Protection system against Overload and Distributed Denial of Service Attacks

Ervin Tóth
SEARCH-LAB
ervin.toth@search-lab.hu

Zoltán Hornák
SEARCH-LAB
zoltan.hornak@search-lab.hu

Gergely Tóth
SEARCH-LAB
gergely.toth@search-lab.hu

Abstract

Detection of, and protection from overload and Denial of Service attacks is a common problem in information system servers. Such situation may be the result of simple overload, such as increased service request rate during peak hours, or a malicious distributed attack originating from many computers. This article describes a solution to this problem, focusing on protection mechanisms against both natural and malicious overload, based on analyzing the unlimited queue of requests. Furthermore, detection and protection scheme against blocking and crash-bug exploiting attacks is presented. The described algorithms were implemented within the Dependability and Security by Enhanced Reconfigurability (DESEREC) EU FP6 integrated project.

1. Introduction

The information society relies on the dependability of IT systems, such as electronic banking, aviation, web, e-mail, etc. Service outage, e.g. caused by an attack could imply huge damage even on domestic scale. Overloading of information systems or servers may result in Denial of Service or in unacceptable response times.

Overload can occur basically because of the following reasons:

- *Natural overload* during the normal operation of the system, caused e.g. by peak hours (New Years Eve), spam, congestion in networks etc. In this case, overload can be considered as a result of improper design of the system from performance point of view (insufficient resources), or the lack of a proper rejection/fallback strategy.
- *Result of a malicious attack* (referred to as Denial of Service or DoS attack). News about synchronized attacks against servers of leading information technology companies show that DoS attacks can easily paralyze Internet services for several hours. Protection against DoS attacks is very hard if an attacker starts the action by penetrating into many weakly protected computers all around the world, use them as zombie systems and execute a synchronized attack against the target (as happened in many cases [1], [2], [3]).

This paper focuses on the detection of, and protection from malicious Denial of Service. We will present a novel system, which can detect and protect against the three main types of DoS.

2. Types of Denial of Service

The main types of DoS are the following:

- *Blocking*: a malicious attacker can very easily block the operation by exploiting the fixed limits (e.g. number of threads) in the server. However this case can be explained as an overload, but detection of this type is significantly different from other overload situations. For example, a number of slow clients may cause blocking by occupying the available fixed number of server threads.

- *Flood*: an attack aims to overload the server by generating an unbearable load. In this case the overload can be much bigger than a natural overload can cause, e.g. a great number of semi-simultaneous requests may cause flooding.
- *Overload*: the server gets more requests than it can handle, but it has enough capacity to select and reject unwanted load and serve the remaining ones. (Optionally rejected requests may get a polite rejection message.) This situation happens if the server has lower computational capacity than the requests would require.

2. Performance Considerations

It is very important to clearly understand what can be the goal of an overload protection subsystem. From the client's point of view two major factors are important: maximum response time and rejected request ratio.

2.1. Response Time, Rejected Request Ratio

From the server's point of view the aim is to stay alive in any cases (avoid final Denial of Service), utilize system resources most effectively to maintain high performance: especially CPU utilization should be 100% in case of overload, while blocking should be avoided.

To control the above performance factors there are basically two performance controlling values: the length of the queue and the number of server threads. However, it will be shown that performance considerations in some sense contradict security considerations.

From the user's point of view the response time and the rejected request ratio are the most important factors. If an adequate overload protection is not applied, response times may increase above a limit, where users won't wait for the response and therefore this request can be considered lost and, on the other hand, server resources were consumed uselessly. This negative feedback can drastically degrade the performance of the server: response times continue to increase and the ratio of lost requests will increase as well, finally resulting in a virtual Denial of Service.

2.2. Performance and Utilization

Since a server is a complex system, its modeling is not straightforward. In parallel execution threads can be overlapped without remarkably affecting the execution of each other, but if too many threads are started together, the overall performance will not increase. This thread limit depends on very many factors (request type, hardware configuration, network bandwidth, speed of disk I/O, etc.). It is very hard to determine it in practice, because its value may greatly vary during operation.

That's why another approach is proposed for controlling the performance of the server. Namely, if the utilization of the server resources is the highest possible, the performance of the server should be the highest possible as well. In other terms, the utilization is the highest if no resources are wasted when they could be utilized.

To enable the simple modeling of server systems it is very important to hide the internal structure of a server; it can be considered as a black box and only one performance-measuring factor needed to describe this system: the service rate. This service rate gives how many requests leave the system in one time unit in average. One should also understand that the service rate is independent from the response time and the servicing time.

3. Security Considerations

The limitation of server threads according to a fixed value can lead to the possibility of blocking attacks. Note that if this thread limit is increased to any fixed value an attacker always can use this technique to block the actual server. To avoid blocking, the server should not have (small) fixed limitations on any resources, otherwise a malicious attacker may issue some requests which consume these small amount of resources, hold them allocated and this way block the service of other requests.

The fixed limitation of the length of the queue can prevent the DoS protection subsystem to reject malicious requests (black list) effectively and admit correct requests (white list), since if one determines the maximal length of the queue based on performance considerations then, even if malicious requests can be spotted too many correct requests will be dropped at the end of the queue, so an attacker can reach his goal to deny many good requests and thus virtually block the server.

On the other hand if the allocation of a resource is not limited (either the length of queue, or the number of threads) a malicious flood or even a natural overload can lead to unlimitedly growing response times. This is unacceptable and practically means a Denial of Service from the clients' point of view.

At first sight there is a contradiction here between performance and security considerations. The DoS protection subsystem needs a long run from a flood to recognize the malicious requests more precisely and preferably no requests should be dropped during it, while performance considerations call for a high request rejection rate to maintain response times. Our proposal solves this problem by letting the queue grow and rejecting requests at the beginning of the queue.

4. Traditional Solutions Against Denial of Service

Protection of servers from overloading is thought to be one of the most important tasks of server developers. Existing solutions for http servers include bandwidth and resource throttling, which provides access to certain resources for certain clients/applications. Another option is the limitation of the number of requests, which protects the server from overloading, but the threshold depends on the server hardware configuration and the resource needs of other processes. Keep alive timeout allows more requests to be served in one connection.

Traditional load balancing architecture of a network server can be seen on Figure 1.

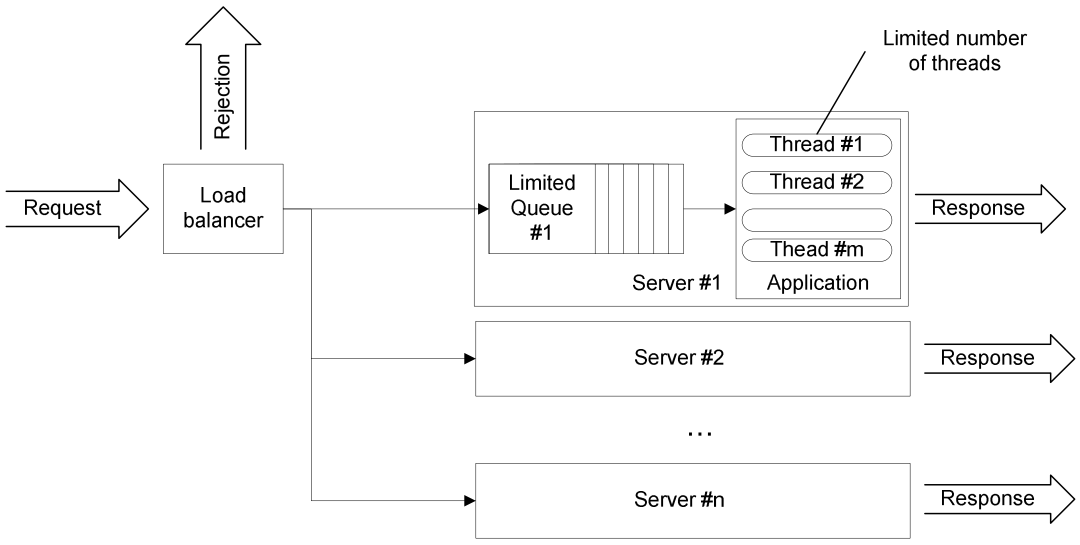

Figure 1. Traditional load balancing

Several issues arise with the traditional methods, e.g. fixed limits make the services inflexible. Since the balancer evenly distributes the load, in case of an overload, each server gets overloaded. The biggest problem is, however, that it employs mainly preventive solutions, but generally lacks the detective and corrective methods.

5. Concept of the Proposed Protection

Since all servers have their own physical limitations, the protection scheme must not allow more requests to be handled and more resource allocations to be carried out than the server is reasonably capable of. From queuing theory [4] and from everyday experiences one may learn that in case of a server with limited capacity the most effective solution to utilize its resources without significant waste is the application of a queue. In case of a queuing system the question is whom to allow entering the queue and when to admit the next request from the queue to the server.

Due to the advantages of multi-threading a computer could serve more than one request simultaneously, which increased the service ratio of the system. However, beyond a limit there was no reason to admit new requests to the server, since the overall performance did not increase, but consumed more and more resources. To adapt best to this behavior, first it had to be solved that only those requests got admitted to the server that can increase the server's performance. However, instead of "performance increase" the approach of "avoiding waste" was used, which means that if some kind of resource would be wasted if new a request would not be admitted to the server, then one is admitted. Otherwise there is no reason to admit a request to the server. In practice this means that if a processor is in an idle state and the server has the available resources to serve, a new request should be admitted.

Queuing Theory assumes that requests should be dropped if the queue is full and therefore tries to determine the queue length. Turning this thinking upside down, dropping the requests when they leave the queue and directly limit their response times. Simply the time when a request arrives into the waiting queue is recorded, and when the request admission control tries to admit it into the server, the waiting time of every request can be simply and exactly calculated. Then the expected response time of this request can almost precisely be estimated by the waiting time plus the estimated service time. Since in case of overload the waiting time is many times larger than estimated service time, this estimation will be a pretty good one, thus it can be simply decided if this estimated response time is acceptable or not. If it is below a certain limit, the request is admitted; if there is no hope that the request will be served in time, it is refused. This solution is in line with usual expectations. In practice after some seconds if there is no response from the server the client drops its request with a timeout error. So there is no reason for the server to try to serve requests which won't produce results within this time limit. With this technique the queue will control itself without any length limitations. Requests are not dropped unnecessarily and the response times can be very precisely controlled as well.

Upon natural overload the number of threads will not increase above a certain small value (usually from 5 to 10). However, with an intensive blocking type attack an attacker can achieve that too many threads will be created. To handle this attack a threshold is defined. If the number of threads increases above this value (somewhere around 100) a security alert can be initiated which kills those server threads that look malicious (run too long, wait too long, allocate too much resource, occupy a resource too long, etc.). So it is important that the above proposal strongly requires a secondary resource monitoring, controlling mechanism to somehow avoid the unlimited growing of the number of threads.

The same monitoring approach can be applied to the queue. Since the above proposed solution will self control the length of the queue in natural cases it won't increase above a certain size (somewhere around 50, that can be calculated using an appropriate minimal service time). However a malicious flood can drastically increase the length of the queue. Again, if a certain threshold could be set (e.g. 1000), then malicious behavior can be detected and a security alert is initiated.

However, if difference could not be made between good and malicious requests, some good requests may be sacrificed and all sources may be added to the black list to preserve the operation of the server. Several of the sacrificed good sources may be also present in the white list as previously well-behaving sources, thus they still have the chance to survive, and stay to be served.

6. Architecture of the Proposed System

In case of a suspected attack every incoming request is filtered. Clients on the black list are prohibited from the server, while in case of a flood attack only those clients get admittance that are on the white list. After these initial steps requests will be put into an unlimited queue. As it was discussed in former sections in case of heavy overload the length of this queue may reach an untolerable level. In this case an alert is invoked by the Queue Length Watch to handle the attack. Requests are admitted to the server through the Waiting Time based Request Admission Control (WTRAC) module. This is the heart of the overload protection technique. According to our view a separate thread is created within the server for each admitted request. Thanks to the WTRAC solution blocking type attacks do not work in this case, however they can effect that the number of threads within the server increases significantly. To handle these attacks another alert is invoked by the Thread Number Watch if the number of threads reaches a critical level. The architecture is depicted on Fig. 2.

The main differences of our system compared to the traditional ones are that we use unlimited queue and unlimited number of threads, perform load balancing after the queue and the WTRAC.

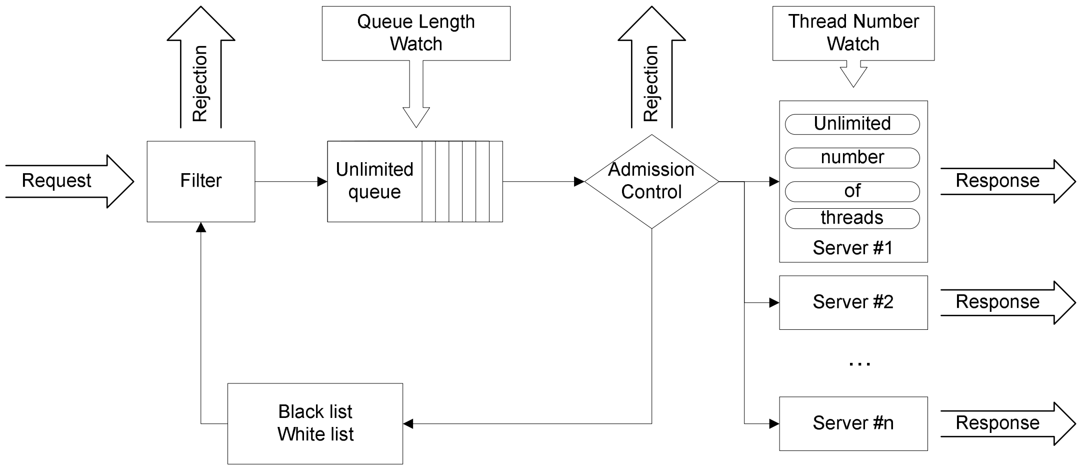

Figure 2. Protection architecture

5.1 Waiting Time based Request Admission Control

The heart of the overload protection concept is this Waiting Time based Request Admission Control method. It should admit requests to the server only when the necessary resources get free. In practice the most crucial resource is the CPU. So when no server threads are running (e.g. because they wait for some I/O result), only then it is reasonable to admit new requests to the server. In a practical implementation the thread states are observe and such situations are recognized easily and reliably.

5.2 Thread Number Watch – Detection of Blocking

Similarly to the Queue Length Watch, the Thread Number Watch handles the unwanted growth of the number of running threads. In natural cases it is unlikely that the number of threads becomes very big if the proposed WTRAC is applied. So a case like this is a clear sign of a blocking type attack. After the Thread Number Watch initiates a security alert, it can more easily select the malicious threads among the running ones than its companion could do in the case of the queue, because blocking type threads can be simply recognized (they run too much, wait too much or block some resources). Then again the sources of these recognized malicious requests can be put to the black list and filtered out in the future.

5.3 Queue Length Watch – Detection of Flood and Overload

Although WTRAC provides a good behavior both from performance and security point of view in case of a malicious flood, it may be imaginable that the unlimited queue grows too long. To handle this situation a Queue Length Watch is applied, which initiates a security alert and then analyzes the content of the queue and decides how to solve the situation. If the queue gets very long, it is sure that the server is facing a malicious flood type attack. In this case the queue contains a lot of malicious requests (several hundreds or a thousand), while the number of good requests are only a few dozens. From this large number of samples the function can more easily select bad ones (more requests from the same source, uniform requests, etc.). In case of a sophisticated attack it is very hard to distinguish between good and bad requests. However in that case some good clients can be sacrificed in order to preserve the operability of the server and the owners of all requests in the queue can be put on the black list. However, those sources that are already on the white list may be omitted from the black list, and thus they will be further served.

Putting together the measurements of the Thread Number Watch and Queue Length Watch yields the conclusion that in case of a long queue, a relatively small number of threads means overload and high number of threads means flood.

5.4 Black and White Lists

One may create a black list that stores those client source identifiers that were identified as malicious ones and therefore are prohibited. Although this black list may provide good solution in case of moderate attacks, if an attacker uses spoofed IP addresses the only possibility to select good requests from the incoming traffic. For this reason one may store source identifiers of good requests on a white list during normal operation and if a DoS attack occurs, based on this list the formerly known good customers can be admitted, while newcomers together with the malicious attacker can be prohibited.

5.5 Benefits

Based on the architecture proposed above the benefits of our system are summarized below:

- Blocking is recognized by the increasing number of threads, thus old threads could be killed.
- Overload is recognized by the growing length of the queue and the most CPU consuming threads could be killed and the content based filter of the black list can be updated.
- Flood is identified by the growing queue and high number of threads, thus content and/or sender based black list can be employed.

This way we can defend against all three types of DoS attacks.

6. Measurements

To test the protection system, a cluster of load generator PCs was established against an Apache server running on Windows XP. A simple http request generator on each of the cluster machines was used to give the stress to the server. Measurements were carried out without overload protection; with overload protection and black list; and finally, with protection but without black list. The parameters were the following: requests were allowed 25 seconds to return with the corresponding response, which was varying in size from 10 to 100 kilobytes. If the response did not arrive within this time limit, the operation was considered a failure. The attacking PCs generated 200 to 400 http requests per second, randomly distributed, for 5 seconds, then 25 seconds of pause was given before the next burst, of which 15 was executed in total. The measurements were repeated with a different number of attacking computers ranging from 2 to 4.

When using protection with black list, the system immediately identified the attackers, thus most requests were not admitted to the server. Naturally, the server survived.

Then the same load was given to the server without the protection system. In this case, the server sometimes crashed, and produced, on the average, 44.15% service ratio in case of two attacking machines and 38.41% in case of multiple attackers.

In the third measurement the black list was switched off and only the self-calibrating WTRAC provided some sort of protection; this time the service rate went up to 53.71% in case of a two attackers and 47.67% at multiple attackers. No server crashes were encountered. The following figure demonstrates the percentage of served requests in the different runs. The dotted line represents the server without protection and the continuous line represents WTRAC queuing.

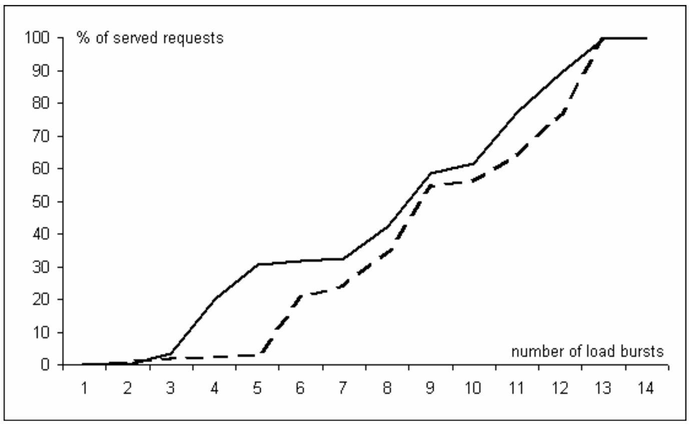

Figure 3. Measurement results

7. Conclusion

This article covered our research on overload detection & protection. The solution uses a new approach for detecting DoS based on the length of an unlimited queue and number of server threads. Old requests, which do not have a chance to enter the server due to too long waiting in the queue, are rejected by the WTRAC as their overall response time would be more than a user is willing to wait. Server processes are also shut down in a similar manner to avoid blocking, thus preserving server functionality. Black and white lists are employed in order to block formerly identified malicious requests and admitting well-behaving customers, respectively. The proposed system can defend against blocking, overloading and flooding type attacks, making it a robust solution against Denial of Service.

8. Acknowledgements

This research and development of the initial prototype of the detection & protection system was done within the Dependability and Security by Enhanced Reconfigurability (DESEREC) [5] project of the Integrated Project of the Sixth Framework Programme of the European Union under the “Information Society Technologies” priority, strategic objective “Towards a global dependability and security framework”.

8. References

[1] “Immense” network assault takes down Yahoo
http://www.cnn.com/2000/TECH/computing/02/08/yahoo.assault.idg/index.html
[2] Cyber-attacks batter Web heavyweights Strikes on eBay, Amazon, CNN.com follow Monday Yahoo! attack
http://www.cnn.com/2000/TECH/computing/02/09/cyber.attacks.01/index.html
[3] Internet quiet after three straight days of attacks Strikes hit E*Trade, ZDNet, eBay, Amazon, others
http://www.cnn.com/2000/TECH/computing/02/10/denial.attack.01/index.html
[4] Thomas G. Robertazzi: “Computer Networks and Systems: Queuing Theory and Performance Evaluation”, ISBN 0-387-94170-3
[5] DESEREC EU FP6 integrated project, http://deserec.eu