Geist: A **G**enerator for **E**-Commerce & **I**nternet **S**erver **T**raffic

Krishna Kant, Vijay Tewari and Ravi Iyer
Technology & Research Labs
Intel Corporation
Beaverton, OR
{krishna.kant|vijay.tewari|ravishankar.iyer}.intel.com

Abstract

This paper describes Geist, a traffic generator for stress testing of web servers. The generator provides a large number of dialable parameters that allow traffic characteristics to range from simple static web-page browsing to the transactional traffic seen by e-commerce front end servers. Unlike other traffic generators, our generator concentrates on the characteristics of the aggregate traffic arriving at the server, which allows for better control of the scaling properties of the traffic and a more scalable generation. In this paper, we describe the traffic characterization and generation process that Geist is based on. We also present the performance of our current implementation, instances of its usage and directions for future work.

1 Introduction

The work reported here was motivated by the need to understand the implications of web traffic characteristics on the architectural aspects of a web/e-commerce server. This requires generation of realistic Internet traffic from a server's perspective without having to explicitly emulate network components or protocols. This server-centric focus also means that client-level aspects of the traffic are relevant only to the extent they affect the aggregate traffic. Furthermore, lab testing environment often requires that the servers be driven to their capacity (and in fact beyond the capacity into overload region). This requires the capability to generate heavy traffic (e.g., more than 10K requests per second). It has been amply noted in the literature that the traditional socket interface makes overloading web-servers quite difficult [1, 2, 24]. Finally, the increasing dynamic content of web pages makes generation of realistic transactional characteristics particularly challenging, especially in e-commerce environments. In this paper, we present our Internet traffic generator, Geist, that has been designed to deal with these and other important issues.

The server centric view of Geist brings into focus two different methods of generating traffic: (a) user emulation, and (b) aggregate traffic generation. In the first approach, behavior of individual user is emulated in some detail and the aggregate traffic is simply the super-position of traffic generated by individual users. The second approach generates the aggregate traffic directly. Section 2 discusses the pros and cons of these two approaches and motivates the aggregate traffic generation approach adopted by Geist.

Internet traffic is known to show complex temporal characteristics including long-range dependence, traffic irregularities at intermediate time scales, and nonstationarity. Further complexity arises in the transactional composition of the requests which not only retrieve stored "files" but also often dynamically construct web pages by running scripts, perhaps embedded with queries to backend databases. Thus the traffic generator should be able to control both temporal and transactional properties of the traffic. In an e-commerce environment, certain transactions may be secure, and such transactions are comparatively very expensive on both client and server ends [17]. Geist, unlike many other traffic generators can generate mixed secure/nonsecure traffic.

Generation of aggregate traffic with complex characteristics is often computationally intensive, which makes it difficult to ensure that a request is actually generated very close to the intended time. Geist addresses this issue by splitting the generation into two steps referred to as *trace generation* and *traffic generation* respectively. Trace generation handles all the complexities of computing the actual time and parameters for the requests, whereas the traffic generation step simply reads the trace and issues the requests. The added advantage of this approach is that the trace could very well have been derived from HTTP logs from a live site. The ability to use actual traces is crucial since the complex features of a live traffic may be very difficult to reproduce in statistically generated traffic. Finally, irrespective of how the trace is generated, the traffic generation step needs to be designed to provide sufficient scalability, flexibility and efficiency. To accomplish this, various system issues such as process scheduling, traditional socket interface and TCP window flow control need to be kept in mind when designing the traffic generator. The design of Geist's traffic generation component is presented in detail in Section 4.

In order to understand the performance aspects of our traffic generator, we have used GEIST for generating traffic in a number of projects such as server overload control, SSL performance study and proxy server evaluation. In addition, we have performed tests to make sure that the traffic generated by GEIST preserves the properties of the requested traffic. These and other issues related to the accuracy of the traffic generator is discussed in detail in Section 5.

In related work, some other well known tools are Microsoft's Web Application Stress Tool (WAST) [23], HTTP-perf [24] from HP Labs, and SURGE [5] from Boston University. These tools mostly take the user emulation approach whereas Geist attempts to generate aggregate traffic directly. The first two tools are very limited in terms of the temporal properties that the traffic is allowed to have. SURGE, on the other hand, supports self-similarity by making each "user equivalent" an on-off process. In contrast, Geist supports asymptotically self-similar, multifractal, and non-stationary arrival process. However, Geist is not locked into producing self-similar traffic; it can also generate traffic with much shorter dependencies, including none. Geist also supports detailed transactional characterization of the traffic. However, SURGE does explicitly support embedded requests, a feature, that is currently not implemented in Geist. Geist's model for temporal locality in accesses is taken from SPECweb96/99 benchmarks, rather than emulated explicitly as in SURGE. Unlike the current version of Geist, the HTTP-perf tool supports multiple requests per connection and cookies; however, it is designed basically for deterministic inter-request time on a single client. Of the above tools, only Geist and WAST can generate a mix of non-secure and secure (using SSL) traffic.

2 Aggregate Traffic Generation vs. User Emulation

In this section we discuss advantages and disadvantages of user emulation vs. aggregate traffic generation. The purpose of the discussion is to motivate the decisions made in the design of Geist.

User emulation involves creating a process or thread that mimics the actions of an individual user. In such a case, traffic characterization must be in terms of behavior of an individual user. In particular, the important traffic parameters include think-time (time between receiving response to the previous request and making a new one), correlations between successive think times, type of transaction issued, dependencies between successive transactions, abandonments, retries, etc. In the web environment, a single request from the user actually results in downloading of multiple *embedded* files (usually advertisements, images, logos, client-side scripts, etc.) in addition to the requested document. All these characteristics are easily represented by the user emulation technique.

Although user emulation allows fine control over behavioral aspects, there are a few serious drawbacks of this approach in the context of exercising an Internet server with synthetic traffic. In such experiments, an important goal is to control certain aggregate properties for the traffic hitting the server. Thus the user emulation approach would require establishing a relationship between the aggregate properties and that of individual users. In general, the relationship between the aggregate and individual processes is quite involved, which makes the control of aggregate traffic parameters difficult using the user emulation approach.

The second problem with user emulation is that the traffic seen by the server is not merely a function of the user behavior but also that of highly variable network delays, retransmissions, segmentation/reassembly, etc.. In a laboratory setup intended to test a server, the network is typically a single high-speed (e.g., Gigabit) LAN segment with negligible latency. Thus, in order to accurately reflect the traffic hitting the server in real-life, the traffic generated by the clients should already include the network effects. Accurately accounting for the network impact on the individual user traffic is again very difficult in general.

The third problem with user emulation is scalability: testing a server at a high throughput level may require many more processes or threads than the O/S can efficiently support. Moreover, since a user cannot generate the next request until the previous one has been satisfied, it is difficult to do stress testing of the server. Basically, the offered load will reduce as the server load increases (because of increased response times), and the ability to independently control the offered load is lost.

A direct generation of aggregate traffic can address these problems easily. In particular, in section 3.1 we discuss the asymptotic self-similarity, multifractal properties, and nonstationarity that are commonly observed in Internet traffic, and show a way of generating traffic with those properties. Since the goal in aggregate traffic generation is to issue requests at the correct time, one process or thread can keep track of multiple requests. Also, the responses can be handled by separate processes/threads, which means that the generation of next request is not conditioned on receiving response for the previous request. Thus, the method can support large offered loads independent of the server load. The main implementation problem here is timely handling of the responses without having to commit threads/processes at the time of request.

By its very nature, aggregate traffic generation has no notion of individual users; therefore, it may run into difficulties when detailed use-level modeling is needed. For example, in an e-commerce environment, the successive transactions issued by a given user must be in proper order to be sensible (e.g., a purchase must follow the shopping cart transaction). Similarly, handling of user abandonments and retries requires identification of actions of individual users. Geist handles these issues by identifying representative *virtual user sessions* within the aggregate stream and marking those with session-id, transaction type and other information to ensure that ordering constraints are met. These issues are discussed in more detail in section 3.4.

3 Trace Generation

In this section, we discuss the major issues in generating a request trace. In Section 3.1 we review the complex temporal properties shown by Internet traffic and their characterization. Section 3.2 discusses how Geist provides control over these properties in the generated traffic. Section 3.3 discusses issues concerning the nonstationarity of Internet traffic and how to generate such traffic. Sections 3.4 and 3.5 discuss transactional properties and other characteristics of the retrieved objects. The control of a number of temporal and transactional properties of the trace is done via a number of parameters. Appendix C shows a sample input file specifying these parameters. The trace generator writes the trace to a file, which can be further split up for use by each client in the traffic generator part. The trace generator also includes some additional capabilities such as simulation of simple queuing systems, emulation of file-cache, and the creation of the file-system that will be exercised by the traffic generator. In this paper, however, we only discuss the trace generation issues.

3.1 Temporal Properties of Web Traffic

The temporal properties of traffic can be examined either in terms of inter-arrival times or the arrival counting process (i.e., number of arrivals in a "slot" of certain duration). The latter is usually more robust, and will be used here as well. An appropriate slot duration is the time-period over which the average number of arrivals are in the range of a few 10s. Apart from the marginal distribution (or rather the first two moments of it), the correlation structure of the arrival process is crucial for temporal characterization and needs to be addressed carefully. The next two subsections briefly discuss two major effects that shape the correlation structure of the request traffic arriving at a server.

3.1.1 User Behavior

It is well-known by now that the behavior of the user during a session alternates between high activity and quiet periods, which can be conveniently captured by an on-off random process. Furthermore, the durations of on and off periods tend to be heavy-tailed, i.e., there is significant probability of encountering extremely long periods. A super-position of a large number of such arrival processes can be shown to exhibit long-range dependence or *self-similarity* [32]. That is, the aggregated process that operates at time scales of milliseconds (e.g., a web server experiencing a few hundred hits per second) would show very substantial correlations over time

scales of the order of minutes or longer. A large number of measurement studies in the literature suggest self-similarity to be a universal property for high-speed networks [22, 6]. Most of these studies have taken a "network centric view" where the traffic metric of interest is the bytes traveling over a link. In contrast, ours has always been a "server centric view", where we are primarily concerned with the requests coming into the server and the responses generated. Since the on-off and heavy-tailed property is typically observed at multiple time-scales, it is not surprising that the request count process will also show properties similar to byte count process. Our extensive analysis of web and e-commerce request traffic [16, 19, 33] confirms this. Appendix A provides a brief introduction to self-similarity and discusses certain concepts that are needed throughout this paper.

The correlations in the arrivals process (and, in particular, the self-similarity) can substantially affect the queuing behavior of the traffic and must be accurately reflected in the generated traffic depending on the time-scales captured by the queue [8, 12, 27, 11]. Since we do not know a priori what time-scales are of interest, it is crucial that the traffic generator be able to generate self-similar traffic, if needed. Geist provides this capability. In fact, Geist also supports generation of non self-similar traffic as well (including both medium and short range dependent traffic).

3.1.2 Network Dynamics Effects

As the traffic passes through the network its characteristics are affected not only by the usual queuing/processing delays at the nodes but also network level mechanisms such as TCP flow and congestion control and segmentation/reassembly of packets at intermediate nodes. The time constants of these activities usually fall in 100 ms range which is much less than the time constants involved in user activities. Consequently, self-similarity aspects of the traffic are typically not affected by the network dynamics. Nevertheless, network dynamics has a substantial influence on traffic characteristics and hence on its queuing performance.

A direct emulation of network effects on the traffic makes sense only for network-centric studies; with server-centric studies that we are interested in, representation of network effects would require too much additional machinery. Thus an indirect representation of these effects in the aggregate traffic is ideal for our purposes. It has been shown recently that these effects can be captured by using the concept of *multi-fractal* properties. Multifractality is defined as an extension of self-similarity. Basically, the idea is to capture finer time-scale properties by considering how the higher order moments of the aggregated process decay with the time-scale. For a self-similar process, the nature of this decay is fixed by the H parameter; however, for more general processes, the decay rate could depend on the order of the moment. Reference [10] shows how the multifractal behavior can account for the network dynamics which can be thought as driven by a multiplicative process, much like the *cascade construction* process that results in exact multifractals [13]. As with self-similarity, much of the work to date on multifractal properties considers byte traffic on a network link; however, given the rather small sizes of requests, similar properties apply to request level traffic also. Geist provides the capability of introducing multifractal-like properties in the traffic via a cascade construction process, in case such effects are deemed important. Of course, the user can choose not to introduce these effects.

3.2 Basic Trace Generation

Geist controls the temporal properties of the traffic by using the M/G/∞ paradigm that can generate asymptotically self-similar as well as short and medium range dependent traffic. Appendix B provides a brief discussion of why we chose the M/G/∞ model and how it can generate traffic with different correlational properties. As such, the marginal distribution of the generated arrival process is Poisson (the occupancy distribution in a M/G/∞ queue), however, it can be easily transformed to a process with the desired distribution as detailed in [21]. Such a transformation is adequate at least as far as second order properties are concerned.

The next issue is the introduction of multifractal properties in the traffic, if desired. This aspect is covered in [18]; therefore, the discussion here will be very brief. Basically, we use the *semi-random cascade generator*, which is a random variable C over the interval [0,1] with a mean of 1/2 [10]. To start with, the arrivals are accumulated over $K \triangleq 2^L$ successive slots, where L is an input parameter that describes the time-scale (or "level") at which cascade construction is introduced. The total "mass" (i.e., total number of arrivals over K slots) are then recursively subdivided into left and right components based on the RV C. For example, suppose that we start with a mass N over 2^L slots, and let c_1 an instance of C. Then, in the first step, $N \times c_1$ mass will be allocated to the left 2^{L-1} slots, and the remaining $N(1-c_1)$ mass to the right 2^{L-1} slots.[1] In the second step, the left and right portions are further subdivided. The subdivision continues until either a subinterval contains zero arrivals or a pre-determined level $L_0 < L$ is reached.

It turns out that the actual distribution of the cascade generator C is irrelevant, only its variance matters [18]. Thus, a uniform distribution of [0,1] forms a nice cascade generator (which has a coefficient of variation of 0.577). It is easy to go lower on the variance (or coefficient of variation) by defining following scaled cascade generator C_s as follows:

$$C_s = \eta C + (1-\eta)/2, \qquad 0 < \eta_i \leq 1 \tag{1}$$

Obviously, C_s also ranges over [0,1], has mean 1/2, and has variance $\eta^2 \mathrm{Var}(C)$. The η value must be chosen by the user to mimic desired multifractal impact on the traffic. Geist actually allows a separate η value at each step of cascade construction, but as shown in [18], this generality is not really needed.

It is also shown in [18] that the variance of the cascade generator has a huge impact the traffic characteristics and queuing properties — the queuing delays increase very rapidly with the variance of C. In this sense, the unscaled uniform distribution already provides a much higher coefficient of variation than is likely to be needed in practice. Nevertheless, for the sake of completeness, Geist also provides a few other cascade generator distributions that range over [0,1] and have a mean of 1/2.

As indicated above, Geist allows the cascade construction to occur from level L down to a specified level L_0; however, the results in [18] show that the largest impact on traffic occurs at level L itself; therefore, it suffices to choose L_0 as either $L-1$ (one level of cascade construction) or $L_0 = 0$ (cascade construction continues down to a single slot). Most of the results shown

[1] Since it is necessary to keep the mass integer-valued, we always round $N \times c_1$ to the nearest integer.

in [18] assume the latter, and could be used to choose appropriate values of the parameters L and η based on the observed scaling and queuing behavior of the traffic being modeled.

3.3 Traffic Nonstationarity

Much of the work on web-traffic analysis assumes that the traffic is approximately stationary, which has been confirmed by our own analysis of web traffic from several sources. However, our recent analysis of traffic logs from several business to business (B2B) and business to consumer (B2C) sites shows that even over short intervals of 10-15 minutes, the traffic is often nonstationary [19]. Geist provides the capability to introduce nonstationarity in the traffic should this be important for the study at hand.

In order to avoid any perturbation to the M/G/∞ paradigm, the nonstationarity is introduced at the output end by modulating the number of arrivals during the last slot given by the M/G/∞ process. This is done using a level-shift process that needs the following 3 input parameters:

1. Nonstationarity time-scale (NST), which is the average duration over which the traffic is expected to remain approximately stationary. It is assumed that NST is an integer multiple of the slot-size.

2. A "nonstationarity profile", which can be thought of as a non-negative random variable Z with mean 1.0. This RV provides the relative change in the traffic level over successive NSTs. In Geist, the RV Z can be specified using one of the available distributions (including empirical).

3. Level shift duration. The purpose of this parameter is to allow a smooth transition from one value of the random variable Z to the next.

Reference [19] shows how to estimate the first two parameters. The third parameter is provided to control how quickly the level shifts take place. The basic generation technique is to generate a new value for the random variable Z after every NST seconds, and multiply the total number of arrivals per slot by it. Since the cascade construction happens before the level shift, it is not affected by the level-shift process. This approach also does not affect the correlational properties of the traffic up to the time-scale of NST and introduces non-stationarity as intended.

3.4 Transactional Composition

Representation of transactional behavior is another important aspect of e-commerce traffic generation. A simple classification in this regard may be in terms of HTTP operations, e.g., static Get, dynamic Get, Post, etc. However, a more detailed classification is often necessary based on the server-side scripts invoked by a request in order to build the dynamic web page supplied to the user as response. The primary motivation for making such distinctions is that different scripts may have very different characteristics in terms of CPU and I/O requirements. In an e-commerce environment, server side scripts may involve interaction with the middleware (e.g., shopping cart, search server) or the backend database functions (e.g., product availability, product features, purchase, billing, etc.). Another important aspect of transactional composition is

the use of secure HTTP (or HTTPS) for some (possibly all) transactions. As shown in [17], an HTTPS is far more expensive than a corresponding HTTP transaction, thereby making the distinction necessary.

Reference [20] studies data from a number of B2C and B2B e-commerce sites in order to identify a somewhat generic transactional characterization of e-commerce traffic. This characterization involves identifying major transaction types, their security properties (i.e., secured or not), and dependencies between them. In order to account for dependencies properly, it is essential to first identify individual user sessions and then find the ordering constraints. We represent user transactions and their dependencies by means of a "state machine" or a Markov chain. Each state could then be further classified as "secure" or "nonsecure".

As shown in [20], B2C and B2B environments differ substantially, and require different generic models. B2C environments typically involve a lot of "window-shopping" but very little real purchase. Thus, all purchase related transactions can be lumped into a single category; in fact, it may be adequate to lump all transactions accessing the backend database (with the exception of product searches) into a single category. In contrast, B2B typically involves substantial purchasing activity, and a detailed representation of various steps (e.g., shopping cart, order placement, order inquiries, payment, etc.) may be needed. Also, in a B2B environment, all transactions are typically secure. Geist allows for all these possibilities allowing definition of a list of "states", each associated with a server side script. Ordering constraints can be specified by defining transition probabilities between states. Although the current implementation supports only zeroth and first-order Markov chain models, it is possible to extend it to higher order chains.

As stated above, the transactional classification applies to individual users, whereas we start by generating the aggregate traffic. Therefore, the traffic first needs to be split up into requests for an individual "virtual" user. Geist does this as follows: First it computes the number of virtual users K as the product of the aggregate mean arrival rate and the inter-request time for an individual user, both of which are input parameters to Geist. The successive arrivals are then marked equiprobably to belong to one of the K user streams. Each user stream is then assigned a unique-id and the transactions are marked according to the Markov chain model.

3.5 Request Process and Response Sizes

The actual server-side script executed by a transaction embodies the resource requirements in processing the transaction. Geist only indicates which scripts are executed and how often; it does not explicitly specify the properties of these scripts. Such an approach gives maximum flexibility to the experimenter in writing the scripts (or simply using the existing scripts for the e-commerce application of interest). It is easy to extend Geist to generate some simple scripts based on a set of given parameters, but this is currently not implemented.

Irrespective of the of the type of access (dynamic or static), the characteristics of the response sent out as a result are important from the perspective of both the client and the network. Geist provides control over response size by requiring that every Get request retrieve a "file" from a given "file-set". As in SPECweb99 benchmark, it is assumed that a dynamic GET simply creates some additional data (assumed to be rather small in size), which is appended or prepended to the static file being requested. The characteristics of the file-set are explicitly given, and are used by the file-set-create functionality of Geist. The access pattern to the file-set is explicitly specified

and kept orthogonal to the transactional characteristics. For example, a lot of computation in the server-side script or the fact that that transaction is a HTTPS transaction has no bearing on the size of the file retrieved.

The request size process is straightforward and is adequately described by the request size distribution. The random processes associated with requested files (or objects) need a more careful characterization. First, we need to specify the distribution of the size and type of files served by the Web server. Second, we need to specify the distribution of accesses to these files. For both of these, we take an approach similar to (but more general than) the one adopted by the SPECweb99 benchmark [3]. This approach establishes temporal locality of accesses indirectly, as compared to the more explicit scheme used by SURGE [5].

We divide the available files into K classes, numbered, $1, \ldots, K$, where is K is an input parameter. Each class has the same $M-1$ file indices numbered $1, \ldots, M-1$, for some input parameter M. The file sizes in class i are M-times that of file sizes in class $i-1$. In particular, file $j \in 1, \ldots, M-1$ of class i has size CjM^{i-1} where C is again an input parameter. The total number of available files are distributed among the K classes according to certain fractions $q_i, i \in 1..K$. The total number files thus assigned to a given class are further distributed among the $M-1$ possible indices according to certain given fractions. The total number of files can be distributed among a set of "directories"; this distribution is for convenient disk storage only — directories have no other significance.

The $M-1$ files in a class can be assigned a "popularity index" which describes the relative file access frequency. (This mapping is identical for all classes.) By convention, it is assumed that smaller popularity index means *more frequent* accesses. Next, actual access probabilities can be defined over the popularity index as a monotonic function. Often, a Zipf distribution is quite realistic for this mapping, i.e., the probability that the popularity index exceeds some value n is given by:

$$P(A > n) = Cn^{-\alpha}, \qquad 1 \leq n \leq N \tag{2}$$

where N is the number of files. Here $\alpha > 0$ is a parameter of the distribution and C is a constant that ensures that all probabilities sum to 1. Typical values of α observed in real systems are around 1. Note that if no popularity index is used, the access probabilities may be generated by using a non-monotonic function (e.g., the truncated Poisson distribution used in SPECweb96 [3]).

4 Traffic Generation Engine

Given that the traffic generation part merely reads necessary request parameters from a file, the important issues for it are timing accuracy, scalability, flexibility, and efficiency. Scalability is crucial in a lab testing environment where one is often interested in determining the ultimate capability of the system under test, and thus may subject it to far higher loads than would be reasonable in practice. For example, live servers are rarely engineered to run at more than 25-30% CPU utilization, whereas in a lab environment, it is often desirable to find out the achievable throughput when the CPU utilization is as close to 100% as possible. Also, laboratory systems are often "pure" in that they typically run only the main application of interest (e.g., HTTP

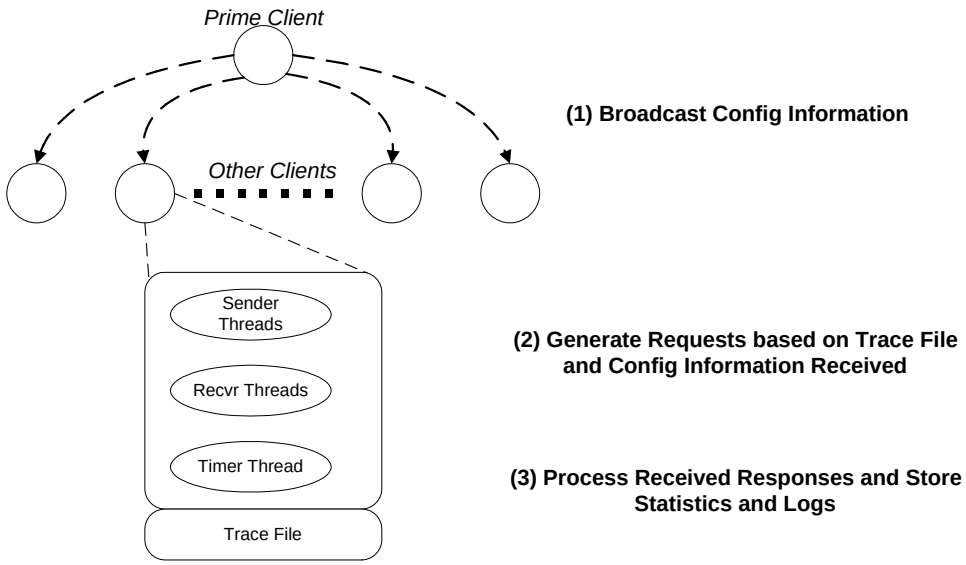

Figure 1: Overview of Geist Traffic Generation

request server) and thus can support much higher throughput. For example, current web-servers using 4-processor machines can easily support 10,000 hits per second. Thus, the traffic generator must be capable of scaling up to much larger throughput levels than those observed in practice.

Given the high burstiness of Internet traffic, the ability to overload a server and study its behavior is an important capability for a traffic generator. An important requirement for overloading the server is that the generator not be affected by the long response times of an overloaded server. Although the aggregate traffic generation approach can, in principle, handle this issue, the traditional socket interface and TCP window flow control make this a rather thorny issue. In this section, we discuss these issue and detail the architecture and current implementation of the request generation engine. Appendix D shows a sample configuration file for the traffic generator.

The overall architecture of traffic generation is based upon using an arbitrary number of client boxes, each of which runs an instance of the Geist traffic generation engine. Figure 1 depicts the architecture. As shown in the figure, each client consists of pool of sender threads, receiver threads and a timer thread. Additionally it requires a trace file that defines the request sequence. This trace file can be generated by splitting the original aggregate trace file in a variety of ways. One simple manner in which this can be achieved is a round robin splitting based on the number of clients being used in the test. It is expected however that in emulating e-commerce transactions a more sophisticated splitting mechanisms would be needed in order to maintain transactional coherence. After splitting the trace file one of the clients needs to be specified as a prime client.

The prime client is tasked with the responsibility sending global information to the other client boxes including such information as IP address of the HTTP server to be targeted for the test, the duration of the test and configurable parameters like the number of sender/receiver threads. Additionally the prime client is responsible for time synchronization amongst all clients.

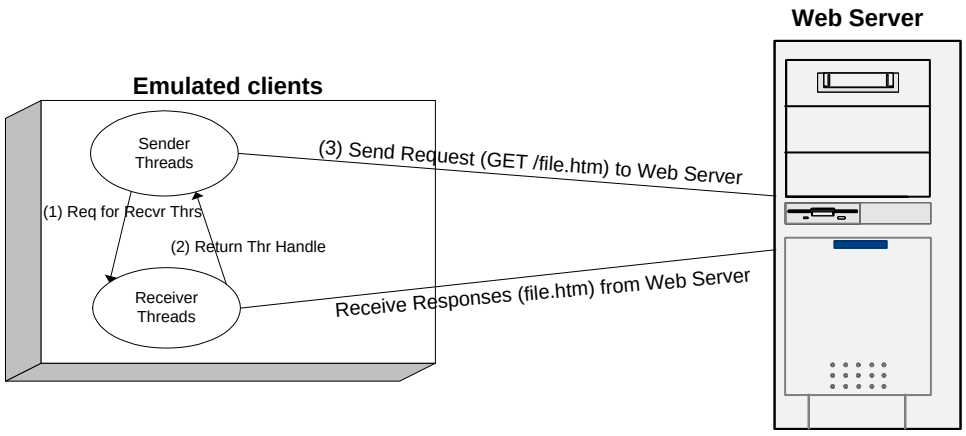

Figure 2: Steps in the Request Generation Process

This is achieved by broadcasting a time interval after which all the clients reset their virtual clocks to zero. All time decisions are based on this time zero from here on. This information is obtained from a configuration file and can be suitably modified by the user. The above is the only additional tasks performed by the prime client. After performing these functions the prime client behaves like any other client box.

The architecture of each client is based on a multithreaded model. Based on information received from the prime client a pool of sender and receiver threads is created. In addition a single "timer" thread is created. This thread is tasked with the responsibility of enforcing the period of test run. The sender threads read information from the trace file. As an example, if there are five sender threads, the first thread will make the first request from the trace file, the second thread the second request and so on. This action is performed in a modulo fashion so the first thread will also make the sixth request.

The iterative process of generating requests to the web server is shown in Figure 2. In an initial implementation of Geist, prior to sending a request, the sender threads obtained a handle on a receiver thread for each request that they were going to make. This receiver thread was obtained from the pool of receiver threads allocated at initialization. The delegation of responsibility of receiving a response to a thread other than the ones tasked with actually making requests, ensured that there is no feedback into the request generation process and requests are generated independent of the load on the server. However, this architecture required a rather large number of receiver threads. In order to make the traffic generator more scalable, we revised this implementation by allocating a receiver thread only when the response from the web server arrives at the client system. The detection of responses at the client system is accomplished using the *select()* call, which monitors a set of file descriptors (sockets in this case) for readiness in terms of reading data. This significantly reduced the number of receiver threads required and therefore made Geist more scalable and suitable for server overload studies.

Each request line in the trace file has information about the time at which this request needs to be made in order to maintain the arrival process correlation. The sender thread blocks on a timed synchronization primitive. At the appropriate time the sender thread makes a TCP

connection to the server and makes the HTTP request for the specified file as dictated by the trace file. At this point the sender thread releases the assigned receiver thread which then blocks on a "recv" call on the open TCP connection. On receiving the HTTP response the receiver thread time stamps the arrival, logs this information and then marks itself as free, thereby making itself available in the pool of free receiver threads.

This process continues till such time the "timer" thread indicates that the duration of the test is over. At that time, the sender and receiver threads are killed with the exception of the receiver threads that are still waiting for a response. On receiving a response, the waiting receiver threads exit and the test run terminates.

The current request generation engine has been implemented in the C language on the Win32 platform. All thread management routines, synchronization primitives and timer routines are based on the native Windows API. In this implementation, the communication between the prime client and the other clients is based on UDP broadcast. This restricts the implementation to having all clients in the same subnet. In the future, we plan to facilitate working across multiple subnets. This can be done by using hierarchy of prime clients, each in a different subnet. We also plan to integrate the pthreads library into the traffic generator, thereby making it compatible to Linux/Unix and other variants.

Since the traffic generator relies heavily on traces for input, it is important that it is not hindered by the I/O subsystem capabilities. This has been achieved in our current implementation by using memory mapped I/O when reading in trace files. Other important features of our traffic generator include support for accessing secure web pages via secure sockets layer (SSL) [9], and communication via HTTP proxies. Secure requests use the OpenSSL [26] library which allows us to study the impact of different suites of cryptographic algorithms on web server performance. For simplicity, a certain percentage of static traffic is designated as secure which translates into certain user requests being secure. Geist also supports secure and non-secure requests via proxies. This allows Geist users to gain insight into proxy server performance. Furthermore, in order to provide flexibility in the content of the HTTP request the user can add custom headers in the configuration file which will be used at the time of making the request. This makes the architecture extensible to support new HTTP request headers.

5 Accuracy and Performance of Geist

In order to validate the performance of the traffic generation engine, we developed a logging utility. This utility is essentially a mini HTTP server. Just as a typical web service, the logger runs on the system under test and receives requests at specified ports. However, for each request received, it maintains the arrival time-stamp and logs it into a file. This log file represents the aggregate traffic as seen by the web server. The difference between the logged request times and those specified in the trace file measures the accuracy of the traffic generator. Below, we present some results from experiments conducted to understand observed slippage in the aggregate arrival of requests.

The hardware characteristics of the clients would place a limit on the maximum request rate that it can support. To find this limit we generated deterministic traffic with varying inter-arrival times and ran experiments over 5 minute steady state periods. We determined

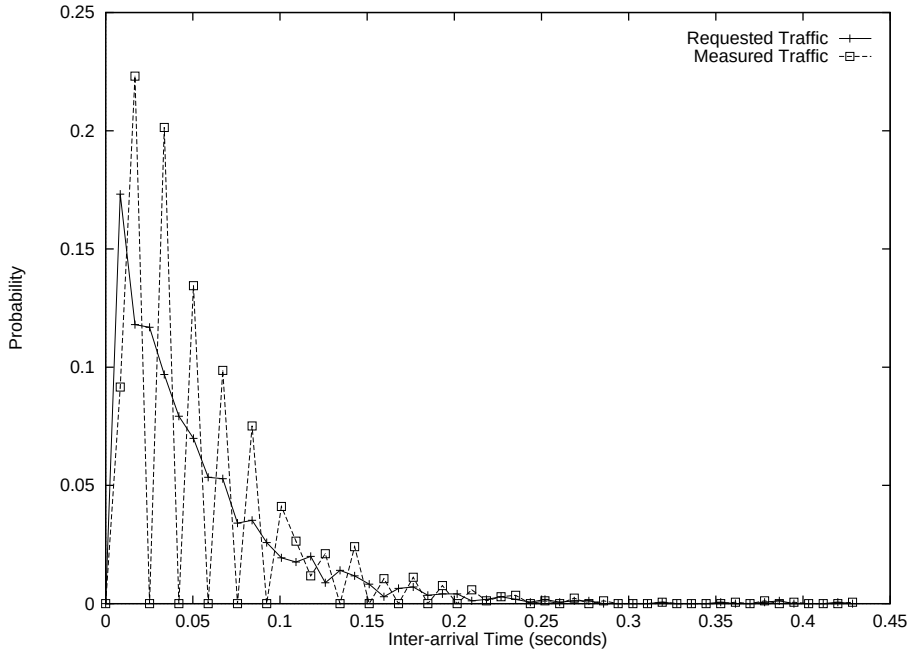

Figure 3: Inter-arrival distributions of requested and measured traffic

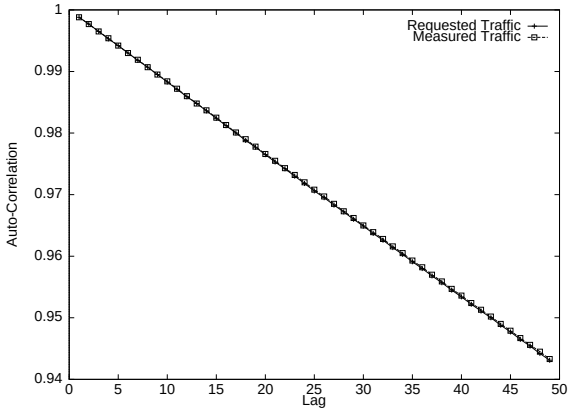

Figure 4: Autocorrelation plots of requested and measured traffic

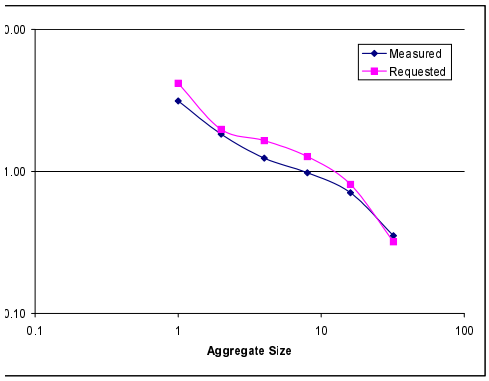

Figure 5: Variance-time plots of requested and measured traffic

how far the number of request actually made lagged behind the number of intended requests (i.e., requests appearing in the trace file). It was found that with the current implementation our clients could comfortably support about 25 requests/sec. Therefore, the slippage data is shown here for this case. In fact, the data shown here used only one client for traffic generation. Slippage studies with a large number of clients are yet to be performed.

For the study, we generated asymptotically self-similar traffic with $H = 0.8$. We compared the following parameters for the requested and measured traffic:

1. Interarrival time distribution, to examine the impact of clumping due to scheduling granularity.

2. Auto-correlation function for small lags, to examine short-term and medium-term correlations.

3. Variance-time plots, to examine long-term correlations and the H parameter.

Figure 3 shows the comparison between interarrival times. As expected, the scheduling granularity of 10 ms tends to clump interarrival times around multiples of 10 ms. However, remarkably, other than the clumping, the two distributions are very similar.

Figure 4 shows the autocorrelation function up to a lag of 50. It is seen that the autocorrelation functions are exactly identical for measured and requested traffic. In fact, we found that other values of maximum lags also gave identical plots. It can be concluded from this that short and medium term correlations are not perturbed by the scheduler.

Figure 5 shows the variance-time plot for the arrival process. Here we see some differences in the two curves. Because of the asymptotic self-similarity, the variance-time plot is not expected to be linear at small time-scales. At very large time-scales, the amount of data available is inadequate for accurate estimate, and one would generally expect a faster decay than what the desired H value would suggest. This is clearly seen in the last line-segment of the plots. Unfortunately, the amount of data is too small to give a substantial mid-range where one could fit a regression line and estimate the H value. Therefore, the H values were very crudely estimated, and turn out to be about 0.78 for the requested traffic and 0.77 for the measured traffic. Both of these are sufficiently close to the target value of 0.8 to suggest that the self-similarity is also pretty well preserved by the traffic generator.

We have used Geist already in generating traffic for a number of projects including the study of SSL performance, proxy server study, and a study of overload control strategies for web servers. Here we briefly describe the last application, which is more fully described in [14]. In this study, we studied the impact of loading a web server beyond its capacity to serve static and dynamic web pages. Using Geist, we showed that overloading the web server leads to significant drop in throughput and response time. Figures 6 & 7 depict the results obtained from that experiment. One of the advantages of using Geist (as opposed other traffic generators like Microsoft's WAS tool) here was that the traffic generated by the clients was not dependent on the implicit feedback of the varying load on the server.

Furthermore, we also used Geist to study the performance improvements using three proposed overload control mechanisms. Two of these proposed schemes involved feedback information from the web server to the emulated clients. Geist was modified to receive this feedback (via a UDP channel) and respond by throttling the inter-arrival rate of the outgoing traffic. While the typical uses of Geist would be to study the impact of different types of traffic on various architectures, this example depicts the flexibility of Geist's architecture for performing studies that require dynamic source traffic modifications based on feedback.

6 Conclusions and Future Work

In this paper, we have described Geist, a tool that we developed for generating realistic traffic for exercising web-servers and e-commerce front-end servers. The tool has the ability to mimic key

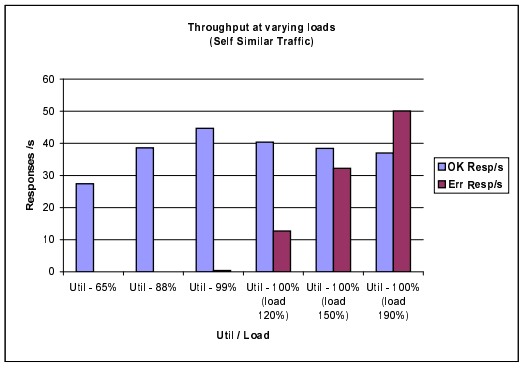

Figure 6: Throughput vs. offered load (Self Similar Traffic, no overload control)

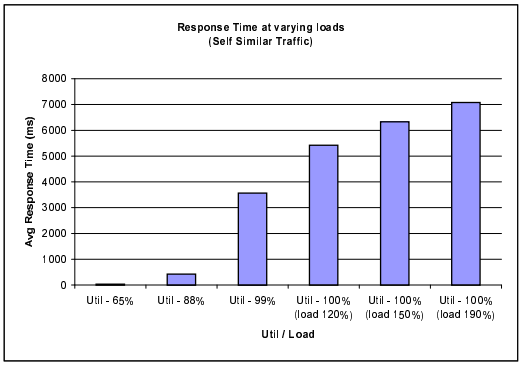

Figure 7: Response time vs. offered load (Self Similar Traffic, no overload control)

temporal properties of Internet server traffic: asymptotic self-similarity, multifractality at intermediate time-scales, and nonstationarity at time-scales of several minutes or more. Although each generated request is an HTTP operation (currently, only GET), it can further specify invocation of a server-side script to implement the appropriate transactional semantics. Ordering constraints between various transactions can be enforced via a state machine model. Size and locality characteristics of the downloaded objects are controlled by defining a "file-set" such than every request must retrieve a file from this file-set according to specified access characteristics. The downloaded file does not affect subsequent transactions issued by the user. This approach helps to dissociate the actions of the server-side scripts and the characteristics of the retrieved files and completely avoids the rather difficult problem of semantically relating requests and responses.

Unlike other web traffic generation tools that emulate individual user behavior, Geist generates aggregate traffic directly in order to easily control aggregate traffic parameters. User level attributes such as transaction types are then assigned by splitting the traffic into virtual users. Geist also supports secure HTTP traffic by incorporating calls to OpenSSL library [26] for client-side SSL handshake and encryption/decryption. Finally, Geist includes two distinct parts: trace generation and traffic generation. This separation offers advantages of efficiency and potential for using other traces in the traffic generator part.

Geist is to be regarded as work in progress, and can be enhanced in many ways — some of which are trivial in that they merely require more coding, while others may require changes to basic design and O/S interface. For example, the request generation engine currently supports only one HTTP request per TCP connection. Specifying additional parameters to support persistent connections is straightforward; however, pipelining of HTTP/1.1 requests would require a somewhat different type of characterization, where batches of embedded requests are explicitly identified. The current implementation does not mark requests as initial and embedded requests. If persistent connections do not stay idle for significant time-periods, the current approach of creating receiver threads when the connection is established become more scalable.

User abandonments and retries are very prevalent in web and e-commerce environment but their nature is not well understood. Two major effects of abandonments are (a) wasted work on the server, and (b) access size distribution being considerably skewed towards the low end

as compared to the accessed file size distribution. Abandonment may be partly inherent in user behavior (user decides that he no longer needs the information) and partly a result of response delays (user loses patience and times out). Geist currently does not explicitly support abandonments and retries. We plan to include this capability in order to allow exploration of optimizations such as abandonment-friendly scheduling disciplines and caching of initial portions of large files, etc.

Currently, Geist does not support POST operation because it is not clear how to properly handle form-based interaction in a traffic generator. Cookies are also not supported. In fact, before providing support for cookies, one needs to examine the fundamental question of how much semantic detail of user interaction is sensible to support from a server performance perspective. Our ongoing work shall address this and related concerns in web and e-commerce traffic generation.

References

[1] G. Banga, P. Druschel, and J.C. Mogul, "Better Operating System features for faster network servers," Proc. of workshop on internet server performance (WISP), June 1998.

[2] G. Banga and P. Druschel, "Measuring the capacity of a web server," Proc of USENIX symposium on Internet Technologies and Systems, Monterery, CA, Dec 1997.

[3] "An explanation of the SPECweb99 benchmark", available at `www.specbench.org/osg/web99`. (SPECweb96 information available at the URL `www.specbench.org/osg/web96`.)

[4] P. Abry, P. Flandrin, M.S. Taqqu, and D. Veitch, "Wavelets for the analysis, estimation, and synthesis of scaling data", report available from `www.serc.mit.edu.au/ darryl`.

[5] P. Barford, and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation", Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp. 151-160, July 1998.

[6] M. E. Crovella, A. Bestavros, " Self-similarity in world wide web traffic evidence and possible causes", Proceedings of the ACM SIGMETRICS 96, pages 160-169, Philadelphia, PA, May 1996.

[7] T. Daniels and C. Blondia, "Asymptotic Behavior of a Discrete Time Queue with Long Range Dependent M/G/∞ Input", Technical Report, Dept of Math and Computer Science, University of Antwerp, Belgium, April 1998.

[8] A. Erramilli, O. Narayan and W. Willinger, "Experimental queuing analysis with long range dependent packet traffic", IEEE/ACM trans on networking, April 1996.

[9] A.O. Freier, P. Karlton, P.C. Kocher, "The SSL Protocol, V3.0", IETF draft at `www.netscape.com/eng/ssl3/draft302.txt`.

[10] A. Feldmann, A.C. Gilbert and W. Willinger, "Data Networks as Cascades: Investigating the multifractal nature of Internet WAN traffic", Proc. 1998 ACM SIGCOMM, pp42-55.

[11] M. Grosslauser and J.C. Bolot, "On the relevance of long-range dependence in network traffic", Proc ACM SIGCOMM 96, pp15-24.

[12] D. Heyman and T.V. Lakshman, "What are the implications of long-range dependence for VBR video traffic engineering", IEEE/ACM trans on Networking, Vol 4, pp301-317, June 1996.

[13] R. Holley and E.C. Waymire, "Multifractal Dimensions and Scaling Exponents for Strongly Bounded Random Cascades", Annals of Applied Probability, Vol 2, pp 819-845, 1992.

[14] R. Iyer, V. Tewari and K. Kant, "Overload Control Mechanisms for Web Servers", to appear in Performance and QoS of Next Generation Networks, Naogya, Japan, Nov 2000.

[15] K. Kant, "Introduction to Computer System Performance Evaluation", McGraw Hill, 1992.

[16] K. Kant and Y. Won, "Server Capacity Planning for Web Traffic Workload", IEEE transactions on knowledge and data engineering, Oct 1999. pp731-747.

[17] K. Kant, R. Iyer and P. Mohapatra, "Architectural Impact of Secure Socket Layer on Internet Servers", ICCD, Sept 2000.

[18] K. Kant, "On Aggregate Traffic Generation with Multifractal Properties", proceedings of GLOBECOM'99, Rio de Janeiro, Brazil, pp 1179-1183.

[19] K. Kant and M. Venkatachelam, "Modeling traffic non-stationarity in e-commerce servers", working paper.

[20] K. Kant, M.Venkatachalam, "Characterization of front-end e-commerce servers", Working paper, available at `kkant.ccwebhost.com/download.html`.

[21] M. Krunz and A. Makowski, "A source model for VBR Video traffic based on M/G/∞ Input", Technical Report, Univ of Maryland.

[22] W.E. Leland, M.S. Taqqu, W. Willinger and D.V. Wilson, "On the selfsimilar nature of Ethernet traffic", IEEE/ACM trans on networking, Vol 2, No 1, pp 1–15, Fen 1994.

[23] Microsoft Web Application Stress Tool, `msdn.microsoft.com/library/periodic/period00/stresstool.htm`

[24] D. Mosberger and T. Jin, "HTTPPERF: Atool for measuring web server performance", Technical Report, HP Labs, 1998.

[25] O. Narayan, "Exact asymptotic queue length distribution for fractional brownian motion", Advances in Performance Analysis, Vol 1, pp39-63, 1998.

[26] The OpenSSL Project, `www.openssl.org`, last accessed Nov. 2000.

[27] K. Park, G. Kim, and M. Crovella, "On the effect of traffic self-similarity on network performance", Technical report CSD-TR 97-024, Dept of computer science, Purdue university.

[28] M. Parulekar and A. Makovski, "M/G/∞ input processes: A versatile class of models for network traffic", Proc of IEEE Infocom 97, April 1997.

[29] V. Paxson, "Fast, approximate synthesis of fractional gaussian noise for generating self-similar network traffic", Computer Communication Review, Vol 27, No 5, pp 5-18, Oct 1997.

[30] M. Roughan and D. Veitch, "A study of the daily variation in the self-similarity of real data traffic", SERC technical report, 1998.

[31] M.S. Taqqu, V. Teverovsky and W. Willinger, "Is network traffic self-similar or multifractal?", To appear in *Fractals*.

[32] W. Willinger, M.S. Taqqu, R. Sherman, and D.V. Wilson, "Self-Similarity through High Variability: Statistical Analysis of Ethernet LAN traffic at the source level", Proc of SIGCOMM 95, pp100-113.

[33] Y. Won, "A study of web traffic from Library of Congress web-servers", Unpublished report, Sep 1998.

Appendix A: Self-Similarity

Informally, self-similarity refers to the fact that the process aggregated over increasing time scales looks "similar", i.e., aggregation does not result in significant smoothing. Apart from the rather trivial case where the process has infinite variance and thus does not lend itself to smoothing, self-similarity is primarily caused by long-range dependence (LRD), i.e., significant correlations that persist over very large lags.

Let $\{X_t, i = 1, 2, \ldots\}$ denote a covariance stationary process with autocorrelation function $r(k) = E[(X_i - \mu)(X_{i+k} - \mu)]/\text{Var}(X_t)$. Let $X_i^{(m)}$ denote the aggregated process with block size m defined as the sample means of blocks of size $m > 1$. More precisely:

$$X_j^{(m)} = \frac{1}{m} \sum_{i=m(j-1)+1}^{mj} X_i \tag{3}$$

Let $r^{(m)}(k)$ denote its autocorrelation function. Then, X_i is long-range-dependent (LRD) if $\sum_{\forall k} r_k = \infty$, i.e., the autocorrelation function r_k decreases sufficiently slowly with lag k to have infinite sum. It is easy to show in this case that $\lim_{m\to\infty} m\text{Var}(X_i^{(m)}) = \infty$, i.e., the variance of sample means decays slower than $1/m$.

A random process is called *exactly self-similar* if $r^{(m)}(k) = r(k)$, for all m and k, i.e., aggregation has no effect at all on the correlation structure. It is called *asymptotically self-similar* if $\lim_{m\to\infty} r^{(m)}(k) = r(k)$, i.e., self-similarity holds for large m (i.e., at large time scales). *Fractional Gaussian Noise* (FGN) is a well-known example of an exactly self-similar process. It is defined as the zero-mean Gaussian process with autocorrelation function $r(k)$ given by:

$$r(k) = \frac{1}{2}[(k+1)^{1-\alpha} - 2k^{1-\alpha} + (k-1)^{1-\alpha}] \tag{4}$$

for lag $k = 1, 2, \ldots$, and $1 < \alpha < 2$. Reference [32] shows that FGN can be obtained as a superposition of a large number of on-off processes. In particular, consider a n iid (independent,

identically distributed) on-off processes, each with Pareto-like on/off time distributions with parameter α in the open interval (1,2). That is, if U is the on or off time, $P(U > u)$ $u^{-\alpha}$ with $1 < \alpha < 2$. Then, as $n \to \infty$, the superposed arrival process tends to FGN with $r(k)$ as given above.

Self-similarity is often characterized using the *Hurst parameter* H, which, is related to α parameter of Fractional Gaussian Noise as $H = (3 - \alpha)/2$. Note that for ordinary (i.e., short range dependent) random processes, $H = 0.5$, whereas for self-similar processes, $0.5 < H < 1$. For an exactly self-similar process, it can be shown using equation (4) that the variance of the aggregated process $X_j^{(m)}$ goes down with m as $m^{-\beta}$ where $\beta = \alpha + 1$. This implies that a plot of $\log(X_j^{(m)})$ against $\log(m)$ (commonly known as *variance-time plot*) is linear with slope $\beta = 2(1 - H)$. This yields one simple technique for checking if a given process is approximately self-similar and for estimating the corresponding H parameter value. Typically, self-similarity is observed in real data only for large "time-scales" (i.e., large values of the batch size m). Thus, real arrival processes are only asymptotically self-similar. However, the variance-time plot is still applicable except that the H parameter is determined by using the portion of the plot only for large time scales.

In the data analysis, it is often convenient to choose m as powers of 2. In this case, one could think of applying equation (3) recursively using batch size of 2 each time. With this, $k = \log_2(m)$ takes on integer values $1, 2, 3, \ldots$, and is regarded as the (dyadic) "time-scale". We follow this convention in this paper and in the design of Geist.

There are a number of other techniques to check for asymptotic self-similarity and to determine the H value [22]. One rather recent technique based on the Abry-Veitch (AV) estimator [4], which is used in [18] to study the properties of the trace generated by Geist. This estimator is based on the wavelet analysis of the given process and has been shown to be much more robust than the traditional methods. This estimator starts with a logscale plot of the energy in the wavelet coefficients (for the wavelet transform of the arrival time series) vs. the time-scale. Such a plot is known as the *logscale diagram* and shows the scaling properties of the arrival process over the time-scales of interest. If the plot shows a linear behavior over a set of time-scales consistent with long-range dependence, the corresponding Hurst parameter is estimated by a modified linear regression procedure detailed in [4].

Appendix B: Generating Self-Similar Traffic

There are a number of methods available in the literature to generate self-similar arrivals. Most of these attempt to generate a FGN-like process [32, 29]. For example, following the result in [32] mentioned above, a self-similar process can be generated by a superposition of a large number of on-off processes. This method has scalability problems as discussed above and we decided not to use it. Most other FGN-oriented methods have similar limitations. Therefore, we use a method based on M/G/∞ processes [28, 21]. Generation by this method is intuitive, efficient, admits easy control of parameters, and provides a simple way of introducing multifractal properties. Furthermore, it is known that a M/G/∞ process driving a deterministic queue yields heavier queue length tails than FGN for the same H parameter value [11, 7, 25]. Thus, from an engineering perspective, M/G/∞ traffic provides a more conservative (hence more desirable)

arrival process.

Most of the details concerning the use of M/G/∞ process for traffic generation as used by Geist are covered in [18]; therefore, we only provide a brief overview here.

An M/G/∞ process is defined as follows: Consider a discrete-time M/G/∞ queue with some time-slot Δ as the time unit. All Poisson arrivals during a time-slot are put into service at the beginning of the next time-slot. Let $P(S = k), k = 1, 2, \ldots$ denote the mass function of the service time S in units of time-slots. Let $\hat{S}$ denote the residual service time of a customer. Obviously, $P(\hat{S} = k) = \frac{P(S>k)}{E[S]}$. It is well-known that the queue-length distribution in this system at the end of each slot would be Poisson with mean $\lambda = \lambda_0 E[S]$ where λ_0 is average number of arrivals per slot to the M/G/∞ queue. However, the queue-lengths at the end of successive slots are correlated with autocorrelation function $\rho(k) = P(\hat{S} > k)$. Thus, if we use this queue-length process to generate arrivals for the system of interest, we have the following arrival process A: the marginal distribution of A is discretized Poisson with rate λ per slot and $P(\hat{S} > k)$ gives the autocorrelation function. The detailed equations may be found in [18, 21]. As shown in [21], M/G/∞ model can be used for generating traffic with long-range, medium-range and short-range dependence. For long-range dependence, we need the following form for $\rho(k)$:

$$\rho(k) = \alpha k^{-\beta}, \qquad 0 < \beta < 1 \tag{5}$$

where $\alpha = \rho(1) = 1 - 1/E[S]$. Then the generated arrival process is asymptotically self-similar with Hurst parameter $H = 1 - \beta/2$.

It is shown in [21] that a variable bit rate (VBR) video traffic is best modeled as *medium-range dependent* (MRD) by choosing $\rho(k)$ as:

$$\rho(k) = \alpha e^{-\beta\sqrt{k}}, \qquad 0 < \beta < 1 \tag{6}$$

where $\alpha = \rho(1)e^{\beta} = [1 - 1/E[S]]e^{\beta}$. Finally, for the short-range dependent case, we have:

$$\rho(k) = \alpha e^{-\beta k}, \qquad 0 < \beta < 1 \tag{7}$$

where α is again given by $e^{\beta} = [1 - 1/E[S]]e^{\beta}$.

The traffic generator supports all three types of dependencies discussed above. In all cases, the decay rate β is assumed to be given. For robustness, we determine $E[S]$ by requiring that $P(S = 1)$ (the probability of requiring 1 slot worth of service) is set at a given value η. Since $E[S] = 1/(1 - \rho(1))$,

$$P(S > k) = \frac{\rho(k) - \rho(k + 1)}{1 - \rho(1)} \tag{8}$$

From equation (8) and the fact that $P(S > 0) = 1$, we can solve for $\rho(2)$ as

$$\rho(2) = \rho(1) - [1 - \rho(1)](1 - \eta) = 1 - (2 - \eta)/E[S] \tag{9}$$

Therefore, by substituting for $\rho(2)$ from equations (5) – (7), we get:

$$E[S] = \frac{2 - \delta - \eta}{1 - \delta} \quad \text{where} \quad \delta = \begin{cases} 2^{-\beta} & LRD \\ e^{-\beta(\sqrt{2}-1)} & MRD \\ e^{-\beta} & SRD \end{cases} \tag{10}$$

Since the M/G/∞ system gives us only a discretized arrival process, the next step is to generate times of individual arrivals. Since the marginal distribution of the slot arrival process is Poisson, Geist attempts to retain that at smaller time scales by trying to make interarrival times within a slot exponentially distributed. To achieve this, if there are n arrivals within a slot, all those arrivals are assigned precise arrival times by using uniform distribution over the slot [15].

Appendix C: Sample input file for Trace Generator

This file illustrates some of the capabilities of Geist. It generates traffic that is asymptotically self-similar with $H = 0.8$, has multifractal properties with parameters $L = 5$, $L_0 = 1$, and $\eta = 1.0$, and remains stationary over 900 second intervals. The dynamic Get operations access one of the four server side scripts ("search", "order", "status", "payment") according to a first order Markov chain. The generated traffic contains one period of 200% overload as indicated.

The file-set and file access pattern is designed after the SPECweb99 benchmark, which provides identical directories of 36 files each. The 36 files are divided into 4 classes, each with 9 file sizes. Each file sizes is mapped to a file popularity index, and access distribution over the popularity index is Zipf. The access distribution over directories is also Zipf.

```
double temp;                          # A temporary variable declaration
num_classes = 4; sizes_per_cl = 9;    # 4 classes of files, each with 9 sizes
markov_chain_order = 1; num_asp_scripts = 4;
GO                                    # End of preamble: sets various array sizes
web_server_root   = "/web_traf_root/";
https_root_dir    = "secure_fs"; asp_root_dir = "scripts";
asp_file_names    = {"search", "order", "status", "payment"};
asp_access_probs  = { {0.35, 0.30, 0.20, 0.15}, {0.25, 0.20, 0.30, 0.25},
                      {0.45, 0.40, 0.10, 0.05}, {0.30, 0.20, 0.20, 0.30} };

slot_size_mult    = 10;               # Slot size in no of avg interarrival times.
correlation_type  = long_range;       # Nature of M/G/inf service time distribution
st_decay_rate     = 0.40;             # Decay rate of M/G/inf service time tail
one_slot_st_prob  = 0.032;            # Prob that the service needs only one slot
transient_period  = 300.0;            # Time in secs to bring generater to steady state
ss_run_time       = 900.0;            # Steady-state run time in seconds
reporting_time    = 900.0;            # Statistics reporting time in secs
avg_think_time    = 5.0;              # Avg think time of a user
avg_arrival_rate  = 30;               # arrivals (or ops) per sec.
iat_coeff_var     = 1.0;              # Coeff of variation of interarrival time.
static_get_frac   = 0.70;             # Fraction of static GET operations
static_ssl_frac   = 0.0;              # Frac of static traffic that is secure
cascade_gen_slots = 32;               # Consecutive slots for cascade construction
cascade_gen_limit = 1;                # Time in slots below which cascade is not used.
cascade_var_sf    = {1.0..1.0:16};    # Cascade variance multiplier of 1.0
cascade_gen_parms = {Uniform, 0, 1.0, 0, 0, NULL};  # Unif(0,1) cascade generator
ns_profile_parms
   = {Triangular, 0.75, 1.25, 1.0, 0.0, test_dist}; # Nonstationarity profile
```

```
stationary_period = 900;                              # Stationary period
level_shift_period= 20.0;                             # Takes 20s to shift levels
action_type       = gen_req;                          # Create a trace file of requests
ovl_start_time = 300; ovl_duration = 200;             # 200 sec overload at time 300 sec
ovl_magnitude = 2.0; ovl_cycles = 1;                  # One cycle of 200% overload
req_size_parms
      = {Log_normal, 50, 1460, 40, 20, NULL};         # Lognormal(offset, max, mean, std)
min_file_size     = 100;
num_directories   = 25 + \int(avg_arrival_rate/5);    # SPECweb99 rule for #directories
class_popul_probs  = {0.25, 0.25, 0.25, 0.25};        # Frac of files in each class
temp = 1.0/sizes_per_cl
file_popul_probs  = {temp..temp: sizes_per_cl};       # 11.1% of files belong to each size
class_access_probs = {0.35, 0.50, 0.14, 0.01};        # Access freq over classes
fpop_to_fno_func  = {5, 4, 6, 3, 7, 2, 8, 9, 1};      # File popularites
fp_ref_freq_parms =
       {Zipf, 1, num_directories, 1.0, 0.0, NULL};    # Dir access dist is Zipf
pi_ref_freq_parms
      = {Zipf, 1, sizes_per_cl, 1.0, 0.0, NULL};      # Access dist over popul. index
GO
```

Appendix D: Sample input file for Traffic Generator

This file illustrates the basic information required in a configuration file for the traffic generation engine. This file is used by the prime client to broadcast configuration information to all the clients to be used in the test.

```
web_server: 10.8.14.123   # IP address of the web server under test.
port_no: 80               # port number of the web server usually 80
ssl_port_no: 44           # ssl port number of the web server usually 443
proxy_server: 0.0.0.0     # IP address of the proxy server (0.0.0.0 implies no proxy)
proxy_port_no: 790        # port number of the proxy server.
num_sender_threads: 10    # the number of sender threads to be created per client.
num_recv_threads: 100     # the number of receiver threads to be created per client.
config_port: 5000         # UDP port on which global information will be broadcast.
warm_time: 1              # time for which information will not be logged
run_time: 30              # the run time for the test run.
time_to_start: 2          # time after which clients reset their virtual clock to 0.
requests_per_attempt: 1   # number of requests made per attempt by  sender thread.
cipher_suite: RC4-MD5     # cipher suite to be used.
```