

# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



## **ISA Projekt 2020 / 21**

Petr Pouč - xpoucp01

10. listopadu 2021

# Obsah

<b>1</b>	<b>Implementace kompatibilního klienta</b>	<b>1</b>
1.1	Zpracování vstupních argumentů . . . . .	1
1.2	Komunikace se serverem . . . . .	1
1.2.1	Vytvoření spojení . . . . .	1
1.2.2	Získání odpovědi od serveru . . . . .	2
<b>2</b>	<b>Zachycená komunikace</b>	<b>2</b>
<b>3</b>	<b>Wireshark dissector</b>	<b>3</b>
<b>4</b>	<b>Zdroje</b>	<b>5</b>

# 1 Implementace kompatibilního klienta

Cílem této části práce bylo naimplementovat kompatibilního klienta, který by ve formátu vstupu i výstupu měl co nejvěrněji napodobovat referenčního klienta. Implementace je rozdělena do několika klíčových úseků, které budou popsány v následujících sekcích.

## 1.1 Zpracování vstupních argumentů

Zpracování argumentů se provádí přímo ve funkci main. Pomocí for cyklu procházím všechny hodnoty argv a parsuji na jednotlivé hodnoty (ip adresa, port, příkaz a další argumenty příkazu)

Přípustné vstupy jsou:

```
-h nebo --help
-a <address> <command> nebo --address <address> <command>
-p <port> <command> nebo --port <port> <command>
```

Argumenty značící adresu a port lze libovolně kombinovat, nezáleží u nich na pořadí a zároveň je lze použít zároveň (-a <address> -p <port> <command>). Za hodnotu <command> lze dosadit příkazy register, login, send, fetch, list, logout, Každý z těchto příkazů má ještě vlastní argumenty např. send <user> <subject> <message>.

V případě nesprávného vstupu klient vrací na chybový výstup hlášku “Wrong arguments”.

## 1.2 Komunikace se serverem

Referenční klient podporuje překlad doménových adres a oba druhy protokolu ipv4 i ipv6. Ještě před samotným spojením jsem si tedy musel zjistit o a jaký typ adresy se jedná. Za tímto účelem jsem si vytvořil funkci ipv4\_or\_ipv6, která mi vrátí hodnotu 4 nebo 6. Na základě této hodnoty poté vytvářím samotný socket.

```
//vytvoření socketu
if(protocol == 4){
    socket_data = socket(AF_INET , SOCK_STREAM , 0); //socket pro ipv4
}else if(protocol == 6){
    socket_data = socket(AF_INET6 , SOCK_STREAM , 0); //socket pro ipv6
}else{
    printf("Could not create socket");
    exit(1);
}
```

Obrázek 1: Vytvoření socketu

Funkce využívá vestavěnné funkce getaddrinfo, která stejně jako referenční klient podporuje překlad doménových jmen.

### 1.2.1 Vytvoření spojení

Celé spojení se serverem probíhá ve funkci socket\_connect, zde volám funkci connect, která jako argumenty bere již vytvořený socket, adresu ukazující na strukturu sockaddr a délku struktury sockaddr. Pokud funkce proběhne úspěšně, je navázané spojení se serverem.

## 1.2.2 Získání odpovědi od serveru

Nyní již máme vytvořené spojení se serverem a můžeme na něj posílat dotazy, na které server bude odpovídat. Na zaslání dotazu od klienta slouží funkce `Socket_send`. Do této funkce kromě socketu předávám již naparsovaný požadavek, který je pro každý příkaz unikátní. Pro příkaz `register` vypadá požadavek takto: `(register "jmeno" "aGVzbG9v")`.

Na získání odpovědi ze strany serveru slouží funkce `server_recv`. Zde využívám funkci `read`, ta čte získané data a ukládá je do proměnné `server_msg`. Hodnotu proměnné `BUFFER` jsem nastavil na 262143, tzn., že můj klient dokáže od serveru přijmout až 26kB dat. Z odpovědi následně extrahuji token, který se ukládá do souboru `login-token.txt`.

```
//extrakce login-tokenu
int index_start = 22;
int k = 0;
char token[1024] = {0};
while(server_answer[index_start] != '\n'){

    token[k] = server_answer[index_start];
    index_start++;
    k++;
}
//uložení tokenu do souboru login-token.txt
fp = fopen("login-token.txt", "w+");
fprintf(fp, "%s", token);
fclose(fp);
```

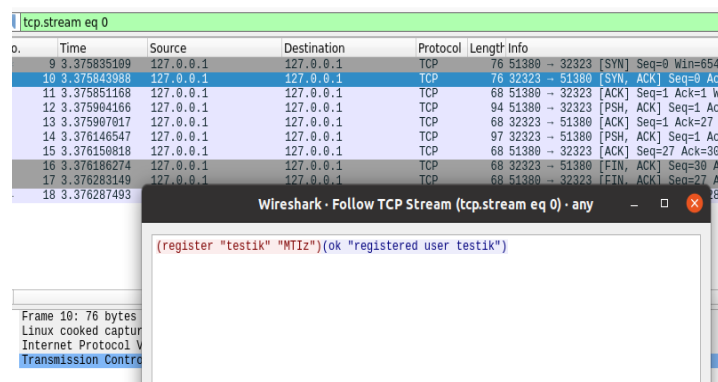
Obrázek 2: Extrakce tokenu z odpovědi serveru

Tento token musíme vždy posílat uvnitř dotazu na server, v opačném případě není server schopen odpovídat.

Všechny hesla v komunikaci mezi klientem a serverem jsou kódované hashovací funkcí `base64`, ta je implementována ve funkci `base64Encoder`.

## 2 Zachycená komunikace

Komunikace která probíhá mezi klientem a serverem je TCP protokol. Pokud není ze strany klienta stanoveno jinak komunikace probíhá na portu 32323 a `localhost` adrese.



No.	Time	Source	Destination	Protocol	Length	Info
9	3.375835109	127.0.0.1	127.0.0.1	TCP	76	51380 → 32323 [SYN] Seq=0 Win=65535
10	3.375843988	127.0.0.1	127.0.0.1	TCP	76	32323 → 51380 [SYN, ACK] Seq=0 Ack=51380
11	3.375851168	127.0.0.1	127.0.0.1	TCP	68	51380 → 32323 [ACK] Seq=1 Ack=1 Win=0
12	3.375904166	127.0.0.1	127.0.0.1	TCP	94	51380 → 32323 [PSH, ACK] Seq=1 Ack=1
13	3.375907617	127.0.0.1	127.0.0.1	TCP	68	32323 → 51380 [ACK] Seq=1 Ack=27 Win=0
14	3.376146547	127.0.0.1	127.0.0.1	TCP	97	32323 → 51380 [PSH, ACK] Seq=1 Ack=27
15	3.376150818	127.0.0.1	127.0.0.1	TCP	68	51380 → 32323 [ACK] Seq=27 Ack=30
16	3.376186274	127.0.0.1	127.0.0.1	TCP	68	32323 → 51380 [FIN, ACK] Seq=30 Ack=27
17	3.376283149	127.0.0.1	127.0.0.1	TCP	68	51380 → 32323 [FIN, ACK] Seq=27 Ack=30
18	3.376287493	127.0.0.1	127.0.0.1	TCP	68	32323 → 51380 [FIN, ACK] Seq=30 Ack=27

Frame 10: 76 bytes on wire (608 bits) captured on interface eth0  
Linux cooked capture  
Internet Protocol Version 4  
Transmission Control Protocol

Wireshark · Follow TCP Stream (tcp.stream eq 0) · any

(register "testik" "MTIz") (ok "registered user testik")

Obrázek 3: TCP protokol

Na přiloženém obrázku lze vidět, jakou strukturu mají zprávy mezi klientem a serverem. Červeně je označená žádost klienta, modře odpověď serveru.

### 3 Wireshark dissector

Dissector jsem se rozhodl psát v jazyce Lua, z důvodu velkého množství dokumentace pro tyto účely. Odchycenou komunikaci jsem pojmenoval jako `ISA Protocol`. Informace o protokolu ve formě stromu jsem řešil pomocí takzvaných “fields”.

```
local isa_fields =
{
  RAW_msg = ProtoField.string("ISA.raw", "Raw Message", base.utf8),
  err_code = ProtoField.string("ISA.version", "Response", base.utf8),
  sender = ProtoField.string("ISA.sender", "Sender", base.utf8),
  client_command = ProtoField.string("ISA.command", "Command", base.utf8),
  payload = ProtoField.string("ISA.command", "Payload", base.utf8),
  payload_sub = ProtoField.string("ISA.command", "Message object", base.utf8),
  payload_body = ProtoField.string("ISA.command", "Message body", base.utf8),
  message_length = ProtoField.uint8("ISA.length", "Length", base.DEC),
  request_id = ProtoField.uint8("ISA.type", "Req_id", base.DEC, msgtype_valstr),
  opcode = ProtoField.uint8("ISA.type", "Opcode", base.DEC, msgtype_valstr),
  response_to = ProtoField.uint16("ISA.response", "Response", base.DEC),
  src_F = ProtoField.string("ISA.src", "Source"),
  dst_F = ProtoField.string("ISA.dst", "Destination"),
}
isa_protocol.fields = isa_fields
```

Obrázek 4: Vytvoření jednotlivých komponent stromu pro informace o protokolu

Hlavní část probíhá ve funkci `dissector`. Prvně si zjistím velikost bufferu, a jelikož protokol TCP odpovědi segmentuje, znamená to, že buffer nemusí obsahovat celou zprávu. Tuto nepříjemnost řeší následující kód:

```
if buffer(length - 1, 1):string() ~= ")" then
  pktinfo.desegment_len = DESEGMENT_ONE_MORE_SEGMENT
  pktinfo.desegment_offset = 0
  return
end
```

Obrázek 5: Segmentace odpovědi

Následně v dissektoru získám původní odpověď od serveru, tu mám v programu pojmenovanou jako `raw_message`. Tuto proměnou dále parsuju na jednotlivé složky, jako například adresa, port, nebo délka zprávy, a ukládám je do stromové struktury.

Odpověď pomocí funkce `Split(string_raw, " ")` rozdělím na jednotlivé části oddělené mezerami a následně si zjistím příkaz, který klient poslal serveru.

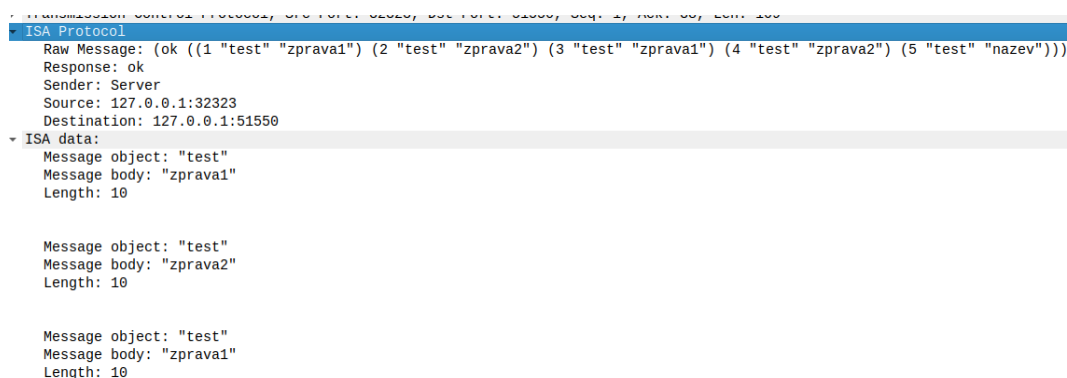
```
--COMMAND
command = (parsed_args[1]):sub(2)
if(sender_msg == "Client") then
  tree_data:add(isa_fields.client_command, command)
end
```

Obrázek 6: Získání příkazu, který uživatel zaslal serveru

Odpověď každého příkazu má jiný formát. V kódu je jasně oddělené formátování odpovědi pro jednotlivé příkazy. Na parsování používám převážně tyto funkce: `Split`, `append`, `find`, `string.gsub`.

```
if(command == "send") then
    --smazání uvozovek
    subject = string.gsub(parsed_args[4], "'", " ")
    body = string.gsub(parsed_args[5], "'", " ")
end
```

Obrázek 7: Ukázka parsování pro příkaz `send` a následné přidání do stromové struktury



Obrázek 8: Vzhled dissectoru pro příkaz `list`

Do informací o protokolu vepisuji zprávu, kterou server odpovídá na žádost klienta. Použil jsem příkaz `pinfo.cols.info:append(string_raw)`

Protocol	Length	Info
ISA Protocol	120	Response: (ok "user logged in" "dGVzdDE2MzQ2NjkyNTIwNTkuODU0")
ISA Protocol	130	Request: (send "dGVzdDE2MzQ2NjkyNTIwNTkuODU0" "test" "zprava1...)
ISA Protocol	87	Response: (ok "message sent")
ISA Protocol	130	Request: (send "dGVzdDE2MzQ2NjkyNTIwNTkuODU0" "test" "zprava2...)
ISA Protocol	87	Response: (ok "message sent")
ISA Protocol	135	Request: (send "dGVzdDE2MzQ2NjkyNTIwNTkuODU0" "test" "nazev" ...)
ISA Protocol	87	Response: (ok "message sent")
ISA Protocol	105	Request: (list "dGVzdDE2MzQ2NjkyNTIwNTkuODU0")
ISA Protocol	177	Response: (ok ((1 "test" "zprava1") (2 "test" "zprava2") (3 "...)
ISA Protocol	108	Request: (fetch "dGVzdDE2MzQ2NjkyNTIwNTkuODU0" 2)
ISA Protocol	98	Response: (ok ("test" "zprava2" "text"))
ISA Protocol	108	Request: (fetch "dGVzdDE2MzQ2NjkyNTIwNTkuODU0" 3)
ISA Protocol	99	Response: (ok ("test" "zprava1" "teloi"))
ISA Protocol	107	Request: (logout "dGVzdDE2MzQ2NjkyNTIwNTkuODU0")
ISA Protocol	85	Response: (ok "logged out")

Obrázek 9: Informace o protokolu

## 4 Zdroje

<https://www.geeksforgeeks.org/encode-ascii-string-base-64-format>

<https://gitlab.com/wireshark/wireshark/-/wikis/Lua/Dissectors>

<https://gitlab.com/wireshark/wireshark/-/wikis/Lua/Examples>