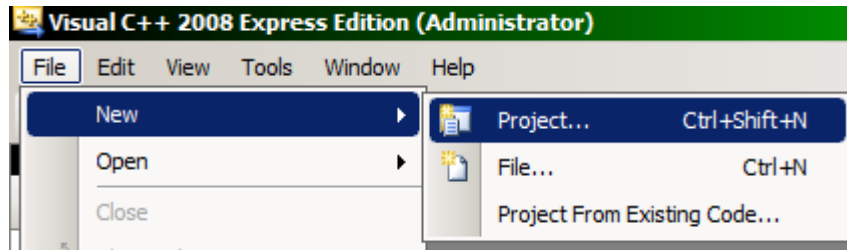


Table of Content

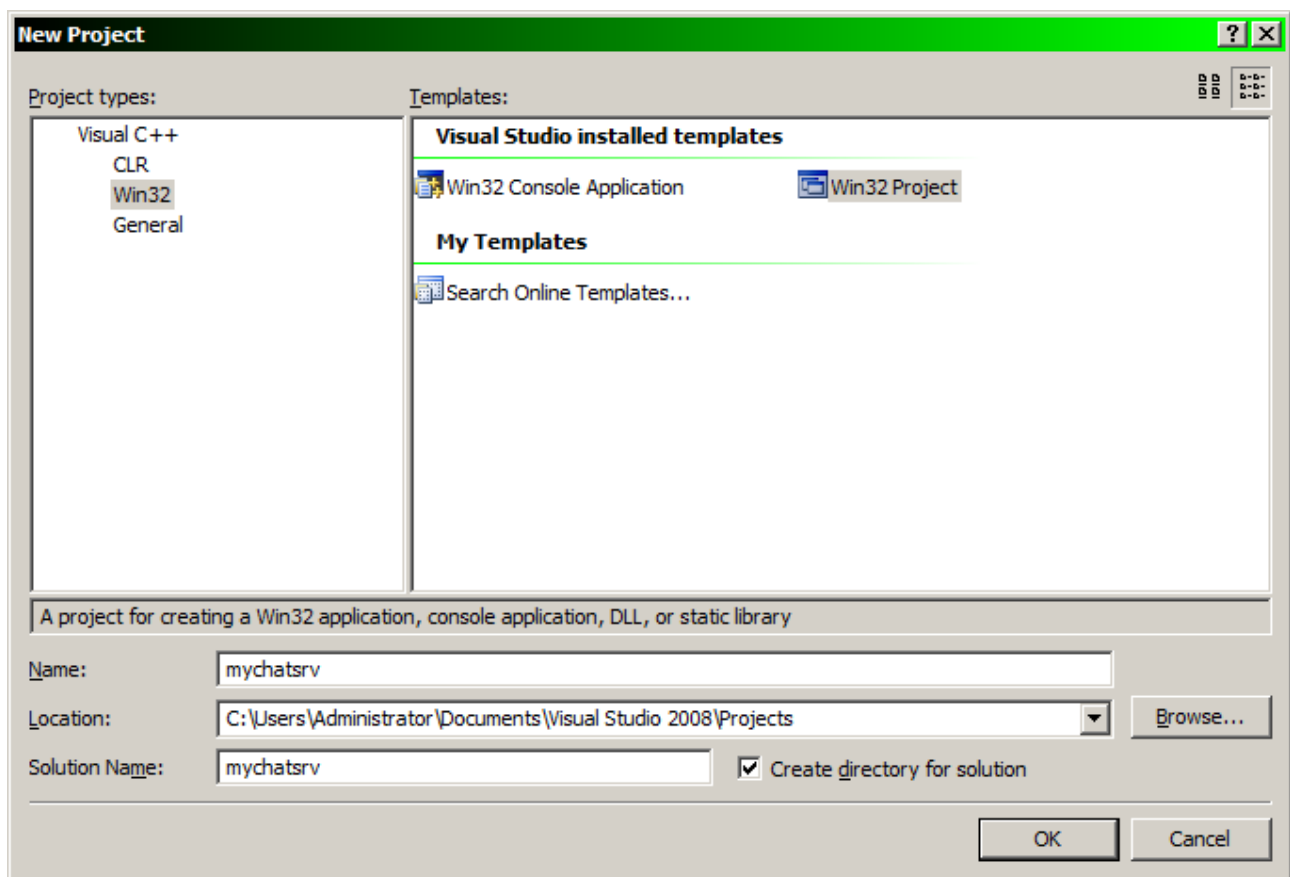
Write the sample application.....	2
tear down the ssl layer.....	8
Communication flow between the client and server.....	22
Process of connection.....	22
Client message sending.....	24
Server message delivery.....	25
Server Send Satus.....	26
Creation of LUA script.....	27
Protocol creation.....	27
init function.....	27
Dissector Function.....	27
Register Protocol.....	27
Whole Code at this milestone.....	27
set protocol name.....	29
Write the dissector.....	29
cycle to analyze the actual part of the packet.....	29
The whole script until now.....	31
extend the dissector with missing packet types.....	32
The whole script.....	34
fuzzing.....	36
Autogenerate the datamodel by struct2peach.....	36
Autogenerate the datamodel by peachshark.....	36
Create the peach .xml file.....	39
Create the datamodel.....	44
Create the state model.....	55
Create the Agent.....	55
Create the Test section.....	56
Create the Run section.....	56
Whole chat.xml file.....	56
Run the fuzzer.....	60
Fuzz the message sending.....	61

Write the sample application

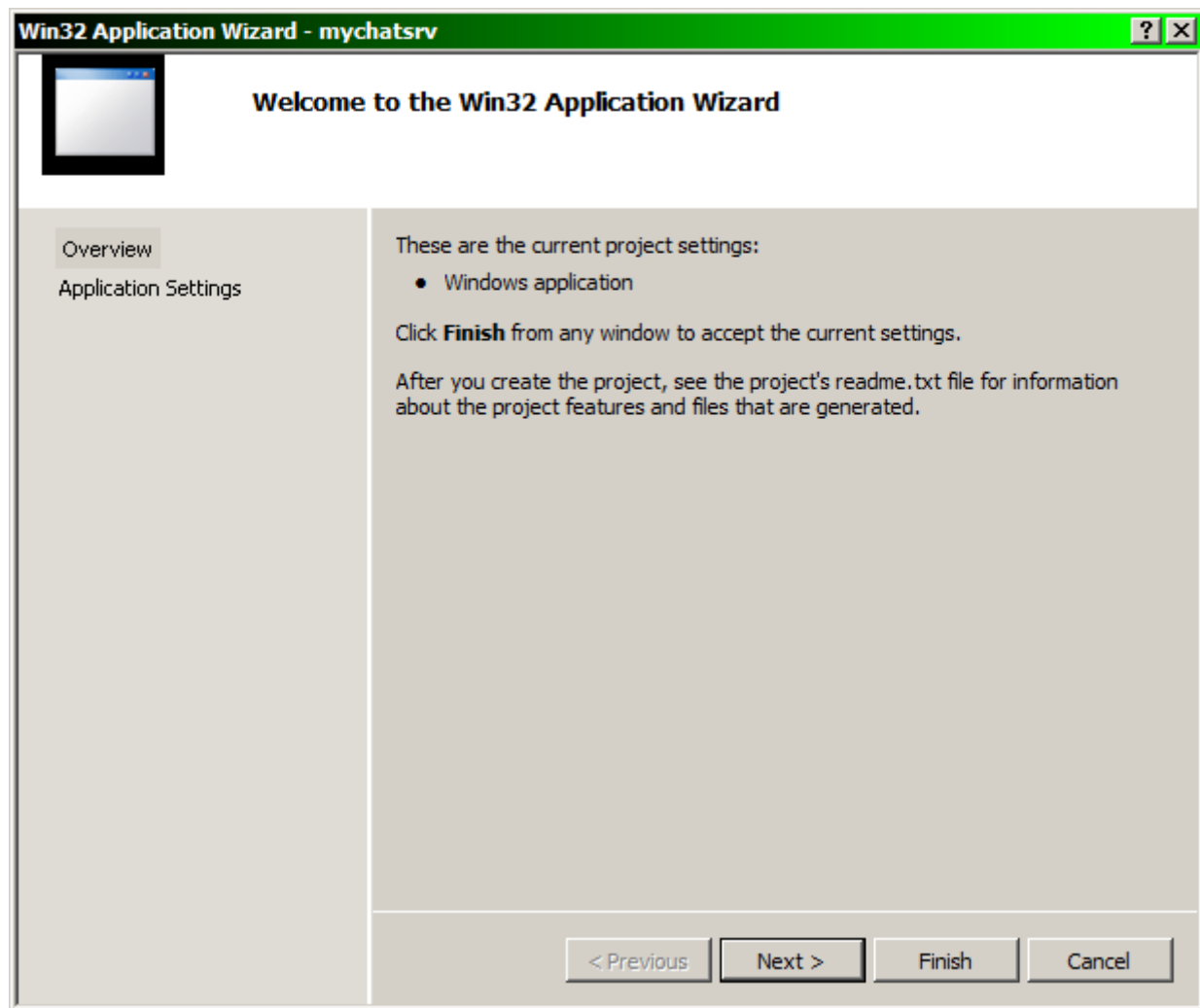
Create a new project



Select the Win32 \ Win32 project, and give it a name



on the welcome screen of the wizard click to the next button



select the "Windows Application", then click to the finish button.

**Application Settings**

Overview

Application Settings

Application type:

- ☒ Windows application
☐ Console application
☐ DLL
☐ Static library

Add common header files for:

- ☐ ATL
☐ MFC

Additional options:

- ☐ Empty project
☐ Export symbols
☒ Precompiled header

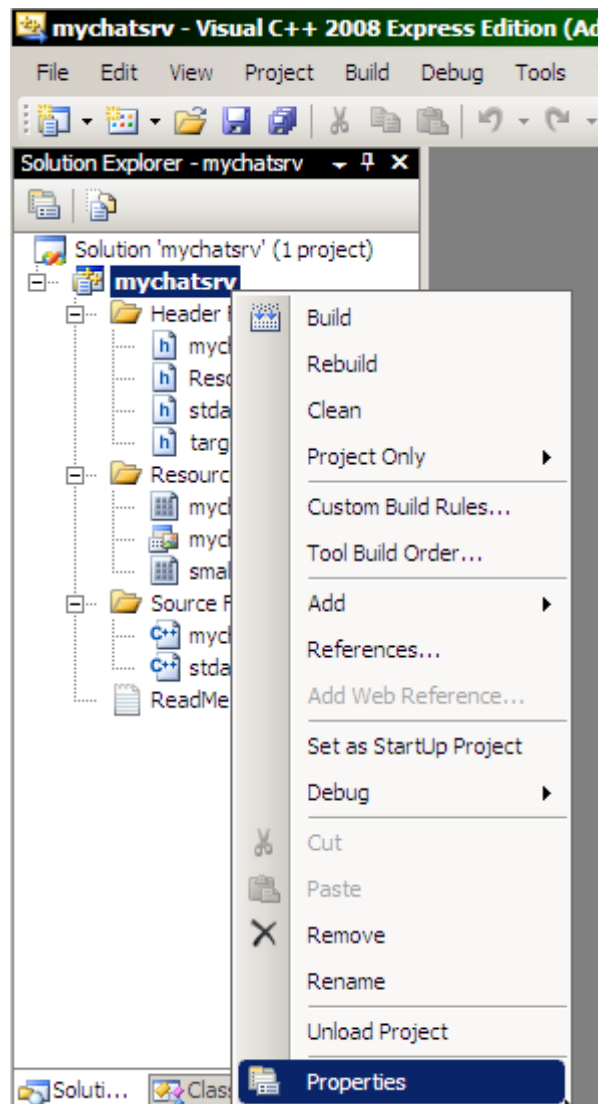
< Previous

Next >

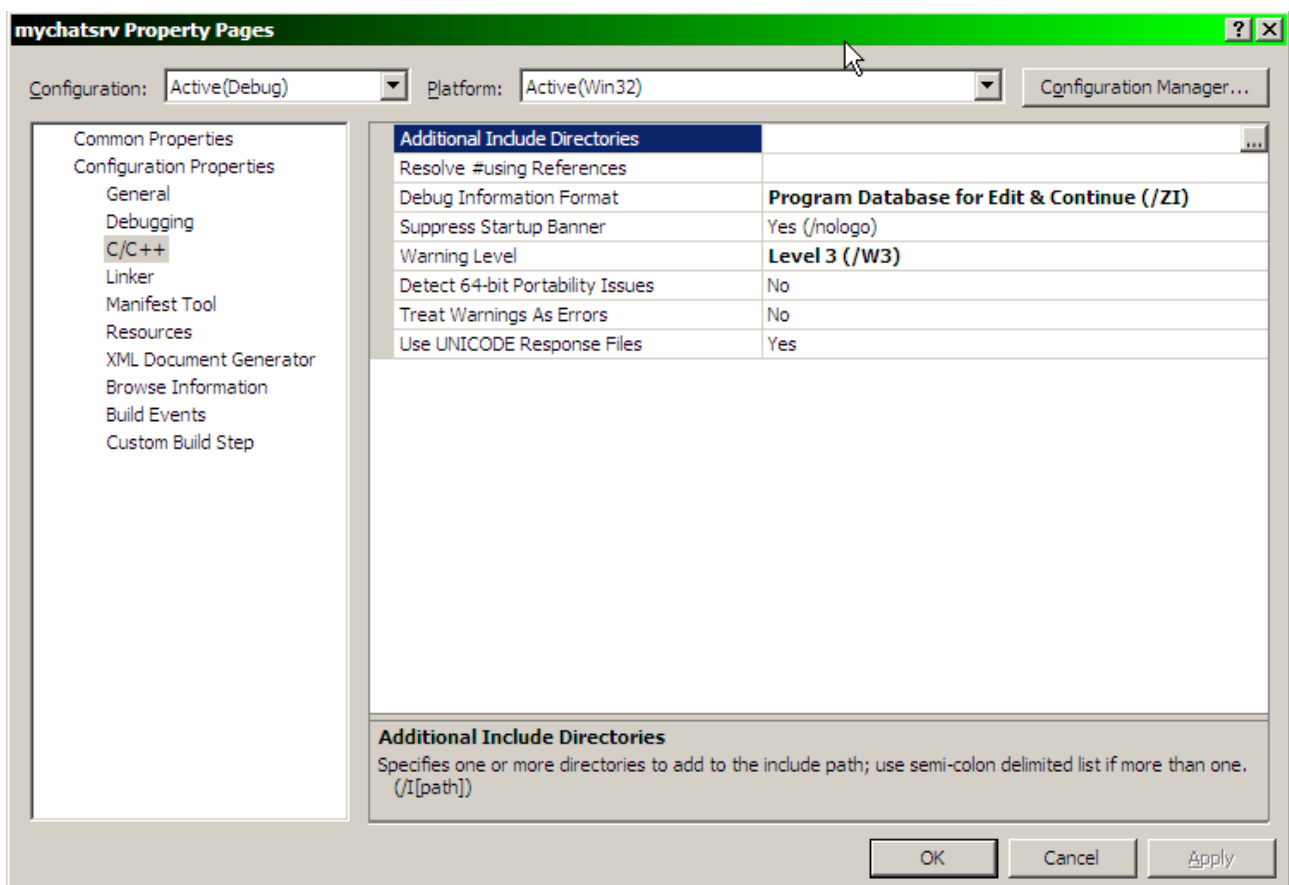
Finish

Cancel

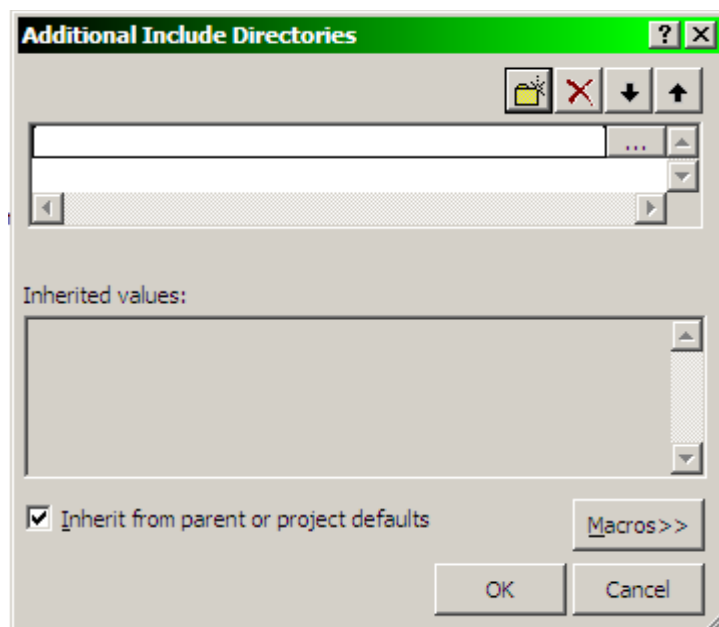
Right click on the new project and from the popup menu select the properties command:



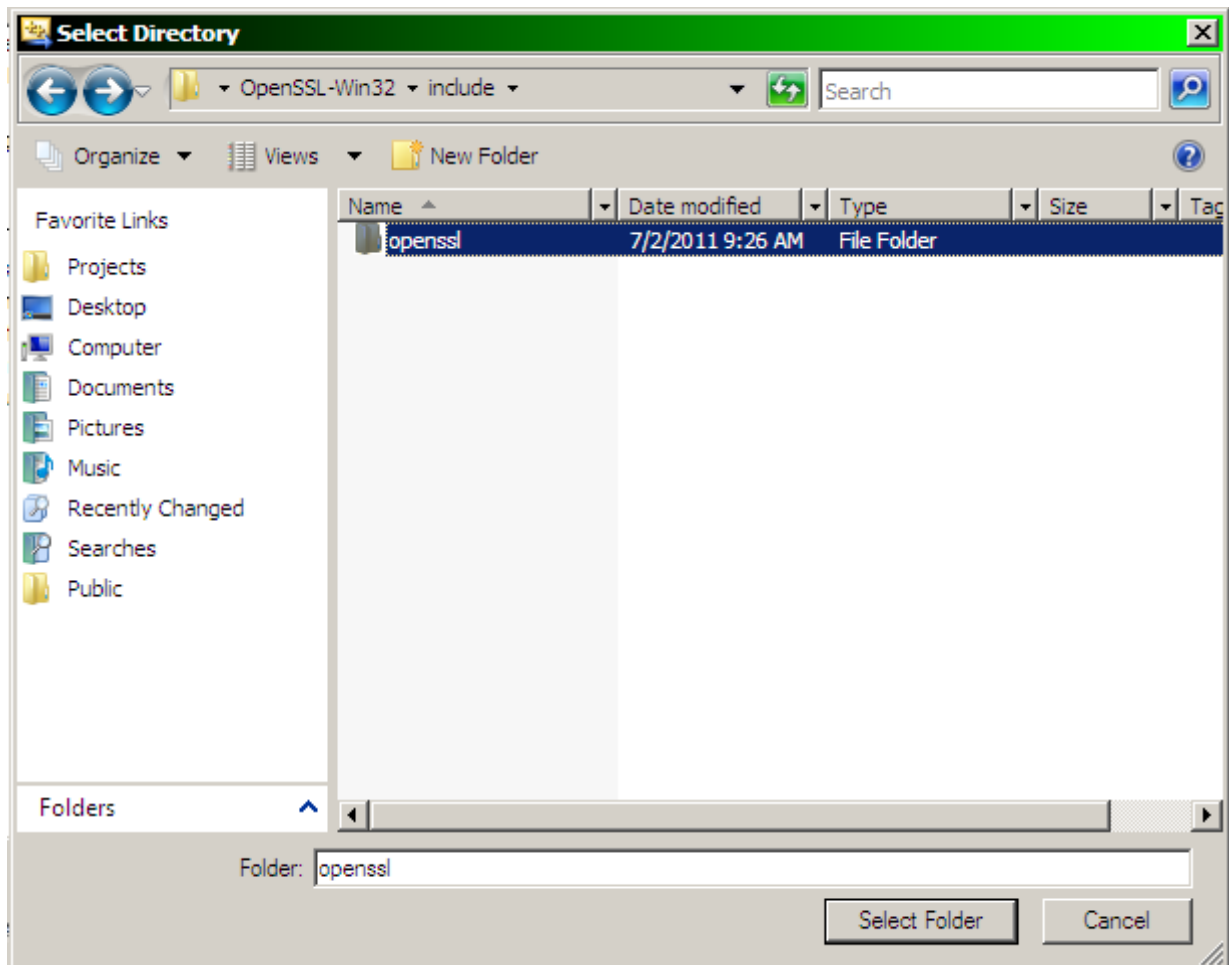
Go to the "Configuration Properties", and within that the C/C++. There select the "Additional Include Directories", and click on the "..." icon at the right side



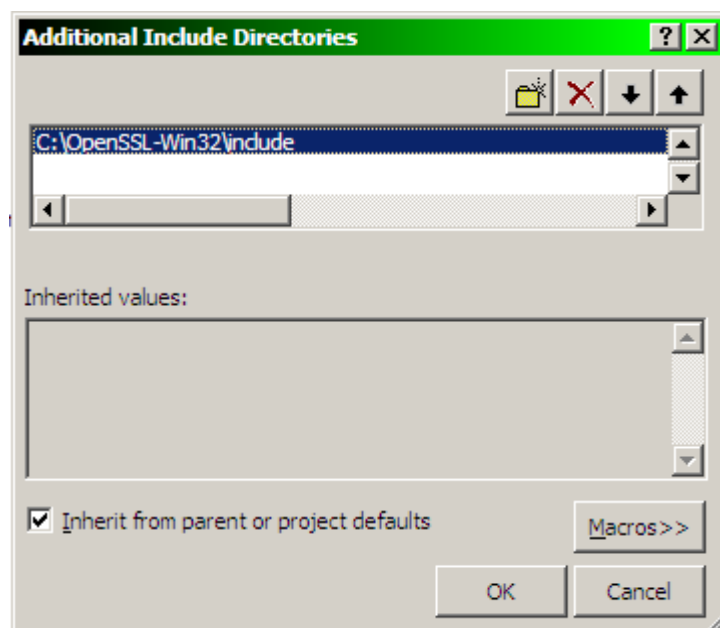
click on the new line icon, and click to the "..." icon.

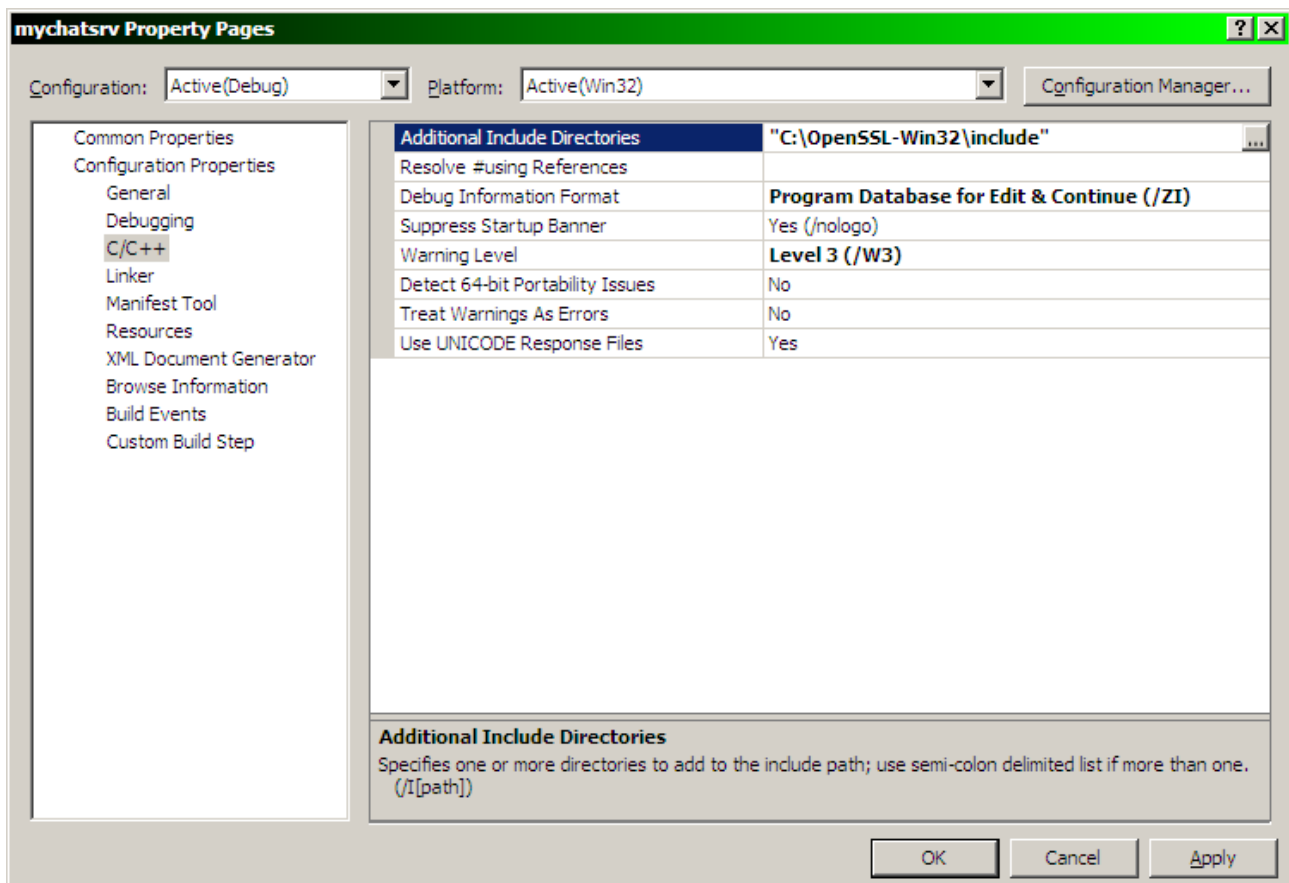


Select the openssl install directory, within that the include directory: "C:\OpenSSL-Win32\include"

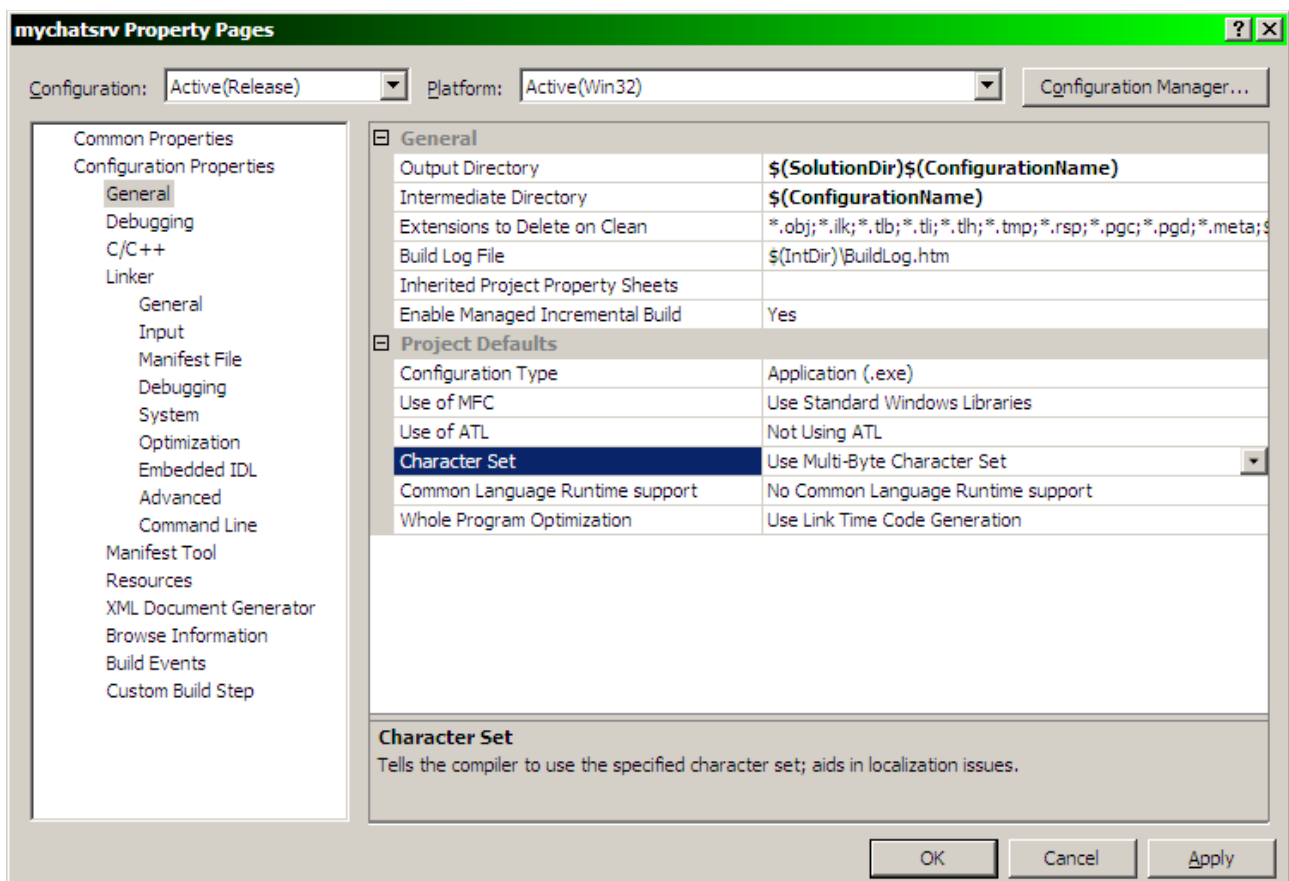


Click to the OK button

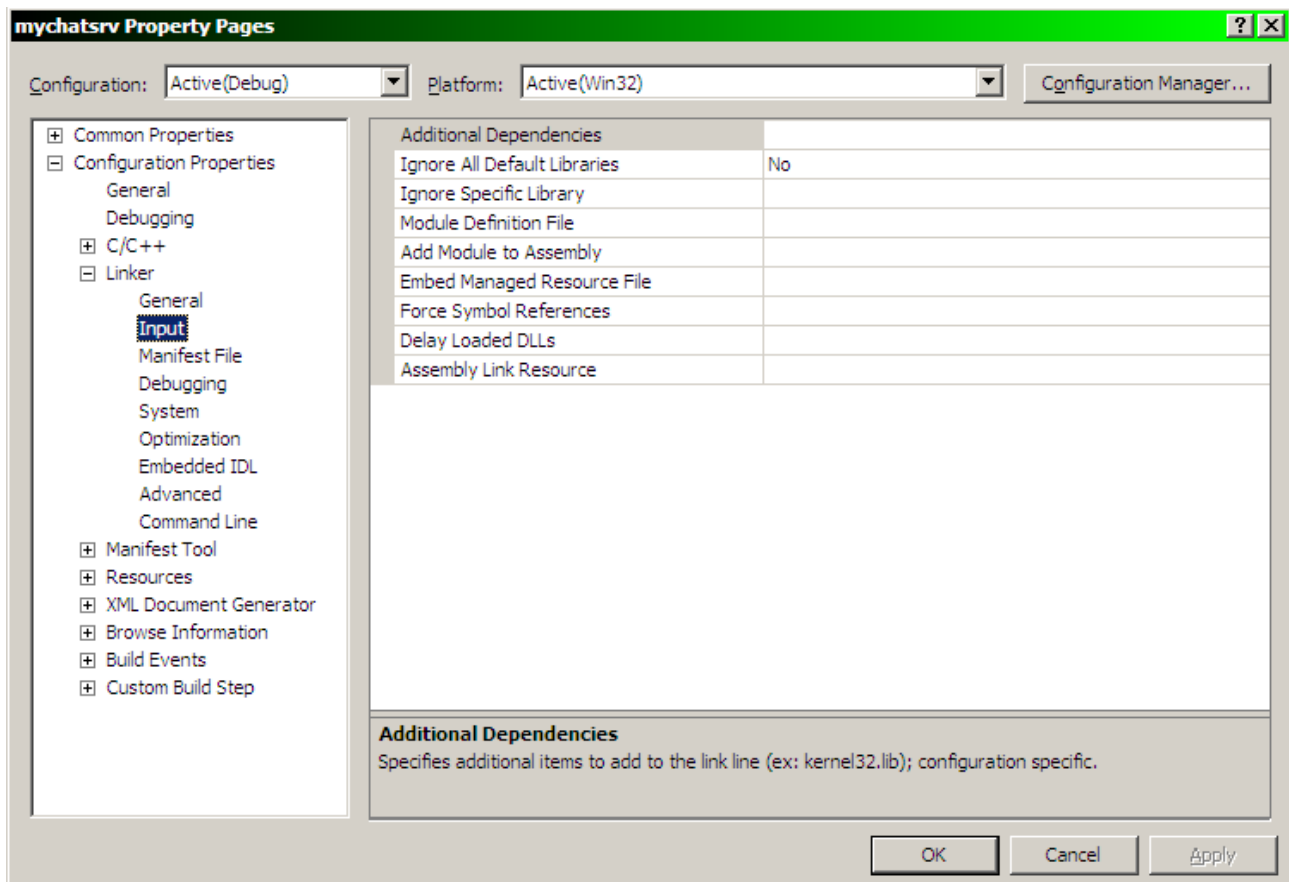




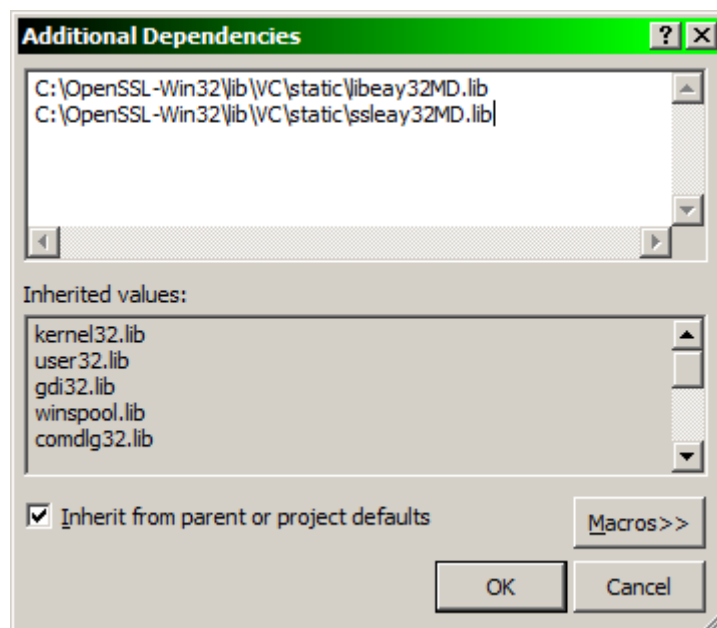
Then go to the "Configuration Properties \ General" and set the "Character Set" to Use Multi-Byte Character Set.



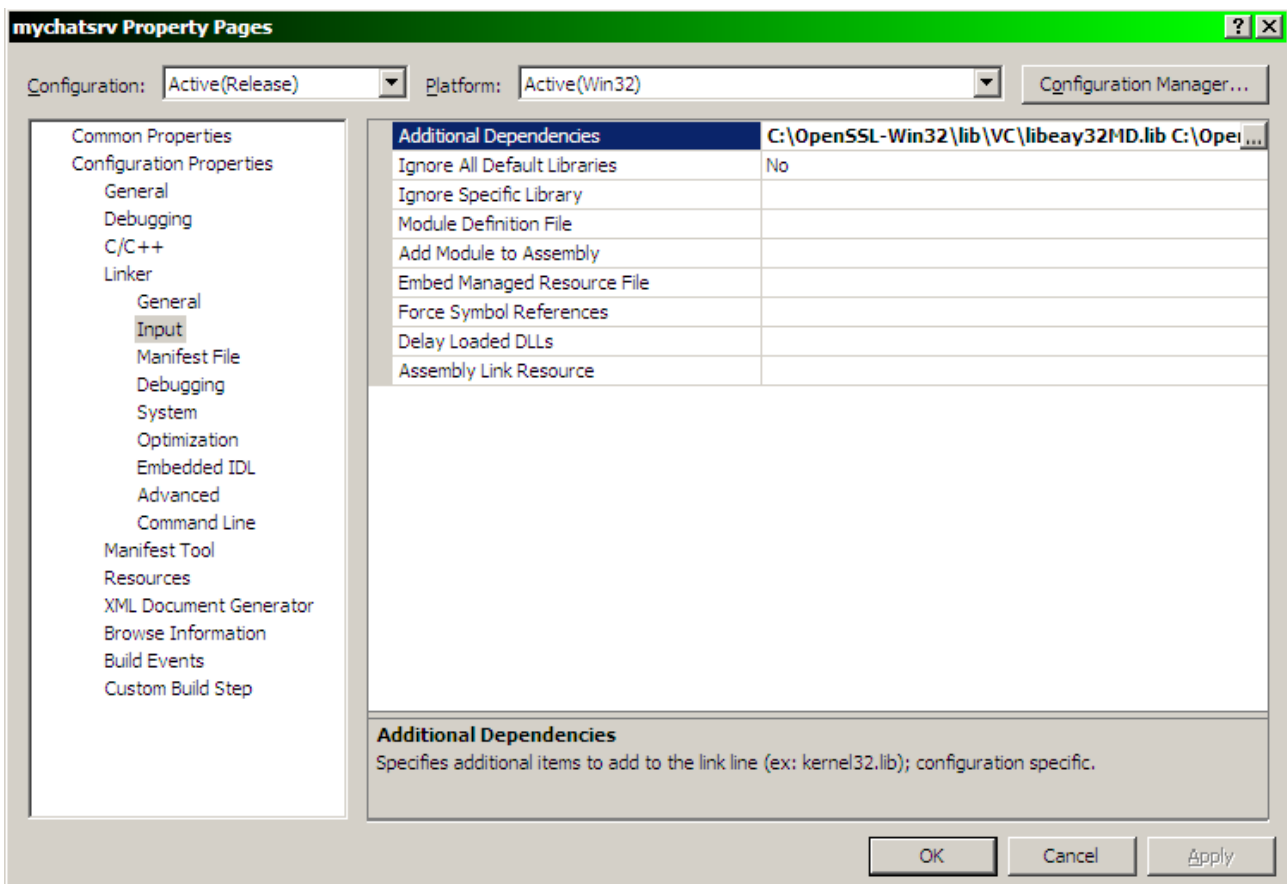
Then go to the "Configuration Properties \ Linker \ Input". Then select "Additional Dependencies", and click on the "..." icon at the right side of it.



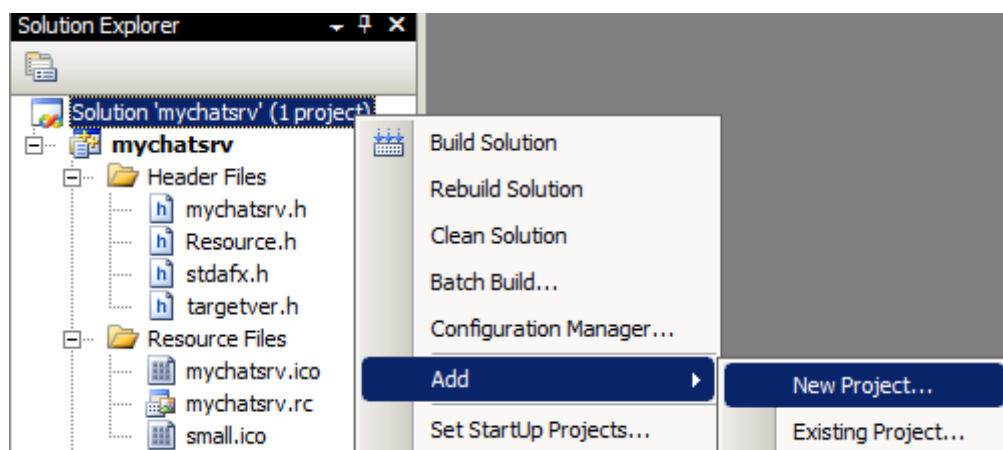
At the new windows add the libeay32MD.lib and ssleay32MD.lib files from the openssl install directory \ lib \ VC \ static, then click to the OK button.



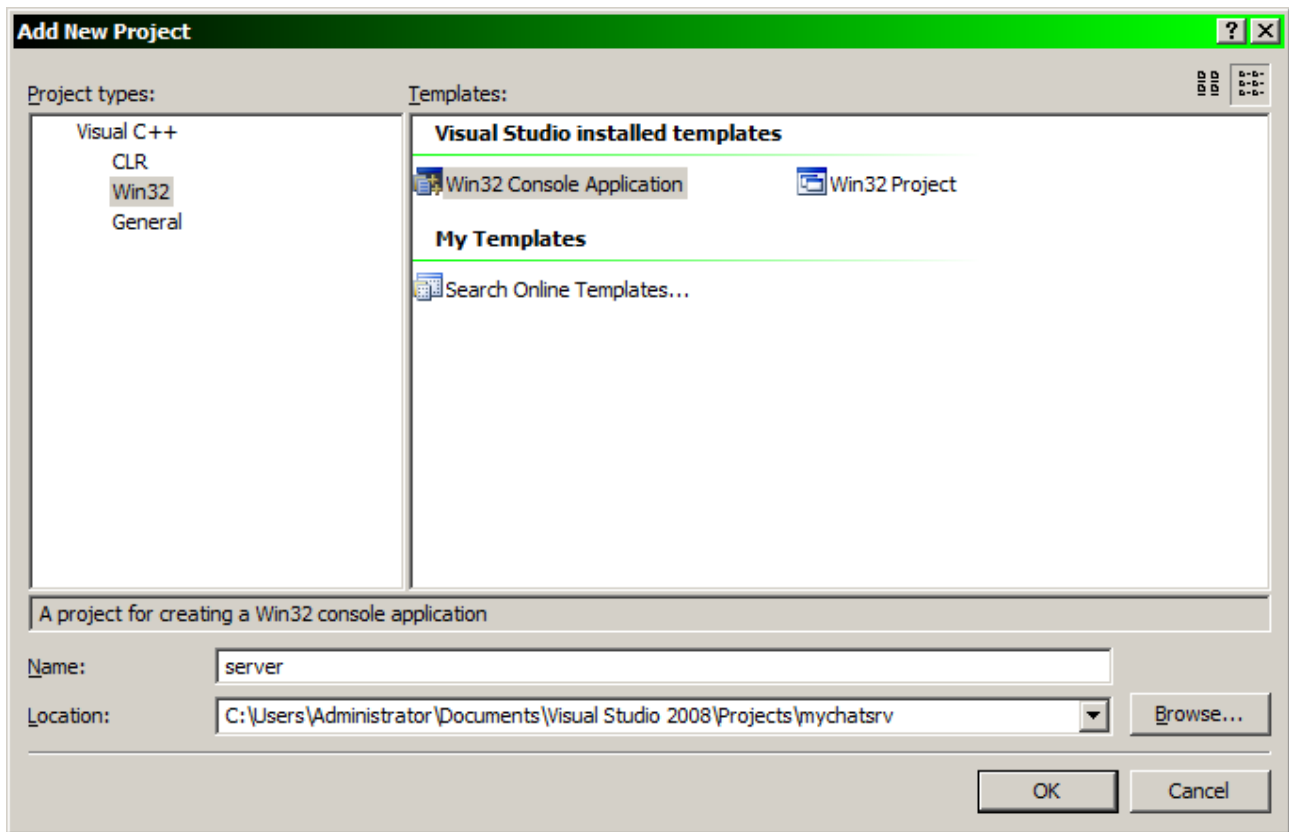
We will get the next:



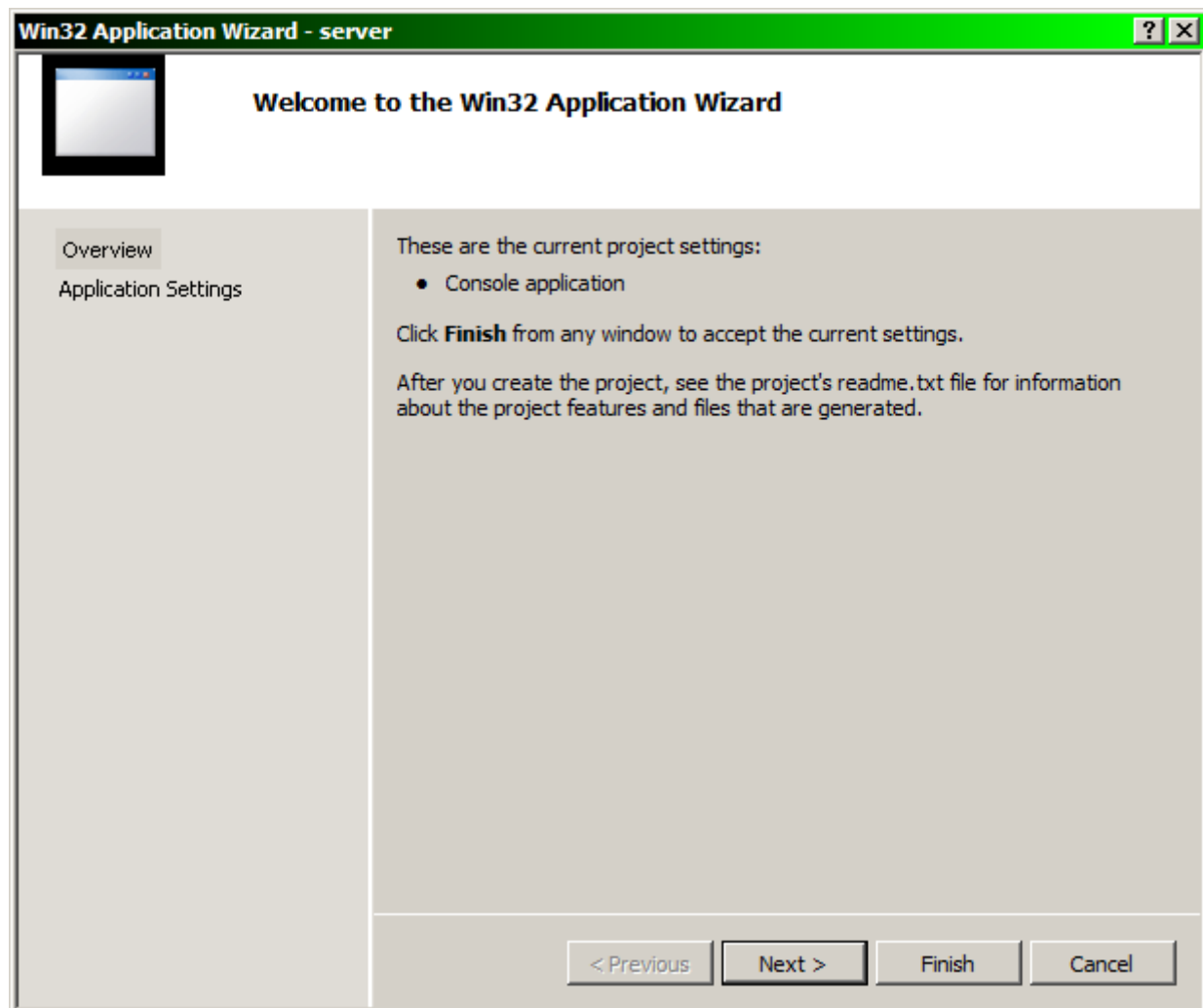
then right click to the Solution and from the popup menu select the Add \ New Project ... command.



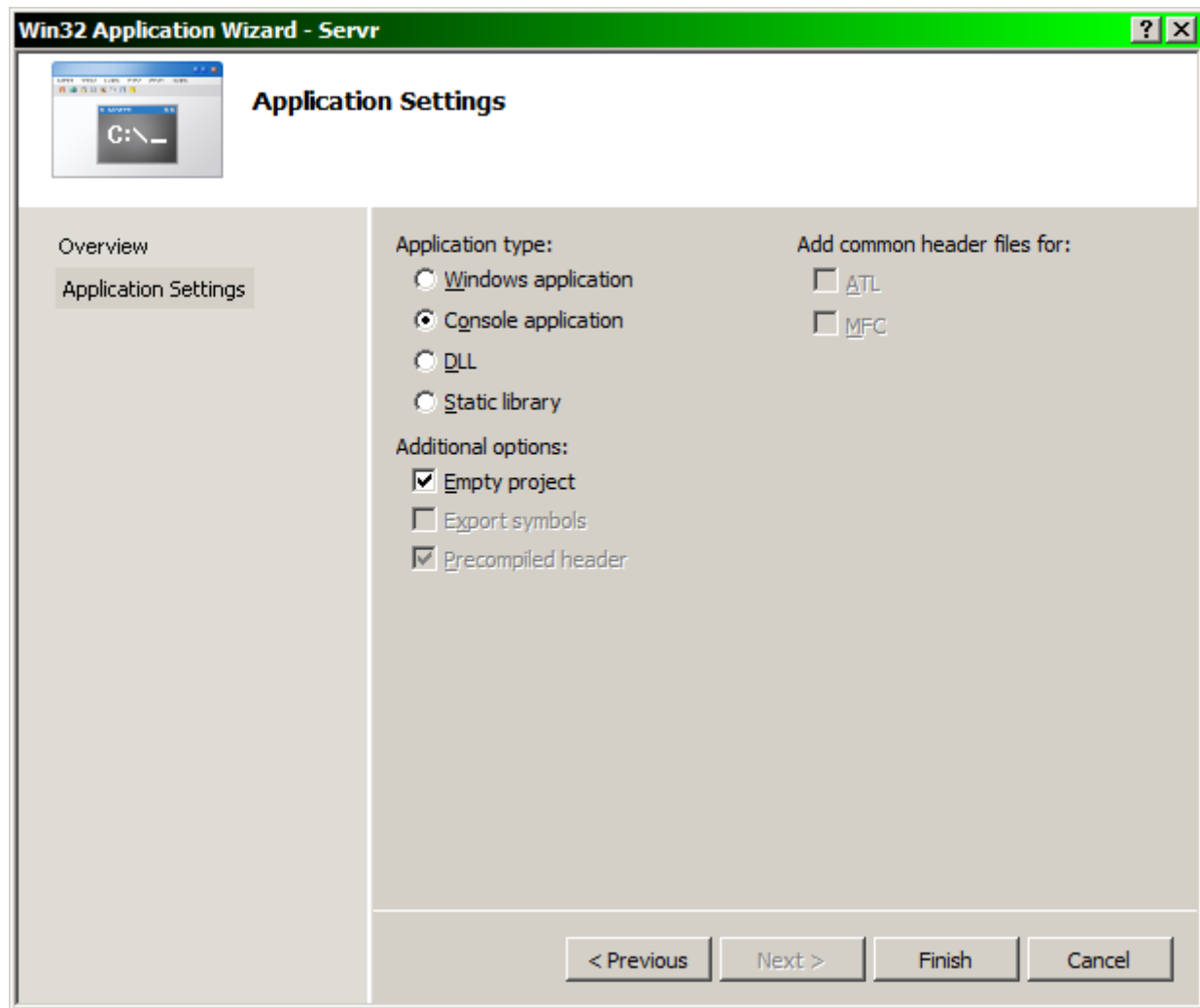
Select the Win32 \ Win32 Console Application. Give it a name, and click to the OK button.



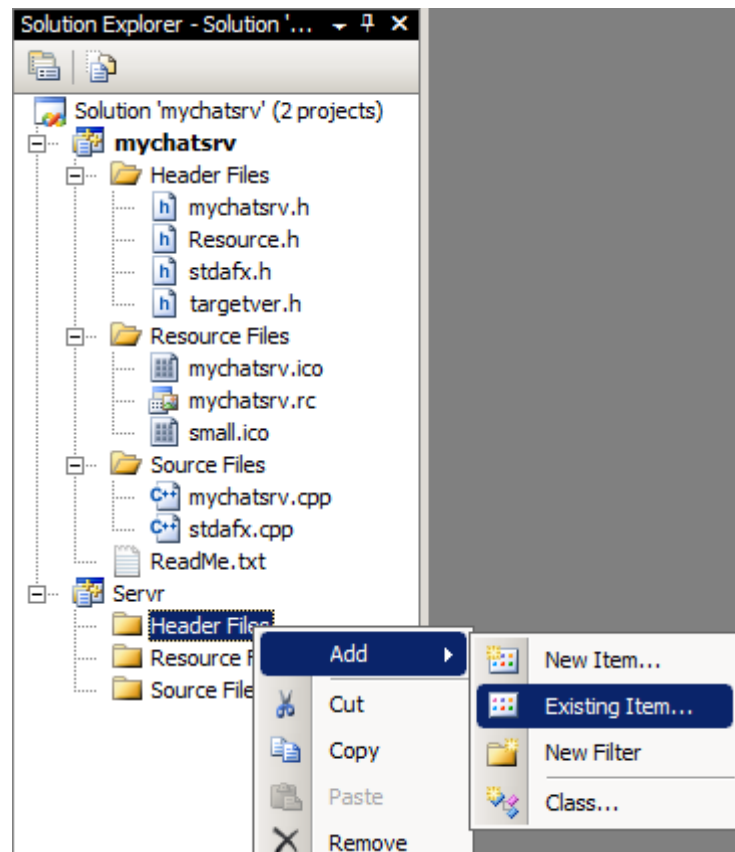
On the Welcome screen click to the next button



Then click to the finish button

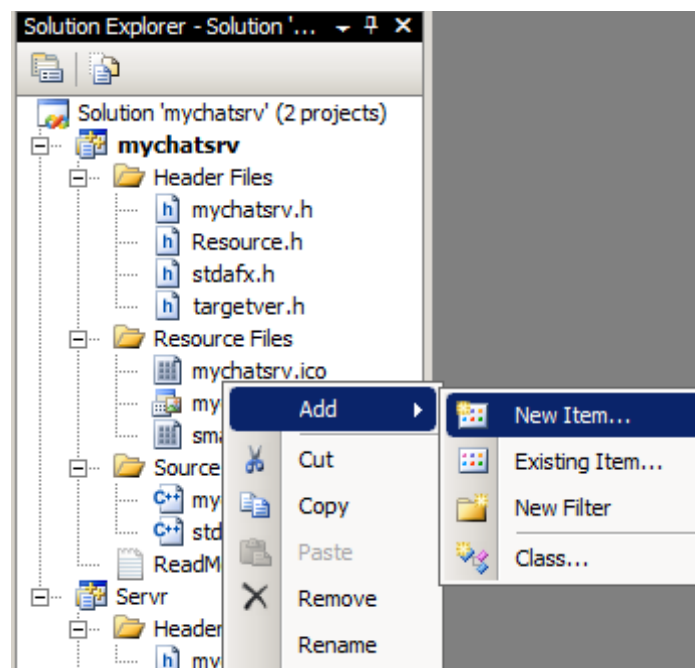


right click to the "Header Files" of this new project, and select the Add \ Existing item

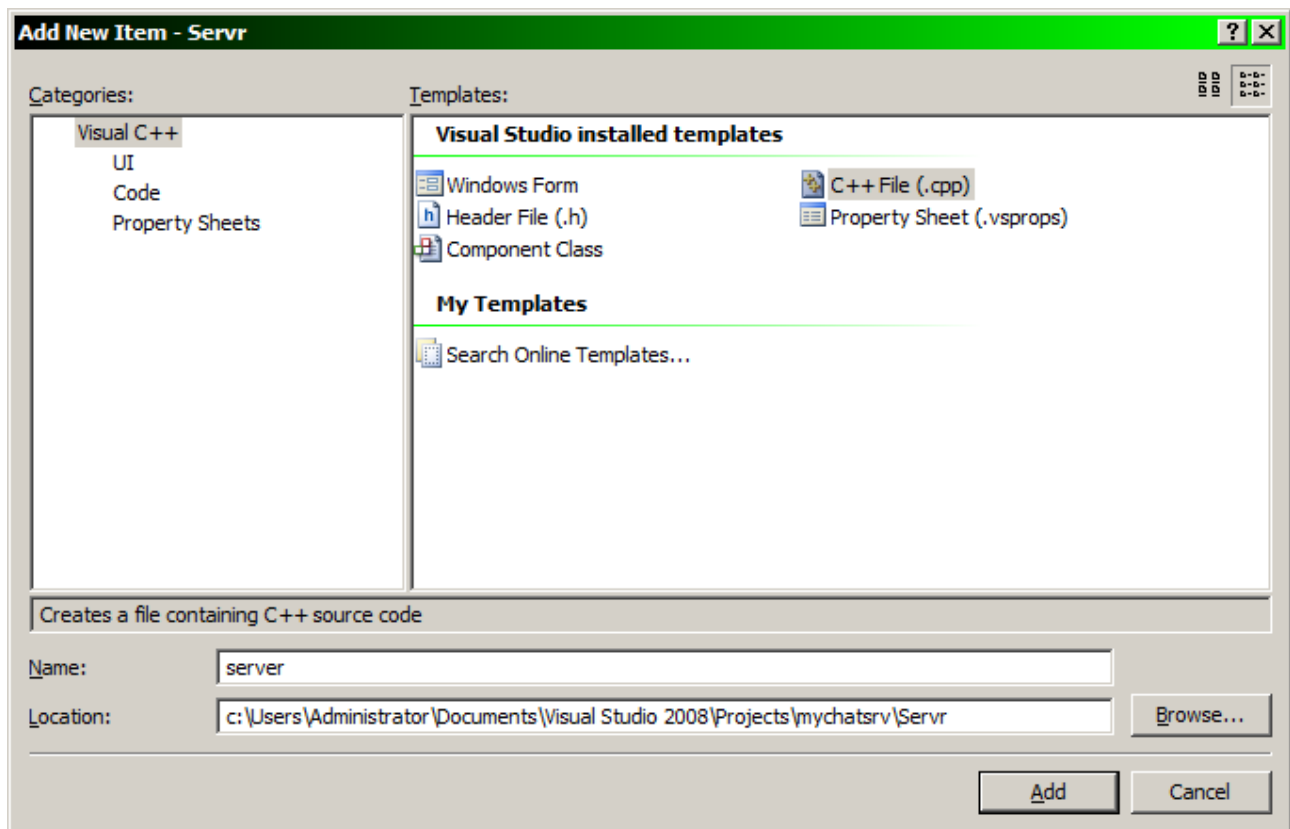


Add the mychatsrv.h from the previous project.

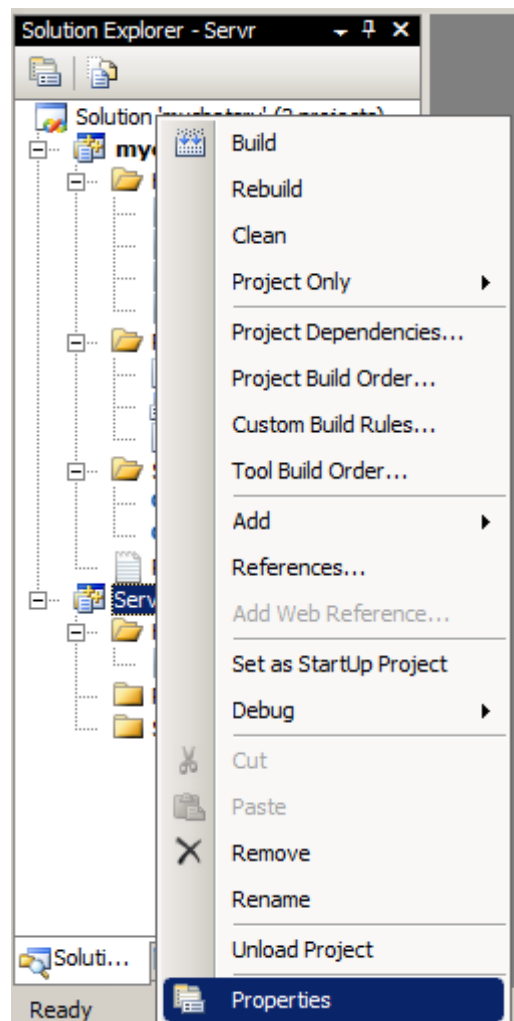
Right click again to the "Source Files" of this project, and select the Add \ New Item ... command



Select the C++ file from the dialog box, and give a name to the file:



Then right click to this project, and select the properties command from the popup menu.



peel the ssl layer

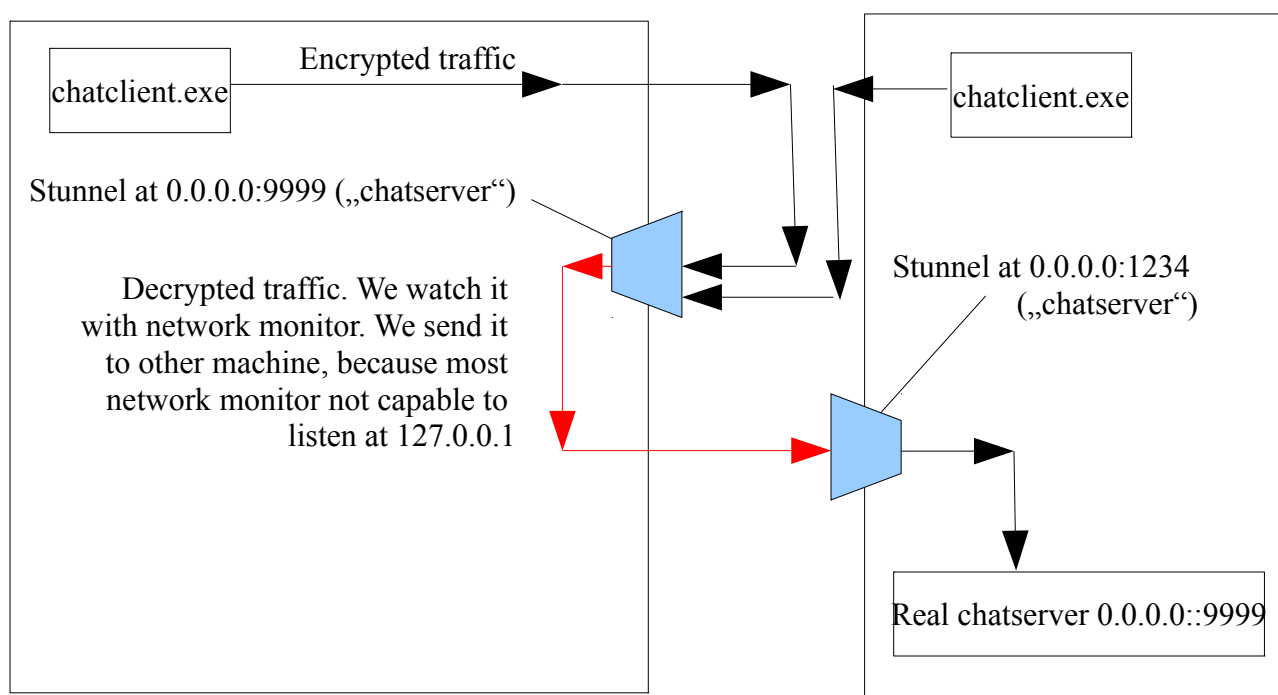
Unfortunately the traffic is ssl encrypted. If we were just try to fuzz it in this way we were fuzz only the openssl layer, but we want to fuzz the encrypted data within the ssl tunnel. To do it we must peel the ssl layer.

To do it we need a certificate (we will use now the openssl to generate one, but any other method can be used). The windows version can be downloaded for example from the next URL:

<http://www.openssl.org>

And we need the Stunnel application. The Stunnel is able to create an ssl tunnel for us.

First we capture the traffic, to see the communication



Install Openssl on the usual way, then go to the new openssl directory:

```
cd C:\OpenSSL-Win32\bin
```



Create a key to your certificate server:

```
openssl genrsa -des3 -out ca.key 1024
```

```
Administrator: Command Prompt
C:\OpenSSL-Win32\bin>openssl genrsa -des3 -out ca.key 1024
Loading 'screen' into random state - done
Generating RSA private key, 1024 bit long modulus
.....+++++
.....+++++
e is 65537 (0x10001)
Enter pass phrase for ca.key:
Verifying - Enter pass phrase for ca.key:
C:\OpenSSL-Win32\bin>_
```

Then create a public certificate from the keyfile:

```
openssl req -config openssl.cfg -new -x509 -days 1001 -key ca.key
-out ca.cer
```

answer to the questions appropriately

```
Administrator: Command Prompt
C:\OpenSSL-Win32\bin>openssl req -config openssl.cfg -new -x509 -days 1001 -key
ca.key -out ca.cer
Enter pass phrase for ca.key:
Loading 'screen' into random state - done
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:HU
State or Province Name (full name) [Some-State]:Budapest
Locality Name (eg, city) []:Budapest
Organization Name (eg, company) [Internet Widgits Pty Ltd]:company
Organizational Unit Name (eg, section) []:myou
Common Name (eg, YOUR name) []:chatserver.local
Email Address []:a@chatserver.local
C:\OpenSSL-Win32\bin>_
```

```
openssl x509 -in ca.cer -outform DER -out ca.der
```

```
Administrator: Command Prompt
C:\OpenSSL-Win32\bin>openssl x509 -in ca.cer -outform DER -out ca.der
C:\OpenSSL-Win32\bin>_
```

by default the openssl want to use the demoCA directory so we crate it:

```
cd \OpenSSL-Win32\bin
md demoCA
cd demoCA
md newcerts
cd ..
```

Create an empty file called c:\OpenSSL-Win32\bin\demoCA\index.txt

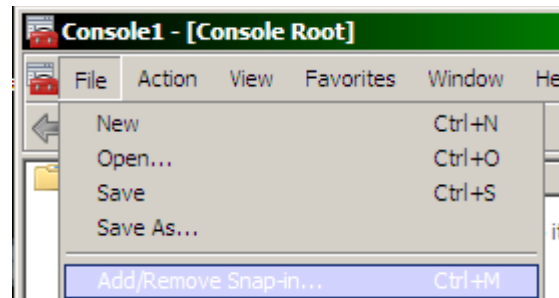
Create file called c:\OpenSSL-Win32\bin\demoCA\serial (without extension). The content of it should be 01

Add the CA certificate to our trusted root certificate store.

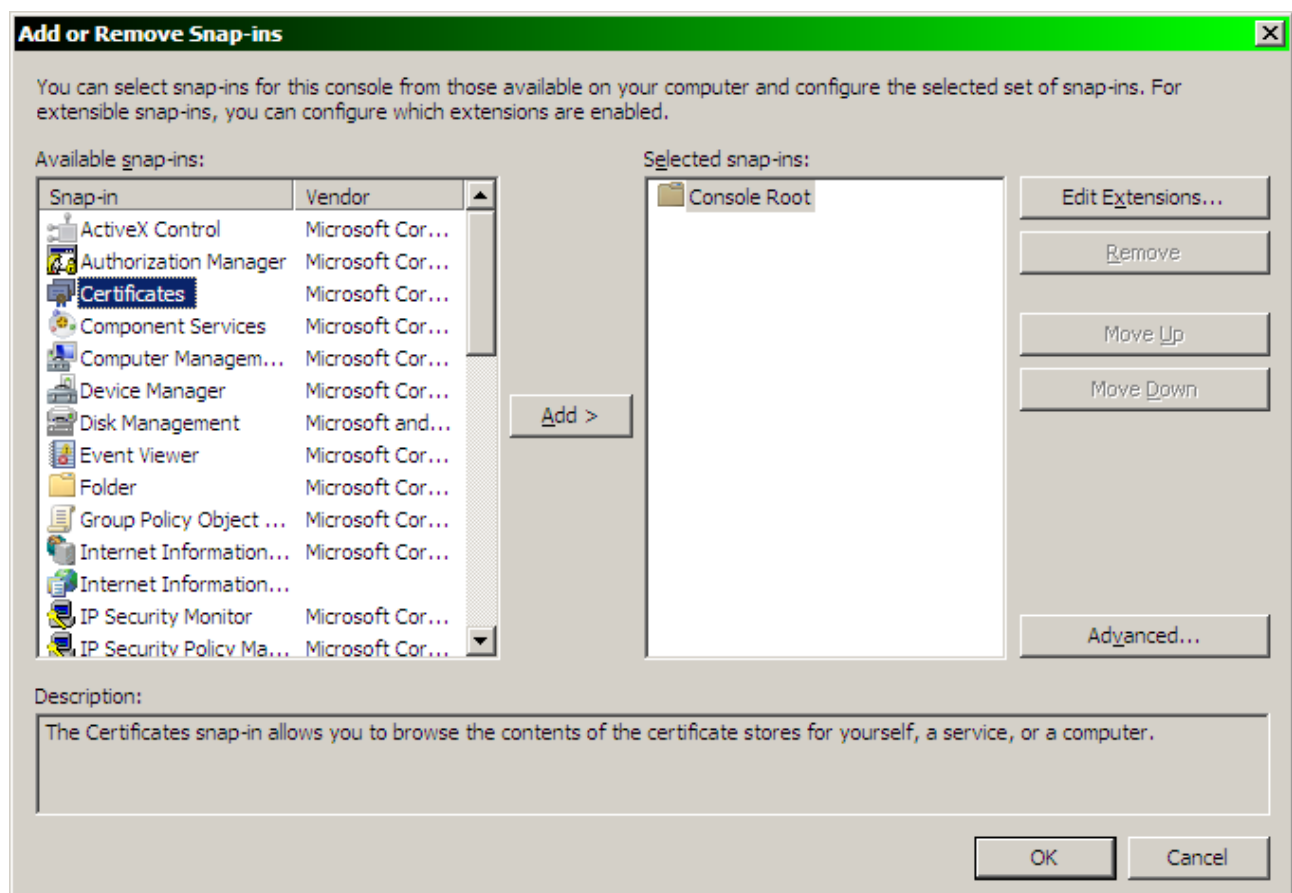
Start an mmc console



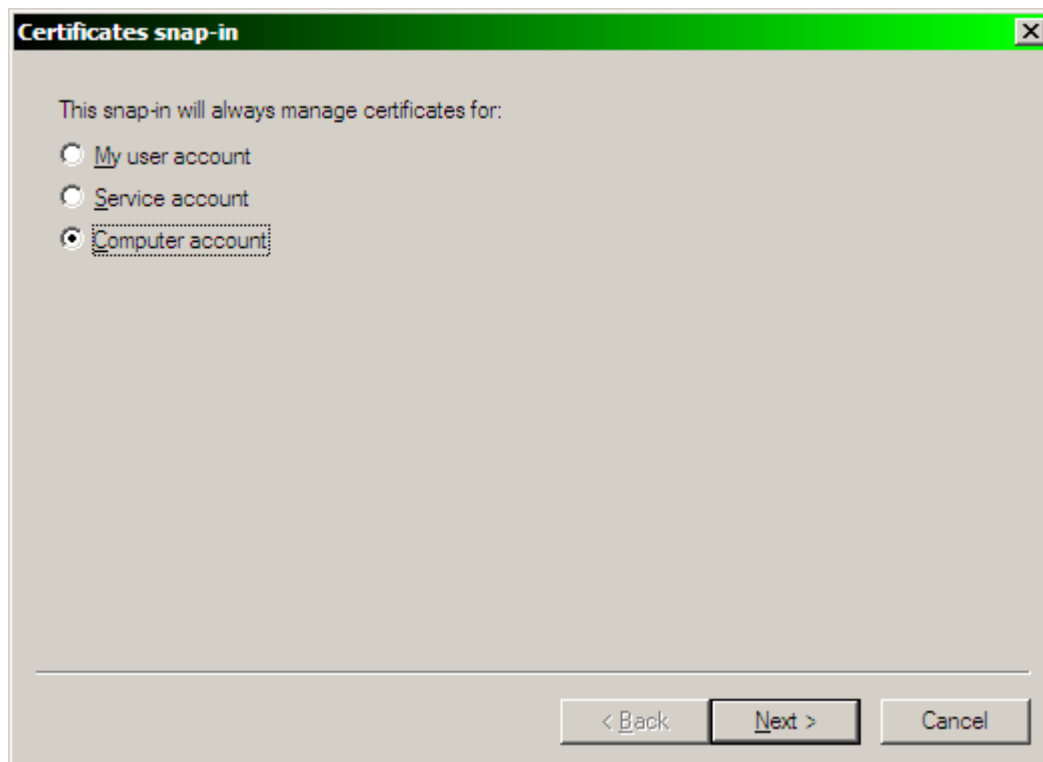
select file \ add Remove snapin:



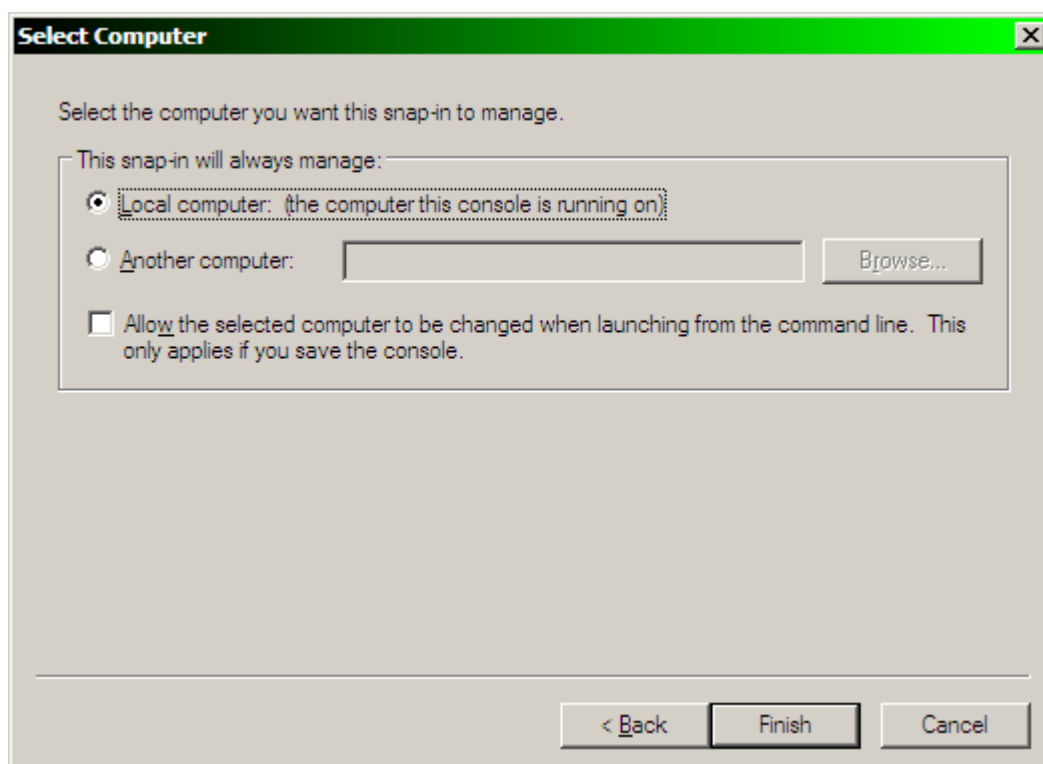
from the popup window select certificate then press the add button



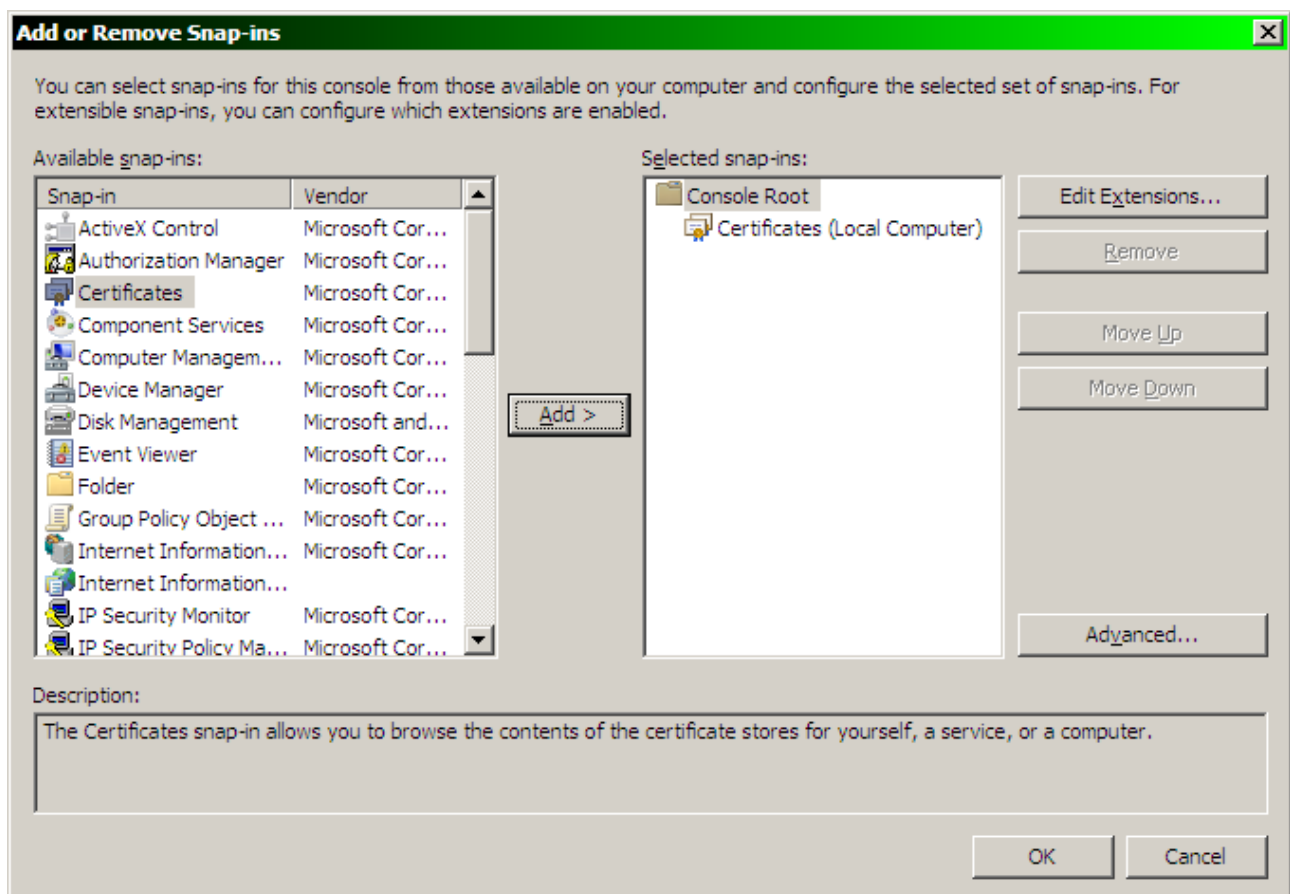
in the new window select computer account then click on the next button



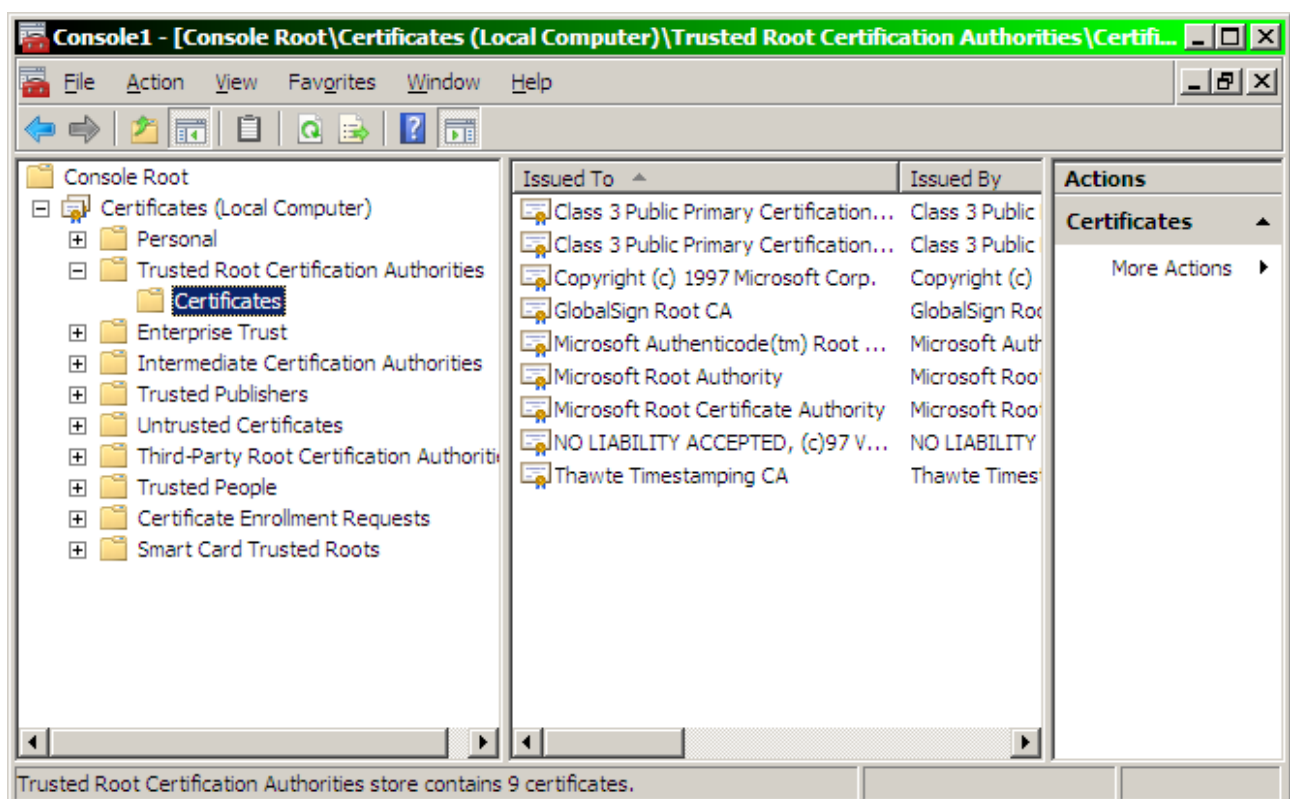
Select local computer, then click to the finish button:



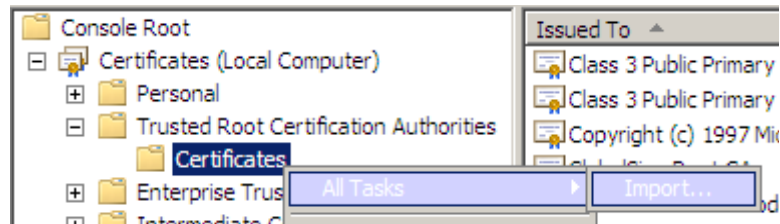
then click to the ok button



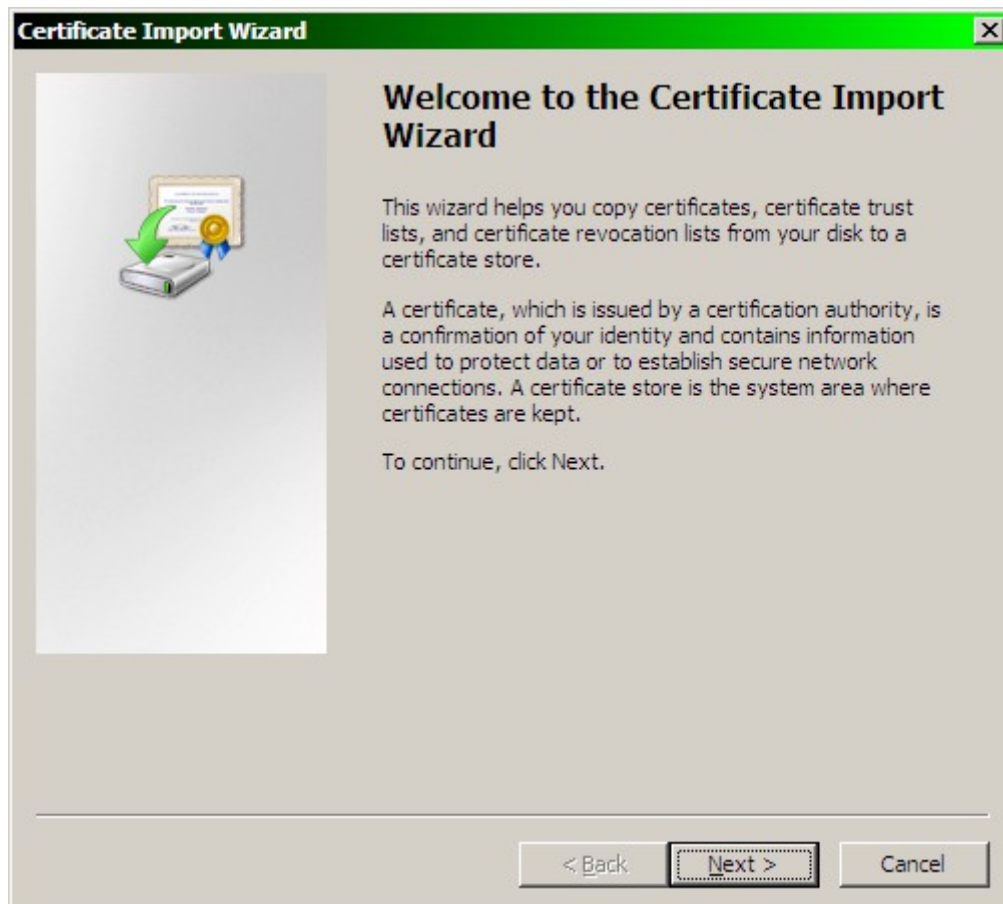
naviget in the tree to the certificates \ trusted root certificate authorities \ certificates:



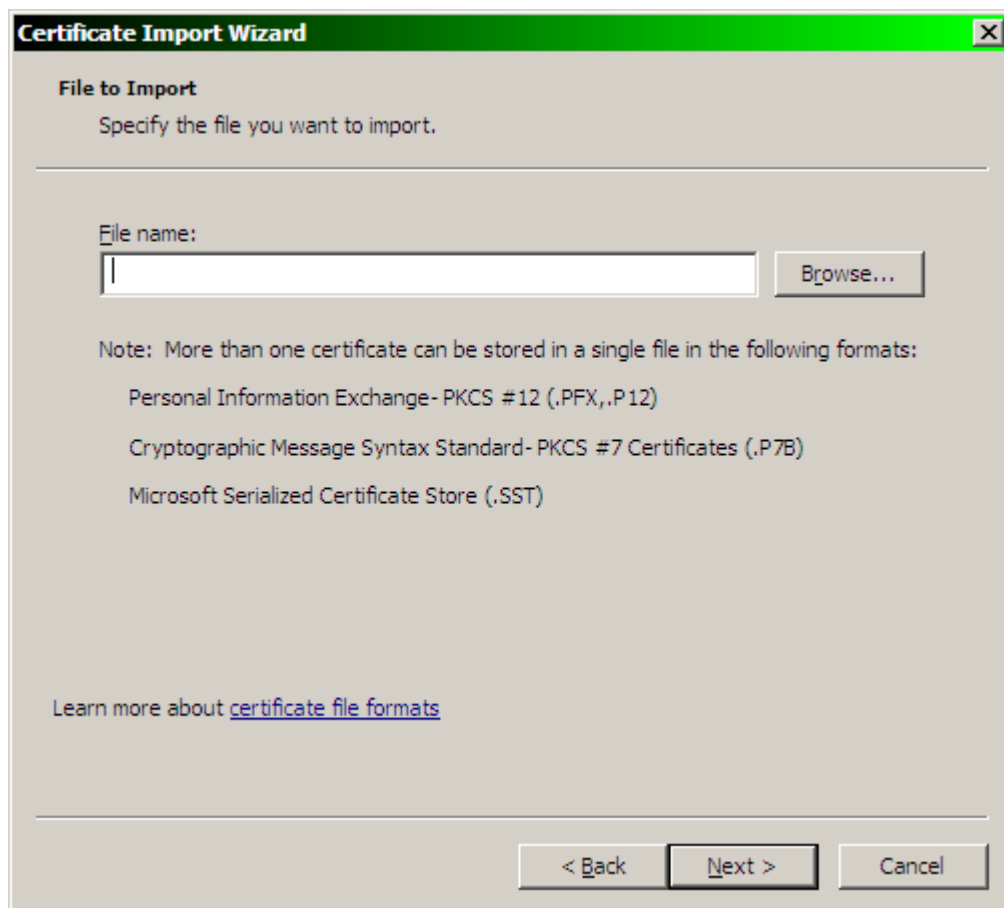
right click on it and from the popup menu select All tasks \ import



on the welcome screen of the import wizard click to next button:



click to the browse button:



Certificate Import Wizard [X]

File to Import
Specify the file you want to import.

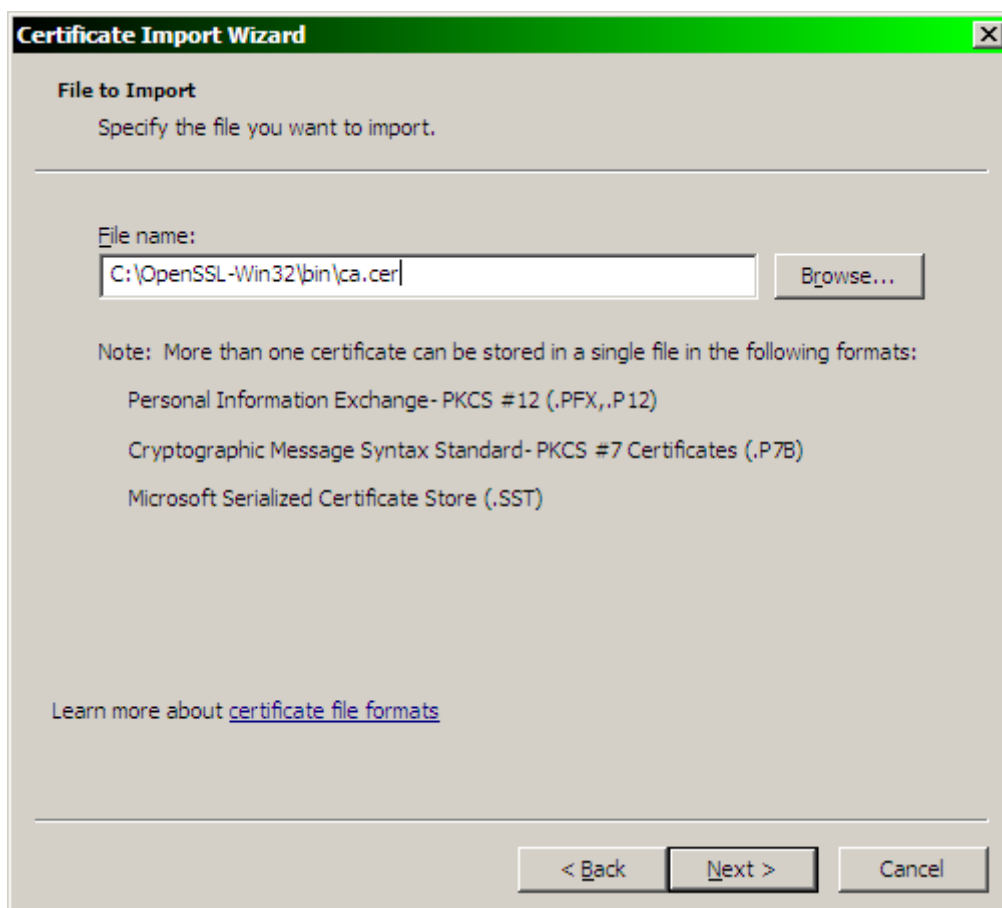
File name:

Note: More than one certificate can be stored in a single file in the following formats:

- Personal Information Exchange- PKCS #12 (.PFX,.P12)
- Cryptographic Message Syntax Standard- PKCS #7 Certificates (.P7B)
- Microsoft Serialized Certificate Store (.SST)

Learn more about [certificate file formats](#)

and select the C:\OpenSSL-Win32\bin\ca.cer file, then click to the next button



Certificate Import Wizard [X]

File to Import
Specify the file you want to import.

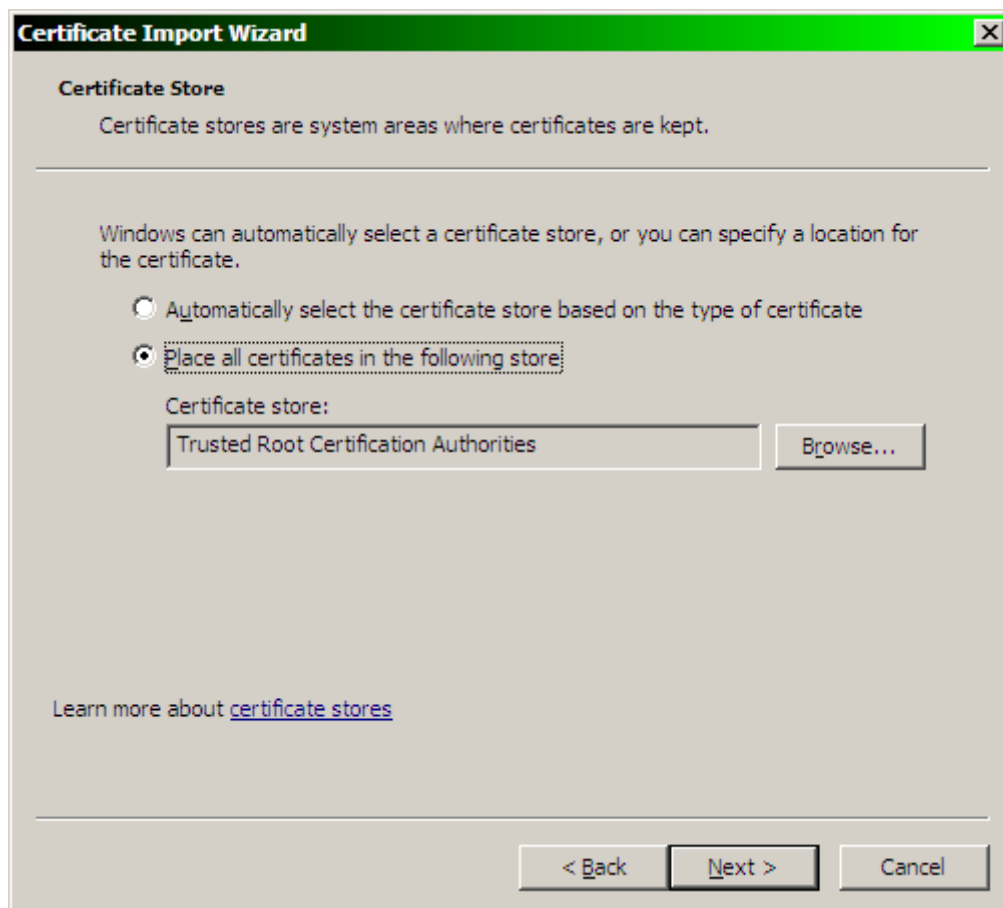
File name:

Note: More than one certificate can be stored in a single file in the following formats:

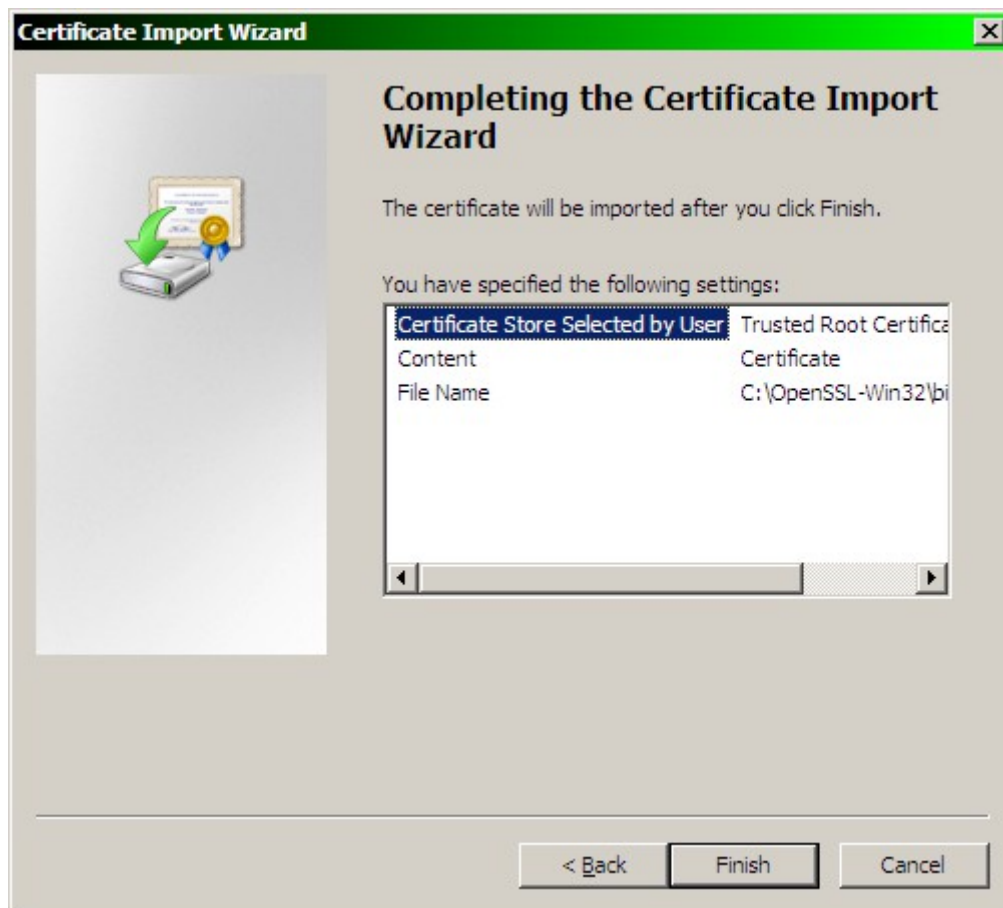
- Personal Information Exchange- PKCS #12 (.PFX,.P12)
- Cryptographic Message Syntax Standard- PKCS #7 Certificates (.P7B)
- Microsoft Serialized Certificate Store (.SST)

Learn more about [certificate file formats](#)

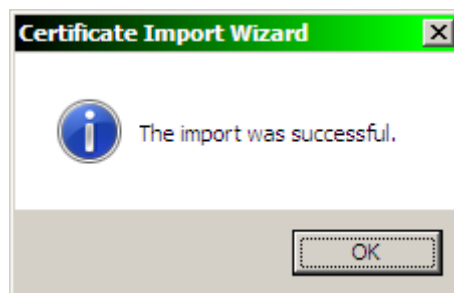
in the other window click to the next button again



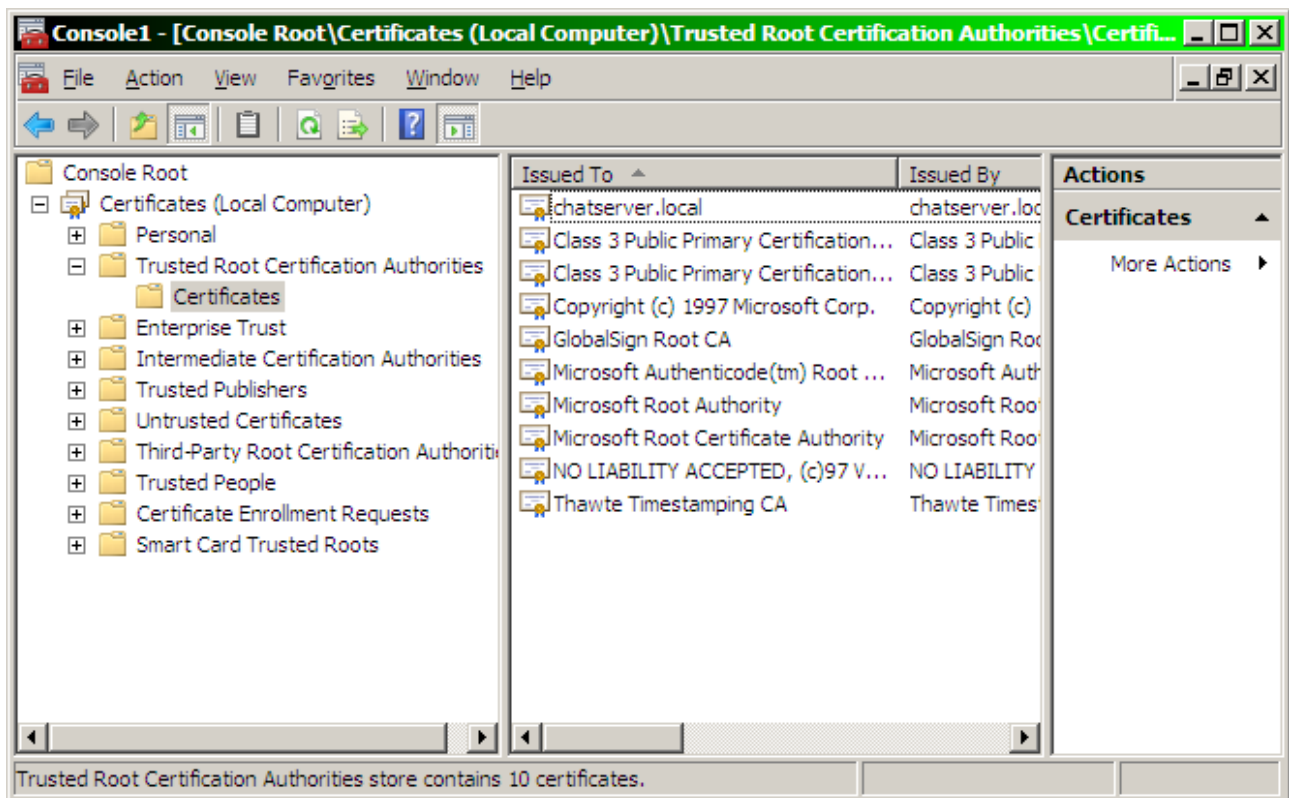
Then click to the finis button



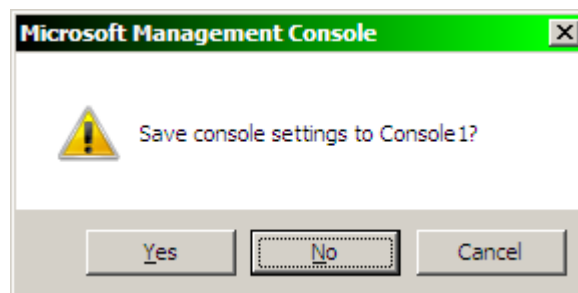
the computer imports the certificate click to the OK button:



The root certificate will be added:



close the mmc console:



Create a server certificate

now our root certificate is created and accepted by the computer, we have to create the two certificates for the two tunnels. For this we must create a request, it can be done by openssl as well, but in that case you should edit the openssl.cnf file, to use the webserver template, so I used the IIS, to generate a request, and only signed that with the openssl.

To do this start the IIS management console Start menu \ control panel \ Administrative tools

Sign the certificate request:

```
openssl ca -policy policy_anything -config openssl.cfg -cert
ca.cer -in request1.txt -keyfile ca.key -days 360 -out iis1.cer
```

type the password of your ca.key file

Open a command prompt, and convert the cer and pfx file to the format required by stunnel:

```
cd \OpenSSL-Win32\bin
```

```
openssl x509 -inform der -in iis1.cer -out iis1.pem
openssl pkcs12 -in iis1.pfx -out iis1.key -nodes -nocerts
Enter Import Password:
```

```
copy iis1.pem "c:\Program Files\stunnel"
copy iis1.key "c:\Program Files\stunnel"
```

Start the Stunnel to listen on port 6789 and forward what it get to 192.168.168.101:1234

After it we should create two config files for the Stunnel, the first on the machine where the chat server runs save it to the (c:\Program Files\stunnel).

stunnel.conf:

```
fips = no
cert = iis1.pem
key = iis1.key
[mylistener]
accept  = 6789
connect = 192.168.168.101:1234
```

Then we must start the stunnel:

```
stunnel.exe
```

then check, if it really listens on port 6789

```
netstat /na
```

search for the next line:

```
TCP  0.0.0.0:6789      0.0.0.0:0      LISTENING
```

Also the two Stunnel icon must be seen next to the clock on the notification area.

We should create another conf file for the other machine (because our sample application is quite simple, we should allow all chipers, including the anonymous ones):

stunnel.conf:

```
ciphers = ALL:eNULL
fips = no
[masik]
client = yes
accept = 1234
connect = 127.0.0.1:9999
```

Start the stunnel:

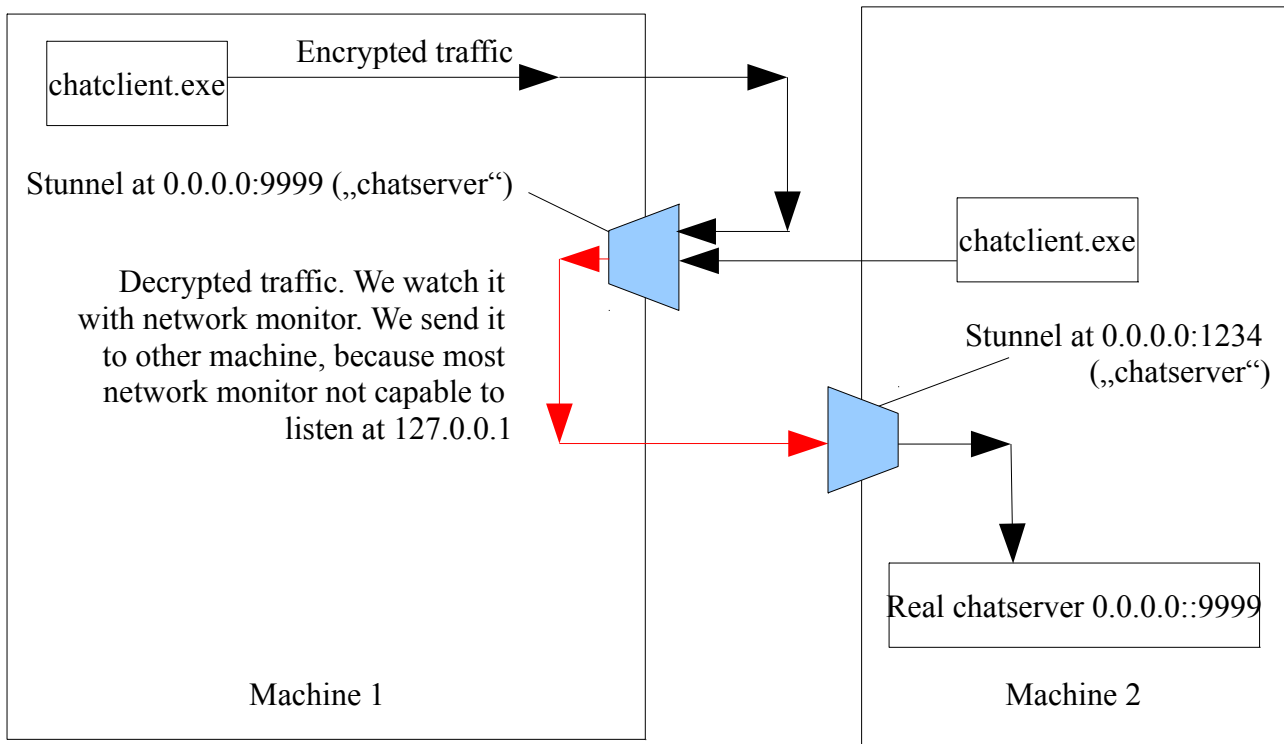
```
stunnel.exe
```

then check, if it really listens on port 1234

```
netstat /na
```

search for the next line:

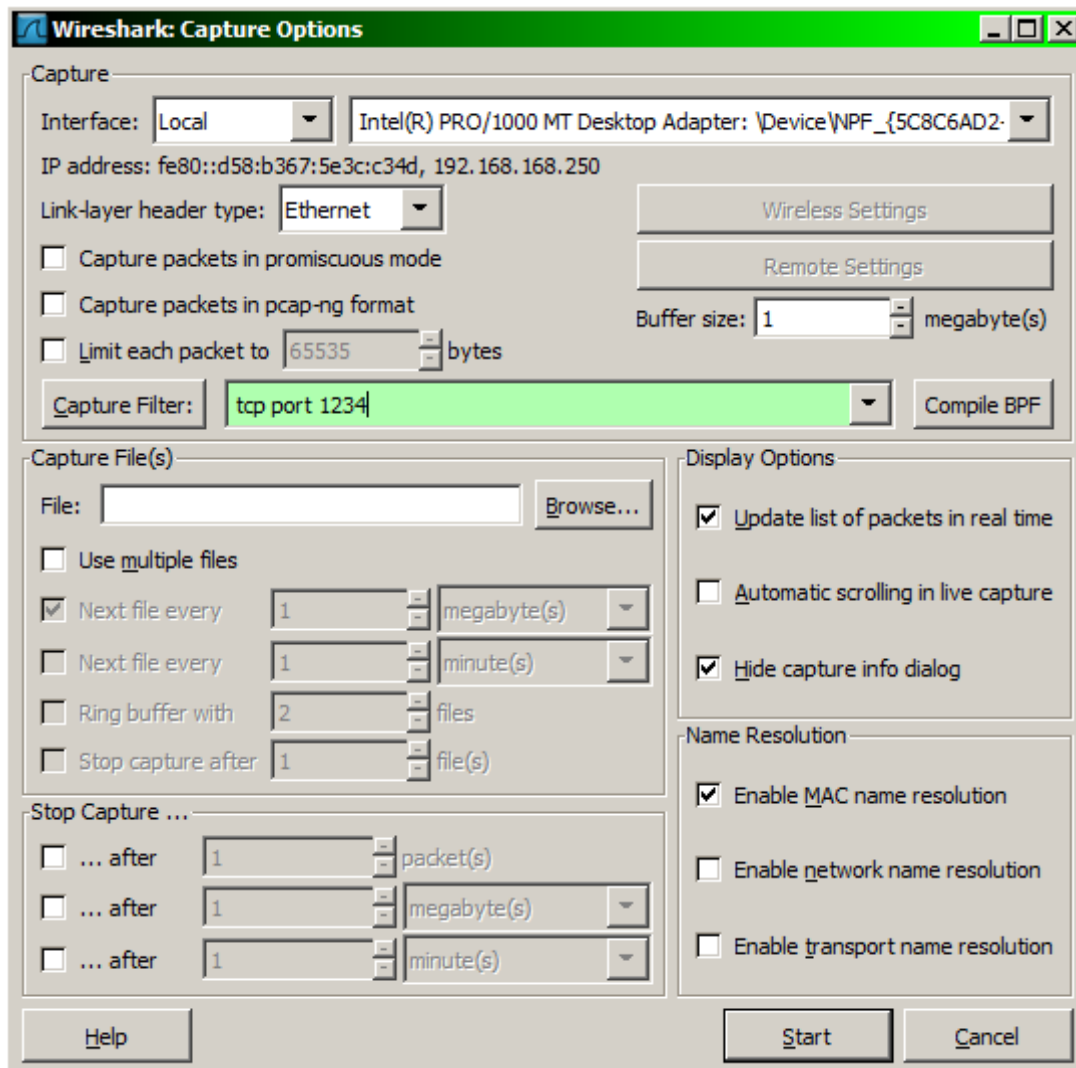
```
TCP 0.0.0.0:1234 0.0.0.0:0 LISTENING
```



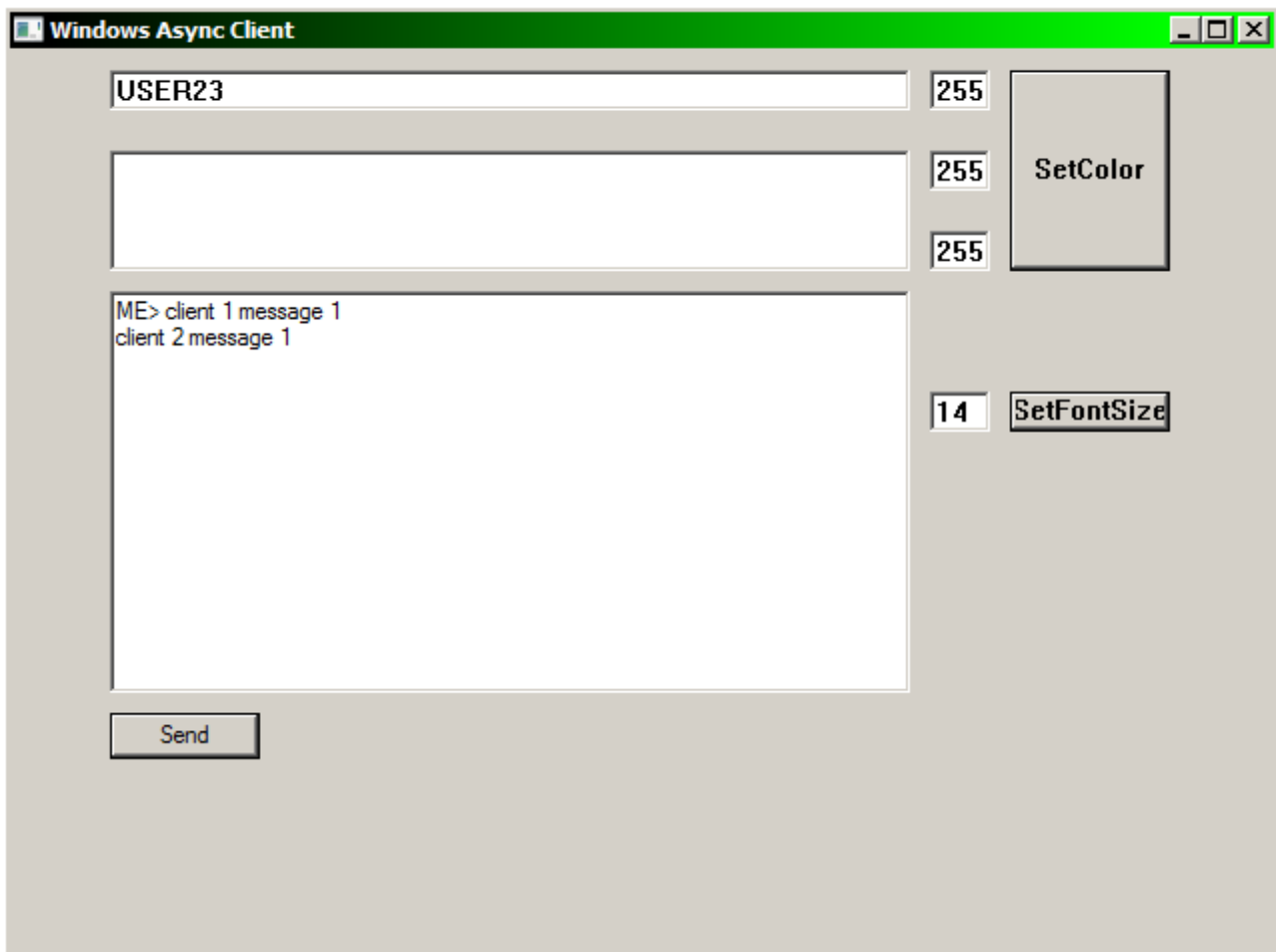
Start the chatserver at machine 1:



now start the wireshark, to capture the traffic going to or coming from the tcp port 1234:



Start the client application at two machines, and send some text from one to the other.



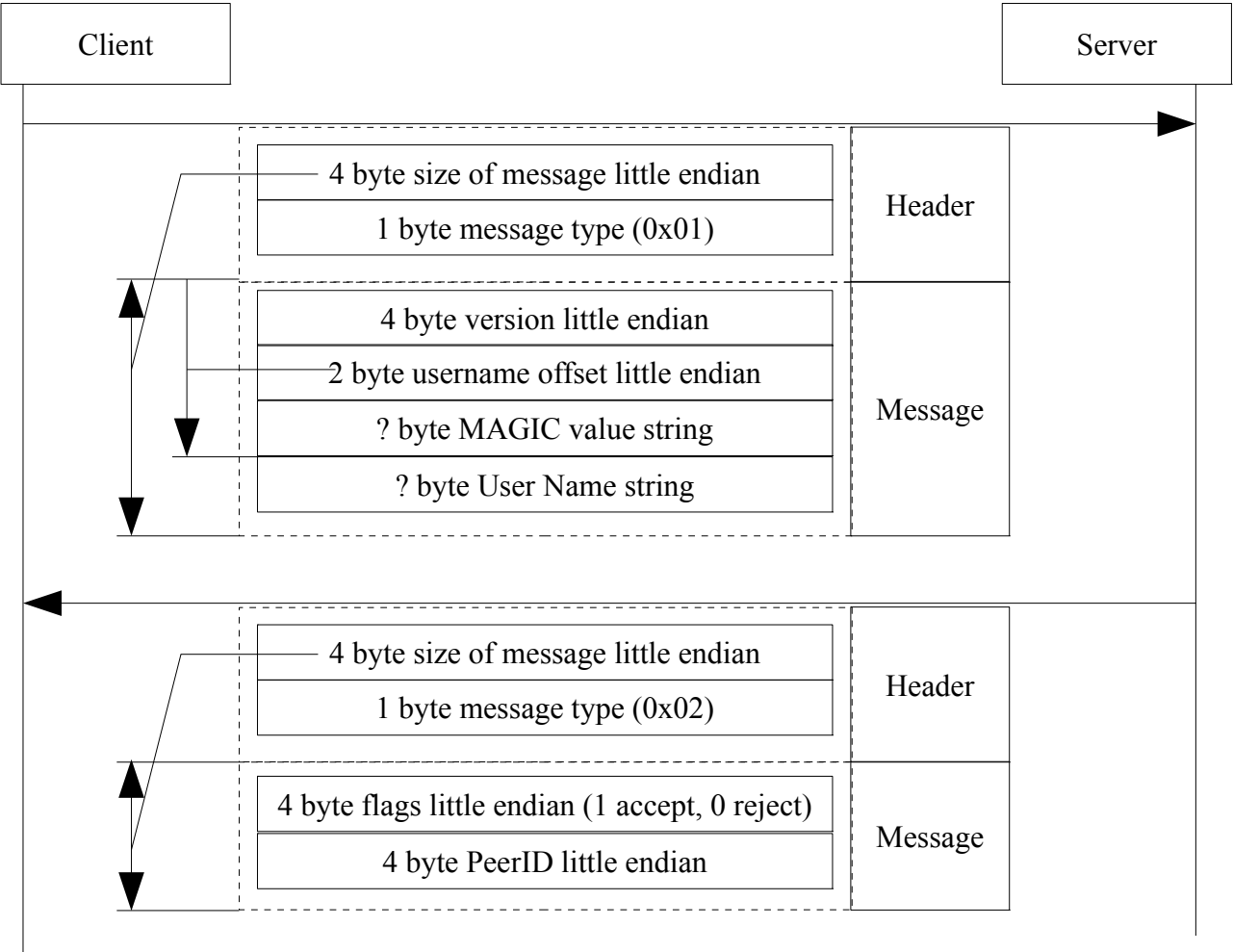
The stop the capture, and close the chat clients and server. We will capture the unencrypted data.

But as we can see the Wireshark does not understand our protocol. In this way it is quite difficult, to analyze what is going on. So our first task (it is not mandatory, just makes our later job easier). To use the Wireshark we should write a custom dissector. It can be written in C or in LUA languages. Now we will do it in LUA, because that is a bit easier than the C.

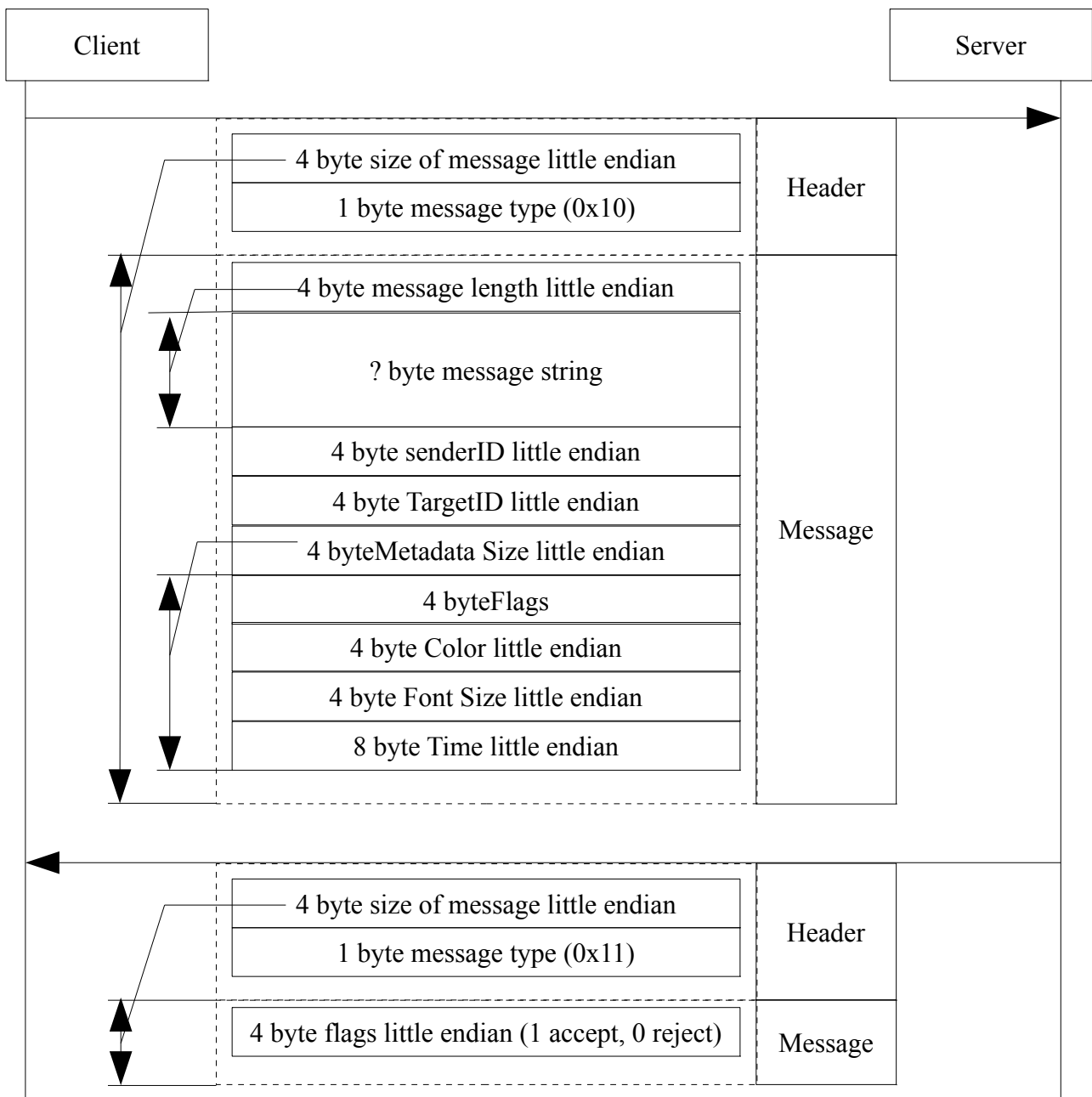
Intel(R) PRO/1000 MT Desktop Adapter (tcp port 1234) [Wireshark 1.6.2 (SVN Rev 38931 from /trunk-1.6)]							
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help							
Filter: Expression... Clear Apply							
No.	Time	Source	Destination	Protocol	Length	Info	
8	15.597890	192.168.168.101	192.168.168.250	TCP	66	1234 > 49218	[SYN, ACK]
9	15.597995	192.168.168.250	192.168.168.101	TCP	54	49218 > 1234	[ACK] Seq
10	15.601404	192.168.168.250	192.168.168.101	TCP	75	49218 > 1234	[PSH, ACK]
11	15.699086	192.168.168.101	192.168.168.250	TCP	67	1234 > 49217	[PSH, ACK]
12	15.699106	192.168.168.101	192.168.168.250	TCP	67	1234 > 49218	[PSH, ACK]
13	15.897523	192.168.168.250	192.168.168.101	TCP	54	49218 > 1234	[ACK] Seq
14	15.897938	192.168.168.250	192.168.168.101	TCP	54	49217 > 1234	[ACK] Seq
15	36.065328	192.168.168.250	192.168.168.101	TCP	113	49217 > 1234	[PSH, ACK]
16	36.074869	192.168.168.101	192.168.168.250	TCP	113	1234 > 49218	[PSH, ACK]
17	36.197120	192.168.168.250	192.168.168.101	TCP	54	49218 > 1234	[ACK] Seq
18	36.264329	192.168.168.101	192.168.168.250	TCP	60	1234 > 49217	[ACK] Seq
19	47.091201	192.168.168.250	192.168.168.101	TCP	113	49218 > 1234	[PSH, ACK]
20	47.099479	192.168.168.101	192.168.168.250	TCP	113	1234 > 49217	[PSH, ACK]
<div> <div>Frame 11: 67 bytes on wire (536 bits), 67 bytes captured (536 bits)</div> <div> <div>Ethernet II, Src: InproCom_22:22:22 (00:08:22:22:22:22), Dst: 00:08:44:44:44:44 (00:08:44:44:44:44)</div> <div>Internet Protocol Version 4, Src: 192.168.168.101 (192.168.168.101), Dst: 192.168.168.250 (192.168.168.250)</div> <div>Transmission Control Protocol, Src Port: 1234 (1234), Dst Port: 49217 (49217), Seq: 1, Ack: 49218, Window: 65535, Len: 67, Ecn: 0, Flags: [PSH, ACK]</div> <div>Data (13 bytes)</div> <div>Data: 1000000001010000000a004348 [Length: 13]</div> </div> </div>							
0000	00 08 44 44 44 44 00 08	22 22 22 22 08 00 45 00	...DDDD.. """"..E.				
0010	00 35 04 c6 40 00 80 06	23 4c c0 a8 a8 65 c0 a8	.5..@... #L...e..				
0020	a8 fa 04 d2 c0 41 3d f1	f3 95 f5 dc 2a 04 50 18A=...*.P.				
0030	01 00 6c 44 00 00 10 00	00 00 01 01 00 00 00 0a	...lD....				
0040	00 43 48		.CH				

Communication flow between the client and server

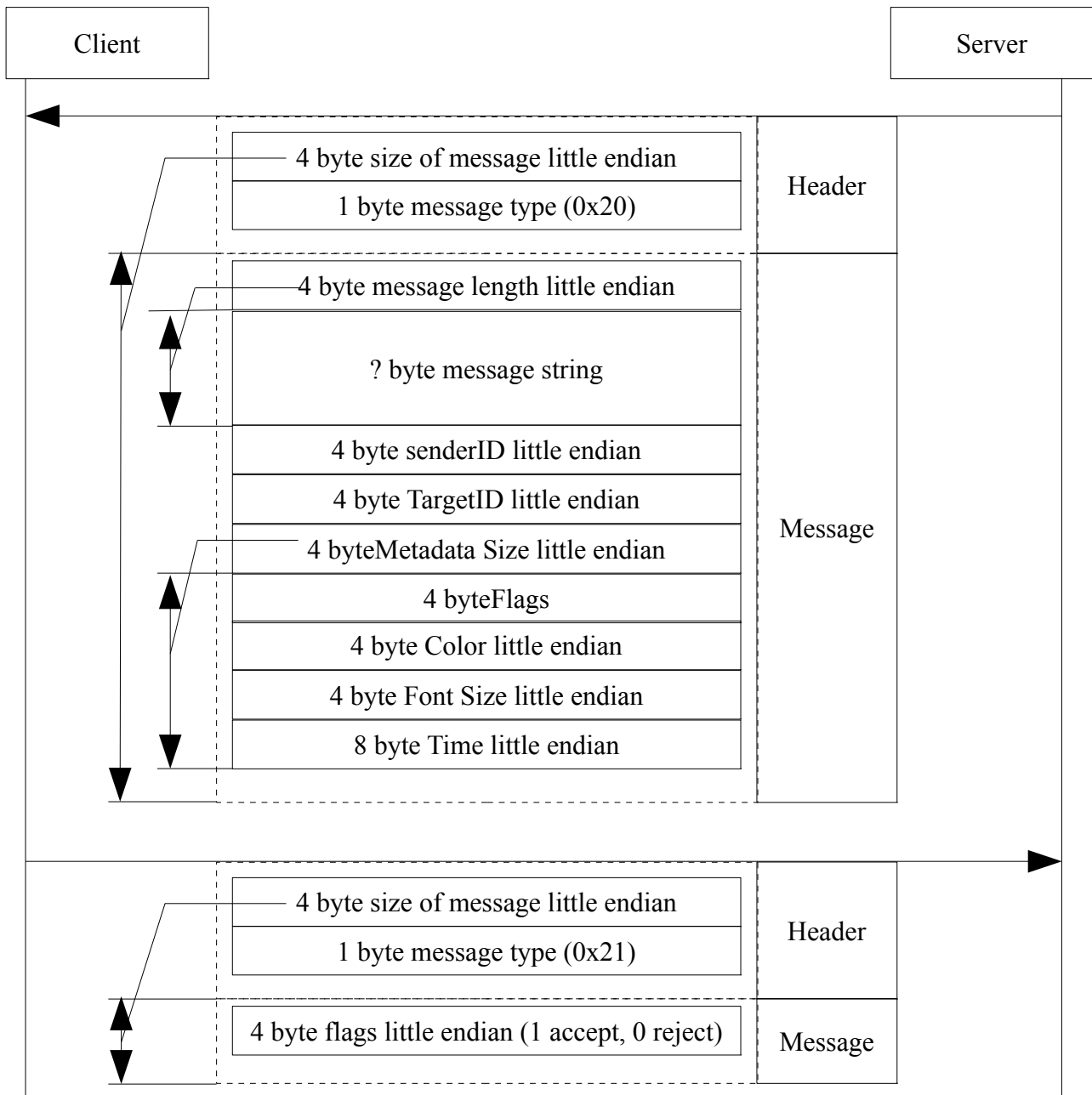
Process of connection



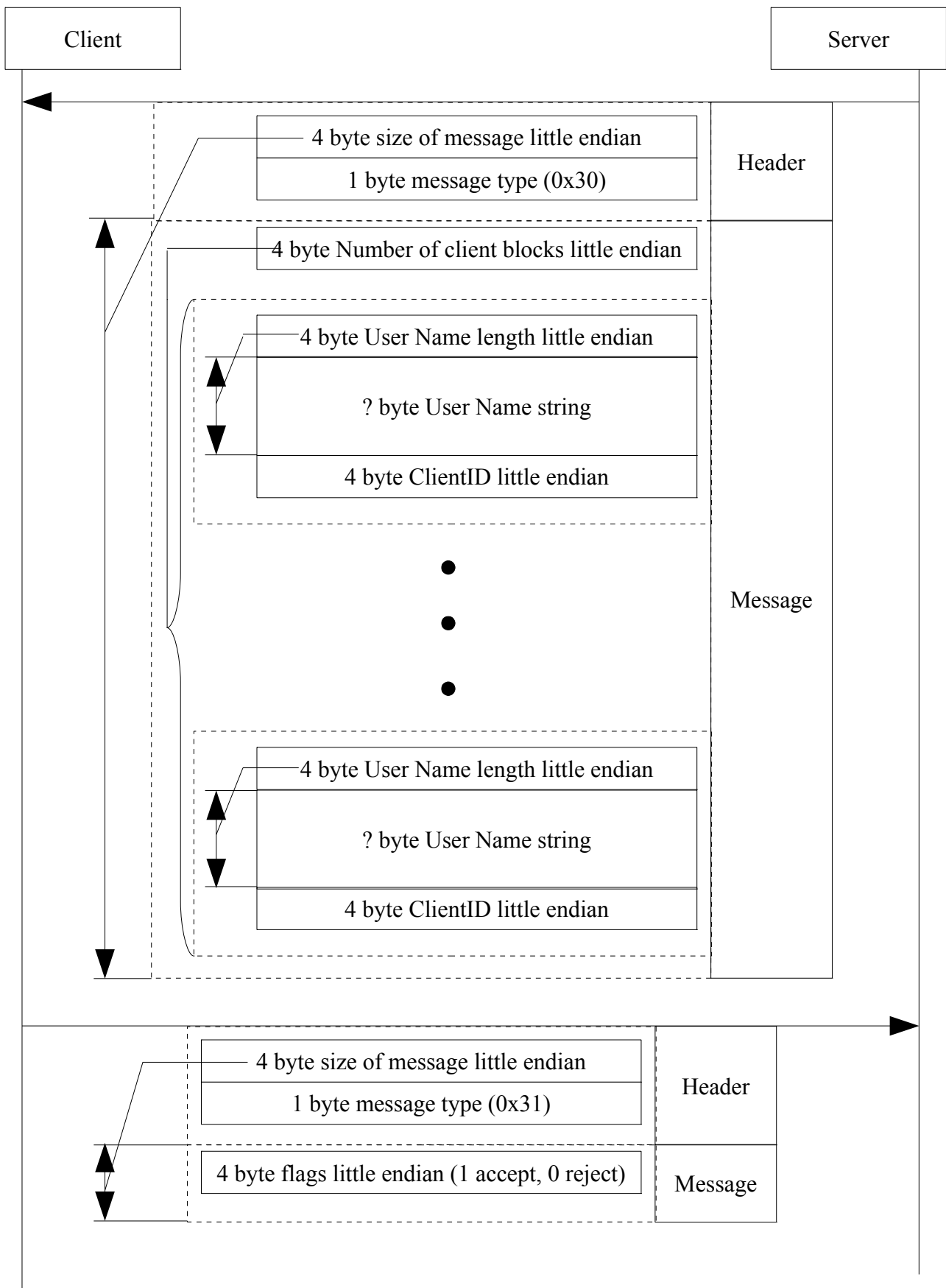
Client message sending



Server message delivery



Server Send Satus



Creation of LUA script

Protocol creation

To create a dissector first we must register a new protocol to the wireshark. To do it we create a Protocol object. It requires two input parameters, a protocol name, and a description. Also we can define the fields of the protocol for filtering and other purposes. I did not need it so I left the field list empty. It is done by the next two lines, it will be the first two lines of our dissector:

```
p_myproto = Proto ("myproto", "MY Potocol")
p_myproto.fields = {}
```

init function

If we create a protocol we must define two functions the init function, and the dissector function. The init function is called once when the protocol first called, and it has no input parameters. Now I do not want to do anything at the initialization state so it will be an empty function, and it remains that:

```
function p_myproto.init()
end
```

Dissector Function

The dissector function has three input variables, the packet itself, now I will call it as buf. The packet metadata now I will call it as pkt. And finally the root shows, where are we in the protocol tree.

```
function p_myproto.dissector (buf, pkt, root)
end
```

Register Protocol

After created we should register the protocol in wireshark. We will register it to the TCP port 8000 because I started to use that one to see the decrypted traffic. Before we register our protocol to that port we save the original protocol connected to that TCP port (most probably there is nothing, and we do not want to use it, but who knows what need later, better to have it). It can be done with the next code segment:

```
local tcp_dissector_table = DissectorTable.get("tcp.port")
original_dissector = tcp_dissector_table:get_dissector(1234)
tcp_dissector_table:add(1234, p_myproto)
```

Whole Code at this milestone

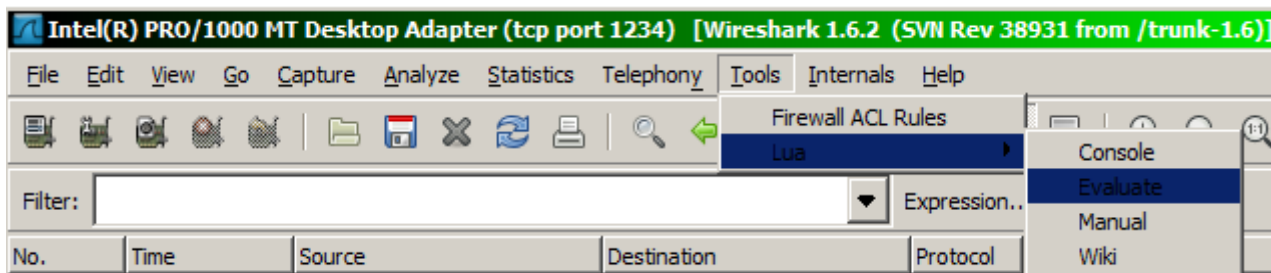
the code until this point looks like as:

```

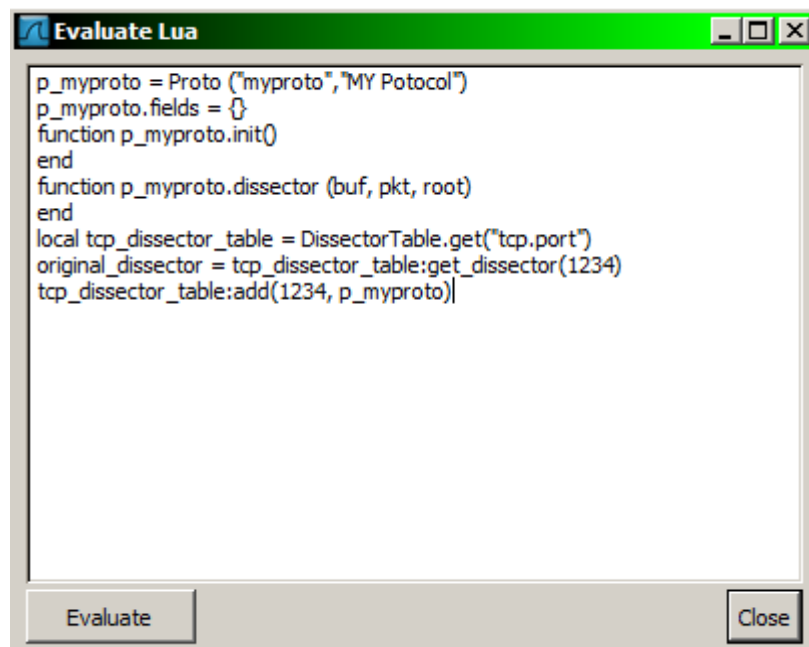
p_myproto = Proto ("myproto","MY Potocol")
p_myproto.fields = {}
function p_myproto.init()
end
function p_myproto.dissector (buf, pkt, root)
end
local tcp_dissector_table = DissectorTable.get("tcp.port")
original_dissector = tcp_dissector_table:get_dissector(1234)
tcp_dissector_table:add(1234, p_myproto)

```

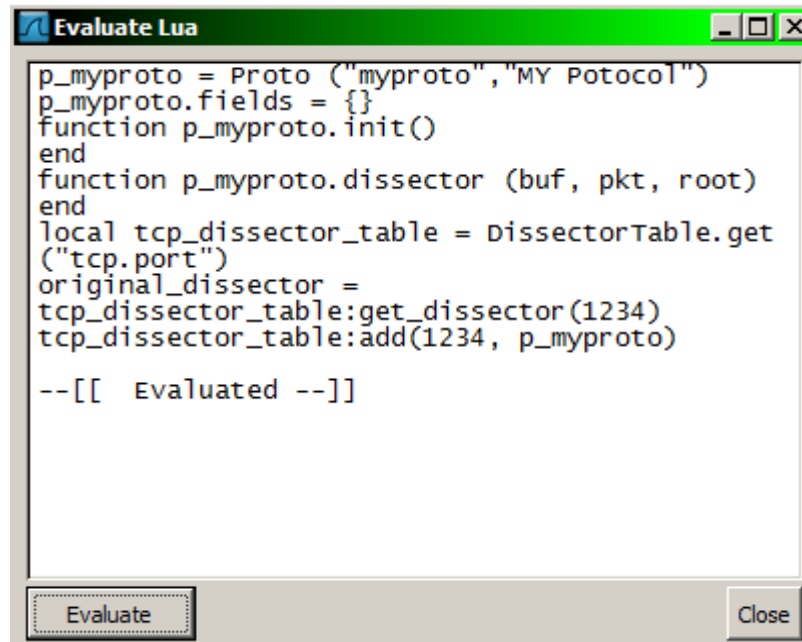
Open the LUA console, by clicking the Tools \ LUA \ Evaluate



Copy the code, then click to the evaluate button



The evaluate LUA window change:



This script does not do anything yet, so we can start to develop the dissector part.

set protocol name

Set the protocol name to our protocols name in the Protocol column

```
pkt.cols.protocol = p_myproto.name
```

Write the dissector

First we initialize some variable necessary to us. We define an actualpos variable, what shows us from which position we want to decode the packet. It need because more then one message can be sent in one packet, if the TCP stack or openssl compacts the messages. At first we start from 0 position so the initial value of it should be 0. Then we create a branch represents our protocol, because we do not want to pour everything into the root. It is done by root:add command. Now we do not give it description and the buf(0) without length means all data in the packet.

```
actualpos=0
outertree = root:add(p_myproto, buf(0))
```

now we can start to analyze the packet. The header length is 5 bytes so we start a cycle until there is more than 5 byte unprocessed data in the packet.

```
while (buf:len()-actualpos) > 5 do
end
```

cycle to analyze the actual part of the packet

- we define a length variable, and set it to 0, we will store the length of the message in it.
- Define a cmdstr variable, it will contain the type of the message. We give it a default value " = UNKNOWN"
- Define a cmd variable, it contains the message type in numerical format. The message type can be read from the fourth byte and it has one bytes length the :uint() convert it to number.
- the length can be read from the 0 position and it is a 4 byte little endian number, the :le_uint() converts it to number

```
length=0
cmdstr = " = UNKNOWN"
cmd=0
cmd = buf(actualpos+4,1):uint()
length = buf(actualpos+0,4):le_uint()
```

Now we should check if there is every data in the buffer (the TCP can break the data to more packets, and it may be only a part of it). If the length in the packet is more than the remaining part of the packet then we should read the next packet as well. To do it we set the pkt. desegment_offset to the actual position because we want to continue the dissection process from that position. And we tell in the pkt.desegment_len variable how many bytes missing. Then we return, and pass back the pkt structure, to sign to the wireshark we want to read more data.

```
if length > buf:len()-actualpos-5 then
    pkt.desegment_offset = actualpos
    pkt.desegment_len = length - (buf:len()-actualpos-5)
    return pkt
end
```

We create a new branch within our tree to this message. It is done by the outertree:add command, the outertree variable stores our subtree created not long ago, it will contain our protocol, and it represents the bytes from actual position till the end of the message we must add 5 because the message size does not contains the header size what 5 bytes. Then within this branch we create a leaf by the subtree:add command. The first four bytes represent the length of the message and as description we give it "Length: " concatenated with the length we read (the two points are concatenation)

```
subtree = outertree:add(p_myproto, buf(actualpos,length+5))
subtree:add(buf(actualpos+0,4)," Length: " .. length)
```

unfortunately there is no case in LUA so we must examine the value of cmd by lot of ifs to print out the internal message structure based on the message type.

- We check if the value of cmd is 0x01 what means Client Connect Request.
- We set the cmdstr to " Client Connect Request"
- add a new leaf 1 bytes from the position 4 represents the command, we highlight it, and type as description the byte code of the message, and the meaning of it. As we already know this type of message contains the following fields:
 - Version 4 bytes number,
 - offset of the username 2 bytes little endian data,
 - magic value a string,
 - username string starts from the offset position was given in the offset

```
if cmd == 0x01 then
```

```

        cmdstr = " Client Connect Request"
        subtree:add(buf(actualpos+4,1)," Command : " .. cmd ..
cmdstr)
        subtree:add(buf(actualpos+5,4)," version : " ..
buf(actualpos+5,4):le_uint())
        offset = buf(actualpos+9,2):le_uint()
        subtree:add(buf(actualpos+9,2)," username offset : " ..
offset)
        subtree:add(buf(actualpos+11,offset-6)," Magic : " ..
buf(actualpos+11,offset-6):string())
        subtree:add(buf(actualpos+5+offset,length-offset)," Username
: " .. buf(actualpos+5+offset,length-offset):string())
    end
end

```

Finally we add to the outer subtree the message type to be nicer, and we move the actualpos to the end of this block.

```

subtree:append_text(cmdstr)
actualpos=actualpos+5+length

```

The whole script until now

```

p_myproto = Proto ("myproto","MY Potocol")
p_myproto.fields = {}
function p_myproto.init()
end
function p_myproto.dissector (buf, pkt, root)
    actualpos=0
    outertree = root:add(p_myproto, buf(0))
    while (buf:len()-actualpos) > 5 do
        length=0
        cmdstr = " = UNKNOWN"
        cmd=0
        cmd = buf(actualpos+4,1):uint()
        length = buf(actualpos+0,4):le_uint()
        if length > buf:len()-actualpos-5 then
            pkt.desegment_offset = actualpos
            pkt.desegment_len = length - (buf:len()-actualpos-5)
            return pkt
        end
        subtree = outertree:add(p_myproto, buf(actualpos,length+5))
        subtree:add(buf(actualpos+0,4)," Length: " .. length)
        if cmd == 0x01 then
            cmdstr = " Client Connect Request"
            subtree:add(buf(actualpos+4,1)," Command : " .. cmd ..
cmdstr)
            subtree:add(buf(actualpos+5,4)," version : " ..
buf(actualpos+5,4):le_uint())
            offset = buf(actualpos+9,2):le_uint()
            subtree:add(buf(actualpos+9,2)," username offset : " ..
offset)

```

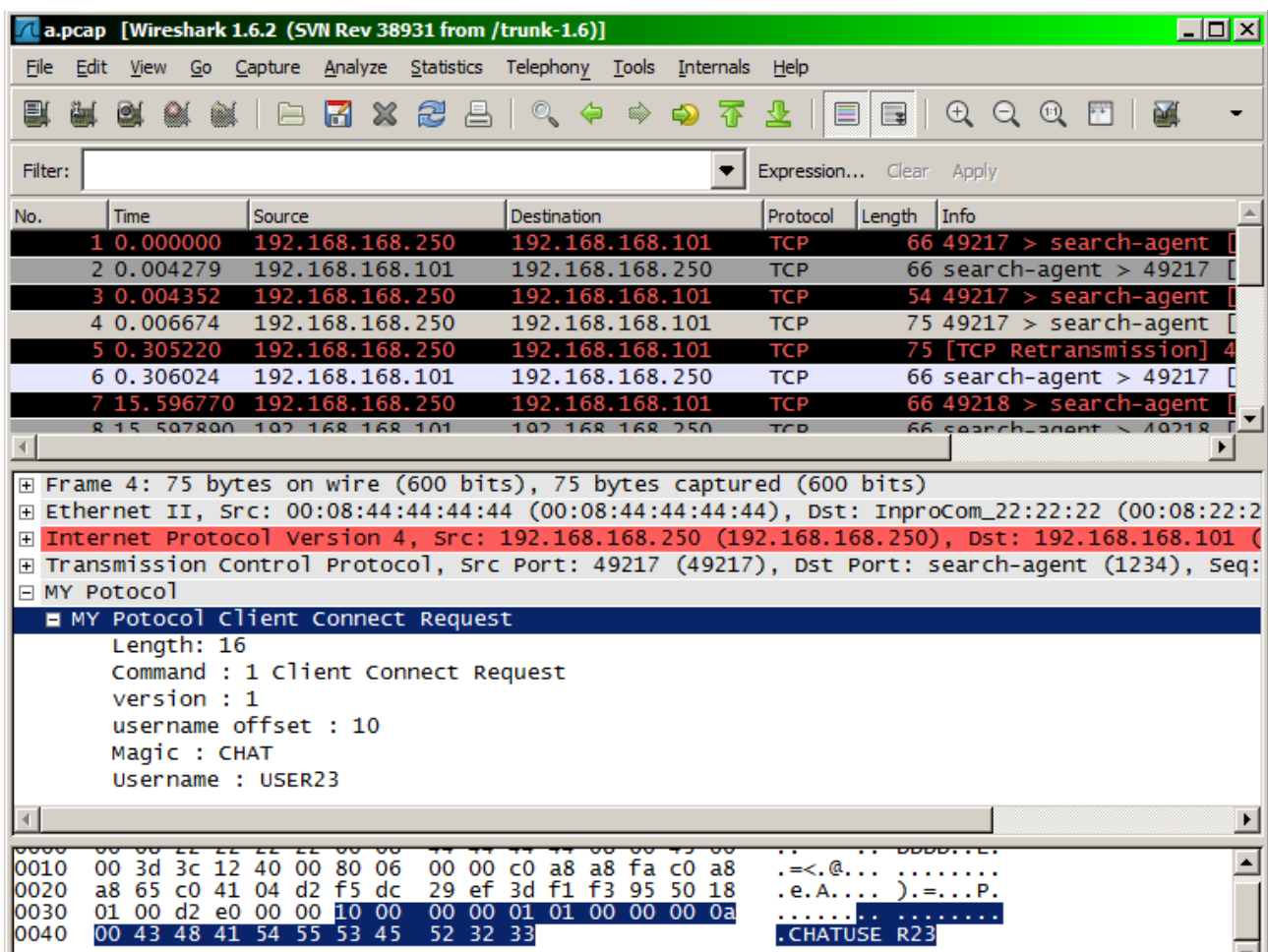


```

        subtree:add(buf(actualpos+11,offset-6)," Magic : " ..
buf(actualpos+11,offset-6):string())
        subtree:add(buf(actualpos+5+offset,length-offset)," Username
: " .. buf(actualpos+5+offset,length-offset):string())
    end
    subtree:append_text(cmdstr)
    actualpos=actualpos+5+length
end
end
local tcp_dissector_table = DissectorTable.get("tcp.port")
original_dissector = tcp_dissector_table:get_dissector(1234)
tcp_dissector_table:add(1234, p_myproto)

```

Now save the previous capture, and restart the wireshark. Then evaluate this LUA script as already described. Then load the previously saved capture.
Now when you check the first packet you can see, instead of the previous TCP data the meaning of the different bytes are nicely visible.



extend the dissector with missing packet types

```

if cmd == 0x02 then
    cmdstr = " Server Connect Response"
    subtree:add(buf(actualpos+4,1)," Command : " .. cmd ..
cmdstr)

```

```

        subtree:add(buf(actualpos+5,4)," flags : " ..
buf(actualpos+5,4):le_uint())
        subtree:add(buf(actualpos+9,4)," PeerID : " ..
buf(actualpos+9,4):le_uint())
    end

    if cmd == 0x10 then
        cmdstr = " Client Send Message"
        subtree:add(buf(actualpos+4,1)," Command : " .. cmd ..
cmdstr)
        messagelength = buf(actualpos+5,4):le_uint()
        subtree:add(buf(actualpos+5,4)," message Length : " ..
messagelength)
        subtree:add(buf(actualpos+9,messagelength)," Message : " ..
buf(actualpos+9,messagelength):string())
        subtree:add(buf(actualpos+9+messagelength,4)," SenderID :
" .. buf(actualpos+9+messagelength,4):le_uint())
        subtree:add(buf(actualpos+13+messagelength,4)," TargetID : "
.. buf(actualpos+13+messagelength,4):le_uint())
        subtree:add(buf(actualpos+17+messagelength,4)," MetadataSize
: " .. buf(actualpos+17+messagelength,4):le_uint())
        subtree:add(buf(actualpos+21+messagelength,4)," Flags : " ..
buf(actualpos+21+messagelength,4):le_uint())
        subtree:add(buf(actualpos+25+messagelength,4)," Color : " ..
buf(actualpos+25+messagelength,4):le_uint())
        subtree:add(buf(actualpos+29+messagelength,4)," FontSize : "
.. buf(actualpos+29+messagelength,4):le_uint())
        subtree:add(buf(actualpos+33+messagelength,8)," Time : " ..
buf(actualpos+33+messagelength,8):le_uint64())

    end

    if cmd == 0x20 then
        cmdstr = " Client Send Message"
        subtree:add(buf(actualpos+4,1)," Command : " .. cmd ..
cmdstr)
        messagelength = buf(actualpos+5,4):le_uint()
        subtree:add(buf(actualpos+5,4)," message Length : " ..
messagelength)
        subtree:add(buf(actualpos+9,messagelength)," Message : " ..
buf(actualpos+9,messagelength):string())
        subtree:add(buf(actualpos+9+messagelength,4)," SenderID :
" .. buf(actualpos+9+messagelength,4):le_uint())
        subtree:add(buf(actualpos+13+messagelength,4)," TargetID : "
.. buf(actualpos+13+messagelength,4):le_uint())
        subtree:add(buf(actualpos+17+messagelength,4)," MetadataSize
: " .. buf(actualpos+17+messagelength,4):le_uint())
        subtree:add(buf(actualpos+21+messagelength,4)," Flags : " ..
buf(actualpos+21+messagelength,4):le_uint())
        subtree:add(buf(actualpos+25+messagelength,4)," Color : " ..
buf(actualpos+25+messagelength,4):le_uint())
        subtree:add(buf(actualpos+29+messagelength,4)," FontSize : "
.. buf(actualpos+29+messagelength,4):le_uint())
        subtree:add(buf(actualpos+33+messagelength,8)," Time : " ..

```

```

buf(actualpos+33+messagelength,8):le_uint64())
end

```

The whole script

```

p_myproto = Proto ("myproto", "MY Potocol")
p_myproto.fields = {}
function p_myproto.init()
end
function p_myproto.dissector (buf, pkt, root)
    actualpos=0
    outertree = root:add(p_myproto, buf(0))
    while (buf:len()-actualpos) > 5 do
        length=0
        cmdstr = " = UNKNOWN"
        cmd=0
        cmd = buf(actualpos+4,1):uint()
        length = buf(actualpos+0,4):le_uint()
        if length > buf:len()-actualpos-5 then
            pkt.desegment_offset = actualpos
            pkt.desegment_len = length - (buf:len()-actualpos-5)
            return pkt
        end
        subtree = outertree:add(p_myproto, buf(actualpos,length+5))
        subtree:add(buf(actualpos+0,4)," Length: " .. length)
        if cmd == 0x01 then
            cmdstr = " Client Connect Request"
            subtree:add(buf(actualpos+4,1)," Command : " .. cmd ..
cmdstr)
            subtree:add(buf(actualpos+5,4)," version : " ..
buf(actualpos+5,4):le_uint())
            offset = buf(actualpos+9,2):le_uint()
            subtree:add(buf(actualpos+9,2)," username offset : " ..
offset)
            subtree:add(buf(actualpos+11,offset-6)," Magic : " ..
buf(actualpos+11,offset-6):string())
            subtree:add(buf(actualpos+5+offset,length-offset)," Username
: " .. buf(actualpos+5+offset,length-offset):string())
            end
            if cmd == 0x02 then
                cmdstr = " Server Connect Response"
                subtree:add(buf(actualpos+4,1)," Command : " .. cmd ..
cmdstr)
                subtree:add(buf(actualpos+5,4)," flags : " ..
buf(actualpos+5,4):le_uint())
                subtree:add(buf(actualpos+9,4)," PeerID : " ..
buf(actualpos+9,4):le_uint())
                end
                if cmd == 0x10 then
                    cmdstr = " Client Send Message"
                    subtree:add(buf(actualpos+4,1)," Command : " .. cmd ..
cmdstr)
                    messagelength = buf(actualpos+5,4):le_uint()

```

```

        subtree:add(buf(actualpos+5,4)," message Length : " ..
messagelength)
        subtree:add(buf(actualpos+9,messagelength)," Message : " ..
buf(actualpos+9,messagelength):string())
        subtree:add(buf(actualpos+9+messagelength,4)," SenderID :
" .. buf(actualpos+9+messagelength,4):le_uint())
        subtree:add(buf(actualpos+13+messagelength,4)," TargetID : "
.. buf(actualpos+13+messagelength,4):le_uint())
        subtree:add(buf(actualpos+17+messagelength,4)," MetadataSize
: " .. buf(actualpos+17+messagelength,4):le_uint())
        subtree:add(buf(actualpos+21+messagelength,4)," Flags : " ..
buf(actualpos+21+messagelength,4):le_uint())
        subtree:add(buf(actualpos+25+messagelength,4)," Color : " ..
buf(actualpos+25+messagelength,4):le_uint())
        subtree:add(buf(actualpos+29+messagelength,4)," FontSize : "
.. buf(actualpos+29+messagelength,4):le_uint())
        subtree:add(buf(actualpos+33+messagelength,8)," Time : " ..
buf(actualpos+33+messagelength,8):le_uint64())
    end
    if cmd == 0x20 then
        cmdstr = " Client Send Message"
        subtree:add(buf(actualpos+4,1)," Command : " .. cmd ..
cmdstr)
        messagelength = buf(actualpos+5,4):le_uint()
        subtree:add(buf(actualpos+5,4)," message Length : " ..
messagelength)
        subtree:add(buf(actualpos+9,messagelength)," Message : " ..
buf(actualpos+9,messagelength):string())
        subtree:add(buf(actualpos+9+messagelength,4)," SenderID :
" .. buf(actualpos+9+messagelength,4):le_uint())
        subtree:add(buf(actualpos+13+messagelength,4)," TargetID : "
.. buf(actualpos+13+messagelength,4):le_uint())
        subtree:add(buf(actualpos+17+messagelength,4)," MetadataSize
: " .. buf(actualpos+17+messagelength,4):le_uint())
        subtree:add(buf(actualpos+21+messagelength,4)," Flags : " ..
buf(actualpos+21+messagelength,4):le_uint())
        subtree:add(buf(actualpos+25+messagelength,4)," Color : " ..
buf(actualpos+25+messagelength,4):le_uint())
        subtree:add(buf(actualpos+29+messagelength,4)," FontSize : "
.. buf(actualpos+29+messagelength,4):le_uint())
        subtree:add(buf(actualpos+33+messagelength,8)," Time : " ..
buf(actualpos+33+messagelength,8):le_uint64())
    end
    subtree:append_text(cmdstr)
    actualpos=actualpos+5+length
end
end
local tcp_dissector_table = DissectorTable.get("tcp.port")
original_dissector = tcp_dissector_table:get_dissector(1234)
tcp_dissector_table:add(1234, p_myproto)

```

fuzzing

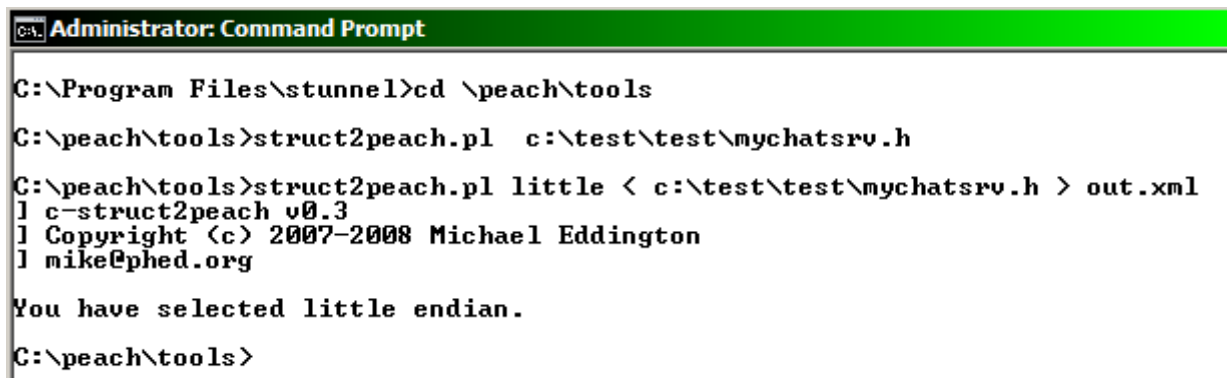
Autogenerate the datamodel by struct2peach

To create the whole datamodel from nothing can be a bit time consuming, but fortunately the peach fuzzer has a small tool at the C:\peach\tools directory called struct2peach.pl, what is able to create the data models from a C++ header file. Of course this have some limitations:

- The struct2peach does not handle the enum type fields.
- The data model logic for example that one filed is the size of a block, or offset of some data block is not described in the header file, so it will not be in the created model obviously, it must be added manually.
- If non standard data types are used the perl code should be modified, to handle them (see a bit later)

We need the following command:

```
cd c:\peach\tools
struct2peach.pl little < c:\test\test\mychatsrv.h > out.xml
```

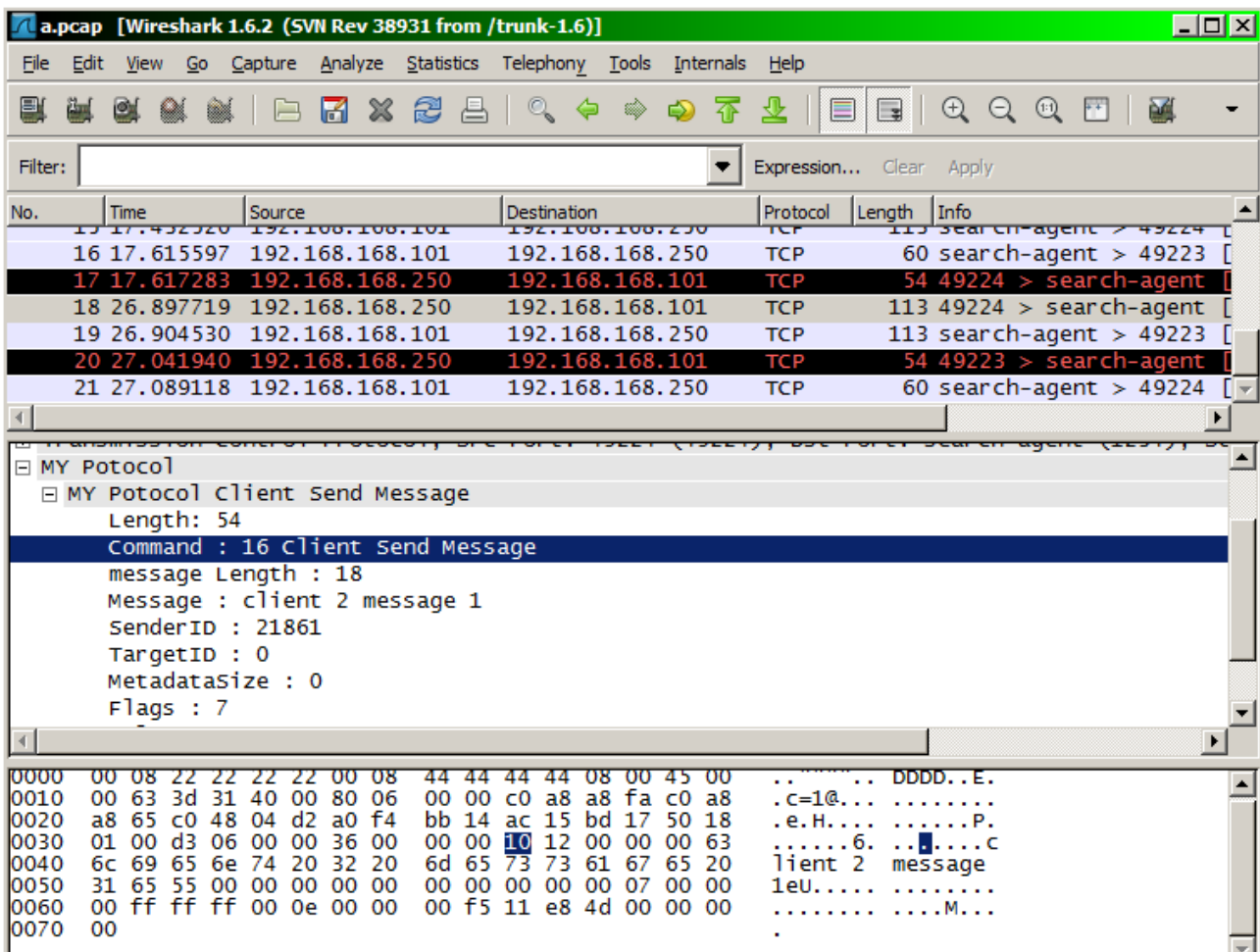


```
C:\Program Files\stunnel>cd \peach\tools
C:\peach\tools>struct2peach.pl c:\test\test\mychatsrv.h
C:\peach\tools>struct2peach.pl little < c:\test\test\mychatsrv.h > out.xml
| c-struct2peach v0.3
| Copyright (c) 2007-2008 Michael Eddington
| mike@phed.org
You have selected little endian.
C:\peach\tools>
```

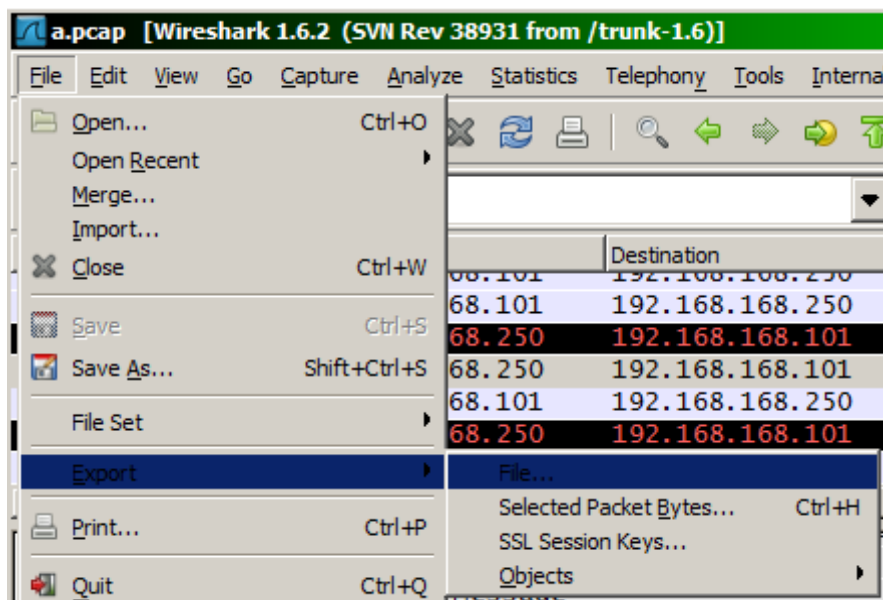
Autogenerate the datamodel by peachshark

Another method what is effective, if we want to fuzz data included in one network packet. In this case we can export the packet from wireshark in xml format, and use the peachshark included in the peach fuzzer, to generate the data model from it. It requires that, the wireshark should have a dissector to the protocoll (but we just created one, so it is fine now). See an example to this as well.

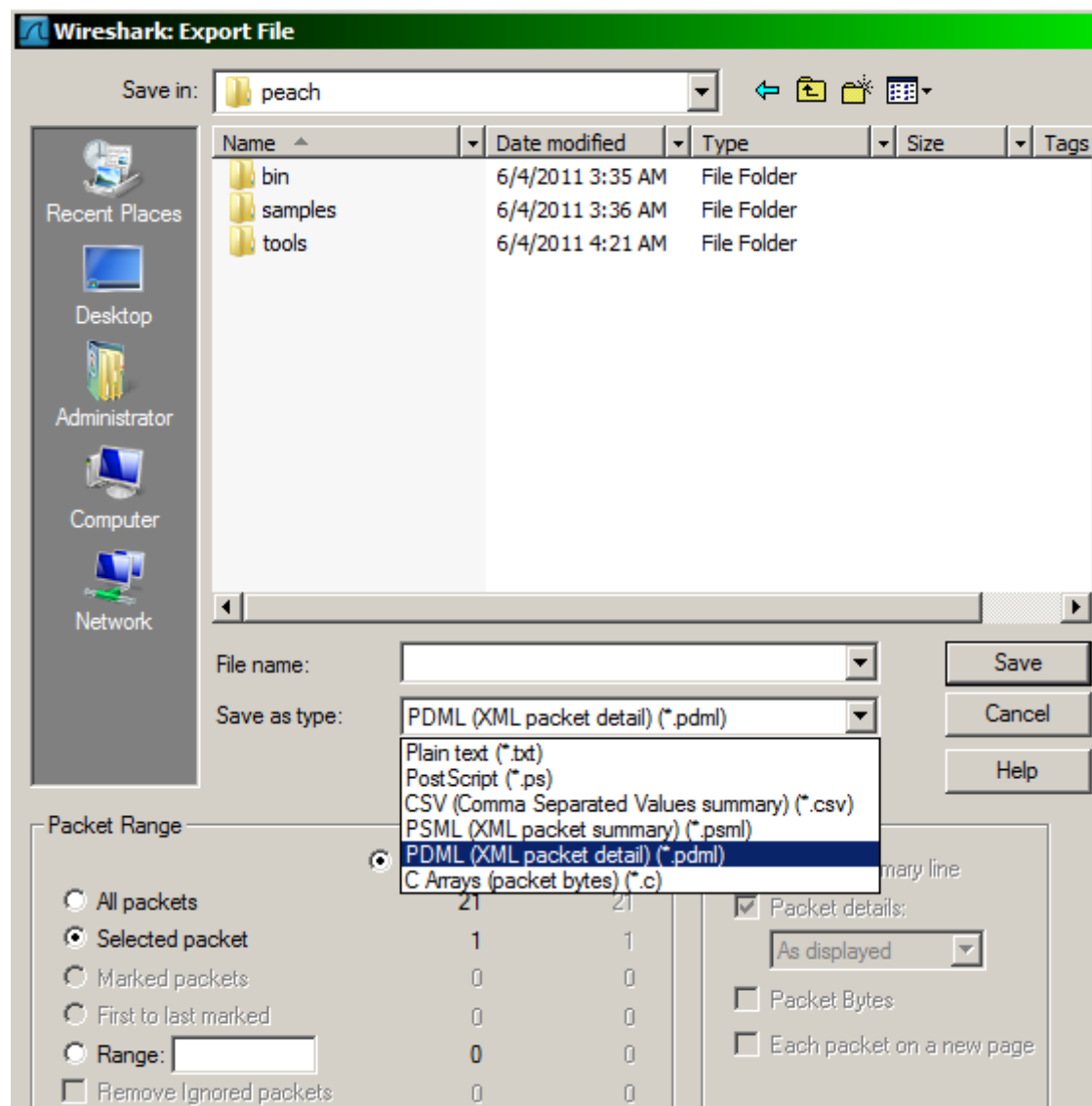
We already captured a traffic when the client sent message. It was the 18th packet, and our dissector compiled shows the details of it.



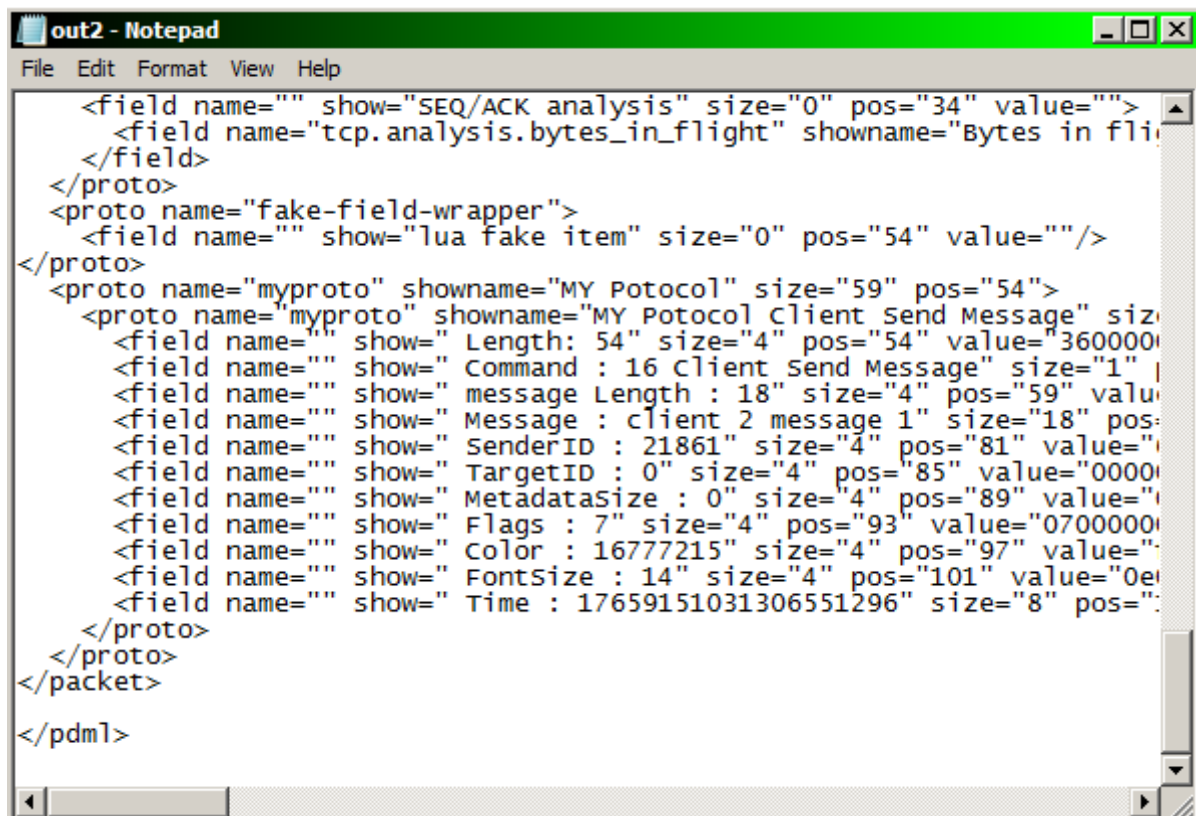
now select the file \ export \ file command



in the next window select the place where you want to save the packet, the format should be PDML XML packet detail, and it should contain only ONE packet, because the peachshark is only capable to decode one packet a time. And select "selected packet" Packet Range.



Open the exported pdml (xml) file, you will see something like this:

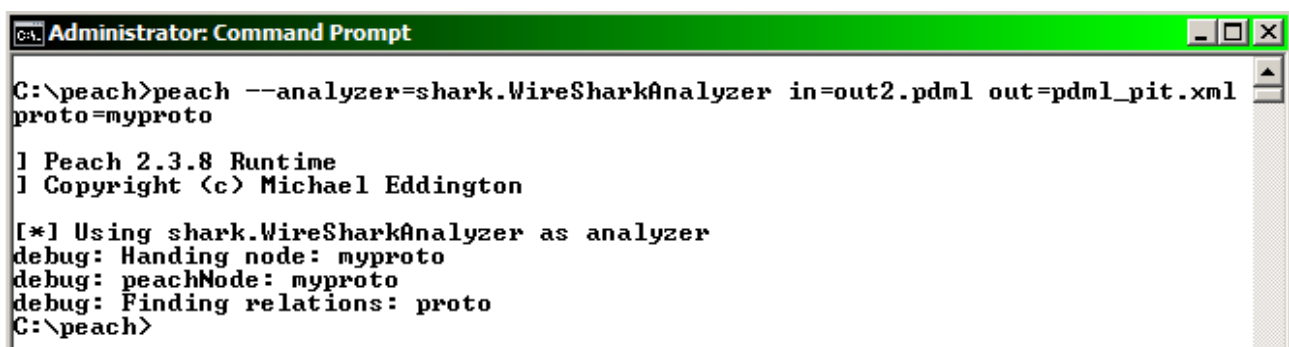


```
<field name="" show="SEQ/ACK analysis" size="0" pos="34" value="">
  <field name="tcp.analysis.bytes_in_flight" showname="Bytes in fli
</field>
</proto>
<proto name="fake-field-wrapper">
  <field name="" show="lua fake item" size="0" pos="54" value=""/>
</proto>
<proto name="myproto" showname="MY Potocol" size="59" pos="54">
  <proto name="myproto" showname="MY Potocol Client Send Message" siz
    <field name="" show=" Length: 54" size="4" pos="54" value="360000
    <field name="" show=" Command : 16 Client Send Message" size="1"
    <field name="" show=" message Length : 18" size="4" pos="59" valu
    <field name="" show=" Message : client 2 message 1" size="18" pos:
    <field name="" show=" SenderID : 21861" size="4" pos="81" value="
    <field name="" show=" TargetID : 0" size="4" pos="85" value="0000
    <field name="" show=" MetadataSize : 0" size="4" pos="89" value="
    <field name="" show=" Flags : 7" size="4" pos="93" value="0700000
    <field name="" show=" Color : 16777215" size="4" pos="97" value="
    <field name="" show=" FontSize : 14" size="4" pos="101" value="0e
    <field name="" show=" Time : 17659151031306551296" size="8" pos="
  </proto>
</proto>
</packet>

</pdm1>
```

from here you can read the name of the protocol we want to create the datamodel from is the "myproto" (we already knew it, because we created the dissector, but we checked it to be sure). Now use the following command, to create the datamodel:

```
peach --analyzer=shark.WireSharkAnalyzer in=out2.pdm1
out=pdm1_pit.xml proto=myproto
```



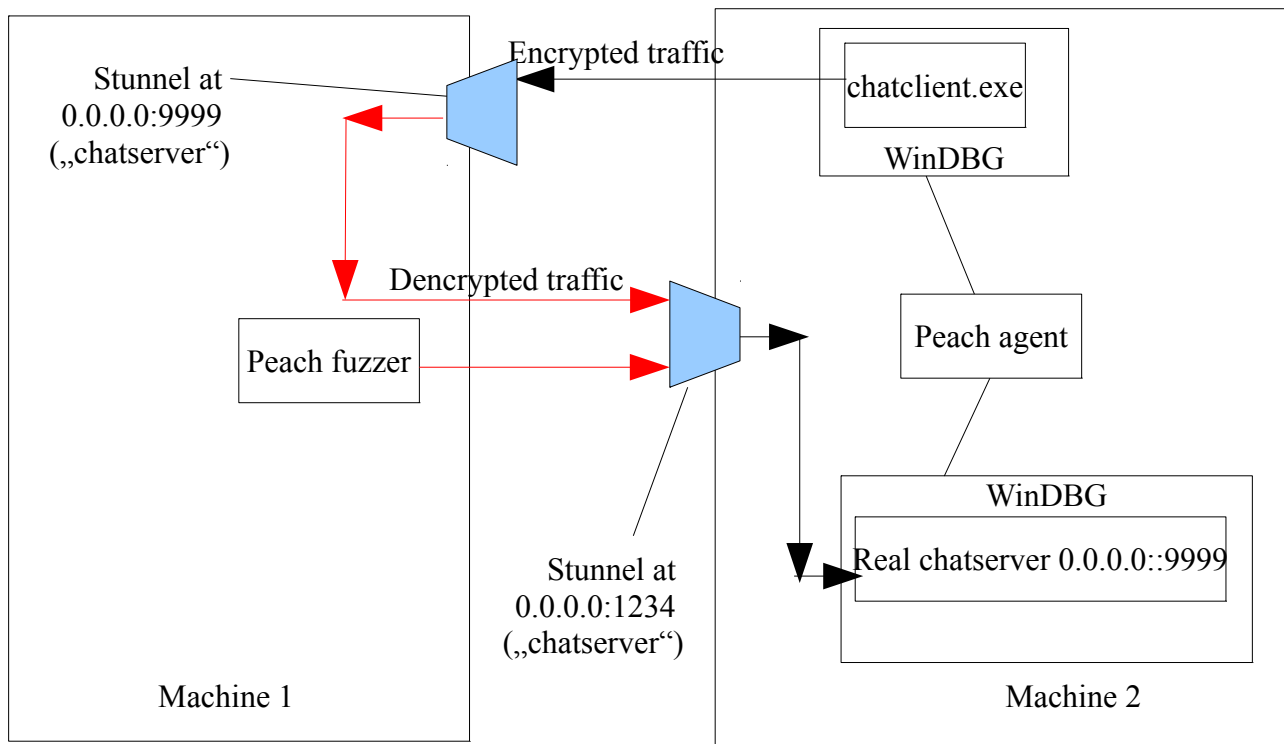
```
C:\peach>peach --analyzer=shark.WireSharkAnalyzer in=out2.pdm1 out=pdm1_pit.xml
proto=myproto

1 Peach 2.3.8 Runtime
1 Copyright (c) Michael Eddington

[*] Using shark.WireSharkAnalyzer as analyzer
debug: Handing node: myproto
debug: peachNode: myproto
debug: Finding relations: proto
C:\peach>
```

Create the peach .xml file

We want to simulate us a bad client tries to attack the other clients and server. In drawing it is the following:



Because we have a lot of different packets we will use the file generated by the struct2peach. It looks like as:

```

<?xml version="1.0" encoding="utf-8"?>
<Peach version="1.0" author="struct2peach"
description="Autogenerated from header files">
  <!--
  A struct2peach.pl Generated DataModels.
  -->
  <!-- Import defaults for Peach instance -->
  <Include ns="default" src="file:defaults.xml" />
  <!-- Import some common complex data types into the
  "pt" namespace -->
  <Include ns="pt" src="file:PeachTypes.xml" />
  <!-- # STRUCTURES
  ##### -->
  <DataModel name="ChatClientStatusAck">
    <!-- # Flags ( AckFlags Flags;) -->
    <!-- # Unknown type: AckFlags -->
  </DataModel>
  <DataModel name="ChatClientSendMsgFoot">
    <!-- # SenderID ( int SenderID;) -->
    <Number name="SenderID" size="32" endian="little"
signed="true" />
    <!-- # TargetID ( int TargetID;) -->
    <Number name="TargetID" size="32" endian="little"
signed="true" />
    <!-- # MetadataSize ( int MetadataSize;) -->
    <Number name="MetadataSize" size="32" endian="little"
signed="true" />

```

```

        <!-- # Flags ( MsgFlags Flags;) -->
        <!-- # Unknown type: MsgFlags -->
        <!-- # Color ( int Color;) -->
        <Number name="Color" size="32" endian="little"
signed="true" />
        <!-- # FontSize ( int FontSize;) -->
        <Number name="FontSize" size="32" endian="little"
signed="true" />
        <!-- # MsgTime ( long long int MsgTime;) -->
        <!-- # Unknown type: long int -->
    </DataModel>
    <DataModel name="ChatClientDeliverAck">
        <!-- # Flags ( AckFlags Flags;) -->
        <!-- # Unknown type: AckFlags -->
    </DataModel>
    <DataModel name="ChatServerStatusMsg">
        <!-- # Flags ( ServerStatusFlags Flags;) -->
        <!-- # Unknown type: ServerStatusFlags -->
        <!-- # TargetID ( int TargetID;) -->
        <Number name="TargetID" size="32" endian="little"
signed="true" />
    </DataModel>
    <DataModel name="ChatServerDeliverMsgFoot">
        <!-- # SenderID ( int SenderID;) -->
        <Number name="SenderID" size="32" endian="little"
signed="true" />
        <!-- # TargetID ( int TargetID;) -->
        <Number name="TargetID" size="32" endian="little"
signed="true" />
        <!-- # MetadataSize ( int MetadataSize;) -->
        <Number name="MetadataSize" size="32" endian="little"
signed="true" />
        <!-- # Flags ( MsgFlags Flags;) -->
        <!-- # Unknown type: MsgFlags -->
        <!-- # Color ( int Color;) -->
        <Number name="Color" size="32" endian="little"
signed="true" />
        <!-- # FontSize ( int FontSize;) -->
        <Number name="FontSize" size="32" endian="little"
signed="true" />
        <!-- # MsgTime ( long int MsgTime;) -->
        <!-- # Unknown type: long int -->
    </DataModel>
    <DataModel name="ChatServerConnectResponse">
        <!-- # Flags ( ConnectResponseFlags Flags;) -->
        <!-- # Unknown type: ConnectResponseFlags -->
        <!-- # PeerID ( int PeerID;) -->
        <Number name="PeerID" size="32" endian="little"
signed="true" />
    </DataModel>
    <DataModel name="ChatClientConnectRequest">
        <!-- # ClientVer ( int ClientVer;) -->
        <Number name="ClientVer" size="32" endian="little"

```

```

signed="true" />
    <!-- # UsernameOffset ( short int UsernameOffset;) -->
    <!-- # Unknown type: hort int -->
</DataModel>
<DataModel name="ChatServerDeliverMsgHead">
    <!-- # MsgLength ( int MsgLength;) -->
    <Number name="MsgLength" size="32" endian="little"
signed="true" />
</DataModel>
<DataModel name="ChatClientSendMsgHead">
    <!-- # MsgLength ( int MsgLength;) -->
    <Number name="MsgLength" size="32" endian="little"
signed="true" />
</DataModel>
<DataModel name="ChatServerSendAck">
    <!-- # Flags ( AckFlags Flags;) -->
    <!-- # Unknown type: AckFlags -->
</DataModel>
<DataModel name="ChatHeader">
    <!-- # MsgSize ( int MsgSize;) -->
    <Number name="MsgSize" size="32" endian="little"
signed="true" />
    <!-- # MsgType ( MsgTypesFlags MsgType;) -->
    <!-- # Unknown type: MsgTypesFlags -->
</DataModel>

    <!-- #
#####
-->
</Peach>

```

The result looks like as follows:

```

<datamodel name="something">
    <!-- original line in the header file -->
<translated line>
...
</datamodel>

```

So we always see the original line in the header file, and the translated line. Sometimes we can find something like:

```

<datamodel name="something">
    <!-- original line in the header file -->
<!-- Unknown type: xxx -->
...
</datamodel>

```

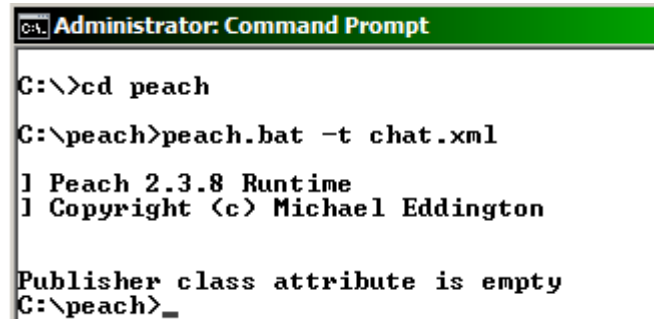
it means, that data type is not a native C data type, so it is not included in the struct2peach, so it could not translate it. We have to do it by hand, and we should add the logical connections between the elements.

To demonstrate this process I describe in detail how the "ChatClientSendMsgFoot" datamodel should be modified. First of all, I recommend to copy the c:\peach\template.xml as a new file for

example chat.xml.

Now open a command line, and enter to the peach directory, then test the chat.xml file:

```
cd c:\peach
peach -t chat.xml
```



```
C:\>cd peach
C:\peach>peach.bat -t chat.xml
Peach 2.3.8 Runtime
Copyright (c) Michael Eddington

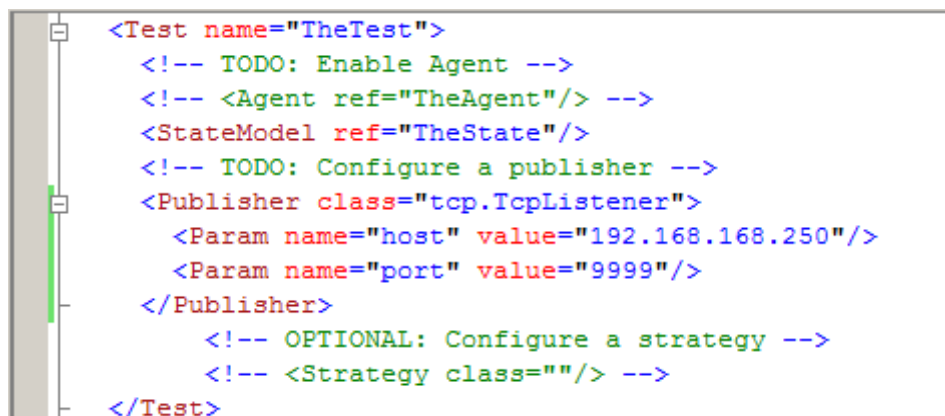
Publisher class attribute is empty
C:\peach>_
```

As we can see we get a nice error message. First we correct the peach file, to be able to test, if we do any syntax error.

We should find the
<Test name="The Test">
element and within that find the
<Publisher class=""/>

Then modify it to:

```
<Publisher class="tcp.Tcp">
  <Param name="host" value="192.168.168.101"/>
  <Param name="port" value="1234"/>
</Publisher>
```



```
<Test name="TheTest">
  <!-- TODO: Enable Agent -->
  <!-- <Agent ref="TheAgent"/> -->
  <StateModel ref="TheState"/>
  <!-- TODO: Configure a publisher -->
  <Publisher class="tcp.TcpListener">
    <Param name="host" value="192.168.168.250"/>
    <Param name="port" value="9999"/>
  </Publisher>
  <!-- OPTIONAL: Configure a strategy -->
  <!-- <Strategy class=""/> -->
</Test>
```

Now we will not got any error message:

```
Administrator: Command Prompt
C:\peach>peach.bat -t chat.xml
1 Peach 2.3.8 Runtime
1 Copyright (c) Michael Eddington
File parsed without errors.
C:\peach>
```

Create the datamodel

Now we will start to do the datamodel.

Then copy the original version inside it after the `<datamodel name="thedatamodel">...</datamodel>` section>. I recommend it, because now if you have a good xml editor the schema is validated, and it helps you in the typing. You will get something like:

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://phed.org/2008/Peach"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://phed.org/2008/Peach
/peach/peach.xsd">
  <!-- Import defaults for Peach instance -->
  <Include ns="default" src="file:defaults.xml"/>
  <!-- TODO: Create data model -->
  <DataModel name="TheDataModel">
    <String value="Hello World!" />
  </DataModel>
  <DataModel name="ChatClientSendMsgFoot">
    <!-- # SenderID ( int SenderID;) -->
    <Number name="SenderID" size="32" endian="little"
signed="true" />
    <!-- # TargetID ( int TargetID;) -->
    <Number name="TargetID" size="32" endian="little"
signed="true" />
    <!-- # MetadataSize ( int MetadataSize;) -->
    <Number name="MetadataSize" size="32" endian="little"
signed="true" />
    <!-- # Flags ( MsgFlags Flags;) -->
    <!-- # Unknown type: MsgFlags -->
    <!-- # Color ( int Color;) -->
    <Number name="Color" size="32" endian="little"
signed="true" />
    <!-- # FontSize ( int FontSize;) -->
    <Number name="FontSize" size="32" endian="little"
signed="true" />
    <!-- # MsgTime ( long long int MsgTime;) -->
    <!-- # Unknown type: long int -->
  </DataModel>
  <!-- TODO: Create state model -->
```

```

<StateModel name="TheState" initialState="Initial">
  <State name="Initial">
    <Action type="output">
      <DataModel ref="TheDataModel"/>
      <!-- <Data fileName="sample.png" /> -->
    </Action>
  </State>
</StateModel>
<!-- TODO: Configure Agent -->
<Agent name="TheAgent">
</Agent>
<Test name="TheTest">
  <!-- TODO: Enable Agent -->
  <!-- <Agent ref="TheAgent"/> -->
  <StateModel ref="TheState"/>
  <!-- TODO: Configure a publisher -->
  <Publisher class="" />
  <!-- OPTIONAL: Configure a strategy -->
  <!-- <Strategy class="" /> -->
</Test>
<!-- Configure a single run -->
<Run name="DefaultRun">
  <!-- TODO: Change log path if needed -->
  <Logger class="logger.Filesystem">
    <Param name="path" value="logs"/>
  </Logger>
  <Test ref="TheTest"/>
</Run>
</Peach>
<!-- end -->

```

As we can see the MsgTime, and the Flags has unknown data types, so correct those ones, and delete the unnecessary comment line. The MsgTime is 8 bytes (64 bit) long. So we can write it as:

```
<Number name="MsgTime" size="64" endian="little" signed="true" />
```

The flags are 4 byte (32 bit) long, so it can be written as:

```
<Number name="Flags" size="32" endian="little" signed="true" />
```

Now this part looks like as:

```

<DataModel name="ChatClientSendMsgFoot">
<Number name="SenderID" size="32" endian="little" signed="true" />
<Number name="TargetID" size="32" endian="little" signed="true" />
<Number name="MetadataSize" size="32" endian="little"
signed="true" />
<Number name="Flags" size="32" endian="little" signed="true" />
<Number name="Color" size="32" endian="little" signed="true" />
<Number name="FontSize" size="32" endian="little" signed="true" />
<Number name="MsgTime" size="64" endian="little" signed="true" />
</DataModel>

```

We know from the protokoll diagram, the "MetadataSize" is nothing else, but the size of the next four elements (flags, color, fontsize, and msgtime) together. To be able to add the size reference we should put these four elements into a block, then the model will change to:

```
<DataModel name="ChatClientSendMsgFoot">
<Number name="SenderID" size="32" endian="little" signed="true" />
<Number name="TargetID" size="32" endian="little" signed="true" />
<Number name="MetadataSize" size="32" endian="little"
signed="true" />
<Number name="Flags" size="32" endian="little" signed="true" />
<Number name="Color" size="32" endian="little" signed="true" />
<Number name="FontSize" size="32" endian="little" signed="true" />
<Number name="MsgTime" size="64" endian="little" signed="true" />
</DataModel>
```

This part is modified as:

```
<DataModel name="ChatClientSendMsgFoot">
<Number name="SenderID" size="32" endian="little" signed="true" />
<Number name="TargetID" size="32" endian="little" signed="true" />
<Number name="MetadataSize" size="32" endian="little"
signed="true">
  <Relation type="size" of="Metadata"/>
</Number>
  <Block name="Metadata">
    <Number name="Flags" size="32" endian="little" signed="true" />
    <Number name="Color" size="32" endian="little" signed="true" />
    <Number name="FontSize" size="32" endian="little"
signed="true" />
    <Number name="MsgTime" size="64" endian="little"
signed="true" />
  </Block>
</DataModel>
```

One might recognized, we just defined the Flags as a simple 4 byte (32 bit) integer value, but in reality those are flags, so every bit has some meaning. The meaning itself is defined in the header file (in the MsgFlags type). The question, if we want to define the meaning of every bit, or enough, to handle the flags as a four byte number. It depends on your decision. I write that version as well:

```
<DataModel name="ChatClientSendMsgFoot">
<Number name="SenderID" size="32" endian="little" signed="true" />
<Number name="TargetID" size="32" endian="little" signed="true" />
<Number name="MetadataSize" size="32" endian="little"
signed="true">
  <Relation type="size" of="Metadata"/>
</Number>
<Block name="Metadata">
  <Flags name="Flags" size="32" endian="little">
    <Flag name="colorincluded" size="1" position="0"></Flag>
    <Flag name="fontsizeincluded" size="1" position="1"></Flag>
    <Flag name="timeincluded" size="1" position="2"></Flag>
  </Flags>
```

```

    <Number name="Color" size="32" endian="little" signed="true" />
    <Number name="FontSize" size="32" endian="little"
signed="true" />
    <Number name="MsgTime" size="64" endian="little" signed="true" />
</Block>
</DataModel>

```

If you similarly copy and modify all the datamodels you by in the struct2peach you will get something like the next chat.xml file:

```

<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://phed.org/2008/Peach"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://phed.org/2008/Peach
/peach/peach.xsd">
    <!-- Import defaults for Peach instance -->
    <Include ns="default" src="file:defaults.xml"/>
    <!-- TODO: Create data model -->
    <DataModel name="TheDataModel">
        <String value="Hello World!" />
    </DataModel>

    <DataModel name="ChatHeader">
        <Number name="MsgSize" size="32" endian="little"
signed="true" />
        <Number name="MsgType" size="8" signed="false"></Number>
    </DataModel>
    <DataModel name="ChatServerSendAck">
        <Flags name="AckFlags" size="32" endian="little">
            <Flag name="accept" size="1" position="0"></Flag>
        </Flags>
    </DataModel>

    <DataModel name="ChatServerConnectResponse">
        <Flags name="AckFlags" size="32" endian="little">
            <Flag name="accept" size="1" position="0"></Flag>
        </Flags>
        <Number name="PeerID" size="32" endian="little"
signed="true" />
    </DataModel>
    <DataModel name="ChatClientConnectRequest">
        <Number name="ClientVer" size="32" endian="little"
signed="true" />
        <Number name="UsernameOffset" size="16" endian="little"
signed="true">
            <Relation type="offset" of="UserName"/>
        </Number>
        <String name="Magic" value="CHAT" nullTerminated="false"
type="char"></String>
        <String name="UserName" nullTerminated="false"
type="char"></String>
    </DataModel>

```



```

    <DataModel name="ChatClientSendMsgHead">
        <Number name="MsgLength" size="32" endian="little"
signed="true" />
    </DataModel>
    <DataModel name="ChatClientSendMsgFoot">
        <Number name="SenderID" size="32" endian="little"
signed="true" />
        <Number name="TargetID" size="32" endian="little"
signed="true" />
        <Number name="MetadataSize" size="32" endian="little"
signed="true">
            <Relation type="size" of="Metadata"/>
        </Number>
        <Block name="Metadata">
            <Flags name="Flags" size="32" endian="little">
                <Flag name="colorincluded" size="1" position="0"></Flag>
                <Flag name="fontsizeincluded" size="1"
position="1"></Flag>
                <Flag name="timeincluded" size="1" position="2"></Flag>
            </Flags>
            <Number name="Color" size="32" endian="little" signed="true"
/>
            <Number name="FontSize" size="32" endian="little"
signed="true" />
            <Number name="MsgTime" size="64" endian="little"
signed="true" />
        </Block>
    </DataModel>

    <DataModel name="ChatServerDeliverMsgHead">
        <Number name="MsgLength" size="32" endian="little"
signed="true" />
    </DataModel>
    <DataModel name="ChatServerDeliverMsgFoot">
        <Number name="SenderID" size="32" endian="little"
signed="true" />
        <Number name="TargetID" size="32" endian="little"
signed="true" />
        <Number name="MetadataSize" size="32" endian="little"
signed="true">
            <Relation type="size" of="Metadata"/>
        </Number>
        <Block name="Metadata">
            <Flags name="Flags" size="32" endian="little">
                <Flag name="colorincluded" size="1" position="0"></Flag>
                <Flag name="fontsizeincluded" size="1"
position="1"></Flag>
                <Flag name="timeincluded" size="1" position="2"></Flag>
            </Flags>
            <Number name="Color" size="32" endian="little" signed="true"
/>
            <Number name="FontSize" size="32" endian="little"
signed="true" />

```

```

        <Number name="MsgTime" size="64" endian="little"
signed="true" />
    </Block>
</DataModel>
<DataModel name="ChatClientStatusAck">
    <Flags name="AckFlags" size="32" endian="little">
        <Flag name="accept" size="1" position="0"></Flag>
    </Flags>
</DataModel>
<DataModel name="ChatClientDeliverAck">
    <Flags name="AckFlags" size="32" endian="little">
        <Flag name="accept" size="1" position="0"></Flag>
    </Flags>
</DataModel>

<DataModel name="ChatServerStatusMsg">
    <Flags name="ServerStatusFlags" size="32" endian="little">
        <Flag name="accept" size="1" position="0"></Flag>
    </Flags>
    <Number name="TargetID" size="32" endian="little"
signed="true" />
</DataModel>

<!-- TODO: Create state model -->
    <StateModel name="TheState" initialState="Initial">
        <State name="Initial">
            <Action type="output">
                <DataModel ref="TheDataModel"/>
                <!-- <Data fileName="sample.png" /> -->
            </Action>
        </State>
    </StateModel>
<!-- TODO: Configure Agent -->
    <Agent name="TheAgent">
</Agent>
    <Test name="TheTest">
        <!-- TODO: Enable Agent -->
        <!-- <Agent ref="TheAgent"/> -->
        <StateModel ref="TheState"/>
        <!-- TODO: Configure a publisher -->
    <Publisher class="tcp.Tcp">
        <Param name="host" value="192.168.168.101"/>
        <Param name="port" value="1234"/>
    </Publisher>
        <!-- OPTIONAL: Configure a strategy -->
        <!-- <Strategy class="" /> -->
    </Test>
    <!-- Configure a single run -->
    <Run name="DefaultRun">
        <!-- TODO: Change log path if needed -->
        <Logger class="logger.Filesystem">
            <Param name="path" value="logs"/>
        </Logger>

```

```

        <Test ref="TheTest"/>
    </Run>
</Peach>
<!-- end -->

```

Now we let us think over what we have done until now. We created all the data models separately, but in the communication flow we see that, some of them are connected. The best example is the header. It is at the beginning of every sent or received packet. We can copy the ChatHeader datamodel to the beginning of every other datamodel, but that is obviously a very inefficient solution, if something changes the modification will be difficult. So we do something else. We can reference in a datamodel to another one, then the content of the referenced datamodel will be included in the new one, and obviously new elements can be added, or existing in the referenced one could be overdefined. I do again the ChatClientSendMsgFoot as an example.

Copy the ChatHeader datamodel, and let us start to modify (we copy it, because we will need sometimes the not modified version, as I will show)

```

<DataModel name="ChatHeader">
  <Number name="MsgSize" size="32" endian="little" signed="true" />
  <Number name="MsgType" size="8" signed="false"></Number>
</DataModel>

```

Now modify it. First rename it to ChatHeaderWithBlock. Then add to the end of it a new Block element, After it we are able to add the logical constraint, the MsgSize should be equal to the size of the MsgData:

```

<DataModel name="ChatHeaderWithBlock">
  <Number name="MsgSize" size="32" endian="little" signed="true">
    <Relation type="size" of="MsgData" />
  </Number>
  <Number name="MsgType" size="8" signed="false"></Number>
  <Block name="MsgData">

    </Block>
</DataModel>

```

As we can see the MsgData block is left empty. Obviously, because it will contain different element in different cases.

Now Create a new ChatClientSendMsg datamodel (we have a ChatClientSendMsgHead, and ChatClientSendMsgFoot model, because in the header file it was divided to two parts), Then add this new ChatHeaderWithBlock to the ChatClientSendMsg datamodel as reference. Within this new datamodel we define a block, and the content of it should be the content of ChatClientSendMsgHead datamodel, so we simply add it as a reference. But in the ChatClientSendMsgHead there is a number, and that number should be the size of the Message, what we are going to define only now, so we must overdefine it. Also we should add the string contains the message, and we should add the Foot Block, it should contain the ChatClientSendMsgFoot, so we add it as reference. There is one more, but a crucial point, in the ChatHeaderWithBlock there is block called MsgData, and we should add all these new elements into that Block, because the MsgSize should contain the size of the message. The addition of an element to a referenced datamodel is very easy, you only should name it properly. The name must be in the format parentobject.elementname. From this one can deduce, the name tag otherwise must not contain the point as a valid character, we can use it only in case of referencing. We will get

something like:

```
<DataModel name="ChatClientSendMsg" ref="ChatHeaderWithBlock">
  <Block name="MsgData.Head" ref="ChatClientSendMsgHead">
    <Number name="MsgLength" size="32" endian="little"
signed="true">
      <Relation type="size" of="MsgData.Message"/>
    </Number>
  </Block>
  <String name="MsgData.Message" nullTerminated="false"
type="char"></String>
  <Block name="MsgData.Foot" ref="ChatClientSendMsgFoot">
  </Block>
</DataModel>
```

Again I recommend to test it after the modification, it helps to find if one do a syntax or other error. After the modification of the other datamodels the chat.xml file looks like about this:

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://phed.org/2008/Peach"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://phed.org/2008/Peach
/peach/peach.xsd">
  <!-- Import defaults for Peach instance -->
  <Include ns="default" src="file:defaults.xml"/>
  <!-- TODO: Create data model -->
  <DataModel name="TheDataModel">
    <String value="Hello World!" />
  </DataModel>

  <DataModel name="ChatHeader">
    <Number name="MsgSize" size="32" endian="little" signed="true"
/>
    <Number name="MsgType" size="8" signed="false"></Number>
  </DataModel>
  <DataModel name="ChatHeaderWithBlock">
    <Number name="MsgSize" size="32" endian="little"
signed="true">
      <Relation type="size" of="MsgData" />
    </Number>
    <Number name="MsgType" size="8" signed="false"></Number>
    <Block name="MsgData">

      </Block>
    </DataModel>

    <DataModel name="ChatServerSendAck" ref="ChatHeaderWithBlock">
      <Flags name="MsgData.AckFlags" size="32" endian="little">
        <Flag name="accept" size="1" position="0"></Flag>
      </Flags>
    </DataModel>
    <DataModel name="ChatServerConnectResponse"
ref="ChatHeaderWithBlock">
```

```

    <Flags name="MsgData.AckFlags" size="32" endian="little">
      <Flag name="accept" size="1" position="0"></Flag>
    </Flags>
    <Number name="MsgData.PeerID" size="32" endian="little"
signed="true" />
  </DataModel>
  <DataModel name="ChatClientConnectRequest"
ref="ChatHeaderWithBlock">
    <Number name="MsgData.ClientVer" size="32" endian="little"
signed="true" />
    <Number name="MsgData.UsernameOffset" size="16"
endian="little" signed="true">
      <Relation type="offset" of="UserName"/>
    </Number>
    <String name="MsgData.Magic" value="CHAT"
nullTerminated="false" type="char"></String>
    <String name="MsgData.UserName" nullTerminated="false"
type="char"></String>
  </DataModel>

  <DataModel name="ChatClientSendMsgHead">
    <Number name="MsgLength" size="32" endian="little"
signed="true" />
  </DataModel>
  <DataModel name="ChatClientSendMsgFoot">
    <Number name="SenderID" size="32" endian="little"
signed="true" />
    <Number name="TargetID" size="32" endian="little"
signed="true" />
    <Number name="MetadataSize" size="32" endian="little"
signed="true">
      <Relation type="size" of="Metadata"/>
    </Number>
    <Block name="Metadata">
      <Flags name="Flags" size="32" endian="little">
        <Flag name="colorincluded" size="1" position="0"></Flag>
        <Flag name="fontsizeincluded" size="1"
position="1"></Flag>
        <Flag name="timeincluded" size="1" position="2"></Flag>
      </Flags>
      <Number name="Color" size="32" endian="little" signed="true"
/>
      <Number name="FontSize" size="32" endian="little"
signed="true" />
      <Number name="MsgTime" size="64" endian="little"
signed="true" />
    </Block>
  </DataModel>
  <DataModel name="ChatClientSendMsg" ref="ChatHeaderWithBlock">
    <Block name="MsgData.Head" ref="ChatClientSendMsgHead">
      <Number name="MsgLength" size="32" endian="little"
signed="true">
        <Relation type="size" of="MsgData.Message"/>

```

```

        </Number>
    </Block>
    <String name="MsgData.Message" nullTerminated="false"
type="char"></String>
    <Block name="MsgData.Foot" ref="ChatClientSendMsgFoot">
    </Block>
</DataModel>

<DataModel name="ChatServerDeliverMsgHead">
    <Number name="MsgLength" size="32" endian="little"
signed="true" />
</DataModel>
<DataModel name="ChatServerDeliverMsgFoot">
    <Number name="SenderID" size="32" endian="little"
signed="true" />
    <Number name="TargetID" size="32" endian="little"
signed="true" />
    <Number name="MetadataSize" size="32" endian="little"
signed="true">
        <Relation type="size" of="Metadata"/>
    </Number>
    <Block name="Metadata">
        <Flags name="Flags" size="32" endian="little">
            <Flag name="colorincluded" size="1" position="0"></Flag>
            <Flag name="fontsizeincluded" size="1"
position="1"></Flag>
            <Flag name="timeincluded" size="1" position="2"></Flag>
        </Flags>
        <Number name="Color" size="32" endian="little" signed="true"
/>
        <Number name="FontSize" size="32" endian="little"
signed="true" />
        <Number name="MsgTime" size="64" endian="little"
signed="true" />
    </Block>
</DataModel>
<DataModel name="ChatServerDeliverMsg"
ref="ChatHeaderWithBlock">
    <Block name="MsgData.Head" ref="ChatServerDeliverMsgHead">
        <Number name="MsgLength" size="32" endian="little"
signed="true">
            <Relation type="size" of="MsgData.Message"/>
        </Number>
    </Block>
    <String name="MsgData.Message" nullTerminated="false"
type="char"></String>
    <Block name="MsgData.Foot" ref="ChatServerDeliverMsgFoot">
    </Block>
</DataModel>

<DataModel name="ChatClientStatusAck" ref="ChatHeaderWithBlock">
    <Flags name="MsgData.AckFlags" size="32" endian="little">
        <Flag name="accept" size="1" position="0"></Flag>

```

```

    </Flags>
  </DataModel>
  <DataModel name="ChatClientDeliverAck"
ref="ChatHeaderWithBlock">
    <Flags name="MsgData.AckFlags" size="32" endian="little">
      <Flag name="accept" size="1" position="0"></Flag>
    </Flags>
  </DataModel>

  <DataModel name="ChatServerStatusMsg" ref="ChatHeaderWithBlock">
    <Flags name="MsgData.ServerStatusFlags" size="32"
endian="little">
      <Flag name="accept" size="1" position="0"></Flag>
    </Flags>
    <Number name="MsgData.TargetID" size="32" endian="little"
signed="true" />
  </DataModel>

  <!-- TODO: Create state model -->
  <StateModel name="TheState" initialState="Initial">
    <State name="Initial">
      <Action type="output">
        <DataModel ref="TheDataModel"/>
        <!-- <Data fileName="sample.png" /> -->
      </Action>
    </State>
  </StateModel>
  <!-- TODO: Configure Agent -->
  <Agent name="TheAgent">
  </Agent>
  <Test name="TheTest">
    <!-- TODO: Enable Agent -->
    <!-- <Agent ref="TheAgent"/> -->
    <StateModel ref="TheState"/>
    <!-- TODO: Configure a publisher -->
  <Publisher class="tcp.Tcp">
    <Param name="host" value="192.168.168.101"/>
    <Param name="port" value="1234"/>
  </Publisher>
    <!-- OPTIONAL: Configure a strategy -->
    <!-- <Strategy class="" /> -->
  </Test>
  <!-- Configure a single run -->
  <Run name="DefaultRun">
    <!-- TODO: Change log path if needed -->
    <Logger class="logger.Filesystem">
      <Param name="path" value="logs"/>
    </Logger>
    <Test ref="TheTest"/>
  </Run>
</Peach>
<!-- end -->

```

Create the state model

Now we finished the datamodels so we can start to create the state model. It controls the order of messages we send and receive.

We can create a lot of different state model depending on which part of the communication we want to test. First we create a simple state model to fuzz only the connection request, and later we extend it.

The connection is quite easy, we just do the TCP connection, and after that we send a ChatClientConnectRequest message. And this is all, practically we are even not interested about the answer of the server. If there is any error it will freeze, what we will detect by the debugger..

To do it we write the following state model:

```
<StateModel name="TheState" initialState="Initial">
  <State name="Initial">
    <Action type="connect"></Action>
    <Action type="output">
      <DataModel ref="ChatClientConnectRequest"/>
    </Action>
  </State>
</StateModel>
```

Create the Agent

The job of the agent is to monitor the server, and the other client, what is a real one. To do a clear test we want to restart both the server and the real client at every test circle. Because of it we have to use the process.Process monitor class (the debugger.WindowsDebugEngine monitor is also capable to start a new process, if you give it the CommandLine parameter, but that is not capable to restart the process for every test circle).

We should start the server process, then the client process, it means two monitors, then we should attach to the started processes by the debugger.WindowsDebugEngine monitor. It can be done on the following way:

If we want to monitor the Heap in more detail then we can use the process.PageHeap monitor as well, I added it to this example too.

The agent definition looks like as follows:

```
<Agent name="TheAgent">
  <Monitor class="process.Process">
    <Param name="Command" value="C:\test\Release\server.exe"/>
    <Param name="RestartOnEachTest" value="true"/>
  </Monitor>
  <Monitor class="debugger.WindowsDebugEngine">
    <Param name="ProcessName" value="Server.exe"/>
  </Monitor>
  <Monitor class="process.PageHeap">
```



```

    <Param name="Executable" value="Server.exe"/>
</Monitor>
<Monitor class="process.Process">
    <Param name="Command" value="C:\test\Release\Client.exe"/>
    <Param name="RestartOnEachTest" value="true"/>
</Monitor>
<Monitor class="debugger.WindowsDebugEngine">
    <Param name="ProcessName" value="Client.exe"/>
</Monitor>
<Monitor class="process.PageHeap">
    <Param name="Executable" value="Client.exe"/>
</Monitor>
</Agent>

```

Create the Test section

The last step is to create the test itself, it does nothing else, but start to call the previously defined state model, and use the Agent. The default written in the template.xml is almost good, the only thing we should do is to enable the usage of the agent

This section looks like as follows:

```

<Test name="TheTest">
    <!-- TODO: Enable Agent -->
    <Agent ref="TheAgent"/>
    <StateModel ref="TheState"/>
    <!-- TODO: Configure a publisher -->
    <Publisher class="tcp.Tcp">
        <Param name="host" value="192.168.168.101"/>
        <Param name="port" value="1234"/>
    </Publisher>
    <!-- OPTIONAL: Configure a strategy -->
    <!-- <Strategy class=""/> -->
</Test>

```

Create the Run section

This is the last part of the xml file, if one want to modify the logging behaviour, should modify it, for me now the default in the template.xml is good, so I do not modify it.

Whole chat.xml file

```

<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://phed.org/2008/Peach"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://phed.org/2008/Peach
/peach/peach.xsd">
    <!-- Import defaults for Peach instance -->
    <Include ns="default" src="file:defaults.xml"/>

```

```

    <!-- TODO: Create data model -->
    <DataModel name="TheDataModel">
        <String value="Hello World!" />
    </DataModel>

    <DataModel name="ChatHeader">
        <Number name="MsgSize" size="32" endian="little" signed="true"
/>
        <Number name="MsgType" size="8" signed="false"></Number>
    </DataModel>
    <DataModel name="ChatHeaderWithBlock">
        <Number name="MsgSize" size="32" endian="little"
signed="true">
            <Relation type="size" of="MsgData" />
        </Number>
        <Number name="MsgType" size="8" signed="false"></Number>
        <Block name="MsgData">

            </Block>
        </DataModel>

        <DataModel name="ChatServerSendAck" ref="ChatHeaderWithBlock">
            <Flags name="MsgData.AckFlags" size="32" endian="little">
                <Flag name="accept" size="1" position="0"></Flag>
            </Flags>
        </DataModel>
        <DataModel name="ChatServerConnectResponse"
ref="ChatHeaderWithBlock">
            <Flags name="MsgData.AckFlags" size="32" endian="little">
                <Flag name="accept" size="1" position="0"></Flag>
            </Flags>
            <Number name="MsgData.PeerID" size="32" endian="little"
signed="true" />
        </DataModel>
        <DataModel name="ChatClientConnectRequest"
ref="ChatHeaderWithBlock">
            <Number name="MsgData.ClientVer" size="32" endian="little"
signed="true" />
            <Number name="MsgData.UsernameOffset" size="16"
endian="little" signed="true">
                <Relation type="offset" of="UserName"/>
            </Number>
            <String name="MsgData.Magic" value="CHAT"
nullTerminated="false" type="char"></String>
            <String name="MsgData.UserName" nullTerminated="false"
type="char"></String>
        </DataModel>

        <DataModel name="ChatClientSendMsgHead">
            <Number name="MsgLength" size="32" endian="little"
signed="true" />
        </DataModel>
        <DataModel name="ChatClientSendMsgFoot">

```

```

    <Number name="SenderID" size="32" endian="little"
signed="true" />
    <Number name="TargetID" size="32" endian="little"
signed="true" />
    <Number name="MetadataSize" size="32" endian="little"
signed="true">
        <Relation type="size" of="Metadata"/>
    </Number>
    <Block name="Metadata">
        <Flags name="Flags" size="32" endian="little">
            <Flag name="colorincluded" size="1" position="0"></Flag>
            <Flag name="fontsizeincluded" size="1"
position="1"></Flag>
            <Flag name="timeincluded" size="1" position="2"></Flag>
        </Flags>
        <Number name="Color" size="32" endian="little" signed="true"
/>
        <Number name="FontSize" size="32" endian="little"
signed="true" />
        <Number name="MsgTime" size="64" endian="little"
signed="true" />
    </Block>
</DataModel>
<DataModel name="ChatClientSendMsg" ref="ChatHeaderWithBlock">
    <Block name="MsgData.Head" ref="ChatClientSendMsgHead">
        <Number name="MsgLength" size="32" endian="little"
signed="true">
            <Relation type="size" of="MsgData.Message"/>
        </Number>
    </Block>
    <String name="MsgData.Message" nullTerminated="false"
type="char"></String>
    <Block name="MsgData.Foot" ref="ChatClientSendMsgFoot">
    </Block>
</DataModel>

<DataModel name="ChatServerDeliverMsgHead">
    <Number name="MsgLength" size="32" endian="little"
signed="true" />
</DataModel>
<DataModel name="ChatServerDeliverMsgFoot">
    <Number name="SenderID" size="32" endian="little"
signed="true" />
    <Number name="TargetID" size="32" endian="little"
signed="true" />
    <Number name="MetadataSize" size="32" endian="little"
signed="true">
        <Relation type="size" of="Metadata"/>
    </Number>
    <Block name="Metadata">
        <Flags name="Flags" size="32" endian="little">
            <Flag name="colorincluded" size="1" position="0"></Flag>
            <Flag name="fontsizeincluded" size="1"

```

```

position="1"></Flag>
    <Flag name="timeincluded" size="1" position="2"></Flag>
</Flags>
    <Number name="Color" size="32" endian="little" signed="true"
/>
    <Number name="FontSize" size="32" endian="little"
signed="true" />
    <Number name="MsgTime" size="64" endian="little"
signed="true" />
</Block>
</DataModel>
<DataModel name="ChatServerDeliverMsg"
ref="ChatHeaderWithBlock">
    <Block name="MsgData.Head" ref="ChatServerDeliverMsgHead">
        <Number name="MsgLength" size="32" endian="little"
signed="true">
            <Relation type="size" of="MsgData.Message"/>
        </Number>
    </Block>
    <String name="MsgData.Message" nullTerminated="false"
type="char"></String>
    <Block name="MsgData.Foot" ref="ChatServerDeliverMsgFoot">
    </Block>
</DataModel>

<DataModel name="ChatClientStatusAck" ref="ChatHeaderWithBlock">
    <Flags name="MsgData.AckFlags" size="32" endian="little">
        <Flag name="accept" size="1" position="0"></Flag>
    </Flags>
</DataModel>
<DataModel name="ChatClientDeliverAck"
ref="ChatHeaderWithBlock">
    <Flags name="MsgData.AckFlags" size="32" endian="little">
        <Flag name="accept" size="1" position="0"></Flag>
    </Flags>
</DataModel>

<DataModel name="ChatServerStatusMsg" ref="ChatHeaderWithBlock">
    <Flags name="MsgData.ServerStatusFlags" size="32"
endian="little">
        <Flag name="accept" size="1" position="0"></Flag>
    </Flags>
    <Number name="MsgData.TargetID" size="32" endian="little"
signed="true" />
</DataModel>

<!-- TODO: Create state model -->
<StateModel name="TheState" initialState="Initial">
    <State name="Initial">
        <Action type="connect"></Action>
        <Action type="output">
            <DataModel ref="ChatClientConnectRequest"/>
        </Action>

```

```

        </State>
    </StateModel>
    <!-- TODO: Configure Agent -->
    <Agent name="TheAgent">
    <Monitor class="process.Process">
        <Param name="Command" value="C:\test\Release\server.exe"/>
        <Param name="RestartOnEachTest" value="true"/>
    </Monitor>
    <Monitor class="debugger.WindowsDebugEngine">
        <Param name="ProcessName" value="Server.exe"/>
    </Monitor>
    <Monitor class="process.PageHeap">
        <Param name="Executable" value="Server.exe"/>
    </Monitor>
    <Monitor class="process.Process">
        <Param name="Command" value="C:\test\Release\Client.exe"/>
        <Param name="RestartOnEachTest" value="true"/>
    </Monitor>
    <Monitor class="debugger.WindowsDebugEngine">
        <Param name="ProcessName" value="Client.exe"/>
    </Monitor>
    <Monitor class="process.PageHeap">
        <Param name="Executable" value="Client.exe"/>
    </Monitor>
</Agent>
    <Test name="TheTest">
        <!-- TODO: Enable Agent -->
        <Agent ref="TheAgent"/>
        <StateModel ref="TheState"/>
        <!-- TODO: Configure a publisher -->
    <Publisher class="tcp.Tcp">
        <Param name="host" value="192.168.168.101"/>
        <Param name="port" value="1234"/>
    </Publisher>
        <!-- OPTIONAL: Configure a strategy -->
        <!-- <Strategy class=""/> -->
    </Test>
    <!-- Configure a single run -->
    <Run name="DefaultRun">
        <!-- TODO: Change log path if needed -->
        <Logger class="logger.Filesystem">
            <Param name="path" value="logs"/>
        </Logger>
        <Test ref="TheTest"/>
    </Run>
</Peach>
<!-- end -->

```

Run the fuzzer

Start the peach fuzzer in agent mode. It can be done by the

```
Administrator: Command Prompt - peach.bat -a
Microsoft Windows [Version 6.0.6002]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\Users\Administrator>cd \peach
C:\peach>peach.bat -a

] Peach 2.3.8 Runtime
] Copyright (c) Michael Eddington

] Peach Agent

//-> Listening on [9000] with no password
```

Then start the fuzzer

```
Administrator: Command Prompt

C:\>cd peach
C:\peach>peach.bat chat.xml_
```

Then wait.

Fuzz the message sending

This state model fuzz only the login process of the chat client, but most probably we want to fuzz as the client sends the message.

To do it we modify the state model. After we sent the ChatClientConnectRequest we should wait for a ChatServerConnectResponse packet.

When we got the ChatServerConnectResponse packet we should check in it, if the server accepted the connect request or not. If not we should go to the end, and start a new test cycle. If the server accepted the connection we should send a message. To do it we should examine the received packet. It can be done by the changestate action. This action jumps to another state, and it has a when attribute, where we can write the condition of the jump.

But to jump to another state we should define more states. Now I define the following one:

- **Initial:** it will do the TCP connection, and immediately after the connection jump to the next (doconnection) state without any condition:

```
<State name="Initial">
  <Action type="connect"></Action>
  <Action name="changetodoconnection" type="changeState"
ref="doconnection"></Action>
</State>
```

- **doconnection:** this state will send the ChatClientConnectRequest. To be able to really connect we must define acceptable default data, it can be done by the data element. Then in

this state we wait for the ChatServerConnectResponse data type. After we got it we should examine the content, now need the changestate, and the condition. In one condition we examine if we got a ServerConnectResponse packet. If not we jump to the last state (End). Otherwise we examine the next condition, if the server accepted our connection attempt, if not we jump again to the last state (End). And finally (non of the previous comditions were true) we jump to the next state (dosendmessage)

```
<State name="doconnection">
  <Action type="output">
    <DataModel ref="ChatClientConnectRequest">
      </DataModel>
    <Data DataModel="ChatClientConnectRequest">
      <Field name="MsgType" value="0x01" valueType="hex"/>
      <Field name="MsgData.ClientVer" value="0x00000001"
valueType="hex"/>
      <Field name="MsgData.Magic" value="CHAT"/>
      <Field name="MsgData.UserName" value="USER00"/>
    </Data>
  </Action>
  <Action name="ReceiveServerConnectResponse" type="input">
    <DataModel ref="ChatServerConnectResponse"/>
  </Action>
  <Action name="CheckIfServerConnectResponse" type="changeState"
ref="End" when="int(StateModel['doconnection']
['ReceiveServerConnectResponse']['ChatServerConnectResponse']
['MsgType'].getInternalValue()) != 2"></Action>
  <Action name="CheckIfConnectionRejected" type="changeState"
ref="End" when="int(StateModel['doconnection']
['ReceiveServerConnectResponse']['ChatServerConnectResponse']
['MsgData']['AckFlags']['accept'].getInternalValue()) !=
1"></Action>
  <Action name="changetodosendmessage" type="changeState"
ref="dosendmessage"></Action>
</State>
```

- **dosendmessage:** we must create the the message, but again we have a problem, the server just sent us a peerID, and we should set the senderID exactly to this value. To do this we use another action type, what is called slurp. This element is capable to copy a value from one element to another element. In case of slurp one must define the source, and the destination in the format of //State//actionname//valuenam. If the valuenam contains point ve should separate those.instead of point by a // (practically we should define the path to the element we want to copy and every level is separated by //).

```
<State name="dosendmessage">
  <Action type="slurp"
valueXpath="//doconnection//ReceiveServerConnectResponse//MsgData/
/PeerID" setXpath="//doChatClientSendMsg//SenderID" />
  <Action name="doChatClientSendMsg" type="output">
    <DataModel ref="ChatClientSendMsg"></DataModel>
    <Data DataModel="ChatClientSendMsg">
      <Field name="MsgType" value="0x10" valueType="hex"/>
    </Data>
  </Action>
```

```

    <Action name="changetodosendmessage" type="changeState"
ref="End"></Action>
</State>

```

- End: this is the last state, it close the TCP connection, and we can step to the next iteration

```

<State name="End">
    <Action type="close"></Action>
</State>

```

All these these modifications together will give the next state model:

```

<StateModel name="TheState" initialState="Initial">
    <State name="Initial">
        <Action type="connect"></Action>
        <Action name="changetodoconnection" type="changeState"
ref="doconnection"></Action>
    </State>

    <State name="doconnection">
        <Action type="output">
            <DataModel ref="ChatClientConnectRequest">
            </DataModel>
            <Data DataModel="ChatClientConnectRequest">
                <Field name="MsgType" value="0x01" valueType="hex"/>
                <Field name="MsgData.ClientVer" value="0x00000001"
valueType="hex"/>
                <Field name="MsgData.Magic" value="CHAT"/>
                <Field name="MsgData.UserName" value="USER00"/>
            </Data>
        </Action>
        <Action name="ReceiveServerConnectResponse" type="input">
            <DataModel ref="ChatServerConnectResponse"/>
        </Action>
        <Action name="CheckIfServerConnectResponse" type="changeState"
ref="End" when="int(StateModel['doconnection']
['ReceiveServerConnectResponse']['ChatServerConnectResponse']
['MsgType'].getInternalValue()) != 2"></Action>
        <Action name="CheckIfConnectionRejected" type="changeState"
ref="End" when="int(StateModel['doconnection']
['ReceiveServerConnectResponse']['ChatServerConnectResponse']
['MsgData']['AckFlags']['accept'].getInternalValue()) !=
1"></Action>
        <Action name="changetodosendmessage" type="changeState"
ref="dosendmessage"></Action>
    </State>

    <State name="dosendmessage">
        <Action type="slurp"
valueXpath="//doconnection//ReceiveServerConnectResponse//MsgData/
/PeerID" setXpath="//doChatClientSendMsg//SenderID" />
        <Action name="doChatClientSendMsg" type="output">
            <DataModel ref="ChatClientSendMsg"></DataModel>

```



```
    <Data DataModel="ChatClientSendMsg">
      <Field name="MsgType" value="0x10" valueType="hex"/>
    </Data>
  </Action>
  <Action name="changetodosendmessage" type="changeState"
ref="End"></Action>
</State>

  <State name="End">
    <Action type="close"></Action>
  </State>
</StateModel>
```