An Approach to Quantification and Analysis of Quality in Distributed Virtual Environments

H. Lally Singh*, Denis Gračanin†, Krešimir Matković‡
*Google, New York, United States
†Virginia Tech/Department of Computer Science, Blacksburg, VA 24060, United States
‡VRVis, Vienna, Austria
lally@acm.org, gracanin@vt.edu, matkovic@vrvis.at

Abstract— **We present an implementation-independent methodology for measuring, analyzing, and comparing Distributed Virtual Environment (DVE) system performance. The methodology comprises of a process of requirements elicitation and their conversion into measurable objectives. The process for determining quality requirements for a DVE is discussed, with a focus on interaction–based scenario analysis. A measurement tool is introduced to provide the low-overhead, high-rate metric data required for measuring DVEs while running under simulation or production operations. An example measuring a DVE's primary loop is presented.**

I. Introduction

Quantified quality is an important topic of interest in Distributed Virtual Environment (DVE) systems. With increasing demands for better performance, better scalability, and additional functionality, quantification becomes a natural component of a system improvement process. Key concerns within measurable performance objectives include the number of sustainable users within a given resource envelope, synchronization facilities to support specific simulation accuracies, and high-demand collaborative interactions.

With all the problems DVE system designers have to solve, a common subset is present throughout: synchronization and simulation [1], [2]. These activities are often significant components of a DVE engine's workload — driving significant parts of the resource requirements for operating the system. While we analyze entire DVE systems, we focus our analysis for these two areas specifically in the hope that the models and instrumentation points we have built can be reused with minimal effort.

The DVE types have different synchronization systems, but they have a common structure that we will exploit in the analysis therein. While some systems may have different classes of synchronization connections — peers, clients, identification servers, etc., we believe that they can be analyzed with the process described in this paper.

Focusing on the most reusable subset of a large family of systems, we have chosen to focus on client-server DVE systems. The choice offers both a good demonstration of the methodology and a good set of reusable results — more sophisticated topologies will have to do the same work.

While the process depends on standard algorithm analysis and software engineering techniques, a gap was found in available instrumentation tools. A new tool, `ppt`, is constructed for the low-overhead, high-rate, and high-precision requirements of DVE analysis.

II. Related Work

While DVE engines simulate physics in discrete time, other events simultaneously occurring lend them to use discrete-event simulations internally. These other events include network and user interface I/O, timeouts, etc. Using discrete-event simulation, network and input events may be handled together within a simple framework [3]. Simulations have accuracy requirements unique to their context, those appropriate for DVEs will be discussed later in this paper.

Kim et al. [4] specifically discuss and advocate for incremental development of both VR systems and behavioral models of them. They developed a CASE tool called ASADAL/PROTO that assists in the construction of message-sequence and data-flow diagrams, as well as statecharts. Seo et al. [5] performance engineering through a process they call "LOD Engineering." In this process, a budget is determined for the maximum number of polygons to simultaneously display. Then, the budget is split into different tiers of Level Of Detail (LOD), with higher tiers corresponding to objects that have fewer per-object polygon budgets and less user sensitivity to their quality.

Robinson [6] provides a high-level methodology for analyzing the quality of a simulation, in terms of software engineering and the process of social change:

1) *Quality of the Content* — How well does the simulation development process fits with the original requirements? This includes problem specification, development of the model and software, experimentation, analysis and reporting.
2) *Quality of the Process* — How was the simulation work performed? It is discussed in terms of "the socio-political work" which may include the relationship between the modeler and customer, stakeholder confidence, and timeliness of the work.
3) *Quality of the Outcome* — How useful the work is within the scope it was done for? This includes short-term action done as result of the simulation, and long-term perceptions of the utility of the results.

Watson et al. [7] evaluate the effects of frame rate on task performance. The research evaluated both frame rates and variations in frame rate as dynamic factors affecting user task performance. The research showed that user susceptibility to changes in frame rate is higher at lower frame rates (10 Hz) than at higher ones (20 Hz). At the higher frame rates, variations "have little or no effect on user performance" [7] for the types of tasks they used in the experiment.

For traditional First Person Shooter (FPS) games, Quax et. al. [8] provide an analysis of the effects of latency and jitter on user performance, as measured by their in-game-score in Unreal Tournament 2003. Using a router that simulated specific latencies and jitters, they found that users had noticeable impairment when the round-trip-time (RTT) surpassed 60 ms. Bhatti and Henderson [9] found that the number of times the player kills per minute (KPM) dropped from 1.456 KPM to 0.6233 KPM when lag was artificially introduced. Similarly, the number of times the player was killed per minute jumped from 0.6042 KPM to 1.430 KPM.

Rolia and Vetland [10] discuss both direct measurement and statistical methods for determining the resource requirements of distributed applications. Direct measurement is effective, but getting good measurements can often be very difficult, even more so in heterogeneous environments. When not possible, the statistical methods they discuss can be used — assuming that its possible to get representative input that can drive a good sampling of the systems usage.

Qin et al. hand-instrument code that sends measurements to a performance data server, which collects and analyzes the data, building a performance model [11]. In their example, they instrumented and measured a DCE application to build a Layered Queue model.

Corwin and Braddock [12] investigate the use of metrics in distributed systems, from development through operations. They examine the values of development metrics and models to quantify assumptions, and operational metrics to assist in adjusting system policies and planning upgrades.

Benford et al. [13] use a feedback loop of Area of Interest (AoI) information to determine the best clusters of data flows. Using these clusters, they continuously (re)allocate QoS streams to for better DVE performance. Similarly, Chuang and Wu [14] used adaptive network QoS monitoring to adjust the coding rate of transferred MPEG media to maintain a given quality level over the network.

At the network level, the Internet's best effort service is only one of many defined. RFC 2211 [15], allows for controlled load service, acting as a best effort link over a lightly loaded link. This reduces loss and jitter significantly, leading to stochastically stabler performance for DVE synchronization. The DVE will provide more predictable performance due to the additional stability in its synchronization system.

III. Proposed Approach

We present a process methodology, derived from Software Performance Engineering [16] to construct a scalable DVE system. Specific focus is given to the distinguishing traits of DVE systems: (1) when the load is steady, the system runs in a steady-state, continuous form; (2) performance of the system is expected to vary with available resources, gracefully adjusting as they change; and (3) large components of the system load are determined by both the human behavior in its players and the system-specific artistic assets created to populate the scene graph.

We use three resource estimators as the system performance model to manage through the engineering process: network bandwidth, CPU capacity, and memory (real available memory) requirement.

Developers may create additional models for additional resources which may bottleneck during runtime, such as dedicated hardware (e.g. server-side CUDA GPUs) or router memory. We believe that the methods used for determining the three presented are easily transportable to other resources as needed.

We define a DVE's scalability as the set of relationships between the number of logged-in users and the system's resources. These relationships are expressed as functions upon the number of logged-in users.

We also define three categories of models:

1) **Load Model** that includes the actual values of variables and the distributions of events coming into the system at different levels of load;
2) **Resource Model** that translates the load model's values into actual resource requirements (CPU, memory, network bandwidth), the RR-envelope for short; and
3) **Performance Model** that describes the system behavior requirements for acceptable performance. This model often includes: the minimum update rate to a client for their own state, the minimum rate for a client to receive updates about objects fitting various criteria, and the numerical accuracy of the physical simulation.

The Load Model has separate estimates for each software component involved in critical scenarios (Table I).

TABLE I
System parameters

	Input Processing	Simulation	Output
Load(N)	=(param_inp)	(param_sim)	(param_out)
mem(N)	=mem(N, param_inp)	+mem(N, param_sim)	+mem(N, param_out)
cpu(N)	=cpu(N, param_inp)	+cpu(N, param_sim)	+cpu(N, param_out)
net(N)	=net(N, param_inp)	+net(N, param_sim)	+net(N, param_out)

The Resource Model estimates the resource usage ramifications of those components at those loads, and totals them into a top-level estimate for the system load. The steps are listed below. We prefix each with a letter denoting which cycle they belong to: (P)reflight, (A)nalysis, (E)ngineering, and (M)odeling.

1) (P) *Assess Performance Risk* — Determine what levels of scalability are desired, and how much engineering effort it is worth.
2) (P) *Identify/Establish Critical Paths* — If designing a DVE from scratch, determine the sort of DVE to

build: a peer to peer system, a client-server system, or a federated hybrid. If using an existing system, determine which it is. Once classified, determine which software components compose the performance and scale-critical paths through the software.

3) (P) *Identify Critical State* — Determine what data structures are used for the critical paths. Common examples include the scene graph and any per-client connection state.
4) (P) *Establish Scalability Objectives* — Determine the critical variables that determine load, and what performance envelope we want the system to exhibit under that load.
5) (A) *Gather Behavior Data* — Gather users usage data to determine how users commonly end up using the system.
6) (A) *Build/Update a Load Simulator* — Modify or re-implement the client software to behave, without human intervention, representatively like the observations.
7) (M) *Build/Update Models* — Construct or update the load, resource requirement, and performance models.
8) (M) *Evaluate Assets* — Analyze the structure of the artistic assets, to determine their effects on load. For example, the vertex count of the player's avatar, to determine number of comparisons executed during a collision check.
9) (A,E,M) *Instrument Engine* — Place instrumentation into the engine for the load, resource requirement, and performance models.
10) (A,E,M) *Simulate* — Run the load simulator against the engine, enable instrumentation, and record data.
11) (A,E,M) *Evaluate Model* — Look at the instrumentation data and determine if the model gives enough accuracy, if the engine can provide the performance required, or if it is even objectively feasible to build such a DVE with the performance requirements. If the model is insufficiently accurate, go back to the "Build/Update Models" phase. This back-arc completes the Modeling Cycle.

    If the engine does not perform sufficiently well, go to the "Update Engine" phase, which will carry on to "Build/Update Models," completing the Engineering Cycle. If the DVE itself is infeasible with current technology, then modify it in the "Update Engine" phase and go back to "Gather Behavior Data" for the new DVE. This closes the Analysis Cycle.

    If the models are sufficiently accurate and show a sufficiently-performing DVE, then no additional cycle is needed and one may consider the process DONE.
12) (A) *Update Environment* — Changes in aspects discovered during analysis may be needed for the virtual environment as a whole. For example, it may be found that there are large open areas where many players congregate, causing quadratic load on the collision detection system or sizes of updates to clients. In such a case, some partitioning of that space, or motivations for users to go elsewhere, may be beneficial. After the environment has been updated, new behavioral data is required, and the analysis cycle begins it next iteration.
13) (E) *Update Engine* — With behavioral and instrumentation data, some changes may be desired to make the system perform. Changes from tuning algorithms to changing entire software components may be necessary. When complete, the engineering cycle restarts at the "Build/Update Model" stage.
14) *DONE* — The engine has been measured to perform as desired within the expected resource constraints. Simplify the instrumentation to reduce overhead, but leave enough to allow observation of the system in production.

The phrase "Build/Update" denotes that the first time through that part of the cycle, something must be built, while the second and onward times, the built thing must be updated somehow. We presume that an initial design is available at the outset; it should, at a minimum, be a list of software components to be used to fill the requirements.

There may be an initial "preflight" phase that is executed once for the project. There may be cases when preflight instructions need re-execution, but we'll consider them a restart of the entire process. Next, we have three cycles: analysis, engineering, and modeling. The analysis and engineering cycles contain the modeling cycle.

The resulting process is shown in Figure 1.

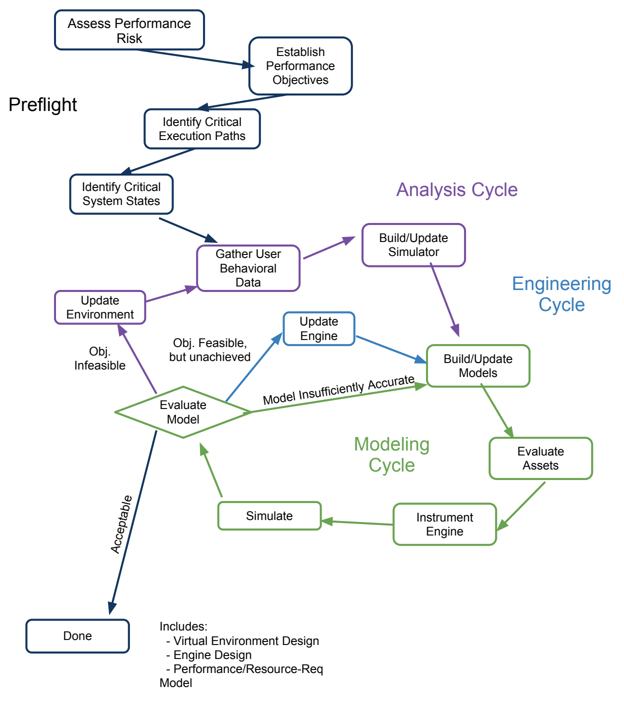

Figure 1. Analysis and modeling procedure.

Our incremental method starts with an analysis of the system. We map the critical behaviors to specific paths through the code of specific software components. We denote these paths as critical paths. We break the code down into blocks, put together a rough $O(f(N))$

model for it — for some function $f(N)$ on the number of logged-in users N — and add instrumentation to confirm or refute that model. In the confirmation case, the instrumentation should serve as a basis for calibrating the model. Simulation will fill the model in with data from the instrumentation.

For DVEs, we start with the bulk-sum resource usage: the total per-top-loop time for the engine, the total memory usage, and the total bandwidth used by the process. Iterations through the modeling cycle will indicate which model components need finer accuracy to determine whether a constraint or requirement is met. Some parts of the system may fit within the constraints with sufficient margin that only the crudest model is needed.

Confidence interval analysis should indicate which components of the model are acceptable and which ones need additional modeling work. Additionally, simulating at high levels can identify software components that will need additional work — be it modeling, re-engineering, or DVE concept alteration — to become acceptable. For additional details [17] is recommended.

A. Preflight: Risk Exposure, Objectives, Paths and State

When beginning the scalability engineering process, or the engineering project as a whole, some initial work up-front is needed.

First, the value of the system's ability to handle load must be determined in terms of its balance with available engineering time. During the process, the return on the investment of effort — in terms of performance enhancements — must be understood such that realistic goals may be kept, and to provide guidance in terms of major re-work versus less major tuning of the system as-is.

If a system is intended to be primarily used with a small number of simultaneous users for some time, with intentions of scaling higher later, it may be worthwhile to only work for a smaller load-case now, and expend the additional engineering effort later in the system's lifetime.

An initial, minimal run through the modeling and simulation stage may be useful to determine the current state of affairs. The overall process is intended to be incremental, with initial phases using very coarse models, instrumentation, and simulators at the start. A single engineer could put together a minimal load simulator involving recordings of themselves only, repeated up for a given number of players.

Once the effort is properly scoped, the desired performance objectives are stated. Specific quantitative performance values are best, when tempered with a good understanding of the cost-benefit tradeoffs for them. The specific values can be used to structure the model, as discussed later.

Finally, the software system can be analyzed. Even if the software is not complete, the system must at least have the overall design — the set of software components and their interconnections — specified. Using that, the system state and the paths through the code can be elicited. The critical system state will include items such as the scene graph and per-client data structures such as any retransmit data, sliding window buffers, etc. The critical paths are traces through the code, across component boundaries, that act as primary drivers for the performance and scale characteristics of the system. Key candidates include the simulation engine and the networking system.

B. Analysis: Data Gathering and Simulation

Before diving into the systems analysis, data must be gathered on how people will use the virtual world. Human behavior can be complex, and is easier to observe and record than to predict. To that end, a user behavior study is the first part of the analysis cycle.

The system is set up in a lab environment, with participants on individual machines. The machines are set to record their screens during the users' use of the DVE. The server is set to record basic instrumentation — at the most basic level, this may be the output of top, but preferably includes some basic "white-box" instrumentation as will be used later in the modeling cycle.

User behavior is recorded for different counts: a single user, a pair, three at a time, and upwards until the maximum number of feasible users (determined by lab space, ability to recruit, post-activity data analysis capacity, etc) is met. Each user count level should be run until a stable state is reached. In the case that the users have to participate in some linear activity, such as a collaborative task or fantasy quest, then they should repeat it a few times to get stable behavior. If substantial learning occurs between iterations, new groups of participants may be needed for each cycle.

A two-phase analysis is done to analyze the gathered data. First, each recording is played back individually. At a sample interval relevant for the DVE, categorize the user behavior. Each category is a distinct behavior that could load the DVE system differently. Each interval should be fast enough to catch any short behaviors representatively within the duration of the study.

Next, combine all the frequencies of all the categories. If two categories end up overlapping, the recordings may need re-analysis for a new categorization scheme. Again, the criterion for this is the performance requirements to support the behavior acceptably and the sorts of load applied upon the DVE system as a result.

C. The Inner Cycle: Modeling

One part of the preflight process is determining how much effort to put into the analytical process. We use an incremental analyze-and-simulate method to both let us scale the effort, and focus it on the most quantitatively significant parts of the system.

When we find unacceptable resource usage, we construct a more detailed model. The model illustrates the RR-envelope ramifications for the software components chosen to build the system. DVE software engineers can then reexamine their design decisions to squeeze out better performance.

1) Code Analysis: Our term Code Analysis is a calibrated algorithm analysis technique, applied across the critical code paths.

Load Reification: First, the we convert the load values from our load-performance envelope to real expected event values and rates within the source code. For our load parameter N, we expect that each client will transmit updates to us at 10 Hz. That is $10N$ input packets per second, and roughly $10N$ state changes.

Additionally, that means that our scene graph has N avatars, plus a number $proj(N)$ of projectiles in-flight. We will propagate the input values into values of key variables of the software component that handles input and the scene graph. We can start with estimating the values of only a few variables, and go further in-depth as we find is needed. Most importantly, we instrument the values we put in our load model. Instrumentation techniques are covered in Section IV. This way, we can verify our load model.

Value Propagation: The reified load values are applied to the code within a software component. When that component uses another, we use the relationships between the load values and the parameters of the invoked component to build a load model of the called component. Again, instrumentation is added as desired to verify this load model. Through this mechanism, we propagate the reified load values into a full load model for the the transitive closure of the call graph of the critical paths of the system.

Resource Model Derivation: With the component load model complete, we convert the load into resource requirements. Normal algorithm analytic techniques can derive a basic resource model for the CPU: we simply add instrumentation to calibrate the constants. Network I/O is a transmitted rate expectation, multiplied by a transmitted size expectation. Additional fixed overhead can be added as a last constant on the expectation. Memory size is more complex. We have to build an expected state of data structures within the system under load. This includes the scene graph and any network protocol data. In the latter category we have any data needed for reliable transmission of events, protocol data, and network socket information.

D. Back-Arcs: Analysis, Engineering

Had some of our performance elements not been sufficient, several options are available. In the most serious case, the model may show that the desired performance isn't feasible within the constraints and the virtual environment. In other case, the engine may need a software component change, or some tuning. Alternatively, the model may simply have too-large error bounds for sufficient confidence.

In the worst case, we may have had to modify the virtual environment. For example, some large areas in the DVE may simply have too many users interacting, requiring that their respective updates include each other, or their respective simulation work to involve too many other objects or avatars. The experience may have to be rethought fit within the resource and performance constraints. Alternatively, some specific tasks within the DVE may simply be too straining. For example, a user interface for remote teleportation that gives live previews of possible landing sites may simply require too many updates from too many parts of the DVE. Or, a user with binoculars may require too many changes to their area of interest anytime that they move. Such concepts would need adjustment to make the DVE feasible.

If the DVE does need conceptual changes involving the ways that users may interact with the system, then additional user-behavioral data must be collected after the changes. The simulator needs to then be updated for the new observed behavior. After that, the engineering cycle can begin to validate the new DVE concepts.

In other cases, some component of the DVE engine, such as the the simulation algorithm — or more likely the scene graph data structure — may need to change to maintain performance in the most common states of the system. Or, some code or parametric tuning may simply be necessary to drive down come constants in the model. Whatever engine changes are needed, the changes will not change the ways the user interacts with the system. Thusly, the gathered user behavioral data and consequential load simulator are usable for the next iteration. An updated model and instrumentation for the changes to the engine are all that are needed. If some of the changes affect the system's sensitivity to properties of the assets, they may need re-evaluation.

Finally, the models may simply be insufficiently accurate. The distribution of the object simulation times may be modeled with a simple Gaussian distribution with high variance, when the reality is bimodal. The model simply needs an update for the higher-resolution component model. Asset analysis may be necessary for the updated model components, as well as engine instrumentation.

IV. Instrumentation Techniques

Collecting time-sensitive measurements from running applications can be difficult. The collection process may take enough time away from the observed application that it alters the results. For applications running in production environments, the collection process may occur occasionally, causing the process of enabling and disabling the collection to potentially interfere with system performance.

The `ppt` tool was created to specifically service these constraints. Unlike more general solutions [18], within the observed application, it only requires one system call each for startup and shutdown, and only a small structure fill and copy during observation. No specialized in-kernel work is needed. In high-load situations, where the observations are less important than the timely execution of the primary work, `ppt` treats observer-CPU starvation as null values, resuming normal observations when the scheduler allows. The observation and storage of data is a separate process, which can be prioritized much lower on the system scheduler than the production programs.

Users describe *frames* of individual data to `ppt` — traces, or fragments thereof — and `ppt` generates code to link into the executable to be measured, as well as a reader program. Will will first discuss the frame description, then embedding the generated code, the transfer mechanism,

```
/* Options section */
emit C
/* required, and the only current value */

/* Frame declaration */
frame example {
  int   kind, who;
  time  sleep, start, end;
} // no semicolon needed.
```

Figure 2. A frame description format example (comments are as in C99).

the capture process, and finish up with a discussion on caveats.

A. Describing Frames

The goal is to organize the measurements into *frames*, roughly corresponding to a subset of C `structs`. `ppt` generates code to save the current value of each member in a frame, and to emit the frame out to shared memory. The user will be responsible for invoking the code save values for each member, and for emitting the frame.

The frame description format is best explained by an example (Figure 2):

An initial `emit` statement indicates the language of the observed program. Only “C” is currently supported, but the generated code will compile and link with “C++” programs.

This example declares a frame called *example* that contains two integers `kind` and `who` and three timestamps: `sleep`, `start`, and `end`.

Each frame is automatically reordered to minimize padding overhead, and a definition for C is generated. The only types allowed are `int`, `float`, `double`, and `time`. The first three are their native equivalents on the platform, and the last is a single timestamp — currently a `struct timeval`, with microsecond precision.

If a member is not filled in before being emitted to shared memory, it will retain its previously-set value. The initial value of each member is 0.

B. Embedding Generated Code

For C, each frame will a header and source file generated (Figure 3). The header declares a C version of the frame type, prefixed with `pptframe_` and suffixed with `_t`. Additionally, it will define C macros for each member, one each for `int`, `float`, and `double`, and two for `time`.

In the example code, we have a simplified DVE core loop that waits up to 32 milliseconds (`sim_interval`) for an event. At the minimum: it won’t catch up if more than one simulation step is required to meet real-time and doesn’t consider the event processing or simulation delays in recalculating the timeout. The event can be a transmitted user update (the most frequent case that we expect), a user login, or user logout. When an event is received, the remaining part of that 32 milliseconds is returned (`Event`’s `remaining_time`). That remainder is the time we wait the next time through the loop. Eventually we time out, either due to a period of time with no events, or a zero remainder from the prior event wait.

```
#include "example.h" // the generated header

struct Event {
  enum { .. } kind;
  int remaining_time, user;
  ... // the event data.
};

//  Wait up to 'timeout' for a new packet,
//  returning it or NULL if timed out.
extern Event *next_event(int timeout);

void primary_loop() {
  int remaining_time = 32;
  while (1) {
    struct timeval my_sleep;
    gettimeofday(&my_sleep);

    // manually save the time
    WRITE_EXAMPLE_SLEEP(my_sleep);
    Event * ev = next_event(remaining_time);

    // or use the provided macro
    WRITE_EXAMPLE_START_TIMESTAMP();
    // handle the event
    if (ev) {
      handle_event(ev);
      remaining_time = ev->remaining_time;
      WRITE_EXAMPLE_WHO(ev->user);
      WRITE_EXAMPLE_KIND(ev->kind);
    } else {
    // it was a timeout.  Run the simulation
      WRITE_EXAMPLE_KIND(-1); // -1=timeout
      WRITE_EXAMPLE_WHO(-1); // no user.
      simulate_upto(1);
      remaining_time = 32;
      send_new_states();
    }
    WRITE_EXAMPLE_END_TIMESTAMP();
    ppt_write_example_frame();
  }
}
```

Figure 3. Instrumented Code

At this timeout, we simulate the virtual world forward a single time step (of 32ms). Using `ppt`’s generated macros, we store the kind of event (update, login, logout, timeout), the amount of time slept for the event, and the time taken for processing the event. Note that in this example, if more than one `Event` is available, `remaining_time` will be the same as `timeout`.

In this example, clients will transmit a `LOGIN` event (with an undisclosed server response), followed by 10 Hz `UPDATE` events, followed by a `LOGOUT` event.

When building our example DVE, we simply need to link the generated `example.c` file with the program executable.

The data collected per event is relatively small, but the information inferable from it is substantial:

1) *Core Loop* — The durations of `end–start` timestamps indicate how much processing time is spent in the engine per event. Summing the ranges into per-second groups, and dividing by one second indicates the proportion of the processor the system is using at the measured load level.
2) *Event Frequency* — The tuples of (`kind`, `start`) indicate how often events are received at a given load configuration. Note that this is at maximal observation rate allowed by the core loop — if the core loop was using less of the processor, it may have been ready to receive them earlier.

3) *Event Distribution* — The tuples of (who, kind, start) provide data for several analyses. During periods of bursty logins or logouts, we'd be able to observe their packet-receive distributions in relation to the actual events; we use real timestamps, so we can correlate these event times to the login/logout times of the load simulator. Additionally, the jitter of user update times can be observed for possible client-side effects (e.g. uneven client CPU utilization may result in high transmit jitter) or network link effects. Finally, we can look for "phase" effects. If users logged in at uniformly-distributed times, we'd expect (and observe through LOGIN) that their update times stay uniformly distributed. A uniform event distribution keeps the CPU load even and predictable. If the events start forming clusters, the responding CPU work can result in bursty CPU utilization.
4) *Data Loss* — With the same tuple of (who, kind, start), we can detect missing events. A client should be transmitting its state at a constant rate of 10 Hz. If some clients transmit slower, we can further analyze them for the root cause. Note that to verify the lack of an event, we have to make sure there were no gaps in the *sequence number*, discussed in the next section.

C. ppt's Transfer Method

The transfer mechanism involves three programs. First, the program being observed has its generated ppt_write_*_frame() function, with the associated macro calls to fill in the frame. Second, an *agent* program creates shared memory and notifies the program being observed of it. Finally, a *listener* attaches to the shared memory and saves everything observed, in order. A fourth generated program converts the saved binary observations into a text table, and is runnable when the observations are complete.

The frame is converted to an in-memory layout, and the members are reordered to minimize space while keeping them on their respective alignments. A sequence number is prepended to the frame definition. The number is incremented at each frame.

ppt declares a C struct to represent it, a global instance of it, an integer for the shared memory handle, and a pointer to the mapped shared memory instance. By default, the handle is zero, and the pointer isNULL.

The ppt_write_*_frame() routine will check the buffer pointer — to determine if observation is enabled — and copy the structure's values to the buffer. Upon hitting the end of the buffer, it will start writing at the beginning again.

The listener reads as many frames it can with increasing sequence numbers — minus on exception for sequence number roll-over — and writes them to disk with no translation, then it sleeps. The sequence number constraint prevents reading an old value again.

1) Overhead: The overhead on the listener can be counted directly. Each call to a member-writing macro is a single variable assignment of that variable's type. Each frame write includes two comparisons, one structure copy, two writes to the sequence number, and an increment, comparison, and reset of a pointer. The sequence number also acts as a temporary lock.

When the agent first attaches on, the observed process will invoke shmat(2) to attach the shared memory. Similarly it will invoke shmdt(2) when the observation has ended.

2) Rate Adaptation: If the observed process writes faster than the listener can read, some values will be lost — but their loss will be detected in a gap in sequence numbers, the listener will continue to read data and adapt to the new rate. The rate adaptation occurs by modulating the sleep time. Similarly, an event slowdown will result in the delay increasing, up to a hard-coded maximum of 2 seconds.

The initial sleep delay is currently hard-coded to 100ms. The delay is adjusted if less than 1/8th, or more than 7/8ths of the buffer was read. The sleep delay and read-frame count are used to estimate the current write rate, and the delay is then adjusted to match 1/2 of the buffer size. A hard-coded minimum of 10ms throttles the maximum CPU time the listener can take.

The read-write process continues until the agent notifies the program being observed to detach the shared memory. The listener is terminated, and the shared memory removed.

3) Notification Mechanism: The term "notification" is a simplification; through the /proc filesystem, ptrace(2), and analysis of the program's ELF tables, the variable for the shared memory handle in the program being observed is directly modified. When the observation begins, the ppt_write_*_frame() routine detects the new value and attaches the segment, making it ready to write to. Similarly, when the handle variable is set back to zero, the routine will detach the memory and ignore requests to write the frame out. This mechanism further minimizes the intrusion upon the operation of the observed program.

D. Capturing Data

The listener is a custom-generated program, emitted as LLVM bitcode [19]. The bitcode format enables JIT interpretation for quick turnaround during prototyping, and full-on optimized compilation when the format is stable. It attempts to minimize the amount of work per scan through the shared memory, by re-tuning its delays and buffered I/O, it stays off the processor as much as possible. For situations where ppt can be run on a system with LLVM, but the system running the generated code doesn't have LLM, a slower C equivalent listener is also generated. When the data capture process is complete. Another ppt-generated program reads the binary frames and outputs a text file separating each field by a tab character, and every frame by a new line.

```
seqno kind who sleep.tv_sec sleep.tv_usec ...
501 1 1 1294621640 50116 1294621640 35661...
...
```

V. Results

Analyzing the Torque [20] engine, we first determined that the protocol only allowed a maximum of two packets' worth of data to transmit to each client at a time. This substantially limited the number of clients that a Torque-based system could handle at one time. In our user studies, the limits were hit in the mid 20s. Our performance risk for system quality is that we have too many updates to send one or more clients at a time, which would go well-beyond the two packets limit.

The critical path for Torque was a simple (1) process inputs, (2) simulate, and (3) process outputs cycle. Our model followed it closely. The cycle itself was dominated by simulation time. The I/O processing was essentially linear — there was a sort, but its time-coefficient was so small that it was lost in the measurement noise. The critical state of the engine is the scene graph. The per-client connection state would become more important in higher-precision models.

Using a load simulator built from observed user behavior, we were able to simulate 70 simultaneous users (each receiving a poor update rate, but still logged–in and able to provide load to the server). `ppt` allowed us very fine-grained control with little overhead. On a 1 GHz equivalent virtual CPU share, the result was a $\approx 2,500$ event/sec data feed that stayed between (CPU) 0.00% and 0.01% in `top`.

Our resulting model showed, the mean time for a single simulation step for a player was 0.2978 milliseconds, but the variance was 4.6303 (ms^2) — a multimodal distribution of times, corresponding to the hot spots within the virtual environment. Updates on the environment should work to reduce the peak simulation times, probably through reducing the exposure to those dense "hot spots" in the virtual environment.

VI. Conclusion

Through its use in the analysis of game engines in our research, the process described here provides a straightforward way to determine the scalability of a DVE by determining the relationships between load, resource requirements, and performance.

Future work on `ppt` will center on finding reliable methods to collect timestamps without invoking a system call and enable multithreaded writing to the buffer. A discriminated union type for the frame would be useful. The frame could support detailed data for multiple event types simultaneously, with the caveat that all frames would take as much buffer space as the largest variant.

Acknowledgment

This work was supported in part by Science Foundation Ireland grant 03/CE2/I303_1 to Lero-The Irish Software Engineering Research Centre. This research was also partially supported by the Ministry of science education and sports of the Republic of Croatia, grant no. 036-0362027-2028 "Embodied Conversational Agents for Services in Networked and Mobile Environment".

References

[1] H. L. Singh, D. Gračanin, and K. Matković, "A load simulation and metrics framework for distributed virtual reality," in *Proceedings of the 2008 IEEE Virtual Reality Conference (VR '08)*, 8–12 Mar. 2008, pp. 287–288.

[2] H. L. Singh and D. Gračanin, "Load characterization for distributed virtual environments," in *Proceedings of the 1st International Workshop on Concepts of Massive Multi-user Virtual Environments (CoMMVE'09)*, Kasel, Germany, 5 Mar. 2009.

[3] D. H. Eberly, *3D Game Engine Architecture*. Morgan Kaufmann, 2005.

[4] G. J. Kim, K. C. Kang, H. Kim, and J. Lee, "Software engineering of virtual worlds," in *Proceedings of the ACM symposium on Virtual reality software and technology*, ser. VRST '98. New York, NY, USA: ACM, 1998, pp. 131–138. [Online]. Available: http://doi.acm.org/10.1145/293701.293718

[5] J. Seo, G. J. Kim, and K. C. Kang, "Levels of detail (lod) engineering of vr objects," in *Proceedings of the ACM symposium on Virtual reality software and technology*, ser. VRST '99. New York, NY, USA: ACM, 1999, pp. 104–110. [Online]. Available: http://doi.acm.org/10.1145/323663.323680

[6] S. Robinson, "General concepts of quality for discrete-event simulation," *European Journal of Operational Research*, vol. 138, no. 1, pp. 103–117, Apr 2002.

[7] B. Watson, V. Spaulding, N. Walker, and W. Ribarsky, "Evaluation of the effects of frame time variation on VR task performance," *Virtual Reality Annual International Symposium, 1997., IEEE 1997*, pp. 38–44, Mar 1997.

[8] P. Quax, P. Monsieurs, W. Lamotte, D. D. Vleeschauwer, and N. Degrande, "Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game," in *SIGCOMM 2004 Workshops: Proceedings of ACM SIGCOMM 2004 workshops on NetGames '04*. New York: ACM Press, 2004, pp. 152–156.

[9] T. Henderson and S. Bhatti, "Networked games: a QoS-sensitive application for QoS-insensitive users?" in *RIPQoS '03: Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS*. New York: ACM Press, 2003, pp. 141–147.

[10] J. Rolia and V. Vetland, "Parameter estimation for performance models of distributed application systems," in *CASCON '95: Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1995, p. 54.

[11] M. Qin, R. Lee, A. E. Rayess, V. Vetland, and J. Rolia, "Automatic generation of performance models for distributed application systems," in *CASCON '96: Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1996, p. 33.

[12] B. N. Corwin and R. L. Braddock, "Operational performance metrics in a distributed system. part i.: Strategy," in *SAC '92: Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing*. New York: ACM Press, 1992, pp. 867–872.

[13] C. Greenhalgh, S. Benford, and G. Reynard, "A QoS architecture for collaborative virtual environments," in *Proceedings of the seventh ACM international conference on Multimedia (Part 1)*. New York: ACM Press, 1999, pp. 121–130.

[14] H.-K. Wu and P.-H. Chuang, "Dynamic QoS allocation for multimedia ad hoc wireless networks," *Mob. Netw. Appl.*, vol. 6, no. 4, pp. 377–384, 2001.

[15] J. Wroclawski, "Specification of the Controlled-Load Network Element Service," RFC 2211 (Proposed Standard), Sept. 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2211.txt

[16] D. W. Smith, *Brute Force: Betrayals*. New York: Balantine Books, 2002.

[17] R. K. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[18] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '04. Berkeley: USENIX Association, 2004, pp. 2–2.

[19] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ser. CGO '04. Washington, DC: IEEE Computer Society, 2004, pp. 75–.

[20] GarageGames, "Torque game engine," http://www.garagegames.com/ [Last accessed 28 Apr. 2011].