GEIST: A Generator for E-Commerce & Internet Server Traffic
A User Manual

K. Kant, V. Tewari & R. Iyer
Intel Corporation

License/Copyright Information

OpenSSL LICENSE ISSUES

0.0.1 OpenSSL License

Original SSLeay License

Section(s) on GEIST Overview

1 Introduction

The work on Geist 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 aggregate 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. In particular, the detailed client behavior (e.g., time between mouse clicks, number of links per page, probability of following a link, etc.) are not directly relevant. 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 is worth noting in this regard that the traditional socket interface makes overloading web-servers quite difficult. Finally, the increasing dynamic content of web pages makes generation of realistic transactional characteristics particularly challenging, especially in e-commerce environments. Geist, has been designed to deal with these and other important issues.

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 [5]. Geist, unlike many other traffic generators can generate mixed secure/nonsecure traffic.

2 Components of GEIST

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 Geist suite plans to provide a set of perl scripts to convert HTTP logs into the same format as generated by the trace generator.

The format of the trace is as follows:

ReqId ReqTime ReqSize ReqType ReqCmd ReqPathFile where each field can be described as follows:

ReqId is the sequence ID for the transaction.

ReqTime is the time (in secs) at which the request should be made.

ReqSize is the size of the request packet.

ReqType used to denote directives for the request (i.e. secure, non-secure, no-cache option, etc)

ReqCmd is the HTTP transaction type (for e.g. GET).

ReqPathFile is the filename (inlcuding path in some cases) to be accessed at the server

More details on the generation of a trace with appropriate characteristics can be found in subsequent sections.

Section(s) on GEIST's Trace Generator

3 Trace Generator Statements

The trace generator takes an input file with extension ".inp" and generates one or more output files depending on the action requested. The input file may contain a series of variable declarations, assignments, print statements, "GO" command, or action type specifiers. These are explained in the following subsections. All statements are terminated by a ";". A statement may extend over multiple lines; however, any line breaks must occur after a delimitor (a comma, arithmetic operator, assignment operator, etc.). Multiple statements can appear in a line. Note that the program reads statements one line at a time; thus, a very long statement contained entirely on one line may exceed the line-buffer size ("line_length" parameter defined in misc.h with a default value of 480 chars) and result in a barrage of error messages. To avoid this, very long statements should be split on multiple lines. Comments can be put anywhere by using the character "#". Everything after "#" until the end of the line is ignored.

3.1 Variable Declarations

Geist provides a number of *predefined variables* to control various temporal and transactional properties of the traffic, properties of the web-pages (or files) accessed by the request, and format/type of output generated. These variables have known types and should not be declared. Geist also allows C-like declarations of user defined variables; these variables are generally needed to do some temporary computations. (Only scalars and 1-D arrays can be defined this way). Declarations can appear anywhere in the program, and will remain defined until redefined or until the end of program. (Redefinition is allowed, but results in a warning message.) The array dimension can be declared using arbitrary (integer) experssions involving constants, predefined variables or already declared and assigned user defined variables. Geist recognizes 4 basic types in the declarations:

integer Same as C int.

double Same as C double.

boolean An enumerated type with values "false" and "true".

string A char array with a maximum size as defined by the parameter "name_len" in the module misc.h (default 80 chars).

distribution A structure to receive distributional values (Explicitly declared distribution variables not fully implemented yet).

The variables declarations use a C-like syntax, as in:

integer count, length; double prob[num_classes * sizes_per_cl]; string cgi_scripts[4];

where "num_classes" and "sizes_per_cl" are predefined variables (explained later).

3.2 Assignment Statements

Assignments to various variables can be made in any arbitrary order (except for the assignment to the 4 structural parameters that must be done before the first GO statement as already explained above). The RHS of an assignment could be a complex expression including built-in functions described below. Limited forms of array and distributional assignments are also permitted. The array indices could themselves be integer valued expressions. Geist follows C-like type rules in expression evaluation and assignments. For example, a/b is considered to be an integer division if both a and b are integer and a real division if either of them is a double type. Similarly, assignment of an integer value to double goes without incident, but assignment of double to integer will result in a warning (and truncation).

The basic arithmetic operators are +, -, *, / (both integer and real) and have the usual precedence rules. Real numbers can be given in the usual F or E notation. A unary "+" is allowed only following the letter "e" or "E" in the E notation, but unary "-" works as expected. Currently, relational and boolean operators are not supported because their usefulness was unclear. However, they are trivial to add and may be supported in later versions. A number of other operators and functions are also supported as listed below:

% (modulus operator) $a\%b$ gives remainder of a divided by b. Here b must be an integer but a can be real.

∧ (exponentiation) $a \wedge b$ gives a raised to the power b. If both a and b are integers, the result is an integer, else it is double. As expected, "∧" has higher precedence than the multiplicative operators.

\abs (absolute value); the argument can be integer or double.

\int (floor of a double value),

\real (conversion from integer to double),

\log (natural logarithm),

\exp (exponential function),

\sqrt (square root),

\sin (trignometric sine),

\asin (trignometric inverse sine),

\tan	(trignometric tangent),
\atan	(trignometric inverse tangent),
\fact	(factorial of an integer valued expression),
\choose	(x \choose y where x and y are integers),

For scalars, the assignment statement syntax is same as in C. In addition, restricted array assignments are allowed and have some unique features. The following are some examples of valid assignment statements (assuming that appropriate variable declarations exist):

```
num_directories = \int(10 * \sqrt(throughput/5));
asp_file_names = "search", "order", "status", "payment";
test_array1 = a, 3.279e-4, a+b/2;
test_array2 = test_array1*a/b;
```

The first example shows assignment to a scalar integer. Irrespective of the type of variable "throughput", the sqrt function will yield a double value; thus the expression value must be converted to integer using the \int() function before assigment to the integer valued "num_directories". The second assignment shows initialization of an array of strings. The last two assignments show two ways of doing array assignments. The assignment to "test_array1" merely specifies the list of values (or expressions); in this case, the number of list elements must match the dimension of the array, else an error message will be generated. In the last assignement, every element of "test_array1" will be multiplied by a/b and the result assigned to the corresponding element of "test_array2". Again, the dimensions of the two arrays must match. The array to array assignment as supported by Geist is somewhat restricted as follows: (a) only one array can appear on the RHS, and (b) this array must be the very first operand in the expression.

In order to conveniently specify a list of values for array assignments, Geist allows a range operator ".." in expressions. For double values, the range expression is X .. Y : N where X and Y are arbitrary expressions and N is an integer value expression. Let x, y, and n denote, respectively, the values of these expressions. Then the range expression evaluates to n equi-spaced values starting at x and ending at y. That is, the values are $x, x+\alpha, \ldots, x+(n-1)\alpha$ where $\alpha = (y-x)/(n-1)$. Here n is required to be at least 2. Note that x and y could be identical, in which case the range contains repetition of the same value. For integers, only a simplified form of the range, expressed as X .. Y is allowed, and it refers to all integers in the given range. Integer ranges can also be used on the LHS to specify the subset of an array to be assigned. For example, the following specifies a discrete probability mass function over the integers 1..9:

```
num = 9; prob[1..num] = 0.12, 0.18 .. 0.04 : (num-1);
prob[num+1 .. size] = 0;
```

where the last statement introduces another feature of assignments, that is, if the RHS is a scalar and the LHS an array, the same scalar value is assigned to all the elements.

3.3 Print Command

This command simply prints out the value of the given variable(s). The syntax is "print" followed by list of comma-separated variables. All variables are printed in the format <name> = <value> separated by commas. All variables included in a print statement are printed on the same line, however, each print statement starts printing on a new line. Note that the arguments to print statement must be variables, not expressions. However, complete arrays or their subranges can be printed. For example, if X is an array "print X[3..7]" will print elements X[3]..X[7]. The indices in the range can, of course, be arbitrary integer valued expressions. Geist also provides a special form of print statement with the syntax "print *". This will print out current values of all the variables.

As indicated earlier, if one of the variables in the print list is a time-dependent variable, the print statement will also be executed at the end of every time-slot. This provides an easy way of generating a trace of time-dependent variables.

3.4 Program Execution

The trace input file does *not* provide any general program control statements (if-then-else, do-while, etc.) There is only one program control statements in Geist: a "GO" statement. The first appearance of GO statement allocates storage for all predefined arrays. This storage allocation is controlled by the following *structural parameters*:

num_classes Number of file classes in the model.

sizes_per_cl No of file sizes per class.

num_asp_scripts No. of ASP scripts invoked.

markov_chain_order Order of the Markov chain describing transactions.

arr_stat_buckets Number of buckets for reporting arrival process statistics.

Assignments of values to these parameters *must* be done before the first GO statement, else strange things might happen. Note that Geist does assign default values to these (and most other) parameters; however, if the default values don't agree with what you intended, the result could be segmentation fault (arrays too small), arbitrary values (arrays too large), probabilities not summing to 1.0, etc.

The subsequent occurrences of "GO" statement are a signal to execute the program with current input parameter values. After execution, the control returns to the next statement in the input file, so it is possible to keep modifying the input parameters and running the program.

The GO statement may have 0, 1 or 2 parameters. Accordingly, the syntax of the GO statement is as follows:

```
GO;          # No parameters
GO(<additional_run_time>);          # One parameter
GO(<additional_run_time>, <status>);          # Two parameters
```

The first parameter specifies the *additional* steady state run time for the program in seconds. If the GO statement is given without any parameters, the previous value of the additional run-time applies. Thus, for example, if we have GO(1000) followed by perhaps some parameter changes and simply a GO, the program will run again for 1000 seconds. Note in this regard that if the very first GO statement (the one that allocates arrays) contains an argument, the run-time will still be set to this value so that subsequent GO statements don't necessarily need the argument.

Between successive GO statements, the user would typically want to change certain parameters. Some of these changes could be trivial (e.g., changing some output switches in order to get more or less detailed output) and it makes sense to simply continue the traffic generation/simulation. In other cases, the change could be substantial (e.g., change to the traffic or file access characteristics), in which case, a reset and reinitialization is called for. The second parameter of GO statement indicates this. The second argument is an enumerated type which can take the following values:

end_simul End the program immediately. This is just for convenience during debugging — this would make the program ignore the rest of the input and exit. Obviously, the first argument is irrelevant in this case.

resume_simul Resume simulation w/o any state or statistics resetting.

stat_reset Reset the accumulated statistics and then resume. All state variables remain untouched. This is basically what happens during warmup time; however, on the second GO statement, Geist automatically performs the warmup so an explicit use of this option is not needed first time around.

queue_reset Reset statistics and "queue" state before resume. Here "queue" refers to the built-in capability of using the generated traffic for a queuing system simulation (explained later). If no such simulation is being performed, this option is equivalent to "stat_reset". Note that there may be a need to do a warmpup of the queue following a reset; this needs to be done explicitly.

full_reset This makes the entire program start over and reinitialize everything. This choice *must* be used if any of the structural parameters is changed. A warmup may be needed following the reset.

Some of the other program control variables are as follows:

transient_period Warmup period in seconds to bring the traffic generator to steady state. (This is not a warmup period for system queue or the file-cache.)

reporting_time Time-period in seconds for reporting statistics. This is useful to report statistics at intermediate points.

action_type Specifies the action to be performed.

window_size_mult Number of slots over which arrivals are accumulated for generating the arrival time-series.

Note that in order to ensure that the steady-state period starts at simulation time 0, the simulation time is set to negative <warmup_time> initially. (Thus if you enable statistics printing during warmup period, you would see negative times!)

3.5 Time Function and Load Control

In addition to the functions covered above, Geist provides one more function, called \time. \time(x) gives the time of an event as specified by the parameter x which can take the following values:

t_current Starting time of the current time-slot in the simulation.

t_ovl_start Time of the last start of the overload period.

t_ovl_end Time of the end of last overload period.

t_stat_reset Time of last statistics reset (obtained from the second argument of the last GO statement with values given as stat_reset, queue_reset or full_reset).

t_full_reset Time of last full reset (obtained from the second argument of the last GO statement that specifies a full_reset).

One frequent use of this function is to vary the system load as a function of time. Almost any variable concerned with run-time behavior can be made time-dependent by using the \time function directly or indirectly. For example, the average arrival rate can be made to vary in a sinusoidal manner between 5/sec and 15/sec by specifying:

avg_arrival_rate = 10.0 *(1.0 + 0.5*\sin(\time(current)))

Note that if a variable's value is time-dependent, its use in another statement makes that statement also time-dependent. Geist handles these dependencies correctly, however, it could lead to some surprising side effects. For example, the statement "print avg_arrival_rate" following the above assignment will print avg_arrival_rate every clock tick since this print statement becomes time-dependent itself.

Geist provides a more explicit way of controlling the load, and the \time function could be useful in that context as well. Often, it is desired to change the load during certain time window(s) but keep it normal otherwise. This is usually required for overload studies, where the load is increased beyond the capacity during a given window and then restored to the normal value to allow the system to recover. It may be desired to do this several times, and Geist provides support for it. The variables involved in this specification are as follows: (they all have "ovl" in their name, which refers to overload; however, the load may never reach into the overload region)

ovl_start_time Starting time of the overload episode.

ovl_on_time Duration of overload on-time (in secs).

ovl_off_time Duration of overload off-time (in secs).

ovl_cycles Number of overload on-off cycles (each of duration ovl_on_time + ovl_off_time) starting at time ovl_start_time.

ovl_magnitude Magnitude of the overload relative to the normal load. (As indicated above, this number could well be < 1.0, in which case we would really have an under-load, instead of overload).

Without the \time function, only constant overloads can be simulated using the above features. With the \time function in "ovl_magnitude", it is possible to generate sawtooth or other interesting waveforms. The main advantage of doing this for "ovl_magnitude" (instead of "avg_arrival_rate") is that the effect lasts only during the "overload period" (starting at ovl_start_time and going on for ovl_cycles).

The use of \time() function in these contexts introduces some complications which we discuss now. Without the \time() function, it suffices to evaluate the values of all parameters just once during the initialization. However, the presence of \time() function requires that each time-dependent expression be stored and re-evaluated at each "tick" during run-time. The efficiency obviously depends on how a "tick" is defined. It is convenient and quite efficient to consider the value of "time_slot" parameter (say T), as the duration between successive reevaluations of time-dependent expressions. However, the rather large granularity of "time-slot" parameter introduces some gotchas. Let K = ovl_start_time and suppose that the overload cycle duration is N slots.

Assume (as expected) that the time setting is done before the evaluation of any time-dependent expressions. Now consider the expression (\time(t_current) - \time(t_ovl_start)). Obviously, its value will go from 0 to (N-1)T during the overload cycle. In contrast, if the time were continuous, the expression value would actually go from 0 to N*T instead. Thus, if it is desired to generate a sawtooth waveform during the overload period that goes from a load of 1.0 to 2.0, one would have to write:

```
ovl_off_time = 0.0;
ovl_magnitude = 1 + (\time(t_current) - \time(t_ovl_start))/(ovl_on_time - time_slot);
```

The non-intuitive part here is the subtraction of the time_slot in the denominator and is a result of discretized evaluation. Note that similar issues can arise with other expressions such as \time(t_ovl_end) - \time(t_current). These anomolies must be kept in mind when using timing expressions.

4 Trace Generator Functionality

The Geist trace generator can do a number of things in addition to generating traffic. These activities and the related parameters are described in this subsection

4.1 Program actions

The basic functionality is specified by the parameter "action_type" given by the following enumerated type {create_fc, gen_req, sim_queue, sim_fc}. These functions are as follows:

create_fc Create the file-system on the server to be used by the generated requests. This capability would be typically used only once and on the server in order to create the necessary file system. Although any traffic related specification in the input file is irrelevant as far as file-system creation is concerned, it is highly recommended that the same input file be used for both file-system creation and traffic generation so that any inconsistencies are avoided. This capability is better accessed via the -c option on the command line (discussed later), although both methods are supported.

gen_req Generate requests. This generates the traffic trace and writes it out to the file <in_file>.trc where <in_file> is the root name of the input file. The syntax of each request line is explained in section 4.3.

sim_queue Simulate server. In this case, the generated traffic is directly fed to a rather trivial single-server queue model of the server, henceforth called *server queue*, to distinguish it from a queue needed to generate the arrival process itself. This is often useful for studying the queuing properties of the generated traffic in a very simple setting. The available distributions can be used for the *normalized* the service time. *The normalization is with respect to the average arrival rate, which makes the mean of the distribution same as queue utilization.* This is done for convenience: it is much more convenient to say that you want a target utilization of 70% instead of having to give the precise average service time. Explicit limit can be specified on the queue length at the server; any overflow arrivals are simply discarded. The simulation results include blocking probability (percentage of arrivals dropped), throughput, and mean/std-dev of queue length and response times.

Currently, the target utilization is applied without regard to the additional overhead of executing the dynamic GET scripts. (Needs fixing). However, the overhead of executing ASP scripts is specified by the parameter "asp_file_rel_pl" which gives the relative service time of dynamic scripts. Using this, the overhead of dynamic scripts is accounted for in computing the actual service time during the server simulation.

sim_fc Simulate file-cache. This is a somewhat more realistic simulation of the server in that (a) a LRU file cache is emulated for the requested files, and (b) the request service time is influenced by the disk and network I/O requirements as well. In this case, only the average value of the normalized target utilization is used since sample to sample variation is dependent on the size of the file associated with the request.

As with "sim_queue" option, the overhead associated with dynamic gets is added in exactly as in the above.

The last three choices above make a progression in the sense that "sim_queue" obviously subsumes traffic generation, and file-cache modeling subsumes traffic generation and queue simulation. The only exception is that in case of "sim_queue" and "sim_fc", the traffic trace file is not generated.

When <action_type>=sim_fc, the service time for static GETs is a function of three relative path-length parameters: (a) file-cache management path-length η_fc per file request, (b) disk I/O path-length η_d per IO request, and (c) network I/O path-length η_n per packet, where "path-length" is defined as the number of CPU instructions executed for these operations relative to those for the entire transaction. The given target utilization U (i.e., the average value of the specified normalized system service time distribution) is still used to compute the *base service time* S_0. In particular, $S_0 = U/\lambda$ where λ is the average arrival rate. This is used to estimate *processor speed* (or instructions/sec) as η_0/S_0. Thus, the actual service time S of a transaction that requests a file f of length L_f byte is given by:

$$S = \begin{cases} S_0 + \eta\_n * (L\_f/\overline{L} - 1) * S_0/\eta_0 & \text{file } f \text{ cached} \\ S_0 + (\eta\_n * (L\_f/\overline{L} - 1) + \eta\_d) * S_0/\eta_0 & \text{file } f \text{ not cached} \end{cases} \quad (1)$$

where $\overline{L}$ is the average file size. Note that if the file cache cannot hold the entire file-set, the actual server utilization will always be higher than the specified target value.

The variables associated with these options include

qlength_limit Maximum number of requests in the system queue. Any arrivals when the queue is full are simply dropped.

system_st_parms System queue normalized service time (or utilization) parameters.

max_cached_files Maximum number of files allowed to be cached.

cached_file_frac File cache size given as a fraction of the total file-set size. A file is not cached even if the cache has room if the number of cached files is already at <max_cached_files>.

asp_file_rel_pl Relative path-length associated with the execution of ASP scripts (an array of size <num_asp_scripts>). The path-lengths are relative to the path-length of the entire transaction for static file access.

disk_io_pl Relative path-length for a single disk I/O operation excluding the file-cache management. Its default value is 0.125, roughly based on SPECweb99 path-length on an Intel platform.

fc_mgmnt_pl Relative path-length for retrieval of a file from the file-cache management. Its default value is 0.0625, roughly based on SPECweb99 path-length on an Intel platform.

ntwk_io_pl Relative path-length for sending/receiving one packet over the network. Its default value is 0.0760, basically a guess.

4.2 Time series type

Closely associated with "action_type" is the integer variable "time_series_type" which controls what time series are written out. Currently, the following values are supported. (For clarity, we show the values as 2-digits in all cases, but since the value type is integer, leading zeros are not required).

The arrival time-series written out modified according to the parameter "peak_mean_ratio" which specifies the maximum value as a multiple of the average value (i.e., average number of arrivals within an arrival counting window). If the number of arrivals within a window exceed the maximum value so defined, the excess arrivals flow to successive windows. That is, if a large number of arrivals occurred in a certain window, this window and next several windows will contain the max arrival count until all the excess arrivals are exhausted. This "saturation" of arrival process can be effectively disabled by choosing a rather large value for "peak_mean_ratio" (e.g., 10 or more). The time_series_type can take the following values:

00: No arrival/queue-length time series generated.

01: Writes out the saturated arrival counting process in the "short" format to the file <in_file>.arr where <in_file> is the root name of the input file. This process is defined as the no of arrivals in successive time periods of duration window_size_mult * slot_size. In the "short" format, each value is separated by spaces with a -1 at the end of the file. This format is usually appropriate for scaling analysis of the traffic.

02: Writes out the saturated arrival counting process to the file <in_file>.arr in the "long" format. In this format, each line contains the pair (<seq_no> <arrival_count>). The file is ended with just a -1 on the last line. This format is suitable for plotting the arrival process.

03: Very similar to the action under the value "01", except that the written out time-series is reported as a fraction of the specified maximum value and is in the range 0.0 to 1.0. This value can be interpreted as utilization corresponding to the arrival counting process.

04: Very similar to the action under the value "02", except that the written out time-series is reported as a fraction of the maximum, i.e., saturated arrival count divided by the maximum value. This value can be interpreted as utilization corresponding to the arrival counting process.

10: Writes out the queue length process to the file <in_file>.qlts where <in_file> is the root name of the input file. In this case, the action_type must be "sim_queue" or "sim_fc", else nothing will happen. The output consists of one line upon each arrival to or departure from the system queue indicating the event time and the current queue length (with the arriver/departer taken into account). The syntax is:

<indicator> <event_time> <queue_length>

where the <indicator> is 0 for arrivals and 1 for departures. The file is ended with a -1 on the last line.

11: Writes out queue length process (if action_type > gen_req) as well as the arrival counting process in the "short" format.

12: Writes out queue length process (if action_type > gen_req) as well as the arrival counting process in the "long" format.

Because of the number and complexity of the parameters controlling the arrival process, it is often desirable to monitor the marginal distribution of the generated process and perhaps even try to adjust it by a trial and error procedure. (No such facility is required for the correlational properties since these parameters are given as input.) Geist reports the mean and standard deviation of the inter-arrival time in the output file. Note that for a pure $M/G/\infty$ process, the interarrival time distribution should ideally be exponential and hence the mean and std_dev should be identical. In reality, this could be perturbed by three factors: (a) the edge effects of a small slot-size (e.g., slot_size_mult< 10), (b) inadequate warmup time, and (c) inadequate run time. Note that for the long-range dependent traffic, an increase in the H parameter lengthens the acceptable warmup and run times dramatically.

When multifractal properties are introduced in the traffic (i.e., casc_gen_slots> 1), it perturbs the marginal distribution. In fact, with a large value of casc_gen_slots, the marginal distribution of the arrival process (not inter-arrival process) will tend towards log-normal. In Geist context, the desirable values of casc_gen_slots are generally quite small (64 or less) and it is very difficult to provide an a priori control over, say, the coefficient of variation of the inter-arrival time. Generally, the multi-fractal properties will tend to increase the coefficient of variation, but the desirable value will need to be set by trial and error.

To further assist examination of the arrival process, Geist provides bucketization of the arrival process. The array "bucket_threholds" provides the bucket boundaries, with the number of buckets (or the size of the array bucket_threholds) controlled by the parameter arr_stat_buckets. For example, "bucket_thresholds $= \{0.85, 0.65, 0.45, 0.0\}$" as used in section 6.6 specifies that the first bucket contains all samples that are $\geq 85\%$ of the maximum value, second bucket contains all samples that are between 65% and 85% of the maximum value, etc.[1] Note that the last value specified must necessarily be 0.0, unless you want to leave out some samples. The output file reports on the percentage of samples that fall in each of these buckets. It is possible to achieve something close to a target bucketization by modifying the following parameters

slot_size_mult	Controls the granularity of the reported samples. For example, if the utilization process is monitored and slot_size_mult=10, the only utilization factors in the generated time series will be 10%, 20%, 30%,...
iat_coeff_var	Controls the variance of the generated series.
st_decay_rate	Controls how much mass goes into the tail.
cascade_gen_slots	Controls how much the mass distribution over buckets is perturbed (higher values spread out the mass over high and low ends more).
cascade_gen_parms	The variance of this distribution also controls how much the samples are spread out over the buckets.

Of course, the bucketization will be affected by a host of other parameters as well. For example, the parameters controlling overload and stationarity properties will obviously affect the

[1]This bucketization always corresponds to the scaled, saturated process as with time_series_type values of 03 and 04.

bucketization.

4.3 Trace File Format

With action_type=gen_req, the resulting trace file has a *short_format* and an *long_format*. The syntax for these is as follows:

```
<short_format> ::= <seq_no> <arrival_time> <request_size> <attribute> <oprn>
[<script>] <file>
<long_format> ::= <client_id> <user_id> | <short_format>
```

The long format is used as an intermediate format for splitting the trace among a set of clients. The eventual output of the trace generator as used by the traffic generator is the short format. The various parameters in the short format are explained below:

1. Request sequence number (starting at 1)
2. Request time in seconds (steady state period starts at 0)
3. Request size in bytes
4. Request option indicated by an integer. Currently only 3 choices are supported, more could be added in the future.
   - 0: Non-secure (non-SSL) request and current copy has not expired.
   - 1: Non-secure (non-SSL) request and current copy has expired.
   - 2: Secure (SSL) request (one handshake per request scenario).
5. HTTP operation (currently only GET is supported).
6. URL. The trace generator creates only the following 4 forms of URLs:
   (a) Non-SSL request to a file, with syntax /dir_<dir_no>/file_<file_no>
   (b) SSL request to a file, which may have a different root directory than normal files (given by https_root_dir) and thus have some prefix path.
   (c) Path to an ASP script (given by asp_root_dir and scrip name) w/o any paramters.
   (d) Path to an ASP script with single parameter, which is a file.

Following are some examples of the request lines in short format:

```
46 35.3072 91 1 GET /dir_002/file_023
112 51.6019 47 0 GET /scripts/work.asp?/dir_023/file_047
```

The first example specifies that request no 46 is intended to be generated at absolute time 35.3072 seconds, has a size of 91 bytes and involves retrieving file no 23 from directory 2. The second request has a similar interpretation except that this is a dynamic get by running a script called work.asp, which generates some dynamic content that is appended "added" to file 47 in directlry 23 and returned. The intent here is that the script will generate some dynamic content and append or prepend to the file, however, these are not the only possibilities. In fact, since the semantics of the ASP file is not specified in Geist, the script is free to do whatever it wishes with the file argument, including just ignoring it. The only important point to note is that any perturbation to the sizes of the files will cause a mismatch between the stated and actual characteristics of the generated traffic.

File expiration is relevant when using Geist to generate traffic for testing a proxy server functionality. In this case, the proxy server will serve unexpired requests from a local copy, if one exists. In case of an expired request however, the proxy server will invalidate the local copy (if any) and proceed to get a fresh copy from the designated native server. As expected, the file obtained from the native server will be cached locally until it is either explicitly invalidated or it is displaced due to lack of local storage. The time-to-live period for each file is defined via the distribution <expiry_time_parms> which gives the expiration time in seconds. When the trace generator encounters a file-request for the first time, it uses this distribution to generate a TTL for the file. Every successive request for the file results in checking if the TTL has expired, and if so, the request option explained above is set to 2.

One difficult aspect of specifying dynamic GET is the required parameters for the ASP script. In general, scripts could have arbitrary parameter requirements and supporting them explicitly is simply impractical. Instead, Geist takes the following approach:

1. The trace generator produces at most one parameter, which is a file-name. The assumption is that the scripts used have capabilities to generate all required parameters. For example, when setting up the server for the experiments, one could write a front-end "search.asp" which internally generates the required parameters and then invokes the actual search script. This approach is adequate for testing purposes so long as the client doesn't need to know the parameter. Geist does not specify precisely what sort of distributions and other information will be used to generate the parameters.

2. When the input to the traffic generator comes from an actual HTTP log instead of the trace generator, the input will have the necessary parameters. The traffic generator passes such parameters to the server without examining them.

The trace file is generated in the short format if the number of clients "num_clients" is given as 1. If num_clients>1, the resulting trace must be further divided up among various clients. Geist handles this in two steps. In the initial trace generation phase, the output is generated in the long format, which includes all the fields in the short format preceded by two others:

client_id Indicates the client that should generate this request. (Client id's start at 0).

user_id Virtual user id that generates the request. (For information on virtual users, please see the discussion on transactional characterization).

The trace file in long format can be split into client-specific files, each of which is again in the short format. This splitting is done by invoking the trace generator with -s option. Note that the traffic generator part can only take the short format at input.

5 Available Probability Distributions

The trace generator input file needs to specify probability distributions for a variety of purposes including cascade construction, request sizes, life-time of documents, nonstationarity profile, reference frequencies, and system queue normalize service time). A distribution is specified using the syntax:

{<dist_name>, <offset>, <max_value>, <mean_value>, <aux_value>, <pointer>}

The various elements are explained below:

dist_name Name of the distribution, which can be Empirical, Exponential, Gamma, Log_normal, Pareto, Zipf, Poisson, Polynomial1, Polynomial2, Uniform, Triangle.

offset Convenient to interpret it as min value for Uniform, Empirical, and Zipf distributions.

max_value Maximum value. Except in cases where the random numbers of generated internally using an empirical distribution, rejection method is used to enforce the maximum value (i.e., if the generated value exceeds the maximum, a new random number is generated). This could make the generation very inefficient if the probability of exceeding the maximum value is large. A <max_value> of -1 can be specified if no limit is to be place on it.

mean_value Usually the mean (expected value) either including the offset (for Zipf) or excluding the offset (for all other distributions where mean is given). The distributions where <mean_value> DOES NOT represent the mean include:

Empirical: <mean_value> is the spacing between successive values of the discrete set.

Triangle: <mean_value> is the actual mode of the distribution.

Polynomial1 <mean_value> is the α value, i.e., the slope of the curve at minimum and maximum values.

Polynomial2 <mean_value> is $-\alpha$ value, i.e., negative of the slope of the curve at minimum and maximum values.

aux_value Interpretation depends very much on the distribution; this value may be unused, used to represent standard deviation, or some other parameter (see below).

pointer NULL for all distributions other than Empirical. For empirical, it gives the pointer to the list of values specifying the empirical distribution (more details follow). Internally, the program also uses this field for distributions that are easier to handle by explicitly listing the values (e.g., limited support Pareto/Zipf distributions).

In the following, we describe each of the available distributions by providing specific examples:

Empirical Geist does not support a completely general description of empirical distribution which would require a list of value and probability pairs. Instead, it only allows a mass distribution specified at equally spaced points. Of course, the minimum and maximum values and the step-size can be specified explicitly. Here is an example:

```
double prob_list[10];
prob_list = {0.04, 0.07, 0.11, 0.15, 0.19, 0.15, 0.12, 0.08, 0.06, 0.03};
ns_profile_parms = {Empirical, 4.5, 36.0, 3.5, 1.0, prob_list};
```

This defines a distribution over the range (4.5, 36.0) with a spacing of 3.5. The list of probabilities at the $N = 1 + (36 - 4.5)/3.5 = 10$ points is given by pointer to the user-defined array “prob_list”. Note that declared size of “prob_list” can be larger than 10, but the first 10 elements should still sum to 1.0. The fifth parameter is unused for this distribution.

Exponential This specifies a shifted exponential distribution restricted to some maximum value. Here is an example:

```
req_size_parms = Exponential, 40, 1460, 67.0, 0.0, NULL;
```

This specifies a exponential random variable with (original) mean of 67.0, shifted up by 40. That is, the effective mean of the distribution is 107.0. The maximum value is limited to 1460 (this is for the shifted random variable, i.e., after including the offset of 40). The fifth parameter is not needed here. The convention used in this and other distributions defined over the positive real line is that the distributional parameters (<mean_value>, <aux_value>) are specified without regard to <offset> and <max_value>. The <offset> value is used as a shift and <max_value> as a cut-off value. That is, the random number generation first generates a value according to <mean_value> and <aux_value> parameters, adds in <offset> and if the resulting value exceeds <max_value>, this instance is rejected and the next one generated. Note that the truncation means that the generated distribution will NOT have a mean of <offset> + <mean_value>.

Gamma This specifies the shifted gamma distribution restricted to a finite value. Here <mean_value> and <aux_value> specify the mean and std deviation of the original (unshifted) Gamma distribution. (For the purposes of RV generation, Geist computes the α and β parameters during the initialization phase).

Log-normal This generates the shifted Log-normal distribution restricted to a finite value. Here <mean_value> and <aux_value> specify the mean and std deviation of the original (unshifted) Gamma distribution.

Pareto The description for this is almost identical to that for Zipf except for the essential difference that Pareto is real-valued. In particular, exactly one of the <mean_value> and <aux_value> parameters must be set to 0.

Poisson This specifies the shifted, truncated Poisson distribution. The given <mean_value> does not include the shift and the <aux_value> parameter is not needed.

Polynomial1 This specifies a 3rd degree spline in the range <offset> and <max_value> with a maximum value at the mean (i.e., at $(¡offset¿+¡max\_value)/2$) and symmetric around it. In this case, <mean_value> is not the mean, but rather the slope of the curve at the end points. This distribution may be useful for the cascade generator.

Polynomial2 This is basically a mirror image of Polynomial1 wrt horizontal axis. That is, it has value minimum of 0 in the middle and maximum value at <offset> and <max_value> with <mean_value> giving the negative of the slope at the endpoints.

Triangle This specifies the triangle distribution in the range <offset> and <max_value> with mode (tip of the triangle) at <mean_value>. Here the <mean_value> obviously accounts for <offset>. Triangle-int provides the integer counterpart to Triangular distribution and must have all 3 parameters as integers.

Uniform This specifies the uniform distribution in the range <offset> and <max_value> with <mean_value> and <aux_value> being irrelevant. Uniform-int provides the integer counterpart to Uniform distribution and must have both parameters as integers.

Zipf Here the <aux_value> parameter represents the α value of the distribution and the given <mean_value> DOES include the offset. Note that α and the unconstrained mean_value are simply related by $\alpha = 1/(1 - min\_value/mean\_value)$, thus only one of these should be specified. The convention adopted is as follows: If it is desired to explicitly specify the unconstrained mean, this should be indicated by providing a value of 0 for <aux_value>. Similarly, if it is desired to explicitly specify α, the <mean_value> parameter should be given as 0. Geist will correctly fill in the missing value; that is, any subsequent references to the parameters of this distribution will yield correct values of both the parameters. Since Zipf is integer-valued, the given <offset> and <max_value> must be integers. (For efficient RN generation, all Zipf distributions are first converted to a discrete distribution by Geist).

In all cases, Geist reports the mean and standard deviation of the distribution as it processes it. Currently, with the exception of Zipf distribution, the <max_value> is ignored in these calculations.

6 Traffic Characteristics

This document concentrates mostly on how to use Geist. The underlying theory behind Geist is amply discussed in [6, 7] and is covered here only briefly. In particular, Geist does not attempt to emulate individual users and directly generates the aggregate traffic as seen by the server. This allows a direct control over the global properties of the traffic which makes it possible to generate traffic with given correlational properties (long, medium or short range dependent) and allows convenient introduction of multifractal and non-stationarity properties in the traffic.

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. 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 [10]. Such a transformation is adequate at least as far as second order properties are concerned.

6.1 Basic M/G/∞ Traffic Generation

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 [7, 10]. As shown in [10], 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 $\rho(k)$:

$$\rho(k) = \alpha k^{-\beta}, \qquad 0 < \beta < 1 \tag{2}$$

where $\alpha = \rho(1) = 1 - 1/E[S]$ (this is obtained by observing that $\rho(k) - \rho(k+1) = P[S > k]/E[S]$ and substituting $k = 0$.) Then the generated arrival process is asymptotically self-similar with Hurst parameter $H = 1 - \beta/2$ where β is the decay rate of the variance as a function of aggregation.

It is shown in [10] that a variable bit rate (VBR) video traffic is best modeled as *medium-range dependent* (MRD) by choosing $\rho(k)$ as:

$$\rho(k) = \alpha e^{-\beta\sqrt{k}}, \qquad 0 < \beta < 1 \tag{3}$$

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

$$\rho(k) = \alpha e^{-\beta k}, \qquad 0 < \beta < 1 \tag{4}$$

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

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 $E[S] = 1/(1 - \rho(1))$,

$$P(S > k) = \frac{\rho(k) - \rho(k+1)}{1 - \rho(1)} \tag{5}$$

From equation (5) and the fact that $P(S > 0) = 1$, we can solve for $\rho(2)$ as

$$\rho(2) = \rho(1) - [1 - \rho(1)](1 - \eta) = 1 - (2 - \eta)/E[S] \tag{6}$$

Therefore, by substituting for $\rho(2)$ from equations (2) – (4), we get:

$$E[S] = \frac{2 - \delta - \eta}{1 - \delta} \qquad \text{where} \qquad \delta = \begin{cases} 2^{-\beta} & LRD \\ e^{-\beta(\sqrt{2}-1)} & MRD \\ e^{-\beta} & SRD \end{cases} \tag{7}$$

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 [4]. If a marginal distribution other than exponential is desired, Geist can transform the inter-arrival time series to one that has the desired distribution. Currently, Geist only allows a specification of the coefficient of variation of inter-arrival times. It uses this to construct a suitable branching Erlang type of marginal distribution.

For long-range dependent traffic, the parameter β is best calculated from the desired H parameter value using $\beta = 2(1 - H)$, and thus must lie in the range [0,1]. For medium and short range dependence, the parameter could be an arbitrary positive number — however, since $e^{-\beta}$ decreases very rapidly with β, moderate and large values are unlikely to result in much different behavior. In particular, a β around 5-6 with SRD will yield effectively a Poisson process. Note that in the M/G/∞ pardigm, the relationship $\rho(1) = 1 - 1/E[S]$ precludes that the correlations will be negligible between closely spaced samples. This is not an issue in practice since any random number generation scheme will necessarily yield high correlations between close-by points.

The variables relevant for controlling basic traffic properties are described below.

slot_size_mult Slot_size multiplier. This gives the slot_size in the units of inter-arrival time. The slot-size is needed to generate the M/G/∞ traffic. Generally, a slot-size of 10-20 is quite adequate and there is no need to fine-tune this parameter.

correlation_type This is a enumerated type with valid values of "short_range", "medium_range" and "long_range" as described above.

st_decay_rate The service time decay rate factor denoted by β above. See the discussion above about on how to choose this.

one_slot_st_prob Probability that the service at the M/G/∞ queue needs only one slot (denoted as η above). Generally, there is little reason to change this value.

avg_arrival_rate Average arrival rate (per second).

iat_coeff_var Multiplier to the inter-arrival time coefficient of variation in the default case (i.e., with iat_coeff_var= 1). Currently, iat_coeff_var< 1 is not implemented.

If no multifractal properties are specified for the traffic (i.e., casc_gen_slots=> 1), the given iat_coeff_var should actually be achieved by the generated traffic (although only approximately because of the slot-boundary effects).

6.2 Introducing Multifractal Properties

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 is necessary for network-centric studies where the network components (routers, links, proxy servers, firewalls, etc.) are represented explicitly; for most server-centric studies, testing over the WAN or duplicating the network in the laboratory is usually impractical. Geist addresses this issue by providing an indirect way of accounting for the network impact on the traffic. This is based on the recent work that has shown that network impacts 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. 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.

The main characteristic of a multifractal arrival process is that it can be described via a mass redistribution according to a special random process called a *cascade generator*. In particular, Geist uses a so called *semi-random cascade generator* C which is defined as a random variable over the interval [0,1] with a mean of 1/2 [3]. Basically, the idea is to take a "mass" (or number of arrivals) over a large interval and subdivide them recursively into left and right portions by using instances of the RV C. If this process were to be repeated indefinitely, it leads to the true mathematical object called the multifractal; however, the practical use of subdivision must necessarily be very limited.

To apply subdivision to the generated M/G/∞ process, the generated arrivals are first 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. Suppose that we start with a total of N arrivals over 2^L slots. Let c_1 denote the first instance of the random variable 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. (Since it is necessary to keep the mass integer-valued, $N \times c_1$ is always rounded to the nearest integer.) 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 [7]. Furthermore, the variance has a huge impact the traffic characteristics and queuing properties — the queuing delays increase very rapidly with the variance of C. Thus, a uniform distribution of [0,1] (with a coefficient of variation of 0.577) already provides much higher variance than is likely to be needed in practice. Of course, a uniform distribution with a narrower range (e.g., Unif(α, $1-\alpha$) with $0 < \alpha < 0.5$) can provide the required scaling of the variance [by a factor of $(1-2\alpha)^2$]. Of course, one could also use other distributions such as Triangular, Polynomial1, Polynomial2, and Poisson.

Geist allows the cascade construction to occur from level L down to certain number of levels; however, it is found that [7] that the largest impact on traffic occurs at level L itself. The variables relevant for controlling basic traffic properties are described below.

cascade_gen_slots Number of slots over which arrivals are accumulated for cascade construction. Must be chosen as a power of 2, i.e., 2^L where L is as defined above. If no cascade generation is desired, this parameter should be set to 1.

cascade_gen_parms Cascade generator distribution. Must have the range [0,1] (or smaller) and must be symmetric about the mean of 0.5. A range narrower than [0,1] is one way to reduce the variance of the distribution. The example in section 6.6 illustrates this for the uniform distribution.

max_casc_levels Maximum number of levels for which the subdivision process described above will continue. The subdivision process also terminates when an interval contains only one arrival or the interval size is down to a single slot. Thus, if the subdivision is required down to the level of single slot, it is okay to set this parameter to some large value [i.e., larger than $\log_2$(cascade_gen_slots)].

6.3 Traffic Nonstationarity

Most of the web-traffic studies assume a stationary traffic; however, in many environments, the traffic cannot be considered to be reasonably stationary over periods larger than 10-15 minutes. Geist provides the capability to introduce nonstationarity in the traffic, should this be important for the study at hand. This is done by modulating the number of arrivals during each slot (as generated by the $M/G/\infty$ process) by a *nonstationarity profile* which essentially specifies the *level shift* distribution. The variables involved are as follows:

ns_profile_parms Specifies the distribution of the level shift process which must be a non-negative random variable Z with mean 1.0. This could, of course, be any of the distributions supported by Geist.

stationary_period This is the duration of time for which the level shift process remains at a given level. That is, a new value of the <ns_profile_parms> RV is generated once for each successive stationary_period. The total number of arrivals in a slot is then this value multiplied by the number of arrivals determined from the $M/G/\infty$ model. The <stationary_period> must be an integer multiple of the slot-size.

level_shift_period The purpose of this parameter is to allow a smooth transition from one value of the random variable Z to the next. That is, the shift between the levels is not abrupt, but instead occurs linearly over the period given by <level_shift_period>.

The nonstationarity profile distribution be determined from a real traffic trace (e.g., HTTP logs of a web-site) and may well be an empirical distribution. An essential part of this estimation is determination of the stationarity period as well. A constant stationarity period is assumed here for simplcity in parameter estimation.

6.4 Transactional Composition

For e-commerce front-end servers, various types of user transactions (e.g., product search, product order, order status, payment, etc.) will typically access appropriate scripts for performing these functions. In Geist, we assume that each dynamic request invokes exactly one ASP script and this request, together with all static file accesses following it, forms a user transaction. Typically, user transactions have certain ordering properties, which are conveniently described via a Markov chain. For example, a product order is likely to be followed by the payment transaction. Geist supports both zeroth order and first order Markov chain models. The Markov chain order is defined by the variable markov_chain_order which can currently take values 0 and 1 only.

It is important to note that the Markovian transactional model describes the behavior of a given user, rather than that of the aggregate traffic. As such, Geist has no concept of individual users since it generates aggregate traffic arriving at the server directly. In order to handle transactional ordering and possibly other user-level characteristics (e.g., cookies, abandonments, retries, etc., which are currently not implemented), Geist uses the concept of a *virtual user*. The number of virtual users is determined as the product of "avg_think_time" and "avg_arrival_rate". Then, each request is equiprobably assigned to one of the users subject to the constraint that the time between successive requests by the same user never goes below 1/10th of avg_think_time. In the long trace format, user_id's are also included in the trace file. Although the traffic generator part currently has no use for user-id's (and, in fact, doesn't recognize the long trace format), user-id's could be useful for more advanced features.

The parameters relevant for transational properties are as follows:

num_asp_scripts Number of ASP scripts used by dynamic GETs.

asp_file_names Names of ASP scripts (an array of size <num_asp_scripts> and of type string).

markov_chain_order Order of the Markov chain defined over ASP scripts. Current valid values are 0 or 1 only.

asp_access_probs ASP access probabilities. If <markov_chain_order>=0, this is a 1-D array giving the probabilities of invoking individual ASPs (irrespective of the last ASP invoked). If <markov_chain_order>=1, it is a transition probability matrix where the (i, j)th entry gives the probability of invoking the jth ASP following the invocation of the ith ASP.

avg_think_time Gives the average user think time between successive requests and used to compute number of virtual users. Note that each virtual user operates according to its own Markov chain.

num_clients Number of clients over which the trace is to be distributed (i.e., number of clients to be used for actual traffic generation).

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.

6.5 Request Process and Response Sizes

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 file-set specification is as follows. Files belong to one of K classes, numbered, $1, \ldots, K$ and 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:

$$P(A > n) = Cn^{-\alpha}, \qquad 1 \leq n \leq N \tag{8}$$

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 [1]).

The variables relevant for file sizes and accesses are as follows:

num_classes Number of file classes (denoted as K in the above). For example, <num_classes>=4 in SPECweb99.

min_file_size Size of the smallest file in the file-set. This is the size of the first numbered file of class 1 (denoted as C in the above). For example, $C = 100$ in SPECwebb99.

sizes_per_cl Number of file sizes per file class (same as $M-1$ in the above). For example, <sizes_per_cl>=9 for SPECweb99. In this case there are 9 files in each class with *relative* sizes of $1, 2, \ldots, 9$ and successive classes have size jumps by a factor of <sizes_per_cl>+1 = 10. So, with the default <min_file_size>, class1 file sizes are $100, 200, \ldots, 900$ bytes, class2 sizes are $1000, 2000, \ldots, 9000$ bytes, and so on.

req_size_parms A distribution giving the request size characteristics.

class_popul_probs Fraction of files assigned to each class (an array of size <num_classes>). Normally, (and in the SPECweb99) each class would get equal number of files.

num_directories Number of directories. Each directory contains <sizes_per_cl> $\times$ <num_classes> files. The total number of files are divided among various classes according to <class_popul_probs>. In SPECweb99 <num_directories> is given by (25 + <avg_arrival_rate>/5).

file_popul_probs Fraction of files within a class assigned to various sizes (an array of size <sizes_per_cl>). Normally (and in SPECweb99) each size category would get equal number of files.

fp_ref_freq_parms Access frequency distribution over the directories (numbered 1 to <num_directories>). In SPECweb99, this distribution is Zipf with $\alpha = 1.0$.

class_access_probs Relative probability of accessing files in a class (an array of size <num_classes>). In SPECweb99, the relative probabilities are (0.35, 0.50, 0.14, 0.01).

fpop_to_fno_func A permutation mapping of file popularity index to file size index. For SPECweb99, the mapping is $(5, 4, 6, 3, 7, 2, 8, 9, 1)$. That is, the popularity index of 1 maps to file no 9 in the class, index 2 maps to file no 2, etc.

pi_ref_freq_parms Popularity index reference frequency distribution. This is a discrete distribution over the popularity indices. In SPECweb99, the distribution is Zip over the range (1..9) with $\alpha = 1.0$.

Geist also allows specification of location of various directories. The corresponding variables are:

web_server_root Location of the web-server files (directory where the directories for non-secure files are located).

https_root_dir Root directory (located in <web_server_root>) for secure HTTP files.

asp_root_dir Root directory for ASP scripts (located in <web_server_root>)

6.6 An annotated Example of Input File

```
#debug[echo_tokens] = 1;
debug[echo_input] = 1;
#debug[init_stats] = 1;
#debug[big_cum_dists] = 1;
#debug[transient_stats] = 1;
#debug[rand_no_gen] = 1;
#debug[file_no_gen] = 1;
#debug[mass_redist] = 1;
#debug[binary_search] = 1;
#debug[mgi_actions] = 1;
#debug[queue_actions] = 1;
#debug[fc_actions] = 1;
#debug[arrival_times] = 1;
#debug[loading_parms] = 1;
#debug[time_dep_parms] = 1;

double temp, test_dist[27], std_sf;
```

```
#****************ALWAYS DO THIS FIRST AND FOLLOW UP WITH A GO STMNT*************
#
max_casc_levels = 16; # Not necessary to assign values to these
#markov_chain_order = 2;
markov_chain_order = 1;
num_classes = 4; sizes_per_cl = 9; num_asp_scripts = 1; arr_stat_buckets = 4;
GO;

peak_mean_ratio   = 2.95;       # Max arrrival sample size compared with avg
one_slot_st_prob  = 0.032;      # Prob that the service needs only one slot
window_size_mult  = 1;           # #of slots for accumulating arrivals

##>>__________ The following deals with ASPs and traffic fractions ____________

web_server_root   = "root/";
https_root_dir    = "secure_fs";
asp_root_dir      = "";
#asp_file_names    = {"search", "order", "status", "payment"};
#asp_file_rel_pl    = {0.10, 0.02, 0.02, 0.05};
#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} };
asp_file_names    = {"LotsOfWork"};
asp_file_rel_pl   = {0.20};
asp_access_probs  = {1.0};
static_get_frac   = 1.0;         # Frac of static GETs (secure or not)
ssl_traf_frac     = 0.0;         # Frac of secure traffic (static or dynamic)
expiry_time_parms = {Uniform, 8.0, 16.0, 1.0, 0, NULL};
     # Expiry time of contents in proxy cache (in case going through proxy)

#----------------------------------------------------------------------------
##>>___________ The following deals with SRD vs. LRD traffic _____________

##>>______________ Uncomment the following for SRD traffic
# correlation_type = short_range; # Nature of M/G/inf service time distribution
# st_decay_rate    = 2.4;        # Decay rate of M/G/inf service time tail

##>>______________ Uncomments the following for LRD traffic
 correlation_type  = long_range; # Nature of M/G/inf service time distribution
 st_decay_rate     = 0.20;     # Decay rate of M/G/inf stime tail = 2*(1-H)

##>>___ Set following to >1 to introduce multifractal properties along with LRD
cascade_gen_slots = 64;          # Consecutive slots for cascade
std_sf = 0.65;
cascade_gen_parms = {Uniform, 0.5 - std_sf/2, 0.5 + std_sf/2, 0, 0, NULL};

#----------------------------------------------------------------------------
##>>_____________ The following is for arrival_rate & related parms ________

slot_size_mult    = 25;         # Slot size in no of avg interarrival times.
bucket_thresholds = {0.85, 0.65, 0.45, 0.0};  # buckets to monitor arrival process
avg_arrival_rate  = 500.0;      # Average arrival rate/sec
iat_coeff_var     = 1.0;        # Interarrival time coeff of variation
avg_think_time = 5.0;           # This is needed to generate virtual user IDs
num_clients       = 1; # No of clients used for workload generation

##>>____ As an alternative, we can try the following, which gives a full sine
#    wave with a period of ovl_on_time+ovl_off_time and range of 0.5 to 2.5.
#
#  ovl_on_time = 30.0; ovl_off_time = 30.0;  ovl_magnitude = 1.0;
#  temp = \time(t_current)/(ovl_on_time + ovl_off_time);
#  avg_arrival_rate  = 10.0 *(1.5 + \sin(temp));

##>>____________ The following deals with load variations _____________

ovl_start_time = 0.0; ovl_cycles = 0;
double slot_size;
slot_size = slot_size_mult/avg_arrival_rate;

##>>_______________ Uncomment exactly one of the following:

ovl_on_time = 100000; ovl_off_time = 100000; ovl_magnitude = 1;

##>>_____ This generates an increasing sawtooth between 1 & 2 (no off period)
# ovl_on_time = 30.0; ovl_off_time = 0.0;
# temp = (\time(t_current) - \time(t_ovl_start))/(ovl_on_time - slot_size);
# ovl_magnitude = 1 + temp;
```

```
##>>____ This generates a half sine wave (with max of 3.0) & equal off period
# ovl_on_time = 30.0; ovl_off_time = 30.0;
# temp = (slot_size*5 + \time(t_current) - \time(t_ovl_start))/ovl_on_time;
# ovl_magnitude = 1 + 2*\sin(3.1418 * temp);

##>>________ This generates a square wave between levels 1.0 & 2.0
# ovl_on_time = 60.0; ovl_off_time = 60.0;
# ovl_magnitude = 2.0;

#------------------------------------------------------------------------------

###############################################################################
req_size_parms    = {Log_normal, 50, 1460, 40, 20, NULL}; #offset, max, std, mean
test_dist = {6.289e-03, 1.258e-02, 6.289e-03, 1.887e-02, 1.887e-02, 6.289e-02,
1.887e-02, 8.176e-02, 6.918e-02, 1.132e-01, 6.289e-02, 1.069e-01,
5.031e-02, 5.031e-02, 4.403e-02, 4.403e-02, 7.547e-02, 6.289e-03,
1.887e-02, 2.516e-02, 4.403e-02, 2.516e-02, 6.289e-03, 6.289e-03,
1.258e-02, 6.289e-03, 6.289e-03};
#
ns_profile_parms = {Empirical, 4.7, 30.7, 1.0, 1.0, test_dist};
#
# Given the original histogram for nonstationarity, we want to recast it so
# that the difference successive x-values is 1.0.  For this, we need to divide
# them by the factor (x_max - x_min)/(n_values - 1).  This gives the new range
# 4.7 to 30.7.
stationary_period = 9000000;         # Stationary period
level_shift_period= 20.0;      # Time to shift from one stationary level to next
min_file_size     = 100;
num_directories   = 25 + \int(avg_arrival_rate/5);  # sweb99 rule for #directories
temp = 1.0/sizes_per_cl;
file_popul_probs  = {temp..temp:sizes_per_cl};
class_popul_probs  = {0.25, 0.25, 0.25, 0.25};   # popul dist over classes

fp_ref_freq_parms = {Zipf, 1, num_directories, 0.0, 1.0, NULL};
# Reference pattern over directories is Zipf with alpha=1
class_access_probs = {0.35, 0.50, 0.14, 0.01};  # access freq over classes
#                    1, 2, 3, 4, 5, 6, 7, 8, 9
```

```
fpop_to_fno_func  = {5, 4, 6, 3, 7, 2, 8, 9, 1};
# file popularity index to size_index mapping for each class
pi_ref_freq_parms = {Zipf, 1, sizes_per_cl, 0.0, 1.0, NULL};
# Reference pattern over popularity index is Zipf w/ alpha=1
max_cached_files  = 10000;       # Maximum no of files allowed to be cached.
cached_file_frac  = 0.025;    # Size of file-cache as frac of file-set-size
qlength_limit     = 250000;   # Max requests in the system buffer
action_type       = gen_req;  # Generate traffic (no concurrent simulation)
time_series_type  = 04;       # General arrival time-series in long format
system_st_parms = {Uniform, 0, 0.7, 0.7, 1.0, NULL};
reporting_time    = 3200;     # Statistics reporting time in secs
transient_period  = 3200;     # Time to bring system to steady state

#  print *;       # This will print out everything (optional)
debug[echo_input] = 0;        # This will turn of echo_input for time_dep stmnts
GO(3200, resume_simul);
# In this second occurrence of GO, the second argument still doesn't matter
# since the program will do the warmpup and go ahead with steady state
# simulation.  On subsequent GO's, # the second argument will matter.

##### Make sure that transient and run times are multiples of cascade_gen_slots X
##### slot_size, since the program will extend it to this multiple!!
```

7 Program outputs and invocation

7.1 Auxiliary output

The trace generator provides an boolean array called "debug" whose components can be set to get additional debug output which goes either to standard output (if the program is invoked with option -t) or to the file <in_file>.out. The possible values for the array index and its meaning are listed in Table 1.

By default, all switches are set to false. The switches can be turned on or off at selected places in order to get only the desired output. For example, "echo_input" can be enabled only for those statements where it is suspected that the statement is not being read properly. As another example, one could issue a GO statement to run the program for some time, then turn on some run-time switches, run for some more time to get more detailed output, and turn off the switches

switch_name	Explanation
The following switches are in effect during program reading	
echo_input	Echo each input line after it is processed.
echo_tokens	Echo individual tokens as they are parsed from the input. (Useful primarily for program debugging).
init_stats	Provide program initialization statistics.
big_cum_dists	Print out big cumulative distributions (these are not printed by init_stats to avoid clutter).
The following switches are in effect during program run	
rand_no_gen	Print out unif(0,1) and the transformed random numbers as they are generated.
file_no_gen	Print variables connected with the generation of the files accessed by the requests.
mass_redist	Print statistics related to the cascade construction process for introducing multifractal properties in the traffic.
binary_search	Print binary search progress for finding keywords.
mgi_actions	Print insert and deletes from the $M/G/\infty$ queue used in arrival process generation.
queue_actions	Print inserts into and deletes from the system queue.
fc_actions	Print stack management actions connected with maintaining the LRU file cache.
arrival_times	Print arrival times in a slot before and after adjustments for coefficient of variation (CVAR) different from 1.
loading_parms	Prints out necessary information every time the loading level changes. See section 3.5 for clarification.
time_dep_parms	Prints out values of time-dependent parameters at the beginning of every time-slot.
transient_stats	Print traffic statistics during the transient (or warm-up) period.

Table 1: Auxiliary output switches for Geist trace generator

to return to the normal output.

The switch transient_stats only controls the normal statistics that is spewed at the end of each "reporting_time" period. It does not control other other outputs, say, for example the output corresponding to the swithc "arrival_times" above. In the current version, a decision was made to disable all auxiliary output specified by the above switches during the transient period. If such output is desired for debugging purposes, one way to obtain it is by setting the transient_period to 0 (or some other small value) so that the auxiliary output will start precisely when the user wants it.

7.2 Trace Generator Compilation and Invocation

The trace generator has been verified to work on both Redhat Linux 7.3 and MS Windows NT/2000. Before compilation, you need to get hold of random number package called "ranlib" which is available under GNU public license from http://netlib.bell-labs.com/netlib/random/. You only need ranlib.tar.c.gz. For compilation, you need to go to the "geist_trg" directory. Compilation steps are as follows:

Linux: Use the included Makefile (in the src directory) with "ranlib" source directory path set up correctly. Before using Makefile use the instructions in it to build the ranlib archive libran.a. Also make sure that the compile time variable "WINDOWS" at the top of the file create_file.c is not defined.

Windows: (Use of Microsoft Visual C++ 6.0 is assumed here.) First build the ranlib static library or a DLL. Next create a new project called geist_trg. Include various files as follows: (1) include mginf.cpp, misc.cpp, read_parms.cpp, search_ex.c under source files, (2) include misc.h, global.h, ranlib_cpp.h and search_ex.h under header files, and (3) ranlib library, and mginf_vars.c under resource files. Do not include create_file.c anywhere but do make sure that the compile time variable "WINDOWS" at the top of this file is defined. Build the project.

The trace generator is run using the following command:

```
geist_trg [-a] [-c] [-n] [-r <seed>] [-s] [-t — -o <out_file>] [-r <rn_seed>] <in_file>
[<settings>]
```

where

<seed> The new seed value (a positive integer).

<in_file> is the input file containing the program parameters. It carries the extension .inp, but is given without extension.

<out_file> is the (optional) output file containing the run statistics.

<settings> are program parameter settings made directly from the command line.

The program options are as follows:

-a option appends (instead of overwriting) all of the above output files.

-c option creates the file-set and then proceeds as usual.

-n option means no simulation; program quits after initialization.

-o option allows a different output file name, given without any extension. All output files (not just the .out file) take on this new root name.

-r option provides an arbitrary seed to the random number generator.

-s option means that an existing trace file should be split into client specific files. The current value of num_clients in the input file must be same or a submultiple of the one used for trace generation.

-t option redirects .out file to stdout; other files are not affected.

Note that the -c option does basically the same thing as the statement “action_type = create_fc;” in the input file. The only difference is that with the -c option, one could still specify some other action in the input file and the program will proceed with that following the file-set creation.

The program creates 4 output files, each with identical root name, but with a different extension as follows:

.out file contains all initialization and simulation statistics.

.trc file contains the generated traffic trace (optional).

.arr file contains arrival time-series w/ window_size buckets (optional).

.qlts file contains queue_length time-series for queue simul. (optional).

If the -o option is specified, the root names of all these files is the same as <out_file>, otherwise, it is same as <in_file>.

It is possible specify the values of input parameters directly from the command line. All such specifications override the ones in the input file since they are read in after the input program has been read completely. All command line assignments must be enclosed within a pair of ” ” in order to avoid any special interpretation by the shell. They also come syntactically after the input file. For example:

```
mginf myfile ”avg_think_time = 7.5; action_type[gen_req] = true;”
```

Note that the last semicolon is necessary, else the program will report a parsing error. As in the input file, the RHS of the assignment could be an arbitrarily complex expressions, although the main use of command line specification is to change values of a few parameters.

Section(s) on GEIST's Traffic Generator

8 Usage Information for GEIST's Traffic Generator

GEIST's traffic generator engine basically generates the stream of requests as specified in the trace file(s). Typically, the trace file(s) are created using GEIST's trace generator (as described earlier in this manual). Currently, the traffic generator only supports the generation of GET requests (supporting other HTTP transactions is planned for the future).

9 Source Package for GEIST's Traffic Generator

The source package provided was developed on Microsoft's VC++ package. To build it, you need to go to the "geist_tfg" directory in the distribution. In addition to the provided source code, you will need OpenSSL components (to enable generation of secure traffic).

9.1 Installation of OpenSSL Components

GEIST allows the incorporation of OpenSSL (version 0.9.6h) to make requests for secure documents via the TLS protocol. In order to build GEIST, you will need to download and build the OpenSSL libraries (http://www.openssl.org/source). GEIST has been tested using version 0.9.6h of the OpenSSL libraries. Please refer to the documentation with OpenSSL for building the OpenSSL libraries.

A successful build of OpenSSL will generate among others 2 DLL's (ssleay32.dll and libeay32.dll) and 2 library files (ssleay32.lib and libeay32.lib). The build process for GEIST requires that the project link options for additional dependencies (Project => Properties => Link, Additional Dependencies) contain a link to the generated library files (ssleay32.lib and libeay32.lib). This also requires the header files (from OpenSSL) to be copied to the *include* directory. In order to run GEIST using OpenSSL, the corresponding DLL's (ssleay32.dll and libeay32.dll) must be copied to folder used during runtime.

9.2 Command Line Options

The following is a command line usage description for GEIST.

```
geist.exe [-p] [-m configfilename] [-x bcastportnum] -f [tracefilename] [-l]
-p      :       To be used for prime client(s) only
-m      :       Config File Name (config.txt by default)
-f      :       Trace File Name
-x      :       UDP port number for info exchange between clients
```

```
-l      :       To be used only if a logger is used on the server
```

10 Features of GEIST's Traffic Generator

GEIST allows you to generate the following types of traffic:

Static and Dynamic GET requests – ReqCmd of each transaction in the trace is always "GET"

Non-Secure and Secure Requests – ReqType of each transaction can be either 0 (for non-secure) OR 2 (for secure)

Requests to Cacheable or Non-cacheable – ReqType of each transaction can be 0 (for non-secure, cacheable) OR 1 (for non-secure, non-cacheable)

Direct Requests OR Requests via a Proxy – The configuration file (described in more detail in the next section) has an entry for the proxy server address (if this is left as 0.0.0.0, then a direct web server request is assumed)

Overload Situations – This can be easily done by generating a trace with periods of overload activity as desired.

11 Execution Architecture of 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. 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 overall trace file mentioned earlier 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 various other configurable parameters like the number of sender/receiver threads. These and other parameters specified to control the behavior of the traffic generator are provided as input to the request generation engine in a configuration file such as the one shown

Parameter	Sample Value	Explanation
web_server	10.11.12.13	IP address of the web server under test.
port_no	80	Port number of the web server usually 80
ssl_port_no	443	SSL port number of the web server
proxy_server	0.0.0.0	Proxy server IP address (0.0.0.0 for no proxy)
proxy_port_no	790	Port number of the proxy server.
sender_threads	10	Number of sender threads per client
recv_threads	100	Number of receiver threads per client
config_port	5000	UDP port no. to broadcast global information
warm_time	1	Warm up time
run_time	30	Run time for the test run
time_to_start	2	Virtual clock reset delay at each client
req_per_attempt	1	No of requests by each sender thread on wakeup
cipher_suite	RC4-MD5	Cipher suite to be used for SSL handshake

Table 2: Example Configuration (see provided sample config file for more details or entries)

below. Finally, the prime client is also responsible for time synchronization amongst all clients. 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. 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 as follows. In this initial implementation of Geist, prior to sending a request, the sender threads obtains a handle on a receiver thread for each request that they are going to make. This receiver thread is 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, ensures that there is no feedback into the request generation process and requests are generated independent of the load on the server.

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.

12 Tuning GEIST's Traffic Generator

The traffic generator can be tuned to provide improved accuracy by adjusting the following paramaters:

1. Number of GEIST clients used – the higher the better (but lack of synchronization between clients is an important consideration to keep in mind)

2. Number of Sender Threads – the higher the better (but client processor power needs to be kept in mind)

3. Number of Receiver Threads – the higher the better (needs to be enough to generate sufficient outstanding requests)

4. Number of Requests per Attempt – the lower the better (depending on whether arrival rate characteristics are more important OR amount of load is more important)

Outstanding Issues

13 Documentation / Implementation Issues

This section contains a few things that could be improved in GEIST.

13.1 GEIST's Trace generator

```
1. No perl scripts yet to convert HTTP logs into the same format as
   generated by the trace generator.

2. Declaration of distribution variables -- Declaration is handled
   properly, but the assignment of the type A=B where both A and B are
   distributions is not implemented.

3. Order of arguments in GO statement: should the run-time really be
   first? Also, should the warmpup really be implicit or explicitly
   performed by the user.

4. All parameters should be specified in the trace generator input file
   and then passed on to the traffic generator.

5. Service in the system queue needs to correctly handle not only the
   file-serving part (which is okay) but also the overhead of executing
   ASP scripts. Needs a careful reexaminination.
```

13.2 GEIST's Traffic Generator

1. The method of receiving responses from the server can be improved as follows. Instead of having a pool of several receiver threads all waiting on the open sockets, one (or a few) listener thread(s) can be used to poll the open sockets for incoming data and then assign receiver threads to these sockets. This will reduce the occupancy of the receiver threads and also the number of outstanding threads at any given time. An initial implementation of this technique is underway.
2. The documentation of the code really needs to be improved for readability.
3. It would be nice to be able to port the traffic generator to a Linux platform
4. It would be nice to add support for POST transactions and cookies (requires work on the trace generator also).

References of Interest

References

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

[2] 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.

[3] 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.

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

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

[6] K. Kant, V. Tewari, and R. Iyer, "Geist: A generator of e-commerce and internet server traffic", Proc. of ISPASS 2001, Nov 2001.

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

[8] K. Kant and M. Venkatachelam, "Modeling traffic non-stationarity in e-commerce servers", Proc. of SPECTS 2002, San Deigo, CA, July 2002, pp 949-956.

[9] K. Kant, M. Venkatachalam, "Transactional Characterization of Front-end e-commerce Traffic", Proc. of GLOBECOM 2002, Taipei, Taiwan, Nov 2002.

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

[11] 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.

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

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

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