DDOS DETECTION BASED ON TRAFFIC SELF-SIMILARITY

A thesis submitted in partial fulfilment of the requirements for the

Degree

of Master of Science in Computer Science

in the University of Canterbury

by Delio Brignoli

University of Canterbury

2008

DDoS detection based on traffic self-similarity

Contents

List of Figures

List of Tables

List of Examples

Acknowledgments

I wish to thank the following people:

My supervisors: Associate Professor Ray Hunt and Dr Marco Reale.

William Rea for providing his expert's feedback regarding long memory processes.

Associate Professor Dr Tim Bell for listening and providing useful comments.

Abstract

The river's gentle roar comes from many quiet drops of water.

—from Hermann Hesse's Siddhartha.

Distributed denial of service attacks (or DDoS) are a common occurrence on the internet and are becoming more intense as the bot-nets, used to launch them, grow bigger. Preventing or stopping DDoS is not possible without radically changing the internet infrastructure; various DDoS mitigation techniques have been devised with different degrees of success. All mitigation techniques share the need for a DDoS detection mechanism.

DDoS detection based on traffic self-similarity estimation is a relatively new approach which is built on the notion that undisturbed network traffic displays fractal like properties. These fractal like properties are known to degrade in presence of abnormal traffic conditions like DDoS. Detection is possible by observing the changes in the level of self-similarity in the traffic flow at the target of the attack.

Existing literature assumes that DDoS traffic lacks the self-similar properties of undisturbed traffic. We show how existing bot-nets could be used to generate a self-similar traffic flow and thus break such assumptions. We then study the implications of self-similar attack traffic on DDoS detection.

We find that, even when DDoS traffic is self-similar, detection is still possible. We also find that the traffic flow resulting from the superimposition of DDoS flow and legitimate traffic flow possesses a level of self-similarity that depends non-linearly on both relative traffic intensity and on the difference in self-similarity between the two incoming flows.

Part I

Concepts

Chapter 1

Background

The objective of our research is to study the feasibility of "Distributed Denial of Service" (or DDoS) detection by measuring changes in the self-similarity of network traffic, under the condition that both legitimate and attack traffic display self-similar properties.

Informally, DDoS is an umbrella term used to describe a family of abnormal network conditions. The two defining characteristics of DDoS are its "distributed" nature and the degraded network service functionality that is its outcome. As the name implies, DDoS can lead to complete unavailability of the targeted services. In our research we will focus our attention on flooding-type subset of DDoS, which is arguably the "hardest" class of DDoS to prevent and to mitigate. Flooding DDoS exploits statelessness of the internet routing infrastructure and a large number of vulnerable network hosts to deliver an overwhelming amount of undesired traffic to the target of the attack.

Our research is inspired by the work on "loss of self-similarity" [AM04] and "error detection using self-similarity" [SM01]. These papers discuss the effects of disturbing normal traffic flow with either DDoS traffic or packet loss. In both cases the observed effect is degradation in the natural self-similarity present in network traffic. Changes in self-similarity of network traffic make detection of DDoS possible.

Self-similarity is a scale invariance property typical of fractals. In the context of network engineering it enables us to construct simple traffic models and possesses the attractive property of not requiring invasive traffic analysis for its estimation. For instance: recording the time-series of traffic intensity (sampled at a given frequency) is enough to estimate self-similarity. This is a notable advantage compared to analysis techniques that require accessing a packets' header or content.

Because of its flooding nature, DDoS increases the number of packets to be analysed. Therefore self-similarity estimation, with its lower "per-packet data collection cost", is an attractive technique. The low statistics collection overhead of self-similarity estimation could enable embedding DDoS detection solutions in high speed network infrastructure equipment where computation resources are committed almost entirely to routing.

Existing papers on DDoS detection via self-similarity estimation are based on the assumption that the DDoS traffic is not self-similar. However, from the review of existing literature and the analysis of software tools used to mount DDoS attacks, it can be inferred that these tools are capable of generating a self-similar flow of packets.

Our question then becomes: if both legitimate and malicious traffic display self-similar characteristics, is detection of a DDoS still possible? In other words, we want to establish how the legitimate and malicious traffic flow contribute to the self-similar characteristics of the resulting traffic as seen by the target in an hypothetical DDoS attack.

In this chapter we introduce concepts and notions used throughout the research project. The nature of *Distributed denial of service* attacks is explained and two possible taxonomies to organize them are described. In order to understand how DDoS attacks are mounted we also outline the design, life-cycle and features of *Bot-nets*.

Self-similarity is another central concept in this research which is introduced in Section 1.2; its uses in the context of network engineering and its role in DDoS detection are further explained in the following sections.

1.1 Distributed denial of service

Informally, *Distributed denial of service* (or DDoS) is an umbrella term used to describe a family of abnormal network conditions. The two defining characteristics of DDoS are its *distributed* nature and the degraded service availability that is its outcome. As the name implies DDoS can lead to the unavailability of services offered over the network.

The *distributed* nature of a DDoS distinguishes it from other types of *Denial of Service* or DoS. DoS can be used to indicate any abnormal condition that results in a degradation of a service offered over the network; in this sense DoS is a superset of DDoS.

A simple fictional example of non-network related DoS follows: imagine a post office run by a single very diligent, but not very bright, clerk. The clerk will process customers in strict order and will call the next customer only when the current customer is satisfied and does not have any more requests. Now, imagine the village prankster queuing up at the post office early in the morning. The prankster will ask the clerk to carry out a simple task, like selling him a single stamp for instance, every time the clerk offers to help him. By the end of the working day no one standing in the queue after the prankster will have had access to the post office services.

Of course in real life the clerk would quickly realize what is happening. Failing that, the prankster would be removed from the queue by an angry mob. However, in a silicon world the example above is not too far from reality.

Denial of service attacks are not simple to classify because of the variety of situations and modes in which they can occur. A classic approach to building a taxonomy of DoS can be found in [Shi08] where attacks are classified by a 4-tuple *(method, effect, consumed resource, resource location)*. Using this framework our fictional example would be classified as *(exploiting clerk's lack of memory, queue stalled, clerk, post office)*.

A less orthodox approach can be found in [Cam05] where the author uses a dance-floor metaphor to group DoS types depending on how they interact with other dance partners (the actors) on the dance floor (the network). This approach has the advantage of highlighting the relationships between the parties involved in a DoS. Within this framework our fictional example would fall under the *partner subtree* (see Figure 1.1) because the clerk, who unwittingly participates in the practical joke, *dances* with the prankster to the exclusion of others.

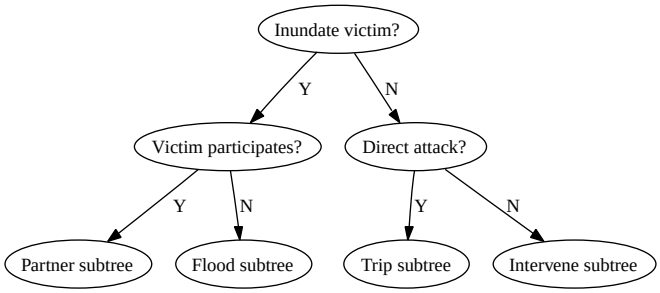


Figure 1.1: Trunk of the "DDoS dance" taxonomy comprising the first three levels of the hierarchy is reproduced here from [Cam05].

In our research we are interested in DDoS belonging to the *flood subtree* in Figure 1.1, in particular we focus our attention on the *subversion tools* and on the *web sit-in* categories (using the terminology from [Cam05], see Figure 1.2) or *DDoS-TE* using the terminology from [AM04].

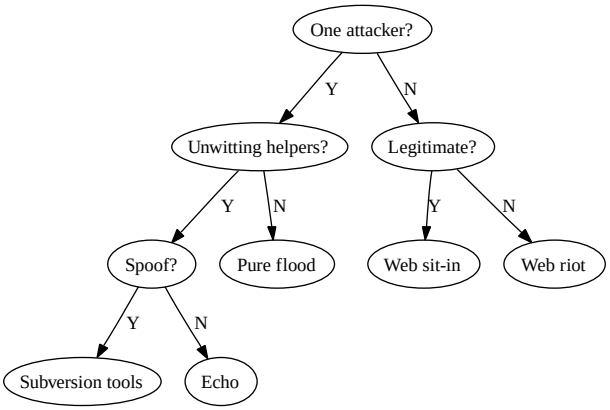


Figure 1.2: Flooding subtree branch for the "DDoS dance" taxonomy. "One attacker" should be interpreted as: one attacker potentially controlling multiple software agents. Reproduced here from [Cam05].

Subversion tools are programs that allow a person other than the rightful owner to remotely control a network connected machine. They can disguise as harmless applications that the legitimate user installs, in which case they are called *trojan* applications, or they are installed and gain access to the machine exploiting one or more security vulnerabilities of the operating system or applications already running on the system; *worms* and *viruses* fall in this category. In some cases subversion tools are installed using social engineering to exploit flaws in security policies and security related processes. An instance of *exploit* falling within the latter category consists in infecting portable media like CD-ROM (disk proper or image files) or thumb drives and using unwitting human vectors to bring malware inside a computer system, bypassing the challenges involved in exploiting remote vulnerabilities altogether.

Web sit-in as used in [Cam05] refers to a large number of distinct users overloading the target with requests with the intention of degrading its functionality. It is possible for a number of users to overwhelm a service with requests even if there was no intention to cause harmful side-effects, for instance: en-mass opening of a news website to read about in important article. We prefer the wider term *flash crowd* to *web sit-in* because the former includes both ill-intentioned and accidental flooding scenarios.

DDoS-TE, short for *DDoS traffic exploits*, is another term that encompasses both the *flash crowd* and the *subversion tools* classes of DoS. In [AM04] DDoS-TE is defined as "a DoS attack whose effectiveness depends on a continuing stream of attack traffic". DDoS-TE exploits the network infrastructure to overwhelm the target with traffic or requests. The attack succeeds when either the computing power or bandwidth available to the target host is exhausted or heavily taxed.

DDoS-TE are hard to mitigate because their success does not depend on weaknesses of the network infrastructure or on flaws in the design of the target system to succeed, but on a more basic principle: *strength in numbers*. One could consider DDoS-TE the worst case scenario DoS attack: even if the target service is free from weak points, it is still vulnerable to this type of attacks. All that is required for the attacker to succeed is to control a great number of network hosts.

1.1.1 Botnets and DDos

Mounting a DDoS attack requires controlling a large number of hosts. Gaining control of hosts is achieved by compromising them with malicious software. These hosts, also known as *agents*, can be then instructed to generate traffic on the behalf of the attacker.

Compromising hosts can be achieved exploiting any local or remote vulnerability. The most common scenario is: an existing exploit *vector* is used to plant the malicious *payload* in the host. The *vector* has the sole function of bypassing the security of the target host and plays no further role in the DDoS attack. The *payload* instead will determine the nature of the DDoS.

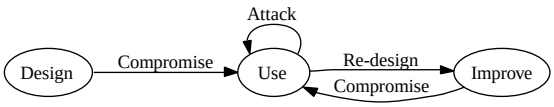


Figure 1.3: A botnet's life-cycle.

Whichever way the hosts have been compromised by an attacker, the end result is one or more rogue processes running on each agent. These processes generally have the following features: hide or obfuscate their presence, listen to instructions from the attacker and execute their requests. Coordinating a large number of agents is a challenging task, so it is natural to expect tools being developed to aid the attacker.

The set of compromised hosts is called *botnet* (short for [ro]bot-net[work]); usually the *botnet* is further identified by preceding it with the name of the tool used to control it or with the name of the payload (or sometimes vector) used to compromise the hosts. For instance the *Storm Botnet* takes its name from the name of the *Storm* worm (payload) which is responsible for its functionality.

Once a host becomes part of a *botnet* it can then be used for a number of purposes, for instance: send unsolicited emails en-mass, mount a DDoS attack or to *click* on advertisement for profit. It is important to understand how an host, once compromised, can be remotely commandeered to perform any task including, but not limited to, the tasks a legitimate user may carry out.

There are, at any given point in time, a number different active botnets (see [DZL06]). Generally they are not controlled by the same person or group. Different botnets compete for control of vulnerable hosts because the value or effectiveness of a *botnet*

grows with the number hosts it is comprised of. Agents become part of a botnet when infected and leave it when malware is removed thus requiring continuous work on the attacker's part to keep the number agents in a botnet at least constant.

The presence of agents is usually discovered only, if at all, after the DDoS attack has begun. Careful monitoring of network activity can facilitate early detection. The measurable side-effect of a DDoS are an increased usage of bandwidth and computational resources. It is of course in the interest of the attacker to consume as little of the agent's resources as possible to avoid detection. With botnets reaching sizes of tens of thousands agents [Sch06] it is possible for attackers to combine small (and hardly detectable) traffic flows from each agent into an overwhelming traffic torrent at the target.

Security researchers do not have access to the source code for DDoS generation tools but rely on executable code recovered from compromised hosts to analyze botnets and infer their design. DDoS tools are in continuous evolution so the results of black box testing and disassembly on an instance of a DDoS payload can quickly become outdated.

In recent years a number of DDoS payloads have been isolated. Attack payload are recovered using *honeypot* systems: carefully monitored network hosts purposely setup to be easily *infected* (see [McC03]). By analysing the payloads a number of common features of botnets have been found (see [Cha02] for an overview). These features include: the use of encryption to conceal communications between the attacker and the agents, some form of authentication to restrict access to the botnet only to its *masters*, multiple tier command and control hierarchy.

Simple encryption is employed to evade detection. The need to keep an agent's resource consumption to the minimum and reduce the payload's footprint drives the choice of encryption algorithm. The goal of the attacker is to protect against detection by casual packet inspection and avoid easy recovery of the authentication credentials, so simple encryption is adequate.

Authentication is necessary to control access to the botnet. The *value* of a botnet is proportional to its size because the *cost* of building a botnet is is dominated by the time taken to identify potentially vulnerable hosts, craft attack vectors and compromise them. The effectiveness of a botnet is also related to its size: the greater the number of hosts in a botnet the higher the volume of traffic that can be generated by its agents. It follows that, for an attacker, it is important to restrict access to a botnet to make up for its *cost*.

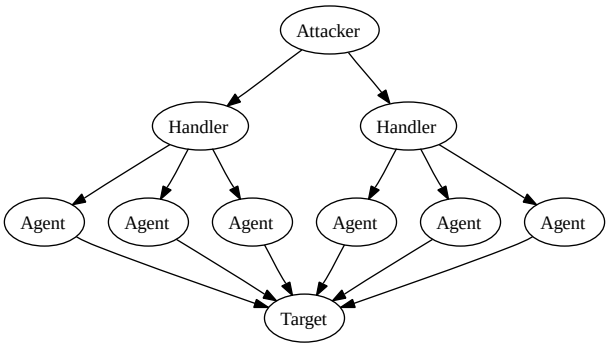


Figure 1.4: 3-tiers botnet. The attacker controls agents indirectly via an handlers tier.

Multiple tier command and control hierarchy (see Figure 1.4) is employed by botnets primarily to distribute bandwidth and workload but it also makes the identification of the attacker more difficult and reduces the likelihood of detection. In multiple tier hierarchies, agents in a botnet are assigned different roles. The role of an agent determines its position in the botnet's *chain*

of command. By having multiple layers (or ranks) the amount of traffic necessary to distribute instructions from a single attacker down to the lower level agents can be programmatically bounded. The agents used as middle tiers will not partecipate in the flooding reducing the total maximum throughput of the botnet.

One of these botnets, called *Shaft*, well represents DDoS generation tools. An analysis of its design and functionality is available in [DLD00]. *Shaft* uses simple encryption to evade detection and authentication to restrict access to its infrastructure. It also features the typical multi tier hierarchy of agents like many other botnets.

Each agent in a *Shaft* botnet exposes the following traffic generation parameters: burst duration, packet size, packet type. Burst duration controls the length of the time period in which the agent generates traffic. Packet size controls the size of each packet generated. Packet type selects which protocol ID is used in the packet's header: UDP, TCP, ICMP. *Shaft* does not allow to control other details of the packet header or its content; a different packet template is used for each one of the previous three choices.

An interesting feature of *Shaft* is the ability of agents to report statistics on packet generation to their parent agents, this is conceivably aimed at tuning the attack parameters dynamically and constitute one of many evolutionary improvements security researchers find in each new generation of these tools.

Security researchers predict growth of botnets in size and expect DDoS tools to undergo evolutionary changes as they are refined to bypass intrusion detection and packet filtering systems.

1.1.2 DDoS detection techniques

Intentional DDoS is different in nature from other types of attacks on network security as it does not attempt to exploit design flaws in the target's system architecture or software, instead it leverages asymmetry in numbers between attacker and target to increase request rate or traffic flow towards the victim(s) beyond their ability to cope.

The ability to setup a DDoS against a given target depends mainly on two factors: the availability of unsecured personal computers connected to the internet and on the design of packet routing in Internet Protocol (IP) based networks.

DDoS and IP networks

IP based networks like the internet scale well to large sizes because the cost of routing packets does not grow with the size of the network. This is thanks to the stateless nature of routing decision in the IP infrastructure; in other words: routers in an IP network do not store *per packet* or *per connection* information, instead each packet is forwarded based solely on the information available in its header. DDoS takes advantage of this to inundate the target with packet that are not part of a meaningful conversation between two network endpoints. The success of the internet, at least from the technical standpoint, is rooted in the stateless routing property of its nodes. For this reasons changing the architecture of the internet to prevent DDoS is not desirable.

DDoS and unsecured hosts

At a high level of abstraction DDoS traffic traverses three major network domains: the source ISPs networks, the core network and the target's ISP network.

Pressuring the owners of unsecured network agents to improve their security is a long term solution to the problem that assumes good will and requires commitment of resources by parties (trojan infected users and source-end ISPs) that are not (or only

marginally) affected by the DDoS they unwillingly participate in. Such solutions require widespread adoption of policies that cannot be easily imposed onto ISPs and end users due to the transnational nature of the internet and it is therefore considered to be outside the scope of this research.

Different technological solution target one or more of these network domains. Some are designed to stop the surge of DDoS traffic at the source in the origin's ISP network other solutions call for improvement of the core network routers and other techniques are designed to be deployed in the vicinity of the victim's servers.

Solutions that target the source ISP network and the core network require widespread adoption and investment in areas with little economic incentive for improvement.

On the other hand a potential victim of DDoS has an economic incentive to implement some form of defense and has control on a small portion of the network: directly in their datacenter and indirectly on their upstream ISP. Techniques designed to be deployed in the vicinity of the victim's servers are more likely to be implemented in the short term. We call these techniques: *target-end* indicating the portion of the network close to the target system, possibly extending upstream to include the target's ISPs network.

Different target-end DDoS mitigation techniques have been proposed in the last few years (see [KLCC06], [MR05], [XLS01], [WZS04], [TV03], [CS05]) claiming different degree of success in limiting the impact of an attack.

Proposed DDoS mitigation techniques can, in general, be broken down in a number of abstract operational blocks: DDoS traffic detector, packet classifier, packet filter, controller. DDoS mitigation solutions differ in the number, nature and placement, in the network, of these operational blocks.

For instance a *traceback* based technique (see [LLY05]) may detect abnormal traffic at the target server and update packet filters located in the ISP routers or even in core routers; another solution may detect DDoS traffic at the edges of an ISP network and filter packets in place while traffic statistics are aggregated from all edge routers by redundant controllers.

All DDoS mitigation techniques require, regardless of their design, a *detection* stage to notify the other components involved when an attack is in progress thus effective and reliable detection is crucial for any mitigation technique to work.

DDoS Detection is not a trivial task because the network infrastructure transporting the packets cannot easily distinguish between legitimate and malicious traffic. Target-end DDoS traffic detection techniques can be classified in three categories:

- Traffic self-similarity based detection

  Self-similarity based detection of DDoS traffic are built on the assumption that unperturbed internet traffic exhibits self-similar characteristics and long range dependence (LRD) [ERVW02]. A few techniques have been proposed for offline or realtime [XLLH04] estimation of traffic self-similarity (and lack thereof).

- Traffic profile deviation from baseline

  The assumption that internet traffic possess LRD properties [ERVW02] allows to describe statistical characteristics of traffic that remain constant or change slowly in time. A collection of statistical characteristics of traffic during normal operations (i.e. in absence of DDoS traffic) constitutes a baseline profiles. Subsequent monitoring of the packet flow can be used to compare realtime traffic profile to the previously recorded baseline to detect deviation.

- Source address based traffic volume accounting

  Proposals exist for special data-structures [GP01] to hold source address based packet accounting information in a space efficient manner, designed to avoid being exploited to mount a DoS attack. The information is then used to filter traffic coming from source address ranges deemed *suspicious*.

We focus our interest on self-similarity based detection of DDoS because it possesses the useful property of not requiring invasive traffic analysis for its estimation. For instance: recording the time-series of traffic intensity (sampled at a given frequency) is enough to estimate self-similarity. This is a notable advantage compared to analysis techniques that require accessing a packets' header or content. Self-similarity based detection also has the potential of detecting new DDoS traffic without requiring training on an attack's specific traffic flow.

Because of its flooding nature, DDoS increases the number of packets to be analysed. Therefore self-similarity estimation, with its lower per-packet data collection cost, is an attractive tool. The low statistics collection overhead of self-similarity estimation could enable embedding DDoS detection solutions in high speed network infrastructure equipment where computation resources are committed almost entirely to routing.

1.2 Self-similarity

What is self-similarity and how does it affect network traffic modelling?

Self-similarity is a term introduced by B. Mandelbrot:

> statistically "selfsimilar," meaning that each portion can be considered a reduced-scale image of the whole
>
> —from the abstract of B. Mandelbrot's 1967 paper 'How Long Is the Coast of Britain? ...' [Man67]

In other words self-similarity refers to a scale invariance property. A classic example of self-similarity is the silhouette of a coastline, as seen from above, at increasing distance: as we move further away, the coast line will look like containing many smaller versions of the initial silhouette. An interesting consequence of self-similarity is: if, still in the context of the above example, we were to shuffle the set of coastline silhouette pictures taken at different distances, it would be very hard to re-order them from the closest one to farthest one; informally proving that the silhouette looks very similar at different scales or in other words it is qualitatively *scale invariant*.

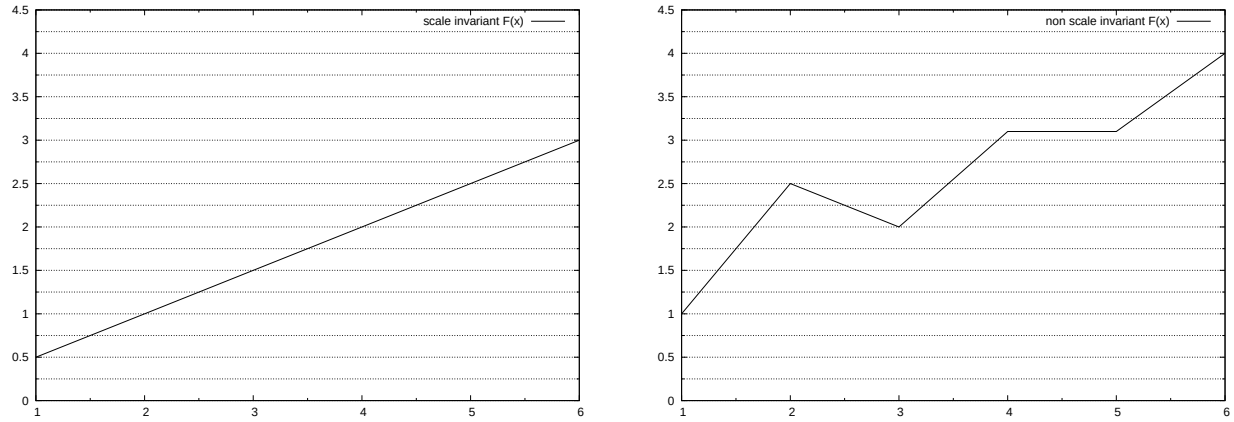
Formally, if a curve *F(x)* is scale invariant under the transformation $x_1 = bx, y_1 = ay$, we have:

$$F(bx) = aF(x) = b^H F(x) \tag{1.1}$$

where the exponent $H = log(a)/log(b)$ is called the Hurst exponent (also known as Hurst parameter or simply H). When $0.5 < H < 1$ the curve displays long range correlation.

When applied to network traffic, self-similarity intuitively means that observing plots of traffic intensity at different time-scales (say 10, 100, 1000 seconds), they will look very similar to the naked eye, thanks to its scale invariance property (see [LTWW94] page 4).

The scale invariance property of self-similar time-series can be visualized using a *log-log* plot. A *log-log* plot is a plot with logarithmic scale on both axis. The *log-log* plot of a scale invariant curve *F(x)* is a straight line and the slope of the line depends on the H exponent (see Figure 1.5). This property of scale invariant curves is used by a number of self similarity estimators to approximate the value of H from a *log-log* plot (see [TTW95]).



(a) (b)

Figure 1.5: Log-log plot of a scale invariant (a) and a non scale invariant (b) curve.

Knowledge of the self-similar nature of network traffic helps building models and simulations with a higher confidence of closely reproducing phenomena such as: queueing delay, packet loss and congestion that occur in real networks.

A very interesting property of scale invariance is that it buys parsimony when describing a process like packet arrival or traffic intensity in time. Instead of having to devise complex models for bursty-ness, that would require several parameters, we can effectively capture most statistically significant aspects of network traffic with three parameters (see [ERVW02] page 801, second column): intensity mean, intensity variance and a self-similarity index.

1.2.1 Self-similarity of Network traffic

In the past network packet arrival times were modelled as Poisson processes mainly because of their relative analytic simplicity. About a decade ago, papers regarding the non-Poisson nature of LAN packet inter-arrival times [LTWW94] started to appear, similar conclusion regarding wide area network traffic followed closely [PPFF95].

These studies suggested network traffic displays a fractal-like behaviour called self-similarity. In the following years the network engineering community became very interested and active in measuring self-similarity and understanding the reason behind fractal-like behaviour in network traffic [CB97, BUBS97, ERVW02, WTSW97, KS02a, FGW98].

Self-similar packet inter-arrival time was initially considered bad news by the network research community. The immediate consequences of fractal-like behaviour are longer average queueing delays and extended congestion times [SCVK02]. Also, at the time, the reasons behind the emergence of self-similarity in network traffic was not clear and most of the initial research was focussed on measuring its impact rather then understanding its origin.

Research attempting to explain the origin of fractal-like behaviour in network traffic provides at least three different plausible causes, each justifying the observed scale invariance for a different range of orders of magnitude. One related to the network infrastructure and protocols, the other related to the nature of the information transferred and the third related to user and application behaviour.

At the lower level, the interaction between transport protocol (TCP) and the network infrastructure (queueing delays, packet loss) can explain pseudo self-similarity across small time-scales close to the average RTT of the network (see [ERVW02]). Pseudo self-similarity is the term sometimes used to describe the fractal-like property of network traffic, because it only holds only for few orders of magnitude of the scale.

In [CB97] Crovella et al. discuss file sizes distribution and user (or application) behaviour as causes of self-similarity in world wide web traffic. The statistical distribution of file sizes (document, video, application size on the WWW) is *heavy tailed* like the statistical distribution of the number of pages of books on a library's shelves (see [Man82]). Informally heavy tailed file sizes distribution means that there is high variability in the sizes and a lot more small files then big files. The concurrent transfer of files with sizes drawn from an heavy tailed distribution explains the emergence of self-similar traffic. Traffic generators based on simultaneous transfers with sizes drawn from Pareto, or other heavy tailed distribution, have been shown (see [HKBN07]) to reproduce self-similarity and better represent real network traffic. This explanation is independent of the specific transport or application, it applies to any protocol supporting bulk data-transfer (not only to HTTP which is the underlying protocol of the world wide web).

User (or application) behaviour can also account for self-similarity. If user activity is modelled as a succession of downloading and reading times and the reading time distribution is heavy tailed then fractal-like behaviour emerges. User behaviour does not seem to be the dominant source of self-similarity and is dismissed (see conclusions in [CB97]) in favour of explaining self-similarity purely by superimposition of concurrent file transfers with sizes drawn from an heavy tailed distribution.

Feldmann et al. propose, in [FGW98], a *multi-fractal* interpretation for network traffic self-similarity. Informally, the multi-fractal interpretation accommodates all of the three causes of fractal-like behaviour in network traffic (network infrastructure, resources sizes distribution, user behaviour); each dominating in a limited range of the time-scale. The network infrastructure and protocol induced effects dominate up to 100 milliseconds then the behaviour induced by high variability in transfer sizes dominates roughly from 100 milliseconds to 10 seconds; user behaviour induced effects dominate the time-scales beyond tens of seconds.

1.2.2 Self-similarity estimators

Self-similarity estimators are approximate methods used to measure the fractal-like property of a time-series. Network traffic can be treated as a time-series by extracting some basic but salient features, namely: the aggregated size (data throughput per time period) over several sequential periods and the packet inter-arrival time. Both aggregated size time-series and packet inter-arrival time-series have been shown to possess self-similar properties (see respectively [PPFF95, CB97] and [BUBS97]).

Hurst et al. introduced the first self-similarity estimator called re-scaled range statistic (or simply: R/S) in [HBS65]. Hurst was a hydrologist and his work on the R/S method was in relation with planning optimal size for water reservoirs given historical

inflow data. The inflow data is an example of what we call a time-series, in this case the sequence of yearly water volume inflow spanning tenths of years.

Many different estimators are currently available, most of which are described in sufficient details in Taqqu et al. empirical study [TTW95]. Estimators like the original R/S method (see [TTW95] Section 3.6), introduced by Hurst himself, are essentially graphical and rely on plotting an appropriately re-scaled time-series on a log-log scale and then apply linear regression to find the slope of the fitting line. Others, like the periodogram method (see [TTW95] Section 3.7), are based on the estimation of frequency density spectrum or alternatively on decomposition using the wavelet transform like [AAVV98]. Higuchi's method (see [TTW95] Section 3.4) computes the fractal dimension of a time-series to achieve the same goal.

Estimators differ in the methodology used to obtain the measure of self-similarity as well as many other areas: computational cost, memory space cost, convergence speed, sensitivity, bias.

1.2.3 Self-similarity and DDoS detection

DDoS detection techniques based on self-similarity estimation rely on the assumption that malicious traffic alters the scale invariant property of legitimate traffic. In Section 1.2.1 we have referenced existing literature supporting the widely accepted notion that network traffic on both LANs and wide area IP networks (the internet) displays self-similar features. Intuitively it seems reasonable that a large number of DDoS agents emitting an intense and continuous flow of packets will bury the natural bursty-ness of the legitimate traffic.

Allen's work on DDoS detection (see [AM04]) confirms that superimposition of self-similar and high intensity non self-similar traffic results in traffic with a significant degradation of the original self-similarity.

A similar but more sophisticated result is achieved by Xiang et al. in [XLLH04]; their approach to DDoS detection is based on a simplified R/S estimator implementation and on the observation the variance of the estimated value for H in time. In their tests they observe a change in the value of H in correspondence of DDoS traffic in accordance with Allen's work.

Li et al. in [Li06] shows how abnormal traffic flow affects averaged H estimates. Their conclusion is that the Hurst parameter estimate always decreases in presence of abnormal traffic.

All the above three studies suggest that (non self-similar) DDoS traffic disrupts the self-similarity of normal traffic and that variations in the estimate of the H parameter are a good indicator for the presence of an attack.

1.2.4 Self-similar DDoS traffic

DDoS detection techniques described in Section 1.2.3 are based on the observation that the presence of a DDoS attack degrades the self-similarity of normal traffic flow. These observation are based on the assumption that DDoS tools do not generate self-similar traffic. We believe this assumption may not hold for long given the current state of the art of DDoS generation tools and their improvement trends.

We know from Taqqu et al. (see [TWS97]) that superimposition of multiple on-off packet trains with Pareto distributed periods results in self-similar traffic. The value of the Hurst parameter for the generated traffic depends on the shape parameter of the

Pareto distribution. We can therefore control the value of the H parameter for the generated traffic by choosing appropriate values for the on-off period distribution.

Shaft, the DDoS tool we discussed in Section 1.1.1, is capable of controlling both start and duration of packet bursts for each of its agents. Controlling burst and duration is sufficient to create an on-off packet train like the one used by the self-similar traffic generator described by Taqque ae al. A clever attacker could instruct its agents to emit packets with Pareto distributed durations, effectively generating self-similar DDoS traffic.

Even though we possess no evidence that *Shaft* has been used in this mode, we believe it is possible to do so. Assuming we are correct in expecting tools like *Shaft* to be able to generate self-similar traffic, then detection of DDoS via self-similarity estimation cannot rely on the malicious traffic flow to lack self-similarity.

1.3 Research project outline

We begin by defining DDoS and discussing its nature and implications for network security. The design and features of DDoS bot-nets are also explained. Potential mitigation techniques are outlined and our choice of focussing on techniques that address the target-end part of the network is justified by highlighting the economic advantages and technical restrictions at play.

The concept of self-similarity and its relation to network traffic is introduced in Chapter 1 followed by a discussion on self-similarity estimation which is further expanded at a later stage when addressing estimators' implementation.

We then proceed to define the network model used by our simulations to help framing the rest of our research and outline requirements on software modules. In chapter Chapter 2 we discuss the abstraction and simplification steps involved in defining a minimal model that is still useful for reproducing the DDoS conditions we are interested in.

Chapter 3 describes the design of the simulation framework used in the rest of the research. The framework is modular and features an improvement over pure event based simulation engines to take advantage of the specific characteristics of our network model.

In Chapter 4 we look for acceptable implementations of self-similarity estimators discussed in the literature. Our requirements mandate that an acceptable implementation should be modular and should allow to be easily integrated with the other software modules we use. Testing proves that readily available estimator libraries are not usable with large network traffic datasets. We re-implement the estimators with large datasets in mind and run thorough tests, comparing results from third party estimators and our implementations to make sure they are correct. We also develop a simple ad hoc distributed computation library and utilities to run the estimators on our collection of traffic traces.

An observation on the implementation of the periodogram estimator method prompted an investigation on a interesting modification to its algorithm which are discussed in Section 4.5.3. The resulting modified periodogram estimator is not used in the following stages of this research because the wavelet estimator, with its multi resolution analysis features, fits better with our requirements.

In Chapter 5 we evaluate two separate instances of the traffic generator described in [TTW95] and reviewed in [HKBN07]. An existing third party implementation and our own are tested using wavelet multi resolution analysis (MRA, see Section 4.7).

The tests results indicate that the quality of generated traffic quickly degrades when moving away from a set of ideal operating parameters. We deem the quality of the generator not sufficient for our simulations and decide to use fractal Gaussian noise (fGn, see [Pax97]) to synthesize traffic trace. The quality of both the fGn source and the shaping process are tested using MRA and we conclude is that our shaping process does not affected the self-similar properties of the fGn time-series.

Using fGn traffic sources and the wavelet self-similarity estimator we test our model to verify that our design decisions are correct (see Chapter 6 for details). We test single source behaviour with and without congestion for different values of the H parameter. We also test the behaviour of the model with two identically setup sources in absence of congestion.

Finally, simulations are run with both sources set to generate traffic for different values of H parameter at different average intensity levels. Chapter 7 contains the description of the simulations scenarios and a discussion of the results.

We observe how the self-similarity of the resulting traffic changes depending on the ratio of the two sources' traffic intensity and the distance of the two sources' value of H. We conclude that, even when DDoS traffic is self-similar, detection is still possible. We also find that the traffic flow resulting from the superimposition of DDoS flow and legitimate traffic flow possesses a level of self-similarity that depends non-linearly on both relative traffic intensity and on the difference in self-similarity between the two incoming flows.

1.4 Summary

Distributed denial of service indicates an abnormal network condition brought about by a large number of hosts acting in concert and that results in a degradation of one or more targeted services offered over the network.

The self-similar nature of attack free network traffic is supported by many studies (see [ERVW02, WTSW97, CB97, BUBS97, PPFF95, LTWW94]) and its degradation in presence of DDoS has been studied before (see [AM04, XLLH04, Li06, Li04]). Degradation of self-similarity can be used as a basis for DDoS detection. DDoS detection is an important part of any mitigation technique and it is worth studying and improving. DDoS detection via self-similarity is, so far, based on the assumption that the malicious traffic flow does not posses self-similar properties.

Given the trend of quick evolutionary improvements of DDoS generation tools, we believe it is likely DDoS tools could be used to emit self-similar traffic contrary to the underlaying assumption of current DDoS detection techniques based on self-similarity estimation. Our review of the analysis of the *Shaft* DDoS generation tool (see [DLD00]) suggests that it could be used to generate self-similar traffic in a fashion similar to one described in [TWS97] thus supporting our thesis regarding self-similar DDoS traffic.

We intend to investigate the result of superimposing self-similar legitimate and malicious traffic flows and deduce under which conditions detection of DDoS is possible.

Part II

Implementation

Chapter 2

Network Model

In the following sections we discuss three network models suitable for simulating the DDoS traffic conditions relevant to this thesis. The intermediate steps and results of the discussion will be presented visually with block diagrams in which the model's components are the building blocks (hosts, routers, links) and the diagrams represent the overall model.

Modelling of the network involves both conceptual and practical simplifications of reality, we attempt to strike the right balance between a simple implementable model and a complete one by including components that, according to existing literature, account for the effects which are the subject of this research project.

We begin by introducing the set of network characteristics the model should preserve, then we proceed to justify our choices and later we introduce the first version of the model.

We will start from a very simple model, we will then extend it incrementally with additional components to account for the phenomena we need to reproduce and to enable us defining new simulation scenarios.

2.1 Requirements

For the model to be useful it is required to reproduce effects influencing self-similarity of network traffic. In this section we list known network properties and for each one, we justify inclusion or exclusion from the model.

A. Observable network traffic behaviour that our network model should preserve are:

1. **Self-similar packet inter-arrival time-series and self-similar aggregated sizes time-series**

   Network traffic characteristics as self-similar packet inter-arrival and aggregate sizes time-series are the focus of the interest of this research project and clearly need to be part of the model.

2. **Packet loss due to congestion and consequent loss of self-similarity**

   In presence of a DDoS attack, which implies increased traffic intensity, the network infrastructure is more likely to discard packets. Discarding traffic in turn affects the packet arrival distribution at the target and may affect the

degree of the traffic self-similarity. A detector based on estimation of traffic self-similarity would thus be affected by congestion. Packet loss due to congestion is therefore relevant to our research.

3. **Bandwidth limits**

   Bandwidth limits on transmission links affect traffic self-similarity in a negative way. When traffic intensity bursts or exceeds in a sustained manner the available bandwidth, packets are queued in the routing nodes experiencing congestion. Queued packets are delayed and in case of prolonged congestion they are discarded.

   As a side-effect of congestion, packets exit the routing node at approximately maximum link speed which in turn results in a loss of self-similarity (see [AM04]). While we want to take into account this effect in our model, we would also like to limit the number of entities modelled. We therefore factor the distributed nature of bandwidth limits into one single rate limiting queue component (see Section 2.4.1) that will approximate loss of self-similarity due to congestion.

B. Observable network traffic behaviour we choose to not model:

1. **Packet transmission delay**

   Packet transmission delay is the result of transmission bandwidth and link length. Packets travelling from source to destination will generally traverse many different links each one with potentially different operating parameters (bandwidth, distance). Also, packets from different sources will also generally take different paths inside the network to reach the same target.

   Under the assumption that the network is stable from the routing point of view (that is: all packets from a source A to a destination B always travel along the same path) the transmission delay is a constant value 'd_{AB}' for a given (source node, destination node) tuple.

   Formally, under the assumption of network routing stability, the following holds: $T_{delay}(A,B) = d_{AB}$ for any pair A, B of source and destination nodes. In our study of DDoS detection, the target of the attack B is determined and thus the previous equation depends only on the source A.

   Exploiting the fact that given a source and destination pair (A, B), the delay 'd_{AB}' is constant, we can imagine shifting the packet source generation time-series by 'd_{AB}' for each packet source A to account for the delay and assume the transmission of packets is instant. Packet arrival time-series as observed by the target node is unaffected by our simplification.

   A further potential simplification is possible if we assume that a constant finite shift of packet emission time-series at the source will not affect the self-similarity of the inter-arrival distribution and of the aggregate traffic. This assumption is valid because by definition the self-similarity of a time-series is a scale invariant property; in other words, it is not affected by affine transformations thus a constant and finite shift of packet emission time-series does not change self-similarity. This last simplification allows us to disregard transmission delays completely in our model. The advantage of this approach is a reduction of entities to be modelled, in this case: transmission links.

2. **Packet processing delays**

   Packet processing delays are also not modelled, similarly to packet transmission delay they are the result of packets traversing the network infrastructure. Processing delay is due to the multiple routing nodes (instead of transmission links, like in the case of transmission delay) traversed by a packet on its way to the destination.

Total packet processing delay for a packet p can be formally defined as: $P_{delay}(A,B,p) = \sum_{n \in N(A,B)} [T_1(p,n) - T_0(p,n)]$ where N is the set of routing nodes traversed by packet p from the source node A to the destination node B and $T_1(p,n) - T_0(p,n)$ is the time a given packet p spends inside a node n before being transmitted.

Packet processing time depends on mainly two factors: the local traffic conditions at each of the routing nodes, which affect the queue length, and the packet forwarding time. Also the queuing discipline implemented by each node could affect packet processing delay, but in this discussion we assume all routing nodes have a strict LILO queuing discipline.

As a result, the cumulative packet processing delay of all nodes traversed is highly variable and has been shown to explain some degree of traffic self-similarity (see [BUBS97]). While in a real network both the packet source's emission distribution and the processing delays have been shown to explain traffic self-similarity, the presence of only one is sufficient for traffic self-similarity to emerge.

We therefore decide to rely only on the packet sources to account for self-similarity and reduce the number of entities in our model by effectively ignoring packet processing delays.

2.2 Initial Model

This is the first and and also the simplest of the three models introduced in this chapter. The other models extend it incrementally to satisfy all requirements listed in Section 2.1. After introducing two components used in the model's diagrams we describe the simplification steps used to reduce a large scale network into our simple model.

2.2.1 Detector component

For the purpose of this thesis we define a detector as a functional block that observes a sequence of network packets flowing in a given direction and provides as output a tuple (flag, confidence), whose elements are respectively: a boolean value indicating if detection of DDoS occurred and a real valued confidence level in the range [0,1].

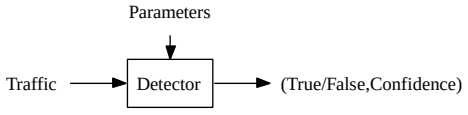


Figure 2.1: Component block for a generic detector including its inputs and outputs.

In general a detector may expose some configuration parameters, may require a realtime clock reference and may have an internal state. To account for all of the above we formally define a detector as any function d of the form $d(\bar{t}, \bar{p}, \bar{s_0}, c) = (r, l, \bar{s_1})$ where $\bar{t}$ is a contiguous subset of the network traffic time-series, $\bar{p}$ is a vector of operating parameters dependent on the function d, c is the current time-stamp, r is a boolean value and l is the confidence level. The vectors $\bar{s_0}$ and $\bar{s_1}$ are respectively the previous and updated state of the detector. The cardinality of $\bar{t}$ is called the 'detector window size'.

The equivalent block diagram component used from here on to describe a detector can be found in Figure 2.1.

2.2.2 Test point component

We also introduce a ‘test point’ block diagram component (see Figure 2.2) to allow connecting detectors (see Section 2.2.1) and other traffic analysers (see the next paragraph) to the network.

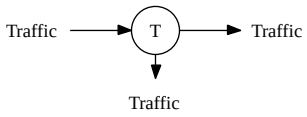


Figure 2.2: Test point component for block diagrams. Forwards a copy of all incoming traffic to its outgoing edges.

The abstract component behaviour is to forward a copy of all incoming traffic to both its outputs. Test points are useful to indicate where traffic is sampled in a network model diagram. Test points can be used to connect detector(s) or *estimators* for other network parameters to the network topology.

Estimators like a Hurst parameter estimator or a traffic intensity estimator are components that do not modify the traffic flow in either content or timing, their presence as no effect on the traffic observed by the target. In general *estimator* components provide a mean to perform computations based on traffic flowing at a particular point in the network model.

2.2.3 Simple model

We proceed to describe the simplest model that still accounts for the phenomena listed in Section 2.1 list A. Given an arbitrary graph representing a network (for instance Figure 2.3a), we partition the original graph in two sub-graphs: one containing all potential DDoS traffic source nodes and the other containing the DDoS traffic target nodes (see Figure 2.3b). Such partitioning is always possible and in practical cases the two partitions will be connected by a small number of edges. In this research project we limit our considerations to partitions connected by only one edge, but the resulting model is applicable to the many-edges case.

Next, imagine collapsing all source nodes into a single ‘traffic source’ component and all target nodes in a single ‘target’ component. The result (see Figure 2.3c or Figure 2.4) is the first version of our network model.

The traffic sources component and target sub-network component in Figure 2.4 represent the entire network topology as a compact model. The notion that a sigle emitter can substitute an entire source graph is a very convenient simplification which is not valid in general when dealing with network simulations. However, it is acceptable when operating under the following conditions: (1) traffic is assumed to be uni-directed from sources to destination, (2) only packet arrival times and sizes are relevant in the analysis of traffic and the traffic source is either (2a) replaying previously captured traffic traces, or (2b) a special traffic generator that reproduces the sizes and arrival time distributions of a cloud of sources.

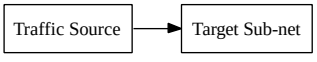


Figure 2.4: Simple network model. All traffic between the traffic source and the target flows across single edge.

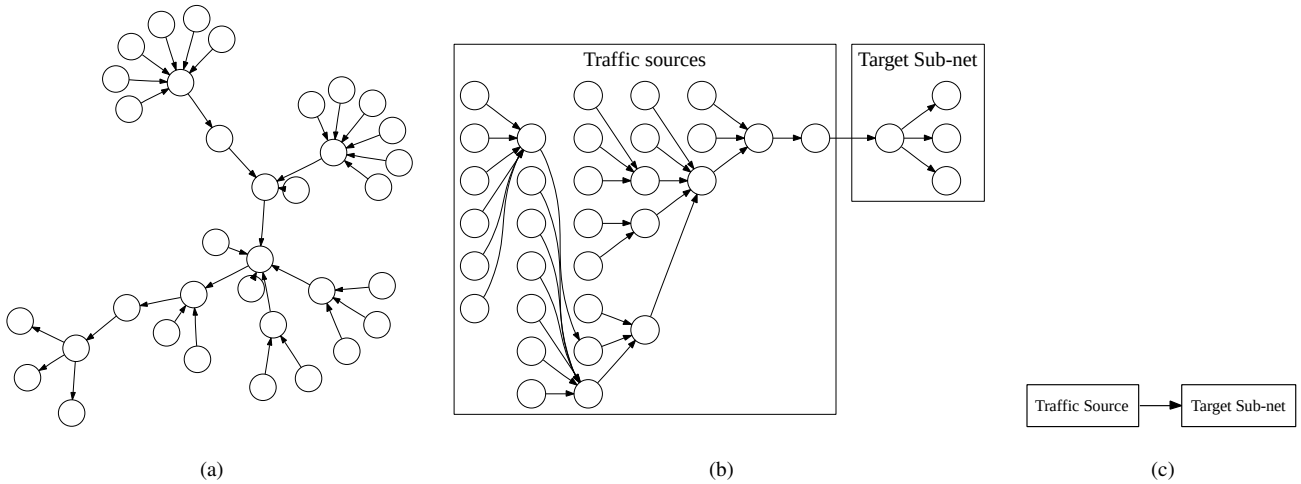


Figure 2.3: Simplification steps to obtain the model described in Section 2.2.3. Step 1: Initial sample network topology (a). Step 2: Hosts are clustered as either traffic sources or destinations (b). Step 3: Cluster are collapsed into opaque blocks (c).

Even a very simple model like the one in Figure 2.4 can be useful to verify the correctness of assumptions on the nature of a traffic source; it can also be used to test the implementation of the component involved without adding further complexity. Testing of estimators and traffic generators will be the primary use of this model during this research project.

2.2.4 Example use case

Figure 2.5 displays an use case for the model we just introduced. To obtain the block diagram in Figure 2.5 from Figure 2.4 we place a ‘test-point’ on the edge between the source and target blocks, then attach an ‘Hurst parameter estimator’ to the free edge of the test-point.

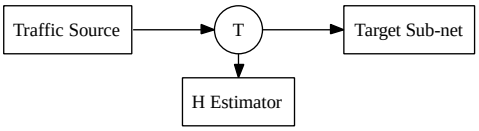


Figure 2.5: Example use case for the simple model: test the source’s H parameter setting or the estimator’s correctness.

This setup can be used to verify a traffic generator’s output or evaluate an estimator for the Hurst parameter.

2.3 Extended Model

The model described in the previous section does not make any assumptions on the nature of the incoming traffic. While it represents a real world scenario for a detector, it does not model either the separation between legitimate traffic sources and attack traffic sources or the notion that traffic observed at the target host is the result of the superimposition of legitimate and malicious traffic.

Traffic arriving at the target in Figure 2.4 originates from many hosts distributed geographically; while it is not possible to establish which nodes originate legitimate traffic and which nodes originate malicious traffic both types of traffic reach the target after being routed via the network infrastructure.

We wish to extend the previous model to include the concept of legitimate and malicious traffic superimposition we just introduced; in order to do so we proceed as follows:

Step 1 We introduce a new functional block (see Figure 2.6) called 'traffic mixer' from here on. The traffic mixer outputs a superimposition of the traffic entering its incoming edges. The traffic mixer output conserves the chronological order of incoming packets and assume all its incoming and outgoing edges have infinite capacity so that no delay or queueing occur.

Step 2 We group all traffic sources conceptually in one of the two categories: malicious and legitimate. We also, for the sake of this model, aggregate the entire routing infrastructure in a single 'traffic mixer'.

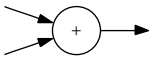


Figure 2.6: Traffic mixer component for block diagrams. Mixes traffic from two sources into one single flow.

Applying the above steps to our first iteration of the model we obtain a second one (see Figure 2.7) which overcomes the limitations mentioned at the beginning of this section by splitting the traffic source into two distinct sources each corresponding to a traffic category: a legitimate traffic source and a malicious traffic source. The traffic generated by the two sources is then 'mixed' and forwarded towards the target host by the 'traffic mixer' component.

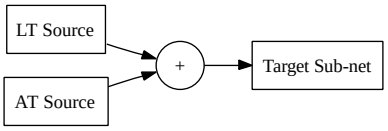


Figure 2.7: Extended network model. Legitimate traffic (LT) and Attack traffic (AT) sources are separated.

2.3.1 Example use case

Our extended model allows us to test the relationship between the Hurst parameter value estimated at the test point (see Figure 2.8) and the Hurst parameter of each of the sources traffic sources.

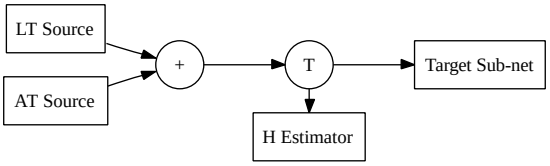


Figure 2.8: Example use case for the extended model: test relationship between H of the sources and H as observed by the target.

2.4 Final Model

The resulting model from the previous section does not account for packet loss due to congestion as required by Section 2.1 list A item 2. We add another class of components to our model to satisfy our original requirement. The component class will be called ‘rate limiting queue’ and one instance will be placed after the traffic mixer component on the model from the previous section. The resulting diagram can be found in Figure 2.10.

2.4.1 Rate limiting queue

The rate limiting queue is internally composed by a finite queue attached to a rate limiter (see Figure 2.9). The rate limiter allows packets to exit the queue at a specific rate while the queue has a maximum finite length and a packet discarding policy. The resulting component can be used to model packet loss and changes to traffic self-similarity due to congestion.

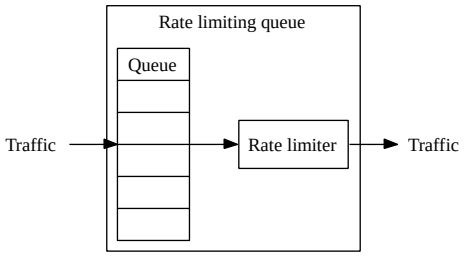


Figure 2.9: Rate limiting queue internals.

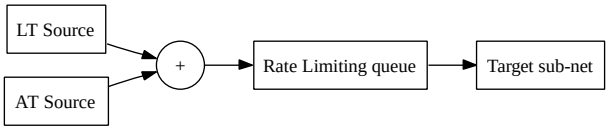


Figure 2.10: Network model complete with rate limiting queue components.

2.4.2 Example use case

The final model use case instance in Figure 2.11 is designed to test the expected loss of self-similarity in case of congestion (see Section 2.1 list A item 2).

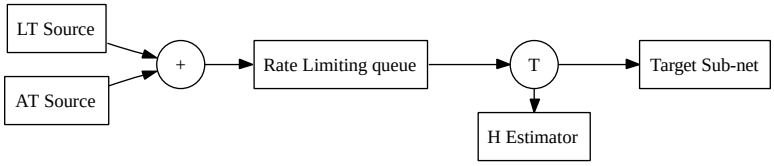


Figure 2.11: Example use case for the final model: test loss of self-similarity under network congestion.

2.5 Summary

Defining a simulation model helps framing the research problem and is the first step towards setting up our network simulations. Self-similarity estimation based DDoS detection requires the network model to reproduce specific real network phenomena like: conserving self-similarity of the sources and packet loss due to congestion. We listed and explained the requirements for the model then proceeded to introduce the model.

A first, simple model, is outlined and then to extended it in two steps: the first to separate legitimate and attack traffic generation and the second to account for packet loss due to congestion. Use cases are listed to illustrate potential usage for each of the three resulting models and justifications for for each incremental step are given.

The model highlights the abstract building blocks we require to proceed with our simulations. The final model features an one-way tree-like network topology and the most significant building blocks are: two independently controllable self-similar traffic sources and a self-similarity estimator. The design and implementation of these two components and the framework that provides the infrastructure to support their functionality are the subject of the next chapters.

Chapter 3

Simulation framework

Simulation is a common part of network related research and a number of simulation tools developed to aid researchers are available. Network simulation tools are usually based on the event based simulation paradigm which is very generic and thus supports a wide class of simulation problems. However, event based simulation, being generic, does not take advantage of the specific features of the model under scrutiny.

In the following sections we will introduce event based simulation design concepts and propose modifications targeted to our network model and aimed at improving performance. The design that results from our modifications differs, from pure event based simulation engines, in that we distinguish two classes of simulated entities: *events* and *traffic quanta* instead of simulating exclusively *events*. *Traffic quanta* are processed in bulk, instead of singularly like in pure event simulation engines, resulting in shorter processing times.

3.1 Event based simulation

Computer simulations are usually run on frameworks that support a simulation paradigm called *event based*. In *event based* simulation systems everything happening at run time is modeled as an *event*. Events represent packet arrivals, simulation parameter changes, timed events and anything else that has a side-effect on the model. A side-effect is any modification to the state of the model including scheduling other events. Examples of side-effects are: changing a component's parameter like a packet source's rate, adding or removing a traffic source.

The basic infrastructure of an event based simulation engine is composed by an event queue data-structure and by an event dispatcher. The main execution loop of such system will retrieve the next event from the queue, set the simulation clock to the time of the event and dispatch the event to its handler. When the call to the event handler returns it is assumed all side-effects of the event are complete including the scheduling of new events; then the loop restarts form the beginning. The simulation continues until an event handler explicitly terminates the main loop or there are no more events in the queue.

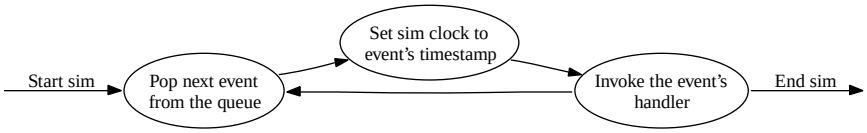


Figure 3.1: Event based simulator main loop.

Event based simulation is an elegant design because it is simple and effective in supporting a very large class of simulations. In practice event based simulation will work for any problem where the causality principle holds. That is, when any given event occurring at time T_0 can only effect the outcome of events occurring at a time T_1 with $T_1 > T_0$.

Event based simulation has the disadvantage of being computationally expensive. The assumption that every side-effect is modeled as an event results in simulation engine having to process separately a very large number of elementary operations. For instance, running a simple network model with a 100Mbit/s traffic source, a router and a sink connected by two links results in approximately 10 millions packet/events per second at the source, but since the packets are also received by the other four components of the network (two links, a router and a sink), the total number of events is five times the number of packets generated by the source: 50 millions per second. In general the number of events processed grows as a function of the number of components added to the model.

The large number of events involved causes the high computational cost of event based simulations because each event, even if elementary, needs to be handled separately and this adds a significant overhead. To explain how this happens we need to look at the main simulation loop described previously. In the simulation engine's main loop each event corresponds to a function call. Each call will result in the CPU storing the state of some registers upon call and restoring them upon return. The function called to process the event will have its own internal setup overhead due to sanity checks, configuration value lookups and any other operation required to prepare for processing the event. The function will then process the event applying side-effects to the model and return. So, for instance, a sequence of one hundred packets emitted by the same source and addressed to the same destination are represented by contiguous events on the queue; they will be processed one at a time: with one hundred calls to the same event handler function.

The obvious optimization consists in grouping the packets and invoking an handler capable of processing a sequence of events. Unfortunately this solution is not applicable except in some specific cases because it breaks the event based simulation model. In event based simulation *each* event's side-effect(s) must be complete before the next event is allowed to affect the model. Breaking this rule results in unreliable simulations because the side-effects of events being processed in one group are not visible to other events within the same block.

Taking advantage of the specific features of the network models used in this research, namely: tree structured network topologies and unidirectional traffic flow; we intend to design a simulation framework that can process traffic more efficiently without resulting in unreliable simulations.

3.2 Framework Design Goals

Our simulation framework aims at being used for the evaluation of DDoS scenarios beyond this research project for networks models under the constraints of Chapter 2. To be usable outside this research the simulation framework has to be modular and extensible. Extensible because it should be possible to add new components or different implementations of existing components, if required. Modular because it should be possible to extend the framework without modifying the existing components.

The simulation framework should provide the basic components required by our simulations and an infrastructure to connect them in a an useful fashion. It should also provide some means of controlling and monitoring the progress of a simulation and collecting the results.

3.3 Model Topology

Taking advantage of the specific characteristics of the problem we are addressing it is possible to simplify the design of the framework simulation.

First we introduce some terminology definitions:

- **model** is the abstract representation of the simulated system. In our case it is the network model described in Chapter 2.
- **topology** refers to the way entities that are part of the model are connected.
- **component** is any node or edge in a model's topology.

The topologies of the network models described in Chapter 2 can be always reduced to a tree structure, we thus define the entity *component* as a n-tree node (see Figure 3.2). This means that each component in the topology will have *n* incoming edges (traffic sources) and exactly one outgoing edge.

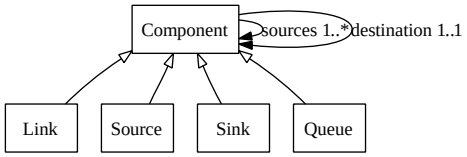
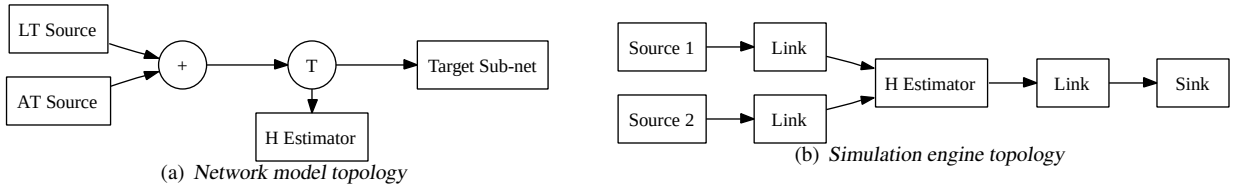


Figure 3.2: Component entity-relation hierarchy diagram.

Every block in a topology diagram is a specialization of the *component* entity. *Link*, *Traffic source*, *Traffic sink*, *Queue* are all examples of specialized *component* entities that represent blocks in a topology (see Figure 3.2).



(a) *Network model topology*

(b) *Simulation engine topology*

Figure 3.3: Example: Transformation from network model topology to internal simulation engine topology.

As an example application we now show how to represent a network model taken from Chapter 2 into an n-tree structure using blocks derived from the component entity. Starting from a block diagram representing a simple network (see Figure 3.3a) we replace all nodes and edges with their corresponding components. The transformation between the two is straightforward except for the placement of the Hurst parameter estimator *H*. The estimator is attached on its own branch in (see Figure 3.3a) while it is attached *in line* with the other components in (see Figure 3.3b). The estimator is moved *in line* with other components to achieve the required n-tree structure layout and the test point block is removed form the topology. Moving blocks from a test point branch to an *in line* layout is always possible thanks to the fact that all analyzer components are required by design to never alter the traffic flow, effectively making them act like a *null* passthrough.

Another noteworthy feature of the transformation is the absence of the traffic mixer block from the resulting tree topology. The traffic mixer block is omitted because its functionality is implicit in the definition of the *component* entity. Any component with multiple sources will automatically merge incoming traffic.

Applying the procedure described above, our restricted network models can be transformed into n-tree *component* structures. These n-tree structures (or component trees) have the property of reproducing the traffic flow from sources to destination when walked from leaves to root. Because of this property, component trees can be used as internal representation of a network model in a simulation engine.

3.4 Simulation Engine Implementation

The simulation framework takes advantage of the specific nature of the network model we are using for our simulations. In the model defined in Chapter 2, traffic flows unidirectionally from sources to destination and the topologies are trees with traffic generators as leafs and the target as root. This is not true in general for network simulations.

Unidirectionality of traffic flow and the tree topology properties of the model allow us to design a simulation engine that is more efficient than a strict event based simulation engine. Before we can introduce the optimizations we need to define the basic features of the framework's design. The main entities involved in our design are: *components*, *events* and *packets*:

- **Components** are the building blocks of the simulated model and equivalent to blocks in block diagrams. Components provide the functionalities required by the model, like traffic generators, queues, links.
- **Events** in this context has the same meaning as in the conventional event based simulation (see Section 3.1) except packets (or cells or frames) are not considered events but they are treated as a separate entity.
- **Packets** are in this context a conventional name for any indivisible unit of traffic. Depending on the type of network modeled by a specific simulation instance the word *packet* could mean an ethernet frame, an ATM cell, an IP datagram or others.

Our design of the simulation engine is based on the observation that the number of packets processed during a network simulation is larger by many orders of magnitude than the number of parameter-changing events. The distinction between *events* and *packets* entities in the list above is a result of this observation.

In a conventional event based simulation engine both packet arrivals and events are processed one at a time. In our design components process packets in blocks thus reducing the overhead by a factor that is in the order of magnitude of the logarithm of the block size. Events are still processed one at a time like in conventional event based simulation engines.

Some restrictions on event scheduling must be imposed to avoid unreliable simulation results. The restrictions are as follows:

1. Events can always be scheduled before the beginning of the simulation
2. Events can always be scheduled during the execution of an event handler
3. Events cannot be scheduled by traffic processing components

If necessary the last restriction can be relaxed into: “traffic processing components can only schedule events with a time-stamp greater then or equal to the one of the first event in the queue” without loss of reliability.

Without the above restrictions a traffic processing component could potentially schedule an event with a time-stamp smaller than an already circulating packet. The scenario for this situation is as follows: the topology in figure Figure 3.4 is used. Source 1 emits traffic up to T_e, source 2 emits traffic up to $T_2 < T_e$, the sink component will process queued traffic up to T_2. If the sink was to schedule an event *n* to alter the rate of source 1 with a time-stamp $T_2 < T_n < T_e$ then the event would not affect the traffic already emitted by source 1 for the time slice $[T_n, T_e]$.

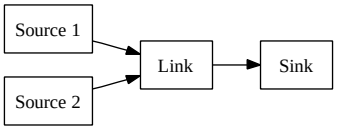


Figure 3.4: Example component topology.

The size of the traffic blocks processed by a component in our optimized engine is determined by the distance between two consecutive events. This is because under the restrictions listed above only events can have side-effects on the model. If during the period between any two events, simulation parameters cannot be changed then every component behaves uniformly during that period.

The main simulation loop in our optimized engine takes advantage of the restrictions on event scheduling by, at each iteration, invoking processing of all traffic between the current value of the simulation clock and the next scheduled event. The next scheduled event is called *event horizon* for the current processing block.

At each iteration all components in the topology are invoked passing the *event horizon* as a parameter. Each component will then generate, queue, filter or process traffic for the time slice starting at the current simulation clock up until the *event horizon*.

For instance, consider the simulation timeline in Figure 3.5 and let the simulation clock be 0.1s. According to the time line the simulation is in the middle of processing block *B4* and the next scheduled event is *E3*. No other event can be scheduled at this point, because only events can schedule other events, so processing will proceed undisturbed until the *event horizon* (*E3*) for the current block is reached.

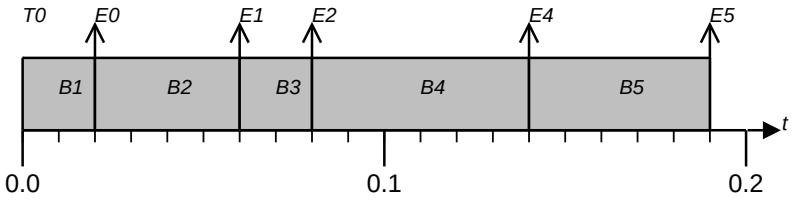


Figure 3.5: Example simulation timeline with events (En) indicated by vertical arrows and processing blocks (Bn) by grey boxes. *T0* is the simulation start time.

The order of invokation of the components is critical because components are connected in a topology and depending on their connections they can play the role of traffic source, traffic destination or both. Sources need to be called before their respective destinations to avoid breaking causality of packet propagation. In classic event based simulation topologies are graphs and thus an unique invokation order cannot be established. However, our restricted set of network models are strictly n-tree structures for which an order of invokation can always be found and is the reverse breath first order.

For instance, in Figure 3.3b the order of invokation for the components would be from left to right: Source1, Source2, Link, Link, H estimator, Link, Sink. This is because the reversed breath first order respects the order in which traffic flows from one component to the next.

Example 3.1 Pseudocode for the framework's main loop

```
def mainloop():
        #Prepare an ordered list of components to be invoked
        breathFirstList = treeOperations.breathFirst(componentsTree)
        reverseBreathFirstList = breathFirstNodeList.reverse()

        #The simulation starts at time 0
        clock.set(0)

        #The main loop exits when the simulation is finished or
        #when there are no more events to be processed
        while simulation.isFinished() or eventQueue.isEmpty():
                #The event horizon is the next event's timestamp
                eventHorizon = scheduler.nextEventTimestamp()
                #Invoke components in reverse breath first order
                for component in reverseBreathFirstList:
                        eventHorizon = component.process( eventHorizon )
                #Update the simulation clock
                clock.set(eventHorizon)
                #Execute the event with timestamp == eventHorizon, if any
                eventQueue.execute(eventHorizon)
```

Because of the n-tree structure of the simulated topology, each component can be in one of three roles: source, sink and passthrough. The three roles correspond respectively to: the leaves, the root and the trunk of the topology. For instance in Figure 3.3b: links, Hurst parameter estimators, rate limiting queue are examples of *trunk* components; Sink is a root component; traffic sources are leaves.

The implementation of a component is based on a different template determined by its role in the topology. The following pseudocode snippets are simplified templates for components implementations in all of the three cases outlined above.

Example 3.2 Pseudocode template for source component

```
def nextPacketTimeStamp():
        #implementation details determine the nature of the generator
        pass

def nextPacket():
        #implementation details determine the nature of the generator
        pass

def process( eventHorizon ):
        #Generate packets up until 'eventHorizon'
        while self.nextPacketTimeStamp() < eventHorizon:
                destinationQueue.push( self.nextPacket() )
        return min( eventHorizon, self.nextPacketTimeStamp() )
```

Example 3.3 Pseudocode template for passthrough component

```
def recordStats( packet ):
        #Optionally record stats for 'packet'
        pass

def modify( packet ):
        #Optionally modify 'packet'
        pass

def process( eventHorizon ):
        #Dequeue packets, process them then enqueue them, up until 'eventHorizon'
        while sourceQueue.nextPacketTimestamp() < eventHorizon:
                packet = sourceQueue.popNextPacket()

                #Optionally modify 'packet' or record stats before queuing it
                self.recordStats( packet )
                self.modify( packet )

                destinationQueue.push( packet )
        return min( eventHorizon, packet.timeStamp() )
```

Example 3.4 Pseudocode template for sink component

```
def process( eventHorizon ):
        #Dequeue packets up until 'eventHorizon'
        while sourceQueue.nextPacketTimestamp() < eventHorizon:
                packet = sourceQueue.popNextPacket()
                #Optionally record stats before dequeueing the next packet
        return min( eventHorizon, packet.timeStamp() )
```

3.5 Summary

Event based simulation is the most common design used by network simulation engines. The reason for its success is rooted in its simplicity and the broad class of problems tractable with it. We propose a modification to pure event simulation engine design to take advantage of the specific nature of our research problem.

Our modification hinges on distinguishing the *event* concept from pure event simulation into two distinguished classes: *events proper* and *traffic quanta*. The advantage of this distinction coupled with the simple tree structure of the network topologies we intend to simulate allows us to process *traffic quanta* in bulk instead of singularly.

We implemented the modification into our modular simulation framework which is used for the rest of the research.

Chapter 4

Self-similarity estimators

There are many estimators for self-similarity covered by literature, most of them surveyed by Taqqu et al. in [TTW95]. Estimators are algorithms that accept a time series as input and produce an estimate for the level of self-similarity as the output.

Most estimation algoritms are graphical in nature and were devised with small datasets (a few thousand observations) in mind. As part of our project we need to evaluate their execution cost against large datasets (in the order of tens of million observations) that are involved when analyzing network traffic.

Recorded network traces are a good candidate for testing estimators because they contain traffic captured on a real network. The traces are processed into a time series representing an arrival process model and then used to benchmark the estimator's behaviour.

The following sections will introduce the relevant concepts and report on the benchmarking of third party estimators and on our own implementation thereof.

4.1 Traffic traces

In these preliminary tests the estimators will be run against a series of traffic traces available for download from the MAWI project website [MAW08]. MAWI is an acronym for the *Measurement and Analysis on the WIDE Internet* and is part of the WIDE Project: *Widely Integrated Distributed Environment*. MAWI traces were chosen because they are readily available for download and they fit our requirement of high speed internet link traces.

Traces with daily traffic for the month of May 2007 recorded at *MAWI samplepoint-F* were downloaded. Each trace records all traffic flowing on a 100Mbit/s trans-pacific link for 15 minutes a day. The length of the traces for the month of May totals around 11Gbytes. This is an example of MAWI trace metadata available from the MAWI project archive [MAW08]:

```
DumpFile: 200705011400.dump
FileSize: 1748.31MB
Id: 200705011400
```

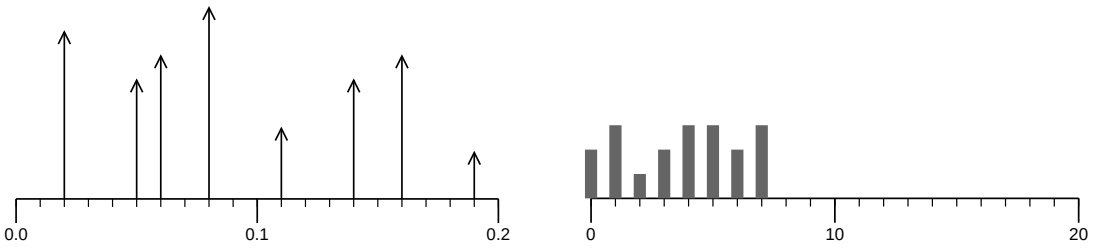
```
StartTime: Tue May 1 14:00:01 2007
EndTime: Tue May 1 14:15:00 2007
TotalTime: 899.17 seconds
TotalCapSize: 1367.78MB CapLen: 96 bytes
# of packets: 24937582 (16658.35MB)
AvgRate: 155.44Mbps stddev:11.07M
```

The traces are recorded using the pcap library format (pcap is short for packet-capture, see [pa08]). Pcap is a popular format and is recognized by a wide number of software tools. We use the *pycap* Python wrapper [Row08] for the pcap library to process traces in our software utilities.

MAWI traces contain packet arrival time-stamp, anonymized headers and no payload. While the payload is removed the information contained in the header is sufficient to derive the size of the payload. Payload size together with packet arrival information allows us to calculate the intensity of the traffic over time. Anonymized headers are not a limiting factor in the context of our research because self-similarity estimation does not require source and destination addresses or protocol specific flag information.

4.2 Traffic process models

The sequence of packets contained in the traffic traces cannot be used directly as input for the estimation algorithm, it must be pre-processed into a time-series first. There are various options for converting traffic traces into a time-series, each option is a different "arrival process model" and highlights specific properties of the traffic flow. For a discussion of possible arrival process models see [HKBN07]. In general the value of the self-similarity estimates depends on the process model applied to a trace. The reason for this dependence is that each process model highlights different dynamic properties of the traffic flow and is further discussed in [AAVV98].



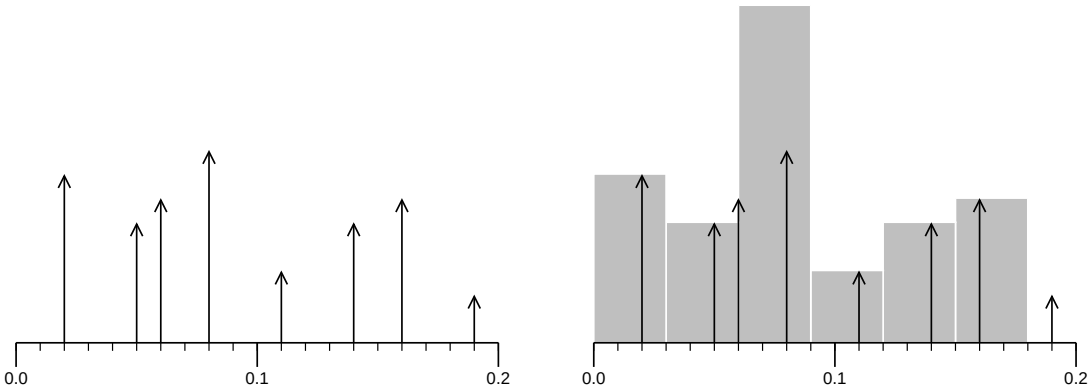
(a) *Packet arrival process. The position of vertical arrows indicates arrival time and their length indicates packet size.*

(b) *Packet inter-arrival process. The height of histograms indicates the time between two consecutive packet arrivals.*

Figure 4.1: Inter-arrival process (b) derived from the packet arrival process (a). The height of the histograms in (b) corresponds to the distance between two consecutive arrivals in (a). Note that the inter-arrival process is not plotted on a time axis but on an incremental packet arrival axis.

The *inter-arrival process* is the sequence of time intervals between packet arrivals. This process model is useful to analyze the distribution of arrival frequencies. If, for instance, five hypothetical packet arrival times in seconds were as follows: 0.3, 0.45,

0.51, 0.523, 0.55; then the inter-arrival process time-series would be: 0.15, 0.06, 0.013, 0.027. As it can be seen in the previous example: the inter-arrival process applied to an input trace of N packets will output a time-series of N-1 observations.



(a) *Packet arrival process. The position of vertical arrows indicates arrival time and their length indicates packet size.*

(b) *Packet aggregate sizes process. The height of histograms indicates the total data transferred in an aggregation period (period=0.4).*

Figure 4.2: Aggregate size process (a) derived from the packet arrival process (b).

The *aggregate sizes process* is obtained by establishing an aggregation period length and counting the number of octects or bits arrival in each period. An *octect* is a sequence of 8 bits and equivalent in length to a byte. The sequence of octects or bits count for each period is the aggregate sizes process time-series. The aggregation period length h must be selected carefully. If h is too small the process degenerates in a series of zeros and sparse positive counts, if h is too big the total number of aggregation periods in a finite trace will decrease reducing the number of available observations and consequently the precision of the estimator.

The inter-arrival process model will be used to reduce the trace to a time-series for testing the estimators. Later in the project we will use the aggregate-sizes process for our simulations.

4.3 Implementation

A simple program was implemented to apply the inter-arrival process model to the MAWI traces and convert them into a compressed text file. File compression combined with the little information required to represent the inter-arrival process results in a dramatic reduction in storage requirements.

The statistical software package *R* [fSC08] is used to run the first round of self-similarity estimator tests. A library called *fSeries* is available for the statistical analysis package *R* and contains functions for self-similarity estimation. Its source code is freely downloadable which makes it a good candidate for use in our project. The availability of source code allows us to assess the details of the implementation and, if required, improve on the existing algorithms.

The estimators available in the *fSeries* library for *R* are: R/S, aggregate variance method, differenced variance method, absolute values method, periodogram method, boxed periodogram method, Higuchi's method, Whittle estimator. Our preliminary tests consist of running all these estimators against the traffic traces and compare the resulting estimates.

We found that none of the estimators in the *fSeries* library could cope with the size of our network traffic traces, when run on computers available to us. The estimators ran out of memory and were ultimately terminated by the operating system even on systems with 4GBytes of physical memory. Only 3 to 4 million observations out of the 12 million contained in a typical MAWI 15 minutes trace could be processed before a crash invariably occurred.

Access to the source code for R's libraries proved useful to understand the nature of the problem. Auditing the source code it appeared that R's internal representation of floating point numbers coupled with frequent array copy operation and failure to release allocated memory were the cause of the problem.

R uses only double precision floating point format (64bits or 4bytes) to represent numbers internally. A MAWI traffic trace contains a number of packets that exceeds 20 millions; the number of observations in the inter-arrival time series is of the same order of magnitude which results in approximately 80 Mbytes of memory storage for each copy of the dataset. *R* operates in such a way that array operations, which are ubiquitous in the implementation of the estimation algorithms, result in allocation of multiple copies of the input dataset. R's internal memory management system does not promptly free memory allocated to the arrays ultimately causing the process to be terminated before completion.

A first attempt at solving this issue entailed analysing each estimator's implementation and removing redundant operations that would adversely impact memory allocation. Although we were successful at reducing the memory footprint when executing the estimators, the reduction proved to be insufficient to solve the original issue. Nonetheless significant improvements to the existing implementation of R's library and a good understanding of the estimation algorithms were achieved.

Leveraging the knowledge of estimation algorithms gained during the first attempt we rewrote a subset of the estimators in a general purpose language. The language of choice for this effort was Python [vR08] and the mathematic library used was *numpy* [Oli08]. Our implementation proved capable of handling MAWI traces in their full length.

4.4 Testing the implementation

A re-write of the algorithms contained in R's *fSeries* library could easily result in an incorrect implementation. While speed and memory consumption are important, insuring the correctness of the results is paramount.

An automated system was implemented to compare results from our library with results from *R* for all newly implemented estimators. The automated tester program launches estimation of the same series in both *R* and our version of the algorithms and then compares results. The test was repeated for a number of distinct traffic traces. For each trace tests were launched against series of lengths ranging from 200 observations to about 1 million observations in steps of 100 observations. All of the estimator functions passed the tests returning results that matched the expected values. These test results confirmed our confidence in the correctness of our implementation relative to the reference library *fSeries*.

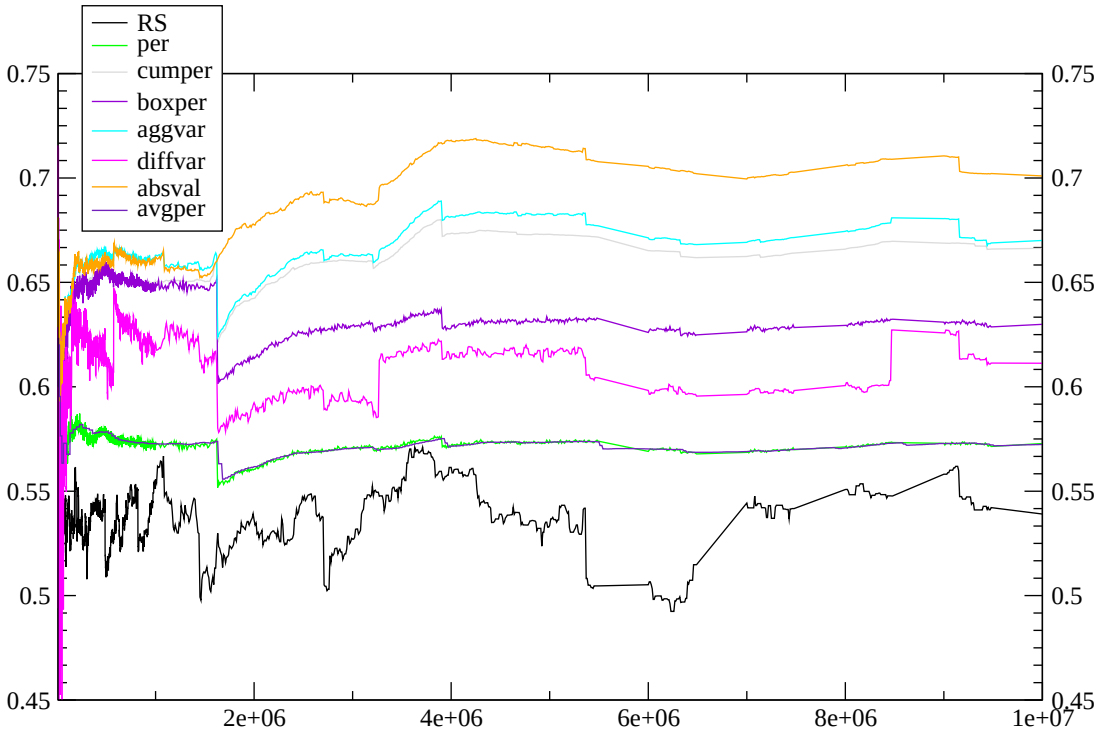


Figure 4.3: Hurst parameter value estimated using different methods against the same traffic trace. The x axis represents the number of observations used for the estimation, counting from the beginning of the trace.

4.5 Online estimation

The self-similar property of a time-series is defined for unlimited stochastic process and applies to the time-series as a whole. In practical applications, time-series are of finite length, still, if the length of a series is large enough, it is acceptable to talk about self-similarity of a finite series.

In our research we are interested in *online* monitoring of network traffic. This means that our self-similarity estimators need to provide an approximation of the Hurst parameter for the traffic that is currently flowing through the estimation block.

An online estimator has two main features: the algorithm has to process new data as it submitted without requiring re-computation on the whole dataset and it has to be designed in such a way that computation on a newly submitted block of data is finished before the next block becomes available. Such algorithms are further discussed in the following sections.

4.5.1 Hurst parameter for current traffic

While the notion of *current* Hurst parameter value does not have any meaning even when relaxing the definition of self-similarity for finite series, it can be assumed that *current* in this context means *related to recent traffic*. In this sense the value of H at time t (H_t) is estimated, for the S series, using the finite subseries S_x: $t\text{-}l<T(x)<t$; where $T(x)$ is the time-stamp of the observation x and l is the *lag* or delay of the estimate. In other words, the last l seconds of traffic are used to estimate the value of H_t.

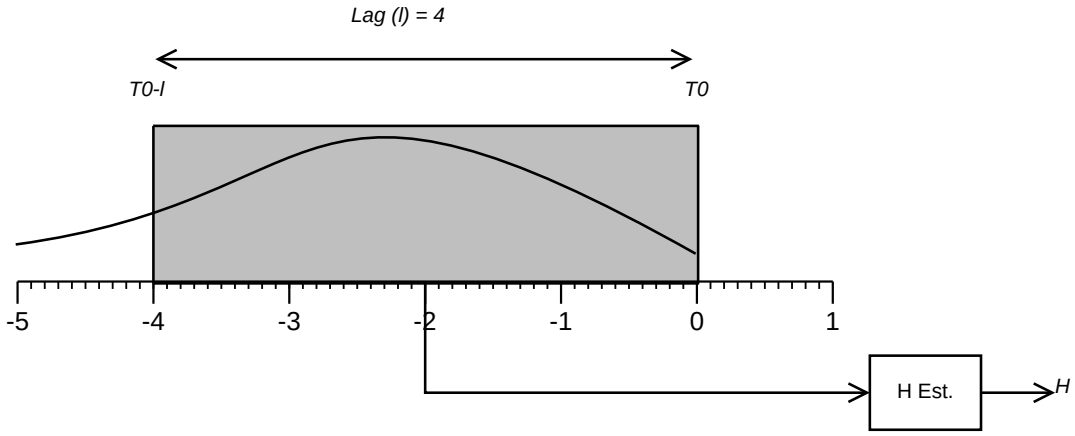


Figure 4.4: Online estimation operates by taking into consideration only the last l seconds of the traffic flow.

The choice of l is mainly guided by two factors: quality of the estimate and acceptable delay. The quality of the H estimate depends on the number of observations available in the time-series. The larger is the number of observations the smaller is the estimation error. The maximum acceptable delay depends on the specific application, but usually cannot be longer than a few seconds because the delay determines the lower bound for the reaction times of any algorithm based on the result of estimation. l should be long enough to result in a small estimation error and short enough to make the results of the estimation useful.

To guide the choice of l it is useful to know how many observations are available per time unit. The number of observations available per time unit depends on the traffic process model (see Section 4.2) chosen for the estimation. The inter-arrival process model will produce a number of observations proportional to the average intensity of the traffic. The aggregate sizes process results in a number of available observations that depends on the duration of the aggregation period duration.

4.5.2 Averaged Hurst parameter

The trade-offs involved in the design of network devices are such that strict constraint are imposed on computational and memory resources. It is crucial to reduce resource consumption to a minimum for the implementation of self-similarity estimation based techniques in such devices.

Even thought the series to be stored in memory is bounded by the choice of the estimation lag l, the number of observations collected in l seconds can still be very large on high speed links. For instance, on a 100Mbit/s link the number of observations for the inter-arrival process series is in the order of 10^6 per second. The execution time of estimator functions depends on the length of the input and for most estimators it grows faster than O(n), where n is the number of observations.

To reduce the memory required by the estimation procedure the series of length l can be divided in K equally sized blocks. then H^* is defined as the mean of H_k over the last K blocks. The memory required to calculate H^* is, at any point in time, a constant proportional to the number of blocks K.

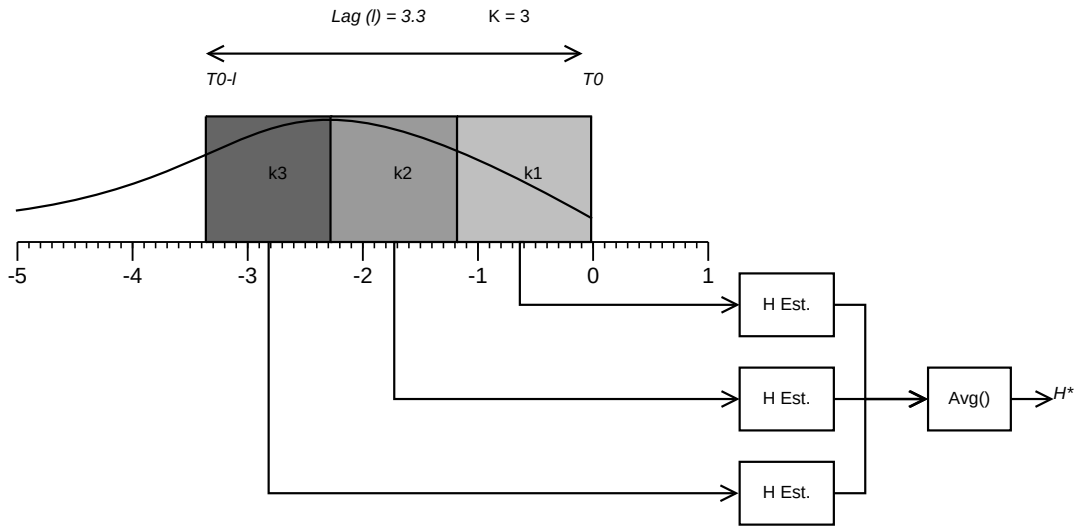


Figure 4.5: Averaged Hurst estimator operates by averaging the H estimates for the last K blocks of the traffic flow.

The non-overlapping block mean estimate H^* is used in [Li06] to detect variation of H in changing traffic conditions and a variation, based on a weighted average, is described in [HKBN07] Section 3.5. The justification for its validity as estimator can be found in [VA99a] Part I Sections D and E.

4.5.3 Averaged periodogram estimator

Using H^* as described in Section 4.5 to reduce estimation memory requirements to a constant is applicable to any estimator algorithm regardless of their design. While a generally applicable solution is very useful, it is also interesting to discuss modifications to the same effect that apply to specific estimators. We stumbled upon such modification for the periodogram method while re-implementing it. However, we make no claims as to the novelty of this modification, as a matter of fact a discussion of a similar modification to the periodogram estimator can be found in [FA96] and in [PT97].

The standard periodogram estimator is based on a spectral analysis. The algorithm is quite straightforward (see [TTW95] Section 3.7): implementations calculate the power spectrum density (PSD) of the input time-series then employ linear regression on the lower 10% of the frequency spectrum log-log plot excluding the DC component. The slope b of the linear regression determines the Hurst parameter with the relation $b = 1\text{-}2H$.

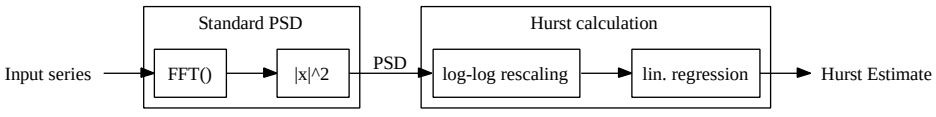


Figure 4.6: Standard periodogram estimator block diagram. Blocks are grouped to highlight the two stages of the estimation algorithm: PSD estimation and Hurst calculation.

The algorithm con be split in two parts independent of each other. The first part is the signal's power spectrum density estimation, the second is the Hurst parameter calculation. This modification to the periodogram estimator is based on the fact that substituting

the first part of the algorithm with a different, but equivalent, power spectrum estimation technique will not affect the validity of the estimation result. However, the change of the power spectrum estimation method will influence the properties of the overall estimator method.

We recall the goal of the modification is to reduce the consumption of memory and computation resources at runtime. In order to do so, the first part of the estimator is substituted by the averaged power spectrum method due to Welch (see [Wel67]). Welch's method consists in dividing the input time-series in K potentially overlapping blocks then multiply each block with a window function and compute the power spectrum density of the result using the Fourier transform. The resulting K power spectrums are then averaged.

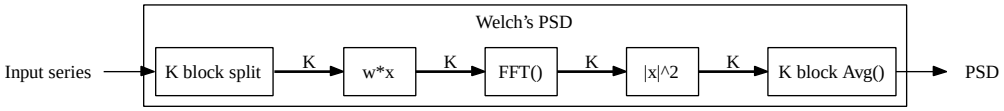


Figure 4.7: Welch PSD estimation block diagram. This block can be used to substitute the standard PSD stage in Figure 4.6.

The result of Welch's method is then used directly as an input for the Hurst parameter calculation stage of the algorithm. This modification has the advantage to require little changes to existing software, in fact, the standard and Welch's power spectrum density estimation methods can be swapped in place.

Resource consumption of this modified periodogram method are lower than its standard counterpart. The memory required to to store the power spectrum estimates is a constant dependent on the number of blocks K. The computation of the PSD for the K blocks can also be executed in parallel if deemed advantageous.

Using the averaged periodogram method influences the properties of the Hurst parameter estimator. Averaging the PSDs of overlapped blocks conserves the expected value of the estimates but reduces the variance of the estimates asymptotically by a factor inversely proportional to the number of blocks K. The effect can be seen in Figure 4.3 comparing the plots of the standard periodogram and the averaged periodogram (respectively *per* and *avgper*). The two plots overlap, but the *avgper* plot is smoother and less subject to jumps.

The averaged periodogram was implemented and tested and is now part of our library of estimators.

4.6 Distributed computation

The next issue to overcome in preparation for our final simulation is estimation speed. While our implementation performs well, mainly thanks to the fact that the entire dataset always fits in physical memory, we predict that the number of repetitions needed by our final simulations coupled with the execution time of the estimators, some of which have computation complexity that is $O(n^2)$ or $O(n^3)$, may result in an unacceptably long total execution time.

To reduce computation time we decided to leverage distributed computation. We attempted, and failed, to install third party distributed job control applications on the department managed lab computers because of operating system environment incompatibilities. So we were forced to develop our own simple job scheduler and distribution application. Based on a client server

architecture where the clients are constantly in a job execution loop either querying the controller for the next job or executing the current one and the server schedules jobs and collects statistics and logs results from clients.

To test our implementation of the estimators together with our job distribution system we ran 8 estimators on a single MAWI traffic trace and plotted the results on the same graph (see Figure 4.3).

4.7 Wavelet Estimator and Multiresolution Analysis

Of all the methods we tested, the wavelet estimator due to Abry and Veitch (see [AAVV98] and [VA99b]) is used in the rest of our research. The reason for preferring the wavelet estimator is its intrinsic multi resolution analysis feature. This estimation method leverages the properties of discrete multiresolution wavelet decomposition to approximate the value of the Hurst parameter.

Informally, wavelet decomposition divides a signal into a number of wavelets. Each wavelet that is part of the decomposition of the original signal is a scaled and translated version of a so called *mother wavelet* function. Wavelet decomposition has the advantage, over the classic Fourier signal transform, of being more suited for analysis of finite and non periodic signals like time-series derives from network traffic.

Multiresolution analysis refers to a signal analysis technique that operates over multiple scales; the concept is strongly related to wavelet decomposition (see [Mal89]). The signals we study are time-series so the term “multiresolution”, in our context, refers to multiple timescales.

The Hurst estimator based on wavelet decomposition is the natural choice for performing MRA because its implementations yield decomposition coefficients for approximately $\log_2$(N)-1-k scales, where N is the length of the input series and k depends on the order of the mother wavelet function used for the decomposition and the intensity of *border effects* that reduce the number of available scales. The decomposition coefficients for each scale are then used to calculate a signal power estimate for the corresponding scale.

We use multiresolution analysis (or MRA) to verify the presence of self-similarity or evaluate the quality of a self-similar traffic source. The result of applying MRA to a time-series can be visualized as a log-log plot (see Section 1.2 for details and Figure 1.5 for an example) of the power estimate of the wavelet decomposition coefficients at different time scales. The horizontal axis of an MRA log-log plot displays the time scales; time scales are also known as *octaves* in the context of MRA because the time base doubles from one scale to the next one up. The MRA log-log plot can be used to verify if a time-series is self-similar and subsequently to estimate the value of the Hurst parameter.

We recall that formally self-similarity is a property that spans all scales and, in the ideal case, the value of the Hurst parameter estimated at different time scales should be the same. This means that, in ideal conditions, all points of the MRA log-log plot lie on a straight line.

We want to verify that the output of the generator is, at least form a practical point of view, self-similar. While the formal definition of self-similarity requires the Hurst parameter to be the same across all scale, practically we accept a generated time-series as being self-similar when its Hurst parameter is almost constant over at least 5 adjacent scales. The choice of 5 scales is a pragmatic one. The minimum number of scales necessary to estimate a value of for H is 3. 5 scales is the result of extending this minimum interval of scales by one scale in both directions.

4.8 Summary

Self-similarity estimator methods are algorithms that accept a time series as input and produce an estimate for the level of self-similarity as the output. Most estimation algoritms are graphical in nature and were devised with small datasets (a few thousand observations) in mind however, analyzing network traffic requires processing large datasets.

Captured traffic traces represent good testing datasets and are thus used for benchmarking the estimators. Traffic traces require processing before being used as input to estimation methods. The processing discards most information contained in the traffic traces except for the arrival times and sizes of data packets. Sizes and arrival times extracted form the traces are used to output a time-series that represents either the inter-arrival process or aggregate sizes process.

We tested existing implementations and found the could not cope with the data sizes involved when analysing network traces and proceeded to re-implement the estimators with large datasets in mind. We tested our implementation of the estimators thoroughly to make sure they were correct and developed a set of simple utilities for distributed computation to support our benchmarking effort.

After testing the estimators and evaluating a modification to the periodogram method, we finally settled for using the wavelet based estimator for our research because of its multi resolution analysis features.

Chapter 5

Traffic generation

Synthetic traffic generation is a wide area of research concerned with the theory and design of algorithmic generation of network traffic. This area of research is driven by the need to simulate or emulate network conditions that are not easily reproducible or for which existing traces are either not available or not easy to obtain.

Network traffic generation is, in purpose and methodology, not unlike test signal generation used in other engineering disciplines. In both cases a synthetic signal, designed to have certain characteristics, is fed into a system as input and its effects are observed. Since the nature of the test signal is known, the effects observed during the tests are more easily explained as a function of the model and the test signal.

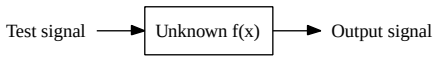


Figure 5.1: Test signals like synthetic traffic can be used to map the unknown behaviour of a system.

Traffic synthesis algorithms, like test signals, are usually application specific: they are designed to fulfill the needs of a specific research case. As a result, each traffic generation algorithm will reproduce specific features or properties of network traffic and lack others.

5.1 Self-similar Traffic Generation

Our network model (see Chapter 2 for details) requires two independently controlled traffic generation blocks (see Figure 5.2, reproduced here form Section 2.4). The two generators must be capable of synthesizing traffic that exhibits self-similar characteristics. Also, to be useful for our simulations, each generator must also, at a minimum, expose two operational parameters: traffic intensity and Hurst parameter.

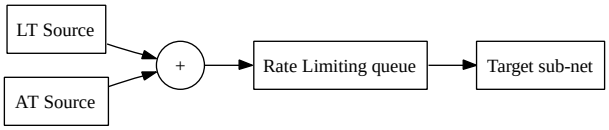


Figure 5.2: The complete network model. The two traffic sources for legitimate and malicious traffic (respectively labelled 'LT' and 'AT' in the diagram) are visible on the left.

Our interest is restricted to generation algorithms capable of approximating the self-similar characteristics of network traffic. Self-similarity emerges in various aspects of traffic traces: aggregated sizes process, count process, packet inter-arrival time-series to name some (see [AAVV98] and [HKBN07] for an overview of possible network traffic process definitions). Algorithmic generation does not, in general, reproduce all of the above. As a result the choice of generator algorithm and estimation process depend on each other.

In this research we use the two most common estimation processes: aggregated sizes process and inter-arrival time process.

Aggregated sizes process

Once a sample period is chosen, the aggregated sizes process consists in the time-series of octect count (or bit count) for each sample period. In other words for each period we count the amount of data that is transmitted and at the end of the period we output the result and reset the counter.

Inter-arrival process

The inter-arrival process, unlike the aggregated sizes process, does not require a sample period to be chosen. It simply consists of the time series of the time-distance between the arrival of each packet and its successor.

The aggregated sizes process and inter-arrival time process require only packet time-stamp and packet size information to be implemented thereby restricting the amount of information our traffic generator needs to reproduce. For instance source and destination addresses, packet header flags, payload are irrelevant to our investigation. This is not only an advantage because it simplifies the implementation of the generator but it is also the very reason why using self-similarity estimation is practically attractive.

5.2 Online Generator Fitness Analysis

Our choice of the "alternating on-off sources" generator described by Taqqu et al. in [TWS97] is guided by two main factors. First we prefer online generators to offline ones because the former allow us to reach higher level of confidence in simulation results; and second the results from a recent empirical comparison by Horn at al. in [HKBN07] strongly suggest the generator described by Taqque et al. is both simple in design and accurate in synthesizing self-similar traffic.

The *offline* category describes traffic generators that for technical reasons are required to run prior to the simulation. The length of an *offline* traffic trace is determined before the simulation starts at the moment of its creation. The length of the trace ultimately limits the execution time of the simulation. If the trace is shorter than necessary a new one needs to be generated and the simulation repeated. To avoid this undesirable situation, a trace longer than the estimated required length can be generated. In

some cases a precise estimate of the required length is not possible and finding a suitable length for a trace becomes a time consuming trial and error exercise.

Online traffic generators produce traffic output incrementally as the simulation proceeds and allow to run a simulation for an unconstrained amount of time. It is desirable to not have an upper bound on simulation time, because it allows to achieve higher confidence on the results' correctness.

The next sections describe the design and implementation of a multiplexed on-off online packet generator based on Taqqu's work.

5.2.1 Multiplexed on-off sources generator

The multiplexing on-off sources generator works by superimposing traffic generated by many independent and strictly-alternating on-off sources.

Alternating on-off sources are part of a broader family of generators called "packet-train" generators. A packet-train generator (see Figure 5.3) emits packets during its activity (or "on") periods and does not emit packets during idle (or "off") periods with no further restrictions on periods duration.

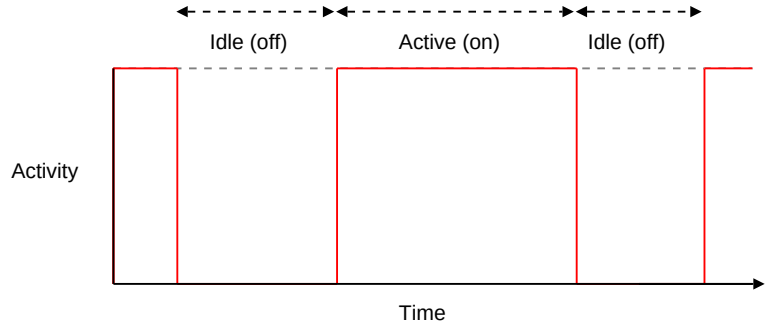


Figure 5.3: Example Time-Activity graph for a packet train generator.

A strictly-alternating on-off source emits a packet train with the following restrictions: the "on" and "off" state strictly follow each other and cannot be of nil duration, which means that a period of activity is always followed by a period of idling which is always followed by another period of activity and so on.

The on-off period durations in each source for the multiplexing generator are Pareto distributed. In general the shape and scale parameters for the "on" and "off" Pareto distributed periods can be different. Also the shape and scale parameters can be different for each single source. The other parameter that governs the behaviour of the on-off source is the rate at which packets are transmitted during the "on" period.

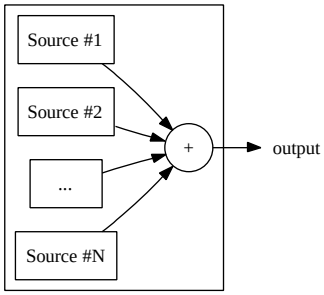


Figure 5.4: The multiplexed on-off generator internally aggregates traffic from N strictly-alternating on-off sources.

Assuming, for simplicity's sake, all packets emitted have the same size and the content (header and payload) of the packet is not relevant, then for a multiplexing generator with N sources the number of parameters involved is (2*2 + 1)*N: shape and scale for "on" and "off" period plus packet emission rate, for each on-off source. The number of parameters is reduced to only 3 (Pareto shape, Pareto scale and packet emission rate) when, following the approach suggested in [TWS97] and tested in [HKBN07], the same value for packet emission rate, "on" period and "off" period distribution is used across all the alternating on-off sources.

When the generator is setup with the reduced set of three parameters then the self-similarity of the resulting generated depends on the Pareto shape parameter, while the average traffic intensity depends on the Pareto scale and packet emission rate.

5.2.2 Implementing the generator

The implementation of the multiplexed on-off generator in our simulation framework is based on the reduced parameter design explained in the previous section. The first of the three parameters is the number *N* of independent alternating on-off sources that are used by the generator. Each one of the *N* on-off packet train generators is then initialized with the same two values for the Pareto shape and emission rate parameters. The implementation is based on the *source* component template introduced in Section 3.4 and its algorithm is based on a simple loop: when invoked to generate traffic for a given time-slice the generator will iterate all *N* sources and instruct them to generate traffic for the current time-slice.

Example 5.1 Pseudocode for the multiplexed on-off sources generator

```
def process( eventHorizon ):
        #Generate packets up until 'eventHorizon'
        while self.nextPacketTimeStamp() < eventHorizon:
                for onOffPacketTrain in sources:
                        eventHorizon = onOffPacketTrain( eventHorizon )
        return min( eventHorizon, self.nextPacketTimeStamp() )
```

Each of the on-off sources will simply alternate between the *on* and *off* state and emit packets at the rate prescribed by the packet emission rate parameter as explained in the previous section.

5.2.3 Testing the generator

After some preliminary tests using MRA to evaluate the quality of the multiplexed on-off sources generator we realized we could not reproduce the results published in Section 5.2 of [HKBN07].

The presence of bugs in our implementation was ruled out by running the same tests on a different implementation of the same generator. These additional tests confirmed our results indicating that bugs were less likely to be the cause of the discrepancy.

We then decided to run extensive tests to map the effect of the generator's parameters on its output. During preliminary tests it was found that the parameters influencing the quality of the output are: Hurst parameter (controlled by the Pareto shape of the on-off periods), the load level (ratio between Pareto scale for the *on* period and Pareto scale for the *off* period) and the number of sources.

Mapping the quality of the generator's output depending on the value of the three parameters listed above using MRA can be achieved by *pinning* one of the parameters at a time and run multiple tests changing the values of the remaining parameters. Table 5.1 lists, for each of the three parameters, the range of values used in the tests.

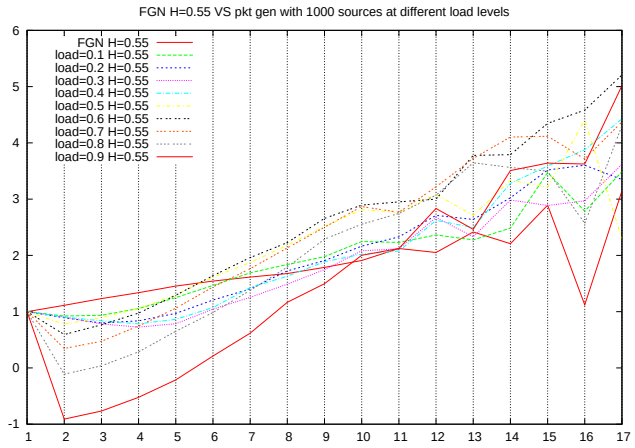
Parameter	**Range**
H	[0.55, 0.95]
Load	[0.1, 0.9]
Number of sources	[100, 10000]

Table 5.1: Multiplexed on-off generator test parameters

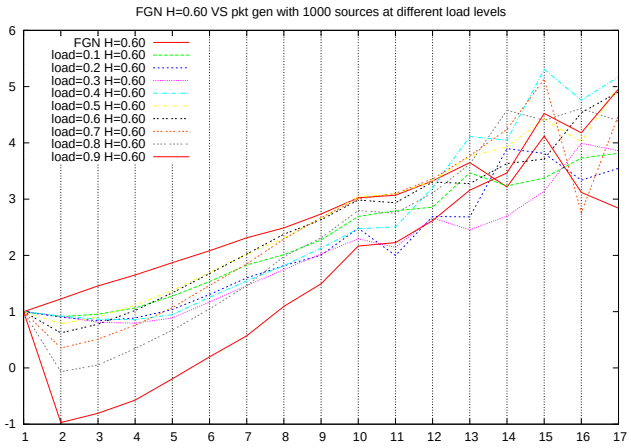
Results of the *H vs load* level tests for 1000 sources are potted in Figure 5.5, Figure 5.6. *H vs load* plots feature a reference fGn trace as a reference plot of self-similar time-series. Ideally the slope of the each line in the plot should be close to the slope of the fGn line. It is apparent from the plots that the load level does affect the slope of the MRA lines. The influence of load level on the quality of the output is more intense for lower values of the Hurst parameter. Independently of the load level, the slope of the MRA lines is consistently different from the expected value (the fGn reference) except for plots where the value of H is close to 0.70 indicating that the quality of the generator is not homogeneous over the range of neither Hurst or load level parameter.

Results for *H vs number of sources* tests with a 0.25 load level are plotted in Figure 5.7. This set of plots highlight the relationship between number of sources, H and the quality of the output. These plots, like the previous ones, feature a reference fGn line. The results indicate that the number of sources does affect the quality of the generator, but does so in a marginal fashion at least in the range of 100, 1000 and 10000 sources chosen for these tests. These plots confirm the influence of the H value on the generator's output quality: the slope of the generator's MRA lines is closer to the slope of the fGn line for values of H in the midrange.

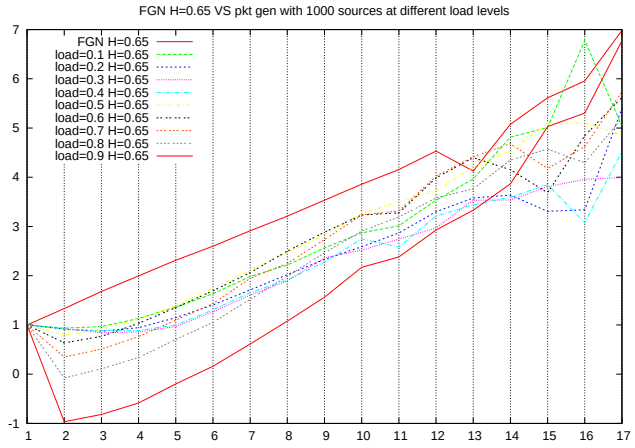
From the results of these tests we conclude that the multiplexed on-off generator is not well suited for our simulations because of its non homogeneous output quality over the range of values for the Hurst parameter.



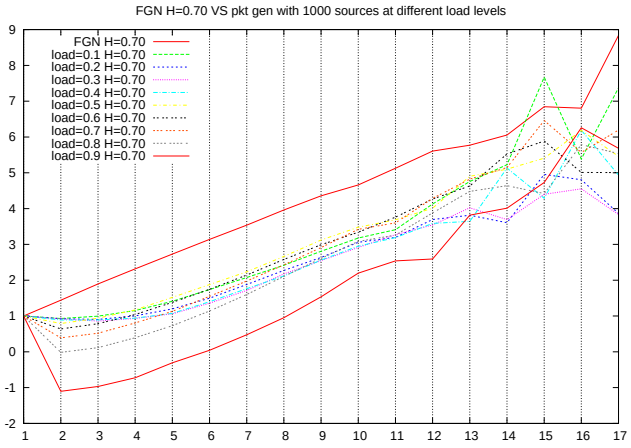
(a) *H = 0.55*



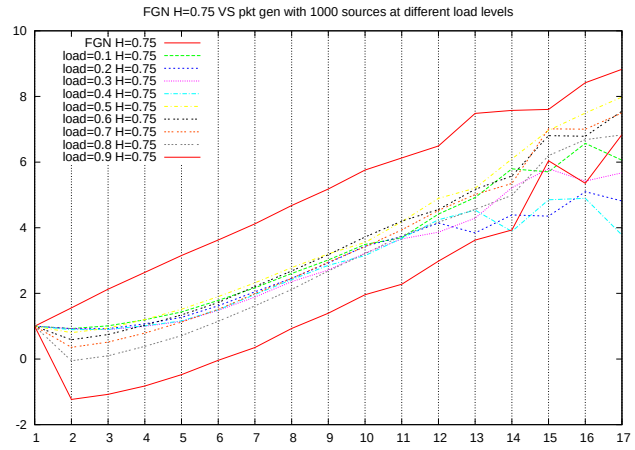
(b) *H = 0.60*



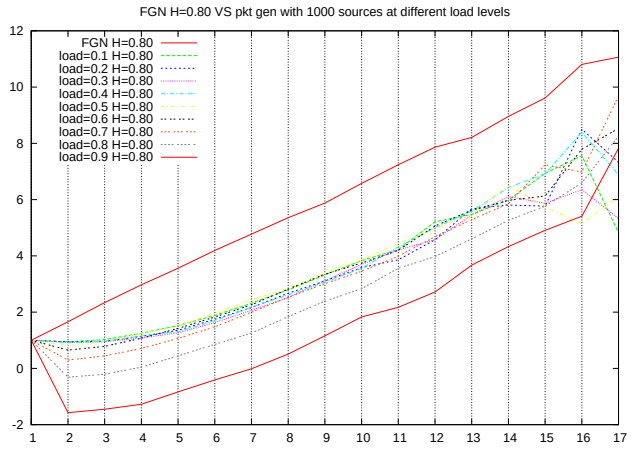
(c) *H = 0.65*



(d) *H = 0.70*

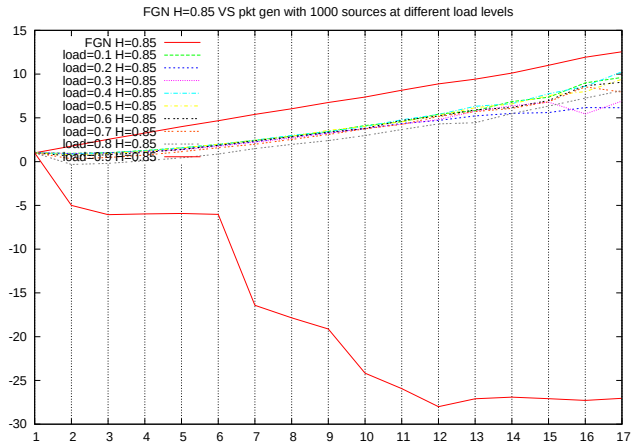


(e) *H = 0.75*

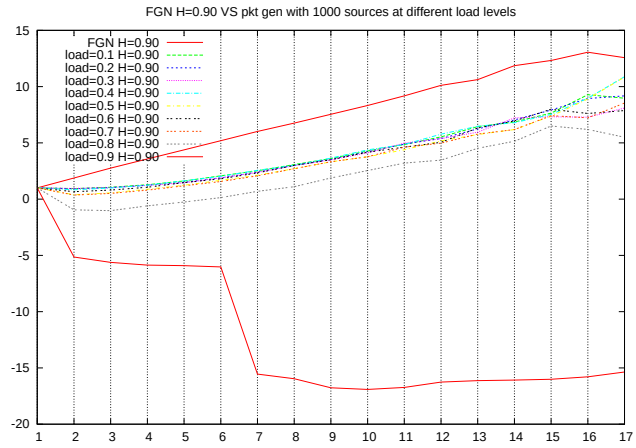


(f) *H = 0.80*

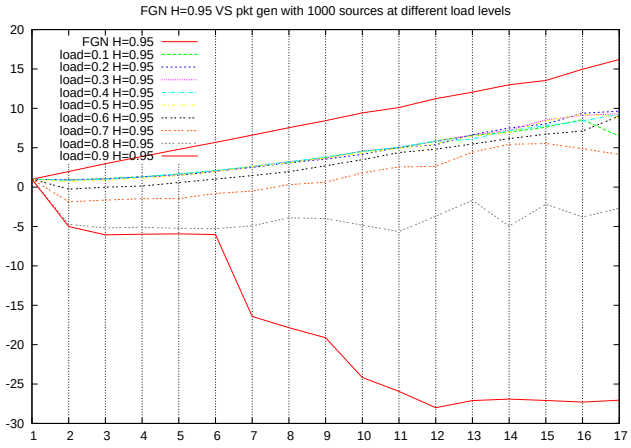
Figure 5.5: MRA plot of fGn versus generated traffic with load = (0.1, 0.9) and number of sources = 1000.



(a) *H = 0.85*

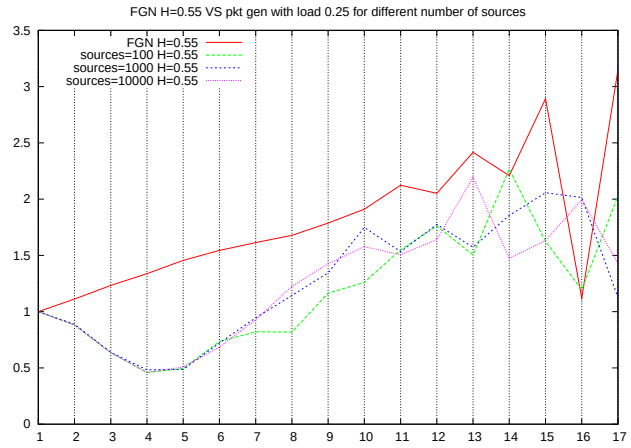


(b) *H = 0.90*

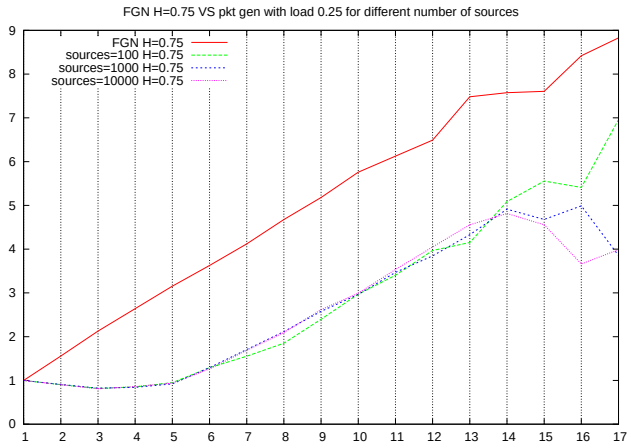


(c) *H = 0.95*

Figure 5.6: Continued from Figure 5.5. MRA plot of fGn versus generated traffic with load = (0.1, 0.9) and number of sources = 1000.



(a) *H = 0.55. 100, 1000, 10000 sources.*



(b) *H = 0.75. 100, 1000, 10000 sources.*

(c) *H = 0.95. 100, 1000, 10000 sources.*

Figure 5.7: MRA plot of fGn versus generated traffic with Hurst = 0.55(a), 0.75(b), 0.95(c) and load = 25%

5.3 Fractal Gaussian Noise Generator Fitness Analysis

Fractal Gaussian noise (also known as fractional Gaussian Noise or fGn) is one of the best known classes of stationary processes with slowly decaying correlation and is, by definition, self-similar. Implementing a traffic generator using an fGn source is a safe choice for obtaining “high quality” self-similar traffic. Nonetheless the transformation of an fGn time-series to the aggregated sized process time-series may degrade its quality. To ensure the self-similarity is preserved we test our generated traces using multi resolution analysis (see Section 4.7).

5.3.1 Implementing the generator

Our requirements for the generator interface are that it exposes two parameters: traffic intensity and Hurst parameter. The generator derives its self-similar properties solely from the fGn noise source which will be programmed to produce an output

time-series with a specific value for the Hurst parameter.

We use the *fgnSim()* function from the *fSeries* library of the *R* [fSC08] statistical analysis package to generate fGn time-series. The *fgnSim()* function uses a spectral method based on [Pax97] to generate its output; the function's parameters are: the number N of observations in the output time-series and the value H of the Hurst parameter. Optional parameters control the mean (defaults to 0) and standard deviation (defaults to 1) of the generated fGn.

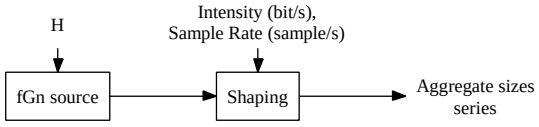


Figure 5.8: Two stages of the fGn traffic generator. The first stage outputs fractal Gaussian noise. The second stage shapes the noise into an aggregate sizes traffic process with the given average intensity and aggregation period equal to the reciprocal of the sample rate.

We are then left with the problem of converting the fGn time-series into a traffic trace and controlling its output traffic intensity. Our traffic traces are still time-series, but they represent the aggregated sizes traffic process. The simplest way of *converting* an fGn time-series to a traffic trace is to interpret each fGn observation as the intensity of the traffic for a time period. Traffic intensity is an integer number representing the total number of bits, octects or packets transferred in the chosen time-period.

As we mentioned above the *fgnSim()* function has an optional parameter to set the mean value of the fGn output. Simply setting the mean parameter to the desired average intensity for the traffic source and rounding the result to the nearest integer would result in an output time-series with the characteristics we require. For the sake of modularity we decide against this approach and prefer to use the *fgnSim()* function with its default mean of 0 as a generator block. We then attach a post-processing block to the output of the fGn source (see Figure 5.8).

The fGn post-processing block treats each value in its input time-series as the total amount of data transferred in a period in accordance with our choice of aggregate size traffic process. Each observation is normalized with the value of the average intensity parameter and rounded to closest smaller integer value. The rounding process alters the statistical properties of the series, but we anticipate it will not affect our results because the value of the average intensity parameter is, for meaningful scenarios, never lower than 10^6 thus making the quantization error smaller than 10^{-6}.

5.3.2 Testing the generator

We know, from our previous tests on the online generator, that the output of the *fgnSim()* function is self-similar over a wide range of scales. This is apparent when looking at the reference fGn plots in both Figure 5.5 and Figure 5.7. In all plots, across values of H form 0.55 to 0.95, the slope of the fGn line appears constant (on a log-log scale) for at least the scales between 1 and 11, confirming our expectations.

In order to test that the self-similar properties of the fGn time-series are not altered by the conversion into an aggregated sizes traffic process we proceeded to compare the MRA plots of the fGn before and after being processed by the second (*shaping*) stage of the traffic generation process (see Figure 5.8).

In the tests multiple fGn time-series of 2^{20} observations each are generated using the *fgnSim()* function. Each time-series is generated with a different value for the Hurst parameter in the interval from 0.55 to 0.95 in 0.05 steps. The resulting nine time-series are processed by the *shaping* stage to obtain nine self-similar traces. Each trace and its corresponding fGn time series are run through MRA and their plots compared.

The plots do not show significant difference between the quality of fGn and the output of our generator. In fact the MRA plots of the fGn time-series and the relative traffic trace overlap to the extent that they appear as a single plot. We conclude that our implementation of fGn traffic generator is an acceptable source of self-similar traffic traces.

5.4 Summary

Our first choice for the traffic generator block was the "alternating on-off sources" generator described in [TWS97] and empirically compared to other generators in [HKBN07]. Section 5.2 describes the reasons for preferring an online generator and the subsequent analysis of its fitness.

After carrying out a thorough analysis of the "alternating on-off sources" fitness, we concluded that the generator, although promising, was not suited for use under the conditions we are interested in reproducing in our simulations. We subsequently decided to turn our attention towards an offline traffic generator. Using a well known source of self-similar time-series like fractal Gaussian noise (or fGn) appeared to be a safe choice (see [Pax97]). In Section 5.3 we report our findings about the fitness of fGn based traffic generator for our simulations.

Briefly, we found that our implementation of the fGn based traffic generator outputs good quality self-similar traffic over a the required range of values for the Hurst parameter (0.5, 1.0) and over wide range of time-scales thus meeting our requirements.

Part III

Results

Chapter 6

Model Validation

Before proceeding with our investigation of DDoS detection, it is necessary to verify that our network model respects the requirements we highlighted in Section 2.1. While in Chapter 2 we moved from requirements to design, justifying our steps via reasoning and referencing existing literature, in this chapter we intend to "close the loop" by defining simulations which show that our model reproduces the expected self-similarity related behaviour.

Simulations are computationally intensive tasks. The complete investigation of a model via simulation requires repeating runs with all possible combinations of values for all of the model's parameters. The number of such simulation-runs grows quickly with the number of parameters, in a combinatory fashion, beyond practical acceptable limits.

A commonly used technique for reducing the parameter-space to an acceptable size, thus reducing the number of simulation runs drastically, is to identify interesting simulation "Scenarios". A "scenario" is a sequence of events (in our case: parameter-value-change events) designed to exercise specific properties of the model under investigation.

In the context of this research, the use of scenarios is twofold: model validation and investigation of DDoS detection.

Model validation is achieved by defining simple scenarios designed to verify if the model reproduces the expected behaviour of a real network.

6.1 Validation Scenarios

The simple scenarios described in the following sections are designed to validate the network model introduced in Chapter 2. Validation consists in devising the simplest scenario that is expected to re-create a given behaviour. The validation scenario is then run and results are observed to verify the presence of the expected phenomenon.

Validation scenarios are designed to exercise the only two non-trivial operating blocks of our model: *traffic mixer* and *rate limiting queue* (see Figure 2.11). The other blocks in the model: links, sources and sink, do not require specific validation because of their straightforward nature (although their are indirectly validated by the scenarios listed in the following sections).

Table 6.1 lists the validation scenarios and the traffic condition under which the test is carried out.

Scenario name	Condition(s)
Model conserves self-similarity (see Section 6.2)	No congestion
Mixer block reproduces traffic super-imposition (see Section 6.3)	No congestion
Model reproduces loss of self-similarity (see Section 6.4)	Severe congestion

Table 6.1: Validation Scenarios

6.2 The model conserves self-similarity

This validation scenario aims at verifying that our network model does not alter the self-similar property of traffic that travels from the sources to the target when no congestion is present.

6.2.1 Scenario setup

This scenario is setup by adding an H parameter estimator to the base model immediately before the target (see Figure 6.1).

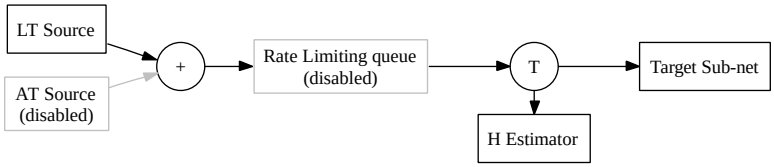


Figure 6.1: Network model setup for validation scenario number one: model conserves self-similarity

Only one of the potential traffic sources is used to avoid exercising the traffic mixer block which, in this case, acts as a passthrough.

To satisfy the “no congestion” condition the rate limiter is disabled. When no limit is imposed on the link speed by the rate limiter, the queue length is irrelevant and is left unspecified.

Multiple synthetic traffic traces with H values ranging from 0.55 to 0.95 in 0.05 steps and an average intensity of 100Mbit/s are used as traffic source. A simulation is run for each one of the traces.

Table 6.2 lists the parameter values used for this scenario.

6.2.2 Expected outcome

We expect the self-similarity estimator to yield a result close (within an estimator dependent tolerance) to the nominal value of the H parameter for the test traffic trace.

Parameter	Value
Source 1 intensity	100Mbit/s
Source 1 H	0.55-0.95
Source 2 intensity	nil
Source 2 H	nil
Rate limit	+inf
Queue length	+inf

Table 6.2: *Model conserves self-similarity* scenario parameters

6.2.3 Simulation results

The traffic at the target is split into 256 sub-traces each 4096 observations long, the value of the Hurst parameter is then estimated for each of the sub-traces and averaged. The variance of the H estimates is then used to compute the confidence intervals.

The process described above is repeated for all traffic traces and the results for are listed in Table 6.3. For each one of the fGn traffic traces we list the mean value of H, its variance and the corresponding 99% confidence interval.

Nominal	Mean H	Var H	99% Confidence Interval
0.55	0.5559	0.0002	0.0363
0.60	0.6105	0.0003	0.0407
0.65	0.6675	0.0002	0.0401
0.70	0.7202	0.0002	0.0416
0.75	0.7723	0.0002	0.0356
0.80	0.8260	0.0003	0.0408
0.85	0.8818	0.0003	0.0424
0.90	0.9332	0.0002	0.0389
0.95	0.9838	0.0003	0.0413

Table 6.3: Results for *Model conserves self-similarity* validation scenario

All nominal values fall within the confidence intervals of the corresponding averaged estimates according to our expectations for this scenario. However the estimation process is accurate for traces with a nominal H value at the lower end of the range and less accurate for traces with a nominal H value at the upper end. This is clearly visible in Figure 6.2 where the continuous line represents the expected value for the Hurst parameter. The horizontal axis represents the nominal value of H for the traffic trace while the results of the estimation are plotted against the vertical axis.

The lower accuracy of the estimator towards the upper end of the range could be caused by a number of factors, but likely related to the small sample (4096 points) used to estimate H for each one of the sub-traces. We can rule out the network model from

being responsible because the same estimation process run directly on the traffic traces yields the same numerical results as Figure 6.2. At this stage, we decide to not investigate the causes because of time restrictions.

The width of the confidence intervals is a measure of our analysis tools resolution and will restrict the choice of values of H for our DDoS simulations. When choosing Hurst parameter's value pairs for the traffic sources care will have to be taken to pick values for which the respective confidence intervals do not overlap. For instance: assuming the confidence interval half widths are approximately 0.4 like in the results of this test, then the value pairs (0.55, 0.60) or (0.70, 0.75) would not be acceptable because their confidence intervals overlap.

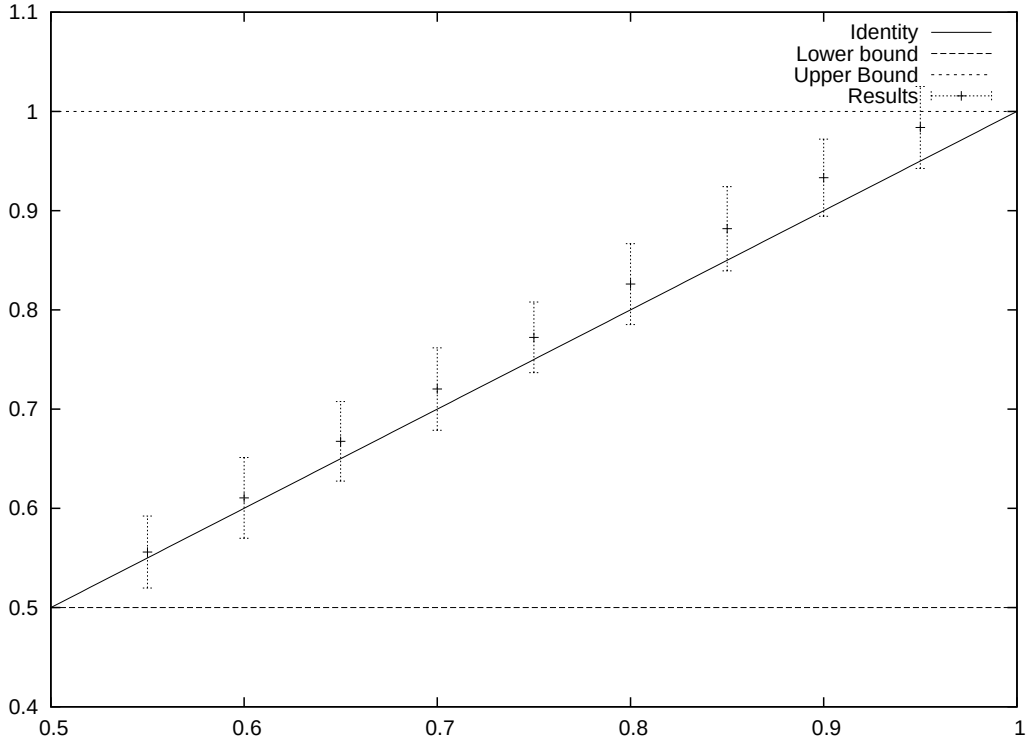


Figure 6.2: Results for model validation scenario number one: model conserves self-similarity

6.3 The traffic mixer component conserves self-similarity of the traffic source

This validation scenario aims at verifying that the traffic mixer operational block of our network model correctly outputs the superimposition of its two traffic inputs.

6.3.1 Scenario setup

This scenario is setup by adding an H parameter estimator to the base model immediately before the target (see Figure 6.3).

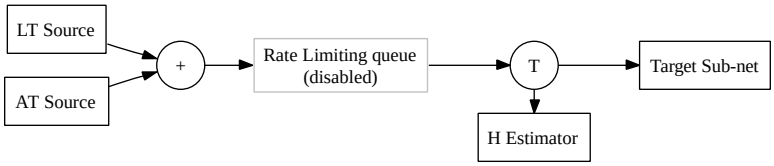


Figure 6.3: Network model setup for validation scenario number two: traffic mixer component conserves self-similarity

To satisfy the "no congestion" condition the rate limiter is disabled. When no limit is imposed on the link speed by the rate limiter, the queue length is irrelevant and is left unspecified.

Multiple synthetic traffic traces with H values ranging from 0.55 to 0.95 in 0.05 steps and an average intensity of 100Mbit/s are used as traffic sources. In each simulation two traces with the same nominal H value and the same average intensity are used as traffic sources.

The relationship between the two traffic source intensities should not affect the value of H of the mixed traffic. 100Mbit/s and 200Mbit/s were chosen arbitrarily. This validation scenario should succeed for any reasonable value for the intensities of the sources.

Table 6.4 lists the parameter values used for this scenario.

Parameter	**Value**
Source 1 intensity	100Mbit/s (any)
Source 1 H	0.55-0.95
Source 2 intensity	100Mbit/s (any)
Source 2 H	same as Source 1
Rate limit	+inf
Queue length	+inf

Table 6.4: *Traffic mixer component conserves self-similarity* scenario parameters

6.3.2 Expected outcome

We expect self-similarity of the two traffic sources to be conserved by the traffic mixer component. The self-similarity estimator measuring the Hurst parameter at the target should yield a result close (within an estimator dependent tolerance) to the nominal Hurst parameter value of the two traffic sources. We also expect the average intensity of the resulting traffic to be the sum of the average intensity of the sources.

6.3.3 Simulation results

The traffic at the target is split into 256 sub-traces each 4096 observations long, the value of the Hurst parameter is then estimated for each of the sub-traces and averaged. The variance of the H estimates is then used to compute the confidence intervals.

The process described above is repeated for all values of H in the range from 0.55 to 0.95 in 0.05 steps. The results for are listed in Table 6.5. For each one of the tests we list the mean value of H, its variance and the corresponding 99% confidence interval.

Nominal	**Mean H**	**Var H**	**99% Confidence Interval**
0.55	0.5559	0.0002	0.0367
0.60	0.6122	0.0003	0.0409
0.65	0.6678	0.0002	0.0393
0.70	0.7192	0.0002	0.0353
0.75	0.7721	0.0002	0.0371
0.80	0.8267	0.0002	0.0391
0.85	0.8812	0.0002	0.0388
0.90	0.9331	0.0002	0.0392
0.95	0.9889	0.0002	0.0370

Table 6.5: Results for *Traffic mixer component conserves self-similarity* validation scenario

The results confirm our expectations: when the traffic from the two sources is superimposed, the H value estimated for the resulting traffic is consistent with the nominal value for the sources.

In Figure 6.4 the continuous line represents the expected value for the Hurst parameter. The horizontal axis represents the nominal value of H for the traffic trace while the results of the estimation are plotted against the vertical axis.

The discussion in Section 6.2.3 about the different level of accuracy at the two opposite ends of the range still applies to these results.

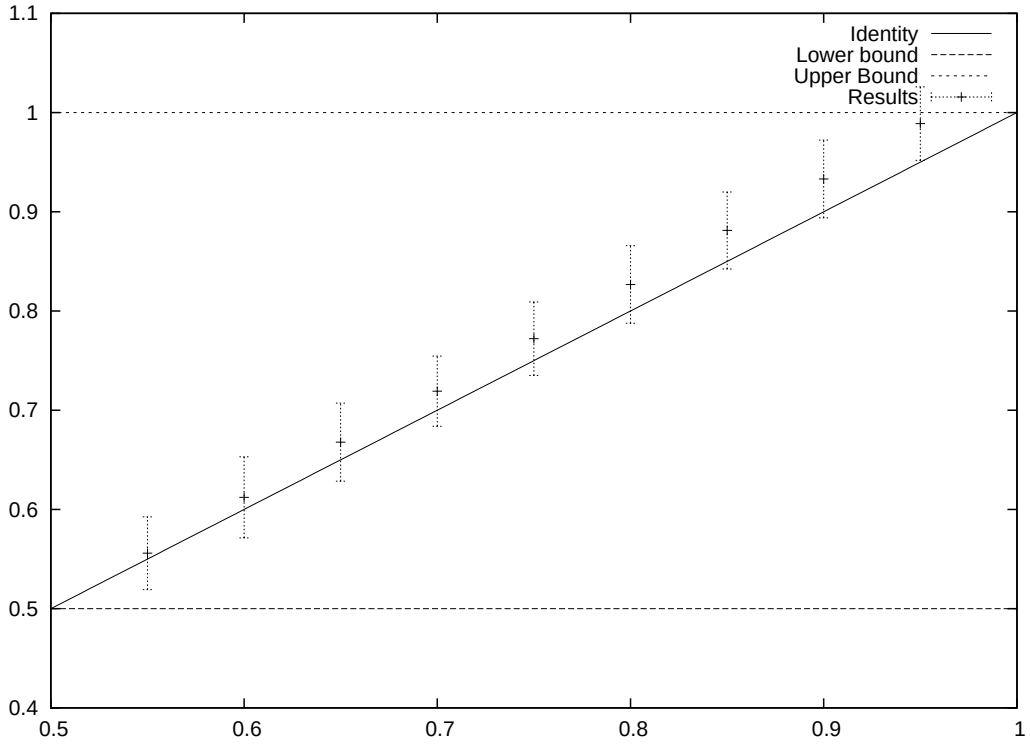


Figure 6.4: Results for model validation scenario number two: traffic mixer component conserves self-similarity

6.4 Congestion results in loss of self-similarity

This validation scenario aims at verifying that the rate limiting queue operational block of our network model causes loss of self-similarity when traffic intensity is above the rate limit.

6.4.1 Scenario setup

The base model is setup for this scenario by adding an H parameter estimator immediately before the target (see Figure 6.5).

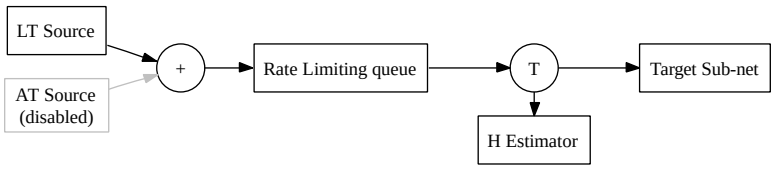


Figure 6.5: Network model setup used for validation scenario number three: congestion results in loss of self-similarity

Only one of the potential traffic sources is used for simplicity. Even thought using both traffic sources in our model would result in the same effect, we prefer the simplest setup that is still capable or reproducing the desired effect.

Multiple synthetic traffic traces with H values ranging from 0.55 to 0.95 in 0.05 steps and an average intensity of 100Mbit/s are used as traffic source. A simulation is run for each one of the traces.

The rate limiter is set to 90Mbit/s, 80Mbit/s and 50Mbit/s in three separate simulation runs to induce increasingly severe congestion. The queue length is arbitrarily set to store a maximum of 2 seconds at 100Mbit/s (the traffic source rate) before packets will start being discarded.

Table 6.6 lists the parameter values used for this scenario.

Parameter	**Scenario A**	**Scenario B**	**Scenario C**
Source 1 intensity	100Mbit/s	100Mbit/s	100Mbit/s
Source 1 H	0.55-0.95	0.55-0.95	0.55-0.95
Source 2 intensity	nil	nil	nil
Source 2 H	nil	nil	nil
Rate limit	90Mbit/s	80Mbit/s	50Mbit/s
Queue length	200Mbit	200Mbit	200Mbit

Table 6.6: *Congestion results in loss of self-similarity* scenario parameters

6.4.2 Expected outcome

When traffic with a known H parameter is forced through a rate limiting queue with a maximum bandwidth that is smaller than the average traffic throughput we expect self-similarity estimated at the target host to drift away from the nominal value specified for the traffic source.

6.4.3 Simulation Results

Results are grouped by rate limit in tables Table 6.7, Table A.2, Table A.3 and plotted in Figure 6.6. Each plot includes three reference lines: an upper bound for the Hurst parameter, a lower bound for the Hurst parameter and the nominal value for the Hurst parameter. The mean estimates are plotted with their confidence intervals.

The first group of simulations was run with a rate limit of 90Mbit/s and an average source intensity of 100Mbit/s. The plot in Figure 6.6a shows how the means of the Hurst parameter estimates are closer to 1.0 compared to the results we obtained in absence of congestion (see Figure 6.2) especially for traffic sources with values of H equal to 0.55, 0.60 and 0.65. The other distinctive feature of the plot is the significant increase in confidence interval width compared to the results obtained in absence of congestion. The increased width of confidence intervals is a consequence of the increased variance of the H parameter estimates series. Variances listed in Table 6.7 are two order of magnitude grater than their correspondent values for the congestion free scenario listed in Table 6.3.

The second group of simulations was run with a rate limit of 80Mbit/s and an average source intensity of 100Mbit/s. The results plotted in Figure A.2 display further displacement of the means from the ideal value. Traffic sources with lower H parameter values are more affected by the shift of the mean towards 1.0 similarly to the first group of simulations. The confidence intervals are overall in the same order of magnitude as the first group of simulation, but comparing corresponding width in the first and

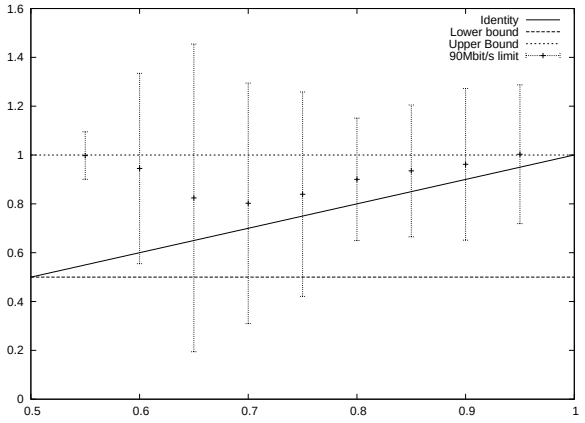
second group we can observe the following three phenomena: as congestion becomes more severe the mean value approaches 1.0, confidence interval widths first increase with the onset of congestion then decrease once the mean value approaches 1.0, as the rate limit increases the means and confidence interval for higher values of H become affected.

The third group of simulations was run with a rate limit of 50Mbit/s and an average source intensity of 100Mbit/s. The results for this group indicate extreme congestion and the relative plot in Figure A.3 is effectively a series of points around the horizontal 1.0 upper bound for the Hurst parameter value. These results are compatible with the three phenomena mentioned in the previous paragraph.

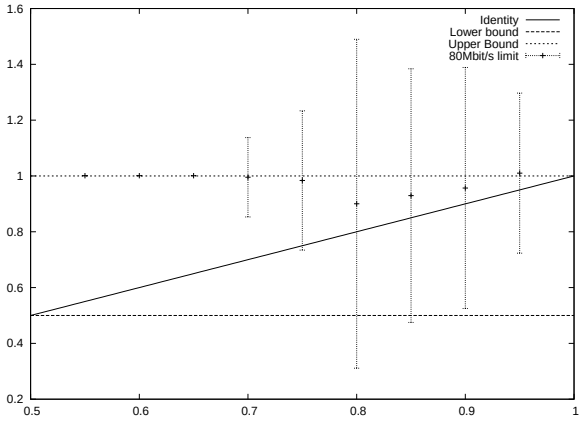
The shift towards 1.0 in the means of estimates observed in these simulations can be explained by observing that, informally, the rate limiter operates by cutting peaks in traffic intensity while the queue levels the valleys. As congestion increases, the traffic exiting the rate limiting queue becomes increasingly flat. In other words as the ratio between the source's average intensity and rate limit increases beyond 1.0, the variance of the average intensity measured after the rate limiting decreases. This condition implies absence of self-similarity. Estimation of self-similarity is not applicable to a flat (non self-similar) traffic trace and the result of 1.0 we obtain is a side-effect of the estimators' implementation.

The widening of the confidence intervals observed in the simulations is the result of the estimation process. We recall that the result's mean H are obtained by averaging the estimates of sequential non-overlapping blocks of observations. In normal conditions the Hurst parameter estimates for all blocks are similar and result in a small variance of the estimates series (see Table 6.3). Under congestion each block is affected differently by the rate limiting queue: the block's peaks and valleys are flattened depending on its intensity, bursty-ness and the state of the queue. A block displaying self-similar features, before entering the rate limiting queue, will partially or totally lose these features resulting in highly variable estimates for the Hurst parameter and thus increasing the variance of the averaged estimates series.

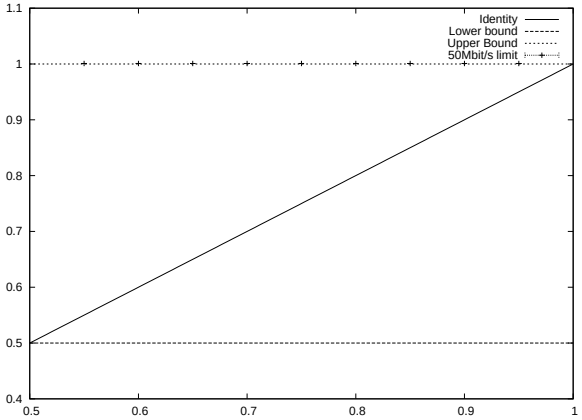
The progressive nature of loss of self-similarity observed in the simulations suggests that traffic with a lower nominal H is affected differently by congestion compared to traffic with an high nominal H. The reason for this phenomenon resides in how the characteristic bursty-ness of self-similar traffic interacts the rate limiting queue. The rate limiting queue is composed of two operational blocks connected as a cascade: a queue and a rate limiter. The overall effects of the rate limiting queue are the result of the cumulative effects of the two operational blocks that constitute it. We have extracted a 4096 observations long block from three traffic traces with different nominal values of the Hurst parameter: 0.55, 0.75, 0.95; and plotted a series of graphs to highlight the effects of each operational block in a rate limiting queue. The plots are presented in three groups and each group is composed by two rows. The first row in each group is an intensity VS time plot. The second row in each group is a log-log plot of the trace displayed immediately above it and obtained with multi resolution analysis. The first group (see Figure 6.7) features the traces in a congestion free scenario. In the second group (see Figure 6.8) the same traces are subject to rate limiting but no queueing. In the third group (see Figure 6.9) the same traces are subject to both rate limiting and queueing. A Comparison of the first rows of both first and second group of plots shows how the features of traces with different nominal values of H are affected differently by rate limiting. Comparing the second and third group of plots we can see how the presence of the queue contributes to erase all valleys from the trace with lower H while traces with higher H still possess some features.



(a) *Rate limit 90Mbit/s.*



(b) *Rate limit 80Mbit/s.*

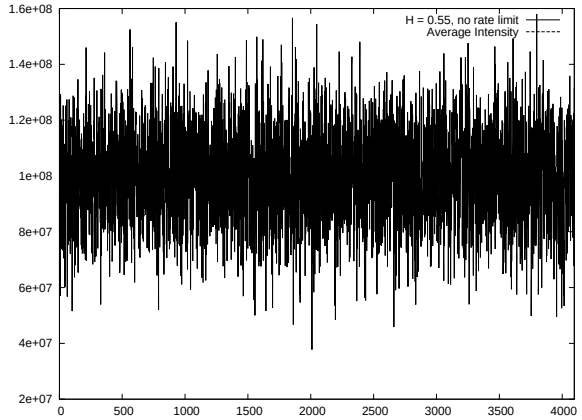


(c) *Rate limit 50Mbit/s.*

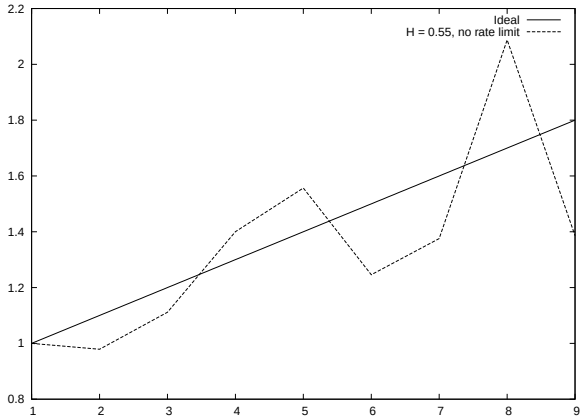
Figure 6.6: Results for model validation scenario number 3.

Nominal	Mean H	Var H	99% Confidence Interval
0.55	0.9977	0.0014	0.0975
0.60	0.9448	0.0229	0.3896
0.65	0.8244	0.0598	0.6301
0.70	0.8024	0.0366	0.4925
0.75	0.8395	0.0264	0.4186
0.80	0.9004	0.0095	0.2507
0.85	0.9351	0.0110	0.2698
0.90	0.9619	0.0145	0.3106
0.95	1.0033	0.0122	0.2843

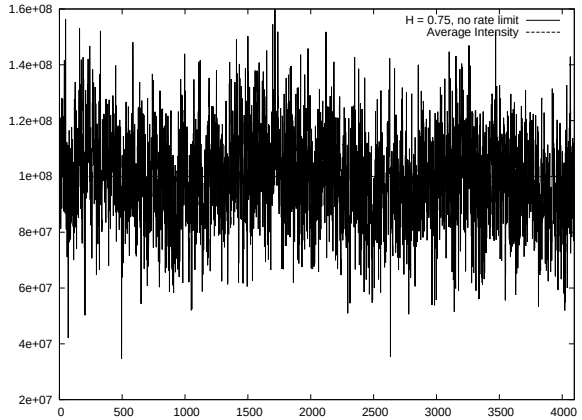
Table 6.7: Results for *Congestion results in loss of self-similarity* validation scenario, case A



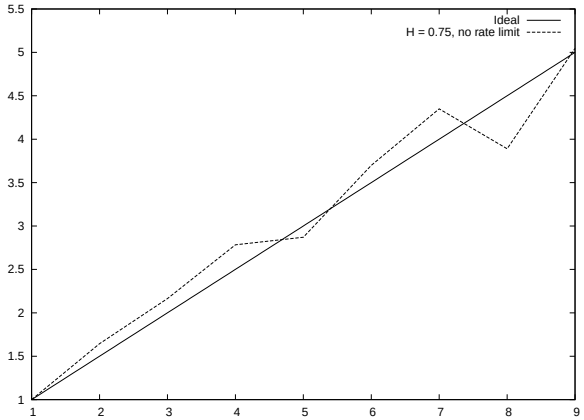
(a) *Intensity vs Time. H = 0.55.*



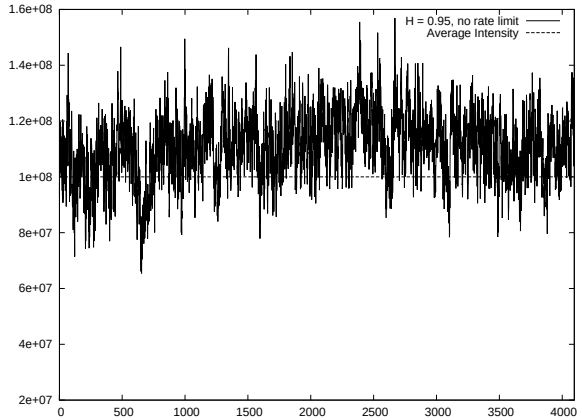
(b) *MRA log-log plot of (a) compared to an ideal H = 0.55 line.*



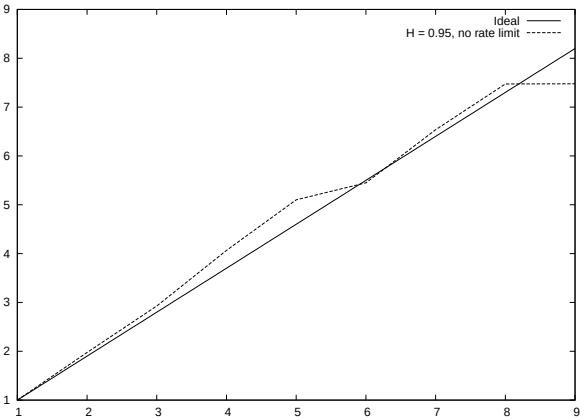
(c) *Intensity vs Time. H = 0.75.*



(d) *MRA log-log plot of (a) compared to an ideal H = 0.75 line.*

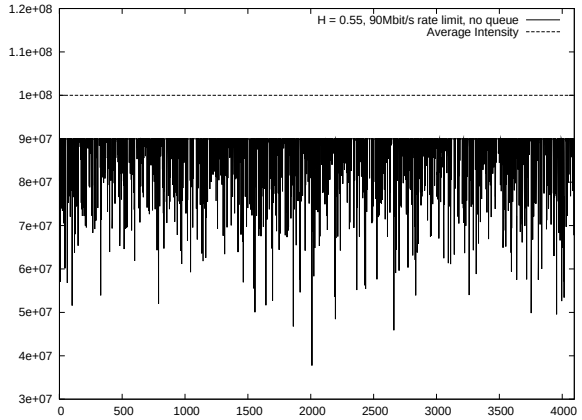


(e) *Intensity vs Time. H = 0.95.*

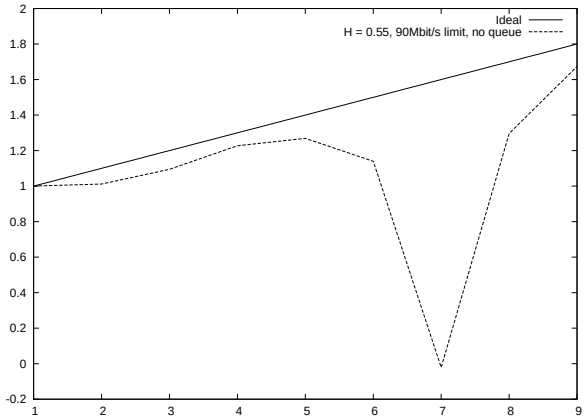


(f) *MRA log-log plot of (a) compared to an ideal H = 0.95 line.*

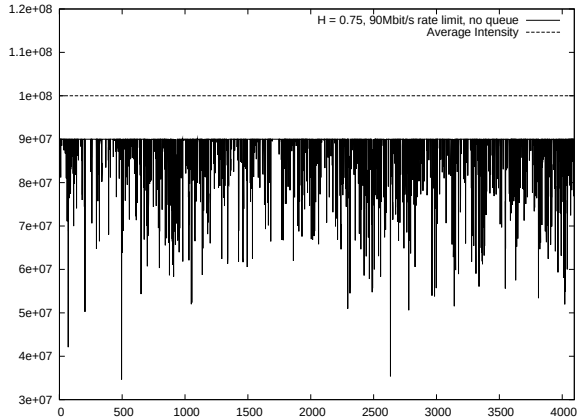
Figure 6.7: Traffic intensity plot (left) and MRA log-log plot (right) for a 4096 observations block with no rate limit. (a) and (b) H=0.55. (c) and (d) H=0.75. (e) and (f) H=0.95.



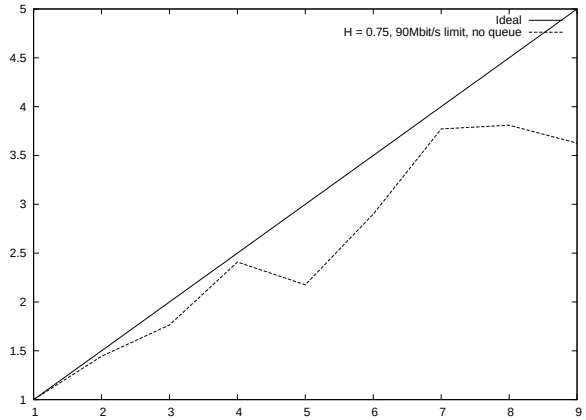
(a) *Intensity vs Time. H = 0.55.*



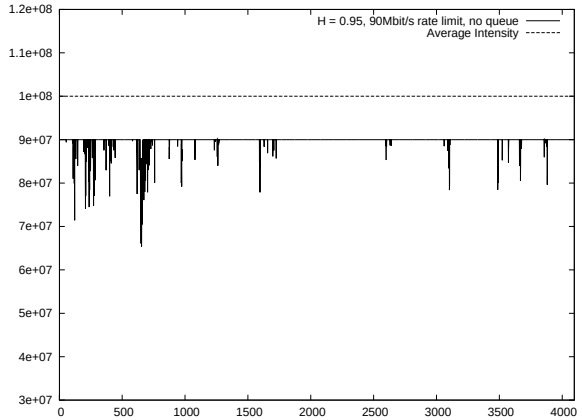
(b) *MRA log-log plot of (a) compared to an ideal H = 0.55 line.*



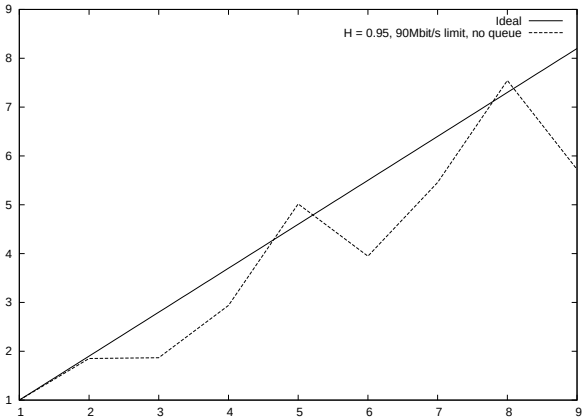
(c) *Intensity vs Time. H = 0.75.*



(d) *MRA log-log plot of (a) compared to an ideal H = 0.75 line.*

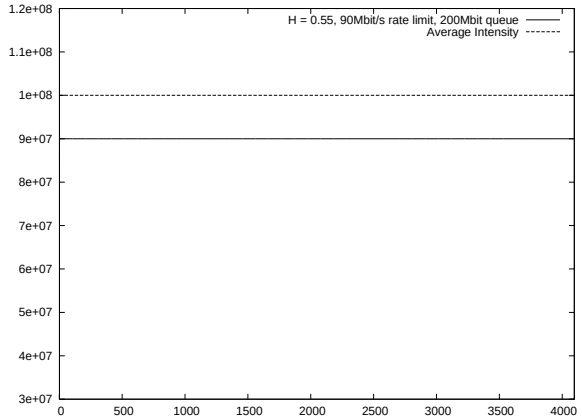


(e) *Intensity vs Time. H = 0.95.*

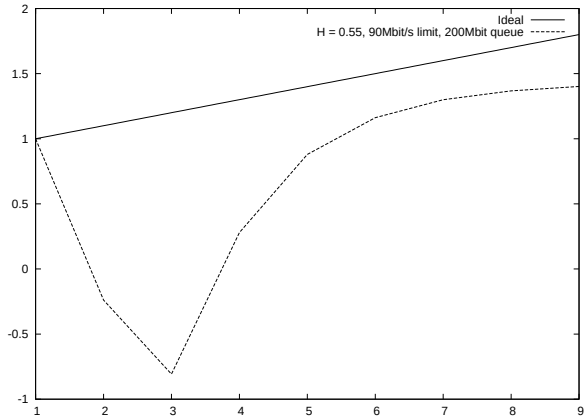


(f) *MRA log-log plot of (a) compared to an ideal H = 0.95 line.*

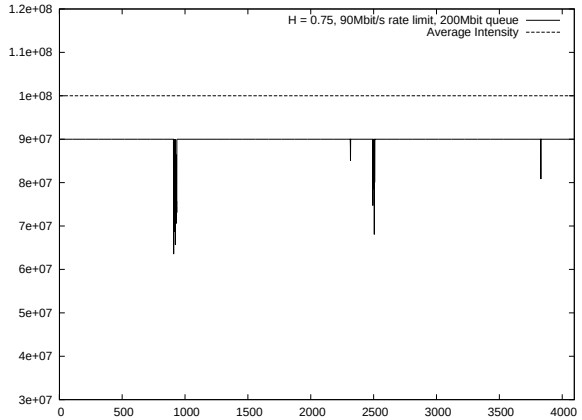
Figure 6.8: Traffic intensity plot (left) and MRA log-log plot (right) for a 4096 observations block with 90Mbit/s rate limit and no queue. (a) and (b) H=0.55. (c) and (d) H=0.75. (e) and (f) H=0.95.



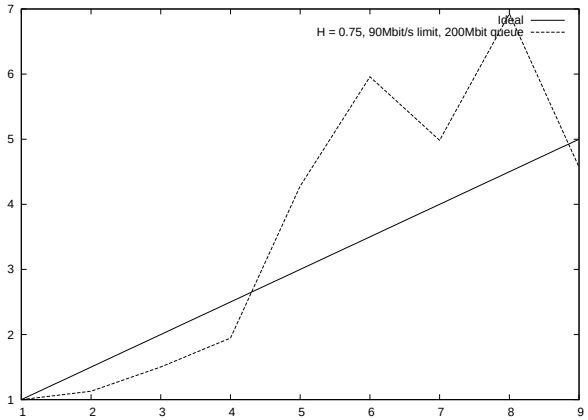
(a) *Intensity vs Time. H = 0.55.*



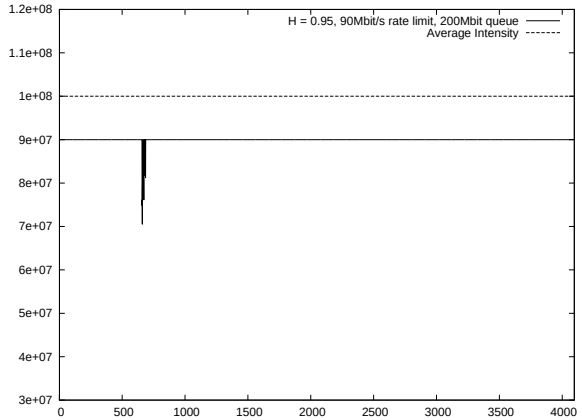
(b) *MRA log-log plot of (a) compared to an ideal H = 0.55 line.*



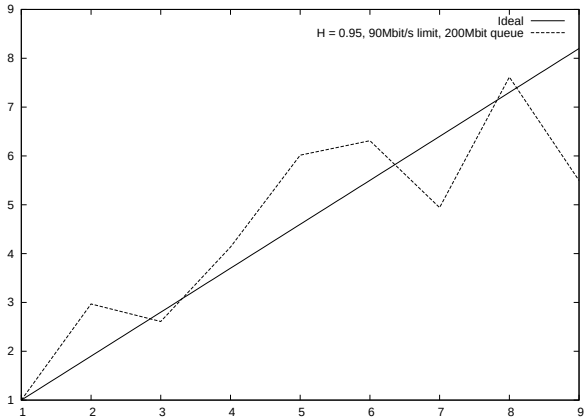
(c) *Intensity vs Time. H = 0.75.*



(d) *MRA log-log plot of (a) compared to an ideal H = 0.75 line.*



(e) *Intensity vs Time. H = 0.95.*



(f) *MRA log-log plot of (a) compared to an ideal H = 0.95 line.*

Figure 6.9: Traffic intensity plot (left) and MRA log-log plot (right) for a 4096 observations block with 90Mbit/s rate limit and a 200Mbit queue. (a) and (b) H=0.55. (c) and (d) H=0.75. (e) and (f) H=0.95.

Chapter 7

DDoS Scenarios

With our model validated by the tests run in the previous chapter we are left with the last phase of our research. We begin our discussion with the reduction of simulation parameters, both in number and in range. The number of parameters is reduced from six to three for congestion free scenarios.

Subsequently we define and run simulation scenarios for the superimposition of two self-similar traffic flows with different values of the Hurst parameter and different average intensities. The results are discussed in the next chapter.

7.1 DDoS Scenario parameters

The network model form Chapter 2 has a total of six of operating parameters, namely: traffic intensity and Hurst parameter for each traffic source, length and maximum throughput for the rate limiting queue.

Parameter	**Unit**	**Range**
Source 1 intensity	bit/s	0 to +inf
Source 1 H	N/A	0.5 to 1
Source 2 intensity	bit/s	0 to +inf
Source 1 H	N/A	0.5 to 1
Rate limit	bit/s	0 to +inf
Queue length	bit	0 to +inf

Table 7.1: Network model parameters

The six parameters of the network model are listed in Table 7.1. Four of the six parameters (traffic intensities for both sources, rate limit and queue length) are defined over an infinite interval. This is not desirable when trying to systematically cover the whole range of value. The ideal situation would be to have a reduced set of parameters defined over a finite interval.

We want to reduce the number of parameters upon which the simulation depends and, if possible, restrict their domain. In order to do so, we first distinguish two classes of simulation scenarios: without congestion and with congestion and address them separately.

7.2 Congestion-free scenarios

Scenarios without traffic congestion operate on a simplified model in which the rate limiting queue block is disabled like in Figure 6.3. The two parameter related to the rate limiting queue do not apply and can be ignored (or set to +inf) thus reducing the number of parameters from six to four.

When operating under the assumption of no traffic congestion there is no packet loss thus the resulting intensity of the traffic at the target is simply the sum of the two sources' traffic intensity: $I_T = I_1 + I_2$. Since the absolute value of I_T is not interesting for our investigation we can chose an arbitrary value for I_1 and define $I_2 / I_1 = k$. The case $k = 0$ is not considered relevant because it reduces the network model to a single self-similar source which is already covered in the model validation scenarios (see Chapter 6). The values for the *k* ratio are chosen to be symmetrical and centered around 1 from 1:10 to 9:10 and from 10:9 to 10:1 (see Table 7.3).

Reducing the number of value pairs for H_1 and H_2 is achieved by choosing equally spaced values in the range from 0.5 and 1.0 and by exploiting the symmetry in the network model as shown in Table 7.2: only the pairs marked by a star are used instead of the entire matrix.

$H_1 \backslash H_2$	**0.55**	**0.60**	**0.65**	**0.70**	**0.75**	**0.80**	**0.85**	**0.90**	**0.95**
0.55	*	*	*	*	*	*	*	*	*
0.60	.	*	*	*	*	*	*	*	*
0.65	.	.	*	*	*	*	*	*	*
0.70	.	.	.	*	*	*	*	*	*
0.75	.	.	.	.	*	*	*	*	*
0.80	.	.	.	.	.	*	*	*	*
0.85	.	.	.	.	.	.	*	*	*
0.90	.	.	.	.	.	.	.	*	*
0.95	.	.	.	.	.	.	.	.	*

Table 7.2: H_1 and H_2 parameters value combinations

We have reduced the initial six to three free parameters. Also, all three parameters take values over a finite range of values (see Table 7.4).

Two series of simulations are run. Each series based on a different scenario designed to observe the effects of self-similar traffic superimposition. The first scenario is designed to exercise superimposition of traffic with the same intensity and different H value. The second scenario exercises superimposition of traffic sources at different intensity ratios.

Ratio	k	Ratio	k
1:10	0.1	**10:9**	1.11
2:10	0.2	**10:8**	1.23
3:10	0.3	**10:7**	1.46
4:10	0.4	**10:6**	1.66
5:10	0.5	**10:5**	2.00
6:10	0.6	**10:4**	2.50
7:10	0.7	**10:3**	3.33
8:10	0.8	**10:2**	5.00
9:10	0.9	**10:1**	10.0
10:10	1.0		

Table 7.3: Source-2 to Source-1 intensity ratios and corresponding *k* value

Parameter	Unit	Range
Sources intensity ratio k = I_2 / I_1	bit/s	1/10 to 10 (see Table 7.3)
Source 1 H	N/A	0.5 to 1 (see Table 7.2)
Source 2 H	N/A	0.5 to 1 (see Table 7.2)
Rate limit	bit/s	+inf
Queue length	bit	+inf

Table 7.4: Simplified network model parameters for congestion free scenarios

7.2.1 Simulation Scenario-1 setup

For the first scenario we set $I_1 = I_2 = 100$Mbit/s and use values from Table 7.2 for H_1 and H_2. A simulation is run for each (H_1, H_2) pair. The results for the 81 simulations are grouped by H value of the first source and listed in nine tables. The table and plot for H = 0.55 are reproduced here while the remaining ones are reproduced in Appendix B to ease readability.

Parameter	**Value**
Source 1 intensity	100Mbit/s
Source 1 H	see Table 7.2
Source 2 intensity	100Mbit/s
Source 2 H	see Table 7.2
Rate limit	+inf
Queue length	+inf

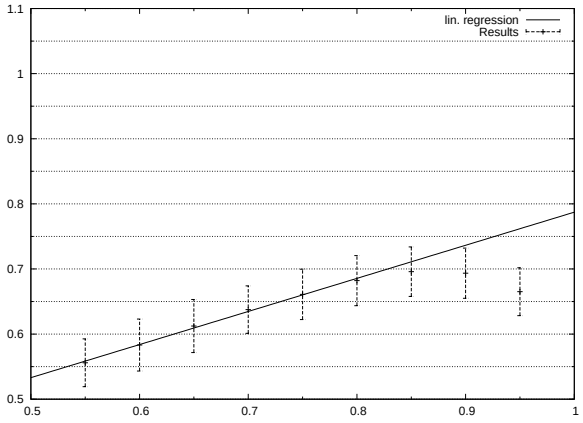
Table 7.5: Parameters for DDoS Scenario number one

7.2.2 Simulation Scenario-1 results

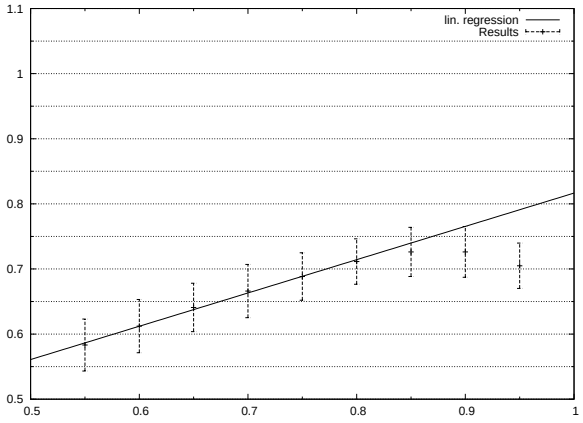
The plots display the mean H value with relative confidence interval measured at the target host and a linear regression curve.

The confidence intervals of the results for this scenario across all value pairs for the H_1 and H_2 sources are not different form the results obtained with a single source during the model validation tests (see Section 6.2.3). This indicates that superimposition does not disrupt self-similarity and our results are not less reliable when operating with two sources.

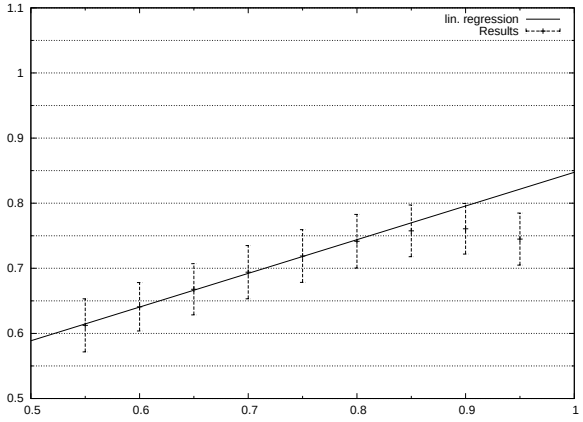
Figure 7.1a shows the results of the superimposition when one source is set to H = 0.55 and the other is changed to a different value from Table 7.2 for each simulation. The plot displays quasi-linear behaviour up to H = 0.85 then the mean values of H decrease instead of increasing. All plots for this scenario exhibit the same feature although the threshold where the trend changes form increase to decrease is higher for plots with the fixed source set to an higher value of H. We can draw two important conclusions from this: first superimposing traces with $H_1 > H_2$ results in a self-similar trace with $H = H_T$ and $H_1 > H_T > H_2$. Second the relation $s : s(H_1, H_2) = H_T$ is likely not linear.



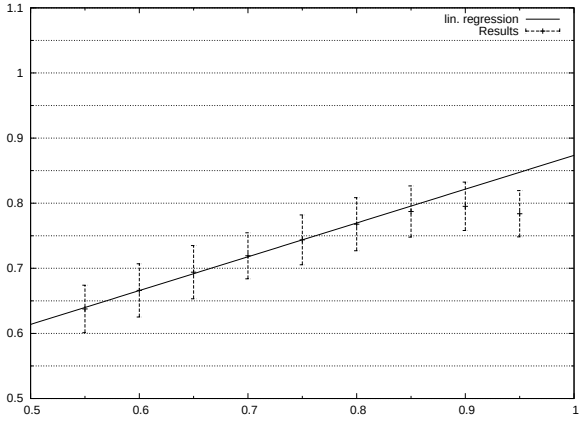
(a) *1st source fixed to H = 0.55, 2nd source varying from 0.55 to 0.95 in 0.05 steps.*



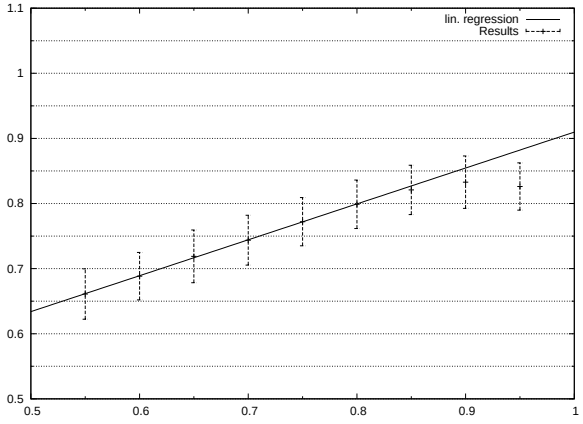
(b) *1st source fixed to H = 0.60, 2nd source varying from 0.55 to 0.95 in 0.05 steps.*



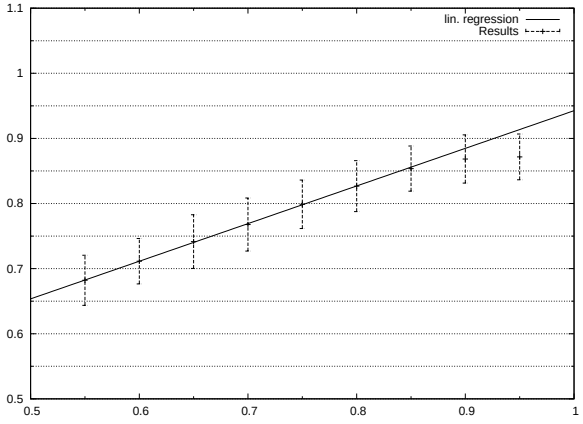
(c) *1st source fixed to H = 0.65, 2nd source varying from 0.55 to 0.95 in 0.05 steps.*



(d) *1st source fixed to H = 0.70, 2nd source varying from 0.55 to 0.95 in 0.05 steps.*

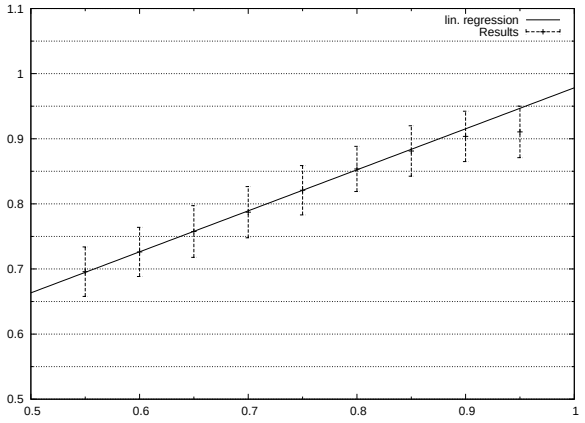


(e) *1st source fixed to H = 0.75, 2nd source varying from 0.55 to 0.95 in 0.05 steps.*

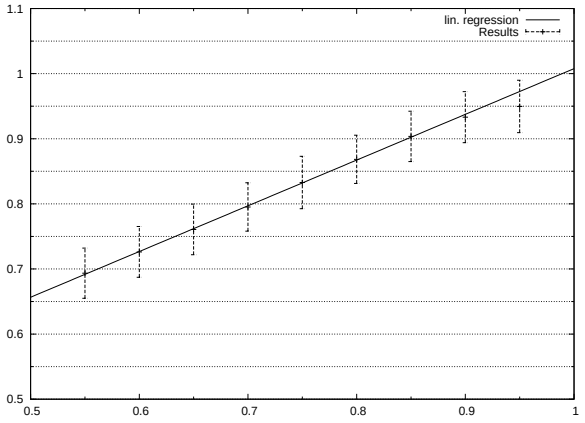


(f) *1st source fixed to H = 0.80, 2nd source varying from 0.55 to 0.95 in 0.05 steps.*

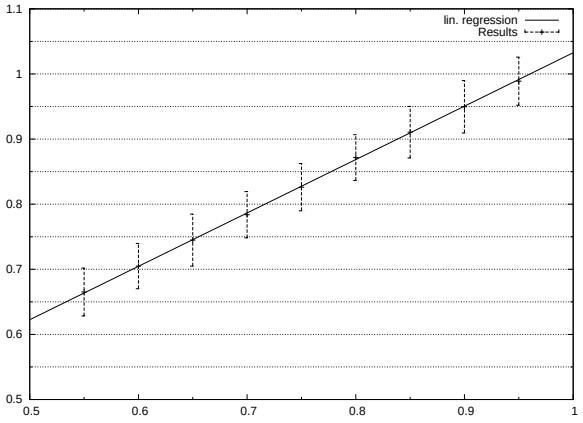
Figure 7.1: Results for simulation scenario number one. Hurst parameter value for superimposition of two sources.



(a) *1st source fixed to H = 0.85, 2nd source varying from 0.55 to 0.95 in 0.05 steps.*



(b) *1st source fixed to H = 0.90, 2nd source varying from 0.55 to 0.95 in 0.05 steps.*



(c) *1st source fixed to H = 0.95, 2nd source varying from 0.55 to 0.95 in 0.05 steps.*

Figure 7.2: Continued from Figure 7.1. Results for simulation scenario number one. Hurst parameter value for superimposition of two sources.

Comparing the different plots we have obtained it appears that the slope of the quasi-linear portion of the graph is not constant. To help us visualise the change in slope, the linear regression based on values for 0.55 <= H <= 0.85 was calculated for each group of results and added to the plot. A standalone plot of the linear regression slope is reproduced in Figure 7.3. The slope plot suggests that sources with a lower value of H contribute more to to the value of H_T than sources with a higher H.

2nd src H	Mean H	Var H	99% Confidence Interval
0.55	0.5559	0.0002	0.0367
0.60	0.5832	0.0002	0.0399
0.65	0.6124	0.0003	0.0408
0.70	0.6376	0.0002	0.0365
0.75	0.6610	0.0002	0.0387
0.80	0.6821	0.0002	0.0385
0.85	0.6959	0.0002	0.0380
0.90	0.6935	0.0002	0.0385
0.95	0.6651	0.0002	0.0368

Table 7.6: Results for Scenario-1 simulation with 1st source H = 0.55

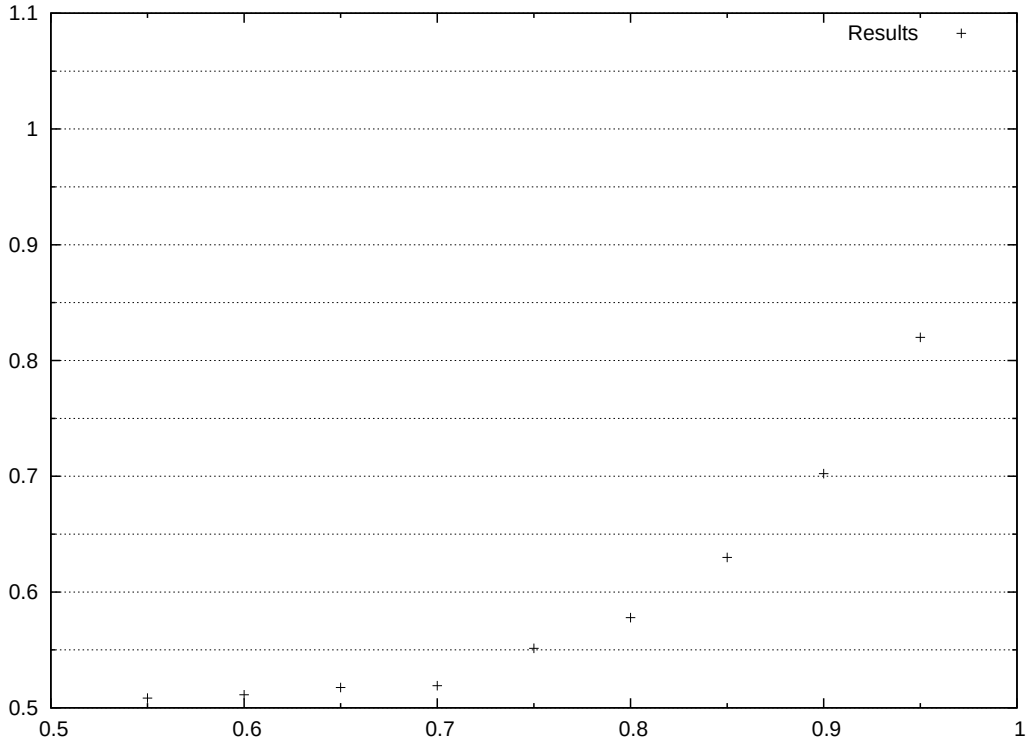


Figure 7.3: Slopes plot. The slope of the resulting curve for each of the simulations in this scenario. Values on the X axis are the H of the fixed source.

7.2.3 Simulation Scenario-2 setup

For the second scenario we set I_1 = 100Mbit/s and use ratios from Table 7.3 to control I_2 intensity. This scenario uses (0.55, 0.75), (0.75, 0.95), (0.55, 0.95) value pairs for H_1 and H_2. A simulation is run for each (H_1, H_2) pair and each intensity ratio.

H	Slope
0.55	0.5083
0.60	0.5113
0.65	0.5175
0.70	0.5191
0.75	0.5513
0.80	0.5779
0.85	0.6299
0.90	0.7022
0.95	0.82

Table 7.7: Slope vs fixed source's H

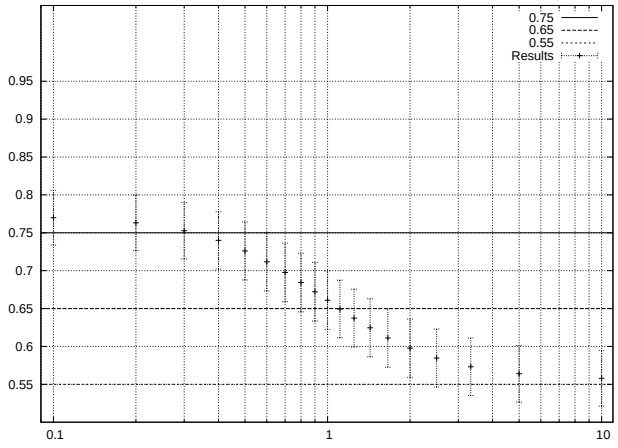
The results for the 57 simulations are grouped by (H_1, H_2) value pair and listed in 3 tables. The table and plot for (0.55, 0.95) are reproduced here while the remaining ones are reproduced in Appendix C to ease readability.

Parameter	Value
Source 1 intensity	100Mbit/s
Source 1 H	0.55, 0.75
Source 2 intensity	100Mbit/s
Source 2 H	0.75, 0.95
Rate limit	+inf
Queue length	+inf

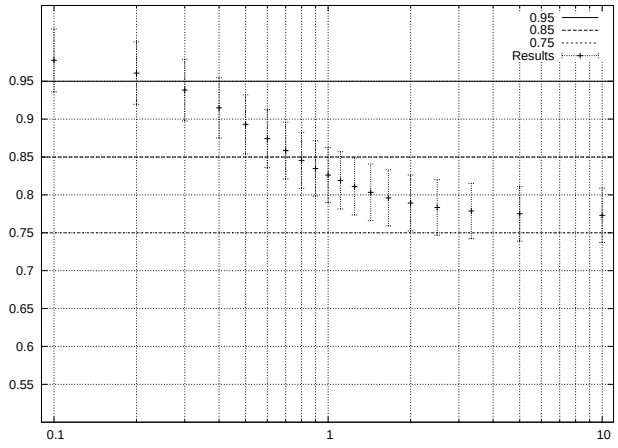
Table 7.8: Parameters for DDoS Scenario-2

7.2.4 Simulation Scenario-2 results

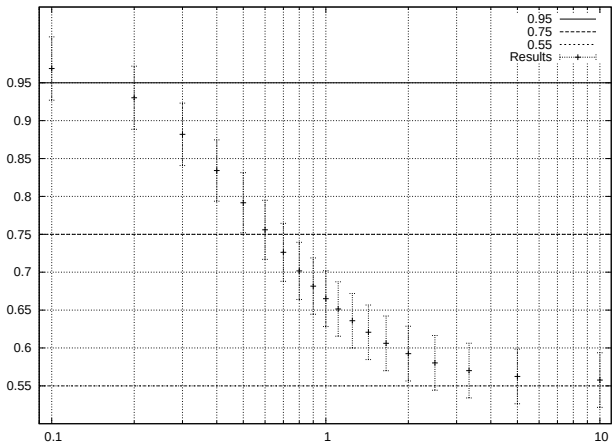
The results of these simulations show that the intensity ratio between two sources with different H influences the resulting H_T at the target host: the resulting H_T moves closer to the value of H for source with higher intensity. Similarly to the previous scenario it appears that sources with lower H dominate over sources with higher H. From the plots it also appears that the relation $r : r_{H_1,H_2}(I_1,I_2) = H_T$ is not linear.



(a) *1st source fixed to H = 0.55, 2nd source fixed to 0.75, intensity ratio from 1:10 to 10:1.*



(b) *1st source fixed to H = 0.75, 2nd source fixed to 0.95, intensity ratio from 1:10 to 10:1.*



(c) *1st source fixed to H = 0.55, 2nd source fixed to 0.95, intensity ratio from 1:10 to 10:1.*

Figure 7.4: Results for simulation scenario number two. Hurst parameter value for superimposition of two sources.

7.3 Non Congestion-free scenarios

Non Congestion-free scenarios use the rate limiting queue to simulate congestion conditions in the network. Because of the unidirectional flow of packets from sources to the target, the traffic mixer and rate limiting queue are traversed by the traffic flow in a sequential manner. In other words, the rate limiting queue is positioned such that it affects already superimposed traffic. Intuitively the effects of the queue on self-similar traffic should be the same whether the traffic is the result of superimposition or is coming form a single source.

Our preliminary simulations agree with intuition and the results from the validation scenario in Section 6.4. Because of the redundancy of the results and time restrictions we do not proceed with a full series of simulations.

Chapter 8

Conclusions

8.1 Conclusions

Our work is based on the results on distributed denial of service detection obtained by [AM04] and was prompted by the observation that, contrary to the assumptions made by existing literature, bot-nets could generate self-similar traffic.

Existing studies predict a decrease in the value of the Hurst parameter when non self-similar DDoS traffic is present. We changed the underlying premise found in the relevant literature by assuming DDoS traffic is self-similar and used network simulations to show how the superimposition of nominal and malicious traffic affects the Hurst parameter and potentially the feasibility of DDoS detection based on self-similarity.

Using an analysis of bot-net worm payloads (see [DLD00]) collected on infected hosts we justified our claim that bot-net infrastructure can be used to generate self-similar DDoS traffic (see Chapter 1).

A simple model of a network reproducing the necessary features of a real network subject to DDoS was defined (see Chapter 2) and later validated using simulations (see Chapter 6).

We selected, tested, modified and when necessary designed and developed software tools to carry out our simulations. DDoS simulation scenarios were defined and run against the network model (see Chapter 7).

The results show that superimposition of traffic flows with different Hurst parameter values result in traffic with an Hurst parameter value that is a *blend* of the sources' ones. The Hurst parameter value of the resulting traffic is influenced by both the H value of sources and their relative intensities. The Hurst parameter value at the target H_T is greater than the $\min(H_1, H_2)$ and smaller than $\max(H_1, H_2)$. The influence of a source on the resulting value H_T increases with his intensity relative to the other source. There are also indications that sources with lower values of H tend to dominate in the resulting traffic self-similarity. The most important result is perhaps the observation that, contrary to previous studies where DDoS traffic is non self-similar, the Hurst parameter can increase in presence of attack traffic.

The other important observation we can draw from our results is that, in presence of congestion, self-similarity estimation becomes unreliable and ultimately meaningless when congestion is so severe to completely flatten the traffic's bursty behaviour.

For an attack to be undetectable it would have to both generate traffic with a value of H close to the one expected at the target and avoid network congestion while aiming at exhausting the target's processing power (rather than its network bandwidth) by sending well formed requests from non spoofed addresses. The value of H expected by the target changes in time and is, in general, unknown by an attacker and not easy to recover. The attacker is required to send well formed requests if he/she wants to occupy processing resources at the target and in order to do so it also requires to establish complete connection which are not possible when the address field in packets is spoofed.

Detection of DDoS based on self-similarity estimation is more difficult but still possible in the light of our results. The Hurst parameter value can increase as well as decrease because of a DDoS thus detectors need to be redesigned to take this into account. Congestion is to be avoided because of its detrimental effects to traffic self-similarity. Avoiding congestion is at odds with the very nature of DDoS and represents the weak point of self-similarity based DDoS detection techniques. On the other hand in presence of congestion detecting DDoS becomes a non-issue because whether or not an attack caused the surge in traffic the measures to be taken are identical.

8.2 Future work

Results of our investigation are limited by the precision of the Hurst estimators available to us. Estimation precision also affect a DDoS detector design and performance and is therefore an interesting area for further research.

DDoS detection based on self-similarity is significantly affected by network congestion. An in-depth study of the effects of rate limiting and queue length on self-similar traffic would be useful to establish a relation between level of congestion and reliability of DDoS detection.

In the course of our research we have implemented and tested the self-similar traffic generator described in [TWS97]. The result of our tests do not agree with neither the original paper nor an empirical traffic generator comparison conducted in [HKBN07]. The reasons for this discrepancy could be the lack of details about the numerous parameters involved in reproducing the conditions of the other peers' simulations. We believe it would be useful to invest time in reconciling these discrepancies. However, thanks to our thorough testing we are confident that our implementation is correct.

Part IV

Appendices

Appendix A

Validation scenario-3, results

A.1 Validation scenario-3 case A, results

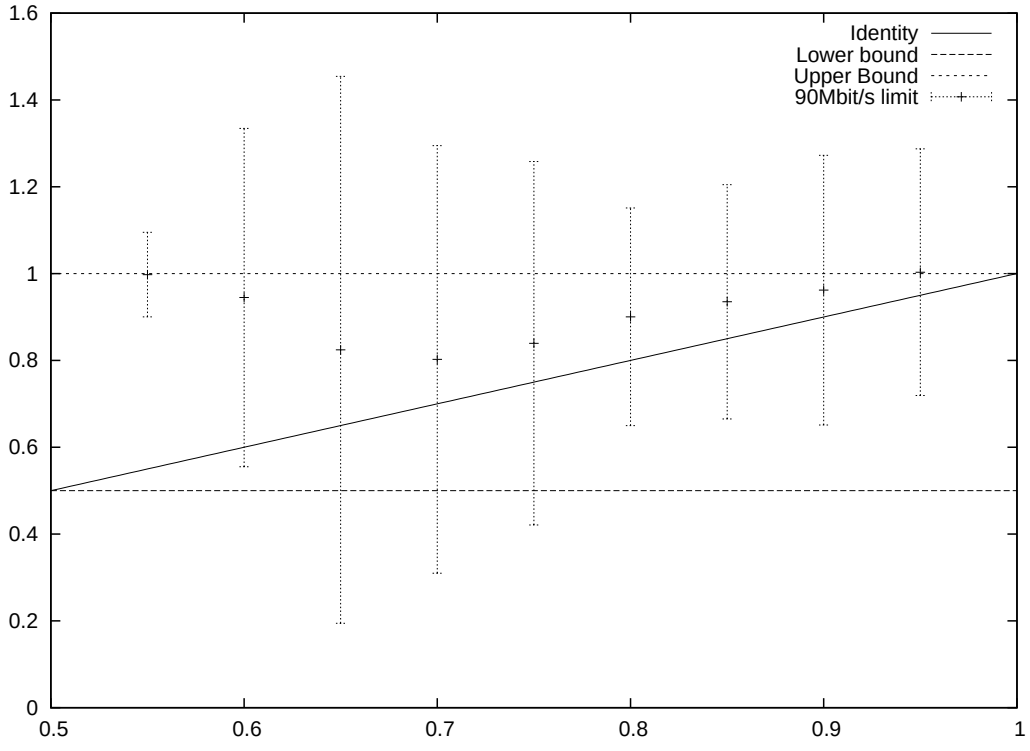


Figure A.1: Results for model validation scenario number 3A. Rate limit 90Mbit/s

Nominal	Mean H	Var H	99% Confidence Interval
0.55	0.9977	0.0014	0.0975
0.60	0.9448	0.0229	0.3896
0.65	0.8244	0.0598	0.6301
0.70	0.8024	0.0366	0.4925
0.75	0.8395	0.0264	0.4186
0.80	0.9004	0.0095	0.2507
0.85	0.9351	0.0110	0.2698
0.90	0.9619	0.0145	0.3106
0.95	1.0033	0.0122	0.2843

Table A.1: Results for *Congestion results in loss of self-similarity* validation scenario, case A

A.2 Validation scenario-3 case B, results

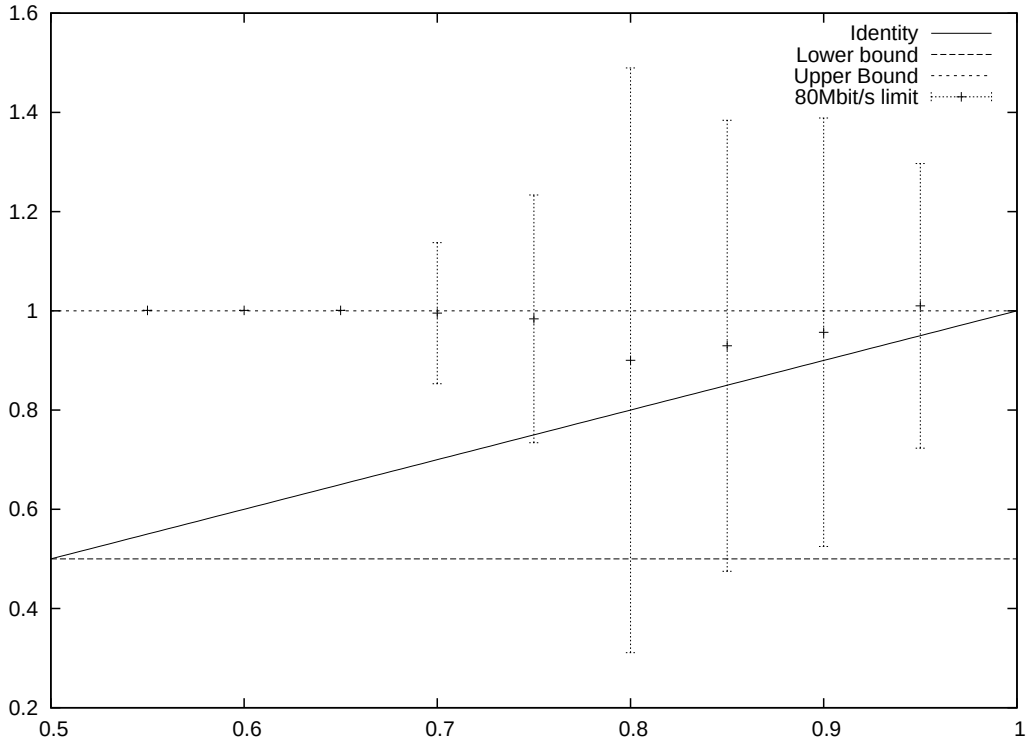


Figure A.2: Results for model validation scenario number 3B. Rate limit 80Mbit/s

Nominal	Mean H	Var H	99% Conf. Int.
0.55	1.0009	2.8399e-29	1.3727e-14
0.60	1.0009	2.8399e-29	1.3727e-14
0.65	1.0009	2.8399e-29	1.3727e-14
0.70	0.9954	0.0030	0.1422
0.75	0.9838	0.0094	0.2497
0.80	0.9004	0.0523	0.5893
0.85	0.9295	0.0311	0.4546
0.90	0.9568	0.0281	0.4318
0.95	1.0100	0.0124	0.2868

Table A.2: Results for *Congestion results in loss of self-similarity* validation scenario, case B

A.3 Validation scenario-3 case C, results

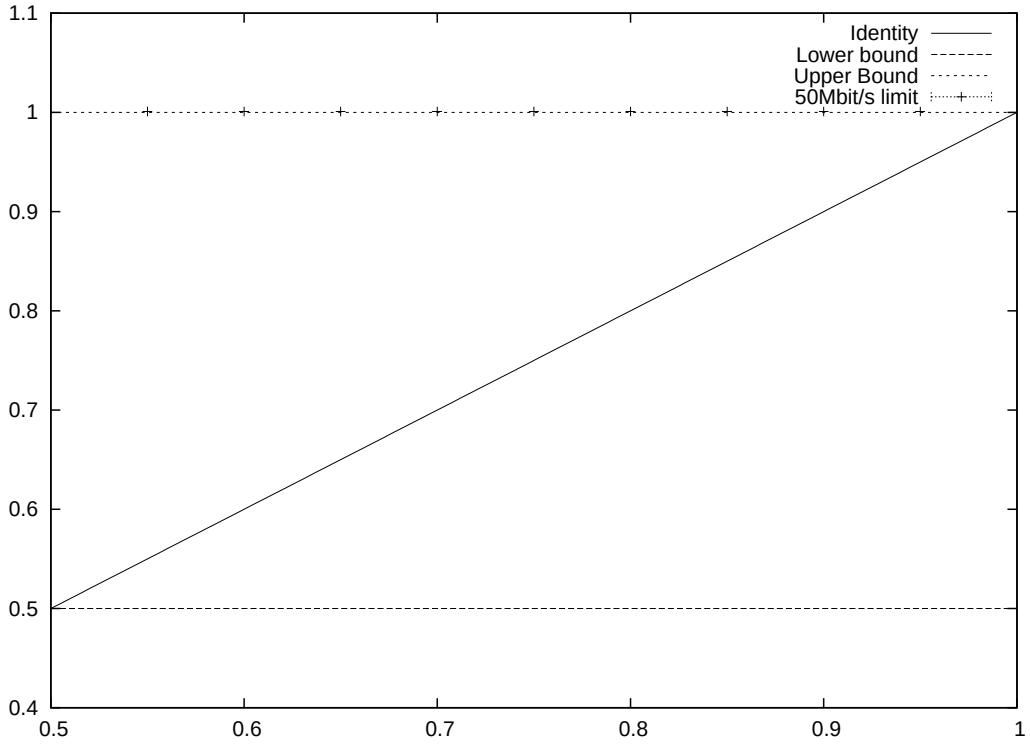


Figure A.3: Results for model validation scenario number 3C. Rate limit 50Mbit/s

Nominal	Mean H	Var H	99% Conf. Int.
0.55	1.0009	1.5974e-29	1.0295e-14
0.60	1.0009	1.5974e-29	1.0295e-14
0.65	1.0009	1.5974e-29	1.0295e-14
0.70	1.0009	1.5974e-29	1.0295e-14
0.75	1.0009	1.5974e-29	1.0295e-14
0.80	1.0009	1.5974e-29	1.0295e-14
0.85	1.0009	1.5974e-29	1.0295e-14
0.90	1.0009	1.5974e-29	1.0295e-14
0.95	1.0009	1.5974e-29	1.0295e-14

Table A.3: Results for *Congestion results in loss of self-similarity* validation scenario, case C

Appendix B

DDoS scenario-1, results

B.1 DDoS scenario-1, fixed Source H = 0.60

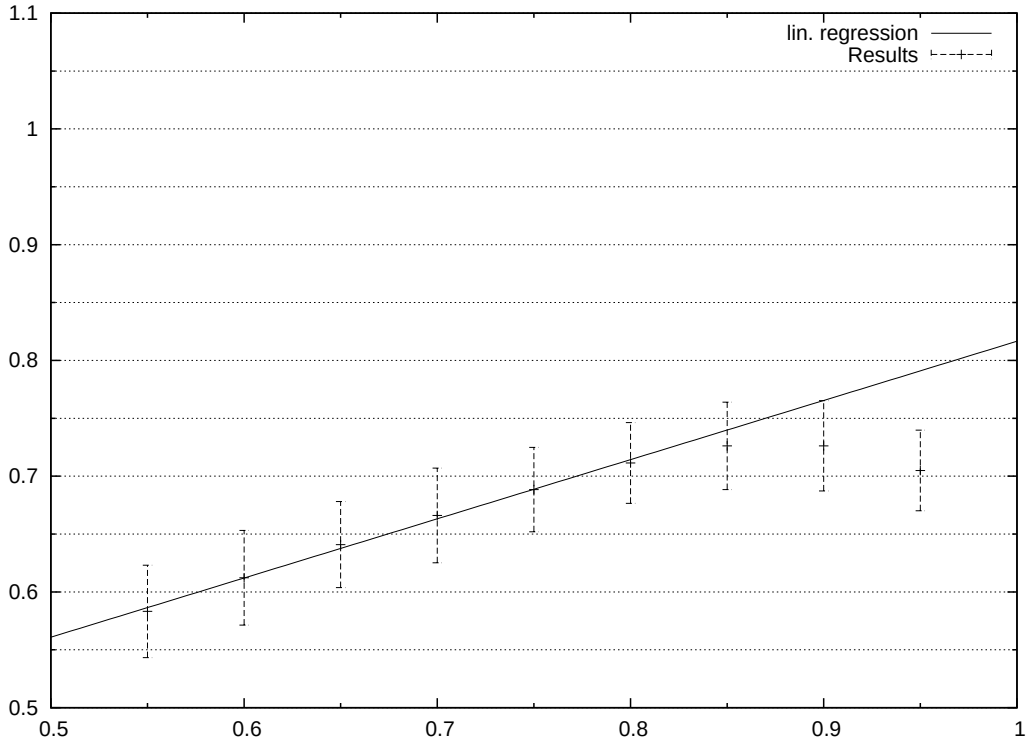


Figure B.1: Results for simulation scenario number one. Hurst parameter value for superimposition of two sources. 1st source fixed to H = 0.60, 2nd source varying from 0.55 to 0.95 in 0.05 steps.

2nd src H	Mean H	Var H	99% Confidence Interval
0.55	0.5832	0.0002	0.0399
0.60	0.6122	0.0003	0.0409
0.65	0.6409	0.0002	0.0372
0.70	0.6661	0.0003	0.0409
0.75	0.6884	0.0002	0.0365
0.80	0.7114	0.0002	0.0349
0.85	0.7262	0.0002	0.0378
0.90	0.7262	0.0002	0.0390
0.95	0.7049	0.0002	0.0348

Table B.1: Results for Scenario-1 simulation with 1st source H = 0.60

B.2 DDoS scenario-1, fixed Source H = 0.65

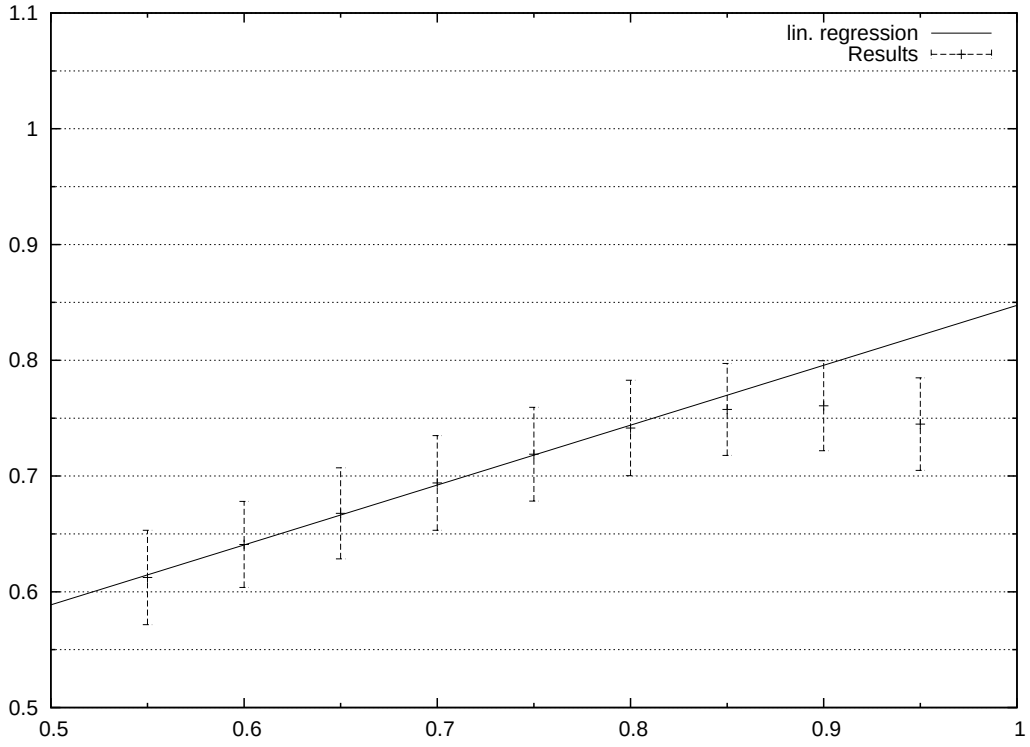


Figure B.2: Results for simulation scenario number one. Hurst parameter value for superimposition of two sources. 1st source fixed to H = 0.65, 2nd source varying from 0.55 to 0.95 in 0.05 steps.

2nd src H	Mean H	Var H	99% Confidence Interval
0.55	0.6124	0.0003	0.0408
0.60	0.6409	0.0002	0.0372
0.65	0.6678	0.0002	0.0393
0.70	0.6940	0.0003	0.0409
0.75	0.7189	0.0002	0.0405
0.80	0.7415	0.0002	0.0412
0.85	0.7575	0.0002	0.0397
0.90	0.7607	0.0002	0.0389
0.95	0.7449	0.0002	0.0400

Table B.2: Results for Scenario-1 simulation with 1st source H = 0.65

B.3 DDoS scenario-1, fixed Source H = 0.70

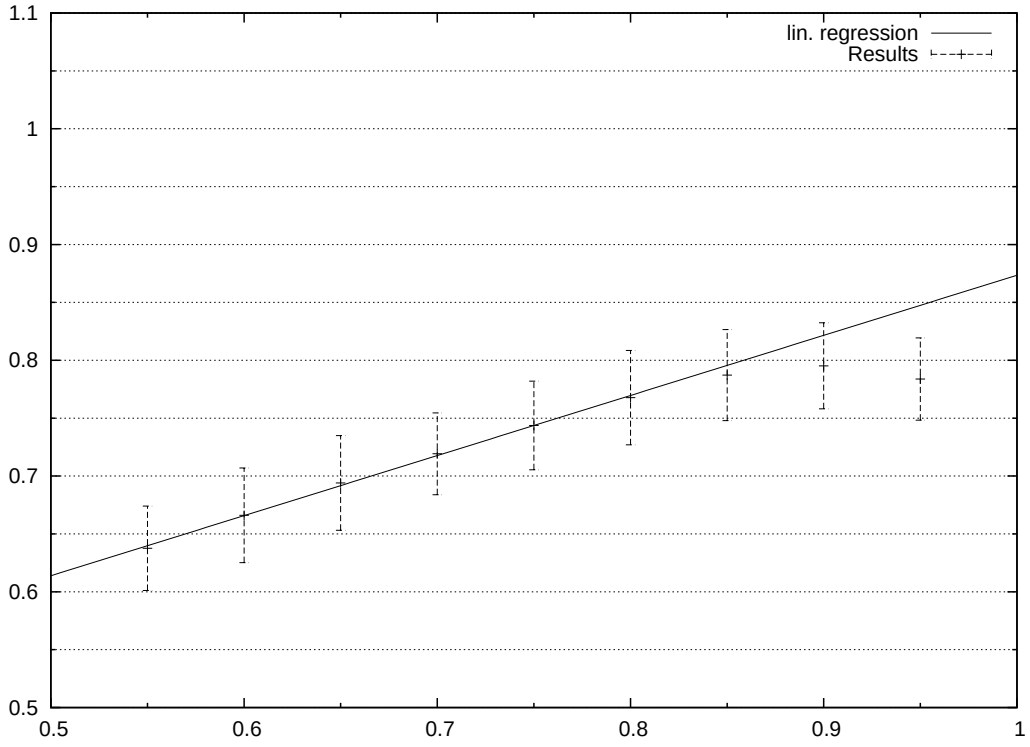


Figure B.3: Results for simulation scenario number one. Hurst parameter value for superimposition of two sources. 1st source fixed to H = 0.70, 2nd source varying from 0.55 to 0.95 in 0.05 steps.

2nd src H	Mean H	Var H	99% Confidence Interval
0.55	0.6376	0.0002	0.0365
0.60	0.6661	0.0003	0.0409
0.65	0.6940	0.0003	0.0409
0.70	0.7192	0.0002	0.0353
0.75	0.7437	0.0002	0.0383
0.80	0.7677	0.0003	0.0408
0.85	0.7872	0.0002	0.0393
0.90	0.7952	0.0002	0.0372
0.95	0.7838	0.0002	0.0355

Table B.3: Results for Scenario-1 simulation with 1st source H = 0.70

B.4 DDoS scenario-1, fixed Source H = 0.75

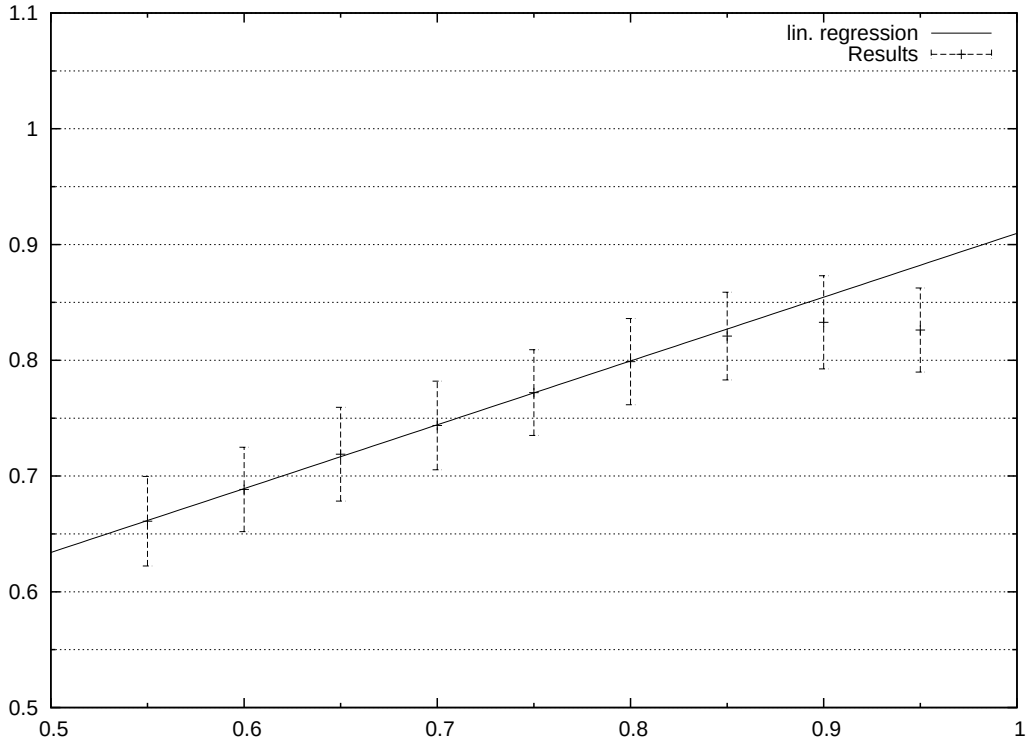


Figure B.4: Results for simulation scenario number one. Hurst parameter value for superimposition of two sources. 1st source fixed to H = 0.75, 2nd source varying from 0.55 to 0.95 in 0.05 steps.

2nd src H	Mean H	Var H	99% Confidence Interval
0.55	0.6610	0.0002	0.0387
0.60	0.6884	0.0002	0.0365
0.65	0.7189	0.0002	0.0405
0.70	0.7437	0.0002	0.0383
0.75	0.7721	0.0002	0.0371
0.80	0.7988	0.0002	0.0372
0.85	0.8209	0.0002	0.0379
0.90	0.8328	0.0002	0.0402
0.95	0.8261	0.0002	0.0363

Table B.4: Results for Scenario-1 simulation with 1st source H = 0.75

B.5 DDoS scenario-1, fixed Source H = 0.80

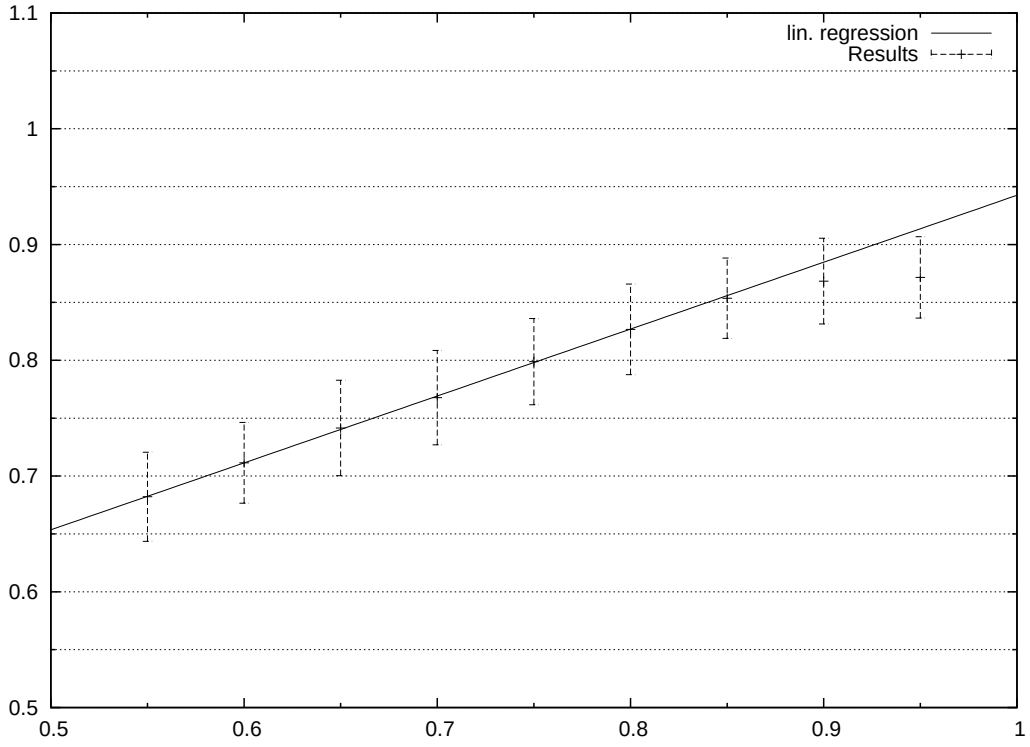


Figure B.5: Results for simulation scenario number one. Hurst parameter value for superimposition of two sources. 1st source fixed to H = 0.80, 2nd source varying from 0.55 to 0.95 in 0.05 steps.

2nd src H	Mean H	Var H	99% Confidence Interval
0.55	0.6821	0.0002	0.0385
0.60	0.7114	0.0002	0.0349
0.65	0.7415	0.0002	0.0412
0.70	0.7677	0.0003	0.0408
0.75	0.7988	0.0002	0.0372
0.80	0.8267	0.0002	0.0391
0.85	0.8536	0.0002	0.0347
0.90	0.8684	0.0002	0.0371
0.95	0.8716	0.0002	0.0352

Table B.5: Results for Scenario-1 simulation with 1st source H = 0.80

B.6 DDoS scenario-1, fixed Source H = 0.85

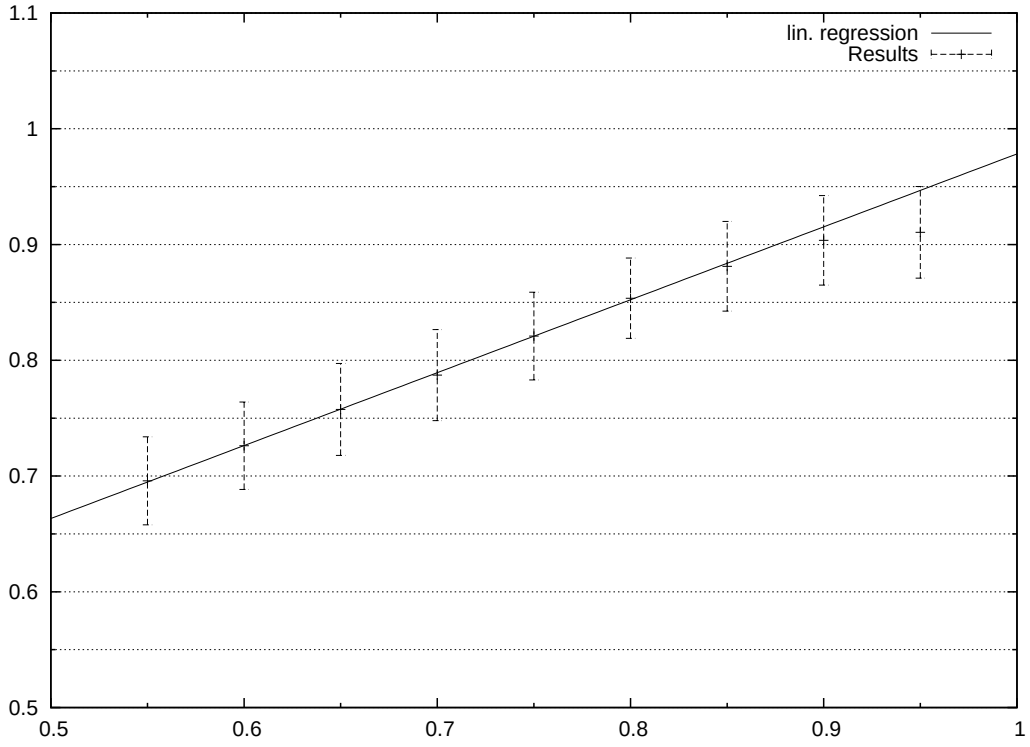


Figure B.6: Results for simulation scenario number one. Hurst parameter value for superimposition of two sources. 1st source fixed to H = 0.85, 2nd source varying from 0.55 to 0.95 in 0.05 steps.

2nd src H	Mean H	Var H	99% Confidence Interval
0.55	0.6959	0.0002	0.0380
0.60	0.7262	0.0002	0.0378
0.65	0.7575	0.0002	0.0397
0.70	0.7872	0.0002	0.0393
0.75	0.8209	0.0002	0.0379
0.80	0.8536	0.0002	0.0347
0.85	0.8812	0.0002	0.0388
0.90	0.9036	0.0002	0.0387
0.95	0.9105	0.0002	0.0396

Table B.6: Results for Scenario-1 simulation with 1st source H = 0.85

B.7 DDoS scenario-1, fixed Source H = 0.90

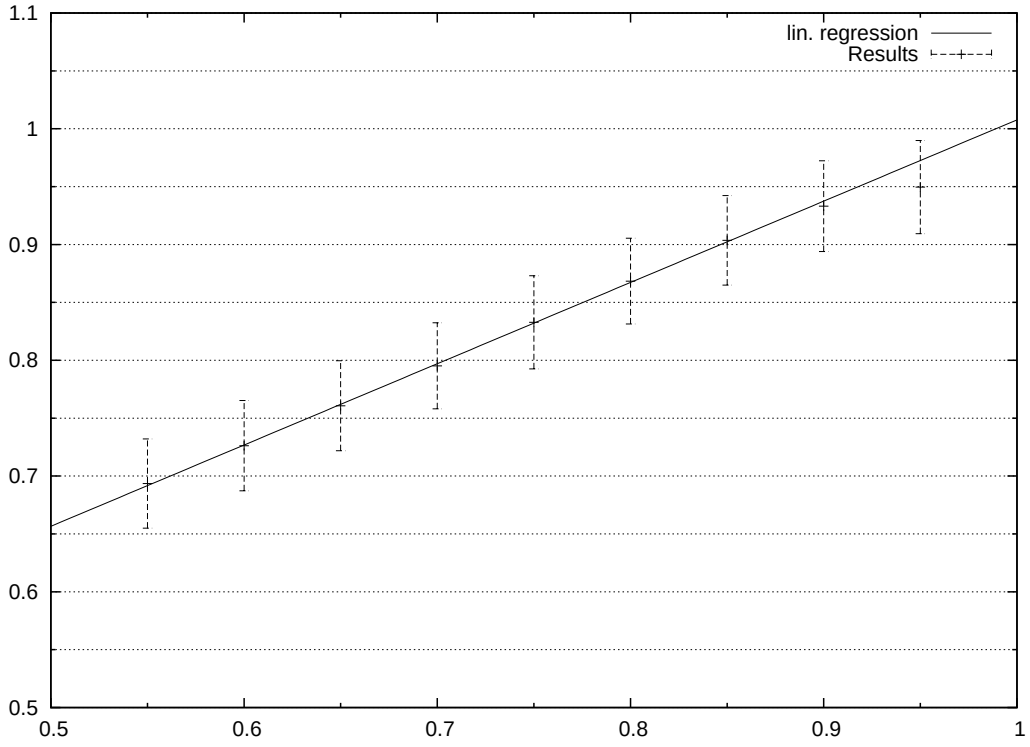


Figure B.7: Results for simulation scenario number one. Hurst parameter value for superimposition of two sources. 1st source fixed to H = 0.90, 2nd source varying from 0.55 to 0.95 in 0.05 steps.

2nd src H	Mean H	Var H	99% Confidence Interval
0.55	0.6935	0.0002	0.0385
0.60	0.7262	0.0002	0.0390
0.65	0.7607	0.0002	0.0389
0.70	0.7952	0.0002	0.0372
0.75	0.8328	0.0002	0.0402
0.80	0.8684	0.0002	0.0371
0.85	0.9036	0.0002	0.0387
0.90	0.9331	0.0002	0.0392
0.95	0.9496	0.0002	0.0402

Table B.7: Results for Scenario-1 simulation with 1st source H = 0.90

B.8 DDoS scenario-1, fixed Source H = 0.95

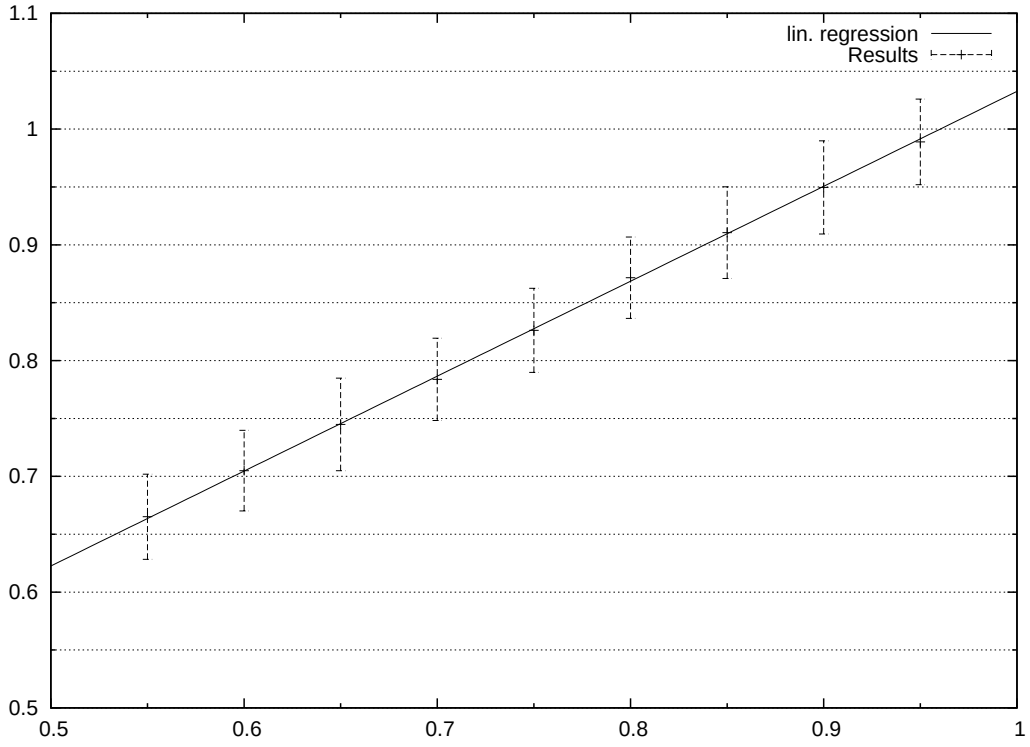


Figure B.8: Results for simulation scenario number one. Hurst parameter value for superimposition of two sources. 1st source fixed to H = 0.95, 2nd source varying from 0.55 to 0.95 in 0.05 steps.

2nd src H	Mean H	Var H	99% Confidence Interval
0.55	0.6651	0.0002	0.0368
0.60	0.7049	0.0002	0.0348
0.65	0.7449	0.0002	0.0400
0.70	0.7838	0.0002	0.0355
0.75	0.8261	0.0002	0.0363
0.80	0.8716	0.0002	0.0352
0.85	0.9105	0.0002	0.0396
0.90	0.9496	0.0002	0.0402
0.95	0.9889	0.0002	0.0370

Table B.8: Results for Scenario-1 simulation with 1st source H = 0.95

Appendix C

DDoS scenario-2, results

C.1 Intensity ratios scenario results (0.55, 0.75)

C.2 Intensity ratios scenario results (0.55, 0.75)

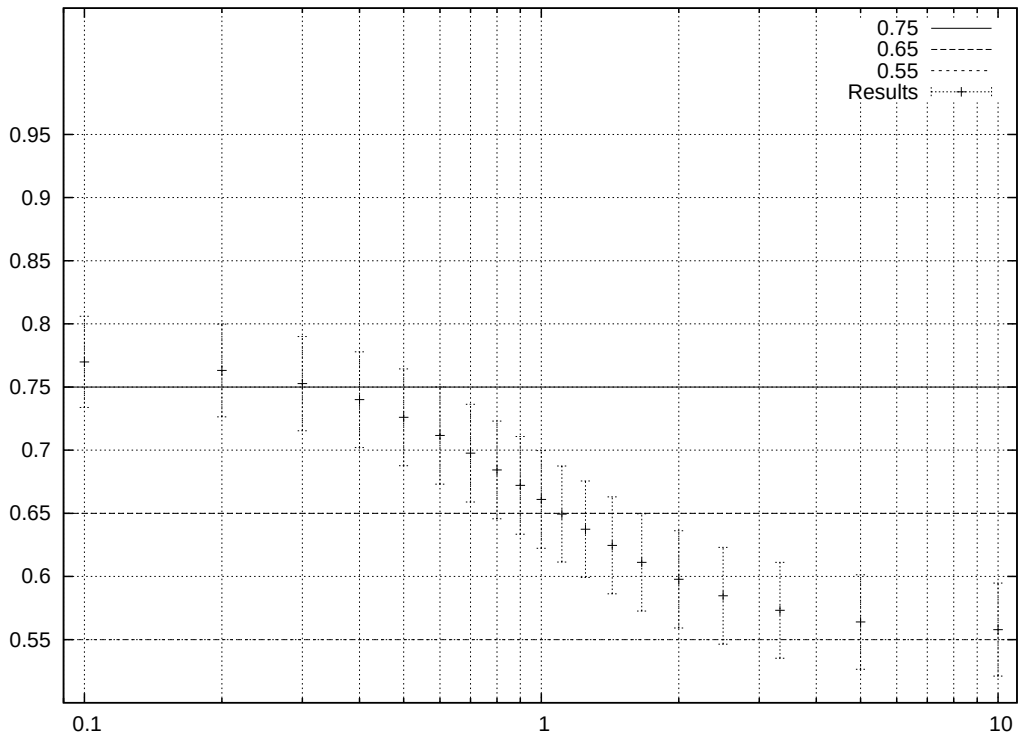


Figure C.1: Results for simulation scenario number two. Hurst parameter value for superimposition of two sources. 1st source fixed to H = 0.55, 2nd source fixed to 0.75, intensity ratio from 1:10 to 10:1.

Ratio	Mean H	Var H	99% Confidence Interval
1:10	0.9688	0.0003	0.0418
2:10	0.9302	0.0003	0.0417
3:10	0.8819	0.0002	0.0412
4:10	0.8342	0.0002	0.0405
5:10	0.7917	0.0002	0.0397
6:10	0.7559	0.0002	0.0390
7:10	0.7262	0.0002	0.0383
8:10	0.7017	0.0002	0.0377
9:10	0.6816	0.0002	0.0372
10:10	0.6651	0.0002	0.0368
10:9	0.6514	0.0002	0.0359
10:8	0.6359	0.0002	0.0360
10:7	0.6207	0.0002	0.0360
10:6	0.6061	0.0002	0.0361
10:5	0.5925	0.0002	0.0361
10:4	0.5803	0.0002	0.0361
10:3	0.5701	0.0002	0.0361
10:2	0.5624	0.0002	0.0361
10:1	0.5576	0.0002	0.0361

Table C.1: Results for simulation scenario number two. Hurst parameter value for superimposition of two sources. 1st source fixed to H = 0.55, 2nd source fixed to 0.95, intensity ratio from 1:10 to 10:1.

Ratio	Mean H	Var H	99% Confidence Interval
1:10	0.7699	0.0002	0.0361
2:10	0.7631	0.0002	0.0367
3:10	0.7527	0.0002	0.0373
4:10	0.7400	0.0002	0.0379
5:10	0.7260	0.0002	0.0383
6:10	0.7116	0.0002	0.0385
7:10	0.6976	0.0002	0.0387
8:10	0.6843	0.0002	0.0387
9:10	0.6721	0.0002	0.0387
10:10	0.6610	0.0002	0.0387
10:9	0.6494	0.0002	0.0380
10:8	0.6374	0.0002	0.0382
10:7	0.6246	0.0002	0.0384
10:6	0.6111	0.0002	0.0385
10:5	0.5977	0.0002	0.0385
10:4	0.5847	0.0002	0.0383
10:3	0.5732	0.0002	0.0380
10:2	0.5639	0.0002	0.0374
10:1	0.5579	0.0002	0.0368

Table C.2: Results for simulation scenario number two. Hurst parameter value for superimposition of two sources. 1st source fixed to H = 0.55, 2nd source fixed to 0.75, intensity ratio from 1:10 to 10:1.

C.3 Intensity ratios scenario results (0.75, 0.95)

Ratio	Mean H	Var H	99% Confidence Interval
1:10	0.9776	0.0002	0.0415
2:10	0.9608	0.0002	0.0412
3:10	0.9384	0.0002	0.0407
4:10	0.9149	0.0002	0.0399
5:10	0.8931	0.0002	0.0391
6:10	0.8743	0.0002	0.0383
7:10	0.8585	0.0002	0.0376
8:10	0.8455	0.0002	0.0370
9:10	0.8349	0.0002	0.0366
10:10	0.8261	0.0002	0.0363
10:9	0.8191	0.0002	0.0378
10:8	0.8111	0.0002	0.0376
10:7	0.8034	0.0002	0.0374
10:6	0.7960	0.0002	0.0371
10:5	0.7893	0.0002	0.0369
10:4	0.7834	0.0002	0.0367
10:3	0.7787	0.0002	0.0364
10:2	0.7751	0.0002	0.0361
10:1	0.7730	0.0002	0.0358

Table C.3: Results for simulation scenario number two. Hurst parameter value for superimposition of two sources. 1st source fixed to H = 0.75, 2nd source fixed to 0.95, intensity ratio from 1:10 to 10:1.

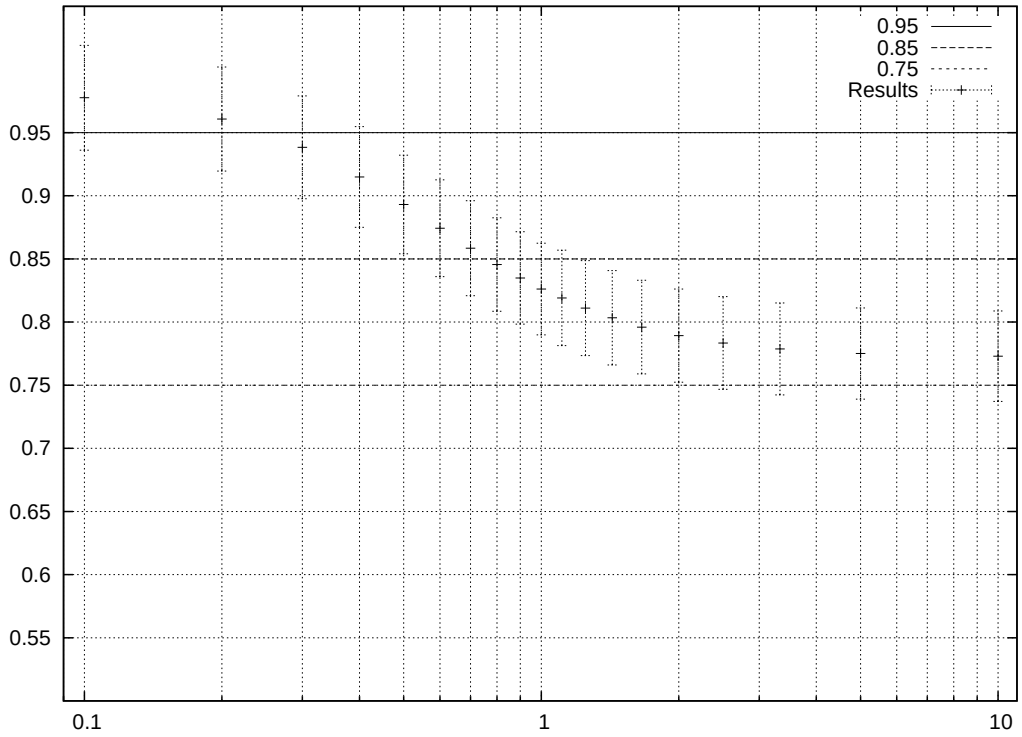


Figure C.2: Results for simulation scenario number two. Hurst parameter value for superimposition of two sources. 1st source fixed to H = 0.75, 2nd source fixed to 0.95, intensity ratio from 1:10 to 10:1.

Appendix D

Acronyms

- DoS : Denial of Service.
- DDoS : Distributed Denial of Service.
- IP : Internet Protocol.
- ISP : Internet Service Provider.
- LAN : Local Area Network.
- LILO : Last In Last Out.
- LRD : Long Range Dependence.
- HTTP : Hyper Text Transfer Protocol.
- NIC : Network Interface Controller.
- OS : Operating System.
- UDP : User Datagram Protocol.
- RTT : Round-Trip Time.
- TCP : Transfer Control Protocol.
- WWW : World Wide Web.
- XML : eXtensible Markup Language.

Appendix E

References

E.1 Bibliography

[AAVV98] P. Abry, P. Abry, D. Veitch, and D. Veitch. Wavelet analysis of long-range-dependent traffic. *Information Theory, IEEE Transactions on*, 44:2–15, 1998.

[AM04] W.H. Allen and G.A. Marin. The loss technique for detecting new denial of service attacks. In *SoutheastCon, 2004. Proceedings. IEEE*, pages 302–309, 2004.

[BUBS97] M.S. Borella, S. Uludag, G.B. Brewster, and I. Sidhu. Self-similarity of internet packet delay. In *Communications, 1997. ICC 97 Montreal, 'Towards the Knowledge Millennium'. 1997 IEEE International Conference on*, volume 1, pages 513–517 vol.1, 1997.

[Cam05] P.L. Campbell. The denial-of-service dance. *Security & Privacy Magazine, IEEE*, 3:34– 40, 2005.

[CB97] Mark E. Crovella and Azer Bestavros. *Self-similarity in World Wide Web traffic: evidence and possible causes*, volume 5. IEEE Press, 1997.

[Cha02] R.K.C. Chang. Defending against flooding-based distributed denial-of-service attacks: a tutorial. *Communications Magazine, IEEE*, 40:42– 51, 2002.

[CS05] S. Chen and Q. Song. Perimeter-based defense against high bandwidth ddos attacks. *Parallel and Distributed Systems, IEEE Transactions on*, 16:526– 537, 2005.

[DLD00] S. Dietrich, N. Long, and D. Dittrich. Analyzing distributed denial of service tools: The shaft case. *Proceedings of the 14th USENIX conference on System administration*, pages 329–340, 2000.

[DZL06] David Dagon, Cliff Zou, and Wenke Lee. Modeling botnet propagation using time zones. 2006.

[ERVW02] A. Erramilli, M. Roughan, D. Veitch, and W. Willinger. Self-similar traffic and network dynamics. *Proceedings of the IEEE*, 90:800–819, 2002.

[FA96] Russell Fischer and Metin Akay. A comparison of analytical methods for the study of fractional brownian motion. *Annals of Biomedical Engineering*, 24:537–543, July 1996.

[FGW98] Anja Feldmann, Anna C. Gilbert, and Walter Willinger. Data networks as cascades: Investigating the multifractal nature of internet wan traffic. pages 42–55, 1998.

[fSC08] The R Foundation for Statistical Computing. The r project for statistical computing, 2008. http://www.r-project.org/.

[GP01] Thomer M. Gil and Massimiliano Poletto. Multops: A data-structure for bandwidth attack detection. *Proceedings of the USENIX Security Symposium*, pages 23–38, July 2001.

[HBS65] H. E. Hurst, R. P. Black, and Y. M. Simaika. *Long-term storage: an experimental study*. Constable, 1965.

[HKBN07] G. Horn, A. Kvalbein, J. Blomskold, and E. Nilsen. An empirical comparison of generators for self similar simulated traffic. *Performance Evaluation*, 64:162–190, February 2007.

[KLCC06] Yoohwan Kim, Wing Cheong Lau, Mooi Choo Chuah, and H. Jonathan Chao. Packetscore: A statistics-based packet filtering scheme against distributed denial-of-service attacks. *IEEE Transactions on Dependable and Secure Computing*, 3:141, June 2006.

[KS02a] T. Kushida and Y. Shibata. Empirical study of inter-arrival packet times and packet losses. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 233– 238, 2002.

[Li04] Ming Li. An approach to reliably identifying signs of ddos flood attacks based on lrd traffic pattern recognition. *Computers & Security*, 23:549–558, October 2004.

[Li06] Ming Li. Change trend of averaged hurst parameter of traffic under ddos flood attacks. *Computers & Security*, 25:213–220, May 2006.

[LLY05] T.K.T. Law, J.C.S. Lui, and D.K.Y. Yau. You can run, but you can't hide: an effective statistical methodology to trace back ddos attackers. *Parallel and Distributed Systems, IEEE Transactions on*, 16:799– 813, 2005.

[LTWW94] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Trans. Netw*, 2:1–15, 1994.

[Mal89] S. G. Mallat. A theory for multiresolution signal decomposition: The wavelet representation, July 1989.

[Man67] Benoit Mandelbrot. How long is the coast of britain? statistical self-similarity and fractional dimension. *Science*, 156:636–638, May 1967.

[Man82] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman, August 1982.

[MAW08] MAWI. Mawi working group traffic archive, 2008. http://mawi.wide.ad.jp/mawi/.

[McC03] B. McCarty. Botnets: big and bigger. *Security & Privacy Magazine, IEEE*, 1:87– 90, 2003.

[MR05] Jelena Mirkovic and Peter Reiher. D-ward: A source-end defense against flooding denial-of-service attacks. *IEEE Transactions on Dependable and Secure Computing*, 2:216, September 2005.

[Oli08] Travis Oliphant. Numpy home page, 2008. http://numpy.scipy.org/.

[pa08] pcap authors. Pcap library, 2008. http://www.tcpdump.org/.

[Pax97] Vern Paxson. *Fast, approximate synthesis of fractional Gaussian noise for generating self-similar network traffic*, volume 27. ACM Press, 1997.

[PPFF95] V. Paxson, V. Paxson, S. Floyd, and S. Floyd. Wide area traffic: the failure of poisson modeling. *Networking, IEEE/ACM Transactions on*, 3:226–244, 1995.

[PT97] Jon D. Pelletier and Donald L. Turcotte. Long-range persistence in climatological and hydrological time series: analysis, modeling and application to drought hazard assessment. *Journal of Hydrology*, 203:198–208, December 1997.

[Row08] Mark Rowe. Python packet capture and injection library, 2008. http://pycap.sourceforge.net/.

[Sch06] G.P. Schaffer. Worms and viruses and botnets, oh my! rational responses to emerging internet threats. *Security & Privacy Magazine, IEEE*, 4:52– 58, 2006.

[SCVK02] B. Sikdar, K. Chandrayana, K.S. Vastola, and S. Kalyanaraman. Queue management algorithms and network traffic self-similarity. In *High Performance Switching and Routing, 2002. Merging Optical and IP Technologies. Workshop on*, pages 319– 323, 2002.

[Shi08] Clay Shields. What do we mean by network denial of service? *Proceedings of the 2002 IEEE Workshop on Information Assurance and Security*, 2008.

[SM01] W. Schleifer and M. Mannle. Online error detection through observation of traffic self-similarity. *Communications, IEE Proceedings-*, 148:38–42, 2001.

[TTW95] M. Taqqu, V. Teverovsky, and W. Willinger. *Estimators for long-range dependence: an empirical study*. 1995.

[TV03] Udaya Kiran Tupakula and Vijay Varadharajan. A practical method to counteract denial of service attacks. pages 275–284, Adelaide, Australia, 2003. Australian Computer Society, Inc.

[TWS97] Murad S. Taqqu, Walter Willinger, and Robert Sherman. Proof of a fundamental result in self-similar traffic modeling. *SIGCOMM Comput. Commun. Rev.*, 27:5–23, 1997.

[VA99a] Darryl Veitch and Patrice Abry. A statistical test for the time constancy of scaling exponents. 1999.

[VA99b] Darryl Veitch and Patrice Abry. A wavelet based joint estimator of the parameters of long-range dependence. *IEEE Transactions on Information Theory*, 45:878–897, 1999.

[vR08] Guido van Rossum. Python programming language – official website, 2008. http://python.org/.

[Wel67] P. Welch. The use of fast fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. *Audio and Electroacoustics, IEEE Transactions on*, 15:70–73, 1967.

[WTSW97] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. *Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level*, volume 5. IEEE Press, 1997.

[WZS04] Haining Wang, Danlu Zhang, and Kang G. Shin. Change-point monitoring for the detection of dos attacks. *IEEE Transactions on Dependable and Secure Computing*, 1:193, December 2004.

[XLLH04] Y. Xiang, Y. Lin, W.L. Lei, and S.J. Huang. Detecting ddos attack based on network self-similarity. *Communications, IEE Proceedings-*, 151:292– 295, 2004.

[XLS01] Yong Xiong, S. Liu, and P. Sun. On the defense of the distributed denial of service attacks: an on-off feedback control approach. *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, 31:282–293, 2001.