VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Počítačové komunikace a sítě – 2. projekt

Varianta ZETA: Sniffer paketů

17. dubna 2021

David Hudák

Obsah

1 Úvod

Tato dokumentace se věnuje zpracování varianty ZETA 2. projektu v rámci předmětu IPK (Počítačové komunikace a sítě).

1.1 Zadání (výtah)

Úkolem projektu je vytvořit síťový analýzator komunikace schopný filtrovat pakety. Pro tvorbu programu je nabídnuta určitá škála programovacích jazyků (C, C++ a C#), ze kterých byl zvolen programovací jazyk C++. Pro práci se sítí bylo povoleno využít knihoven TCPDUMP/LIBPCAP[2] a libnet[4] či jejich případné ekvivalenty pro jazyk C#.

2 Hlavní bloky programu

Tato část se věnuje dílčím blokům programu.

2.1 Zpracování parametrů

První činností programu (krom potřebných deklarací, definic a inicializací) je zpracování parametrů programu. To vykonává knihovna getopt skrze funkci `getopt_long`, která přijímá jak krátké (`-i`, `-p`, `-n`, `-u`, `-t`), tak i dlouhé (`--interface`, `--tcp`, `--udp`, `--arp`, `--icmp`) argumenty. Kromě zapínání přepínačů pro volbu filtrování protokolů a případného zadávání filtrovaného portu či počtu paketů, které je poměrně triviální, je nutné řešit i parametry rozhraní, které jsou klíčové pro fungování programu.

Základním kamenem úrazu je, že hodnota u parametru rozhraní je nepovinný, což znamená, že není možné skrze `getopt_long` zachytávat krátké verze parametrů s mezerou a následně až hodnotou[1]. To je ošetřeno tím způsobem, že při zaregistrování této možnosti dojde k cyklu, který prochází argumenty až k nalezení tohoto parametru a následně zkontroluje, zdali se po něm nenachází použitelná hodnota (taková, která nezačíná na znak „-“. Díky tomuto je možné rozlišit situace, kdy uživatel spustí program s mezerou u tohoto parametru a kdy si uživatel chce nechat vytisknout všechna dostupná zařízení.

2.2 Tisknutí času

Tato část implementace se potýká s problémem nutnosti vypisovat data a časy ve formě i s nulami (tedy například jedna hodina ráno není 1:0, ale 01:00). Toho je nakonec dosaženo nejprve převodem jednotlivých časových a datových údajů do řetězců a následně srovnáváním jednotlivých časových údajů s řádem, ve kterém by se měly nacházet (převážně tedy s 10, například $10 < 10 \rightarrow 10$ a $8 < 10 \rightarrow 08$). Následně je dle takového srovnání konkatenován řetězec času s 0 (nebo ne, pokud to není potřeba).

2.3 Tisknutí paketu

Tisknutí dat v paketu probíhá skrze cyklus, který si musí pamatovat offset (po každé iteraci, jednom řádku, se zvětší o 16, což je v hexadecimální podobě zvednutí druhého řádu zprava o jedničku) a ten srovnává s přijatou délkou paketu. Pro řádek nejdříve dojde k vytisknutí všech bajtů v jejich hexadecimální formě (například a8) a následně v jejich ASCII formě (to platí jenom pro bajty, jejichž hodnota je v rozmezí $\langle 32, 127 \rangle$.

[1] Tato problematika je vlastně celkem zajímavá. Odpověď na ni je třeba zde: https://stackoverflow.com/questions/26431682/why-cant-i-have-a-space-between-option-and-optional-argument-using-getopt.

2.4 Aplikace filtru

Tato problematika využívá datového typu `string` obsaženého v jazyku C++, který poskytuje jednoduchou možnost konkatenace formou operátoru +. Takto jsou nejdříve skládány do řetězce filtry pro ARP, ICMP, UDP a TCP oddělené klíčovým slovem „or“, které umožňuje filtrování přes množinu zvolených protokolů. Následně je přidán port (pokud byl zadán a pokud není kombinován pouze s volbami ICMP a ARP, které možnost portu neposkytují). Následně je takový řetězec zkompilován příkazem `pcap_compile` a nastaven příkazem `pcap_setfilter`.

2.5 Cyklus sběru paketů

Tento problém se dá řešit dvěma způsoby, a to cyklem přes `pcap_next`, kdy si programátor většinu věcí ošetřuje sám, nebo přes integrovaný cyklus `pcap_loop`, který dává menší prostor pro chybu (tudíž byl i využit). Jediné, co je nutné pro něj pak třeba implementovat, je nějaká callback funkce, která ošetří, co se má stát s jednotlivými pakety. Tato funkce je pak víceméně hlavním rozbočovačem programu, jelikož v této implementaci musí projít paket nejdříve na základě nejnižší úrovně a následně rozdělit program dál.

Funkce nejprve musí zavolat příkaz pro tisknutí času ve správném formátu (viz zde 2.2), následně rozhodnout o použitém protokolu (dostupné jsou možnosti IPv4, IPv6 a ARP, více viz 3) a na základě tohoto buď vytisknout potřebné údaje (při protokolu ARP), či zavolat další zpracování dle aktuálního protokolu. Nakonec funkce volá tisknutí všech dat daného paketu (viz 2.3).

3 Hlavičky síťových komunikací

Základním principem, jakým jsou zjišťovány základní údaje o protokolech použitých při komunikaci, jsou přetypování na na datový typ struktury s daným offsetem. Pro to je nutné mít buď nějakou dostupnou knihovnu, která má takovou strukturu k dispozici, nebo si případně implementovat vlastní (například v případě ARP viz 3.3).

3.1 Ethernet

Prvním krokem při zpracování protokolu je extrakce ethernetové hlavičky, která je součástí vrstvy síťového rozhraní a která kromě základních údajů o zdrojové a cílové destinaci obsahuje také hodnotu použitého protokolu síťové vrstvy. Způsob získání takové struktury můžeme vidět zde:

```
auto *eptr = (struct ether_header *) bytes + offset;
```

Kde `ether_header` je struktura z dostupné knihovny, `bytes` jsou data paketu a `offset` je nula (u dalších hlaviček se tato hodnota zvedá o velikost předchozích hlaviček). Z této struktury pak není těžké získat hodnotu následného použitého protokolu v síťové vrstvě (IPv4, IPv6 či ARP)[1].

3.2 IPv4 a IPv6

Podobně jako v předchozím případě lze extrahovat dle předchozího výsledku IPv4 či IPv6 hlavičku (buď dle vlastních informací, nebo nějaké dostupné knihovny). Důležitým rozdílem je, že přetypování provádíme v rámci ukazatele na `bytes` s přičtením velikosti (funkce `sizeof`) ethernetové hlavičky. Dle předchozího výsledku pak také voláme funkci `inet_ntop` buď s parametrem `AF_INET`, nebo `AF_INET6` na zpracování získané IP adresy daného typu.

3.3 ARP

V případě protokolu ARP bylo nutno vycházet z dostupných informací na internetu a nikoliv žádné importované knihovny, která by obsahovala danou hlavičku, protože žádná taková veřejně dostupná zřejmě není. Proto

byla vytvořena struktura, která obsahuje v podstatě jedna ku jedné opsané parametry takové hlavičky dle jednotlivých bajtů (byť jedinou skutečně důležitou informací je v tomto případě zdrojová a cílová adresa). Názvy a velikosti vychází z dostupných informací na Wikipedii[3].

3.4 TCP, UDP a ICMP

Při extrakci IPv4 či IPv6 hlavičky je program schopen rozhodnout, který transportní protokol byl využit, proto jsou rozlišovány protokoly TCP a UDP, jež opět mají v rámci standardních knihoven dostupné hlavičky (opět dochází k přetypování, teď už ale přičítáme jak velikost ethernetové hlavičky, tak i síťové podle daného protokolu) a z nich se pak extrahuje port, na kterém pracují. V případě, že byl z IPv4 (respektive IPv6) přečten jiný protokol než TCP a UDP, automaticky se předpokládá, že se jedná o protokol ICMP (respektive ICMP6).

4 Testování

Program byl primárně tvořen a testován v operačním systému v prostředí WSL[2] s nainstalovaným Ubuntu.

4.1 První testování

V prvních fázích docházelo pouze k testování parametrů a schopnosti přijímat pakety se správným časem, na což nebylo potřeba žádných referenčních nástrojů. V další fázi testování bylo využito nástroje implementované v předchozím projektu předmětu IPK. Jedná se o jednuchý skript, který z dané adresy stáhne jeden (či více souborů). Velkou výhodou tohoto testování bylo, že při jeho spuštení je znám první port a IP adresa cíle (svou adresu pak není těžké zjistit), tudíž nebylo těžké porovnat informace ze snifferu a z daného stahování souboru. S touto množinou možností bylo možné testovat správný čas, offset jednotlivých bajtů v paketu, IP adresy ve verzi 4, některé porty atp. V dalších fázích testování by to však nebylo dostačující, a proto bylo přikročeno k referenčnímu nástroji.

4.2 Testování s nástrojem Wireshark

Nejznámější implementací snifferu packetů je nástroj Wireshark, z něhož do nějaké míry nejspíše vychází i zadání[3]. Z těchto důvodů byl použit pro testování implementovaného snifferu.

K testování docházelo jednoduchým způsobem – implementovaný sniffer byl spuštěn pro nějaké větší množství paketů v prostředí WSL, které díky tomu, že neobsahuje prakticky nic než terminál, samo o sobě má naprosto čistou komunikaci. Tudíž nebyl problém mezi časovým rozdílem paralelního zapnutí Wiresharku, který byl napojen na síť, kterou používá virtuální prostředí. Nejprve proběhlo základní testování s pomocí již dříve zmíněného projektu číslo 1 a „ručního“ srovnání výsledků, které stačilo pro kontrolu správnosti protokolů TCP a UDP.

Následně došlo i ke spouštění složitějších a méně očekávatelných programů (z hlediska toho, že nebyly implementovány autorem). Pro testování připojení ARP byl využit shellovský program `arp`, který sám o sobě umí vyvolat nějakou ARP komunikaci. Pro testování ICMP (a ICMPv6) bylo využito programů `ping` (respektive `ping6`) a adresy google.com (respektive ipv6.google.com). Hlubší testování protokolu IPv6 nebylo plně možné z prostého důvodu, že při testování nebyla k dispozici domácí síť schopná provozovat takové spojení.

4.3 Testování v referenčním stroji

V referenčním stroji nedošlo k tak hlubokému testování jako v případě práce ve WSL, avšak byla ověřena alespoň základní smysluplnost a především zkompilovatelnost v daném prostředí. Program byl primárně testován přes příkazy `ping` a `ping6`.

[2] Návod k instalaci pro zájemce zde https://docs.microsoft.com/en-us/windows/wsl/install-win10.

[3] Nástroj je volně dostupný zde: https://www.wireshark.org/download.html. Pro Linuxovou distribuci Ubuntu stačí napsat do konzole sudo apt-get install Wireshark (netestováno).

5 Závěr

Výsledkem řešení projektu je funkční analyzátor síťové komunikace, který funguje minimálně v unixovém prostředí Ubuntu, ve kterém jsou nainstalovány potřebné knihovny a překladač g++ jazyka C++ minimálně ve verzi C++17.

6 Reference

[1] Free Software Foundation, I.: ethernet.h. [online], Naposledy navštíveno 17. 4. 2021.
URL `https://sites.uclouvain.be/SystInfo/usr/include/net/ethernet.h.html`

[2] Group, T. T.: TCPDUMPl. [online], Naposledy navštíveno 17. 4. 2021.
URL `https://www.tcpdump.org`

[3] Kolektiv: Advanced Resolution Protocol. [online], Naposledy navštíveno 17. 4. 2021.
URL `https://en.wikipedia.org/wiki/Address_Resolution_Protocol`

[4] Schiffman, M.: The Libnet Packet Construction Library. [online], Naposledy navštíveno 17. 4. 2021.
URL `http://packetfactory.openwall.net/projects/libnet/`