A Framework for Designing and Evaluating Distributed Real-Time Applications

Arthur Valadares, Cristina Videira Lopes
Bren School of Information and Computer Sciences,
University of California, Irvine
Irvine, CA, USA
Email: {avaladar, lopes}@ics.uci.edu

***Abstract*—Distributed Real-Time (DRT) systems are among the most complex software systems in the field of Computer Science. The contradictory nature of distributed computing and real-time requirements forces prioritization that is dependent on domain-specific knowledge. This combination of distributed and real-time system requirements generates a new set of properties, and requires a new conceptual framework.**

In this paper, we present a conceptual framework for developing DRT systems. We divide our framework in 3 phases: example applications, design properties, and evaluation. We demonstrate our framework properties with our own Distributed Virtual Environment architecture and other 2 examples: Google's Cluster Architecture and collision avoidance systems. Finally we discuss evaluation of DRT systems through the lens of our own architecture, and conclude with general lessons to apply to other DRT systems.

Our conceptual framework enables developers of DRT systems to foresee and address issues early in the design. Additionally, we expose general lessons and good practices of evaluation for other DRT systems.

I. Introduction

The first measure for a successful software design is the fulfillment of its functional requirements. Second are non-functional requirements, that measure operation characteristics of the software. In DRT applications, non-functional requirements such as scalability and responsiveness surface as a necessary component of the software design. Neglecting DRT non-functional requirements can possibly render the software useless. For instance, a collision avoidance system in cars, such as anti-lock braking systems (ABS), is distributed by nature and must be fault-tolerant and real-time. If either property is not achieved, the wheels will either not behave correctly, or not react in time, and result in a collision accident.

Designing for distributed and real-time systems is challenging due to distribution hindering real-time requirements. Distributed systems partitions applications into independent processes that can be deployed on separate hardware, communicating through a network. The inter-process communication over the network introduces a significant delay for real-time sensitive applications. Thus, it is usually necessary to define a fine balance between the desired level of distribution and real-time requirements when designing a DRT application. The functional requirements must be carefully analyzed when selecting an approach that achieves balance between distributed and real-time nature.

Evaluating the success of a chosen approach is also a non-trivial task. It is often impractical to evaluate a DRT application in terms of real-world scalability, as it may require hundreds to thousands of machines and users. It is then required to pick an experiment and metrics that can be expected to perform similarly to the real-world deployment. Yet assumptions and abstractions of real-world deployment, such as infinite bandwidth, no jitter, no thread context-switching costs, can be made carelessly, resulting on unachievable performance on a real deployment. Furthermore, metrics can be symptoms originating from multiple causes, so choosing and isolating the right metrics that demonstrate feasibility and performance of a solution also requires scrutiny.

To successfully design DRT systems, understanding requirements, properties, trade-offs, and testing are vital. In this paper we present a conceptual framework focused on design and evaluation of DRT applications. In design, we isolate properties inherent of DRT systems, and exposes the trade-offs with example applications. In evaluation, we use our experience developing a complex DVE to formulate guidelines and expectations for testing.

The paper is organized as follows: section II example of DRT applications. Section III describes 6 vital properties of DRT systems, and apply them to our example DRT applications. Section IV is a case study of evaluating our own DRT application: the Distributed Scene Graph with Microcells (DSG-M). Through the lessons learned from DSG-M evaluation, we generalize lessons in planning evaluations for DRT applications. Section VI summarizes our conceptual framework.

II. Example Applications

Designing DRT applications involve carefully balancing out the conflicting requirements of being *distributed* and *real-time*. There is not a single definition of distributed and real-time. For the purposes of this paper, we define these non-functional requirements as the following:

- **Distributed systems** are a set interdependent processes connected by a network with a singular computational purpose. Distributed systems are motivated by two major properties: **fault tolerance** and **parallelism** [1]. We discuss both properties in section III.
- **Real-time** requirement indicates that the application has time-sensitive I/O. Common examples are pipe-

and-filter patterns, such as video decoding, and interactive applications, such as virtual environments and user interface.

DRT applications become necessary when a combination of properties from distributed systems and real-time is required, meaning an application needs to be fault-tolerant or parallel, and real-time. Determining degree of motivation for DRT systems is a domain specific concern.

Next, we present 3 examples of applications in different domains that benefit from different perspectives of DRT properties: distributed virtual environments, Google's Cluster Architecture, and collision avoidance systems.

A. Distributed Virtual Environments

Virtual environments (VEs) are 3D simulations of real world objects and space. VEs have a broad range of uses, from simulation of real world events and rules such as physics, to creating interactive platforms where users can share experiences, engage in communication, and modify the world as they see fit. Users act upon the VE, and expect that their actions will be instantly perceived by other users.

Traditional implementations of VEs are done in a client-server architectural style: users connect to a single server, responsible for maintaining rules, generating reactions, and broadcasting updates to all users. This monolithic server approach has proven to be limited in scalability, both in terms of users, and of interactive or autonomous in-world entities [2]–[5]. Both requirements of DRT are necessary for scaling the number of users and providing an acceptable interactivity delay. The degree of interactivity is highly dependent on users' actions. There are several actively used DRT virtual worlds that have been successful in horizontally scaling VEs [4,6]–[9].

Many scalability approaches for virtual environments involve space partitioning techniques. In earlier space partitioning methods [11,12], space is partitioned in fixed-size large areas of space, sometimes referred as *regions* or *worlds*. A novel and more flexible approach is space partitioning through *microcells* [14], indivisible small areas of space that can be grouped to form custom shaped partitions that better adapt to load. Yet even specialized space partitioning methods alone were shown to be insufficient under certain conditions of load [13]. Another form of partitioning virtual world load proved to be effective where space partitioning failed.

The **Distributed Scene Graph (DSG)** is a client-server architecture for decoupling *Scene* and *operations* [7]. Scene is the data that represents the state of the virtual world, where operations are responsible for reading and writing to the Scene. An example of Scene state is an object's position and velocity. An example of an operation is dropping the object from a certain height, and have physics operations update its state over time. In DSG multiple simulators share and synchronize the Scene, while each simulator can be dedicated to independent groups of operations. For instance, one simulator can perform physics operations, another simulator can perform script operations, while other simulators can handle client commands. Operations read and write Scene data, and DSG synchronizes the Scene across all simulators. DSG was shown to perform better than space partitioned techniques alone under several common use cases.

In **DSG with Microcells (DSG-M)** we redesigned DSG to push scalability further, by allowing simultaneous decoupling of operations and space through *microcell* partitions. DSG-M allows for simulators to be partitioned in both dimensions, enabling better adaptation to load. DSG-M was evaluated through a physics intensive experiment, and partitioning of both functionality (e.g. physics, script) and space, by dividing the region space in half. When compared to DSG, DSG-M results showed a 15% improvement in performance in the worse-case scenario and nearly double for a perfectly partitioned space scenario (i.e. no inter-partition communication).

B. Google's Cluster Architecture

A ubiquitous DRT application example is Google's Cluster Architecture (GCA) [15], used to serve users' search queries in Google's search engine. In GCA, over 15,000 commodity-class PCs are available to process user queries in a distributed manner. Due to the escalated heterogeneous conditions and the lower quality of hardware, Google uses a fault-tolerant algorithm to achieve correct and fast results.

The distributed motivation for GCA is clear: achieve scalable search engine capability by harnessing computing power from thousands of commodity machines. Each request is an input to GCA, which has a time limit to respond with the best matches. This time-sensitivity for query responses imposes real-time requirements to GCA.

C. Collision Avoidance Systems

The first electronic components in automobiles were connected point to point. Sensors provided input to actuators that were directly connected by wires. As the number and complexity of sensors and actuators grew, so did the number of wires and cables in the vehicles. To reduce wiring and enable higher level functionality, cars adopted new architectures for sensors and actuators, connecting components through a network [16].

Modern cars are now capable of using sensors and actuators to perform highly complex functionality. For instance, many cars today have collision avoidance systems (CAS) such as anti-lock-brake-systems (ABS) and electronic stability control (ESC). Both of these systems act on the vehicle's actuators (e.g. brakes, steering) under high risk situations by analyzing data from sensors in the vehicle (e.g. speed) [17]. Failure to perform correctly can have serious consequences, resulting in a car accident.

In accident prevention systems, correct behavior (i.e. preventing a crash) is the major requirement, but achieving correct behavior depends on real-time measurements from sensors and output on actuators.

III. Design Properties

We have discussed the motivations that lead to DRT systems. Once a DRT solution is deemed beneficial for an application, how do we understand the factors and the trade-offs involved in designing a DRT application? In this section, we formulate 6 major properties that capture trade-offs and expected behavior of DRT systems: correctness, fault tolerance, parallelism, time sensitivity, consistency, and overhead costs. We show how each example application handles each property.

A. Correctness

In the abstract view of computation, correctness is deterministic. If an algorithm is correct, execution will produce correct results repeatedly. External factors such as hardware can be abstracted away and should not influence results of computation.

In real-world examples, computing is not as deterministic as software developers may originally conceive it. Hardware and physical devices such as CPU, memory, and networking can influence computation due to exhaustion of resources or heterogeneity of hardware. For instance, a video streaming quality can improve or worsen if executed on different hardware, if memory or CPU is exhausted, or if the network performs poorly. Distributed computing escalates the issue by adding additional hardware with heterogeneous configurations. Determining a result to be correct requires more malleability and room for imprecision.

In DSG-M, correctness is a product of two purposes. The first is user interaction, where correctness is measured as quality of user experience. In terms of non-functional requirements, the user experience is highly correlated with responsiveness (i.e. low latency, jitter) [18,19]. The second is the non-interactive virtual space simulation (e.g. physics), where correctness is measuring how close the simulation was compared to the expected result.

In the Google search engine, correct results are: (1) returning the webpage expected by the user, and (2) return the results in a reasonable time. The latter measurement is easily framed as an interval: the faster the results are returned, the better. The first measure also has different degrees of correctness, depending on the position the correct webpage is ranked in the list of results. Correctness is then measured as a product of request time and rank of expected webpage in results.

For CAS, correctness is avoiding accidents. The ESC system, for instance, must not allow the car to lose traction with the road. ESC systems are developed, experimented, and compared to a car without ESC, to demonstrate degree of improvement. After the initial metrics shows promising results in improving collision avoidance, effectiveness can be confirmed with data when the CAS is deployed. For instance, ABS has shown to have reduced vehicle collisions and injury severity when compared to cars without ABS [20].

B. Fault tolerance

Fault tolerance is a system's ability to survive failures and is a common property of distributed systems. Distributing computation adds more hardware, increasing the chance of a single component failing. As a distributed system grows, so does chance of hardware failure. Through fault tolerant algorithms, a distributed system can be made robust against individual component failures, typically at the cost of overhead resource usage in coordination, replication, and redundancy.

In DSG-M, fault tolerance is present in two degrees. To a higher degree, each simulator instance can be fault tolerant. In this case, if a simulator responsible for the physics simulation of an area of the virtual environment were to fail, a replica of the simulator could take its place without service disruption. To a lower degree, each individual simulator can fail, but the entire system persists. If the same physics simulator were to fail, the physics simulation for that particular area would stop, but all other functionality would still be available.

Fault tolerance is at the heart of GCA's requirements. Instead of using reliable high-end servers, GCA uses commodity-class PCs that are cheaper, but fail more often. Redundant data storage and software-based fault tolerance systems in GCA have demonstrated superior performance for price over high-end servers [15]. Even in face of total failure, consequences would not be drastic, simply resulting in error messages for user queries.

CAS applications need to have high tolerance for failure. The result of a system-wide failure on an application such as an ABS or ESC can result in a car accident. Even further, the accident could be caused by the failure, and not due to the circumstances that was leading to an accident. If, for instance, the sensors feed false information or a failure disables the brakes, the prevention system would be responsible for an accident. To avoid catastrophic failures, several measures are taken to reduce the odds of a system-wide failure, such as replicating sensors and actuators, mechanical fall-back, and fault detection [22].

C. Parallelism and Scalability

Parallelism enables computation to be partitioned and executed in parallel. Partitioning computation may require coordination: the results of the computation performed in parallel needs to be aggregated and examined. Coordination may be required only at the start and end, at a certain rate during the execution, or not at all.

Parallelism comes in multiple forms in software development. In threads, memory is shared, and coordination is achieved simply by coordinating data in memory. In multi-processing, each partition is a system process with its own memory, and inter-process communication methods (e.g. files, pipes) are necessary for coordination. Finally we have networked parallelism, where processes are executed in different hardware, connected through a network. Distributed systems have parallelism of the latter kind. Coordination requires the use of network stacks and messaging protocols between processes.

The advantage of parallelism is increasing computing power by adding more networked hardware resources, improving software *scalability*. This form of scalability is referred as *horizontal scalability*. DRT applications are time-sensitive, and parallelism enables large workloads to be performed within an acceptable time frame for real-time applications. Yet network communication costs are typically high and may void or compete with advantages of parallel computing.

Computing the simulation for a virtual environment can be an astonishingly large workload. If the virtual environment has thousands of users and objects, computing the result of each interaction at a every time step is unfeasible within the limited time frame required for reasonable interactivity, usually in the hundreds of milliseconds. DSG-M partitions this workload into multiple simulators in two ways: by space and by functionality, as described in section II-A. At every time step, each simulator computes its part of the simulation and synchronize

the results through a messaging protocol via network. Users can send actions and receive aggregated updates through a simulator dedicated for client connections, reducing the burden of simulating the environment and updating clients, who are numerous.

Search queries for Google's search engine is a highly parallelizable computational effort. Search queries are divided into two phases. In the first phase, an index server consults an inverted index to match each word in the query to relevant documents. The indexes are divided in shards, and each shard has a set of machines available for computing query words to document identifiers. Initially words are separated and sent to be processed in one or more machines. The coordinator waits for results and move on to phase 2. In phase 2, title, URL, and summary are extracted from documents, and additional services are performed, such ad-serving and spell-checking. The coordinator once again waits for all requests to be returned, then generates the HTML output page for the user. Search queries only require two coordination points: end of phase 1 and 2. The reduced coordination effort and independence of queries makes search engines good workloads for parallelism.

In CAS, parallelism is imposed due to physical separation of components. Sensors and actuators are located near the hardware they measure or control. Since fault-tolerance is essential for CAS, the computations from sensors are run in a centralized but redundant central management computer, and output is distributed to the actuators [22]. Thus there is little parallel computation involved, the distributed components are purposed for input, output, and redundancy.

D. Time Sensitivity

DRT applications have real-time requirements, meaning they have time sensitive I/O. Time sensitivity can be originated from interactivity from users or from computation of other software components, as in a pipe and filter architectural style.

In VEs, user interactivity is a major requirement. Users interact with the VE, by viewing and by modifying the world. DVE architectures must account for time sensitivity, to deliver the illusion of instant I/O. Latencies of up to 500ms can typically be tolerated [19], but higher values will give the impression of lag. Furthermore, modifications are cumulative. If the world is modified, users must be notified quickly, since the new state may influence the user's decision on how to act next. Best examples are games, many of which are reliant on user's reaction to events, such as jumping over a hole or shooting an enemy at the right moment.

In GCA, there is no user interaction during computation. After the query is sent, all computation is done server side, and the result is shown as a webpage. Instead, GCA has computational time-sensitivity when generating results. Search queries are broken down into several different computations, such as looking up reverse index of words to documents, generating ads, and spell checking. Each computation must adhere to a time limit, in order to return the ranking of webpages in HTML format in reasonable time.

CAS is both interactive and computationally time-sensitive. Upon detecting abnormal circumstances (e.g. wheel slip or wheel lock), the system must act in a short period of time. During this period, the user will also be attempting to regain control, and the system must account for the user's input and the sensors to make the best decision on what to instruct the actuators. Since CAS are local networks, low-level hardware (e.g. microcontrollers, controller area network bus) are capable of achieving both the distribution and time-constraints at acceptable rates. [22]

E. Consistency

The consistency property determines how each process node of the distributed system sees state. Will all nodes always have consistent state, or does state converges to consistent? Enforcing consistent states for every node at every point in time would require strong consistency algorithms that may break real-time requirements. For example, two and three phase-commits (2PC and 3PC) are common algorithms for guaranteeing consistency across all nodes. Data is only committed once every node agrees to the current state. Yet 2PC and 3PC require two or more network round-trips per commit in a DRT. For time sensitive applications in the range of hundreds of milliseconds, the network round-trips would interfere with the time-sensitive property. Instead, some DRT applications use *eventual consistency*. Nodes in the DRT system will have slightly different states during execution, but state will eventually converge to the same values.

To provide the interactivity level users expect in a VE, some degree of partitioning or total consistency requirements needs to be relaxed [23]. Prioritizing interactivity and scalability through parallelism, DVEs adopt the less restrictive eventual consistency requirement. Traditional DVEs do load balancing by partitioning space. Objects in an area of space have only one authority (i.e. write permission) process. For optimization, other processes may read-only copies of objects belonging to another process. Since there is only one process with write permission for each object, read-only copies may become stale, but state will always converge.

In DSG-M, different simulators may act on the same area of space, so multiple processes may be granted write permissions. For instance, an object can be moved due to a script, or due to a physics collision. DSG-M uses timestamps to guarantee that state will remain convergent. By overriding updates of lower timestamps by updates with higher timestamps, eventually the state of all simulators will converge to the update of the highest timestamp. It is important to observe that physical timestamps do not guarantee correct order of execution in a distributed system, but rather that the final state will converge. It is a domain-specific concern to determine if ordering negatively impacts correctness. As an example, deletion of objects takes precedence over other state updates, since after deletion the object no longer exists and cannot be updated.

GCA has only two coordination points, so each phase of the search query can afford a total consistency approach. Phase 1 and 2 ends when all results have returned. Furthermore, GCA rarely requires writing operations. Most operations are read-only and data is partitioned and replicated. Even if data is only eventually consistent in the replicas, the difference would not be too significant for future search queries, since queries are independent from one another.

In CAS, consistency plays a larger role. Sensors must return a very precise measurement over time, so that a proper reaction for the actuators can be calculated. Then, actuators must all according to calculations. A misread or late measurement or action can result in an incorrect behavior that could lead to an accident. For this reason, CAS have replication and redundancy of components, in case of failure, misread, or inadequate performance.

F. Overhead Costs:

DRT applications pay an overhead cost for distributing computation. Often the price is in network messaging and in coordination. Messages passed through the network stack can produce latencies from tens to hundreds of milliseconds. High frequency of messages can make latencies worse, and incur significant CPU usage for packing and unpacking messages. Coordination requires computing the partitions, distributing them through the network, and joining the results. When joining results, the coordinator must wait for all processes to respond, meaning the system will move at the speed of the slowest process. Different DRT applications have more or less sensitivity to overhead costs, depending on the degree of network messaging and coordination required.

Overhead costs of distribution in DVEs are a major concern due to 3 characteristics. First, state updates in VEs are frequent, possibly creating a large number of messages to synchronize all simulators. Second, when simulators are partitioned by space, entities need to be migrated between processes when they move out of their boundaries. The migration may involve transfer of large amounts of data, and can only guarantee consistency if the object is locked for writing during migration. Finally, VEs have unpredictable patterns of load. Users gather in unexpected locations, entities may be moving faster or slower, and can be added and deleted to the system in bulk.

These 3 characteristics make scaling VEs difficult. DVE systems must be capable of adapting to dynamic load, and modify load balancing techniques to avoid high overhead costs. DSG-M partitions space by using microcells [14] of custom size and shape, to flexibly adapt the load to the partitioning. This way DSG-M can better adapt the space partitioning to avoid excessive updates and entity migrations. In our worst-case scenario experiments, only a 15% improvement was observed when dividing the space in two. Thus nearly 85% of the computation is being used for the overhead of synchronizing the simulators. The overhead was mostly due to object migration causing numerous messages and creation and deletion of tens of thousands of objects.

Search engines have mostly a coordination cost, since not many messages need to be exchanged. Each search query is translated to a list of tasks, such as index lookup, ad generation, and spell checking. The tasks are then sent to different machines for processing. The coordinator process needs to wait for all requests to be returned so the HTML webpage can be generated and sent to the user. Thus, the query response is as fast as the slowest task. Another overhead cost of GCA is replication: data must be available in several replicas spread across the globe, to improve availability and fault-tolerance.

TABLE I. PRIORITIZATION OF PROPERTIES PER APPLICATION

Property	DVE	GCA	CAS
Correctness	2	4	1
Time Sensitivity	2	3	2
Consistency	4	*	2
Fault Tolerance	5	1	2
Parallelism/Scalability	1	1	*
Overhead Costs	6	5	5

The main cost for CAS is in coordination: receiving input from sensors, calculating actions to be taken, and executing actions on the actuators. The highly specialized hardware and low complexity algorithms assures predictable behavior, and consequently, an acceptable operational overhead cost. Messaging between components have more efficient protocols specific for time-sensitive and fault tolerant messaging [17].

G. Summary

Table I summarizes our previous arguments on properties applied to example DRT applications. The numbers represent the priority orders, where 1 is highest priority. Priorities with the same value are of equal importance or are correlated.

DVEs prioritize scalability, when compared to other architectural approaches to VE. In second place, time sensitivity and correctness are directly related with user experience and simulation results. Consistency is a minor concern, as long as eventually all distributed simulators converge to the same state.

GCA also prioritizes parallelism and scalability. Search engines need to serve thousands of users and queries, and look up millions of documents. A fault tolerant DRT solution provides the mean to scale, and thus is also ranked first. Next is time sensitivity, as users expect queries to be returned in a reasonable interval of time. Correctness is next, meaning the search engine favors returning query responses in time over spending more effort in attempting a better result. Consistency is not a major concern, since eventually consistent data will return correct results except if the query involve recent webpage modifications.

CAS is highly optimized to be correct. Fault tolerance, time sensitivity and consistency are all properties that lead to a better result. Scalability is not a concern, since no more devices are added at runtime. As long as the system performs under the specified requirements, there is no need to scale.

Naturally, overhead costs is last place in every application. Prioritizing any of the other 6 properties will often introduce overhead costs. As long as overhead costs are not interfering with the desired properties, they are acceptable. For example, carrying checksums with network packets increase bandwidth requirement, but improves fault tolerance and correctness.

IV. EVALUATION OF DRT SYSTEMS

In the previous section we focused on the impact of 6 properties on the design of DRT applications. After the design and implementation, the final step is evaluating if the DRT application behaves as intended.

Many of the properties described in Section III are hard to measure, as they are domain-specific and are often correlated with multiple hardware measurements such as CPU, memory,

and latency. Evaluations of DRT must account for multiple external factors, such as hardware, network, and operating system. Furthermore, many DRT applications cannot be tested in real-world conditions, thus simulations must be used to mimic real-world behavior.

This section discusses the issues surrounding evaluation of DRT systems. While we do not have enough information to analyze the evaluation of Google's GCA and automotive CAS, we have considerable experience with evaluating our own DSG-M. In this section, we describe the rationale behind the experiment and metrics of DSG-M, and conclude by generalizing tips and pitfalls of DRT evaluation.

A. Objective

The objective of DSG-M is to enable scalability through parallelism. Yet it is essential that the intended purpose of VEs remain within acceptable boundaries. Another way to frame DSG-M's objective is *to enable scalability through parallelism, while maintaining acceptable behavior for a virtual environment*.

In Section III-A. we have shown that correctness should be considered an interval of acceptable behavior. We argue that correctness in a VE is a product of two properties: user experience and simulation consistency.

The user experience describes how believable or immersive the virtual environment is to the user. We have previously discussed in Section III-D that interactivity, or time sensitivity, is highly correlated with the user experience. It is also known that the purpose of the VE influences user experience and the sensitivity to latency [19].

Simulation consistency is related to the expected outcome of the VE simulation. If the VE has physics, objects are expected to drop with an acceleration similar to gravity. Collisions are expected to conserve momentum. If an object moves through a wall, it is expected to be halted upon collision.

B. Experiment

Measuring user experience in VEs has many variables. To keep the experiment simple, and test the measurable non-functional properties of DSG-M, we chose to use a simulation consistency experiment. Simulation consistency is easier to measure, as it follows rules (e.g. physics) that can be deterministically verified. We decided our first experiment to be a physics-based experiment to measure expected simulation behavior. A user experience experiment is kept as future work.

Physics simulations have 3 properties that make it ideal for a precise evaluation. First, physics results can be compared to real-world results for correctness. Second, by not requiring users, tests can be performed thousands of times, guaranteeing statistically significant results. Third, since the objective is scalability, it is possible to increase load in a perfectly controlled manner.

For the physics experiment, we chose to simulate a device called *Galton box* [24]. Figure 1 shows both the original sketch of the device, and the simulated Galton box on our implementation of DSG-M. The Galton box is a board with multiple rows of equally spaced pegs. Each row from top to bottom adds and extra peg and is shifted, so that the each peg is exactly in the middle of the gap of pegs in the row above. At the bottom there are buckets covering the gap between each two pegs. The device works by dropping balls at the top of the box. Each time the ball falls on a peg, there is a 50% chance of it dropping to the right or to the left of the peg. If multiple balls are thrown in the same fashion, the buckets in the bottom will have a normal distribution of balls per bucket.

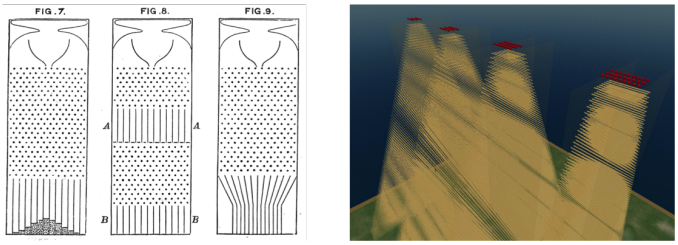

Fig. 1. On the left, the original drawing for a Galton box [24]. On the right, the simulated Galton boxes for the experiment. The boxes at the top drop the balls.

The Galton box experiment has many advantages for evaluating DSG-M. First, it is easy to determine the expected behavior. At the end of the simulation, the balls collected in the buckets should match the binomial distribution, represented by Equation 1. With a large enough number of balls, the distribution becomes normal. Multiplying Equation 1 by the total number of balls dropped at the device will result in the expected number of balls for a bin k.

Second, we can drop tens of thousands of balls, in order to obtain a statistically significant and repeatable result. Third, to add more load we simply need to the balls at faster rates. Finally, the normal distribution nature of the experiment allows us to test DSG-M under worst-case conditions. Most of the balls will be crossing near the middle of the device. If we partition the space so that the Galton box is divided in half, we expected very high overhead costs in migrating objects from a simulator to another.

To generate enough load to overwhelm a singe simulator process, we used 4 Galton boxes of $n = 93$ levels, with 27 droppers each (3 rows of 9). Droppers create balls at a experiment-defined period of *t* seconds per ball. By decreasing *t*, balls are generated faster and simulation load is increased.

Our modified Galton box has 3 rows of droppers, and each row will generate a binomial distribution with a different average. Figure 3 shows an example of the 3 distributions and their overlapping expected distribution with 37,771 dropped balls.

C. Metrics

The two properties we must control for is correctness, represented as ball distribution in buckets, and scalability, represented as the balance of load and performance, with and

$$\binom{n}{k}(0.5)^k(0.5)^{n-k} \tag{1}$$

Fig. 2. Where n: number of rows; k: bin number.

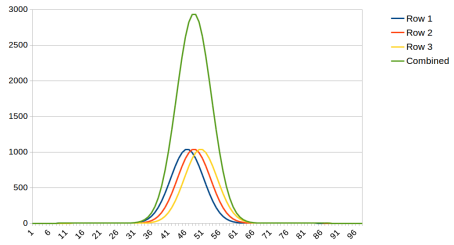

Fig. 3. Example distribution of 3 rows of droppers and the combined distribution for 37771 dropped balls. Notice that there are 93 levels, but 96 buckets, due to the overlapping of binomial distributions.

$$\sum_{b=1}^{96} \sqrt{\frac{(r_i - e_i)^2}{96}} \tag{2}$$

Fig. 4. Where b: bin number; r: number of balls measured in bin b; e and number of balls expected in bin b.

without space partitioning. We had previously presented results of DSG [7], so DSG-M was first compared to DSG. Comparing to a monolithic implementation of a VE is left for future work.

The first metric is ball distributon per bucket, the product of number of balls dropped and Equation 1. To compare the ideal result with our obtained results, we use root mean square error (RMSE) as a parameter. Equation 2 shows how RMSE is calculated for this experiment.

For the scalability metric, we first opted for CPU as a measure for performance, and number of balls in the Galton box as a measure of workload. Other metrics collected were number of messages exchanged for each simulator, number of messages in the queue to be sent, and network bandwidth. We had the following assumptions:

- The number of balls in the Galton box is the major cause of load, driving CPU usage.
- If CPU% is below 100%, the simulator is not overloaded. As more load is generated, CPU% increases linearly.
- Once CPU% reaches 100%, the simulator is overwhelmed and will slow down the simulation.
- If the simulator is overwhelmed, the relation between increased load and decreased simulation speed is linear.
- We can measure how much a simulation slows down by measuring the time interval between the ball creation at the top and collection at the bins. This interval in time should grown when simulation is slowed down.
- By dividing the Galton box in half, the CPU% load and memory usage would be reduced to nearly half, minus the overhead costs of migrating entities and state updates processing and messaging.
- Experiments are performed in a Local Area Network, mitigating inconsistency, latency, and bandwidth issues typically found in Wide Area Networks (WAN). Future experiments would cover behavior in a WAN through a network simulator.

The experiments consists of 37,800 balls being dropped on the 4 Galton boxes. Balls are created at a fixed period of t balls per second, and the experiment is repeated for different values of t. Any balls that do not fall within the boundaries of the bins are discarded from the results.

The expectation was that dividing the region space by half would enable simulation with a faster drop rate (i.e. higher load), and that CPU% would be perfectly correlated with the simulator increase in load.

D. Findings

After running several tests, the data suggested that our initial choice of metrics and some of our assumptions were naive. Figure 5 shows results for physics simulators for two experiments with the same period of ball generation $t = 6$ seconds (i.e 1 ball generated for every 6 seconds, for each dropper). One experiment was partitioned by operation (i.e. one physics simulator), but not by space. The second experiment was partitioned by both operation and space, with two physics simulators dividing the region in half. From observing these results, we evolved our initial set of assumptions to include the following lessons:

1. There are many independent variables in the experiments that are hard to control.

When overloading the unpartitioned simulator for our control experiment, we observed our first deviation from initial assumptions. Figure 5a shows that an overwhelmed simulator in DSG does not slow down physics linearly for a fixed period t. Object creation is done in the script simulator, and object deletion, on the physics. The script simulator is never overloaded on any experiment, and drops balls at a constant speed. The physics simulator eventually becomes overwhelmed with the number of objects, and slows down all physics operation, including object deletion.

The result is as seen in Figure 5b. Balls are being added faster than they are being deleted, and CPU resources are already exhausted. The result is an increase in the interval between creation and deletion, correlated with the increase in number of balls in the Galton box. The more balls to process, the longer the average interval is increased.

The second finding changed two other initial assumptions: (1) number of balls in the Galton box alone is a measure of load; and (2) the impact of network overhead costs. Figure 5c shows the number of balls in the Galton box throughout the experiments, for simulator 1 and 2. The number of balls on each simulator are stagnated, suggesting the effects of the first finding are not in place, and consequently, neither simulator is overloaded. Yet Figure 5d tells a different story. The interval between creation and deletion is steadily increasing, suggesting overwhelmed simulators. Which variables reflect the truth?

The culprit behind the contradicting metrics is network overhead cost. Figure 6 shows that messages intended for the physics simulator were backing up due to network exhaustion. The creation time for the ball was measured at the script simulator, while collection time was measured at the physics simulator. The growing queue size implies that new balls added to the Galton box took longer to arrive at the physics simulator. Since the physics simulator was not overwhelmed, the number

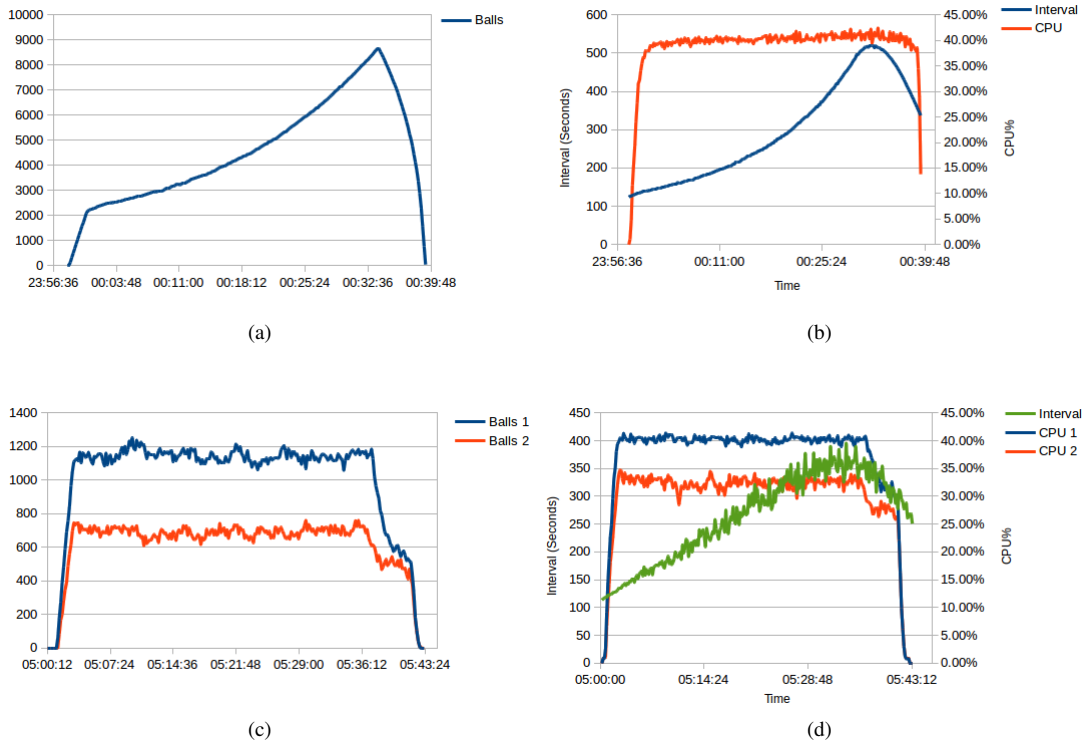

Fig. 5. Number of balls (left), average interval between creation and collection (right), and CPU usage (right) over experiment time interval. **(a) and (b)**: Experiment 1: one physics simulator; **(c) and (d)**: Experiment 2: two physics simulators (1 and 2), dividing region in half.

of balls in the physics simulators did not increase. But as the network got progressively worse, so did the mean time between creation and deletion.

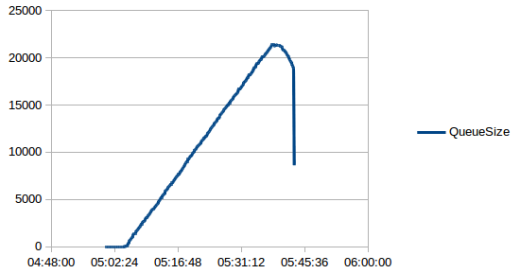

Fig. 6. Message queue size intended for physics simulator over time in partitioned experiment.

2. Hardware usage metrics are correlated with load, but are confounding metrics.

It is tempting to assume CPUs are a measure of processing load, but CPU usage as reported by the Operating System is an oversimplification. Modern CPUs are multicore, and the simulators are not heavily multithreaded. While some functionality is delegated to threads, the main simulation loop is mostly single threaded, to avoid inconsistencies. Physics runs on another separate thread, but is also not multithreaded. Hence, not all cores are used even when the simulator is overwhelmed, and as a result, the maximum CPU% is never achieved. In figures 5b and 5d, CPU usage tops at around 40

Another reason CPU is a confounding metric is that it does not take into account network stack computation. Our experiments involve tens of thousands of messages per second. Sending and receiving this large number of messages takes a toll in packing and unpacking messages through the network stack, but the operating system considers that effort to be system usage, and not part of the simulator CPU time.

Finally CPU usage was insufficient to measure behavior of an overwhelmed simulator. At maximum CPU, we can infer that the simulator is overwhelmed, but we cannot measure how much. By increasing load, the CPU will remain stationed at the maximum value. The effects of an overwhelmed simulator are only seen with application-specific metrics such as frames per second simulated.

The same argument can be made for other hardware usage metrics. While it is tempting to assume network bandwidth is a measure of network load, DSG-M sends very frequent packets containing little data. The network can become overloaded at a bandwidth that is many times lower than the maximum it could handle. Also only an application-specific metric (e.g. message queue size) can measure how overloaded the simulator is, since further increasing network load will result in the same maximum values of bandwidth and number of packets. All of the previous reasons favor use of real networks, as opposed to simulations, where such issues may not surface.

3. The baseline experiment did not match theoretical distribution.

The measure of correctness for our experiment was the normal distribution of balls in bins. Trial runs showed a distribution that resembled a normal curve, but further experiments

showed that the standard deviation of the distribution in the experiment was higher than in theory.

In order measure correctness when using DSG-M compared to DSG, we required a base scenario to compare to. Instead of using the theoretical values for the distribution of balls, we used the measured values on an non-space partitioned scenario with low load. We then reuse the average and standard deviation of this base scenario to compare other scenarios in the experiment, adjusting for the number of balls.

4. Our worst-case experiments caused a bug on .NET interpreter.

To demonstrate that our solution in DSG-M was generalized improvement, we aimed to show that the solution performs better even in face of worst-case conditions of load. When we ran experiments under very high load, using the highest amount of CPU, network, and memory the simulation could afford, we surfaced a bug outside of our environment. Our simulators run on C# with a Linux OS, using the Mono interpreter. After fixing bugs within DSG-M implementation, we found bugs related to Mono memory management that kept the simulator from successfully running the overwhelmed scenario successfully. By recompiling a recent mono with an extra flag and using the right garbage collector, the simulators ran smoothly.

V. Implications for Evaluation of DRT systems

The concerns raised by our experiments with DSG-M were initially discarded as domain-specific flaws. With careful examination in a post-mortem session, and in comparison to our experience with other DRT systems we developed and studied [3], we observed that many of the flaws are common with other DRT systems, but have not been presented as a general evaluation design concern. In this section, we discuss some root causes for evaluation issues.

1. Plan metrics to measure objectives, run experiment, then revisit metric choices.

DRT applications are complex and have several metrics that are distributed in nature. It is difficult to correctly plan the set of metrics that isolate and measure a desired behavior without running and collecting data from experimental runs. In DSG-M, the assumptions about traditional producer-consumer algorithms were out of place.

In DRT systems, expect metrics to return a mixed results of 3 aspects of the application: (1) an eventually consistent fraction of system load; (2) the effort of coordinating computation; and (3) high networking costs from distributed and real-time requirements, not accounted to the process.

2. Application-domain metrics are often more significant measurements than operating system metrics.

Related to the previous point, hardware metrics reported by operating system are a result of multiple effects. Instead of attempting to untangle the hardware metric (e.g. CPU and memory) to find the actual load used by the application, consider using application-domain metrics. In DSG-M, we use two application domain metrics to determine load: (1) number of balls being processed by the simulator; and (2) time interval between creation and collection of balls through the galton box. The first is used to isolate physics simulation load, and the second measures how overwhelmed is the simulator. Having both metrics enabled the detection of an inconsistency in Figures 5c and 5d.

3. Establish base values for correctness measurement.

In Section III, we discuss that correctness is a fuzzy concept, with domain-specific requirements. With all the source of randomness caused by DRT requirements, it is a priority to select which metrics define correctness, and what is the interval of values that are within acceptable range. In DSG-M, we initially ran sample Galton box experiments, noticed a normal distribution, and assumed that the distribution matched theoretical values. After investigation, we noticed that the standard simulator implementation did not achieve the theoretical distribution values that was expected. Thus we had to adapt correctness as being the results of the standard simulator under no load or partitioning. While not ideal, we are more interested in the difference of the end distribution when partitioning and load is added, rather than the theoretical distribution being achieved.

4. Distributed systems implies distributed metrics.

DRT solution have a major concern when collecting measurements in an evaluation: consolidating metrics from heterogeneous sources. Trivial preparations involve using the same hardware, operating system, and networking hardware on every machine running the DRT process. Yet even when the hardware and software setup is the same, symptoms may present in a different process from the cause.

In our DSG-M findings, Figure 5c and 5d showed contradictory metrics of load. Only by looking at Figure 6 were we able to comprehend the effects of the growing queue size. *Erroneous behavior in a DRT application can cause symptoms to be separated from the root cause within any process of the system.*

Another issue is the significance of consolidating metrics, which may cause two or more valid metrics to become invalid when merged. In DSG-M, when partitioning the region in half, we obtain CPU% usage of the two partitioned simulators. When presenting and analyzing the results, how is CPU usage presented? Combining CPU usage (i.e. adding or averaging) values hides the fact that one simulator had more load than the other. Another issue was measuring number of objects in the physics simulator, when objects were piled in the script simulator waiting to be sent, due to network resource exhaustion. *When presenting combined results or partitioned metrics (e.g. physics simulator only) for simplicity, caution is necessary that the summary does not mask an underlying trade-off.*

Finally, distributed metrics implies distributed collection. Evaluating DRT applications will likely result in several log files spread in different testing hardware. Either a network reporting mechanism is used to centralize data collection, experiment data collection scripts need to be prepared. Using network to centralize measurements needs to be used with care, as it can cause observer effects: the measurements can interfere with the application's network coordination. Using separated time logs implies in depending on physical time stamps, and

small deviations in seconds result in logs that are hard to merge.

One example in DSG-M was deviation of sleep intervals in C% under load. Measurements were taken every 10 seconds, and when simulators were overloaded, sleep functions missed deadlines by seconds, causing inconsistent timestamping of logs between processes. *Plan merging and collection of metrics to be distributed from the start.*

5. Stressing DRT systems will stress supporting software and hardware in novel ways.

Stressing an application will also stress the underlying libraries, framework, and operating system. In DRT applications, both processing and networking are stressed simultaneously, as they cannot be isolated. Hence there is increased chance of failure in supporting software and hardware.

Furthermore, DRT applications cause loads that are not typical, and consequently, not properly tested. In DSG-M, we had issues with Mono not properly handling object creation and deletion. Creating and deleting objects as quickly as the simulator was performing is not a common practice in applications. The source of this atypical behavior was object migration, a characteristic of distributed real-time systems. Monolithic real-time applications would not have to migrate objects, and distributed applications would not need to do it as quickly.

VI. Conclusion

We have presented in this paper a discussion on Distributed Real-Time (DRT) applications. We have shown that DRT applications are intrinsically difficult due to the controversial natures of distributed systems and real-time applications. Distributed systems require inter-process network communication, making the real-time property difficult to achieve due to unpredictable latency and jitter. Furthermore, the probability of fault becomes higher as more processes are part of the application, and real-time systems typically cannot afford totally consistent algorithms like phase commits, having to adhere to real-time fault-tolerant mechanisms.

We formulated DRT concerns in the form of 6 properties: correctness, fault tolerance, parallelism, interactivity, consistency and overhead costs. We applied these properties to 3 example DRT applications: distributed virtual environments, Google's cluster architecture, and collision avoidance systems. Finally, we discussed evaluation of DRT applications through the case study of our application, DSG-M. We have shown our findings, and generalized concerns to the evaluation of other DRT applications.

References

[1] S. J. Mullender, "Introduction to Distributed Systems," *CERN EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH - PROCEEDINGS*, pp. 29–46, 1993.

[2] D. Almroth, "Pikko Server," in *Erlang User Conference*, 2010.

[3] A. Valadares, T. Debeauvais, and C. V. Lopes, "Evolution of scalability with synchronized state in virtual environments," in *2012 IEEE International Workshop on Haptic Audio Visual Environments and Games (HAVE 2012) Proceedings*. IEEE, Oct. 2012, pp. 142–147.

[4] D. Brandt, "Scaling EVE Online, under the hood of the network layer," 2005.

[5] T. Blackman and J. Waldo, "Scalable Data Storage in Project Darkstar," Sun Microsystems, Inc., Mountain View, CA, USA, Tech. Rep., 2009.

[6] D. Horn, E. Cheslack-Postava, B. F. Mistree, T. Azim, J. Terrace, M. J. Freedman, and P. Levis, "To infinity and not beyond: Scaling communication in virtual worlds with Meru," Stanford University, Tech. Rep., 2010.

[7] D. Lake, M. Bowman, and H. Liu, "Distributed scene graph to enable thousands of interacting users in a virtual environment," in *Proceedings of the 9th Annual Workshop on Network and Systems Support for Games - NetGames '10*. IEEE Computer Society, Nov. 2010, pp. 1–6.

[8] Linden Labs, "Second Life."

[9] J. Kaplan and N. Yankelovich, "Open Wonderland: An Extensible Virtual World Architecture," *IEEE Internet Computing*, vol. 15, no. 5, pp. 38–45, Sep. 2011.

[10] F. Aurenhammer, "Voronoi Diagrams - a Survey of a Fundamental Geometric Data Structure," *ACM Comput. Surv.*, vol. 23, no. 3, pp. 345–405, Sep. 1991.

[11] C. Carlsson and O. Hagsand, "DIVE A multi-user virtual reality system," *Proceedings of IEEE Virtual Reality Annual International Symposium - VRAIS '93*, pp. 394–400, 1993.

[12] R. C. Waters, D. B. Anderson, J. W. Barrus, D. C. Brogan, M. A. Casey, S. G. McKeown, T. Nitta, I. B. Sterns, and W. S. Yerazunis, "Diamond park and spline: A social virtual reality system with 3D animation, spoken interaction, and runtime modifiability," *Presence: Teleoperators and Virtual Environments*, vol. 6, no. 4, pp. 461–480, 1997.

[13] H. Liu and M. Bowman, "Scale Virtual Worlds through Dynamic Load Balancing," *2010 IEEE/ACM 14th International Symposium on Distributed Simulation and Real Time Applications*, pp. 43–52, Oct. 2010.

[14] B. De Vleeschauwer, B. Van Den Bossche, T. Verdickt, F. De Turck, B. Dhoedt, and P. Demeester, "Dynamic microcell assignment for massively multiplayer online gaming," in *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games - NetGames '05*. New York, New York, USA: ACM Press, 2005, pp. 1–7.

[15] L. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The Google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, 2003.

[16] J. Axelsson, J. Fröberg, and H. Hansson, "A Comparative Case Study of Distributed Network Architectures for Different Automotive Applications." Tech. Rep. 1, 2003.

[17] F. Gustafsson, "Automotive safety systems," *IEEE Signal Processing Magazine*, vol. 26, no. 4, pp. 32–47, Jul. 2009.

[18] P. Quax, P. Monsieurs, W. Lamotte, D. De 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 *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*, ser. NetGames '04. New York, NY, USA: ACM, 2004, pp. 152–156.

[19] M. Dick, O. Wellnitz, and L. Wolf, "Analysis of factors affecting players' performance and perception in multiplayer games," in *NetGames'05*. ACM, 2005, pp. 1–7.

[20] D. Burton, A. Delaney, S. Newstead, D. Logan, and B. Fildes, "Effectiveness of ABS and Vehicle Stability," Tech. Rep. April, 2004.

[21] S. Winkler and P. Mohandas, "The Evolution of Video Quality Measurement: From PSNR to Hybrid Metrics," *Broadcasting, IEEE Transactions on*, vol. 54, no. 3, pp. 660–668, Sep. 2008.

[22] R. Isermann, R. Schwarz, and S. Stolzl, "Fault-tolerant drive-by-wire systems," *IEEE Control Systems Magazine*, vol. 22, no. 5, pp. 64–81, Oct. 2002.

[23] E. A. Brewer, "Towards robust distributed systems," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*, ser. PODC '00, vol. 19. New York, New York, USA: ACM Press, 2000, p. 7.

[24] F. Galton, *Natural inheritance*. Macmillan, 1889, vol. 42.