IMS/DNS (2012.01.28)

Fault tolerance and Load Balancing for IMS

Gábor Fehér, BME

Table of Conten

1 Problem statement

In this scenario the users want to receive an IMS (IP Multimedia Subsystem) service. The services run on several Application Servers (AS). The aim of the service provider is to provide quality service to the users, so the CSCF (Call Session Control Function) should reach the AS fast and reliably. For this purpose the same service is provided by multiple ASes, and they are all available for the users. The goal is to provide reliability and in the case of errors, fast recovery.

Technically, when a user initiate a service request, the user agent sends a SIP request message to the IMS serves. The request reaches the S-CSCF, where the S-CSCF query the HSS (Home Subscriber Server) about the location of the given service. The HSS tells the location by the domain name of the Application Servers. After that the S-CSCF sends the SIP request to the corresponding AS. To get the actual IP location (i.e. the IP address) of the AS the S-CSCF uses DNS (Domain Name Service) queries.

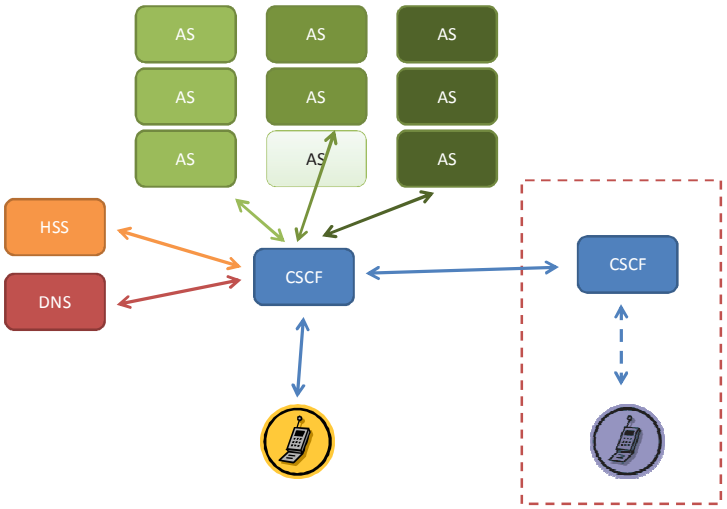

Figure 1 – Service in IMS

However SIP request sent to the AS may be blocked when the AS is unreachable on the network. According to the SIP RFC, in the case of TCP and SCTP transport protocols the S-CSCF waits for 64xT1 time and then initiates a new attempt to an alternative AS. The problem is that T1 is set to 0.5 seconds by default, which is rather high (32 seconds) for a telecommunication service. In the case of UDP this waiting time could be even longer as in this case it is the S-CSCF that manages the retransmission of possibly lost UDP packets. This 32 seconds is an unacceptable delay, the response must be faster. In other hand the S-CSCF waits for that much time only in the case when there is no error message for the connection attempt. As the S-CSCF gets and error message - just like host/port unreachable ICMP messages or a TCP reset - it aborts the waiting state.

The goal of this study is to create a procedure where the S-CSCF is able to connect to the AS within the seconds order. Beyond the increase of dependability, further goal could be to provide load balancing for the AS connections.

Further assumptions:

- During the service provisioning the ASes store states and for this reason during this time they cannot be changed to other servers. Transferring these states to an other AS is out of scope of the study. The rest of the IMS will handle these (hopefully rare) situations.
- On the AS side, any positive response for the TCP or SCTP connection attempt means that the AS is working and it is willing to provide service for the requesting user. In this case, no further connection analysis is required. In the case of UDP, there is no connection establishment phase, so the response message is already the part of the service provisioning.
- The ASes are able to give information on their load conditions. Providing this information also means that they are working and they are accessible. There is no need for further analyzes in order to assure their availability.
- The SIP implementation in the S-CSCF supports the DNS SRV and NAPTR queries. Based on the DNS responses for a given AS query, the S-CSCF utilize the defined orders and weights to select alternative IP addresses when running into blocked ASes. SIP forking is not supposed to be supported.

2 Technical background

2.1 DNS queries in the CSCF

The CSCF use the RFC 3263 in order to get the IP address of the destination. The RFC describes a method, which DNS queries should be sent by the client to get the address of the destination. According to this RFC, the client sends a DNS NAPTR record lookup first. This record tells the contact points to the questioned domain. As an example for the DNS's answer:

```
example.com NAPTR 10 100 "S" "SIP+D2U" "" _sip._udp.example.com.
example.com NAPTR 20 100 "S" "SIP+D2T" "" _sip._tcp.example.com.
```

The answer tells whether there are SIP service entries on the server or not. It also tells the transport protocol and the name, which should be asked from the DNS server. Beside the entries we can find order and preference information as well.

If there is no transport protocol defined in the SIP messages, the client will select a transport protocol here. The next DNS query will ask for an SRV record of the destination service. As an example for the DNS's answer:

```
_sip._udp.example.com SRV 5 100 5060 sip-udp01.example.com.
_sip._udp.example.com SRV 10 100 5060 sip-udp02.example.com.
```

This information already presents the name of the servers and also the IP port, where we can connect. There are also priority and weight information associated to the servers. The client should connect to

the servers based on their priority (the lower the value, the higher the priority is). If there are multiple entries with the same priority, the clients should choose one randomly based on the weight information.

Finally the client should ask the IP address of the server. This is a normal DNS query using the A or AAAA record.

There are some cases when there is no NAPTR information available in the DNS for a particular domain. In this case the client will test all the supported transport protocols using the SRV lookups. If the SRV records are missing, then the client will try the default port and UDP when it is an unsecure connection and TCP when it is a secure connection.

2.2 TCP SYN connections establishment

Whenever the CSCF wants to communicate with the AS using a TCP connection, the TCP channel should be setup first. The TCP connection setup uses a three-way handshake method. In this method the initiator first sends a TCP packet to the receiver without any payload, but setting the SYN flag. Normally the receiver sends back a TCP packet without any payload, setting the SYN and ACK flags. However, at this point there is no connection yet, as the initiator has to confirm the received SYN ACK packet with an ACK packet. Sending the ACK packet the initiator will consider the connection open. The receiver will consider the connection open when it receives the ACK packet.

It may happen the receiver is not available and this way it does not reply on the initial TCP SYN packet. Assuming network problems the initiator tries to resend the SYN packet at certain periods. The rule for the retransmission according to the base RFC is that with each retransmission the backoff time is doubled. The same RFC defines that in the connection establishment phase, as there is no RTT value available, the retransmission timer is initially set to 3 seconds. Usually there is a maximum value defined for the number of unsuccessful retransmissions. After this number of retransmissions the TCP connection initiation is considered as dead and the calling code will get and error message. In Linux systems this retransmission value is 5 by default. Considering the default values for the connection establishment, the CSCF should wait 93 seconds before it would give up the connection initiation to the AS if the AS is unreachable or simply not responding.

On the receiver side the connection is not built up until the TCP ACK message from the three way handshake arrives. This also means that the application layer, i.e. the code where the listening socket was invoked, does not get know about the new connection until this ACK message. If the TCP ACK message in the connection set up phase is never sent out to the AS, then it is possible to abort the connection without any application layer interaction. Of course a missing TCP ACK message would leave the receiver in a half open state, which could lead to resource shortage problems, but there is the TCP RST message that can clear up the situation and free up any previously reserved resource.

3 Possible Solutions

We grouped the possible solutions into three groups based on their orientations. Within the SIP oriented solutions, we investigated the possibility to change the SIP protocol itself in order to achieve

faster responses. Among the DNS based solutions we studied how to modify the DNS in order to achieve our goals. At the end, with network oriented solutions we investigated the possibilities to change architecture or network algorithms to get better reliability and load balancing.

1. SIP oriented solution
   - Parallel SIP conversations / SIP forking

2. DNS based solutions
   - Reliable Server Pooling Protocol (RSerPool) – RFC 5351
   - DNS Load Balancing (Round Robin DNS)

3. Network oriented solutions
   - Anycast routing
   - Cisco Hot Standby Router Protocol (HSRP) – RFC 2281
   - Virtual Router Redundancy Protocol – RFC 2338
   - SIP/TCP hacking

3.1 Parallel SIP conversation

SIP forking gives the possibility to issue SIP request to multiple servers at the same time. Originally the idea is to terminate the call on multiple sites, e.g. both the mobile phone and the desk phone rings at the same time.

In the current situation the CSCF would fork the SIP request and send it to multiple ASes. To determine the number and the order of requested ASes the CSCF would utilize the DNS information. When one of the AS would respond to the request, the CSCF would cancel all the other parallel request.

This solution results an overhead traffic on the network and also in the ASes. This is a clear disadvantage, however the network and the AS might not suffer from overloading. The major disadvantage of this solution, that it is not easy to cancel or undo an ongoing request due to accounting and other reasons.

Because, according to our assumptions, the CSCF does not support SIP forking, and because there is a major disadvantage in the solution, we do not recommend this method.

3.2 Cisco Hot Standby Router Protocol (HSRP) – RFC 2281

Using this method, which is an RFC standard, the reliability of network connections can be increased. In HSRP there are backup routers that replace faulty routers immediately. When the route between the CSCF and AS is protected, the connection's dependability is increased.

The disadvantage of the solution is that there should be backup routers along all routes to ASes, which is a hard requirement. Also, there is no support for load balancing.

As we cannot expect that remote AS connections are protected with HSRP capable routers and backups, we do not recommend this solution. However, whenever it is possible this or similar connection protection protocols have benefits on the local network and thus local CSCF and AS connections.

3.3 Virtual Router Redundancy Protocol (VRRP) – RFC 2338

In our view, this is a similar solution to the previous HSRP solution. VRRP is supported by many routers. Advantages and disadvantages are also similar to the HSRP case. As a solution it is **not recommended**, however it is good to increase the local AS connection reliability.

3.4 Anycast routing

Anycast routing is available in IPv6 networks. With this special routing the packets are forwarded to a destination, which is in a special addressed anycast group. Although the anycast address refer to a group of destinations, finally only one destination gets the transmitted packet. The decision is made by the routers. This method is also used for geographic load balancing, however it is not wide spread. For example there are DNS and NTP anycast servers.

In the current environment the ASes providing the same service would form a single anycast group. Hence the successful connection establishment would be provided by the routers. Load balancing and dependability is based on the routing algorithms.

Disadvantage of this method is that it is not easy to assure that the subsequent packets of the same sessions would be forwarded to the same AS. Also, routers should be prepared to support IPv6 anycast. For this reason, this method is **not recommended**.

3.5 Reliable Server Pooling Protocol (RSerPool) – RFC 5351

The RSerPool protocol provides access to services running from a server pool. The services in the pool are identical and the pool user is connected to one of them. This is an RFC standard from 2008, however the first draft appeared at IETF already in 2001. The design criteria were: lightweight, scalable, expandable, simple and real time responses.

3.5.1 RSerPool introduction

The RSerPool architecture can be seen on the following figure:

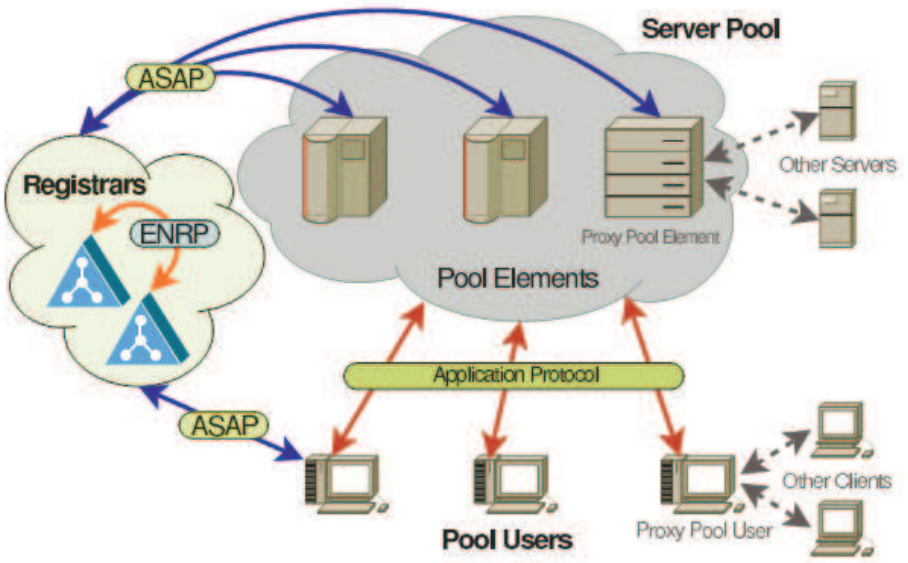

Figure 2 –RserPool acrhitectrue (source: PhD dissertation of Thomas Driebholz)

The servers in the *Server Pool* are called *Pool Elements (PE)*. Each PE in the *Pool* provides exactly the same service. Te individual PEs are identified with a *Pool Element Identifier (PE ID)*, which is a randomly created 32 bit value that the PE gets ate the registration. The Pool is identified by the *Pool Handle (PH)*, which is an arbitrary binary sequence. Most often the Pool Handle is a character sequence with a meaning, just like "WebServerPool". The group of Pools is called *Handlespace*. The handlespaces are usually locals, so the PH are not organized in hierarchy. The following figure shows an example where the Pool stores the IPv4 and IPv6 transport addresses of the PEs, the PE selection policy, plus load information.

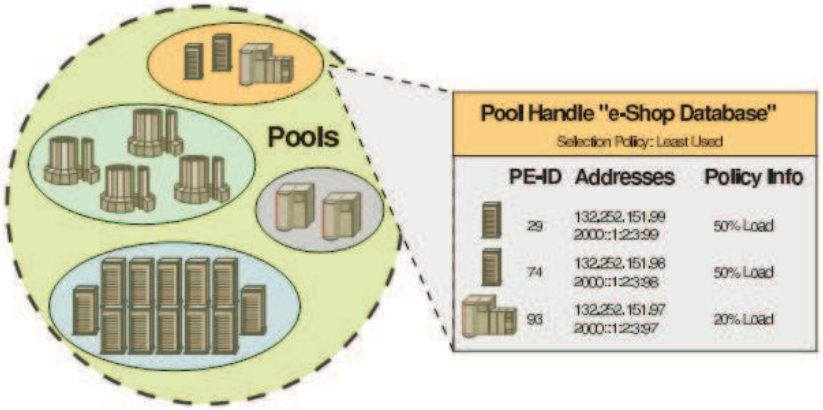

Figure 3 – Handlespace and Pool Hanle (source: PhD dissertation of Thomas Driebholz)

The Handlespace is managed by redundant *Pool Registrars (PR)*. Each Pool Registrar gets a random and unique 32 bit identifier, this is called *Registrar ID (PR ID)*. The communication between the PRs is based on the *Endpoint haNdlespace Redundancy Protocol (ENRP)*, for this reason PRs are often called as ENRP

servers. Using the ENRP communication the PRs negotiate all the entries of the Handlespace, and this way all the PRs are functionally similar. When a PR falls out the other PRs replace them. Using multicast channels the PRs are advertised, so the Pool users are know how to reach them.

The services are requested by *Pool Users (PU)*. They can choose freely a Pool Registrar and ask to reach one Pool Element of a given Pool Handle. The PR, when the requested PH exists, gives the list of the PEs. The list is ordered by the Pool Policy. There can be more than one Pool Policies. Examples for such policies are random selection or least loaded selection.

The registration and the Pool access is done by communicating via the *Aggregate Server Acess Protocol (ASAP)*. The registration of the PE can happen on arbitrary PR, while later this PR will be the *Home PR (PR-H)* for the PE. The PR-H is responsible to monitor those PEs that were registered by itself. The monitoring is performed by periodic *ASAP Endpoint Keep-Alive messages*, moreover the PE also repeats its registration periodically.

The RSerPool architecture can be utilized as well by those servers and users who do not know the ASAP protocol. The *Proxy Pool User (PPU)* handles the non RSerPool users, while the *Proxy Pool Element (PPE)* handles the non RSerPool servers.

3.5.2 Utilizing RSerPool

Supposing that nor the Application Server neither the CSCF code is available for modifications, creating new functions supporting the RSerPool mechanism is not feasible. Instead, it is possible to use the proxy units and those proxies can connect the RSerPool architecture to the IMS system. There are more than one possible scenario for this.

According to the first proposed scenario, the Application Servers would be connected through their individual Proxy Pool Elements. The Pool Users are the CSCFs that use the common Proxy Pool User to connect to the Application Servers. As the PPE are redundant, there is no need for extra protection. However, since the PPU is a single point of failure, its dependability should be improved by other techniques. As PPU is local for the CSCF, its protection can be satisfied.

Whenever the CSCF wants to reach the Application Server, it turns to the HSS. The HSS using the DNS directs the CSCF to the PPU. The PPU is dedicated to the requested service. The PPU uses the ASAP protocol and talks to one of the PR to get the list of the PPEs. Based on the information given by the PR, the PPU connects to the surely available PPE. The PPU embeds the CSCF messages, while PPE takes out the embedded messages and deliver them to the AS. From the view of the IMS system, this is not a fully transparent solution, as the DNS configuration should be altered to return with the PPU address of the requested service. The DNS response is the single IP address of the local PPU, and actually there is no need for multiple addresses (as long as scalability and dependability is not an issue here) because the further redundancy is handled by the PPU.

The combined architecture is presented on the following figure:

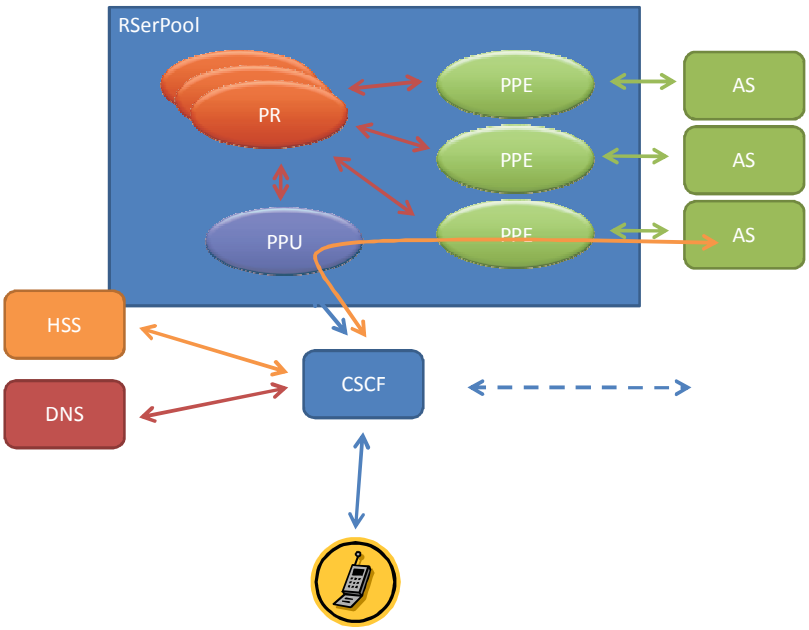

Figure 4 – IMS and RSerPool combination (1)

The second proposed scenario connects the RSerPool architecture to the IMS system via the DNS. The DNS is the Pool User. The ASes are connected with PPEs, just like in the previous case. During the service request, the CSCF turns to the DNS to get information about the location of the AS. Here, before the DNS would reply, it talks to the PR and ask for an available PPE of the requested AS. PPEs are local for the ASes, so we suppose that if the PPE is available then AS is available as well. The DNS learns the AS address from the selected PPE and return this address to the CSCF. It is also possible that the DNS give back a set of addresses to different ASes with weights and priorities. The CSCF will connect directly to the preferred AS, and this connection is already outside of the RSerPool architecture.

This architecture is presented on the following figure:

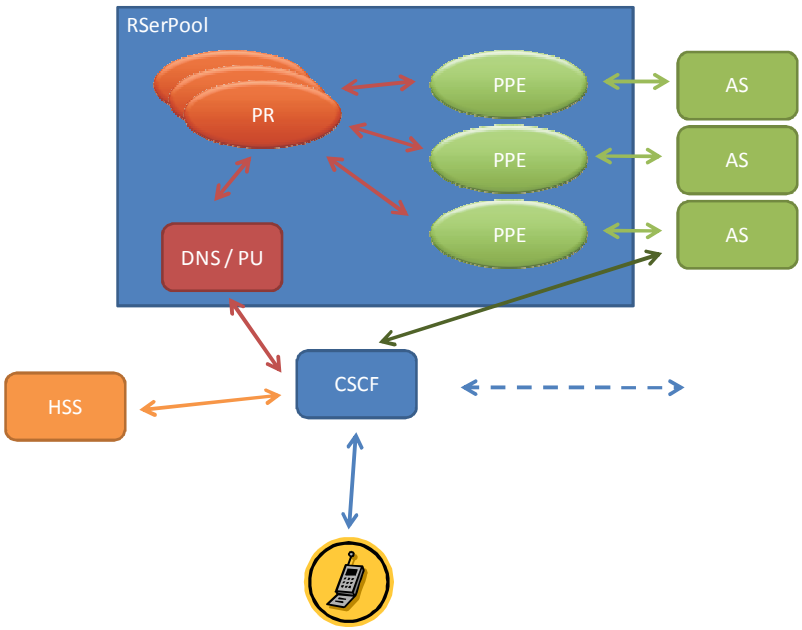

Figure 5 - IMS and RSerPool combination (2)

In order to build up the mentioned scenario, we need a special DNS server that talks the ASAP protocol.

3.6 SIP/TCP hacking

For practical reasons it is not possible to alter the SIP T1 timer's value. For the same reason the 64x multiplier in the SIP implementation that signals the time out for the connection establishment, cannot be changed. This means that the SIP stack always wait for 32 seconds, unless the connection establishment is not aborted by the other peer. With the SIP hacking solution we could initiate a connection establishment abort from outside of the SIP scope.

3.6.1 TCP parameter modifications

When the CSCF uses the TCP transport protocol to the AS, due to its connection oriented nature, it is possible to terminate the connection before the SIP stack's time would expire. As it is described in the technical background section, the regular TCP stack waits 93 seconds before it would declare that the receiver is not available. This is more than the SIP timeout, however setting up different parameters for the TCP flow, this could be faster. Designing maximum 3 seconds for a connection setup, and maximum 3 attempts in the case of lost TCP SYN message, the TCP retransmission timer should be set to 430 milliseconds. This way within roughly 1.5 seconds the CSCF would send 3 connection establishment requests to the AS, and if there is no answer within 3 seconds, the connection would be closed.

Advantages

- There is no need to modify the architecture or other elements of the IMS system. The only thing, which is required, is to set the TCP parameters at the CSCF.
- TCP parameters are easily accessible on most operating systems.

- Using TCP connections, the AS application is not connected until the network connection is set up.

Disadvantages

- The TCP parameter is system wide, therefore it affects other connections as well. Due to the shorter connection establishment timeout other connections may fail.
- More frequent connection establishment packets may cause more network congestion. This should be analyzed before applying this solution.
- Does not provide load balancing.

3.6.2 TCP hacker (wrapper)

A more sophisticated solution is to use a TCP hacker (a kind of proxy or wrapper) to establish the TCP connection between the CSCF and the AS. In this case there is no modification in the CSCF itself, but there is a new element in the architecture. The TCP hacker is transparent, neither the CSCF nor the AS knows about it. During its operation, the TCP hacker captures and modifies the TCP connection setup messages faking the source and destination addresses to pretend a direct connection.

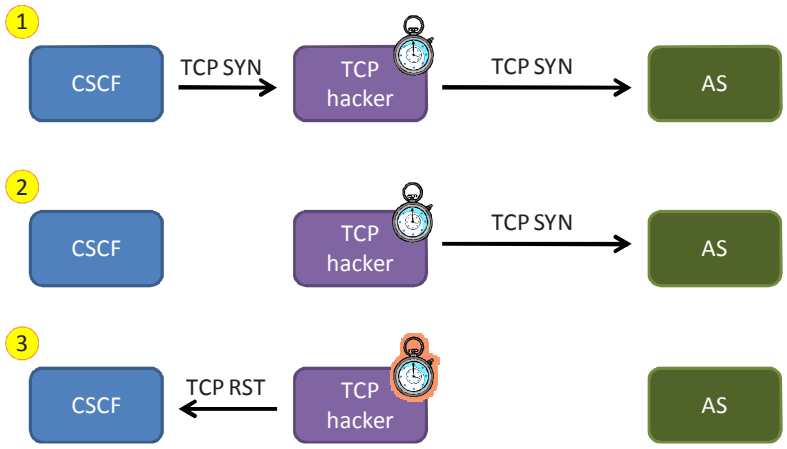

Figure 6 – Unsuccessful connection establishment

Step 1: When the TCP hacker gets a TCP SYN connection request coming from the CSCF, it sends the request to the AS. Step 2: Ignoring the TCP timing parameters at the CSCF, the TCP hacker may send out arbitrary number of TCP SYN messages with arbitrary interval. Step 3: Using its own timer, when the timer expires, it aborts the connection establishment sending TCP RST messages to the CSCF. It is not possible to alert the AS about the connection tear, as there was no answer from that direction.

Otherwise, when the TCP hacker captures a connection setup TCP SYN ACK message from the AS, it removes itself from the connection, so further on the CSCF and the AS will communicate directly.

The retransmission interval for the TCP SYN messages should not be exponential as it is with the normal TCP connection establishment. As AS addresses are from a limited set only, the TCP hacker may learn the average RTT and the loss rate for the given ASes. During the connection setup phase the TCP hacker

can utilize this knowledge and thus it may send the TCP SYN packets according to a calculated timing. With an initial approach, we can say that the TCP hacker should send out the TCP SYN packets with uniform 1.5 RTT intervals. This approach results faster response to connection problems and allows more retires to cope with higher package drops. As an example, assuming 500ms RTT, the TCP hacker sends 3 TCP SYN messages in 1.5 seconds (0 ms, 750 ms, 1500 ms) and if there is no response within 2.25 seconds, then it closes the connection.

Advantages

- There is no need to modify any of the IMS architecture elements.
- TCP hacker can be implemented as a firewall. Filtering of TCP connection establishment messages are easy and fast.
- Similar to the previous solution, the AS application is not connected until the network connection is set up.

Disadvantages

- More frequent connection establishment packets may cause more network congestion. This should be analyzed before applying this solution.
- Does not provide load balancing.

3.6.3 UDP and SCTP transport protocols

The UDP transport protocol is connectionless, so there is no connection setup phase. For this reason the TCP/SIP hacking solution is not available for the UDP case. SCTP is similar to TCP in the manner that it has connection setup phase. This means, that all the TCP algorithms, mentioned before can be modified to support the SCTP transport protocol. SCTP hacking is not detailed in this document.

3.6.4 Sequential or parallel connection attempts

The TCP hacker can be placed into the IMS system different ways.

In the first case, the DNS returns a set of IP addresses of the desired AS and the CSCF tries to connect them one by one, until it the connection setup is successful. In this case, the TCP hacker is placed right after the CSCF and kills (RST) all the connection attempts, which setup time exceeds a certain limit.

In the other case the CSCF could initiate parallel requests to the set of IP addresses that DNS returned. The TCP hacker, which is right after the CSCF, would enable to finish the fastest connection setup, while all the other connections would be killed before their initiation.

There are certain problems with the second case. First, it requires modifying the code of the CSCF in order to be able to initiate parallel connections. Second, the TCP hacker should know which connections belong to the same service request, as only one can remain from them. These two reasons make this solution too complex. However applying a SIP proxy could help a lot here. With this approach the TCP hacker would be a local SIP proxy at the same time. The DNS would return the address of the SIP proxy/TCP hacker so the CSCF would turn to it assuming that it is the AS. At this point, the SIP proxy/TCP hacker asks again the DNS about the given service, and now the DNS would return real AS addresses.

This way the TCP hacker knows all the relating IP addresses. The parallel TCP request would be issued and handled by the TCP hacker itself. Moreover, with the help of the SIP proxy, it would be possible for the CSCF to use the UDP transport protocol. Of course UDP is translated into TCP later by the proxy.

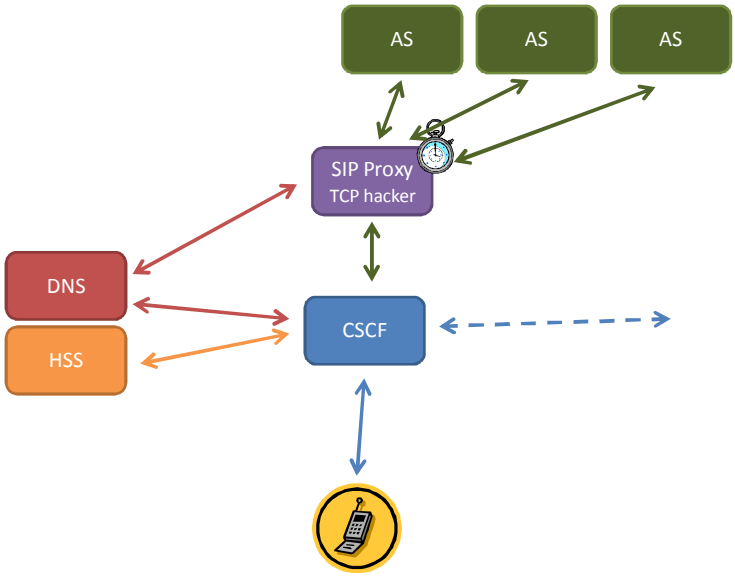

Figure 7 – SIP proxy/TCP hacker doing parallel requests

3.7 Fast DNS updates

The IP addresses of the ASes are provided by the DNS. If the DNS could monitor the availability or the load of the ASes, then the non responding ASes could be cleared from the returned IP address set and the right AS order could be given. The DNS updates on the monitored resources are strictly local, this information would not be propagated, as the propagation is either too slow or else it is fast enough, but also it is too resource demanding.

The availability and load information can be retrieved by other protocols. Using the RSerPool protocol gives back the previously mentioned scenario. Other protocols, such as SNMP, can be considered as well.

4 TCP hacker implementation

Based on the discussion of the possible solutions, there are two alternatives, which should be further analyzed. The first proposed solution is the TCP hacker the second one is the DNS server, which utilizes the information got from RSerPool.

The TCP hacker modifies the connection establishment phase of the TCP sessions. As written in the technical background section, TCP requires 3 messages to be exchanged in order to establish the session. The TCP protocol also describes the timing of these messages. TCP hacker would alter the timing of the first message, namely the SYN message, which is sent from the initiator to the remote node. No other parts of the TCP session would be affected. This also means that the TCP hacker plays

role only in the establishment phase and immediately after the initiator gets a positive response from the remote node, the hacker leaves the session.

4.1 Placement of the TCP hacker

According to the TCP hacker idea, the implementation is should be transparent to the other elements of the network. So, the TCP hacker is considered as a box that can be placed into the network without configuring any other elements. In a simple architecture, the TCP hacker should be placed between the CSCF and the IP router that serves as a gateway out of the CSCF's subnet. Considering a simple scenario with Ethernet interfaces the placement is the following (The IP addresses of the interfaces are fictional):

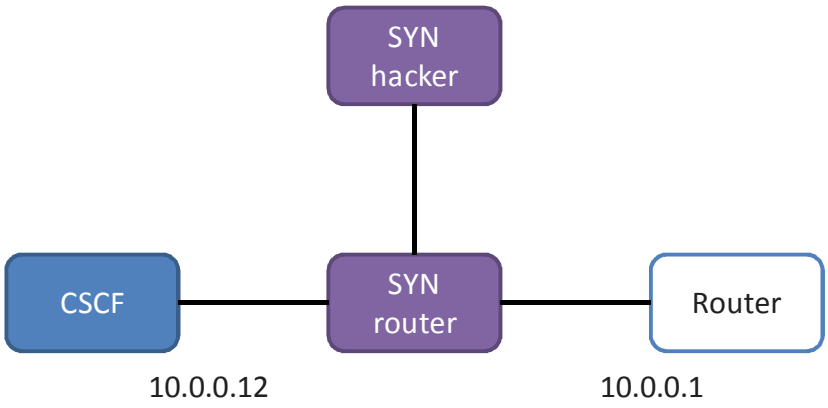

Figure 8 - Placing the SYN hacker into a simple network

Neither the SYN router nor the SYN hacker has IP address. It is not necessary to give them any, as they are communicating on the data link layer. However, for manageability reasons, it is advised to provide IP addresses for them, or even better, they can connect to control machines with secondary interfaces.

4.2 TCP connection setup messages

The following cases describe the behavior of the TCP hacker.

4.2.1 TCP hacker in a successful connection establishment

The connection establishment would perform the following steps:

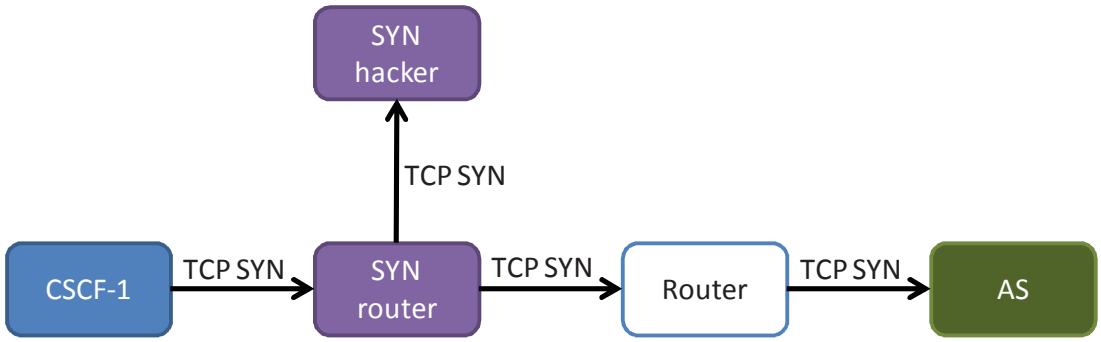

Figure 9 – TCP hacker, successful connection establishment, step 1

In the first step the CSCF wants to create a connection to the desired AS. It already asked the DNS server about the IP address of the AS, so it is ready to initiate the first message, which is the TCP SYN message. The SYN message contains the IP addresses of the CSCF and the AS and also the TCP port numbers. In the planned architecture there is a SYN router in between the CSCF and its router. The role of the SYN router is to redirect all frames, where the TCP SYN or RST bit is set (The necessity of RST bit is described later) to the SYN hacker. This is a layer 2 redirection, however the frame should be analyzed on the TCP level. Beside the redirection, the TCP SYN frame is forwarded directly to the router as well. In this step, the SYN hacker gets the frame, and from its content it gets know that the specific CSCF wants to connect to the specific AS.

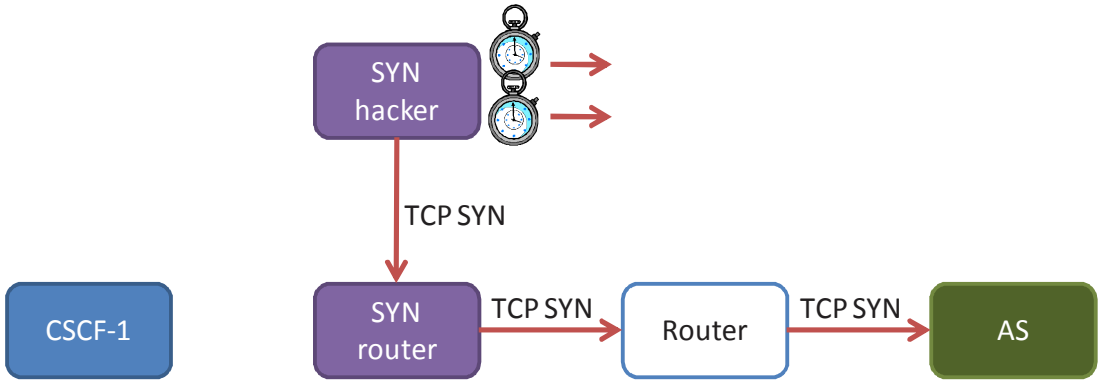

Figure 10 – TCP hacker, successful connection establishment, step 2

In the second step, the SYN hacker creates new SYN messages based on the received one. The timing of the new TCP SYN messages is different as it is in the standard. The interval between the SYN messages is short due to the strict time limitation requirement. The first TCP SYN message is sent immediately towards the AS. This initiation message goes through the SYN router and the outgoing router. The SYN router forwards the TCP SYN message, since it knows that the message comes from the SYN hacker. If the timer of the next SYN message would expire, then it would be sent the same way as the first one.

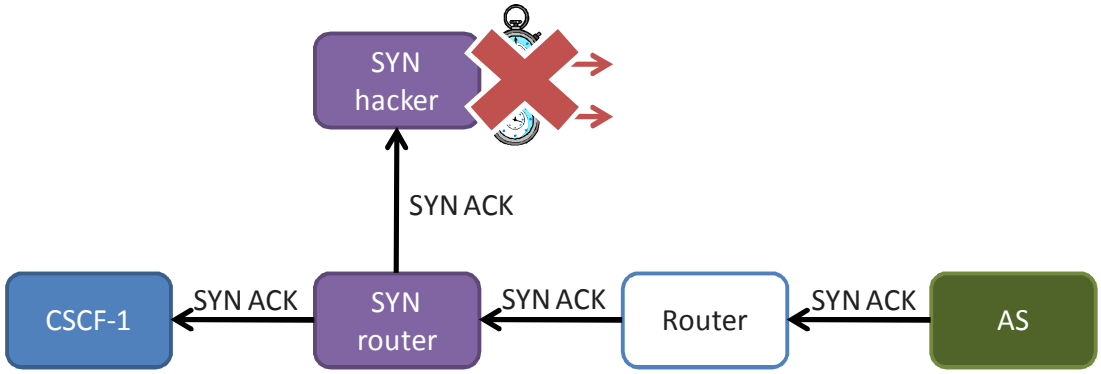

Figure 11 – TCP hacker, successful connection establishment, step 3

In the third step, assuming that the TCP SYN message arrived successfully to the remote node and that node responded with a TCP SYN ACK message, this message arrives to the SYN router. The SYN router

forwards the frame to the CSCF, however one copy of the frame is sent to the SYN hacker as well. The SYN hacker cancels all the timers that were set due to the corresponding TCP session. Further on the SYN hacker steps out from the session.

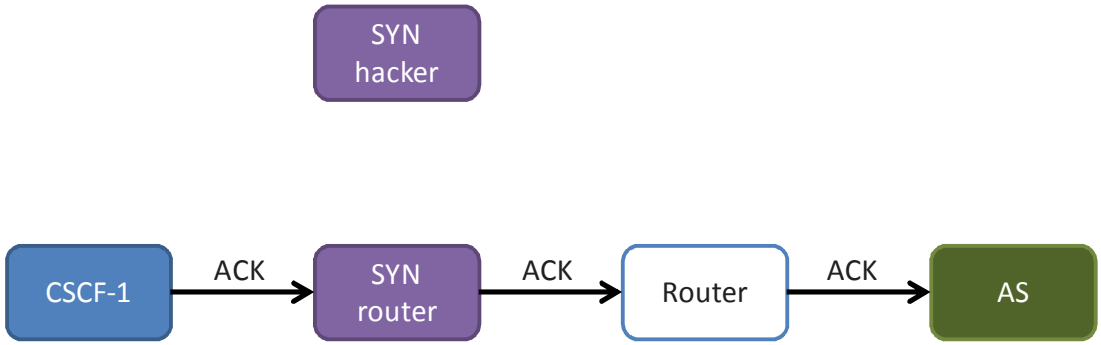

Figure 12 – TCP hacker, successful connection establishment, step 4

In the final step there are no more SYN messages in the connection. As a result the SYN router forwards all the frames between the CSCF and the Router.

4.2.2 TCP hacker in an unsuccessful connection establishment

When the connection establishment is unsuccessful (e.g. the destination port is closed), the remote node may send a TCP RST message back to the initiator. In that case the first 2 steps of the initiator are the same as in the previous, successful case. However the third step is different:

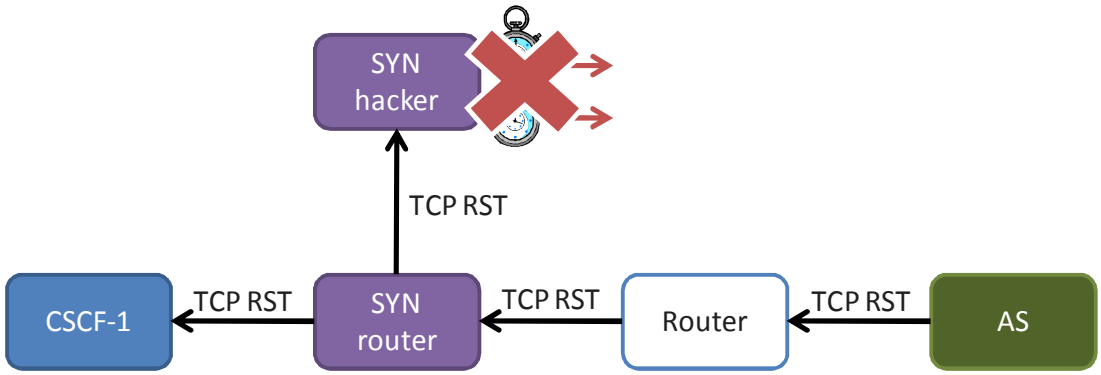

Figure 13 – TCP hacker, unsuccessful connection establishment, step 3

When the AS is not ready to accept the connection it sends back a TCP RST message. The RST message arrives to the SYN router and the router, besides the forwarding to the message to the initiator, also sends a copy of the frame to the SYN hacker. The SYN hacker cancels all timers associated to the corresponding session in order to avoid sending further TCP SYN messages.

4.2.3 TCP hacker with an unresponsive AS

Due to network failures, configuration problems or overloaded systems, it is possible that the remote AS does not answer at all. In this case the first 2 steps of the initiator are the same as in the other cases, however the third step is different:

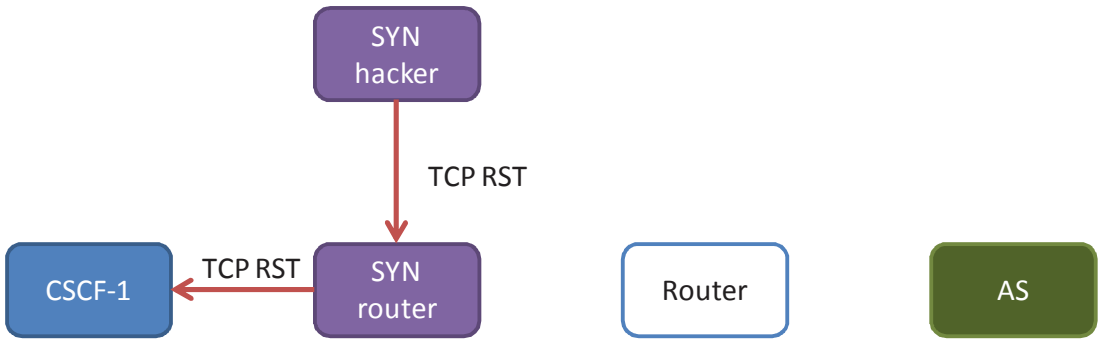

Figure 14 – TCP hacker, expired connection time

In this situation the SYN hacker will try to resend the TCP SYN messages as it is done in the first step. Meanwhile it is also possible that the CSCF sends a repeated TCP SYN message. This message should be ignored by the SYN hacker. Finally, when there is no more retransmission trial at the SYN hacker, the SYN hacker sends out TCP RST messages to the CSCF to terminate the connection establishment. The RST message tells that the source IP address is the AS IP address, so it looks like as the AS would deny the connection establishment.

4.2.4 Parallel TCP setup messages

It may happen that some messages belonging to the same session appear roughly at the same time at different points of the architecture. This is will not make any problems. Generally the first appeared massage is valid, while the rest is invalid.

It may happen that both the TCP RST coming from the SYN hacker and TCP SYN ACK coming from the AS reach the SYN router at the same time. This way the CSCF will get both messages. Only the first message will be considered as a valid message by the CSCF, the other will be silently dropped. If the first message is the TCP SYN ACK message, then the connection will be set up. In the other case the connection will be deleted, while the AS will send more SYN ACK messages thinking that the previous messages lost somewhere. These SYN ACK messages cause no further problems. It would be possible to send an RST message to the AS in this case, however we estimate, that it is a really rare situation, so it does not worth the effort to implement such code.

It may happen as well that the AS and the SYN hacker send their TCP RST messages roughly the same time, so they reach the SYN router at once. Again, this is not a problem. The first message that reaches the CSCF will be valid, while the second one would be invalid, so the CSCF will ignore it.

4.3 Linux prototype implementation

There is a prototype implementation for the TCP hacker. It is running on a Linux machine and the purpose is making a proof of concept implementation. Moreover the scalability of the implementation can be tested.

The implementation is a more simple form of the TCP hacker than described above. Namely, the SYN router and the SYN hacker machines are placed into a single node. For this reason instead of layer 2

messages, they are communicating within the machine. The implemented SYN hacker is depicted on the following figure:

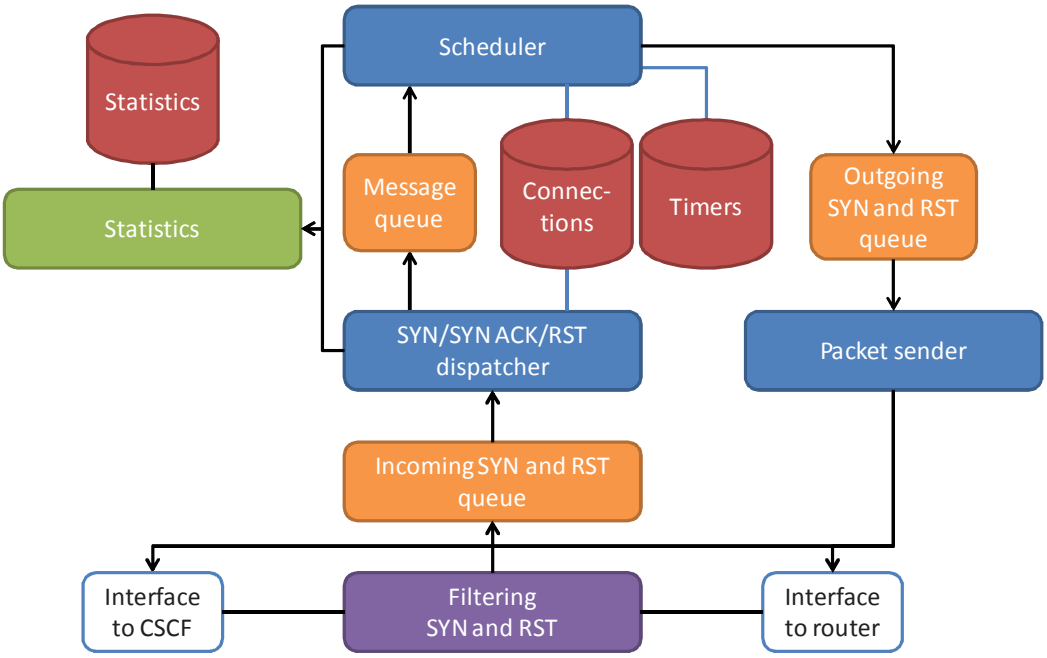

Figure 15 - Implementation of the TCP hacker

4.3.1 Filtering TCP SYN and RST messages

The SYN hacker should capture important TCP connection messages, namely the TCP SYN, TCP SYN ACK and TCP RST messages. To implement packet capturing in Linux we have two different solutions. One of them is the raw socket capture method, the other one is the utilization of the firewall architecture. We decided to implement the second option.

The Linux machine running the TCP hacker has at least two interfaces. The interfaces are bridged together on the data link layer. Thus, the TCP hacker operates as a network bridge. This operation gives us transparency, so the CSCF and the router do not notice the presence of the TCP hacker. In addition to the bridging, on the TCP hacker machine we can filter the forwarded messages. The Linux firewall put all bridged frames into the FORWARD chain.

The SYN and RST filter is simply a firewall rule that filer all messages that are valid TCP messages and either the SYN or the RST flag is set. The filtered frames are placed into a message specific userspace queue. Three different netfilter queues are defined: one queue for TCP SYN, one queue for TCP SYN ACK and one queue for TCP RST. The frames that are taken into these queues will wait there until they are processed. All other frames are bridged immediately.

The main reason to use the firewall architecture instead of raw sockets was the better configurability. Firewall rules can be easily tailored the filter only the desired traffic. Using this approach more complex scenarios can be built up.

The SYN and RST filter plays the role of the SYN router in the non simplified architecture. The message queue would be the Ethernet channel on which the SYN router and the SYN hacker communicate.

4.3.2 SYN, SYN ACK and RST dispatcher

The role of the dispatcher is to get out the frames from the NetFilter Queue and handle them according to their role. The dispatcher is the main thread in the code.

In the case of a SYN packet received, the dispatcher looks up the existing connections using the source and destination IP address and port pairs. When there is already a connection with these parameters, the packet is silently dropped. This situation happens during the normal TCP connection setup phase, when the initiator repeats the SYN sending. When there is no connection associated with the IP address and port pairs, the dispatcher creates a connection entry with the new connection parameters and notifies the scheduler that there is a new connection. Finally the dispatcher instructs the firewall to forward the SYN packet to its destination.

In the case of a SYN-ACK packet received, the dispatcher looks up the existing connections and if there is a matching entry then it notifies the scheduler to delete it together with the corresponding timers. If there is no such entry in the connections database, the dispatcher does nothing. This is a suspicious situation, since normally the there should be a previous SYN message that should have allocated an entry for the connection. In either case, the dispatcher signals to the firewall to forward the SYN-ACK message to its destination.

In the case of an RST packet received, the existing connections entry and the corresponding timers are deleted and the packet is forwarded to its destination, similarly to the previous SYN-ACK case.

4.3.3 Scheduler

The scheduler is responsible to create and delete entries in the connections and the timers databases, and also to repeat the SYN messages for pending connections. This is a separate thread in the code, and communicates to other threads via message queues. The message queues are FIFO channels. The scheduler thread is a sleeping thread. It wakes up for two reasons: one, if there is an incoming message from the dispatcher; second, when a timer expires.

Upon receiving a connection entry creation message, the scheduler creates the connection in the connections database and also sets up a timing entry in the timers database to be able to manage the repeated SYN sending. Receiving a connection entry deletion message, all the timers associated to the connection and the connection itself is deleted.

When a timer expires, the scheduler thread wakes up if it would be sleeping and looks up the connection associated to the expired timer. If there is more SYN retransmission possibilities, the scheduler puts the SYN sending request to the queue of the packet sender and signals to the packet sender to send it. In the current implementation the number of retransmissions is fixed to 3. The time interval between two attempts is also fixed, this is 500 ms.

The scheduler maintains two databases. The first one stores the connections. This is a hexadecimal word tree. The depth of the tree depends on the keys we want to store in the tree. In our case this is 12, which consists of a 4 byte IP source address, a 4 byte IP destination address, a 2 byte source port and a 2 byte destination port. Searching in this database requires 12 steps if there is an entry we are looking for. This solution is a kind of right balance between the memory consumption and operation speed. The other database stores timed actions. This is a queue, where the actions are in the order of their expiration time. The soonest action is on the front of the queue. Storing a new timed action may require to go through the whole queue. This could take significant amount of time depending on the actual length of the queue. This operation can be speed up using indexes for the queue. This feature is not yet implemented.

4.3.4 Packet sender

The packet sender sends out the SYN and RST messages. This is a separate thread just because not to block other procedures waiting for the IO operations. It receives the requests for packet sending from the scheduler. When such a request arrives, the packet sender assembles the packet using the right addressing and sequence numbers, and after that using a raw TCP socket, it sends them out.

4.3.5 Statistics

There is an independent thread, called statistics, which role is to provide information about the internals of the TCP hacker. Indeed, in the current implementation, the statistics thread grabs this information right from the threads, which store these values in a place, which is externally available. The statistics are reported periodically, by default every 10 seconds.

4.4 Possible scalability and robustness improvements

Currently the TCP hacker implementation contains the SYN router and the SYN hacker modules in one piece. As it was mentioned before, this is a simplification for the proof of concept prototype and it is not mandatory at all. Even in the prototype, the SYN router and SYN hacker are separate processes.

Using multiple SYN routers and SYN hackers may boost the scalability and robustness of the solution.

4.4.1 Multiple SYN hackers

It is possible that the SYN router communicates to multiple SYN hackers. The following figure shows this situation.

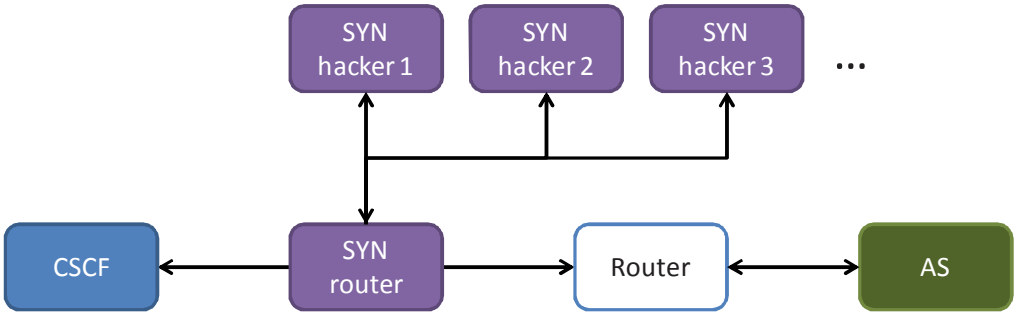

Figure 16 - SYN router with multiple SYN hackers

Here, between the SYN router and SYN hackers there is only a data link layer. It is easier to use a layer 2 communication here and keep the original layers above than create encapsulated packets and applying outer IP addresses.

The SYN router must schedule which connection request goes to which SYN hacker. There are numerous alternatives how to make it. There could be two kinds of approaches. In the first, stateful case, the SYN router would store the ID of the SYN hacker that is associated to the connection, while in the second, stateless case, there would be no association stored. The first approach requires more work from the SYN router, while the latter one may require more work from the SYN hackers. In the stateless case when the SYN router does not know where to send the incoming SYN ACK and RST packets, so it should broadcast them, thus all SYN hackers should look up the corresponding connections.

In the stateful case the SYN router has a picture of the load on the SYN hackers, as it knows how many connections are associated to each individual SYN hacker. Thus the task delegation can be based on the actual load or else it can be random. In the stateless case this is not possible. First, there is no knowledge about the load on the SYN hackers. Second, the task delegation cannot be random, since in that case repeated TCP SYN requests from the same source could go to different SYN hackers and they could work in parallel on the same connection with different timing. A better approach is to select the SYN hacker according to a fix attribute of the connections, which can be considered as a uniformly distributed. For example, some modulus of the source port number. This approach also eliminates the need for broadcasting SYN ACK and RST messages, since the TCP hacker is dedicated to the connection.

Multiple SYN hackers this way may contribute to create a more scalable solution. Considering a stateless SYN router making the connection delegations based on the parameters of the connections, the individual SYN hackers should handle the fraction of the connections. Roughly twice as much SYN hacker can handle twice as much connection.

The multiplication of SYN hackers can increase the robustness as well. It is possible that the SYN router delegates the actual connection setup to two or more different SYN hackers, but with different timings. For example if there is 3 attempts for a connection setup, it could be delegated to three different SYN hackers. The very first TCP SYN packet for a connection is forwarded without the SYN hackers. If at least one of the SYN hacker, whose task is to maintain the connection setup, is working, then there will be a repeated SYN message if it is necessary.

4.4.2 Multiple SYN routers

Not just the SYN hackers, but SYN routers can be repeated in the architecture as well.

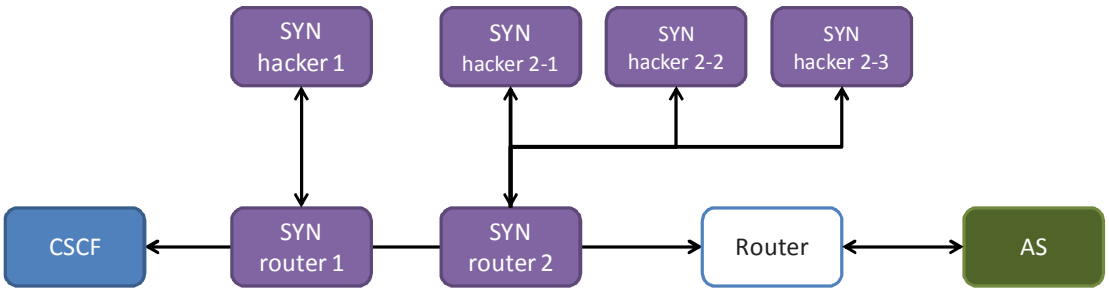

Figure 17 - Multiple SYN routers

The role of the SYN router is simple: it should select the TCP connection setup messages and it should delegate the connection setup to a SYN hacker. Normally this is a job for a firewall, so a single machine should cope with this task. However, if this is not the case, the SYN router can be multiplied as well.

Since there is no interaction between the SYN routers, in order to avoid parallel connection setup handlings, the SYN routers should handle discrete sets of connection. Similarly to the stateless case of multiple SYN hackers, the connections can be shared among the SYN routers based on the connections parameters.

When the goal is to increase the robustness, similarly to the multiple SYN hackers case, it is possible to use multiple SYN routers and perform the same tasks with different timings.

It is also possible to combine the multiple SYN routers and multiple SYN hackers into a mesh of SYN hackers using multiple entry points through the SYN routers. Depending on the setup of the SYN routers, this can increase both the scalability and robustness. The following figure shows this architecture:

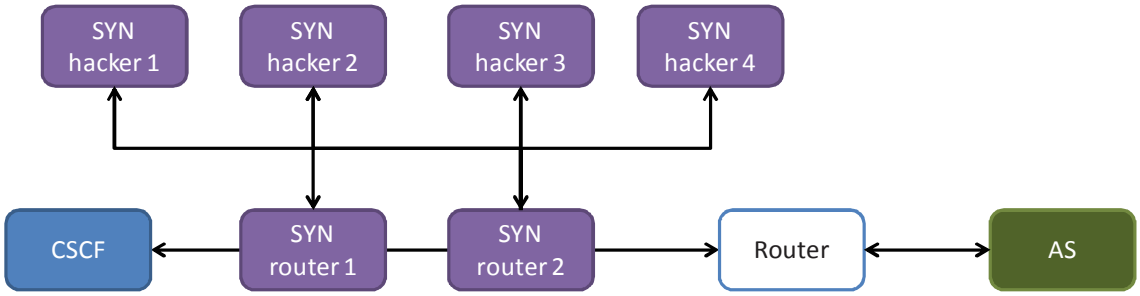

Figure 18 - Multiple SYN routers and SYN hackers

And finally in a real scenario there can be multiple CSCFs and routers as well. Until we are careful with the filtering rules at the SYN routers, and we can assure that all connections has at least one delegated SYN hacker, it is possible to freely combine SYN routers and SYN hackers into the architecture.

5 Testing

Using the implemented proof of concept prototype we made functional and performance test. The aim of the functional test was to demonstrate the right work of the prototype, while in the performance test we investigated the limits of the implementation.

5.1 Functional test scenario

In the functional tests we used three computers with Internet connection. We were not used the IMS architecture, instead we modeled its behavior on a TCP connection. One of the test computers was the TCP initiator and it requested TCP connections to the Internet. The second computer served as a gateway to the Internet. All the connection going outside from this small test network went through on this gateway. In the middle of the connection between the TCP initiator and the gateway we put the TCP hacker. The TCP hacker had console access, there were no IP address assigned to it. The TCP initiator connected directly to the first interface of the TCP hacker, while the TCP hacker connected its second interface to the gateway.

The functional test scenario is displayed in the next figure.

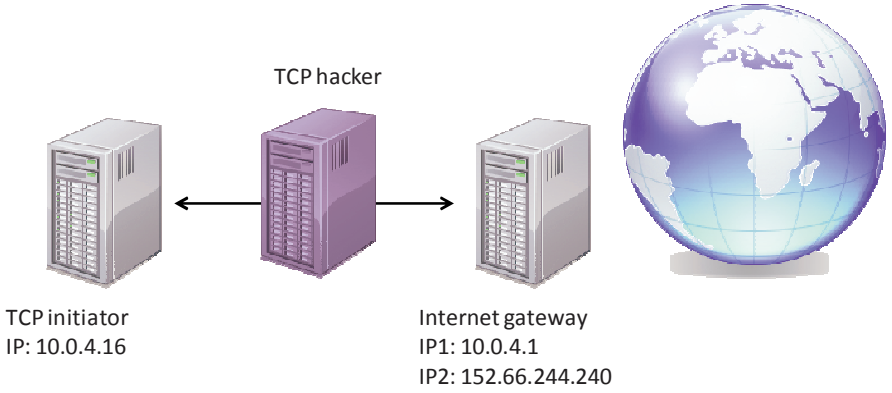

Figure 19 - Functional test scenario

We set up the TCP hacker to bridge the connections on the data link layer between its two interfaces. Using this bridging the TCP initiator gained access to the gateway and beyond, this way no configuration was necessary in the TCP initiator, everything happened the same way as the TCP hacker would not be there.

5.2 Functional test results

In the first test we showed that the TCP hacker does not spoil the TCP connection setups. As in our aspect, the important part is the connection setup only, we used a web browser to initiate connections. The test run normally, the browser was able to display the web pages and the TCP hacker created and deleted the corresponding entries in its connection database successfully. The log files of the run are in the appendix.

In the second test we showed that if the destination is not available, the TCP hacker, suspecting TCP SYN loss, will try to establish the connection by repeating the initial TCP SYN message. After 2 seconds when there is still no answer from the destination, the TCP hacker sends a TCP RST message to the initiator. In the tests we initiated a telnet connection to a non existing IP address from the TCP initiator. Successfully, we could see 4 connection attempts and finally the RST message, just like it was expected. The log files of the run are in the appendix.

During the tests we found no sign of memory leaks, so it looks like that the implementation of the proof of concept prototype was successful.

5.3 Performance tests scenario

During the performance tests we investigated the boundaries of the TCP hacker's performance. In order to maximize the utilization of the TCP hacker, we implemented two additional testing utilities. One of them is called synbomber and the other one is the synbomber-reply.

5.3.1 Synbomber

The synbomber, using a specified interface, sends out a number of TCP SYN connection requests in a burst periodically. The average number of SYN packets in a burst and the average time interval between the bursts can be set. They are both presuming uniform distributions. For the sake of simplicity, the source and the destination of the TCP SYN requests are two different /24 subnets. The number of terminals in the subnets can be set also. At each TCP SYN request the synbomber selects a source and destination terminal and port number randomly using uniform distributions. The synbomber listen on the answers for the TCP SYN requests and keep the connection request active while there is no response for it. Unlike real TCP connection setups, a connection request is never terminated if there is no answer for it. This is good, since we can test, whether all connection setups are handled by the TCP hacker. The synbomber itself never repeats the connection request. When the synbomber get the response for a certain connection request, independently from the answer (it can be SYN ACK or RST) it drops the specific connection. The test is about the connection setup so the rest of the connection's activity is out of scope of the investigation.

In the implementation of the synbomber, we used two separate threads for sending and receiving TCP packets. The sender thread use a raw TCP socket for sending out the packets, while at the receiving thread we used the packet capture (pcap) library. There is also a third thread that gathers statistic information about the successful and unsuccessful connection setups.

5.3.2 Synbomber-reply

The synbomber-reply is the pair of the synbomber. It simulates TCP services that reply for the connection requests. There are no real services behind the addresses and ports, so there will be no further actions when a response is sent. Similarly to the synbomber, the source and destination /24 subnets can be set at the running time. The destination subnet is a filter, when an incoming connection request does not belong to this subnet, then this request is silently dropped. In addition to the IP subnets of the connection addresses, it is possible to define a dropping probability for individual IP addresses. Thus it is possible to set for a certain IP address to do not answer for all incoming connections, or it can be arbitrary dropping probability.

In the implementation of the synbomber-reply, the sending and receiving operations are separate threads in order not to block each other. Here as well we have a statistics thread telling internal information of the working threads.

5.3.3 Scenario

In the performance test scenario we had three computers in a network. There was a synbomber node that initiated tons of connections using the synbomber utility. There was another node running the synbomber-reply utility, which responded for the connection requests. In the middle of the two machines, there was the TCP hacker, helping the synbomber to establish the connections. There was no need for extra gateway computers, the synbomber and synbomber-reply were set up as a gateway for the other machine. The scenario is shown on the following figure:

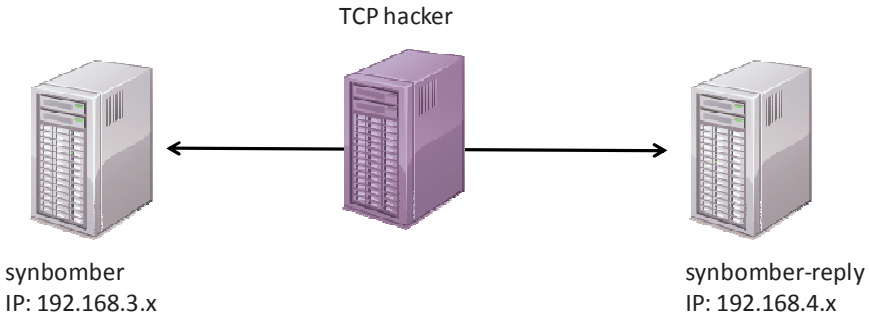

Figure 20 - Performance test scenario

The computers used in this scenario can be considered as general desktops. They had moderate speed two-core AMD processors (AMD Athlon II X2 240e) with 4 GB RAM running a live Linux Debian distribution. The network interfaces were TP-link gigabit PCIe interfaces.

5.4 Performance test results

We performed more performance tests.

5.4.1 Delay measurement for a single connection

In this test we measured the delay caused by the packet forwarding on the TCP hacker. This delay consists of filtering the packet, passing it to user space, creating or deleting entries in the connections database, passing the packet back to the firewall and bridging the packet.

We measured an unloaded TCP hacker with a single connection. This gives us the minimum delay, we assume worse results in the loaded cases. We measured the delay separately for the TCP SYN and TCP SYN ACK packets.

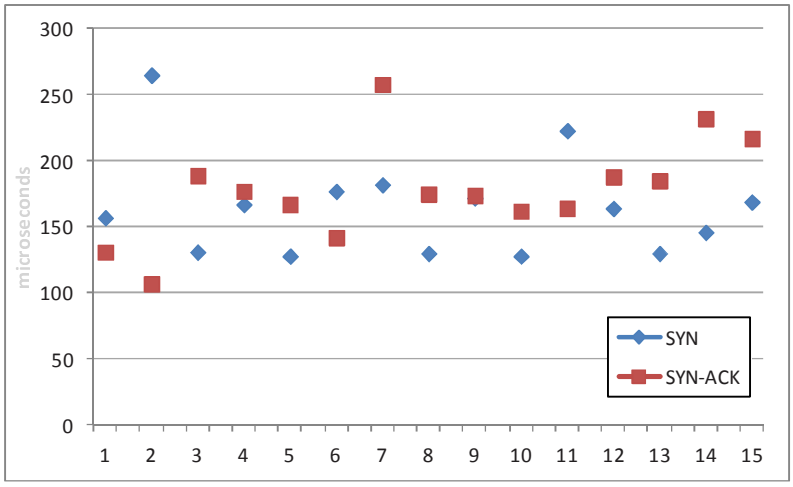

Figure 21 - Unloaded forwarding delay

The measurement results are on the figure above. The most interesting values are the minimum and the median and average. The following table shows these results.

	Minimum [µs]	Median [µs]	Average [µs]
SYN forward	128	164	164.6
SYN-ACK forward	107	175	177.9

The results show that the delay introduced by the TCP hacker is negligible in the unloaded case.

5.4.2 Memory consumption of the TCP hacker

In this test we measured how much memory is required for the operation of the TCP hacker. The TCP hacker stores the connections in its memory using a hexadecimal tree, which depth is 12 entries. In addition to the connections database there is storage for timers and message queues. Considering these consumption factors the heaviest one is the connections database.

Measuring the memory consumption for the timers database and the message queues are straightforward. Each connection in the database takes exactly one entry in the timers database. Each new connection or an acknowledged connection requires 2 entries in the message queue plus one entry in the packet queue. However these message and packet queues are only temporary, as soon as the required action is over, the entry is removed from the queue.

Measuring the memory consumption of the connections database is not as easy as the others, as the number of the new node required to store a new connection entry depends on the content of the database and the actual entry as well. It was easier to measure the consumption on the implementation. For this test we used the synbomber to generate connections, while the synbomber-reply did not run.

This way the connections were unterminated and remained in the database. The statistics thread reported the memory consumption of the connections database.

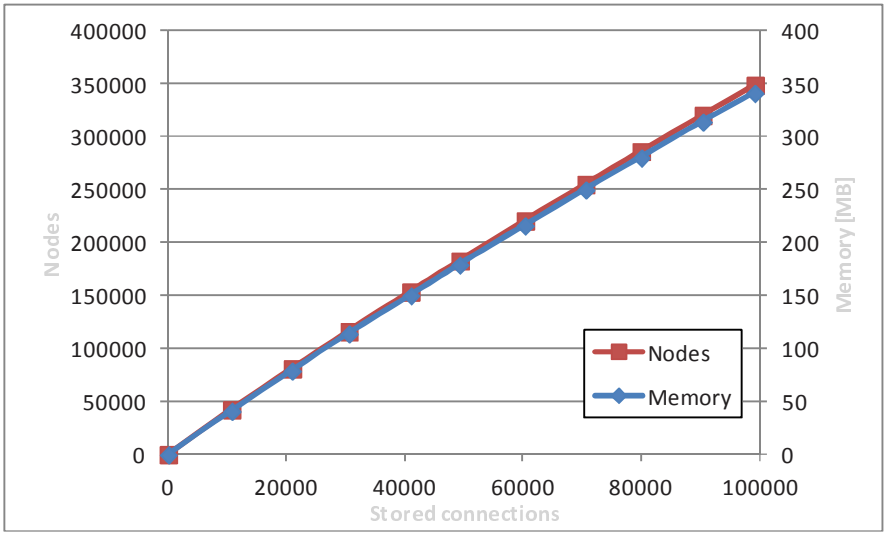

Figure 22 - Memory consumption of stored connections

The results of the measurements are on the figure above. The memory consumption grows together with the node number. This is the expected behavior since they are the new nodes that consume the memory. Each node takes 1028 bytes in the current implementation. The node number is not a linear combination of the connection number, however it is close to it. While the first stored connection requires 12 new nodes, after 10000 stored connection a new one requires only 3.95 in average, and after 100000 connections, this number goes down to 3.51.

The current implementation is not really optimal due to the storage order of the addresses and ports. The current order is the following: source address, source port, destination address, destination port. We can assume that in a real environment the source address, destination address and destination port are coming from a small set. Only the source port has a high variance. For this reason it would be more optimal if the source port would be the last value to store. Considering that the connections are kept in the database for at most 2 seconds, 1 GB memory can stand 150000 new connection per second, which is a fairly high value.

The raw measurement data are in the appendix.

5.4.3 Performance of the loaded TCP hacker

In this measurement we tested the TCP hacker under a loaded condition. We used the synbomber to load the TCP hacker with connections and measured the processor load on the TCP hacker and the delay of the forwarded messages.

We set up the synbomber with the test parameters and measured the TCP hacker process CPU load on the TCP hacker node. We got the following results:

Average SYN in burst	Average interval between bursts [ms]	SYN messages per seconds	CPU load on TCP hacker [%]
10	100	1000	2
25	100	2500	6
50	100	5000	15
100	100	10000	30
5	50	1000	3
20	50	4000	9
50	50	10000	30
70	50	14000	40

Figure 23 - TCP hacker processor load

Unfortunately we could not go further up with the SYN message per seconds value, as around 14000 SYN messages per seconds the Linux firewall user space queue was filled. At this rate the TCP hacker is slower to receive the packets from the queue than the synbomber that fills the queue. Further investigations with the packet capture library showed that that approach has almost the same limit. A possible solution is to run more TCP hackers and this way share the queues among the incoming SYN flow, as it is mentioned in the section about scalability.

Otherwise the process load is not high, suggesting that a general desktop computer can cope with around 30000 SYN request per seconds.

Along with the processor load we also investigated the processing delay of SYN and SYN-ACK messages, similarly to the unloaded case. The results are in the following table:

Average SYN in burst	Average interval between bursts [ms]	SYN messages per seconds	Average processing of SYN [µs]	Average processing of SYN-ACK [µs]
10	100	1000	1011	1367
50	100	5000	7622	10825
5	50	1000	557	610
20	50	4000	2592	3679
30	50	6000	4934	6736

40	50	8000	7908	10273
50	50	10000	9996	12854

As the results show the processing delay depends not only on the number of SYN messages per seconds, but also the distribution of the incoming SYNs. The TCP SYN connection requests arriving in a burst put high load on the TCP hacker. This situation can be observed in the following figures that show the processing time of SYN requests at different burst parameters.

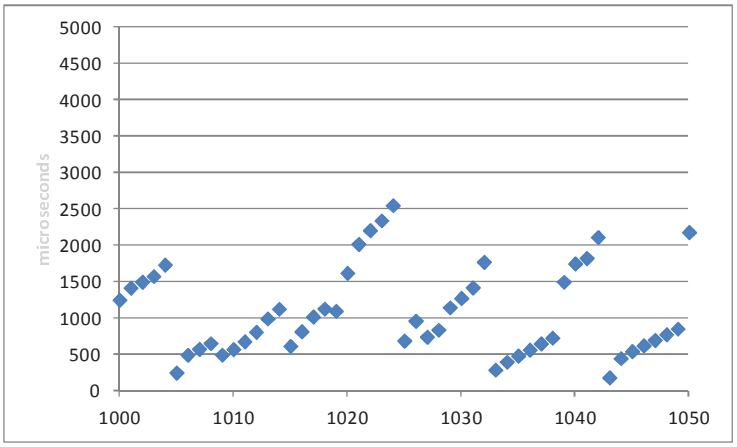

Figure 24 - SYN processing delay @ 10/100

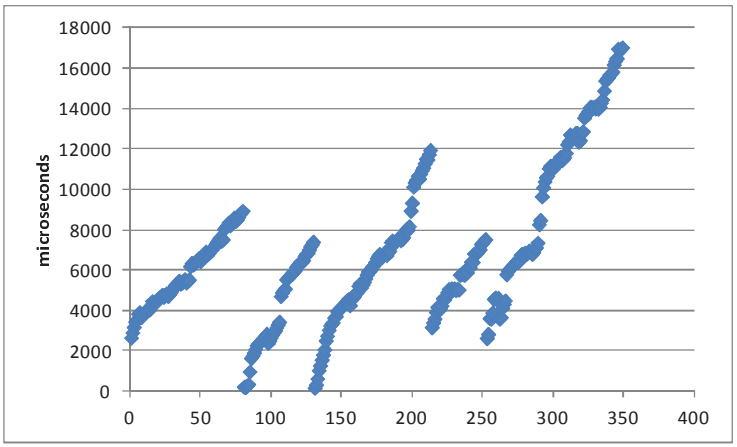

Figure 25 - SYN processing delay 50/100

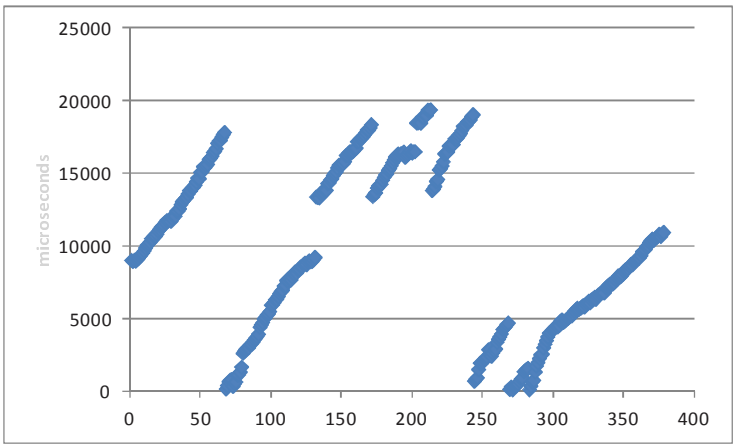

Figure 26 - SYN processing delay 50/50

We can observe that the processing delay increases for the SYN packets in the burst. Actually these packets wait in the message and packet queues to be processed.

After a certain point when the processing of the SYN packets are large enough that a new burst arrives before the TCP hacker could finish with the packets of the previous burst, the queue will fill up and the TCP hacker will stop with an error message.

6 Conclusions

In the IMS environment the CSCF cannot wait for the SIP timers to terminate a broken connection, as this interval is too large for a telephony service. There are more solutions to solve this problem. This document describes two different solutions. One of them utilizes the RSerPool framework, which provides always available service connection. An alternate solution is the TCP hacker that terminates a TCP connection setup after a given time limit, without the help of the remote node.

The TCP hacker solution is further analyzed in this document. There is an implemented prototype, which is analyzed in functional and performance tests. The functional test validates the concept and proves the right behavior of the implementation. The performance tests show that the current implementation, running on a general desktop PC, is able to cope up to 15000 TCP connection setup per seconds. There are different setups advised in this document to increase the scalability and robustness of the current solution.

7 Appendix

7.1 Functional test run log

7.1.1 Successful connection setup

Log file of the TCP hacker:

```
1327753795:654 INFO  [main]: packet received.
1327753795:654 INFO  [syn_callback]: SYN packet received. 10.0.4.16:51419 -> 152.66.115.224:80
1327753795:655 DEBUG [scheduler_thread]: Message code: 1001
1327753795:655 DEBUG [scheduler_thread]: New action to be scheduled.
1327753795:655 DEBUG [scheduler_thread]: Add action 10.0.4.16:51419 -> 152.66.115.224:80
1327753795:655 DEBUG [scheduler_thread]: Next action is at 1327753796:154000000.
1327753795:655 DEBUG [scheduler_thread]: Scheduler is waiting at most 499 ms.
1327753795:728 INFO  [main]: packet received.
1327753795:728 INFO  [synack_callback]: SYN-ACK packet received. 152.66.115.224:80 ->
10.0.4.16:51419
1327753795:728 INFO  [synack_callback]: Connection answered in 73 ms.
1327753795:728 DEBUG [scheduler_thread]: Message code: 1002
1327753795:728 DEBUG [scheduler_thread]: Delete action & node.
1327753795:728 DEBUG [scheduler_thread]: Delete action 10.0.4.16:51419 -> 152.66.115.224:80
1327753795:728 DEBUG [scheduler_thread]: Scheduler is waiting, no timed action.
```

Tcpdump log files on the interface to TCP initiator:

```
12:29:55.654672 IP 10.0.4.16.51419 > 152.66.115.224.80: Flags [S], seq 2435206890, win 5840,
options [mss 1460,sackOK,TS val 394712 ecr 0,nop,wscale 5], length 0
12:29:55.728083 IP 152.66.115.224.80 > 10.0.4.16.51419: Flags [S.], seq 56896001, ack 2435206891,
win 65535, options [mss 1460], length 0
```

Tcpdump log files on the interface to gateway:

```
12:29:55.654928 IP 10.0.4.16.51419 > 152.66.115.224.80: Flags [S], seq 2435206890, win 5840,
options [mss 1460,sackOK,TS val 394712 ecr 0,nop,wscale 5], length 0
12:29:55.727959 IP 152.66.115.224.80 > 10.0.4.16.51419: Flags [S.], seq 56896001, ack 2435206891,
win 65535, options [mss 1460], length 0
```

7.1.2 Unsuccessful connection setup

Log file of the TCP hacker:

```
1327762797:87 INFO  [main]: packet received.
1327762797:87 INFO  [syn_callback]: SYN packet received. 10.0.4.16:45386 -> 152.66.244.244:23
1327762797:88 DEBUG [scheduler_thread]: Message code: 1001
1327762797:88 DEBUG [scheduler_thread]: New action to be scheduled.
1327762797:88 DEBUG [scheduler_thread]: Add action 10.0.4.16:45386 -> 152.66.244.244:23
1327762797:88 DEBUG [scheduler_thread]: Next action is at 1327762797:587000000.
1327762797:88 DEBUG [scheduler_thread]: Scheduler is waiting at most 499 ms.
1327762797:587 INFO  [scheduler_thread]: SYN sending. 2. attempt. 10.0.4.16:45386 ->
152.66.244.244:23
1327762797:587 DEBUG [scheduler_thread]: Next action is at 1327762798:87000000.
1327762797:587 DEBUG [scheduler_thread]: Scheduler is waiting at most 500 ms.
1327762797:587 DEBUG [pktsender_thread]: Message code: 2001
1327762797:587 INFO  [pktsender_thread]: Send SYN packet. 10.0.4.16:45386 -> 152.66.244.244:23.
1327762797:587 DEBUG [pktsender_thread]: Packet sender is waiting.
1327762798:87 INFO  [scheduler_thread]: SYN sending. 3. attempt. 10.0.4.16:45386 ->
152.66.244.244:23
1327762798:87 DEBUG [scheduler_thread]: Next action is at 1327762798:587000000.
1327762798:87 DEBUG [scheduler_thread]: Scheduler is waiting at most 500 ms.
1327762798:87 DEBUG [pktsender_thread]: Message code: 2001
1327762798:87 INFO  [pktsender_thread]: Send SYN packet. 10.0.4.16:45386 -> 152.66.244.244:23.
1327762798:87 DEBUG [pktsender_thread]: Packet sender is waiting.
1327762798:588 INFO  [scheduler_thread]: SYN sending. 4. attempt. 10.0.4.16:45386 ->
152.66.244.244:23
1327762798:588 DEBUG [scheduler_thread]: Next action is at 1327762799:87000000.
1327762798:588 DEBUG [scheduler_thread]: Scheduler is waiting at most 500 ms.
1327762798:588 DEBUG [pktsender_thread]: Message code: 2001
1327762798:588 INFO  [pktsender_thread]: Send SYN packet. 10.0.4.16:45386 -> 152.66.244.244:23.
1327762798:588 DEBUG [pktsender_thread]: Packet sender is waiting.
1327762799:87 INFO  [scheduler_thread]: RST sending. 10.0.4.16:45386 -> 152.66.244.244:23
1327762799:87 DEBUG [scheduler_thread]: Scheduler is waiting, no timed action.
1327762799:87 DEBUG [pktsender_thread]: Message code: 2002
```

```
1327762799:87 INFO  [pktsender_thread]: Send RST packet. 152.66.244.244:23 -> 10.0.4.16:45386.
1327762799:87 DEBUG [pktsender_thread]: Packet sender is waiting.
```

Tcpdump log files on the interface to TCP initiator:

```
14:59:57.087546 IP 10.0.4.16.45386 > 152.66.244.244.23: Flags [S], seq 2835748628, win 5840,
options [mss 1460,sackOK,TS val 2651445 ecr 0,nop,wscale 5], length 0
14:59:59.087159 IP 152.66.244.244.23 > 10.0.4.16.45386: Flags [R.], seq 2835748628, ack
2835748629, win 0, length 0
```

Tcpdump log files on the interface to gateway:

```
14:59:57.087789 IP 10.0.4.16.45386 > 152.66.244.244.23: Flags [S], seq 2835748628, win 5840,
options [mss 1460,sackOK,TS val 2651445 ecr 0,nop,wscale 5], length 0
14:59:57.587236 IP 10.0.4.16.45386 > 152.66.244.244.23: Flags [S], seq 2835748628, win 5840,
options [mss 1460,sackOK,TS val 2651445 ecr 0,nop,wscale 5], length 0
14:59:58.087219 IP 10.0.4.16.45386 > 152.66.244.244.23: Flags [S], seq 2835748628, win 5840,
options [mss 1460,sackOK,TS val 2651445 ecr 0,nop,wscale 5], length 0
14:59:58.588127 IP 10.0.4.16.45386 > 152.66.244.244.23: Flags [S], seq 2835748628, win 5840,
options [mss 1460,sackOK,TS val 2651445 ecr 0,nop,wscale 5], length 0
```

7.2 Memory consumption

```
1327783453:163 DATA  [statistics_thread]: Connections stat: FGTree: addresses: 0 mem: 0 MB,
nodes: 0
1327783463:164 DATA  [statistics_thread]: Connections stat: FGTree: addresses: 10730 mem: 41 MB,
nodes: 42385
1327783473:166 DATA  [statistics_thread]: Connections stat: FGTree: addresses: 20951 mem: 79 MB,
nodes: 81046
1327783483:166 DATA  [statistics_thread]: Connections stat: FGTree: addresses: 30543 mem: 114 MB,
nodes: 116307
1327783493:167 DATA  [statistics_thread]: Connections stat: FGTree: addresses: 41004 mem: 150 MB,
nodes: 153710
1327783503:167 DATA  [statistics_thread]: Connections stat: FGTree: addresses: 49298 mem: 179 MB,
nodes: 182810
1327783513:168 DATA  [statistics_thread]: Connections stat: FGTree: addresses: 60333 mem: 216 MB,
nodes: 220783
1327783523:168 DATA  [statistics_thread]: Connections stat: FGTree: addresses: 70579 mem: 250 MB,
nodes: 255223
1327783533:168 DATA  [statistics_thread]: Connections stat: FGTree: addresses: 79946 mem: 280 MB,
nodes: 286234
1327783543:168 DATA  [statistics_thread]: Connections stat: FGTree: addresses: 90399 mem: 314 MB,
nodes: 320321
1327783553:169 DATA  [statistics_thread]: Connections stat: FGTree: addresses: 99181 mem: 341 MB,
nodes: 348715
```