HTTPreject: Handling Overload Situations without Losing the Contact to the User

Joerg Schneider, Sebastian Koch
Department of Electrical Engineering and Computer Sciences
Technische Universitaet Berlin
Berlin, Germany
{*komm,seb*}*@cs.tu-berlin.de*

***Abstract*—The web is a crucial source of information nowadays. At the same time, web applications become more and more complex. Therefore, a spontaneous increase in the number of visitors, e.g., based on news reports or events, easily brings a web server in an overload situation. In contrast to the classical model of distributed denial of service (DDoS) attacks, such a so-called flash effect situation is not triggered by a bulk of bots just aiming at hurting the system but by humans with a high interest in the content of the web site itself. While the bots do not stop their attack until told so by their operator, the user try repeatedly to access the site without knowing that the repeated reloads effectively increase the web server's overload. Classical approaches try to distinguish between real user and harmful requests, which is not applicable in this scenario. Simply restricting the number of connections leads to very technical error messages displayed by the users' client software if at all. Therefore, we propose a mean to efficiently block connection attempts and to keep the user informed at the same time. A small subset of HTTP and TCP is statelessly implemented to display simple busy messages or relevant news updates to the end user with only few resources. In this paper we present the protocol subset used and discuss the compatibility problems on the protocol and client software level. Furthermore, we show the results of performance experiments using a prototype implementation.**

I. Introduction

In the beginning of 2010, there was a huge ash cloud over Europe. Due to the possible harm to aircrafts, the air traffic in Europe was shut down for more than a week. Unfortunately, the prediction of the ash cloud movement was only valid for a few hours in the future. Therefore, airline operators and clients had to reschedule their plans multiple times a day. Many airline and airport websites were unable to cope with the storm of visitors requesting information on the status of the ash cloud. Unfortunately, most users behaved like a bot net node during a distributed denial of service attack: If they couldn't get the whole website they just send another request by reloading. However, most users were unaware that they effectively increased the overload situation by their behavior.

In this paper we propose an efficient way to block HTTP requests on the IP layer while still keeping the user informed. We use a simplified HTTP and TCP protocol to statelessly block requests by answering with a short status message. We base our idea on the assumption that the user would have behaved differently in the example above if they had received a simple but meaningful status update, i.e., in the example whether the airport is open or the airline is operating in Europe. However, in this paper we present only the protocol simplification and discuss the technical impact of the simplification. In future work we will analyze in detail whether the assumption on the user behavior holds for the given scenario and other.

In the following, we start with a clearer definition of the application scenarios of our approach. After discussing existing approaches, we present our approach to statelessly block HTTP requests. Then we discuss the impact of the simplifications on the protocol and user client compatibility. We show the results of a performance study done with a prototype implementation, before concluding with the open questions to be handled in future work.

II. Application Domain: Human initiated overload situations

Before discussing the details of our approach, we want to differentiate its application domain from the classical distributed denial of service (DDoS) attacks. A DDoS attack is usually performed by a set of coordinately acting computers trying to request the service as often as possible but without using the results sent back [1]. There are some specific subclasses, e.g., SYN floods where the bots do not use the higher level service but try to allocate resources by sending just the initial TCP handshake packet. The attacker's aim is to block as much resources as possible, such that legitimate users cannot access the service anymore or only with restrictions. Therefore, the general countermeasure is to distinguish between the legitimate requests and the attackers requests.

In our scenario, all requests are made by legitimate users. Some exemplary situations of flash effects:

- A transport system has major disturbances, e.g., the volcano ash stopping Europe's air traffic. Here, many travelers need updated information.
- Some news of importance to many people are published like articles covering a major accident or a political

vote. In this case, a number of people are interested in detailed information and news updates.
- A smaller website is linked from a high traffic news page. This example is a bit out of line regarding the other two examples. However, this so called *slashdot effect* [2] happens quite often leading to overrun websites.
- A TV show has a accompanying website with further information or the option to win a prize. After or even during the broadcast of the TV show the interest in the website drastically increases [3].

All situations have in common that the number of users drastically rises quickly and keeps high for a while. However, as the resources are restricted and cannot be extended on short notice, only few users' requests can be handled. But as all requests come from real persons seeking information, there are no illegitimate users to detect and block.

Nonetheless, one could simply allow a certain number of requests and block further requests of regular users as it is done with bots. The blocked user will eventually get a cryptic error message from their client software. For example, dropping the connection initiation packet leads to an uncertain state on the client side. Usually, a "loading…" animation is shown until a timeout is reached. Then, most browsers ask the user whether there is a problem with the Internet connection. As the user knows that there is no problem on his side, he just retries to reach the website. This is especially true, if the user desperately needs some information on the website as explained above. Summing up, the behavior of a user is similar to a bot in a DDoS case. However, we expect the user to stop retrying if he gets a more personalized and specific error message or a quick summary of the information needed.

III. Related Work

Distributed denial of service (DDoS) attacks are widely discussed in the literature [1], [4]. However, as discussed before, we focus more on flash effects - overload situations created by an instant increase of regular visitors. Jung et al. discussed in [3] two example situations in detail—a TV show website and a website covering official election results. They showed the different behavior of flash effects and automated DDoS attacks. Furthermore, they discuss techniques for web server operator to cope with flash effects - mainly how to drop a part of the connection attempts and how to use content distribution networks. More recently, the properties of different content distribution network and Cloud solutions were discussed in [2]. As discussed by the authors, a Cloud based webserver with dynamic capacities has to be prepared well in advance. Especially, if complex web applications with multiple tiers have to be distributed in the Cloud. Furthermore, the additional capacity in the Cloud is not infinite and can become quite cost intensive.

A formal model of flash effects was developed in [5] and then used to show the gain of additional hierarchical proxy servers. Such proxy server or content distribution networks can also base on a peer-to-peer architecture as discussed in [6] and [7]. However, either the user has to enable the proxy or the website operator has to publish its information using a content distribution network to benefit from the worldwide peer-to-peer based load balancing. Furthermore, such a caching schema is only applicable for static content.

Jamjoom and Shin developed another formal model which also takes into account that a single website request leads to multiple HTTP requests, e.g., for embedded images [8]. Therefore, they propose not to drop random connection request but to implement an all-or-nothing strategy for the connections of a client. They also analyzed how the different operating systems behave if the connection attempt is not answered. As it is also our assumption, they argue that user tend to retry if only a technical error message is given after a timeout.

An example for deciding in advance which connection to route to the server was developed by Yaar et al. in [9]. They extended the TCP header with a capability field and the router on route to the server could already decide to drop packets without a capability or a revoked capability. This concept realized a prioritization between the connections and thus allowed to provide the service at least for a small subset of the clients. Another approach is described by Chen and Heidemann to adaptively adjust the admission control, i.e., which packets to drop and which to forward [10]. They showed that even in a heavily attacked system the timing of the successful connection is not much disturbed. However, up to 32% of the connections have been dropped to protect the other connections. These dropped connection lead to uncertain states on the client side. In our approach the idea is not to drop the connection attempt, but to answer it with a short but meaningful message. In order to give access to the user with the highest use, Golze et al. propose a schema based on proof-of-work functions, i.e., the client has to solve a challenge to show its interest in the service [11].

There have been also proposals to extend the operating system kernel to handle flash effects. A very effective form of distributed denial of service attacks are SYN floods. Here the attacker sends only connection initiation packets. The server then set up a context for the each of these new connections and waits for further packets. After a while a lot of data sets for these incomplete connections fill the memory and the server cannot accept new connections. A kernel level solution is the SYN cookies concept [4]. It eliminates the need to set up a connection context for each SYN packet as it encodes the most important information in the TCP sequence number of the SYN/ACK packet which will be sent back by the client in the final ACK packet. Nonetheless after the successful handshake, a connection context will be used to keep track of the TCP connection. SYN cookies need no

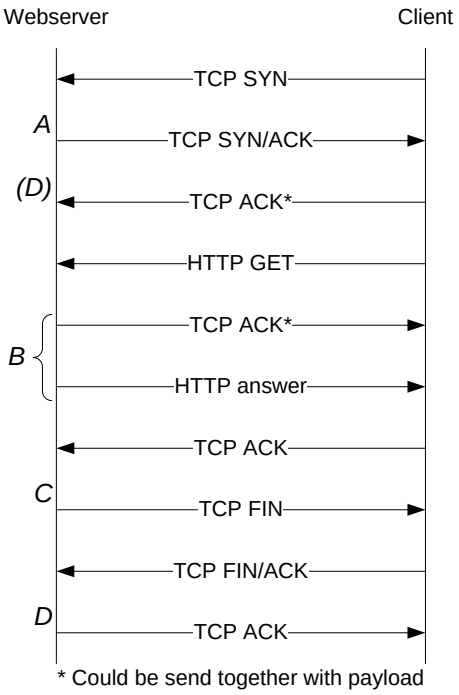

Figure 1. The packets send for a simple HTTP GET request where request and answer fit in a single IP packet each. The letters next to the web server time scale denote the cases in the flowchart in Figure 2.

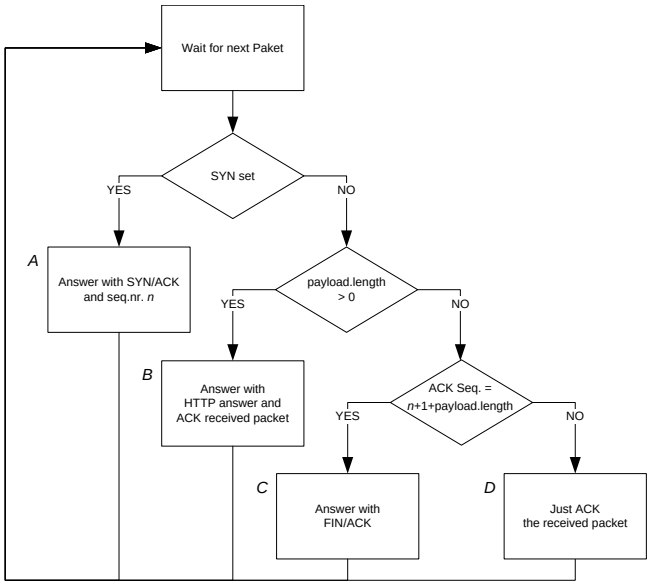

Figure 2. With some simplifications, no connection state is needed on the server side. Each packet can be answered without knowing the history of the connection.

state information on the server side during the handshake, but also allow only a small subset of the TCP features. Lenon compared SYN cookies with a hashtable based approach to keep the connection contexts [12]. The general idea of SYN cookies to maintain a connection with the client without keeping state information is extended in our approach to a full TCP connection transporting an HTTP request/response pair.

Adding HTTP functionality to the operating system kernel is also not new. For the Linux kernel there are for example two extensions: the TUX web server[1] and khttpd[2]. Both are implemented within the kernel and can serve static web pages from the file system. They provide nearly all features of a regular web server and can forward the request to a userland web server for example to provide dynamic webpages. Although both are slightly faster than userland web servers, they need to keep state information for each client. Our approach is not to provide a fully functional web server but a simple means to transfer a short piece of information using a simplified, stateless protocol.

IV. The Stateless Web Server

Figure 1 depicts an ideal connection to retrieve a website with HTTP. Such a sequence needs ten IP packets. Some operating systems may combine the first two ACK packets with the following payload packet, but more simplifications are not possible. On the other hand, retransmits and payload data exceeding the IP packet size may require additional packets to be sent.

However, as only a small message will be sent we mimic this ideal connection without keeping the state of the connection. Connection state within TCP is needed to keep track of the sequence numbers in order to see if all packets send are acknowledged by the partner or to detect missing packets from the partner. Furthermore, the TCP stack has to know in which state the connection is, e.g., waiting for the connect handshake, established, or waiting for the termination handshake. HTTP is considered a stateless protocol, i.e., an HTTP request can always be answered without knowing previous requests of the client. Newer versions of the protocol provide means to exchange multiple request/answer pairs in a single TCP connection. However, this behavior is not required, so we do not need to support it here.

To statelessly implement this ideal connection, the server answers only if asked. Therefore, the proposed implementation does not support any timeouts or retransmits if not requested by the client. Based on this simplification, we can now analyze which packet to send back for each received packet.

The connection is always initiated by the client by a SYN packet. This packet will be answered with a SYN/ACK packet (Case A in Figure 2). As it is the first step, it needs no previously stored information. However, normal TCP implementations set up a context and save the sequence numbers together with other parameters. The sequence numbers will

[1] http://people.redhat.com/~mingo/TUX-patches/
[2] http://www.fenrus.demon.nl/

be used in the following step to check whether the next ACK matches or some packet is lost. We use a special sequence number n for all initial packets, so no context is needed here.

The next packet received from the client may be a single ACK packet, which changes the connection state, but can be ignored as we do not keep track of the connection state. To simplify the algorithm the packet will be handled by the default case as explained later (Case D in Figure 2).

In the next step the client send its HTTP GET request. In the ideal case the request fits in a single TCP/IP packet and will be answered with an HTTP response containing the short but informative message (Case B in Figure 2). Additionally, the arrival of the packet will be acknowledged. Normal TCP implementations would need to check the sequence number and to verify and set them accordingly to their state. We assume, everything sent by the client is an HTTP GET request. Thus, we do not parse the actual content of the packet and ignore the clients sequence numbers, too. As we used the fixed sequence number n in the first step and only send a single packet, we use $n + 1$ as the sequence number of this packet. Therefore, even if for some reason this packet is triggered multiple times, it looks like a retransmit for the client.

When the client acknowledges our HTTP response packet, we close the connection with a FIN packet (Case C in Figure 2). The initial sequence number is fixed and the length of the messageis known, too. Therefore, we know when the client acknowledges receiving the whole message.

The connection termination is again a three way handshake. So the client acknowledges the FIN packet with a FIN/ACK which in turn will be acknowledged with an ACK packet (Case D in Figure 2). This is the default rule, if none of the cases described before match. In this case the received data are acknowledged and no other action is done.

As the summary in Figure 2 shows, every packet send by the client can be answered without knowing the actual state of the connection.

In the following section we are going to discuss the implications of the simplifications and in Section VI preliminary results of experiments with a prototype implementation are shown.

V. Can it work?

The aim was to block HTTP requests with a short but hopefully informative message. In this section we are going to discuss the main questions arising around our approach.

Does it work with any browser/operating system?: There are simplifications within the HTTP protocol as well as within the TCP protocol. On the HTTP layer, the actual request type, e.g., GET, HEAD, or POST, is ignored. Because the answer is always the same, the requested resource is ignored, too. Fortunately, HTTP responses have all the same structure. Thus the browser doesn't recognize that it doesn't get the answer to the question asked.

On the TCP level there are more simplifications. If the client sends more than one payload packet it will receive multiple times the same packet with the same sequence number. The client can interpret it as a retransmit of the first packet. However, retransmitting the packet before the timeout violates the protocol. As the server has no connection state, it doesn't realize that a packet was not acknowledged by the client and needs to be retransmitted. But the payload packet sent is also the acknowledgement of a client packet. Thus, the client will retransmit its packet if the server answer is lost. This new packet will trigger the same rule on the server side again. Other TCP features like the flow control are ignored, too. However, for such a short connection it may be acceptable.

We tested the prototype with several browsers running on Linux, Windows, and Solaris. We could see major differences in the TCP implementation of the operating systems. Nonetheless, our prototype was able to successfully send the short message to the browser.

Will the overload message end up in proxies and search engines?: This heavily depends on the HTTP return code used. Using an error code like "503 Service Unavailable" will indicate for automated systems like search engine bots or proxies that the response is not the actual website. However, some browser display their own error message instead of the HTTP response in this case. Therefore, in order to reach all users we set the return code to "200 OK" risking that the overload message is cached. At least for proxies, the caching behavior can be influenced using additional HTTP header.

Does an HTTP GET request fit in a single packet?: It mainly depends on the browser and also the number of installed extensions, e.g., due to the number of supported MIME types or special headers. But even if the browser sends the request in more than one packet it gets the same answer multiple times back. The client operating system should use only one of these packets as they carry the same sequence number. In our experiments only the Internet Explorer on Windows Vista needed two packets. A broader study on the behavior of different browser/operating system combinations is planned as future work.

How does the system behave in a distributed denial of service attack?: If the attacking bots do not follow the protocol any further, e.g., in a SYN flood attack, no state needs to be cleaned. The behavior is similiar to the SYN cookie approach [4]. Otherwise if the malicious bot completes the full TCP and HTTP protocol, the web server serves efficiently the short message. Therefore, the system is not much impacted by a distributed denial of service attack.

Using a predictable sequence number allows TCP session hijacking.: Due to the simplifications an attacker can easily inject packets into the connection. In future work

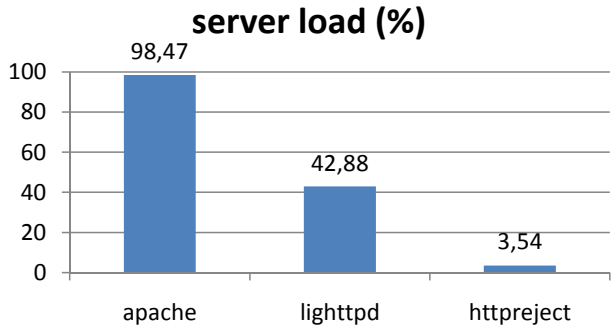

Figure 3. Our httpreject prototype consumes much less CPU time than the userland web servers.

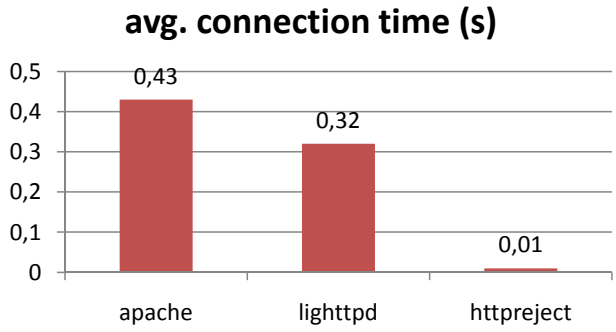

Figure 4. Due to the stateless design, the proposed blocking schema terminates the connections much faster than the userland web servers.

we plan to analyze the actual thread of this attack in this restricted scenario.

There may be more side effects of the simplifications, we didn't observe in the experiments with the prototype. Therefore, we plan together with a usability study also a broader technical study on the compatibility.

VI. Performance Evaluation

The general idea is to answer with a short but informative message. One could argue that it would be enough to replace all the complex, dynamic web pages with a simple static website. Therefore, we analyzed in an experiment how much more connections our stateless blocking mechanism can handle compared to a classical userland web server serving static pages.

For this perfomance evaluation, we do not use our basic assumption of the user stopping to access the website after getting the error message. We simulate rather a storm of uncooperative user or bots accessing the website at a high rate. Furthermore, all user will get the error message while in a productive deployment some user will still get a connection to the real web application.

The concept was implemented as a netfilter[3] module within the Linux kernel. Therefore, it can be activated dynamically as part of the Linux firewall to block statelessly connection atempts.

A. Experimental setup

To determine the performance of the httpreject module in comparison to two userland web servers, we measure the impact of a given rate of parallel HTTP connections on the load of the server host system. We could also measure the maximum connection rate our prototype or the userland web server can handle. However, it is very complex to reliably generate such a high number of connection at a constant rate. The measured result would be very vague. Therefore, we used the CPU load generated by a constant rate of connections assuming that a lower loaded system will handle a higher maximum rate, too.

We simplified the test setup and used only two systems. One plays the server role and the other simulates a storm of user requests. Both systems were connected via a direct gigabit Ethernet link. On the server side we used only a quadcore machine to induce the overload situation, while we used a 16 core machine to generate traffic. Otherwise the machines are equal: equipped with Intel Xeon processors running at 2.4 GHz, 48 GB memory and running a 2.6.32 Linux kernel.

On the server side, we used our prototype implementation as a kernel module, an Apache 2.2[4] web server and a lighttpd 1.4[5] web server. Both userland web servers served only a single static webpage which was equivalent to the short message send by our prototype.

On the client side, we implemented a fast client simulation which used 16 processes. Each process used an own IP address and was able to handle up to 1024 simultaneous connections. Each process issued an HTTP request at a fixed rate independently whether previous requests were already completed or not. This setup allowed us to generate $20,800$ connections per second and 1.2 million connections per experiment.

The kernel on both systems were tweaked to handle so many connections, e.g., by allowing the reuse of waiting sockets and extending the backlog queue. The experiments were repeated until the 95% confidence interval was smaller than 20% of the measured value.

B. Results

The results depicted in Figure 3 show that our proposed stateless blocking mechanism consumes much less CPU time than the userland web server. In contrast to the userland web servers, in our proposed schema no state information needs to be saved or retrieved for each packet. In the userland

[3]http://www.netfilter.org/

[4]http://httpd.apache.org/

[5]http://www.lighttpd.net/

web servers, a TCP connection has to be handled stateful and each HTTP request will also get a context within the web server even if the same small message is served to all clients.

The benefit of this stateless implementation is also visible in Figure 4. Here the average duration of a connection is depicted showing that the userland web servers need 30 times more time in average to answer the requests.

The experiments showed that even if the user do not stop to try accessing the website, our proposed blocking schema is able to handle more connections than a classical userland webserver. The effect of users waiting before retrying will add up to these results.

VII. Conclusion and Outlook

In this paper, we proposed an efficient way to block HTTP connections while at the same time keeping the user informed. Instead of just dropping or rejecting connection attemps in spontaneous overload situations (flash effects), we use a simplified HTTP and TCP protocol to send a short, but informative message to the client. We assume, that a user then does not retry the connection attemp as it is very likely if the connection is just terminated with a technical error message.

We showed that an ideal short HTTP request requires ten packets to be send between client and server. Based on this ideal connection, we developed a schema how the web server can answer every packet of the client without keeping state information. Despite the simplifications of the protocol, we showed the compatibility with a number of systems and discussed the general implications of the simplifications. In a performance study, we compared the prototype implementation within the Linux kernel with two userland web servers serving a single static website. The userland web servers needed at least 10 times more processor time to handle the same number of connections. The concept was implemented as a netfilter module within the Linux kernel. Therefore, it can be activated dynamically during the runtime of the webserver as part of the Linux firewall.

One of the main assumptions behind this paper is that user will not retry to get the actual website after getting an informative message. In future work, we plan a user study to see how the user behave if randomly connections are dropped or if our approach is used. Furthermore, we plan to analyze more thorough the effects of the TCP protocol simplifications.

References

[1] F. Lau and S. H. Rubin, "Distributed denial of service attacks," in *In IEEE International Conference on Systems, Man, and Cybernetics*, 2000, pp. 2275–2280.

[2] J. Elson and J. Howell, "Handling flash crowds from your garage," in *Proceedings of USENIX 2008 Annual Technical Conference on Annual Technical Conference (ATC'08)*. Berkeley, CA, USA: USENIX Association, 2008, pp. 171–184.

[3] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: characterization and implications for cdns and web sites," in *Proceedings of the 11th international conference on World Wide Web (WWW '02)*. New York, NY, USA: ACM, 2002, pp. 293–304.

[4] W. Eddy, "Tcp syn flooding attacks and common mitigations," RFC 4987, 2007.

[5] I. Ari, B. Hong, E. Miller, S. Brandt, and D. Long, "Managing flash crowds on the internet," in *Proceedings of 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS 2003)*, 12-15 2003, pp. 246 – 249.

[6] T. Stading, P. Maniatis, and M. Baker, "Peer-to-peer caching schemes to address flash crowds," in *Peer-to-Peer Systems*, ser. Lecture Notes in Computer Science, P. Druschel, F. Kaashoek, and A. Rowstron, Eds., vol. 2429. Springer Berlin / Heidelberg, 2002, pp. 203–213. [Online]. Available: http://dx.doi.org/10.1007/3-540-45748-8_19

[7] M. J. Freedman, "Experiences with coralcdn: A five-year operational view," in *Proceedings of 7th USENIX/ACM Symposium on Networked Systems Design and Implementation*, 2010.

[8] H. Jamjoom and K. G. Shin, "Persistent dropping: an efficient control of traffic aggregates," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '03)*. New York, NY, USA: ACM, 2003, pp. 287–298.

[9] A. Yaar, A. Perrig, and D. Song, "Siff: a stateless internet flow filter to mitigate ddos flooding attacks," in *Proceedings of 2004 IEEE Symposium on Security and Privacy*, 9-12 2004, pp. 130 – 143.

[10] X. Chen and J. Heidemann, "Flash crowd mitigation via adaptive admission control based on application-level observations," *ACM Trans. Internet Technol.*, vol. 5, no. 3, pp. 532–569, 2005.

[11] S. Golze and G. Muhl, "Fair overload handling using proof-of-work functions," in *Proceedings of International Symposium on Applications and the Internet (SAINT 2006).*, 23-27 2006, pp. 8 pp. –21.

[12] J. Lemon, "Resisting syn flood dos attacks with a syn cache," in *Proceedings of the USENIX BSDCon 2002 Conference*, 2002.