način koristi za veze koje su u CLOSE stanju te čekaju na istek MSL vremena. Naposlijetku, SYN-ACK brojač se poziva prilikom kreiranja novih konekcija, odnosno, prilikom izvođenja trostrukog rukovanja.

Brojač vremena za ponovno slanje

Brojač vremena za ponovno slanje je dio TCP specifikacije, odnosno, TCP automata, posebice dijela za otkrivanje zagušenosti mreže. Naime, svaki poslani podatak se sprema u međuspremnik za ponovno slanje ukoliko još nije primljen ACK paket. Pošiljatelj čeka određeno vrijeme na ACK, te ukoliko ga ne primi, ponovno šalje paket. Vrijeme čekanja se računa bazirano na RTT-u. Struktura tcp_opt sadrži polje packets_out u koju se spremaju informacije o paketima koji još nisu primili ACK.

Brojač zakašnjelog ACK paketa

U implementaciji TCP protokola, susrećemo dva mehanizma slanja ACK paketa: brzi (eng. quick) i zakašnjeli (eng. delayed). Brzi ACK mehanizam se primjenjuje u situacijama kada je poželjno da primatelj šalje što je brže moguće stoga što maksimalna veličina prozora zagušenja nije dostignuta ili kada je potrebno primiti paket van redoslijeda. Mehanizam zakašnjelog slanja primjenjuje se najčešće u interaktivnim sesijama, primjerice, prilikom korištenja telnet ili rlogin servisa. Prilikom takvog načina korištenja, svako slovo koje pošiljatelj šalje potrebno je prikazati na konzoli primatelja. Kada bi se ACK slao za svako slovo, generirao bi se veliki broj paketa koji imaju potencijal zagušiti vezu. Unutar strukture tcp_opt, definirana je struktura ack. Unutar ack strukture, polje pending sadrži informacije o stanju slanja ACK paketa. Pending polje poprima TCP_ACK_TIMER vrijednost ukoliko je definiran brojač zakašnjelog ACK paketa.

Brojač za nul-prozor

TCP protokol postavlju veličinu prozora na nulu, ukoliko je međuspremnik za primanje podataka pun. Svaki put kada aplikacija pročita podatke iz međuspremnika za primanje podataka, provjerava koliko je prostora ostalo na raspolaganju te odlučuje hoće li poslati ACK sa novom veličinom prozora pošiljatelju. Ukoliko se taj ACK izgubi, primatelj i pošiljatelj ostaju blokirani u tom stanju. Iz tog razloga, pošiljatelj implementira brojač za prozor. Taj brojač ispituje ukoliko primatelj ima otvoren prozor. Šalje 1 bajt podataka skupa sa ispitivačkim segmentom, koji ima redni broj za jedan manji od zadnjeg poslanog paketa. Svaki put prije slanja novog segmenta, pošiljatelj provjerava je li slanje moguće pozivom __tcp_push_pending_frames(), koja provjerava je li moguće poslati pakete u red čekanja za slanje, i tcp_data_snd_check() koja se poziva u trenutku prije slanja odgovora, odnosno, ACK paketa pošiljatelju, da provjeri postoje li drugi paketi koji čekaju na slanje. Ukoliko bilo koja od navedene dvije funkcije ne uspije poslati paket, poziva se rutina tcp_check_probe_timer(), koja provjerava je li veličina prozora razlog nemogućnosti slanja podataka. U tu svrhu, provjerava se postoje li paketi koji su dulje u redu čekanja, te je li postavljen drugi brojač koji spriječava trenutno slanje. Ukoliko obje provjere rezultiraju negativnim odgovorom, zaključak je da razlog nemogućnosti slanja mora biti nul-prozor obzanjen od strane primatelja. U tom trenutku se brojač nul-prozora resetira na specifično vrijeme isteka, uzimajući RTT kao osnovu izračuna.

Keepalive brojač

TCP koristi keepalive brojač u svrhu ispitivanja aktivnosti na vezi. Ovaj brojač također čestu primjenu nalazi u interaktivnim sesijama gdje postoji velika mogućnost da nijedan paket neće biti poslan duži vremenski period. Pošiljatelj šalje paket sa rednim brojem za jedan manjim nego zadnji potvrđeni paket. Kada primatelj primi taj paket, očekuje se da će ga interpretirati kao ponovno slanje već dobivenog paketa i poslati ACK u kratkom periodu. Ukoliko pošiljatelj ne primi ACK u očekivanom vremenu, pretpostavlja da na vezi postoji problem. U tom trenutku, brojač se resetira, a veza zatvara. Keepalive brojač se postavlja prilikom uspostavljanja svake nove konekcije, unutar tcp_create_openreq_child() funkcije, ali isključivo ukoliko je aplikacija označila socket opcijom SO_KEEPALIVE. Opcija SO_KEEPALIVE u programskom kodu označava tp->keepopen varijablu.

Ovaj brojač se inicijalizira unutar funkcije tcp_keepalive_timer unutar tcp_init_xmit_timers. Resetiranje brojača se vrši pozivom na tcp_reset_keepalive_timer.

TIME_WAIT brojač

U trenutku kada veza uđe u TCP_WAIT stanje, potrebno je čekati određeno vrijeme (definirano MSL vrijednošću) prije nego se konekcija zatvori. Čekanje prije zatvaranja konekcije se uvodi kako bi se izbjegla situacija u kojoj paketi kasne zbog latencije na mreži, a primatelj zatvori vezu misleći kako nema odgovora od strane pošiljatelja.

TIME_WAIT brojač se inicijalizira pozivom na tcp_time_wait prilikom ulaska u fazu zatvaranja veze.

SYN-ACK brojač

SYN-ACK brojač definiran je u svrhu praćenja zahtjeva za vezom koji nisu bili prihvaćeni određeni vremenski period. Osnovna ideja ovog brojača jest ukoliko primatelj (aplikacija) nije u mogućnosti prihvatiti vezu, zahtjeve za vezom se treba procesirati i poslati odgovor. Osim situacije u kojoj aplikacija nije u mogućnosti prihvatiti vezu, moguće je i da je primatelj u stanju čekanja na ACK nakon poslanog SYN-a. U oba slučaja, pozivati će se SYN-ACK brojač.

SYN-ACK brojač se aktivira u trenutku kada pristigne novi zahtjev za vezom i u reda čekanja se ne nalazi postojeći zahtijev za vezom.

lopt->qlen brojač se inkrementira za jedan svaki put kada pristigne novi zahtjev za vezon, pozivajući tcp_synq_added. Nakon što je veza uspostavljena i trostruko rukovanje uspješno završeno, lopt->qlen se dekrementira za jedan pozivom na tcp_synq_removed.

U trenutku kada lopt->qlen poprimi vrijednost nula, pretpostavlja se kako nema novih zahtjeva za vezom koje treba pratiti te se SYN-ACK brojač zaustavlja, pozivom na tcp_delete_keepalive_timer iz funkcije tcp_synq_removed.

Postavljanje vrijednosti isteka vremena

Ključna varijabla prilikom ponovnog slanja u implementacijama TCP protokola jest RTT vrijednost. Ukoliko se segment pošalje prije isteka RTT-a, riskira se redundantan promet na mreži. Ako se segment pošalje nakon isteka RTT vrijednosti, dostupna propusnost mreže se ne iskorištava u potpunosti. RTT vrijednost se mijenja s vremenom, te svaka TCP implementacija stoga mora imati mehanizam praćenja promjena vrijednosti i reagiranja na njih kako bi se zadržale optimalne performanse veze i prijenosa podataka.

Kako bi se izmjerio trenutni RTT, moguće je poslati paket sa određenim rednim brojem te mjeriti vrijeme potrebno da se primi ACK segment za taj paket. Ova metoda nazvana je uzorkovanjem RTT-a (eng. RTT sampling). Potrebno je sakupiti par uzoraka te na temelju njih izračunati optimalnu vrijednost isteka vremena, odnosno, RTO (eng. Retransmission timeout).

Prosječna RTT vrijednost se računa za svaku uspostavljenu vezu zasebno.

U specifikaciji TCP protokola, unutar polja opcija u segmentu moguće je slati TSopt zastavicu (eng. Timestamps option). TSopt je sastavljen od TSval polja, koje sadrži trenutnu vrijednost sistemskog sata te TSecr (eng. Timestamp echo reply) polja koje “ispisuje” vrijednost postavljenu u TSval.

Implementacija izračuna RTO vrijednosti u jezgri koristi TSopt zastavicu te sistemski sat na granularnosti od jedne milisekunde. Za objašnjenje trenutnog izračuna korištenog od

strane Linux jezgre, potrebno je definirati par specifičnih termina te dvije ususta vljene metode kalkulacije RTO vrijednosti: klasičnu i standardnu.

U specifikaciji klasične metode izračuna, koristi se zaglađena estimacija RTT-a, poznata kao SRTT (eng. Smoothed RTT), po sljedećoj formuli: SRTT ← α(SRTT) + (1 − α) RTT s. Iz formule je vidljivo kako se za izračun nove SRTT vrijednosti, koristi trenutna vrijednost i novi uzorak. Konstanta α je faktor zaglađenja, s preporučenom vrijednošću u intervalu između 0.8 i 0.9, što znači da 80%, do 90% utjecaja na novu vrijednost ima prethodna SRTT vrijednost. U odnosu na klasičnu metodu kalkulacije RTO vrijednosti, standardna poboljšava performanse i točnost na mrežama sa većim fluktuacijama u vrijednosti RTT-a. Izračun upotrebljava srednju vrijednost devijacije kako bi se to postiglo, te uvodit RTTVAR, odnosno, faktor varijacije RTT-a.

Osim opisanih dviju varijabli, Linux jezgra uvodi i mdev i mdev_max. Varijabla mdev sadrži procijenjenu srednju vrijednost devijacije za RTTVAR dobivenu algoritmom iz standardnog pristupa izračunu, a mdev_max sadrži najveću vrijednost devijacije koja ne smije pasti ispod 50 ms. Pri svakom porastu mdev_max, jezgra također ažurira RTTVAR vrijednost na vrijednost od mdev_max. Unutar jezgre je također postavljena varijabla TCP_RTO_MIN, koja označava minimalnu vrijednost RTO vremena, a moguće je postaviti ju prije kompajliranja jezgre ili upotrebom ip naredbe u određenim verzijama jezgre. TCP_RTO_MIN ima zadanu vrijednost od 120 sekundi.

U strukturi tcp_sock deklariranje su ranije spomenute varijable:

```
/* RTT measurement */
            u32     srtt;       /* smoothed round trip time << 3        */
            u32     mdev;       /* medium deviation                     */
            u32     mdev_max;   /* maximal mdev for the last rtt period */
            u32     rttvar;     /* smoothed mdev_max                    */
            u32     rtt_seq;    /* sequence number to update rttvar     */
```

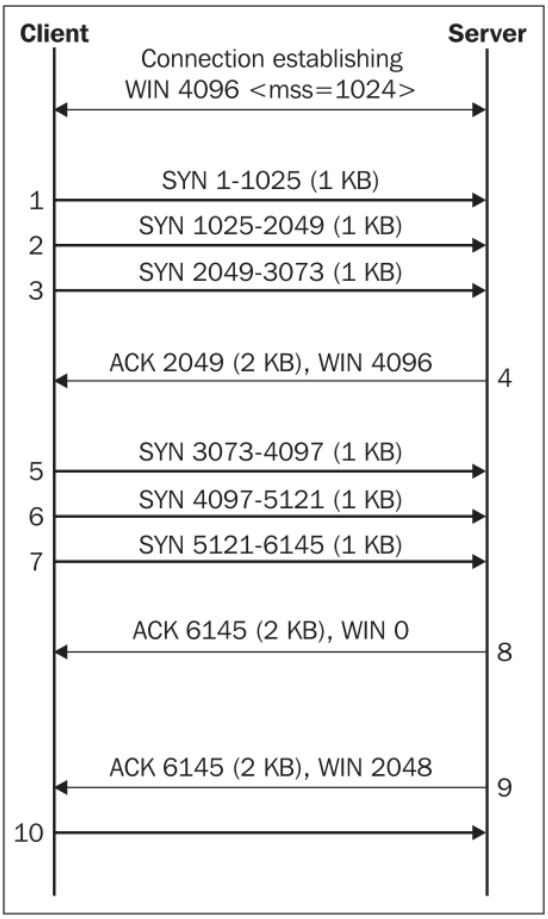
Sam izračun RTO vrijednosti vrši funkcija __tcp_set_rto() po formuli (tp->srtt >> 3) + tp->rttvar.

5.3 Tok podataka i mehanizam pomičnih prozora

Pošiljatelj može slati podatke dok god dobiva potvrde o primitku za poslane pakete. Ukoliko potvrde o primitku prestanu pristizati, pošiljatelj i dalje nastavlja slanje paketa dok ukupan preneseni teret ne dostigne veličinu prozora klijenta, definiranu u bajtima. Mehanizam pomičnih prozora koristi se u TCP protokolu za kontrolu tijeka veze (eng. Flow control). U isto vrijeme se pojam prozora koristi u konceptima kontrole zagušenja, koji su detaljnije opisani u nadolazećim poglavljima.

Na slici _____ prikazan je primjer komunikacije u kojoj je moguće promotriti primjenu prozora. Prilikom ugovaranja parametara veze, primatelj i pošiljatelj su se složili da će veličina prozora iznostit 4096 bajtova. Klijent šalje tri paketa, ukupne veličine 3073 bajtova, za koje ne dobiva potvrdu o primitku. Nije u mogućnosti poslati treći paket veličine 1024 bajta, zbog toga što bi u tom trenutku nadišao veličinu dogovorenog prozora za jedan bajt, stoga čeka potvrdu primitka.

Primatelj šalje ACK za 2049 bajtova, što znači da je 1024 iz trećeg poslanog paketa ostalo nepotvrđeno. Pošiljatelj šalje dodatna tri paketa veličine 1024 bajta, što skupa sa trećim paketom čini 4096 bajtova. Primateljev međuspremnik je u tom trenutku pun te ne može primiti niti jedan paket. Primatelj šalje ACK da potvrdi primitak svih dotada poslanih segmenata te postavlja veličinu prozora na 0, odnosno, ne dopušta daljnje slanje podataka. U sljedećem trenutku, primatelj obradi dio podataka te šalje ponovljeni ACK pošiljatelju sa novom veličinom prozora postavljenom na 2048 bajtova.



5.3.1 Algoritmi pomičnih prozora

Algoritam stani-i-čekaj

Stani-i-čekaj (eng. stop-and-wait) je najjednostavniji pristup pomičnim prozorima. Veličina pomičnog prozora se pri uspostavi veze postavlja na 1 MSS, odnosno, na 1 paket. Nakon svakog poslanog paketa, pošiljatelj čega ACK segment prije nego pošalje sljedeći. Stani-i-čekaj algoritam se u literaturi pojavljuje češće kao jedno od rješenje za problem ponovnog slanja, no tehnika koju implementira može se svrstati i kao jednostavan mehanizam kontrole toka segmenata.

Algoritam vraćanja N koraka

U ovom algoritmu, prozor pošiljatelja je veći od jednog paketa, ali primateljev prozor je jednak jednom paketu. Efektivno, postižu se slične performanse stani-i-čekaj algoritmu na vezama koje su podložne gubitku paketa.

Algoritam selektivnog ponovnog slanja

Selektivno ponovno slanja zahtijeva primatelja sa međuspremnikom, koji može spremiti više od jednog paketa. Za razliku od prethodnog algoritma, nije potrebno odbaciti pakete za koje primatelj nije poslao ACK segment, oni se spremaju u međuspremnik.

5.4 Mehanizmi kontrole zagušenja

Primatelj diktira veličinu pomičnog prozora, uzimajući u obzir veličinu međuspremnika i broj procesiranih, već primljenih paketa. Međutim, problem može nastati i na samoj vezi: ukoliko je pošiljatelj na brzoj mreži, a primatelj na sporoj, paketi koji pošiljatelj šalje mogu stvoriti zagušenje na mreži, prije nego što stignu do primatelja. Usmjerivači na mreži bi u tom trenutku odbacili sve nadolazeće pakete. Kako bi se izbjegao gubitak paketa i preopterećenje mreže, osim pomičnog prozora definiranog na primatelju, definira se i prozor zagušenja na pošiljatelju (eng. Congestion window, CWND). Zadaća prozora zagušenja jest odrediti koliko paketa pošiljatelj može poslati primatelju tako da se primateljeva mreža ne optereti.

Pošiljatelj nikada ne smije poslati više podataka nego što je deklarirano pomičnim prozorom na strani primatelja niti više podataka od onoga što nalaže prozor zagušenja. Prilikom odlučivanja koja je maksimalna količina podataka koju je moguće poslati u određenom trenutku, uzima se manja od dostupne dvije vrijednosti za prozor zagušenja i pomični prozor.

Veličina prozora zagušenja je na početku niski višekratnik maksimalne veličine paketa (eng. MSS, maximum segment size). Za svaki primljeni ACK, prozor zagušenja se poveća za 1 + CWND (slow start).

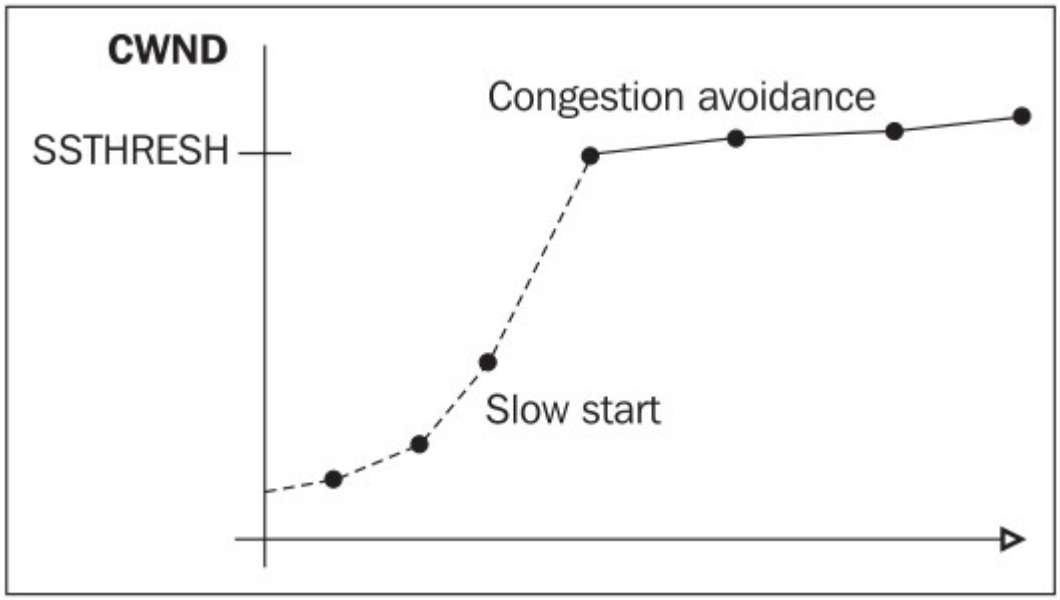
Ovaj pristup povećanju prozora poznat je kao AIMD (eng. Additive increase/multiplicative decrease). U suštini, AIMD kombinira linearni rast prozora zagušenja sa višekratnim smanjenjem u trenutku kada se dogodi zagušenje veze.

Pošiljatelj postupno povećava veličinu prozora, no ograničen je SSTHRESH (eng. Slow-start threshold) vrijednošću. SSTHRESH predstavlja krajnju granicu veličine prozora, nakon čijeg je prijelaska vrlo vjerovatno da će se na mreži dogoditi zagušenje. SSTHRESH se mjeri kao višekratnik veličine paketa. U trenutku kada je ta granica doesgnuta, algoritam prelazi u stanje izbjegavanja zagušenja (eng. Congestion avoidance). U stanju izbjegavanja zagušenja, prozor zagušenja se povećava linearno, po primitku ACK paketa, po formuli CWND = CWND + 1/CWND, dok god pošiljatelj ne prima duplicirane ACK pakete. Prozor zagušenja u stanju izbjegavanja zagušenja raste po danoj formuli sve dok se ne dogodi istek predefiniranog vremena (eng. timeout). U tom trenutku, algoritam izvodi sljedeće operacije:

-- CWND = 1 MSS (veličina prozora zagušenja jednaka je maksimalnoj veličini jednog segmenta)

-- SSTHRESH = CWND/2 (slow-start granica se smanjuje na pola veličine prozora zagušenja)

Na slici _____ je prikazana brzina promjene veličine prozora zagušenja prije dostizanja SSTHRESH vrijednosti te nakon.



5.4.1 Brzo ponovno slanje i brzi oporavak

Ukoliko pošiljatelj primi tri ili više dupliciranih ACK paketa uzastopno, pretpostavlja kako je došlo do gubitka podataka te ponovno šalje sljedeći paket bez čekanja na istek predefiniranog vremena. Dva uzastopna ACK paketa se toleriraju, pretpostavljajući da je sljedeći paket u nizu na odredište stigao kraćom rutom, te će ACK za njega eventualno stići.

Algoritam brzog ponovnog slanja u suštini smanjuje predefinirano vrijeme isteka koje bi pošiljatelj morao čekati prije ponovnog slanja, kako bi se izbjegla dodatna latencija na vezi ako je sigurno da je paket izgubljen.

Ukoliko je brzo ponovno slanje kombinirano sa brzim oporavkom, pošiljatelj nakon ponovnog slanja ne ulazi u slow start, već smanjuje veličinu prozora zagušenja za pola. U trenutku kada je primljen ACK za ponovno poslane podatke (eng. Recovery ACK), pošiljatelj ponovno ulazi u stanje izbjegavanja zagušenja.

5.4.2 Algoritmi kontrole zagušenja

Algoritmi kontrole zagušenja se razlikuju u implementaciji za situaciju kada na vezi dođe do gubitka paketa. Ukoliko gubitka paketa nema, svi algoritmi pokazuju iste performanse i stabilnost.

Na Linux operacijskim sustavima, trenutno korištena varijanta algoritma kontrole zagušenja zapisana je u datoteku /proc/sys/net/ipv4/tcp_congestion_control, a svi dostupni algoritmi se nalaze u datoteci /proc/sys/net/ipv4/tcp_available_congestion_control. Na suvremenim sustavima, najčešće se koristi CUBIC. Algoritmi kontrole zagušenja implementirani su u dvije datoteke: tcp_cong.c (zadana implementacija) te tcp_[ime algoritma].c (ostali podržani algoritmi).

Kontrola zagušenja u jezgri je implementirana kao konačni automat koji prelazi između različitih predefiniranih stanja veze. Stanja koja automat može poprimiti definirana su u tcp.h datoteci pod tipom enum:

```
enum tcp_ca_event {
    CA_EVENT_TX_START,    /* first transmit when no packets in flight */
    CA_EVENT_CWND_RESTART,    /* congestion window restart */
    CA_EVENT_COMPLETE_CWR,    /* end of congestion recovery */
    CA_EVENT_LOSS,        /* loss timeout */
    CA_EVENT_ECN_NO_CE,    /* ECT set, but not CE marked */
    CA_EVENT_ECN_IS_CE,    /* received CE marked IP packet */
};
```

Struktura tcp_congestion_ops je također definirana unutar tcp.h datoteke. Ona predstavlja sučelje prema algoritmima kontrole zagušenja. Sadrži samo pokazivače na funkcije koje se pozivaju u različitim fazama zagušenja:

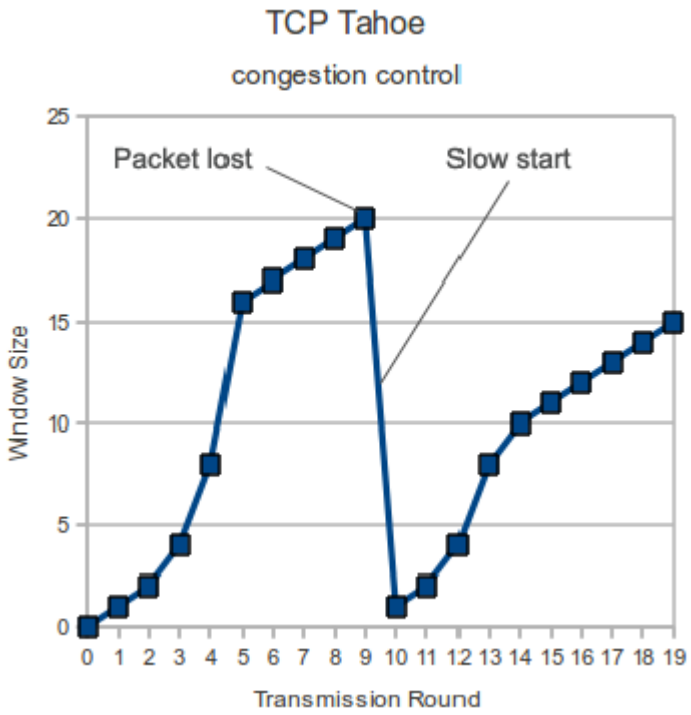
```
struct tcp_congestion_ops {
    struct list_head    list;
    u32 key;
    u32 flags;

    /* initialize private data (optional) */
    void (*init)(struct sock *sk);
    /* cleanup private data  (optional) */
    void (*release)(struct sock *sk);

    /* return slow start threshold (required) */
    u32 (*ssthresh)(struct sock *sk);
    /* do new cwnd calculation (required) */
    void (*cong_avoid)(struct sock *sk, u32 ack, u32 acked);
    /* call before changing ca_state (optional) */
    void (*set_state)(struct sock *sk, u8 new_state);
[...]
```

Primjerice, poziv na init funkciju se događa prilikom primanja prvog ACK segmenta, te se tada ujedno pozove i sistemski definirani algoritam kontrole zagušenja. Funkcija cong_avoid se poziva u fazi izbjegavanja zagušenja, a dopušta povećanje veličine prozora.

TCP Tahoe

TCP Tahoe je najjednostavniji mehanizam kontrole zagušenja. Ne implementira brzi oporavak, a trostruki ponovljeni ACK paket tretira jednako kao i istek predefiniranog vremena. U fazi oporavka od zagušenja, ponovno šalje izgubljeni paket, smanjuje veličinu prozora zagušenja te ulazi u slow start fazu. TCP Tahoe implementira izbjegavanje zagušenja i brzo ponovno slanje.

Na slici ___ je prikazan graf kretanja veličine prozora kroz vrijeme trajanja veze, ukoliko je TCP Tahoe korišteni algoritam kontrole zagušenja.

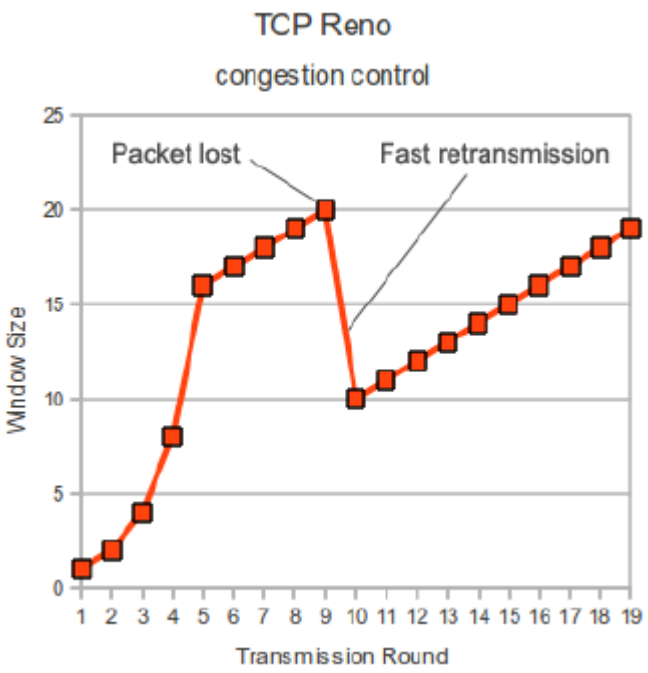


TCP Reno

TCP Reno poboljšava Tahoe u fazi izbjegavanja zagušenja: tretira istek predefiniranog vremena i uzastopni duplicirani ACK paket kao dvije različite situacije. U slučaju uzastopnih ACK paketa, reagira kao i Tahoe, no također implementira i brzi oporavak umjesto direktnog ulaska u slow start fazu. Na slici ____ je prikazana veličina prozora zagušenja u primjerni Reno algoritma. Vidljivo je kako u odnosu na Tahoe, brzi oporavak značajno smanjuje vrijeme potrebno za postizanje faze izbjegavanja zagušenja, odnosno, faze u kojoj je prozor zagušenja optimalne veličine.

Međutim, trajanje faze izbjegavanja zagušenja je i dalje jednako kao kod Tahoe algoritma.

Reno se odabran kao zadana implementacija algortima kontrola zagušenja u jezgri, sve do suvremenijih sustava. Unutar tcp_cong.cc datoteke nalazi se fukcija tcp_reno_cong_avoid() koja implementira originalnu specifikaciju Reno algortima.



TCP NewReno

Ukoliko dođe do gubitka velikog broja paketa, TCP Reno pokazuje relativno loše performanse. TCP NewReno optimizira brzi oporavak te postiže poboljšanja u toj fazi veze. U implementaciji NewReno algoritma, ACK paket za nove podatke (nakon završetka ponovnog slanja) nije dovoljan za izlazak iz faze brzog oporavka u slow start. Zahtijeva se da primatelj eksplicitno potvrdi sve pakete koji su bili izgubljeni prije nego je ponovno slanje započelo.

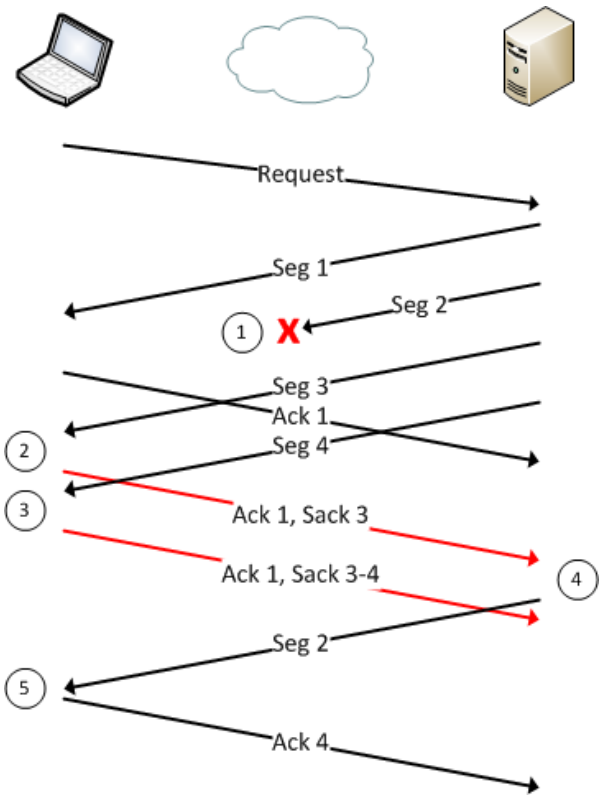
SACK (Selektivni ACK)

Pretpostavimo da je pošiljatelj primatelju poslao tri segmenta, primjerice, 1, 2, i 3. Primatelj je poslao ACK samo za segment 1. U ovoj situaciji postoji vrlo velika mogućnost da su segmenti 2 i 3 primljeni od strane primatelja normalno, ali ACK ne može biti poslan zbog toga što primatelj mora naznačiti da nije primio segment 1. Pošiljatelj u ovom trenutku odlučuje ponovno poslati segment 1, ali i zaključuje kako niti 2 ni 3 nisu primljeni te ponovno šalje i njih.

Iz priloženog primjera možemo zaključiti kako se u sličnim situacijama može generirati velika količina redundantnog prometa te na taj način usporiti samu vezu.

Rješenje problema je predstavljeno u vidu SACK algoritma, koji primjenjuje tehniku selektivnog slanja ACK segmenata. SACK omogućuje primatelju da na duplicirani ACK segment nadoda SACK, čija vrijednost može biti ne-uzastopni niz brojeva koji označavaju koji segmenti su primljeni nakon što je izgubljen segment za kojeg se šalje ACK.

Na slici _____ je priložen pojednostavljen prikaz primjene SACK algoritma:



TCP Vegas

TCP Vegas je algoritam koji koristi vrijeme pristizanja paketa kao mjerilo postojanja latencije na mreži. Naime, Tahoe i Renoe, primjerice, detektiraju zagušenje u trenutku kada se paketi izgube. Fokus nakon toga je da se veličina prozora zagušenja što brže vrati na prethodnu, u fazi izbjegavanja zagušenja. Vegas promatra RTT vrijeme nadolazećih paketa te reagira na zagušenje ukoliko RTT poraste. Vegas uvelike ovisi o preciznoj kalkulaciji osnovne vrijednosti RTT-a (eng. baseline). Ukoliko je osnovna vrijednost preniska, propusnost veze neće biti u potpunosti iskorištena, a ukoliko je previsoka, veza će postati zagušena.

TCP CUBIC

Cubic je algoritam koji poboljšava prethodno korišteni TCP BIC. BIC upotrebom binarnog pretraživanja traži maksimalnu vrijednost veličine prozora zagušenja. U CUBIC implementaciji, veličina prozora je kubična funkcija vremena proteklog od zadnjeg zagušenja. Točka infleksije kubične funkcije postavlja se na vrijednost veličine prozora koja se primjenjivala prije zagušenja.

U konkavnom dijelu funkcije, koji vodi do točke infleksije, CUBIC postupno povećava veličinu prozora do vrijednosti prije zadnjeg zagušenja. Pri dosegnuću te vrijednosti, konveksni dio funkcije postupno povećava tu vrijednost prozora.

6 Primjene

TCP kao protokol nalazi široku primjenu u aplikacijama koje zahtijevaju pouzdan prijenos podataka, primjerice: HTTP, SSH, FTP ili IMAP/POP.

Linux jezgra najširu primjenu nalazi upravo na serverima, odnosno, računalima koji svakodnevno dostavljaju datoteke klijentima, koristeći upravo jedan od navedenih servisa i protokola. Web serveri Apache i nginx najpoznatiji su HTTP serveri na GNU/Linux platformi, a koriste se za jednostavno serviranje datoteka, serviranje kompleksnih web stranica ili videa na zahtijev (nginx video-on-demand modul, koji transkodira u HLS format).

Osim kao web server, GNU/Linux široku primjenu nalazi i kao datotečni server, a u tom slučaju jedan od glavnih servisa jest FTP server, primjerice, vsftpd ili proftpd. Također, često se u primjeni Linux jezgra koristi na ugradbenim računalima, a u tom slučaju se komunikacija sa samim računalom i njegovo održavanje odvija putem uspostavljanja SSH konekcije, koristeći najčešće openssh biblioteku.

Naposlijetku, kao poslijednu primjenu TCP stoga implementiranog u Linux jezgri potrebno je navesti servere elektroničke pošte. Iako je taj servis često prisutan kao popratni proces drugima, nije rijetkost u industriji susresti dedicirane servere elektroničke pošte. Agenti za slanje pošte (eng. MTA, Mail Transfer Agents) kao što su postfix, sendmail ili dovecot nerijetko su ključni servisi u mnogim tvrtkama.

Prednost Linux jezgre u industrijskom okruženju jest svakako fleksibilna implementacija TCP stoga, koja omogućava dodatne optimizacije ovisno o potrebama, te podrška za širok spektar hardvera, odnosno, mrežnih kartica.

6.1 Linux mrežni stog unutar ns-3 simulatora

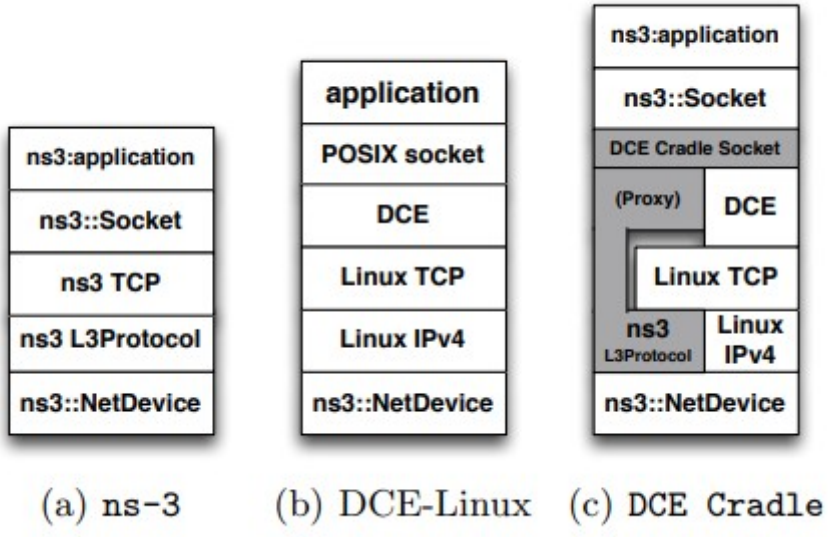
Mrežni simulator definiramo kao računalni program koji oponaša događaje u stvarnoj, fizičkoj, mreži. Događaji na stvarnoj mreži se “zakazuju” u simulatoru, odnosno, određuje se redoslijed izvođenja. Jedan od simulatora koji koristi mrežni stog Linux jezgre jest n2-3 simulator. Implementiran je u programskom jeziku C++, a sastoji se od više desetaka modula.

ns-3 simulator podržava nekoliko implementacija TCP protokola: nativna implementacija (specifično za simulator), NSC (Network Simulation Cradle) programski okvir koji omoguće “umotavanje” koda iz drugih operacijskih sustava ili podsustava u simulator, upotreba mrežnog stoga operacijskog sustava domaćina, te naposlijetku, DCE (Direct Code Execution) na Linux jezgri.

U svakoj od spomenutih implementacija, susrećemo se sa par problema. Primjerice, simulirani protokoli su vrlo često pojednostavljene verzije “stvarnih” protokola, u kojima nisu podržane sve funkcionalnosti koje pruža stvarna implementacija.

Jedan od načina kako taj problem rješiti jest upotrebom DCE implementacije. U tom slučaju koristio bi se jezgrin kod, sa svim funkcionalnostima I sigurnosnim zakrpama, što bi doprinjelo lakoći održavanja koda. Međutim, djelovi ns-3 koda koriste nativnu implementaciju utičnica (primjerice, generator prometa), koje Linux jezgra nije u mogućnosti iskoristiti (POSIX standard). Kao alternativni pristup, u znanstvenom radu sa francuskog Inria instituta, DCE Cradle: Simulate Network Protocols with Real Stacks, predložena je kombinacija NSC I DCE pristupa.

Implementacija DCE-Cradle modifikacije se sastoji od tri koraka: 1) uvesti omotače utičnica koji bi pristupali POSIX utičnicama koje se koriste u DCE-Linux implementaciji, 2) dodati sloj posrednik (eng. *proxy*) koji bi zadržao transparentnost POSIX utičnica prema ns-3 sloju, 3) unaprijediti DCE u smjeru podržavanja ns-3 nativnih utičnica. Na priloženoj slici prikazane su modifikacije stoga, od nativnog ns-3 do DCE-Cradle implementacije.



Slika 9: ns-3 stog u tri implementacije

Kako bi se zadržala transparentnost prema ns-3 sloju, autori DCE-Cradle-a su uveli novu vrstu utičnica koje direktno pristupaju Linux implementaciji mrežnog stoga. Paketi generirani od strane ns-3 simulatora preko spomenutih utičnica “ulaze” u Linux jezgru na daljnje procesiranje, te se nakon procesiranja rezultati vraćaju na mrežno sučelje koje koristi simulator.

U originalnoj implementaciji, konfiguracija IP adresa i ruta se oslanjala također na nativnu implementaciju. Međutim, u DCE-Cradle-u je taj dio simulatora pomaknut u sloj posrednik, u kojem se također kontaktira Linux jezgra.

Naposlijetku, u DCE-Cradle je dodana ekstenzija koja služi kako bi se premostile razlike između asinkronog ns-3 API-ja za utičnice te POSIX API-ja kojeg koristi jezgra. ns-3

utičnice podržavaju samo neblokirajući način rada (eng. *non-blocking*), dok je originalna DCE implementacija zamišljena i za blokirajući i neblokirajući način rada. DCE-Cradle implementacija je usvojila neblokirajući način rada preko LinuxSocketFd::Fnctl poziva, koji omogućava jezgri komunikaciju s aplikacijom bez da čeka završetak procesiranja.

7 Zaključak

Implementacija mrežnih protokola je vrlo često dio računarstva s kojima se ne susrećemo često iz programerske perspektive, već samo kao korisnici. Međutim, na TCP protokolu i umreženju računala se općenito temelji značajan dio informatike i računarskih znanosti. Upravo zbog toga smatram kako je vrlo korisno razumjeti neke od fundamentalnih principa i ideja koje su prethodile suvremenoj komunikaciji.

Tijekom izrade ovog diplomskog rada, susrela sam se s velikim brojem nepoznatih koncepata i izraza, za čije razumijevanje sam nerijetko trebala usporediti i do nekoliko članaka i knjiga kako bi ih u potpunosti razjasnila. Međutim, upravo zbog toga smatram kako sam dobila veliku količinu novog znanja i napredovala u razumijevanju područja informatike i računarstva.

8 Popis priloga

Popis slika

9 Literatura

[1] Benvenuti, *Understanding Linux Network Internals*. Shroff Publishers & Distributors, 2005.
[2] M. A. Venkatesulu and S. Seth, *TCP/IP architecture, design, and implementation in Linux*. John Wiley & Sons, 2009.
[3] W. R. Stevens and K. Fall, *TCP/IP Illustrated*. Pearson Education (US).
[4] W. Mauerer, *Professional Linux Kernel Architecture*. Wiley, 2008.