UNIVERSITY OF CALIFORNIA,
IRVINE

Aspect-Oriented Architectural Style for Distributed Interactive Simulations

THESIS

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Informatics

by

Arthur Rodrigo Sawazachi Valadares

Thesis Committee:
Professor Cristina V. Lopes, Chair
Chancellor’s Professor Emeritus Richard N. Taylor
Researcher Mic Bowman

2016
## TABLE OF CONTENTS

**LIST OF FIGURES**

vi

**LIST OF TABLES**
viii

**LIST OF ALGORITHMS**
ix

**ACKNOWLEDGMENTS**

x

**ABSTRACT OF THE THESIS**

xi

1 Introduction

1.1 Virtual Environments .................................................. 5
1.2 Distributed Simulations ............................................... 7
1.3 Designing Scalable DIS Architectures ................................ 8
  1.3.1 Scaling size .................................................. 9
  1.3.2 Scaling scope ............................................... 10
1.4 Approach .............................................................. 12
1.5 Thesis Statement ...................................................... 15

2 Background on Distributed Interactive Simulations

2.1 Object-Partitioned Architectures ...................................... 18
  2.1.1 SIMNET / DIS ................................................ 18
  2.1.2 MR Toolkit .................................................. 19
  2.1.3 BrickNet ..................................................... 19
  2.1.4 RING .......................................................... 20
  2.1.5 DIVE .......................................................... 20
  2.1.6 MASSIVE ..................................................... 21
  2.1.7 Spline ........................................................ 22
  2.1.8 Second Life / OpenSimulator .................................... 23
  2.1.9 Meru ........................................................... 24
  2.1.10 Kiwano ...................................................... 25
2.2 Service Partitioned Architectures ..................................... 25
2.3 Generalist Architectures ............................................... 26
  2.3.1 Open Wonderland .............................................. 27
  2.3.2 High Level Architecture (HLA) .................................. 27
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Logical view of RCAT.</td>
<td>44</td>
</tr>
<tr>
<td>3.2</td>
<td>RCAT Style Architectural Model.</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Network Model of RCAT</td>
<td>61</td>
</tr>
<tr>
<td>3.4</td>
<td>Jigsaw game client view</td>
<td>65</td>
</tr>
<tr>
<td>3.5</td>
<td>Experiment Client View</td>
<td>77</td>
</tr>
<tr>
<td>3.6</td>
<td>Client to proxy bandwidth (as measured from scenario PS). ©2012 IEEE</td>
<td>80</td>
</tr>
<tr>
<td>3.7</td>
<td>Round-trip time for various configurations. ©2012 IEEE</td>
<td>81</td>
</tr>
<tr>
<td>3.8</td>
<td>Bandwidth between database and server without cache (PSD) and with cache (PScD). ©2012 IEEE</td>
<td>82</td>
</tr>
<tr>
<td>3.9</td>
<td>Two maps of the same 256 m² OpenSimulator region. The left figure shows an unmapped region. The right figure shows the region divided by microcells of size 32x32 m², mapping to Cartesian coordinates (0 – 7, 0 – 7).</td>
<td>87</td>
</tr>
<tr>
<td>3.10</td>
<td>An example of a star topology with 3 non-root actors, and 1 root actor. The region is mapped as 4x4 microcells, the circle and triangles represent entities that were updated.</td>
<td>90</td>
</tr>
<tr>
<td>3.11</td>
<td>An example to demonstrate the 3 entity crossing situations. In the example, actor $A_0$ originates the crossing of the triangle, circle, and diamond entities, sending to actor $A_i$.</td>
<td>91</td>
</tr>
<tr>
<td>3.12</td>
<td>On the left, the original drawing for a galton box [68]. On the right, the simulated galton boxes for the experiment. The boxes at the top drop the balls.</td>
<td>92</td>
</tr>
<tr>
<td>3.13</td>
<td>Topology and subscriptions of actors in the first part of the experiment, and in Scenario 1 of the second part. Each horizontal group of bars represent a galton box. Microcell grid not to scale.</td>
<td>95</td>
</tr>
<tr>
<td>3.14</td>
<td>Topology and subscriptions of actors in Scenarios 2 and 3 of the second part of the experiment. Each horizontal group of bars represent a galton box. Microcell grid not to scale.</td>
<td>98</td>
</tr>
<tr>
<td>3.15</td>
<td>Expected and resulting distributions for Scenario 1 (no partitioning), Scenario 2 (active subscriptions only), and Scenario 3 (passive subscriptions included).</td>
<td>99</td>
</tr>
<tr>
<td>4.1</td>
<td>An architectural view of CADIS.</td>
<td>107</td>
</tr>
<tr>
<td>5.1</td>
<td>Total processing time (push, pull, and update) for producer and consumer in PCC set type scenario.</td>
<td>132</td>
</tr>
<tr>
<td>5.2</td>
<td>Total processing time (push, pull, and update) for producer and consumer in subset scenario.</td>
<td>133</td>
</tr>
</tbody>
</table>
5.3 Total processing time (push, pull, and update) for producer and consumer in projection scenario. ................................................................. 134
5.4 Total processing time (push, pull, and update) for producer and consumer in join scenario. ................................................................. 136
5.5 Total processing time (push, pull, and update) for producer and consumer in update scenario. ................................................................. 136
5.6 Total processing time (push, pull, and update) for producer and consumer in creation scenario. ................................................................. 137
5.7 Logical organization of mobdat in CADIS. .................................................. 141
5.8 Performance comparison between the original event-based mobdat with the CADIS version. The y-axis is processing time in milliseconds for each clock event. The x-axis is minutes of wall-clock experiment time. .................. 143
5.9 Logical organization of the usability study. .................................................. 147

6.1 Example distribution of 3 rows of droppers and the combined distribution for 37800 dropped balls. Notice that there are 93 levels, but 96 buckets, due to the overlapping of binomial distributions. .................................................. 170
6.3 Message queue size intended for physics simulator over time in partitioned experiment. ................................................................. 177
6.4 Three stages of server performance load during a high-load user login. 1: CPU load spike after login; 2: Load during inventory and scene; 3: return to steady-state levels after login complete. .................................................. 179
6.6 Simulation server privileged processor time, comparing light inventory (left) and heavy inventory (right) configurations. .................................................. 183
6.7 Simulation server privileged processor time with a dedicated simulation server for inventory, comparing light inventory (left) and heavy inventory (right) configurations. .................................................. 184
LIST OF TABLES

3.1 Part 1: DSG without partitioning, load measurements. **AL**: Average load (CPU%), **PA**: Physics actor, **RA**: Root Actor, **FD**: Average fall duration of balls in seconds, **AB**: Average number of balls on the galton box, **RMSE**: Root-Mean error. .................................................. 96

3.2 Part 2: DSG-M with space partitioning, at a period of 6 seconds per ball. **AL**: Average load (CPU%), **M**: Process memory in megabytes, **PA1 and PA2**: Physics actor 1 and 2, **RA**: Root actor, **FD**: Average fall duration of balls in seconds, **AB**: Sum of PA1 and PA2 average number of balls, **RMSE**: Root-Mean error. .................................................. 98

5.1 Processing time in milliseconds for each version of mobdat execution in the experiment. **E.B.**: Event-based. .................................................. 142

5.2 Processing time in milliseconds for push, pull, and update on each of the CADIS versions. **C**: CADIS, **O**: Optimized, **R**: Remote, **OS**: OpenSimConnector, **Sumo**: SumoConnector, **Social**: SocialConnector, **updt**: update. . . 144

6.1 Login Experiment Configurations .................................................. 180
## LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th></th>
<th>Algorithm Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Retrieve objects within an area</td>
<td>58</td>
</tr>
<tr>
<td>2</td>
<td>Retrieve object data based on its position and id</td>
<td>60</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

There are many who have played vital roles in the completion of this dissertation. I start by thanking all friends, family, and peers for the support and feedback provided in these long but wonderful years of my PhD studies. In particular, I have would like to acknowledge my advisor Cristina Lopes, who has supported me academically, emotionally, and financially during my studies. No matter how complicated any situation looked (and there were a few of those), we always found some way to solve it by thinking out of the box. Also, I would to thank her for the patience in reviewing every single one of my publications, and for keeping my standards high. The journey was rough, but it is clear how much I learned and improved in these past years, thanks to being constantly challenged by my advisor.

I am also grateful for all the support my parents have provided me since college, both financially and emotionally. The only reason I could focus on my work is because I knew I had their support for any eventuality. I would also like to thank my girlfriend, Paula Moura, who has provided the utmost support at our home, reducing my responsibilities and giving me the chance to focus even more on the work in front of me.

Special thanks to all my collaborators: Cristina Lopes, Rohan Achar, Eugenia Gabri洛va, and Thomas Debeauvais, who have contributed and co-authored several parts of the published papers used in this thesis, and provided valuable feedback. In particular, Thomas and Rohan have been vital peer contributors of many portions of the work in this dissertation.

IEEE granted me permission to include portions of my works published in the International Games Innovation Conference (IGIC) 2012, the Winter Simulation Conference (WinterSim) 2014, and the Winter Simulation Conference (WinterSim) 2016. John Wiley and Sons granted me permission to include the article On designing and testing distributed virtual environments published in the journal Concurrency and Computation in its entirety. CRC Press granted me permission to include my published chapter from the book Software Engineering and Computer Games.
ABSTRACT OF THE THESIS

Aspect-Oriented Architectural Style for Distributed Interactive Simulations

By

Arthur Rodrigo Sawazachi Valadares

Doctor of Philosophy in Informatics

University of California, Irvine, 2016

Professor Cristina V. Lopes, Chair

Simulations are powerful computational tools to comprehend complex systems in a large number of scientific domains. Even with the advance of computational power, the capability to model the real-world in full detail is still far in the future. But at every step forward in computational power, more can be done with discrete simulations to predict and explain physical systems in domains where continuous mathematical representations are insufficient or unknown. The field of distributed simulations was born to leverage simulation capabilities past what a singular computer can perform. Furthermore, distributed simulation frameworks enables a collaborative development environment, where multiple developers can work in independent parts of the same problem – similarly to large software problems being solved by the collaborative effort of software developers working on semi-independent modules. Unfortunately, like in most distributed system applications, distributed simulations are very complex to design and develop. While there are supporting frameworks, like the High Level Architecture (HLA), there is room for improvement in reducing the technical knowledge required to develop for such frameworks.

The goal of this dissertation is to expand the expressiveness of distributed simulation frameworks while reducing the technical complexity to develop integrated distributed simulations independently. This dissertation presents a novel architecture for distributed interactive
simulations (DIS) called Collaborative Aspect-oriented DIS (CADIS). The architecture is divided in two frameworks called PCC and spacetime. PCC (Predicate Collection Classes) is an object-oriented programming model for representing collections of objects. These collections of objects are called relational data types – types that are reclassified based on runtime values. Relational types allows the expressiveness of queries that are common to relational databases to be defined as abstract data types. Spacetime adds automatic synchronization and recalculation of PCC data types in discrete time distributed simulations. Through a simple push, update, and pull process, simulations operate with newly reclassified objects, and any modifications are automatically pushed to the server.

CADIS is evaluated in four different ways. First, a benchmark measures the performance impact of each individual relational operation, demonstrating the many performance improvements in the implementation required to reduce the data exchange between simulation and servers to an acceptable level. Second, a feasibility evaluation using a realistic urban simulation scenario is presented, demonstrating effectiveness in a real-world problem. Third, a case study using CADIS in a graduate course is presented, demonstrating how CADIS can be taught to developers with little simulation background in a small period of time. Finally, a parallel between HLA and CADIS is presented, by analyzing a converted urban simulation event routing system to HLA.

The architecture of CADIS provides two major contributions to the field of distributed simulations: a) it allows the development effort to be partitioned and split among independent groups of developers; b) thanks to relational data types in PCC, simulations can join at runtime with new data models without affecting existing simulations. Furthermore, existing simulations can handle and use new data models that are related to known ones. The larger impact of CADIS is creating a scalable service-oriented development environment, where simulations are providers of data types with specific features, allowing developers to add complexity to data models based off previous implementations.
Chapter 1

Introduction

One of the most powerful scientific approaches to comprehending complex systems has been the use of computer simulations. Simulations are a powerful tool for predicting results of theoretical scenarios that may be impractical or impossible to test in the real world. Simulations are used in a large variety of domains, from fundamental sciences like biology [51, 58, 94, 174] and physics [15, 18, 103, 177], to more recent domains like supply-chain [84, 97, 105, 181], traffic [30, 59, 74, 98, 106, 135, 146, 162], socioeconomics [5, 41, 117, 172], and education [9, 26, 131, 184, 198].

With the fast advancement of computational power in the last decades, simulations have grown larger and become more complex. Despite the increase of individual microprocessor and memory resources, the most significant approach to increasing computational power to match large simulations is through horizontal scalability – hundreds to thousands of interconnected processors working in parallel on the same computational problem. However, simulations are heavily dependent on data (i.e. state), which presents the challenge of maintaining consistent state across multiple computational nodes, specially when interconnected by a network. Distributed systems that support networked connected simulations are called
distributed simulation systems.

Ideally, simulations can advance time as fast as computationally possible – days or years in simulation time can be computed in only hours of wallclock (i.e. real-world) time. Some simulations, however, have dependencies to the real-world that enforces a synchronization of simulation and wallclock time. Because these dependencies are time sensitive, this class of simulations have real-time requirements: state updates must be processed within a deadline, or be discarded. Two common situations where simulations have real-time requirements are human-in-the-loop, when a user is interfacing with the simulation, and when using data from sensors or generating output to actuators. Partitioning and distributing simulations that have real-time requirements proves to be even greater challenge, as state synchronization between nodes must not only be consistent, but also highly available. This class of simulation systems are known as distributed interactive simulations (DIS).

DIS systems have been used primarily for simulating virtual environments. A virtual environment simulates the appearance and physical laws of a real-world environment, allowing a user to be immersed in a virtualization of the real-world. Virtual environments allow users to experience being at a real-world location without incurring the costs and risks of travel. Furthermore, a virtual environment can be adjusted to create scenarios that would be impossible in real-life, like modifying natural laws or accelerating the passage of time. Some of the applications include training [40, 82, 101, 126], education [22, 90, 189], entertainment [77, 92, 111], and studying natural phenomena [8]. Different applications of DIS have different sets of software requirements, which typically involve trade-offs between high availability, consistent state synchronization, frequent updates, high fidelity models, and scalability of number of users and simulated objects in-world.

This dissertation covers the many facets of designing and evaluating DIS systems by presenting an extensive background in DIS architectures and by presenting design experiments of DIS architectures focused on improving scalability. The topic of the dissertation focuses
on a particular kind of scalability problem: the ability to support a multitude of relatively independent aspects of simulations. *The major contribution of this work is a new architecture and programming model that support independent collaborative development – groups of developers that hold distinct domain expertise collaborating towards one large simulation through the integration of smaller parts.*

An example scenario – used later in the dissertation – is the development of an urban simulation. Simulating cities is, unsurprisingly, a complex task, as cities have many different aspects – water and electricity distribution, traffic management, land zoning, and health, to name a few. The domains of expertise are so diverse that a city typically divides these tasks in departments, that are commonly detached in both geographical and organizational dimensions. Many of these departments make use of simulations to improve city policies. Traffic departments use traffic simulations to reach decisions on road closures, lane expansions, and building roads and highways. Water and electrical distribution uses simulations to find efficient ways to improve transportation and distribution networks. Other departments use simulation for similar purposes – understanding existing infrastructural and policy problems and implementing effective solutions.

Evidently, each aspect of the city is not isolated from the others. Changes in land zoning affect traffic, changes in traffic affect health concerns such as air pollution, which in turn may affect other aspects of city planning. However, simulations of these urban aspects often act in isolation: each group of domain experts build highly complex and accurate models of their own area of expertise, while simplifying or using collected data to model behavior from others. The result are simulations that have no feedback mechanism, possibly producing inaccurate or incorrect predictions of the impacts of adopting city policies. *Such problems are the motivation for the collaborative architecture proposed in this dissertation.*

There is only one widely used standard that actively supports independent development of simulations: the High Level Architecture (HLA), created by the Department of Defense
(DoD) to integrate simulations from military contractors. HLA is used to integrate independently developed simulations successfully in the military domain, but is widely considered too technically complex by industry simulation practitioners [19–21]. That is not a fault in the HLA standard, but rather in the lack of a programming model and abstraction layer that allows one simulation to acknowledge entities from other simulations in a modular way – without the knowledge of the intrinsic design of other simulations. There have been multiple architectural solutions for HLA that add an abstraction layer, effectively reducing the complexity of handling simulation distribution.

My approach is distinct from other solutions in that it improves the way abstract data types are defined, allowing simulation models to be mapped to types in a modular way. It is the objective of this dissertation to show that our proposed architecture supports the modular development of simulations that integrate a larger DIS ecosystem, being modular enough that only the name and properties of external entities are necessary for any required inter-simulation interaction.

This introductory chapter explores the domains of DIS systems and architectures, introduces the design experiments in DIS, and presents a summary of the my collaborative simulation architecture. Sections 1.1 and 1.2 will give a brief introduction to the fields of virtual environments and distributed simulations, describing the chronological evolution of DIS system design in research, military, and industry. Section 1.3.1 describes the approaches used in DIS to partition state across multiple nodes. Section 1.3 briefly presents 2 previously designed DIS architectures that are focused on scalability. Sections 1.3.2 highlights remaining concerns in the design of the previous DIS architectures, and proposes the Collaborative Aspect-oriented DIS (CADIS), an architectural approach to designing collaborative simulations, and topic of this dissertation.
1.1 Virtual Environments

Virtual environment architectures have gone through many changes since first incepted in Sutherland’s ”The Ultimate Display” [166]. The overwhelming demand of bandwidth and memory required for executing such systems left the field mostly theoretical until the 90’s, when Personal Computers and network bandwidth were more readily available to the general public. Foundational papers like DIVE [33] and Diamond Park [187] presented groundwork solutions geared towards a viable shared user experience in Distributed Virtual Environment (DVE).

In a typical client-server architectural style, each client sends its actions or position to the server between 5 to 30 times per second depending on the level of interactivity of the environment. As the number of client increases, a bottleneck quickly arises in monolithic architectures (i.e. single-tiered, tightly-coupled components and connectors), as they have to handle more data more frequently while not being able to divide their load due to the stateful nature of the client requests. As CPU, bandwidth or RAM become scarce, it becomes necessary to split bandwidth-demanding components from CPU-demanding components in tiered architectures [2, 3] in which each tier (or layer) is responsible for separate demanding components.

As these applications become more complex, responsiveness and consistency become affected by overwhelmed servers that can no longer respond to the demands of new user requests in a timely manner. Slower responsiveness and occasional inconsistencies damage user immersion on a virtual environment or game, reducing the perceived quality of experience of the application. To provide a good quality of experience (interactivity, responsiveness, and consistency [36]) with a massive number of users, designers impose new constraints that limit the freedom in the application design or the user experience in the environment.

In DIVE, the concept of spatial partitioning is presented in the form of a virtual environment
architecture. Under the assumption that only the "visible world" (i.e. human viewing distance) of clients is important, DIVE divides the world into separate enclosed spaces and delegates their responsibilities to different servers across the network. This constraint greatly reduces the resource strain caused by the broadcasting nature of these applications, but comes with a cost: the assumption of a uniformly divided world in order to scale. In practice, an uneven object and avatar distribution is most commonly observed in virtual environments [89]. Additionally, the worst case scenario of clients interacting in region borders also creates a disruptive user experience and high server load. Hence spatial partitioning does not always suit the load induced by real-life groups of players [35].

Most current DVEs can host up to a few hundred users interacting with each other. However, scenarios such as stadium matches, concerts or conferences may involve thousands of concurrent users potentially all interacting with each other in real-time. Users should not be able to notice that their actions take time to be processed by the system, whether it is a small- or large-scale deployment.

In software engineering, essential problems are problems that cannot be solved by simply relying on Moore’s law or inventing new language paradigms [28]. They are, in essence, hard, and optimizations cannot solve them. In the context of DVEs, computing interactions between users and sending updates to clients is an essential problem, as the CPU and bandwidth required increase quadratically with the number of users. Optimizations such as interest management or bucket synchronization [71] can reduce the amount of bandwidth required between the clients and the communication layer of the system. While such optimizations can be highly effective in current DVE scenarios (small team encounters in first-person shooter games, solo-questing in role-playing games), they do not tackle the essence of the scalability problem, and sometimes even limit the interactions between users [32].
1.2 Distributed Simulations

Distributed simulation systems evolved independently and in parallel to the previous virtual environment systems. The first successful standard for distributed simulations was SIMNET (Miller and Thorpe 1995), funded by the DoD and developed by DARPA. With SIMNET, it became possible to link hundreds of simulators to produce a virtual world, used for real-time, man-in-the-loop, coordination and tactic simulations. SIMNET eventually evolved into the Distributed Interactive Simulation (DIS) standard, becoming an IEEE standard (IEEE 1278-1993) (DIS Steering Committee 1998).

With the participation of industry and academia partners, the conceptual model of DIS evolved into the High Level Architecture (HLA) [87]. The HLA is, as the name implies, higher level abstraction when compared to DIS. In DIS, most of the simulation design decisions, such as networking protocols, are fixed as part of the standard. The HLA is concerned only with higher design decisions, such as the protocols for publishing an subscribing to events and managing simulation time advancement. HLA defines two separate entities: the federates and the runtime infrastructure (RTI). The RTI is the communication bus for simulators to share events and updates. It is also used to specify and share the object model template (OMT), a shared specification of what objects are available to be instatiated. New objects can be defined and shared between simulators.

The federates are user simulators that – together with the RTI – form the federation. The API from the HLA defines how federates should share object models, updates, events, and interactions. The RTI has many commercial implementations, including VT Mak RTI and Pitch pRTI. Two well known open-source implementations are OpenHLA and Portico. Simulators can be adapted to the HLA protocol through what is called an HLA ambassador: a software module that translates events from the RTI to the simulator, and interprets internal events and routes back to RTI.
1.3 Designing Scalable DIS Architectures

After decades of virtual environment and distributed simulation development we are still nowhere near scaling simulations to the life-like complexity of the real world. This task requires improving scalability in two different aspects: by size and scope. Scaling the simulation size is creating a simulation with larger number of simulated entities and users. For instance, simulating the experience of a stadium or a concert is still far from what is currently achievable – current records of virtual environment participants are still in the thousands, whereas stadiums and concerts have tens of thousands of users.

Scaling the simulation scope is creating simulation models that are just as complex as the physical systems. Most simulation models use abstractions to reduce complexity and focus on a particular aspect of the simulation. A traffic simulation, for example, creates and destroys the simulated vehicles. In the real-world, vehicles are manufactured, owned by humans, and driven from a source to a destination for a social reason, such as work, shopping, or leisure. While abstracting such complexities often offers good enough results, there’s a real possibility that the observed results of a simplified simulation may not reflect reality. An example in the traffic simulation scenario is weather – snowy roads lead to vehicles driving slower, keeping larger distances from each other, and increasing the chances of accidents. The scope of a simulation model is nearly infinite – down to the physics that rules atoms – but a good DIS system should scale in scope as it becomes necessary to improve simulation fidelity.

This dissertation is focused in studying scalability of DIS systems. In particular, we study software architectures that support scalable DIS systems. This section details the challenges of scaling simulations by size and by scope.
1.3.1 Scaling size

DIS systems have contrasting requirements that are difficult to satisfy. According to the CAP theorem [27], it is not possible for a partitioned system to be both available and consistent, so scaling up a DIS also implies maintaining an acceptable level of availability and consistency. There are three major concerns to scaling a DIS in size: a) the partitioning of users and entities across the nodes, b) propagating state updates to users, and c) maintaining synchronized state.

(a) **Partitioning:** The most ubiquitous approach to scale DIS in size is to partition the simulation in ways that require less communication between nodes. DIS architectures may be designed to enforce how simulations partition users and entities, or leave the partitioning decision to the simulation developer. A DIS architecture that does not enforce a partitioning method is a generalist architecture. Architectures that implicitly define partitioning methods in DIS are of two types: object partitioning and service partitioning. Although partitioning is a vital part of scaling DIS, it is often a design decision specific to each simulation. There is a large body of research in the topic of partitioning (see Chapter 2), but in this dissertation we focus on the remaining two concerns: user update propagation and state synchronization.

(b) **Update propagation:** A common concern in DVEs, propagating state updates to clients is often referred to as the $O(N^2)$ problem. If each client of a virtual environment with $N$ users sends an update, $N^2$ messages are generated, since each user must receive state updates of others. This mechanic hinders scalability of the number of participating users, since the number of messages sent by a DIS server increases quadratically. Addressing distribution of data to users while handling data partitioning across multiple servers is still an open problem with several proposed solutions in the gaming industry [6, 92], virtual environments [47, 86, 92, 102, 111, 138], and in
distributed simulations [87, 170, 195].

(c) **State synchronization:** Maintaining synchronized consistent state between simulation nodes is a difficult task when simulations are expected to process updates frequently. The simplest solution is pushing this concern to a centralized database or a distributed database, for better scalability. However, both solutions have a penalty in performance: the centralized database does not scale with the number of users and simulated entities, and a distributed database generally cannot guarantee consistency between nodes in the short interval updates required for interactive simulations. Thus, many DIS platforms have tailored specialized algorithms and frameworks to maintain state synchronization between nodes in a way that is consistent and available enough for most real-time simulations [6, 47, 86, 87, 138, 170, 195].

### 1.3.2 Scaling scope

Despite decades of research, there is little consensus in how DIS should be designed, deployed, evaluated, and tested. In particular, efforts of scaling DIS in scope through integration of independently developed simulations have been neglected in lieu of incompatible commercial proprietary simulation frameworks. Since computational resources – such as computing power and network latency – are scarce, many of the architectural solutions for DIS are instead focused on scaling in size. The exception is the HLA standard. HLA was a successor of previously existing standards developed by the military, with the focus of allowing independent contractors to create simulations that may be integrated to demonstrate a realistic combat scenario. Although HLA does not constraint its use for military domain, it is widely regarded as too difficult to adopt by practitioners in the simulation community [19–21]. While military contractors have become highly specialized in developing with HLA, the rest of the simulation community are left with no good models for distributed simulations that is practical for development – with many claiming they see no need for one [19].
This raises the question: what benefits do architectural models of distributed simulation brings? There are two large benefits in using distributed simulation standards. The first – the one previously discussed in this chapter – is scalability. The second reason, and the purpose of this dissertation, is to support independent development and simulation reuse. Since there are no practical universal standards for developing simulations, practitioners are subject to using a plethora of proprietary commercial off the shelf (COTS) solutions that provide no support for integrating other existing simulations. While in the software domain importing libraries and reusing existing applications are common practice, the simulation domain becomes subject to writing existing simulations repeatedly due to a lack of compatibility.

The HLA standard provides a powerful solution to address the compatibility problem, by accurately defining every aspect of a distributed simulation that is required for inter-simulation communication. The most recent version of HLA \[87\] went further and provided a much needed object management service, that allows developers to use object typing for modeling entities. However, objects in HLA are not the same as objects in Object-Oriented Programming (OOP). HLA objects are simply data structures containing properties that can be published or subscribed to. This differs from OOP objects mainly for two reasons: first, HLA objects have no methods; second, HLA objects are not constructable – they are intrinsically a dictionary of property handles to values, used as reference for publication and subscription of properties. HLA fulfills a low-level layer of abstraction, handling synchronization and publish-subscribe requests of properties in data structures. In contrast, a high-level abstraction maps the state updates to OOP objects. Such mapping has been proposed by independent researchers as an abstraction layer on top of HLA \[73, 168, 175\]. However, these approaches were limited in providing the same level of functionality found in OOP languages. Our focus is to improve the programming language paradigm used in DIS to better fit the requirements of independent development of collaborative simulations.
1.4 Approach

This dissertation presents architectural approaches to designing scalable DIS in size (Chapter 3) and in scope (Chapter 4). Scalability of size is tackled as a two-part design experiment, addressing respectively the issues of update propagation and state synchronization. The first experiment, called Restful Client-server Architecture (RCAT), is a REST-based DIS architecture that relies on the deployment of proxies to scale up the number of participating users, mitigating the $O(N^2)$ problem. The second experiment, the Distributed Scene Graph with Microcells (DSG-M), is an improvement of the DSG virtual environment architecture. DSG is a service-partitioned architecture, using replication state synchronization mechanism. DSG-M offers both space and service partitioning while maintaining the highly consistent and available state synchronization of DSG.

Despite the success of both design experiments in improving scalability in size, both approaches proved to be difficult to scale in scope. The topic of this thesis is supporting scalable scope of DIS systems through a novel architecture called Collaborative Aspect-oriented DIS (CADIS). CADIS provides architectural support for collaborative simulations with transparent state synchronization of OOP objects. CADIS has a data-oriented philosophy that encourages all inter-simulation communication be performed through objects, as opposed to the use of events in traditional simulations. What makes CADIS unique from other DIS architectures and frameworks is the unique feature of mapping collections of objects to abstract data types, and the subsequent use of relational algebra to define new collection types based on existing collection types.

To illustrate this feature, I refer back to the urban simulation scenario – discussed in the introduction – with two urban simulation aspects: a traffic simulation that simulates vehicles, and a pedestrian simulation that simulates pedestrians. In traditional OOP, one would expect the class definitions in Listing 1.1. Assume now that the pedestrian simulation wants
to avoid collision (i.e. being run over by a car). The pedestrian simulation would have code to detect a collision situation and resolve it, similar to the code in Listing 1.2.

```python
class Pedestrian:
    @property
def Position(self): return self.position

def Velocity(self): return self.velocity

class Car:
    @property
def Position(self): return self.position

def Velocity(self): return self.velocity
```

Listing 1.1: Example type definitions for Car and Pedestrian in an urban simulation.

```python
class PedestrianSimulation:
    def update_loop(self):
        ( ... )
        for car in self.all_cars:
            for pedestrian in self.all_pedestrians:
                if abs(car.Position - pedestrian.Position) < 0.1:
                    self.pedestrian.move_to_safety()
        ( ... )
```

Listing 1.2: A pedestrian simulation that detects a future collision between car and pedestrian and moves pedestrian to safety.

There are 3 particular issues with the code in Listing 1.2. The first problem is that Lines 4-7 are not intuitively easy to understand, requiring close inspection (or well written comments) of the algorithm to understand its purpose. The second issue is this information might be useful to other simulations, and yet this calculation was made and lost – only the updates to the pedestrian will be shared by the DIS architecture. Finally, the third issue is type bloating.

The developer of PedestrianSimulation is likely responsible for the `move_to_safety()` method to save pedestrians in danger. If the Pedestrian class definition is used by hundreds of
simulations, each simulation will likely add other methods and properties that are limited to their individual concerns and functionality. In existing DIS architectures, type definitions are the same across all simulations. Hence, the Pedestrian class is likely to increase in size considerably, and it is very unlikely that a single developer will understand the entire object. Yet there might collateral effects between methods and properties, which is a perfect recipe for hidden bugs. ¹

One solution to the previous issues is having a way to represent pedestrians in danger as its own type. The first issue – difficulty in understand behavior based on source-code – would be mitigated by the semantics of spoken language. Calling a type EndangeredPedestrian would allow other developers to quickly grasp what objects this collection type represent. Even if reading the algorithm is necessary, the purpose of the type makes understanding the algorithm easier. As for the second issue, having an EndangeredPedestrian type means other simulations have a way to refer specifically to pedestrians in danger, and may make use of this calculated collection for their own purposes. For instance, the traffic simulation can request EndangeredPedestrian objects so it can move the cars out of the way as well. Finally, bloated objects are no longer an issue, since methods that pertain to EndangeredPedestrian would only be available in EndangeredPedestrian type, like the `move_to_safety()` method.

Besides addressing these 3 issues, typing collections enables a set of powerful operations that were previously restricted to flat data structures: relational algebra. Relational algebra is widely used in relational databases (e.g. SQL-based databases) to perform filtering, querying, and modification of data. This dissertation will show that the relational algebra used in databases can be successfully applied to typed collections as well. As an illustration, the EndangeredPedestrian scenario in relational databases is a mapping of a SQL statement that

¹This scenario is well known to us. In developing the OpenSimulator virtual environment platform, one of the largest points of contention in refactoring has been the type SceneObjectGroup. This type represents any in-world object in OpenSimulator. As the platform became richer, SceneObjectGroup became larger, containing methods and properties related to physics, scripting, crossings between simulators, and many other aspects of the virtual environment that were continuously added in the past years.
is similar to this (with a simplified euclidean distance):

SELECT * FROM Pedestrian JOIN Car ON ABS(Pedestrian.pos - Car.pos) < 0.1;

The largest complication in achieving typed collections is efficiency: how frequently does this meta-programming paradigm need to refresh object collections? This is likely the reason why typing collections of objects are non-existent in major OOP implementations. In time discrete simulations, however, there is a period of time when simulations are expecting a new event or a new time step. This is the ideal time to apply modifications to objects and recalculate collections for the next step.

Another complication is maintaining consistency between types. All objects of type EndangeredPedestrian will have a set type that is semantically the same. Changing the position of an endangered pedestrian will obviously need to update the position of the Pedestrian object. This entanglement creates a series of complications, that are addressed by the architecture and programming model of CADIS.

1.5 Thesis Statement

The state-of-the-art DIS architectures and frameworks provide little support to the independent development of collaborative simulations, particularly with regards to state synchronization of independently designed data models. This thesis proposes a novel approach called Collaborative Aspect-oriented Distributed Interactive Simulation architecture, or CADIS. This thesis presents the details of design and implementation of CADIS, supporting the following claim:

**Thesis Statement:** CADIS is a good fit for scalable simulations whose development is independently collaborative. Moreover, CADIS may be the best solution for large-scale
simulations that involve having to integrate different aspects of those simulations.

In the process of investigating this thesis, I will focus on the following research questions: (1) How does CADIS compare to traditional ways of load-partitioning simulations? (2) How does CADIS compare to similar architectural efforts such as HLA? (3) How effective is CADIS in supporting independent, but collaborative development of different aspects of simulations?

The thesis is organized in 7 chapters and an appendix. Chapter 2 provides a broad background in DIS architectures, data distribution, and collaborative simulations. Chapter 3 describes two previous design experiments in DIS that motivates our novel architectural approach. Chapter 4 presents CADIS, architectural approach to DIS systems. Chapter 5 shows a 4-part evaluation of CADIS that presents benchmarks and demonstrates feasibility and ease of adoption. Chapter 6 is an in-depth study of evaluating and testing DIS systems, presented lessons learned in DIS design experiments and applied in CADIS. Finally, Chapter 7 presents an overview of the thesis, followed by an appendix in Chapter 8, containing source-codes and additional benchmark tables.
Chapter 2

Background on Distributed Interactive Simulations

Computer simulations are applied to many different domain areas. Each application has different goals, which in turn require unique sets of guarantees for its software implementation. DIS narrows down some of these requirements, particularly of partitioning synchronized state that must be available for short interval polling, and maintaining reasonable data consistency between nodes. However, the specific requirements on how available and consistent should the shared state be varies depending on the application and purpose of the simulation. This chapter broadly explores the development of DIS applications and platforms, exposing the diversity of requirements that an architecture for DIS must comply. This chapter is organized in 5 sections. Section 2.1, 2.2, and 2.3 presents a history of DIS platform and applications, categorized by their partitioning approach. Section 2.4 presents several approaches to data distribution and state synchronization in DIS. Section 2.5 presents emerging applications of DIS.
2.1 Object-Partitioned Architectures

Object-partitioned architectures model the world as a collection of objects that interact through events. Objects are owned by the processes that simulate them, and typically only one process holds ownership of an object at one time. A common approach to object-partitioning is by dividing the virtual space in regions, and assigning regions to processes. Each simulator contains all parts of the simulations, and the source-code is identical in each process. This approach scales by subdividing objects into smaller sets, until they are small enough to be computed by one process.

2.1.1 SIMNET / DIS

The first successful standard for distributed simulation was SIMNET [126], funded by the DoD and developed by DARPA. SIMNET was designed to address two essential challenges of simulations: 1) building high quality simulations at low cost; and 2) how to network simulations for coordinated exercises and training. SIMNET relied on 3 basic principles.

The first principle was modeling the world as a collection of objects that interact through events. Each simulator simulated one object (e.g. tank, aircraft), and the terrain and environmental information was common knowledge between nodes. The second principle was to use a decentralized, peer to peer architecture. In this architectures, updates only reflected changes to known state. Each node was responsible for maintaining its own state and broadcasting ground truth updates. The receiving nodes were responsible to filter out irrelevant messages, as for instance events that are too far away for the node to perceive. The third principle was the use of dead reckoning to interpolate position updates. Updates had a guaranteed frequency of at least 1/15 seconds, and contained position, direction, and speed. The responsibility of extrapolating the position of remote nodes fell upon receiving
nodes.

With SIMNET, it became possible to link hundreds of simulators to produce a virtual world, used for real-time, man-in-the-loop, coordination and tactic simulations. In 1993 SIMNET was officially made public, evolving into the IEEE Distributed Interactive Simulation (DIS) standard (IEEE 1278-1993) [50]. The DIS standard supported simulating many different classes of simulations (e.g. aircraft, tanks, ships), leading to many different implementations such as [120].

2.1.2 MR Toolkit

The MR Toolkit [157] was focused on networked virtual reality (VR) capabilities, such as head-mounted displays, hand input devices, and sound feedback. Originally, the MR toolkit was not networked and used only for single user virtual reality applications. A year after the first publication, Shaw and Green presented an extension called The MR Toolkit Peers package [156] that enables peer-to-peer connection between the master processes of the MR toolkit. This extension uses UDP over the Internet, sharing updates by establishing a complete graph connection topology between applications. One implementation is EM [186], forming a DVE where each node contained a complete copy of the world.

2.1.3 BrickNet

BrickNet [158] is a client-server DVE architecture that enables object sharing. Clients connected to the server can upload objects, which the server then distributes to all other connected clients based on interest management. Clients must subscribe to objects of interest to receive updates, and can throttle the frequency of updates for trading off consistency and network bandwidth. These fundamental mechanisms for creating and sharing objects in a
shared virtual environment are still the norm in most of current DVE architectures.

### 2.1.4 RING

RING [66] is a hybrid DVE architecture, mixing both a client-server and peer-to-peer approach. Like SIMNET, RING models the virtual environment as a collection of objects that interact through events. Different than previous architectures, each object in RING belongs to only one process. Ownership implies that only one process has write permissions to that object, though other processes may have copies for displaying. Clients connect to servers, in a client-server fashion. The servers are interconnected in complete graph peer-to-peer fashion, and are responsible for filtering messages to reduce network processing exhaustion from update messages. Such exhaustion was common in SIMNET due to broadcasting of every update, causing depletion of CPU due mostly to processing of packets. RING contributes with a novel approach to calculate visibility of objects, only forwarding updates to the subset of clients that can see the updated object. This interest management approach is a form of a **space-partitioning** strategy, common to modern DVE architectures. RING uses not only distance, but also geometric 3D calculations to determine whether an object is visible or not. That is a similar approach to the modern DVE platform Meru [38]. Object ownership is also widely adopted in modern platforms with user-generated content, like Meru and OpenSimulator.

### 2.1.5 DIVE

DIVE [33] is peer-to-peer and models DVE as shared memory over a network. This shared memory is split in terms of worlds – a specific of objects and parameters. A DIVE process – which can be a person or an application – is a member of exactly one world at one time. Worlds are implemented as process groups, and updates are shared by using one multicast per
world. DIVE allows dividing the world into subregions using communication groups. DIVE’s approach is more flexible than traditional space-partitioning, as process groups have no pre-defined virtual space shape or size. Some applications in DIVE have used *auras* – volumes of space around users and objects that determine which other entities should receive updates. However, auras are not part of the architecture, but part of the applications designed to use DIVE.

State is stored locally for high availability, and consistency is achieved by using mutual exclusive locks, a reliable ordered multicast protocol, and distributed object locks. Objects can be composed hierarchically, may have multiple views (i.e. set of graphical representations), and can have attached behavior like scripted objects in modern platforms (e.g. Second Life, Meru).

### 2.1.6 MASSIVE

MASSIVE [76] is a spatial-model for DVEs, with focus on improving interest-management between objects and users. MASSIVE defines some key abstractions in DVEs that can be combined to reduce the number of state updates shared between processes. In MASSIVE, interactions between objects occur in a *medium*, typically audio, visual, or text, though other mediums could be devised. Like in DIVE, objects have *auras*, a volume of space around an object that binds that object to the medium, and enables potential interactions. When an aura of one object intersect with another, interaction becomes possible. Interaction happens at the discretion of the objects, based on the levels of *awareness*: when one object becomes aware of another, manipulated by an object’s *focus* and *nimbus*. Focus describes an observer’s area of attention, whereas nimbus describes the observed object’s area of observability. The observer and the observed may have different focus and nimbus, so the relationship between may not be mutual. When one object becomes aware of another,
a stream of updates of the observed should be sent to the observer. The frequency of those updates should be affected by levels of awareness, as for example the details of buildings that are far away. Nimbus can be increased by changing the medium, as for instance, raising the volume of a virtual radio would make it noticeable to longer distances.

MASSIVE also supports heterogeneous mediums to the virtual environments. For the same virtual environment, one user may connect with a terminal-interface with text-only capabilities, another user may login with a full-graphical viewer, and another with just audio capabilities. All users can still share elements of the virtual environment, as long as their mediums support that multimedia type. This heterogeneity for DVEs proposed in MASSIVE is present in modern platforms as well, as for instance, OpenSimulator allows voice communication without logging in to the simulator with the standard viewer. REST-like interfaces also enable parts of a virtual environment to be available publicly, such as inventory lists, textures, and other virtual assets. In the video game World of Warcraft, for example, characteristics, inventory, and other assets of game characters are available online on the company’s website.

### 2.1.7 Spline

The Scalable Platform for Large Interactive Networked Environments, or Spline [187], is a hybrid space-partitioned architecture for DVEs. Originally, Spline had a pure peer-to-peer topology, but was later changed to have some of the Spline processes act as servers, making it a hybrid architecture. The world is modeled as sets of objects, referred to as the world model. Each Spline process contains a copy of the world model, that is synchronized with other processes. For consistency, Spline uses a relativity model, where world models are only approximately equal across different processes. In order to scale, the world model can be broken up into smaller parts called locales. Objects in a locale only sends updates to
users and objects in the same or nearby locales (i.e. geographically close in virtual space). Each locale is associated with a multicast channel, avoiding the overhead of a peer-to-peer complete graph connection topology.

There are 3 different types of messages in Spline, each requirement: (1) small size fast object updates; (2) large size slow object updates; and (3) streams of data. For the first, typically for small changes in state like position, UDP is used. For the second, for data such as audio, images, or meshes, resources are retrieved by using the World Wide Web (WWW): URLs fetched through HTTP protocol. The last type, used for streams like video and audio, uses UDP, similarly to traditional video and audio streaming done in the WWW.

2.1.8 Second Life / OpenSimulator

Second Life [111] is a proprietary DVE platform where users can enter a virtual environment and live a different virtual life. Users can customize their avatar’s appearance, and may buy and sell virtual objects with a virtual currency. Additionally, a user that owns a virtual land space may add new objects or build them from scratch freely. The added objects will be shared with all visitors of that space. In the 2000s, many companies bought and established their own islands in Second Life to establish their virtual presence in DVEs. Second Life is space-partitioned, with fixed sized areas of 256 $m^2$. Not many details of the architecture are officially divulged by its developer, Linden. Second Life was first popular general-purpose (i.e. non-gaming) virtual environment.

OpenSimulator [138] is an open-sourced VE that uses the same protocol from Second Life. By reverse engineering the message protocols used in the Second Life, OpenSimulator also reverse engineered the Second Life architecture. In OpenSimulator, the simulation is divided in the simulator and services. Simulation services provide common utilities to simulations, such as handling authentication, asset and inventory management, and connecting the simulators in
a grid. The simulator contains all common functionalities of a virtual environment: physics, scripting for objects, managing events (using the Linden UDP, or LLUDP, protocol), and customized modules developers can add as part of the simulation.

### 2.1.9 Meru

Meru [38, 85] is a DVE platform focused on scalability using space-partitioning, similar to Second Life and OpenSimulator. The major improvements in Meru’s spatial partitioning approach is threefold: (1) enhanced geometrical algorithm for determining which objects are visible at a distance; (2) object management, using routing tables to find and reach remote objects; (3) space-partitioned processes are of dynamic-size regions, using hierarchies.

The visibility calculation in Meru is an improvement over previous existing implementations. While RING could calculate walls and geometrical shapes that blocked view, Meru also takes into account the distance and size of objects to determine whether they are visible. For instance, a skyscraper can be very far and still be seen. A rock could be much nearer, but be physically impossible to spot.

As opposed to complete ownership model found in OpenSimulator, Meru allows objects to communicate across processes. Scripted behaviors can send messages to any objects in the virtual world. Those messages are forwarded by the Meru *Application Message Forwarder* component, responsible for maintaining a key-value data storage of the location of every object distributed in the multiple simulator processes. Network throughput is throttled based on distance: objects that are close will exchange frequent updates, while objects that are further apart will send sparser updates.

The space-partitioning used in Meru is dynamic and hierarchical, using kd-trees – binary trees that recursively split an area of space using axis-aligned splitting planes. A region is
always is sub-divided in half, forming two sub-regions. Splitting continues to happen until each region has less than \( n \) objects, with \( n \) being configurable.

### 2.1.10 Kiwano

Kiwano [47] is a space-partitioned architecture for scalable DVEs. Kiwano proposes improves the space-partitioning approach by using a dynamic-sized region based on a Voronoi diagram that adapts to real-time simulation load. As objects move around the virtual space, Kiwano automatically adapts the space partition areas (called *zones*) to balance the workload efficiently. Zones communicate with each other to send updates near the borders of the zones, so interaction is not cut-off at the limits the zones.

### 2.2 Service Partitioned Architectures

Service partitioned architectures models the world as a collection of services that operate on shared data. There is no concept of ownership, processes update the data freely when running services. A simulator contains one simulation, effectively modularizing domain expertise in one module, executed by one process. Service partitioned architectures scale by dividing the workload of a complex simulation into smaller independent services. Each simulator has different responsibilities, and thus, executes different source-code. A flexible option is having a code base that can plug-in modules configured in run-time.

The only known architecture to directly support a service-oriented partitioning approach is the Distributed Scene Graph (DSG) architecture [102]. DSG distinguishes two independent concerns in virtual world simulations, data (*Scene*) and *operations*. *Operations* are organized by functional features into *actors*. 
• **Scene**: Represents the data structure and state of the virtual world and its entities.

• **Operations**: Reads and writes the data structures and state of the Scene. Operations modify the Scene to simulate the virtual world, and are aggregated in functional features.

• **Actors**: Operations can be aggregated by functionality. For instance, physics operations are responsible for updating the Scene with physics-related behaviors such as collisions and acceleration. Script operations would update the same Scene, but concerned with scripted behaviors, such as updating an object size or location. These aggregated functional features can be consolidated in independent simulators, which we refer to as *actors*.

To provide a scalable virtual world, the workload should be partitioned and mapped to multiple servers when the load exceeds the capacity of a single server. Traditional workload partitioning work [33, 45, 86, 92, 118, 165, 187] focused only on Scene partitioning, leaving operations unpartitioned. Limitations of Scene load partitioning [112, 113] were identified, caused by heterogeneity of operations. DSG enables operation disaggregation, so different simulators operating on the same Scene are capable of running on separate hardware and be load balanced independently.

### 2.3 Generalist Architectures

Generalist architectures attempt to leave partitioning to the developers of the simulations, while supporting common simulation services and transactions. Simulations share access to a centralized data storage. Simulators express interest in writing/reading data structures, and the architecture is responsible for synchronizing data automatically. Scalability is entirely dependent on all modularized and partitioned simulation implementations.
2.3.1 Open Wonderland

Open Wonderland [92] is a Java open source toolkit for creating 3D environments. The architecture of Open Wonderland is based on DarkStar [16], a transaction-based approach to scaling DVEs. Instead of relying on object ownership to achieve scalability with data consistency, DarkStar uses data caching and a data access lending mechanism. Each simulator process has a data cache, storing data items (e.g. objects, data structures). When an item is requested for read permission, DarkStar checks if the item is available in cache. If not, a request is sent to retrieve from a database server. The server will either send the data to the requester, or, if the item is currently being written by another process, will be queued and blocked until all current transactions for that item are done.

When a request for writing data is made, the server will either return the write permission or throws an exception in case of a timeout. With the write permission, the process can perform multiple transactions without accessing the database server. Updates to the data structures are queued and periodically forwarded to the database server. Concurrent read requests are queued in the database server and forwarded to the process with the write permission, to be run and returned in an appropriate time. Timeouts have fixed intervals, and processes can reissue queries to the database server when they occur.

2.3.2 High Level Architecture (HLA)

Concurrently to the effort of developing object-partitioned DVEs, another effort aimed to integrate multiple classes of simulations (e.g. tanks, command centers, aircraft) and enable other classes of applications, such as interfacing with live components and data collection. For this purpose, another standard was developed with the participation of industry and academia partners: the High Level Architecture (HLA) [87].
HLA is a general-purpose architecture for distributed simulations, defining a set of protocols and standards to provide common simulation services like time management and publish-subscription of objects. HLA is divided in two component types: the federates and the runtime infrastructure (RTI). Federates are the simulations, data collection applications, or interfaces to live components. All federates use the HLA service interface to share state, send and receive events, and configure properties of the entire simulation, referred to as the federation. The RTI implements the services defined by HLA interfaces, including time management (time-stepped or event-based), object discovery and publish-subscribe requests, and starting and stopping the federation. It is also used to share the object model template (OMT), a shared specification objects and attributes between federates. When objects are created by a federate, the RTI informs other federates interested in this object type of a new instance. Then federates can subscribe to attributes of the object instance and receive updates when state updates occur. The RTI has many commercial implementations, including VT Mak RTI and Pitch pRTI. Two well known open-source implementations are OpenHLA and Portico.

One of the major purposes of HLA is to enable collaborative simulation development: simulations that involve integration of multiple domains of expertise. In the Department of Defense, the most common example is contracting – where companies are paid to develop new military technologies such as missiles, vehicles, or aircraft. Simulations allows such equipment to be tested before production, reducing risks and costs. But the simulation of these technologies needs to be tested in coordination of other equipment and control in a virtual battlefield. HLA allows simulation developers of different domains and companies to build compatible simulations that can ultimately interact in a training scenarios.

The potential to build collaborative simulations was recently used in a civilian research initiative. NASA organized a collaborative simulation project called Simulation Exploration Experience (SEE), originally called Smackdown [53]. The SEE was created to encourage ed-
ucation in distributed simulation and the HLA standard. NASA created a scenario, building a moon base, and gave researchers all the tools necessary to start a moon base simulation. The tools include a federate for coordinating the federation and for delivering common data types, documentation, and functionalities, such as reference coordinates from Earth and the moon, a visualization federate, and APIs for the simulation. From there, researchers from many different universities volunteered to help. Many scenarios were proposed and built for the simulation, including communication systems [57], surveillance and defense of base [29] and a lunar mining operation [171]. The success of SEE was presented at several simulation conferences, with researchers from multiple universities presenting positive experience and results [29,57,171], and positive impact for education in distributed simulation [56]. While the simulation was executed successfully, the product was a toy example: the moon base simulation is not in any predictable way useful to real world NASA missions or plans for a moon base. On their analysis of SEE, Essilfie-Conduah et al. [56] say that SEE has potential to tackle real world M&S engineering problems. The experience from researchers participating in SEE also showed that HLA was difficult to implement, and the lack of tutorial examples made the learning curve even steeper. Essilfe-Conduah et al. also notice that while HLA is highly functional, it is also dense in content.

Distributed simulations like HLA are known to be difficult to use. In a 3 part survey about distributed simulations and HLA, Boer et. al found that most simulation practitioners find there too little cost-benefit in adopting distribute simulation solutions, mainly due to high technical complexity of existing standards and frameworks [19]. To address the technical difficulty of developing distributed simulations in HLA, several layered architectural abstractions have been proposed to improve separation of concerns and allow distributed simulations to be developed similarly to local simulations. One example is SimArch [73], a 5-layered architecture that separates the distributed computing infrastructure (e.g. HLA, CORBA-HLA) from higher-level simulation concerns, aiming to have simulations developed as if they were local. Tang et al. [168] proposes a layered architecture with a Resource Management Fed-
eration (RMF) that integrates the HLA Federation Object Model (FOM) to other forms of data models, such as the ones provided by Web services, while allowing dynamic FOM creation and update. Topçu and Oğuztüzün [175] use a model-view-presenter (MVP) architectural pattern – commonly used in Web applications – for separation of concerns between user interface, simulation logic, and HLA-specific communication, allowing these components to be developed independently. CADIS is also a layered architecture, but is unique in its data-oriented approach to state synchronization. CADIS replaces HLA’s FOM with an object-oriented relational typing model (details in Chapter 4), where simulation events are represented as object updates.

2.3.3 DEVS

Other generalist approaches besides HLA have been proposed and adopted for distributed simulations, particularly focused in the field of supply chain management. One of these approaches is proposed by Zeigler et al. using a formal modeling framework called Discrete Event System Specification (DEVS) [195]. DEVS formally addresses time management, events, and state updates for parallel simulations. To make it distributed, DEVS can run on top of either the Common Object Broker Architecture (CORBA) [139] – the DEVS/CORBA framework DEVS/CORBA [193] – or HLA [194]. In either flavor, the partitioning scheme is generalist, giving the developer the power (and the responsibility) of correctly partitioning the simulation across multiple nodes.

2.3.4 GRIDS

Another generalist architecture for distributed simulations in supply chain management is GRIDS (Generic Runtime Infrastructure for Distributed Simulation). GRIDS [170] uses thin agents for realizing distributed simulation services. Thin agents are similar to traditional
simulation agents, except they are replicated across the distributed simulation nodes, having the capability of moving from one location to another in the infrastructure. GRIDS share some similarity to a service-oriented architecture, except thin agents do not represent the node’s service in a static manner, but rather enables different service-like functionalities to be mobile across all nodes.

2.4 Data Distribution

A vital part of a DIS architecture is state synchronization. Synchronizing the multiple events and entity states across multiple processes is a challenging task, and many architectures either resort to an existing data distribution framework or develop a new one. Most data distribution frameworks use some variation of the publish-subscribe paradigm – a programming model that allows components to communicate anonymously through their interest in a common topic. However, the complexity of DIS synchronization proves simpler publish-subscribe mechanisms to be an insufficiently powerful solution for a DIS architecture. This section covers some advanced solutions for data distribution that better fitting to handle DIS state synchronization.

2.4.1 Publish-Subscribe

The simplest and most well known form of event distribution is the publish-subscribe pattern – publishers announce a topic (typically a string) for publishing data and subscribers interested in updates for that topic perform a subscription. There is no direct interaction between publishers and subscribers – the source does not know the subscribers for the data nor does a subscriber know its source. To implement this anonymous exchange of information, there is broker component that accepts subscriptions and publications requests from
other components and perform the routing of messages appropriately. Information that is published in this paradigm is called an **event**, while the act of delivery is called a **notification**.

Patrick et. al have organized an extensive literature survey of the many existing publish-subscribe approaches (the "siblings") and other similar data interaction paradigms (the "cousins"). The many data distribution services described in their work are categorized in terms of decoupling 3 dimensions: **space**, **time**, and **synchronization**. **Space** decoupling regards the anonymity of the participants – publishers and subscribers are not aware of each other. The **time** decoupling allows events to be published and subscribers to be notified independently. This implies that a publication can happen when subscribers are not available, and a notification may happen when the publisher is no longer available. Finally, the **synchronization** decoupling allows publishers and subscribers to publish and receive notifications asynchronously – neither party is blocked during the interaction.

Within the publish-subscribe siblings, solutions are categorized by the way interest is specified. There are 3 different categories of interest specification: **topic-based**, **content-based**, and **type-based**. **Topic-based** solutions express interest using labels, something as simple as name. Publishers will create events for that label, and subscribers of the label will receive them. **Content-based** approaches allow interest based on the content of the event. A classic example is a stock market system, where a publisher produces stock values and a subscriber can be interested in stock that raises above a value, or drops below a value. **Type-based** interest allows expression of interest in types rather than content or label. In the example of the stock market, a stock event can have two different types: stock request and stock quotes. Stock quotes represent values of stock for sale, while stock requests could be used for brokers to show interest in purchasing stocks. The survey by Patrick et. al lists the most well known approaches in each of the previous categories – both for publish-subscribe siblings and cousins.

CADIS, our novel approach to DIS presented in Chapter 4, is a "cousin" of the publish-
subscribe solution. The *spacetime* framework in CADIS is responsible for maintaining state synchronized between simulations. Spacetime, as the name implies, decouples both space and time dimensions between publishers and subscribers of data. However, different from other solutions described by Patrick et. al, synchronization in spacetime is explicit. Each time step of the simulation implies in *push* and *pull* methods to be called: *push* is a publishing action for creation and modification of objects and *pull* is a subscription action to retrieve state of interested object types. Although spacetime cannot be categorized as a publish-subscribe service (due to its non-decoupling of synchronization) spacetime uses expression of interest similarly to the type-based publish-subscribe pattern. In spacetime, simulations publish and subscribe to new and modified state of objects – instances of data types. Spacetime allows finer tuning of these subscriptions, allowing subscribers to specify interest in creation, modification, or deletion of objects based on data types.

Another unique characteristic of spacetime is the use of dependent types – types that are dependent on runtime state – allowing data types to be associated with filters. Using a similar stock example from before, a CheapStock data type would be a Stock type whose value is below a certain threshold. Thus, spacetime is a combination of both the content-based and type-based interest categories used in publish-subscribe systems.

### 2.4.2 OMG Data Distribution Service (DDS)

DDS [142] is a specification designed by the Object Management Group (OMG) for a data-distribution middleware that provides interoperable data exchange between participants (i.e. publishers and subscribers) of a distributed system. One of the unique motivations of DDS that stands out from other networking middleware is its focus in data-centric real-time data exchange. CORBA, for instance, is a widely used middleware for distributed systems, but CORBA is an object management solution with focus on distributed services. DDS is
concerned with reliably and consistently delivering state updates between participants of the distributed system. For that purpose, DDS has a series of Quality of Service (QoS) properties that allows participants to fine tune their requirements for publishing and subscribing to streams of data.

The ideal of DDS is to provide a global data space, where participants can write and read data from. To achieve this goal in a flexible and scalable way while maintaining QoS properties requires by participants, DDS uses the Data-Centric Publish-Subscribe (DCPS) model – publishers announces and publish information regarding topics, while subscribers inform their interest in existing topics. DDS is responsible to consistently deliver messages from publisher to subscriber, while obeying the configured QoS properties of both publishers and subscribers.

A feature that makes DDS unique from previous solutions is the definition of the topic. In most publish-subscribe implementations, the topic is a string defined by the publisher. In DDS, topics are not only a loosely defined label, but are also tied to a data type. Data type instances are called data objects. *Tying data types to topics is a fundamental design decision of the CADIS architecture, which makes DDS both an interesting design example to follow and a successful proof of concept data distribution model.* CADIS, however, introduces a novel collection data typing programming model that makes DDS an incompatible solution to its data distribution needs.

There are 4 important modules in DDS that implement the publish-subscribe mechanism: Publisher, Subscriber, DataWriter, and DataReader. The DataWriter is the module that interfaces with the application, offering an API that allows the application to communicate the value and existence of data-objects. It is then the responsibility of the Publisher module to distribute that data object to subscribers. Similarly, the DataReader gives the application access to subscribe to data objects based on their data types, and the Subscriber is responsible for receiving and relaying data objects to the application.
Both publication and subscription may be bound by QoS control properties offered by DDS. The list of available QoS properties is extensive, but some of the most meaningful ones related to our architecture, CADIS, are:

- **DEADLINE**: A writer can commit to producing values within a certain interval, and a reader can inform its expectation of receiving a value at a certain interval.

- **TIME_BASED_FILTER**: A reader can request to receive updates only above a certain interval, informing the framework it may not need all updates provided by the producer.

- **CONTENT_BASED_FILTER**: Allows a reader to filter out received data based on its content.

- **HISTORY**: Determines whether the framework keeps the history of changes or only latest values.

The previous QoS controls relate closely to some design decisions of data distribution in our own architecture. For deadline, simulations in CADIS must define a fixed interval where state updates from data types of interest to the simulation are pulled, and modified objects are pushed. Time-based filters and content based filters are also existent in HLA: time-based filters can be set by simulations and content based filter corresponds to a typed collection operation called *subset*. Chapter 4 presents this features in detail. Due to the nature of collection types in CADIS, maintaining a history would be unfeasible in terms of memory space. Hence CADIS adopts a last value configuration only.
2.4.3 HLA Data Distribution Management (DDM)

The Data Distribution Management service of HLA [132] enables region-based interest management combined with the existing object management (OM) and declaration management (DM) services. The OM and DM services provide class-based publish-subscription (CBPS) functionality – simulations can publish and subscribe data types, and HLA’s RunTime Infrastructure (RTI) will appropriately deliver state updates performed by a publisher to interested subscribers. However, if a simulation is very large, there might be too many publishers and subscribers for the RTI to handle in a timely manner.

The DDM service adds a space partitioning based interest management for its existing CBPS. Using a multidimensional coordinate system, publishers in DDM specify an area where their publications should be forwarded to subscribers. Similarly, subscribers will inform an area where they are interested in published data. Any data types that are updated by a publisher will be reflected (i.e. received) by subscribers that have overlapping regions of interest. This is a type of value-based filtering similar to the one existing in DDS and in CADIS, but is specifically tied to the use in space partitioning filtering. Both DDS and CADIS allows value based filtering on any type of data property.

2.5 Collaborative Simulations

Collaborative simulations are simulations that interact dynamically to create a larger simulation environment. Collaborative simulations are often simulations from different domains of expertise. The example used throughout this dissertation is an urban simulation, that may contain, for example, simulation of traffic, pedestrians, land zoning, and electric and water distribution. Each of these domains are often separated in distinct research areas and city departments. However, many of these areas interact in the real-world within the city.
environment. Developing an urban simulation that does not reflect the dynamic interactions between these different domains would not result in a realistic simulation. Enabling interactions between simulations of different domains is the main goal of the CADIS architecture. We present the two largest areas of research where collaborative simulations are a promising solution: urban simulations and virtual geographic environments.

2.5.1 Urban Simulations

Urban simulations model and simulate urban environments and they can be both interactive and non-interactive. Wadell [183] interprets urban simulations as “operational models that attempt to represent dynamic processes and interactions of urban development and transportation”. He argues that urban models have grown in complexity, and simulations are becoming a vital part of decision making by stakeholders for urban policies. A comprehensive history of urban modeling can be seen in [182]. In a talk presented at the Winter Simulation Conference 2014, Fujimoto [65] reiterates the importance of urban simulations today and in the near future. The advent of new technologies (e.g. smart power grid, smart homes) and new vehicles (e.g. unmanned aerial vehicles, electric vehicles, autonomous vehicles) will require redevelopment the urban infrastructure. One of the grand M&S challenges proposed by Fujimoto is to handle the complexity of so many different interacting simulations by composing distributed simulations.

One of the largest areas of research within urban simulations is the simulation of traffic. Pursula [146] provides a comprehensive overview of the different models and approaches to traffic simulation. Mainly, traffic simulation is divided in degree of detail: microscopic, mesoscopic, and macroscopic. The macroscopic models were the first simulation models for traffic, treating traffic as a continuous flows that can be modeled mathematically, for instance, as waves [107]. Microscopic simulations attempt to model and simulate vehicles
individually, determining starting and ending points, and calculating routes through roads and intersections. Most traffic simulation applications used nowadays are of microscopic nature. The car-following model [74] and cellular-automaton [135] variations are typically used in microscopic simulations. Mesoscopic simulations are a mix of macroscopic and microscopic models. One example for modeling mesoscopic simulations is the gas-kinetic approach for two-dimensional traffic flow [134]. Another example of a mesoscopic approach is CONTRAM (COntinuous TRaffic Assignment Model) [106], where movement of traffic is modeled by grouping vehicles in ”packets”. We focus on microscopic simulations for its widespread use and the leveraging of computing power increase from parallel and distributed simulations. There are many distributed and parallel models for large microscopic traffic simulations [60,144,191], that can be scaled independently from our urban simulation platform.

There are multiple microscopic traffic simulator implementations available for use. Some of the more popular ones are SUMO [98], TRANSIMS [162], PARAMICS [30], and VISSIM [59]. Maciejewski [121] provides a comparison between SUMO, TRANSIMS and VISSIM. PARAMICS is focused in parallel and distributed simulations, optimal for use with thousands of cores, tightly coupled and interconnected by network.

2.5.2 Virtual Geographic Environments (VGE)

Virtual Geographic Environments is a novel tool for geographic analysis, allowing collaboration between multiple areas of research in the Geographic Information Science (GIScience) field. Lin et. al [108] were one of the first researchers to discuss VGEs as a major solution for the many challenges of collaboration and multi-dimensional visualization in GIScience. A VGE, in their conception, is broken down into 4 main components: data, modeling and simulation, interactive, and collaborative. The objective of VGE is to allow both the interac-
tion of knowledge from its users and developers and to provide a common platform for data visualization, all built upon the same geo-data content. Each different group of collaborators can work on a different aspect of a geographical problem, effectively creating multiple geo-models – geographical models that apply on a set of geo-data. The geo-data is typically measurements and values, whereas geo-models are methods describing behavior [190].

The collaboration scenarios described by Lin et. al [108] and Xu et. al [190] are a good match to the goals of CADIS – allowing multiple contributors to work on the same data, but model their solutions independently. Existing approaches of VGEs [196, 197] are specific to the domain, relying on sharing existing data models and definitions used in GIScience. Synchronization is done at at the data level, using conflict resolution solutions like transactions and distributed memory. However, these solutions only address the lower level problems of distributed systems. At a higher level abstraction, it is still an unanswered question as to how multiple contributors can share their models in a collaborative VGE (CVGE). Xu et. al specifically raises this question in discussing the conceptual framework of a CVGE:

“Geo-models come from many disciplines, for instance oceanography, atmospheric science, geology, biology, demography, economics, mathematics and physics. A discipline creates its own professional models according to its own research customs. In order to integrate multi-disciplinary models into CVGE, they must first be unified in format. This problem can be explained by taking GIS as an example. GIS unifies geo-data by location (x, y, z) and time (t). Given the location and time, all of the geographic data and information can be associated in GIS. Returning to geo-models in CVGE, what approach can be adopted to unify the huge number of heterogeneous geo-models that are created by multiple disciplines? This should be a topic for future research.”

There are not many implementations of CVGE frameworks. Zhu et. al [197] proposes
a CVGE framework using P2P and grid. The architectural style of the framework is a 3-layered system. The resource layer contains the shared resources between participants, such as data, computing capabilities, storage, and knowledge resources. The service layer maintains synchronization between participants, using both P2P network and grid platforms for maintaining the location of participating nodes and enabling interoperating services. Finally, the application layer is where the user-developed applications reside, having access to service layer API for services like instant messaging, video collaboration, file search, and terrain scene visualization.

Another approach to designing a CVGE framework was proposed by Zhang et. al [196], using Web services for synchronization between nodes. Similarly to Web architectures, this approach is a 3-layered architectural style. The first is the back-end systems layer, maintaining the data and computational resources stored persistently. The second layer is Web services, containing Web servers for processing user requests and storing results to the back-end layer. The client layer is where user-developed applications exist, and communicate with the Web services layer using Web protocols, such as HTTP (for browsers), SOAP, UDDI, and WSDL. The architectures of Zhu et. al and Zhang et. al differs primarily by where data is stored and how they are synchronized. In Zhu et. al, data is stored locally by each participant, and the service layer is responsible for communicating with other participants to maintain a synchronized state. This approach is similar to most early virtual worlds (e.g. SIMNET, DIVE, MASSIVE) and some earlier ones (e.g. Meru and DSG). The Web approach used by Zhang et. al follows a modern trend of centralizing data storage in databases. Second Life and OpenSimulator follows this trend, using a centralized database for all data storage even though a grid is used to maintain update localization of each virtual world and negotiate avatar and objects crossings between regions.

The unification of geo-models is problem space where CADIS can be highly effective. In CADIS, geo-models would be mapped to data types, containing both properties and behav-
ior. By using CADIS dependent types, other models can refer to the same properties (i.e. data) while providing new behavior (i.e. new methods and runtime filtering). CADIS stores all data in a semi-centralized fashion. All data that is related must be collocated, so that queries that use different data types have local access to their properties. However, non-related data may be stored separately, and simulations may specify where should updates from each data type be fetched.
For many software systems, scaling in size can be addressed by adding more computational resources and replicating existing processes. The most ubiquitous example is the Web, that scales up by transparently adding servers running identical processes and fulfilling the same requests for data. This approach is effective when data is read often, but modified scarcely.

In the DIS domain, state is constantly changing and evolving, and updates have real-time delivery requirements to participants. This set of conflicting requirements makes DIS systems difficult to scale, often requiring simulation specific optimizations to scale up in size. From a non-functional software requirement perspective, DIS systems expect to have low latency, high availability, and immediately consistent state. But according to the CAP theorem [27], it is not possible to fulfill these 3 requirements simultaneously. Hence it is the task of a DIS architecture to support trade-offs in partitioning, availability, and consistency according to each individual simulation’s requirements.

In Chapter 1, we presented 3 major concerns of DIS systems that hinder scalability. The
first is partitioning state, which is highly specific to each individual simulation, and can either be enforced (i.e. object and service partitioned) or supported (i.e. generalist) by a DIS architecture. The second concern is networking scalability, also known as the $O(N^2)$ problem: each state update that a client generates must be distributed to all other interested clients. Thus, if each client sends one update, $N^2$ updates must be distributed. The third is synchronizing state across multiple nodes. The first concern is widely explored in existing approaches, presented in Chapter 2. This chapter focuses in addressing the last two concerns through two design experiments: the Restful Client-server Architecture (RCAT) architectural style, and the Distributed Scene Graph with Microcells (DSG-M).

Section 3.1 presents the details of the RCAT architectural style. The work on DSG-M is presented in detail in Section 3.2. Section 3.3 wraps the chapter, looking closely at conflicts between scaling up in size and scaling up in scope, motivating the design of CADIS in the next chapter.

### 3.1 RCAT Architectural Style

RCAT is an architectural style that supports scalable DIS non-functional requirements, but leaves the simulation partitioning design decisions to the developer. RCAT is a layered architecture, isolating network, application logic, and state synchronization in separate layers that can be developed independently.

Figure 3.1 provides a logical description of the architecture. Three horizontal tiers compose it: client management handle the communication with clients, application logic handle computations, and the data management ensures the persistence and synchronization of state. Each tier isolates different system requirements, providing independent evolution of each layers’ components.
In order to deliver non-functional properties that are essential for a large scale server application, we borrow the architectural constraints from the REST architectural style [62]. The objective is to enable three desired non-functional architectural properties for MMVEs: adaptability, fault-tolerance, and scalability.

Figure 3.2 provides an illustration of the RCAT architecture. The blue boxes represent the components, while the yellow boxes represent the connectors. The first layer is the Proxy layer, and consists of the Client Handler, Proxy Linker, and the Application Handler. The Application layer consists of the Application Connector and the Application and Service components. The Data layer consists of the Mapper and the Persistence and Object managers.

The Proxy layer and the Application layer are separated by network connections, and communicate by events. The Application and Data layer are tied together procedurally (through the Mapper), and communicate with other instances across a network as well.

The next subsection presents REST and its relation to RCAT. Following, the 3 tiers of the architectural style are described, addressing the responsibilities of both the tier and each
Figure 3.2: RCAT Style Architectural Model.

individual component, and finally, the physical network view of the distributed deployment of RCAT is presented.
REST

Short for representational state transfer, REST is a software architectural style used most commonly in the World Wide Web. REST provides a set of constraints and rules that if followed, delivers several desired non-functional properties, such as fault tolerance, scalability, and reliability. Through the implementation of its most well known and used middleware, the Apache Web server, it has been the guiding force behind the wide spread use of the World Wide Web in recent years, growing beyond the static web pages to interactive and dynamic content.

REST has been widely disregarded in the MMVE community for performance reasons, but its architectural benefits have shown results as highly scalable and reliable web servers over decades of use. Hardware and network has shown tremendous improvements since the beginning of commercial MMOs in 90’s, and it is now feasible to conceive a RESTful approach to MMOs and MMVEs. RCAT is a study on the impact overhead of using REST in a field dominated by tightly coupled network components.

Following, the summarized architectural constraints from REST and how it relates to MMVEs are described:

- **Client - Server**: In order to provide the best possible quality product, commercial MMVE and MMO developers have been opting for client-server as the standard architectural style. Even traditionally P2P games such as RTS are now becoming client-server owing to the reduced prices of computing power and data storage, and the desire to own and study client traffic data. Additionally, several games have virtual goods that they wish to secure, and reducing client responsibilities makes transactions more secure.

- **Stateless**: No client-specific information (i.e. context) can be used in the server to
treat client requests. This constraint applies to individual client-server requests. With the use of the layered system constraint, it is possible to have externalized state, which is state that is kept on the server side of the system while maintaining each individual client-server interaction stateless. This architectural constraint essentially allows MMVE servers to be deployed the same way Web servers do: scalability through deployment of more servers running identical code. The problem of data synchronization across servers is not solved, rather it is separated from the client request treatment locus, and thus are allowed to evolve independently and be worked on as separate layers.

- **Cache**: Caches are intermediaries between system components. Their use is highly recommended, since they can reduce computations and traffic considerably. Caches contain information that is not supposed to change frequently. The Web is composed mostly of slower interactions that allow for caching very often. In an MMVE many of the updates are state information about the environment. These updates are frequent and must be as consistent as possible, to provide a uniform view of the world to all users. The addition of a distributed caching layer could be the solution to provide fast access of data for the TSO type of requests.

- **Uniform interface**: A RESTful server interface should be able to receive messages with different formats. Metadata should accompany the actual data to help the system understand the message content. MIME-types or timestamps are examples of metadata that apply to MMVEs. With proper understand of the metadata, a client manager layer (section 3.1) is capable of routing client requests to the correct servers.

Additionally, resources should be referenced by a uniform resource identifier (URI). In order to abstract the data domain from the application domain in MMVEs, the application should expose an application domain-specific API that is converted to a URI when a resource should be created, queried, or modified. This is discussed
separately in Section 3.1.

- **Layered system:** Through the use of layers (i.e. where components may only see other neighboring layer components), it is possible to store state on the server side of the system and still maintain every individual interaction stateless. Additionally, layers are planned to separate different concerns such as data and handling connections, affording independent evolution of the layers. When each layer may evolve independently, experts in different fields can contribute to an individual layer without requiring full knowledge of the system.

**Proxy Layer**

The first layer consists of proxies, also called client managers or communication-intensive components. The term *proxy* is used to refer to an instance of the proxy layer. Clients must send their messages to their proxy and receive state updates from servers through the same proxy, making this layer essentially equivalent of a network router, but at the application level.

- **Routing:** Proxies are responsible for routing messages from clients to the application servers and vice-versa. Default routing mechanisms are sticky and random routing. In sticky, the application handler associates one server to one client, and all messages from the same client goes to the same server. In random, message are sent to any available application server.

- **Connection handling:** All proxies must be connected, and maintain the connection to, all application servers participating in the same application session. One client must be connected to only one proxy at a time. Clients may be moved from one proxy to another.
• **App and admin channel**: Each proxy opens two connections to every application server. The app connection is the standard channel for forwarding client requests and receiving server updates. The admin channel is meant to be used for metadata. Common usage are for inter-server messaging, client list updates, and receiving new routing algorithms.

• **Uniform client reference**: When a client connects to a proxy, it must create a uniform, universal, and unique id to reference the client’s session and forward this id to all application servers.

• **Application server registration**: When an application server connects, it sends a unique id for it be referenced at runtime. Proxies must keep a track of all application server ids and forward the id to all other application servers upon registration.

• **Fault handling**: In the case of disconnected clients or application servers, the proxy is responsible to either resolve the fault or return an error to the origin of the routing request. When an application server crashes, it can pick a different application server. When a client crashes, or if another application server is not available, it is responsible to return an error code or message unique to the fault.

• **Application logic independence**: The proxy should not have hard coded, application specific, logic for routing.

• **Runtime routing algorithm**: During runtime, the proxy may receive new routing algorithms to better improve the application performance, as long as all other constraints are obeyed.

• **Messaging**: Proxies are the means of communication between application servers. As such, the proxy layer must provide broadcasting and single server communication capabilities through the registered application server ids.
• **Service routing**: Besides the application servers that handle client requests, different types of servers can provide functionalities to the application. For instance, a physics server could process heavy CPU processing and distribute the results to the application servers. We call these service servers, as they provide services to the application.

Due to the different nature of these servers, they must be treated as separate classes by the proxy. Each service registers its class with the proxy and becomes eligible to receive all or certain types of events from clients. The proxy is responsible to forward the client events to the services as required.

The architecture of the proxy layer is composed of 2 components and 1 connector:

**Client handler component**  The client handler is responsible for all interaction with the clients. As such, it must keep track of clients id to physical address mapping and provide services for the application handler component to reach the clients.

**Application handler component**  The application handler handles all interaction with application and service servers. It must handle server/service id to physical address and provide services for the client handler to reach the application/service servers.

**Proxy Linker connector**  The proxy linker connector is a composition of the **linkage connector** and the **procedure call connector**. The proxy linker’s role is to provide a public API for both components to communicate. The API is exposed in the form of abstract methods that both handlers should implement. When a message is received in the client handler component, it uses the API to send that message to an application server. When a message is received in the application handler and should be forwarded to a client, the application handler calls the API method to deliver the message to the client. This way each handler needs only to treat requests and data from the layer they serve, be it
clients or application servers. The proxy linker connector API for client and proxy handlers respectively can be seen in tables 8.1 and 8.2.

Routing from both client and server ends requires a very high level of I/O operations, which results in a high CPU and network interface usage, main reason why most network routers are implemented in hardware. Implementing the proxy in hardware can greatly increase the speed of routing, but limits how configurable they can be. Bauer et al [12] proposes a hardware implementation called Booster Box, performing similar responsibilities as RCAT’s proxy layer.

Instead, the proxy can be implemented as software, affording easy configuration of routing algorithms in runtime. While the proxy layer must not have any hard coded application logic, applications are allowed to send routing algorithms to the proxy layer to execute.

Routing algorithms give applications control of which clients they wish to treat requests for, allowing better locality placement of data and client requests. The restriction that any server must be able to process requests from any client still holds, but routing algorithms can be used to improve performance for the application.

**Application Layer**

The second tier consists of application and service servers. The terms *application server* and *service server* are used to refer to instances of the application layer, with application and service component instances respectively. Servers are CPU-intensive components that may be suitable for high-end machines optimized for calculations. The application servers are responsible for running the game logic and enforcing the game rules. They receive client information through the proxies of the first layer, and update the data layer accordingly. Traffic to application servers involve any types of actions, from low-latency world state updates (e.g. user positions and actions) to very rare content delivery (e.g. assets or client
patches).

Service servers are similar to application servers in their interest of client events, but they are not meant to handle client requests. Instead they process different perspectives of the application, to take the load off the application servers. Two common usages of service servers is physics calculations and script execution. Service servers are optional components of the application layer.

The application layer servers are divided into two different parts: the application connector and logic. The connector provides transparent user localization and inter-server messaging services. The connector can be reused for any application, while the logic is the application or service specific part that is replaced for different applications.

The architectural responsibilities of application servers are:

- **Process requests**: The main purpose of application servers are to process client requests and if necessary, transmit updates to concerned clients. Most commonly application servers are event driven, but they are allowed to send state updates to clients as needed.

- **Registration**: Application servers must register a unique id with the proxy, which will be forwarded to other servers. This step is necessary for inter-server communication.

- **Metadata**: The layer stores metadata about client and server location so that applications may be developed without needing to keep track of on which proxy clients are located. Through the services of the proxy layer, the application layer is capable of messaging other application servers located across the network.

A corollary to the capability of storing client and server locations is that this information must be maintained when application server or proxies crash. When application server crashes, other application servers receive a server disconnected message. But if
a proxy crashes, the application server must remove from memory all users from the the crashed proxy.

The architecture of the application layer is composed of 2 components and 1 connector:

**Application component**  The request handler that will perform the application specific processing. It connects to the application connector to reach clients and other servers, and connects to the mapper connector to request data required for handling requests.

**Service component**  A service component is a generic component that fulfills a specific service requirement. Example of service components are physic engines and script engines. These components have interest in application events but do not process requests from clients. Rather they have stronger ties to the application component, that requires the services in the service component. The application connector intermediates the interaction of application and service components, which can happen locally or across the network.

Service servers have the same architectural responsibilities save for processing requests. Instead, service servers provide a specific service to the application by interpreting application events, and use the application connector to route messages across the system.

**Application connector**  The application connector provides the service interface for the application component to communicate bidirectionally with clients and other servers located across the network. The application component can send messages to clients or other servers by listing the ids of the clients or servers the messages are intended for.

To provide transparency for the components, the application connector is responsible for maintaining the consistency of the list of servers and users present in RCAT. In case of faults, the connector must process user and server disconnection messages, and also treat
proxy crashes, by removing its users from cache.

The application handler in the proxy layer starts two channels with the application connector: the **admin channel** and the **user channel**. Additionally, the application connector handles messages in two directions: user or other server instance messages from the proxy to the application, and application messages to the user or to other server instances. Thus, there are four dimensions of event treatments: channel (user or admin) and message direction (from proxy to application, from application to proxy.

The application connector events are described in tables 8.3, 8.4, 8.5, and 8.6 in JSON encoded format. For the messages coming from clients and other servers, the connector should decode the routing part of the message and deliver it through the appropriate channel to the application.

Table 8.3 describes admin channel, proxy to application events. There are 5 types of events, new user (**NU**), user disconnected (**UD**), new server (**NS**), server disconnected (**SD**), and others. For the first four types of events, proxies broadcast the event to all application servers. They inform of new users and application servers joining and leaving the middleware. The application connector must keep track of existing users and application servers, and then pass the event to the application for application-specific treatment. Any other type of messages are messages intended for the application from other application servers (see table 8.6), and the application connector needs only to forward it to the application’s admin channel callback.

Table 8.4 shows the proxy to application messages, but through the user channel. The application handler adds the id of the user that sent the message to the application connector. The connector only forwards the message as-is to the application.

Table 8.5 are events arriving from the application to the proxy through the user channel. Application messages to users should carry only one additional metadata, the list of users.
who the event is intended for. The application connector parses the list of user ids. For every user id, the application connector finds the proxy instance that is connected to that user. Finally, the message is forwarded to the user’s proxy through the user channel.

For proxies connected to multiple users in the intended recipients list of the event, the application connector can pack a message to a list of users. The application handler in the component will parse the list, find the connection handler of each client, and forward the message with the application’s event. If no users are specified, the application connector parses its entire list of users and sends the message to their proxies, packing multiple users when possible.

Table 8.6 is the admin channel from application to proxy. This particular channel and direction is used for inter-server communication and proxy information request. The most important event is the registration (REG) message. When the application connector is created, it generates a UUID for identification. When the connection to the proxy is performed, the application connector sends a (REG) message to every admin channel with the recently generated id. One proxy is picked at random (in order to avoid message repetition) to respond to the registration message, by broadcasting to every application server a \{NS:serverid\} message, as seen in table 8.3. The server picked receives an extra parameter, the RT (i.e. retransmit) variable, which when true, informs the proxy it should send out NS messages to all servers.

The application may also request a list of current users and registered servers (LUR and LSR respectively) from any proxy instance. Used primarily for new application servers entering the system, or less commonly, fault recovery and sane-checking algorithms, as the application connector keeps track of all connections and disconnections of servers and users.

For inter-server communication, the application connector allows for broadcasting or targeted message forwarding (BC and FW). In the forwarding message case, it requires a server
id to forward to. Otherwise, the application connector sends the message as-is, which is interpreted by the proxy’s application handler. The application handler parses all of its server ids, retrieving their connections, and sends the message to every server available.

The last two events are for dynamic routing of messages from the client. The first one, MU or move user, is meant for changing the sticky server associated to a client. The proxy’s application handler changes the sticky server associated with the client id in question (U), and new messages will be forwarded to the new server id (SID) passed. The last event, CR or custom routing, is used to send routing scripts. This functionality still hasn’t been standardized in RCAT, and is, for now, disabled. Sending remote scripts have several liabilities and security issues that must be addressed carefully, but will provide improved performance and manageability for RCAT.

Data Layer

The third tier is the data layer. This layer focuses on data manipulation and persistence, and isolates the data synchronization problem. Persistence and synchronization can be addressed by a single database, a distributed database, or even a server solely storing data in memory. What distinguishes the RCAT architectural style to current MMVE approaches is an entire isolation of the application state in a logical layer, and the proposal of an object oriented approach for treating synchronization. As such, domain database techniques and optimizations in parallel data processing can be used independently from any application specific logic. The responsibilities of the data layer are as follow:

- **Conversion**: The data layer is responsible for transforming application-specific requests into resource operations, and convert the resource operation responses back to application-specific format. The application-specific layer is necessary to keep the game logic on the server, simplifying the client requests. The resource operations use
similar URI (or URL) operations as in HTTP standard.

- **Transparency**: The data layer must perform consistency, availability, and partition tolerance transparently for the application. The application should see the layer as a black box that accepts data to store and retrieves consistent data immediately.

- **CAP**: The ideal transparent scenario of consistent, available, and partitioned data can not be achieved simultaneously, as described by the CAP Theorem [27]. The data layer is responsible for implementing the algorithm prioritizing the most important properties for the application that it is serving.
  
  - **Consistency and Partition Tolerance**: To provide horizontally scalable and consistent data, the data layer must implement a data synchronization algorithm that spans across different servers. Most well known approach is space partitioning, where different servers handle data located in different geographical areas of a virtual environment. For different types of applications, other approaches can be used.

  - **Availability**: For MMVE applications, availability is the most desired property. Reducing latency is essential for a good quality of experience in virtual environments, and as such, data must be readily available to respond to client requests. Some applications may tolerate inconsistency or have little data dependency, making availability easier to provide. When possible, caching also greatly improves this property.

To perform the previous responsibilities, the architecture of the data layer consists of one connector, the **mapper**, and two components, the **object manager** and **persistence manager**.
Algorithm 1 Retrieve objects within an area

1: function RETRIEVE_OBJECTS(int x0, int x1, int y0, int y1)
2:     objlist ← []
3:     for all obj in objects do
4:         if x0 < obj.x < x1 and y0 < obj.y < y1 then
5:             objlist.append(obj)
6:     end if
7:     end for
8:     return objlist
9: end function

Mapper The mapper is a connector that controls consistency and partitions the data for horizontal scalability. The connector is named mapper because it maps an application specific query to the location and use of the data that will resolve it, while also arbitrating data access with other instances running in parallel. Another way of comprehending the mapper is as a translator of application level operations into data operations for a distributed system. Mappers are an extension of the application servers for data access, and are connected to the application layer procedurally.

For instance, on a traditional space partitioned scalability approach, the query typically contains an \((x, y)\) pair, which gets mapped to an URI, with a server location and an operation of query, creation, or modification of data. Essentially the mapper provides application specific API for data access, transforming that request into a data request that gets processed by the persistence manager and the object manager.

In a concrete example, an application may need to request all objects located in an area, such as \(0 < x < 50\) and \(0 < y < 50\). The mapper could provide a method that takes an area and returns users. Algorithm 1 shows a sample implementation.

In this particular case, servers are looking up objects stored in memory, in the pointer objects. With space partitioning, only the servers that have their assigned area intersecting or containing the requested area would need to execute the function. If another form of data partitioning is being used, potentially all servers would need to perform the lookup for
objects in that area. Picking the best data partitioning algorithm is essential to avoid such scenario.

If using a centralized database, instead of looking in memory, the function would issue a database request for all objects where $x$ and $y$ are in the boundaries of $(x_0, y_0)$ and $(x_1, y_1)$.

**Persistence Manager**  The persistence manager is a component that manages data access to a central location, typically a database. All instances of the persistence manager running in parallel will point to the same location, making it a hub for synchronization of state. Depending on the application, it may also be used in replacement of the object manager to store, retrieve, and update all application data, but it must be used cautiously: centralizing the data locus may hinder scalability by creating more demand than the central data server can treat. An application may wish to use distributed databases to scale, in which case the persistence manager would connect to different servers but see the same data state.

**Object Manager**  The object manager is an optional component, intended for distributed memory caching algorithms. The object manager enforces that every object may have up to only one authoritative owner, thus solving the request serialization problem in distributed systems. The communication protocol used for resolving resources is the well established HTTP [61], with the exception of subscriptions, which upgrades to a bidirectional TCP or UDP (bidirectional configuration) communication stream, and resource relocation.

Only a subset of HTTP is used in the object manager, with the addition of two new request methods: RELOCATE and SUBSCRIBE. The methods and descriptions can be seen in table 8.7. The responses are HTTP Response types.

In contrast to the environment where HTTP is used, with a defined client and server, in object manager both components can be clients and servers in separate transactions. To
Algorithm 2 Retrieve object data based on its position and id

```plaintext
function RETRIEVE_OBJECT(int x, int y, int uid) ▷ Test
    obj ← None
    if x < x2 then
        obj ← om_retrieve(uid, server1)
    else if x >= x2 then
        obj ← om_retrieve(uid, server2)
    end if
    return obj
end function
```

demonstrate a transaction, take two object managers, origin and server. Origin requests a
GET from server. Server runs a local version of GET to retrieve data from memory, and
returns it to origin. At a later time, server may send method requests to origin as well.

The object manager provides an API for the mapper to request a remote or local object,
which can be seen in table 8.8. The mapper picks the distributed partitioning algorithm
it wishes to use, and the object manager performs a lookup across other nodes in the data
layer to find the owner, query, update, or transfer ownership of the object as requested.

From the mapper perspective, the object manager is a black box that is capable of querying,
modifying, or transferring an object to and from a server, while always maintaining a single
authoritative of objects throughout the data layer. The object manager must also provide
publish-subscribe functionality for each object, allowing a node to stream updates about a
certain object without the need of a request.

Algorithm 2 better illustrates how the object manager can be used. In this scenario, the
world is divided in the x axis at the point x2. All objects with x < x2 are located in server1,
all objects with x >= x2 are located in server2. The function om_retrieve that is omitted
takes two arguments: the user id, and the location where it should look for the object, and
returns the requested object.

In this example, the om_retrieve method converts the arguments into a URI and performs a
GET type of operation (e.g. GET 10.0.0.1/001) to another instance. The object manager on
the receiving side will then process the URI, execute the operation, and return the response. The object manager should also be used to move an object from a server to another. In the space partitioning method, it happens when an object or user moves beyond the area delegated to one server into a new area delegated to another. The mapper should notice this change and use the object manager to transfer the object from one server to another, in order to keep the intended space partitioned mapping consistent.

![Network Model of RCAT](image)

**Figure 3.3: Network Model of RCAT**

**Network View**

The previous sections show the architectural overview of RCAT, but the physical view provides a better visualization of node distribution and how connectors and components communicate across a network. Figure 3.3 presents a sample setup of RCAT with 3 proxies, 2 application/service servers, one database and 12 clients. There are two local area networks.
that run on the server side, isolating bandwidth and network congestion of proxies from the network shared between application and service servers requesting data from each other, databases, and any other source of data. If any proxy becomes overwhelmed, adding new proxies in the proxy layer can reduce the number of users per proxy. If the application or service servers become overwhelmed, the same approach can be done in the application layer.

Network congestion will eventually happen in a simple model as presented in figure 3.3, but it can be greatly diminished by a good network hardware implementation. EVE Online has a very complex multi-layered network middleware [25], to avoid such traffic congestion in their local area networks. thus providing a better approach as demand grows higher than the standard hardware network deployment is capable of handling.

### 3.1.1 Applications

The implementation of RCAT in the form of a middleware enables an application-independent platform for MMVEs and other collaborative, interactive, massive multiuser applications. To provide a concrete view of how a diversity of applications can take advantage of RCAT, a set of example applications are provided, demonstrating how they would operate and scale through the use of the RCAT middleware. All examples use JSON encoding for event messages and data structures.

**Example Application: Jigsaw**

In order to provide a real-world application to test the RCAT middleware implementation, we developed a jigsaw puzzle game. While not a virtual environment application, it has desirable properties for driving the middleware:

- **Top-Down View:** The jigsaw puzzle is difficult to scale with conventional interest
management. While there is a concept of interest, which is the player's frustrum, a player can quickly move from one side of the board to another. Such unpredictable behavior makes space partitioning an inefficient approach, bringing to question what other forms of interest management can be used, if any. Additionally, it provides room for testing interest management that are application independent.

- **Implementation**: A jigsaw puzzle is simple to implement in a reasonable period of time. Virtual environments and MMOs take tens of developers many years to deliver a good product.

- **"Fun"**: Instead of using a research prototype, we have chosen to develop an application that can be appreciated by users. By delivering a game that is fun to play and open-sourced in Github, we hope to introduce and spread the use of the RCAT middleware to game developers, who in turn can contribute back to improve the code and review the architectural style.

The implementation of the client is in Javascript, using HTML 5 to draw pieces and to capture input events. Websocket is used to connect to RCAT. Python is used in the application server and mapper, also connected by Websocket. Next, the implementation details of the Jigsaw application is presented, providing a concrete example.

**Data Structures:**

The main data structure for the Jigsaw game is the jigsaw piece. Each game instance has a setting of number of columns and rows, determining the number of pieces that are created. One of RCATs constraints is that data should be organized into objects. In the case of Jigsaw, the object is the jigsaw piece, identified by a unique id (the universally unique identifier (UUID) is used). Each piece object contains its id, position, user lock, whether it
is bound to the right place or loose on the board, and the column and row that it is supposed to be fit in.

**Client:**

The client is exposed as an HTML webpage, with an HTML 5 canvas in the center. The client application starts requesting the user’s name, for the purpose of score keeping. When the user connects, the RCAT proxy layer generates a new user message that is sent to all application servers. The proxy layer picks one server to treat requests for this client from now on, and the picked server replies with a configuration message. The configuration message is described in table 8.11.

Upon receiving the configuration file, the client downloads the image, splits it according to the configuration, draws the board and the pieces in the location they are meant to be, show the scores, and start listening for events. The game screen\(^1\) can be seen in figure 3.4. Pieces that are bound are displayed in the correct position and cannot be moved (i.e. mouse click events are ignored towards those pieces).

\(^1\)Copyright and photo courtesy from Activision Blizzard
The click and drag events are translated into \textit{pm} events. Releasing the mouse button is translated into \textit{pd} events. If the clients picks up a piece and later receives an event that the piece is being moved by another player, it immediately places the piece in its original position and off the player’s mouse cursor.

One important design decision is that piece move and drop messages are eventually consistent events, and piece binding messages are immediately consistent events and must be guaranteed to be delivered to every client. It is then possible for different clients to see a loose piece in different positions, but it is impossible for clients to have distinctive views of the "bound" state of a piece. Particularly during the initial snapshot, no application server
holds the entire state of all pieces, so the snapshot of piece location can be in an eventually consistent state.

**Application Server:**

The application server parses messages from client and event messages from the proxy layer. It is also responsible for starting the game, creating the pieces, and ending the game. The events parsed by the application server in Jigsaw are:

- **New User (NU):** When a user connects, the client handler component in the proxy layer fires a new user message to an application server. Jigsaw uses sticky routing, so the server compares the serverid on the SS part of the message (see table 8.3) to determine if it should reply. If the server determines it is the handler for this client, it replies with the configuration message in table 8.11. Also adds client to list of connected user, for score keeping purposes.

- **User Disconnected (UD):** Removes user from score keeping connected clients.

- **Piece Move (pm):** Invokes the respective mapper method, `set_piece_movement`. Upon success, broadcast message back to all users (no interest management) or to the returned user list from the mapper.

- **Piece Drop (pd):** Similar to piece movement, but releases lock from users. If bound is set to `true`, checks X and Y value for a match of where the piece is supposed to bound. If X and Y do not match the bound location (i.e. client was mistaken or cheating), call `set_piece_movement` with empty user lock. Otherwise, call `set_piece_bound`. If piece was not bound, forward to interested users, or all in case of no interest management. If piece was bound, inform every client.
The RCAT style does not provide an abstraction from distributed programming. Any application that runs on top of RCAT must be implemented as a distributed application, which leads to an inevitable complex software design. Jigsaw is designed to be distributed, but requires 3 points of synchronization: beginning a game, binding a piece, and ending a game. To begin the game, a server is elected the leader. Simplest algorithm is to just pick the highest serverid. If a leader crashes, the second highest id takes place. The leader creates pieces in random locations and pushes them to the mapper.

To bind pieces, the application server calls the mapper which immediately pushes to the database. Upon return from the mapper, the application server pushes the bind message to every client. It is an essential requirement that all players and servers have synchronous views of bound pieces.

To end the game, the leader must once again be used to avoid all application servers ending the game simultaneously. If no interest management is used, the client can inform the server the game has ended. The game server responsible for the client could check if it did with other servers, and thus reply a game end. But when game clients have interest management, they do not have knowledge of when the game ends. Each individual server in RCAT does not hold the entire state, and also cannot determine game end. To detect game end, the leader constantly checks if all pieces are bound in the database, which is eventually consistent. The downside is that after the last piece is placed, a delay proportional to the database persistence timer will happen when clients are waiting for the game end message.

**Mapper:**

Jigsaw uses RCAT’s object manager for data synchronization. No interest management was implemented, so full broadcast is required for every client action. The mapper provides two methods for piece management, and additional methods for score keeping, which is beyond
the scope of demonstrating RCAT usage. The piece management methods are show in table 8.12.

The initialization phase of the mapper creates a jigsaw table in a MySQL database. RCAT creates jigsaw_obm and jigsaw_reg to locate servers and store objects ownerships for the object manager.

The application server leader randomizes piece positions based on configuration and uses the mapper to insert all pieces in the database. For efficiency purposes, a different method for piece creation was used, so that the object manager does not need to be used, which would cache all pieces in memory. Without a space partitioning algorithm, they would all be stored in the leader’s cache. Pieces will instead be cached on application server per demand. When a piece is required and not found in object manager caches, the object manager will select them from the database, mark ownership, and keep it in cache.

For an improved performance, the Jigsaw mapper can also take advantage of the subscription feature of RCAT. If a piece is located on a different server to where the client piece movement request is being handled, the mapper can start a subscription to the piece until it is dropped and the lock is released. This way only one round trip is needed, and the further messages can be sent through a more performant protocol, such as UDP. Piece binding and drop messages should be taken cautiously, and the mapper should enforce a guaranteed delivery protocol (e.g. TCP) so inconsistency does not become noticeable.

**Fault Tolerance:**

Fault tolerance has not been built-in in the existing version of Jigsaw, but the application was built to support fault handling mechanisms. If a crash happens to one of the Jigsaw applications servers, 4 issues must be treated:
1. *Clients "sticky" to that server will no longer reach it.* Easily and automatically solved by the proxy layer, by changing the "sticky" server associated with clients of crashed server.

2. *Authoritative objects held by crashed server become unavailable and their recent state is lost.* This issue is typically the most troublesome, and can cause possible rollback of data to previously known state. Due to the design decision of Jigsaw, piece movements may become inconsistent without penalty to the user interaction, and may be ignored. If the server crashes between bounding a piece and synchronizing that information, the piece bounding would be rolled back to the state from database, with minor inconsistency noticed by the clients.

3. *Other application servers must be informed of crashed client.* Proxies inform all servers that a server has crashed. Due to lack of a central control, all proxies would send the same message, and application servers simply ignore repetitions.

4. *In the case of interest management techniques that assign unique roles to application servers, the roles must be redistributed.* For instance, in a space distributed world, 2 servers may divide pieces related to their location in the board. Given a board with $x$ pixels in the horizontal axis, if a piece is between 0 and $x/2$ it resides in server 1, otherwise, in server 2. In such a case, if one server crashes, another server instance must be relaunched to take its place, or the mapper must adapt to recreate the space distribution with the number of servers left.

One simple way to perform recovery in space partitioned interest management is to use quadtree structures [150]. If a server leaves, the quadtree is fixed to accommodate a smaller number of available servers.
Multi-purposed Virtual Environments

Virtual environment applications consist of a graphical client viewer on the client side, communicating bidirectionally with a server-side to send the user’s actions, and receive the world simulation updates. The viewer displays an *avatar*, a digital persona that other users’ avatars will see and interact with.

Users input events (i.e. key presses, mouse movements and clicks), and these events can be sent immediately to the server for interpretation and meaning, or can be translated locally into an action type the virtual environment server comprehends. These actions or events are then processed according to rules of the virtual world simulation, and the results of these actions are sent back to all other virtual environment users that must be notified. Visible actions, such as movement and handling of in-world objects, are commonly sent to users that are in view range. Other events might affect all players, for instance a weather change.

To adapt a virtual world application to RCAT, there are 3 parts that need to be developed: the client application, the application server, and the mapper. While the way these parts should be architected depends on the specific requirements of the desired resulting application, we wish to present one possible virtual environment architecture, with a simple set of requirements. The requirements for this example would be avatar movement and in-world object movement.

**Client:**

The client application should receive keyboard and mouse input events and turn them into action events. These action events are represented in JSON encoded format on table 8.13. A mouse click on a virtual world location sends a request to move the avatar to that position. If the location is far from the client, the client application should enforce an update rate and
adjust the X and Y location of the avatar based on the desired speed of the avatar.

For object movement, the client uses the mouse to click on an object, and then click on its new location. Object immediately moves to new location. This is a simplification of a real world scenario where a lock could be placed in the object, so that this object cannot be changed by other clients’ avatars until the original client desires to release the lock. We discuss how the lock mechanism would also work under RCAT in the mapper section.

When receiving messages from the server-side, there are two possible messages. For simplification purposes, the same semantics as in table 8.13 is used, with the only change being that avatar movement requests will come with a user id. If the operation is avatar movement, the client viewer draws the respective avatar at position X,Y. If the operation is object movement, the viewer draws the object at position X,Y.

**Application Server:**

The application server is handed the same messages from table 8.13 with an additional dictionary entry created by the proxy layer: \{U:userid\}. The server parses this event and branches the treatment of the operation into two possibilities, avatar movement or object movement.

When the avatar sends movement updates, the application server invokes the mapper method `set_avatar_new_position(userid,X,Y)`, as detailed in table 8.14. If the return value is true (i.e. success), the application server uses the application connector handle to send the event: \{am: [userid,X,Y] \}, which is broadcasted to all users.

This simple example use no interest management system, but to implement one with RCAT is trivial. The mapper should provide a method to determine interested users (e.g. by distance to the original user, an X,Y value comparison on a database) and return a list of
users. The application server attaches to the avatar movement message the list of users in the form: `{U:user_list}`, which is interpreted by the application connector, rerouting the message to proxies who have users in the user_list.

Object movement has the exact same pattern as avatar movement, and could also have the same implementation of interest management.

**Mapper:**

The mapper is responsible to transform application-domain specific requests into data ones. Table 8.14 shows operations for this simple application.

The mapper has two components in RCAT at its disposal to assist in this task: the persistence manager and the object manager. In the simplest implementation, the mapper could use the persistence manager to centralize all data in the database. This requests from other mapper instances would all be serialized at the database and order and synchronization is achieved, at the cost of loss of scalability due to the use of a centralized database.

A more scalable option would be to use the object manager instead, leaving the persistence manager for fault recovery and object manager synchronization only.

During the initialization phase of the mapper component, one instance is chosen to create the persistent media access, for instance, tables in a database. If object manager is used, RCAT automatically creates an extra entry (e.g. table) used to synchronize object locality and ownership.

When the object manager is not used, requests are immediately forwarded to the persistence manager in the form of the specific persistent media access chosen. For instance, if a SQL language database was chosen, an avatar movement message such as `{am:[50,100]},{U:1234}` would be transformed by the persistence manager into something similar to: `UPDATE`
vedemo SET X=50, SET Y=100 WHERE U=1234.

If the object manager is used, the mapper instructs object manager to retrieve the object. The application independent object manager in RCAT looks for the object ownership through the use of the persistence manager. If no object manager node owns the object, it stores its own server id so that other object manager instances know where to look for this object’s authoritative state. The object will remain in cached (i.e. in memory) until the mapper instructs the object manager to relocate or return it to ”not owned” state. Any updates or queries on the state of this object will be routed to the instance that is marked as owned in the persistence manager lookup by another object manager.

**Fault Tolerance:**

The treatment of runtime errors in multipurpose virtual environments are similar to the ones seen in the previous section, with the Jigsaw puzzle. The difference lies on determining what objects are eventually consistent and immediately consistent. Eventually consistent data (e.g. avatar’s position) can be lost, so when recovering, retrieve last known state from persistent media.

Immediately consistent objects are divided into two categories: critical and ephemeral. Critical immediately consistent objects must **always** be persisted and stored after every data transaction. A ubiquitous example is trading virtual goods and currency. A server must, at all costs, guarantee players will not lose their items and currency due to the server failure. Blizzard’s World of Warcraft performs an entire rollback when treating server faults, including virtual goods and currency transactions, which can be very frustrating for users. The Jigsaw example had only this type of immediately consistent objects.

The ephemeral immediately consistent objects are those that are always supposed to be kept immediately consistent, but are not critical to be saved and restored. For instance, if
users are battling NPCs and the server faults, they will first be sent to a different server for request treatment. NPCs will then either not be restored, or restored to a previously known state. While not the most desirable outcome, the application is allowed to function properly. Users will perceive a glitch of disappearing or health restoration in NPCs, but will not be too adversely affected by it.

**Massive Multiuser Role-Playing Games**

MMO role-playing games, or MMORPGs, are a more complex version of a virtual environment. User movement and actions have similar mechanisms, but there is an increased set of TSO that makes scalability essential for hundreds of users interacting simultaneously. Examples of TSO operations in MMORPGs are battles between players and non-player characters (NPC). Battle actions cause objects (player avatars and NPCs) to constantly be accessed and written to, to update health points when hit or healed, or to update a debit in in-world currency, for instance. These operations have to be serialized and consistent.

With the use of RCAT’s object manager, all requests are serialized to a single node, and scaling becomes a matter of load balancing. Assuming the common scenarios of MMORPGs, multiple groups of users battling separate NPCs, it is possible to balance the resource load by putting groups of clients and the data they are referencing in the same application server and mapper node.

This technique is employed in Blizzard’s World of Warcraft game when users join a group and enter dungeons [124]. The act of entering a dungeon with a group causes all players’ data and connections to be moved to another server that will host an instance of that dungeon. The same data locality technique can be applied to RCAT without the need to deploy a new instance and move players and data.

Yet there are reasons why creating a new instance is useful (e.g. multiple groups can be
in the same dungeon and not interfere with each other). The underlying assumption of the RCAT style is that all application servers share a common view of the data and its semantics, and treat client requests identically.

In such a case, if the system wishes to shard for other purposes, it must transfer to a parallel deployment of RCAT or to an entirely different architecture, possibly a more performant monolithic server given that only a limited and predictable number of users are interacting, and in a much smaller area.

Fault tolerance is covered in the previous section, except for one condition: raid boss NPCs. This particular type of NPC is meant to encounter players for long periods of time, reaching up to tens of minutes of interaction. In the case of a rollback, users will be more gravely affected by the rollback than they would be simple glitches of disappearing NPCs. The application can either make these bosses critical immediately consistent objects (by persisting every state modification), or provide architectural alternatives, such as trusting clients to recover game state, which introduces cheating and security issues that may not be desirable.

3.1.2 Experimental Feasibility Analysis

Setup

The examples in the previous section discuss how an application can be designed to use RCAT and to scale as a distributed system. Yet even if the implementation of MMVEs (and other applications) is shown to be possible on RCAT, there are justified questions of feasibility: for a latency-sensitive application genre such as MMVEs, what is the performance impact of the RCAT style? Does the performance penalty caused by the additional network hops introduced by the RCAT architectural style make an MMVE application unfeasible?

To study the impact of the added latency of RCAT, we implemented a proxy and a game
server in C#\(^1\), and used MySQL for the database. The clients, implemented in Java and Javascript, use WebSockets to connect to the proxy. The proxy and game server communicate via TCP. All messages contained a type field represented by an integer, followed by the parameters of the messages. The types can be seen in table 8.15. Messages exchanged were kept simple, and are detailed in table 8.16. Mapper is present only as a simple forwarded to persistent manager, which stores every message in the database.

Clients were represented as colored boxes in the graphical client viewer, and sent to the server x and y positions where they are moving to. Figure 3.5 shows a sample screen with multiple players. Colors were used to help real users tell boxes apart. Servers broadcasted the same information (i.e. client, x, and y) to all other clients so they can apply it to their model. The application handlers use sticky method to assign servers to clients.

\(^1\)the source code is available at https://github.com/Mondego/rcat-gs
We used commodity machines: quad-core Intel i7 HT with 8GB RAM for the proxy, server, and database. All the machines ran on Windows 7 and shared a 1Gbps connection through a switch. The round-trip time from proxy reception of a client message to proxy forwarding of a server message was measured every second. We report its average over each 3-minute run.

We simulated players using 5 to 50 bots sending position messages 20 times per second. When receiving a client message, the proxy forwards it to the game server, which stores the player’s new position in the database, retrieves the list of clients to notify (in this case:
everyone), and sends back to the proxy the list of clients and the message to relay. The bots moved up and down, changing direction when hitting the viewer’s boundaries. The visual client was used only for debugging purposes and were not part of the experiment.

We ran five different configurations. In the PS configuration, the proxy and server are on the same machine, communicate through loopback, and there is no database (the server updates positions in RAM). In PSD, a database is added on the same machine. When compared to PS, PSD shows the impact of the database without any network overhead. PScD resembles PSD, except the game server caches the list of users. P-SD is based on PSD too, except the proxy has been moved to another machine. Finally, in P-S-D the proxy, server, and database are on three different machines; this is the typical RCAT style. The last two scenarios highlight the impact of the network on performance.

**Proxy/Client manager**

Fig. 3.6 confirms the quadratic increase in bandwidth from the proxy to all the clients. This underlines the need for a client manager layer as a way to separate that quadratic network load from the rest of the system.

**RCAT overhead**

Fig. 3.7 shows the performance of RCAT (configuration P-S-D) as compared to PS. For 50 clients, the backend round-trip time of P-S-D ($\mu = 59ms$, 99th percentile = 330ms) is an order of magnitude larger than the round-trip time of PS ($\mu = 3.2ms$, 99th percentile = 19ms).

Two factors explain the noteworthy performance divergence. First, comparing P-SD to PSD shows that the network overhead between server and database can cause up to 30ms extra
latency for 50 clients. Second, comparing PSD to PS shows that even without any network overhead, the database adds 10 to 15 ms of latency for 50 clients. This shows the need for a distributed data management component faster than a relational database.

Additionally, the use of the traditional TCP stack is not very efficient for MMVE traffic. Due to Nagle’s algorithm, packets are held back for a period of time before they are sent, so more packets can be sent under the same TCP header. This wait adds latency when there’s little amount of data to be sent. In applications where latency is essential but bandwidth is reasonably low, Nagle’s algorithm becomes a hassle. The first few seconds of data in this experiment were excluded due to erratic latency caused by this algorithm. While it is possible to disable Nagle’s algorithm, C# and .NET does not provide any way to do it in the networking library.

**Cache improvements**

As seen in Fig. 3.7, P-Sc-D is bound between P-S-D and PS. Not having to request the list of clients for each message received saves the application a round-trip to the database. Compared to P-S-D, P-Sc-D reduces the latency by 13.5 ms on average.

Moreover, Fig. 3.8 presents the bandwidth requirements of the data layer. In the PSD configuration, the communication from the server to the database grows linearly with the number of clients, but communication from database to server grows quadratically. That quadratical growth is due to the constant requests for the list of all clients. As the number of clients increases, an increasingly longer list is returned increasingly more frequently. Caching this list of clients, as shown in configuration PScD, tames the quadratic bandwidth increase into a linear one.

This pinpoints the critical role of caches in the realization of RCAT. MMVE-specific caches running at the application-level increase the application’s complexity, may hamper scalabil-
ity, and can introduce bugs. Therefore, there is a need for a caching component specifically designed for MMVEs, but in a general-purpose manner such as COSAR [72].

![Client to proxy bandwidth](image)

Figure 3.6: Client to proxy bandwidth (as measured from scenario PS). ©2012 IEEE

Impact of MMVE Engineering

### 3.1.3 General Use

This paper has focused on MMVEs, mostly for being the class of applications that have the most demanding requirements. Yet the design of this architectural style has been free of any application specific design, including the premise of its use for an MMVE. RCAT can be used for any application that runs on a network and is meant for high user interactivity. With the growth and ubiquity of Web applications, we expect to see new uses for highly interactive applications in the future.
Software Reusability

As with REST, the RCAT architectural style brings an out-of-the-box scalable solution for implementing massive multiuser applications. With the addition of RCAT’s implemented middleware, the development barrier of MMOs could be brought to the smaller scale industry, that has difficulty of developing an entire network layer in addition to a game. The same way Apache brought down the barrier of developing Web applications, RCAT has the potential to bring new development in a field that is currently dominated by large industries that consistently maintain their practices hidden, the same way IIS [125] and other proprietary formats did with the Web.

Virtual Environments and the Web

Presently web servers are optimized for high availability for tens or hundreds of thousands of users. Usual Web applications have little demand to read and modify data at fast speeds, so no consistency is needed. In case of conflicts, which are rare, users are instructed to retry
The Web has become a platform for interactive social applications, with Facebook and Twitter leading phenomena of social networks. The new interactive Web requirements seen in Facebook and Twitter are becoming similar to MMVEs: fast rate synchronized state updates, interest management (e.g. friends list), and backend distributed service oriented architectures for data analytics [173] to name the most important. The update rate is many times slower than a virtual environment, but the number of users sharing the same resources is much higher. Web applications are already proving to be too slow for users, who demand more consistent and immediate interaction. Additionally, the social networks are adding multiuser gaming (e.g. Facebook apps), which is essentially the traditional network gaming on top of HTTP.

In light of these events, new protocols of bidirectional communication in the manner of TCP are being developed and approved, the most known being WebSockets. The bidirectional communication protocols will bring faster interaction to the Web, but without the constraints from REST, fault tolerance and scalability will no longer be an automatic feature. There
is a need for research directed at the highly interactive social Web that is growing, melding traditional REST and HTTP with full bidirectional TCP and sockets.

3.1.4 Discussion

We have presented the RCAT architecture in section 3.1, presented examples of applications in section 3.1.1, and discussed performance feasibility in section 3.1.2. After an extensive study of the proposed architecture, we wish to revisit the non-functional properties we set for RCAT: adaptability, fault-tolerance, and scalability.

Adaptability is discussed in section 3.1.1, where it is demonstrated how multipurpose MMVEs and similar (e.g. MMORPGs) can be adapted to RCAT. The core of the adaptability of the RCAT style is resulting from enforcing the stateless property, originally from the REST style. Because client request handlers (i.e. proxies and application servers) do not require to keep state, new instances of request handlers can be deployed to scale on demand. We have discussed previously that the stateless property is not a solution to the essential problem of MMVEs, but rather it isolates and externalizes the essential data synchronization issues from other parts of the architecture that can be parallelized.

The concrete example in section 3.1.1 demonstrates how a non-MMVE applications can take advantage of the same architectural style to develop a distributed system in order to scale. While this demonstration was not extensive to the multitude of all types of massive multiuser interactive applications, we covered many general aspects that are seen in interactive applications, such as updating objects, broadcasting events, and interest management capabilities.

Scalability has been discussed marginally, mostly because RCAT enables scalability, but does not provide it out of the box. Developing scalable distributed systems requires a
very application specific approach that an adaptable architectural style can not provide. RCAT provides the tools that guarantees consistency over a distributed system, which allows software designers to focus on how the application can run distributed as opposed to the mechanics of guaranteeing consistency.

Section 3.1.1 presents a concrete example of a distributed application, the Jigsaw puzzle. In the jigsaw application, scalability is achieved by a deep understanding of the application’s requirements, determining what pieces requires immediate and eventual consistency. Only then, after determining how can requests and objects be distributed across different nodes, does RCAT provides the tools to improve scalability.

Fault-tolerance has been discussed in section 3.1.1, for each application type. Runtime errors are still possible, and are capable of creating inconsistent states and loss of data for applications. It is inevitable that applications that use non-persistent (i.e. memory) media for authoritative data are capable of data loss. What RCAT provides is the mechanism to perform rollbacks from persistent media and keep the application running, despite a proxy or application server crash. The capability of rerouting users to different servers keep users unaware of server-side faults, except for inconsistencies and rollbacks that applications must perform if data is lost. Even proxy crashes can be made transparent, if the client application negotiates a new connection with another proxy.

3.1.5 Conclusion

Achieving scalability in a system that requires a unified view of the game state under low latency and high update frequency is an essentially hard problem. Existing solutions only work in particular cases of MMVEs and do not address the core problem: scaling an MMVE to thousands of concurrent users interacting with each other.
RCAT is an architectural approach to a middleware for massive multiuser applications. The proposed three-tiered architectural style, named RCAT, utilizes REST principles to deliver the desirable non-functional requirements of scalability, adaptability, and fault-tolerance, to applications that wish to scale to massive numbers of interacting users. The operations and events used in RCAT were presented in detail, including how the connectors interact to handle a distributed environment.

To demonstrate how a multitude of applications may take advantage of RCAT in order to scale and be fault-tolerant, 3 sample applications were presented – of which one has been implemented and is available open-source. The implemented example, a multiplayer jigsaw puzzle, provides a challenging application for scalability and is a simple and attractive game for users and developers. Finally, we discussed how RCAT can succeed in delivering adaptability, fault-tolerance, and scalability. RCAT has a place not only within MMVE applications, but also with a growing trend of fast, large, and stateful applications that are becoming commonplace in the Web.

### 3.2 Distributed Scene Graph with Microcells

The Distributed Scene Graph with Microcells (DSG-M) architecture is the combination of the DSG architecture and the microcell space partitioning proposed by Vleeschauwer et. al. [45]. Revisiting the description of DSG from Chapter 2, DSG distinguishes two independent concerns in virtual world simulations, data – referred to as the Scene – and operations. The Scene consists of data structures and state of the simulated entities, whereas operations read and write to the Scene. A collection of operations that fulfill a simulation service are referred to as actors. An example of an actor is physics: a physics actor is a collection of operations that fulfill the service of simulating physics in the virtual world, by updating the state of physical objects in the Scene. Actors in DSG are independent
simulators that operate on the same Scene. Hence, DSG is entirely partitioned by services (as opposed to object-partitioning) since all actors must share the exact same Scene. While this service-partitioning approach has proven to be better at load balancing in many scenarios where object-partitioning was not [113], there are also scenarios where object-partitioning is effective. Ideally, it should be possible to partition the virtual world simultaneously by service and by space. That is the goal of DSG-M.

The DSG-M architecture introduces flexible space partitioning to DSG, by allowing actors to have different – but possibly overlapping – Scenes. Multiple physics actors can, for instance, be instantiated and assigned to reduced physics simulation workloads by being assigned smaller areas of space to process. Our work adapts microcells – a dynamic space partitioning technique – to DSG, enhancing load balancing through flexible space partitioning, in parallel to service partitioning. We refer to our adapted architecture as DSG-Microcell, or DSG-M.

### 3.2.1 Interest Management

Integrating DSG with microcells poses some challenges. The original concept of microcells is constrained to space partitioning only, with no disaggregation of operations. This way each partition has only one authoritative simulator capable of performing write operations. By merging operation partitioning, there is no longer a single authority for each entity – in DSG it is possible for multiple actors to perform conflicting updates on the Scene. Thus, while the conceptual approach of microcells remains the same, DSG-M must set more constraints to microcells so state can be propagated and modified properly.

Microcells in DSG-M are rectangle shaped areas of pre-defined size. By using a Cartesian coordinate system, actors can be assigned to a collection of microcell subscriptions to improve load balancing. Actors subscribe to microcells to show interest in read and/or writing to entities and the environment spaces of the subscriptions. Microcells are also used for
interest management, to provide consistent update propagation. Subscriptions determine which actors should receive updates, based on the position where the update occurred. If one actor creates, modifies, or deletes an entity, all actors subscribed to that microcell will receive the update. Actors not interested in updates for that microcell will be spared the update messages.

An example of a microcell assignment can be seen in Figure 3.9. A region of 256 meters squared, which is the standard region size of OpenSimulator and Second Life, is mapped into 8x8 microcells of 32x32 meters. We represent subscriptions as a collection of Cartesian coordinates \((X,Y)\). As an abbreviation, we also use \((X_0 - X_1, Y_0 - Y_1)\) to represent a collection of microcells that are contained in the larger rectangle of coordinates: \(\text{Rect}((X_0, Y_0), (X_0, Y_1), (X_1, Y_0), (X_1, Y_1))\)

Figure 3.9: Two maps of the same 256 m\(^2\) OpenSimulator region. The left figure shows an unmapped region. The right figure shows the region divided by microcells of size 32x32 m\(^2\), mapping to Cartesian coordinates \((0 - 7, 0 - 7)\).

The mapping and subscription sets of microcells in DSG-M may take any shape. It is common to have overlapping subscriptions of actors of different types. An example could be a script engine actor and a physics actor both subscribing to the same microcell. The script actor would handle script events, and the physics actor would process the physics Scene. Conflicts are resolved by a simple timestamp mechanism, enforced in DSG [123].
3.2.2 Active and Passive Subscriptions

To determine read or read/write permissions, each microcell subscription must be subscribed in one of two ways: active or passive. Active subscriptions to a microcell are subscriptions with read and write access to the state of the microcell, while passive subscriptions are read-only access.

Active microcell subscriptions give the simulator full control of the microcell’s entities and environmental parameters (e.g. terrain), allowing any operations (i.e. creation, deletion, modification) to be performed and synced to actors subscribed to the same microcell.

Passive subscriptions are meant for read-only purpose: actors who are interested in updates but do not wish to operate on the microcell. Passive subscriptions are particularly useful if the actor needs to respond quickly to incoming entities from microcells outside of its subscription set.

A common example for passive subscription is in actors responsible for physics. Passive subscriptions are potentially useful in improving physics simulation correctness (e.g. collisions) near the borders of microcells, by allowing dead-reckoning of the entities in the passive microcells. The passively-subscribed physics actor will account for the entities of the passive areas, without broadcasting updates in that space to other actors.

3.2.3 State Propagation

State is propagated between actors in a subject-observer pattern, where each actor can be both a subject and an observer. At the start of the simulation, subjects inform observers of their microcell subscriptions. When an update is applied by an actor, it must notify all the relevant observers.
Notifications are triggered when an update is applied, due to one of two conditions: the subject actor created the update or the update came from another actor. Independent of how the notification is triggered, the actor determines the position in space where the update was applied. With the position, the actor determines the corresponding microcell coordinate. Finally, the actor attempts to match that microcell to the microcell subscriptions of observers, obtained at the start of the simulation. The actor then sends the update to every observer subscribed to the microcell.

In our current design, actors are organized in a star topology. The actors are divided into two categories: root actors and non-root actors. Non-root actors can only send updates to root actors. Root actors have two responsibilities. First, they act as messaging hubs, receiving messages from non-root actors and forwarding them to observers. Second, they are persistent actors responsible for storing and maintaining Scene data. While other actors may be instatiated at runtime, root actors must always be available, to determine the set of valid microcells for updates, and for consistent state propagation. Root actors are also referred to as persistence actors. Further work on persistence actors can be seen in [123].

An example of a star topology can be seen in Figure 3.10. Actor A is subscribed to the left half of the space, Actor C is subscribed to the right half, and Root Actor and actor B are subscribed to the entire space. The circle and triangles represent entities being updates. As an example, if an update is performed on the circle in Actor B, it is sent to the Root Actor, who forwards it to Actor A. If an update is performed on the triangle in Actor B, it is sent to the Root Actor, who forwards it to Actor C.

### 3.2.4 Crossings

If an actor is capable of generating position updates for an entity, it is possible that the entity will eventually cross the boundary of a microcell. The smaller the areas are partitioned,
Figure 3.10: An example of a star topology with 3 non-root actors, and 1 root actor. The region is mapped as 4x4 microcells, the circle and triangles represent entities that were updated.

the more entity crossings are likely to happen between two partitions. Entity crossings can be costly operations: they may introduce the latency of packing, sending through a network, unpacking, and recreating the entity in the virtual world stack, and may increase the bandwidth due to shipping of a large amount of data across the network.

There are 3 possible situations where entity crossings may happen. Let $m_1$ be a microcell where the entity is originating from; $m_2$ be the microcell where the entity is heading towards; $A_0$ be actively subscribed to $m_1$ and is originating or forwarding the update that will cause the crossing; and $A_1...A_N$ be the list of actors connected to $A_0$, with subscription to either $m_1$ or $m_2$. For every actor $A_i$ in $A_1...A_N$, the following situations are possible when a position update triggers a crossing at the origin $A_0$:

1. $A_i$ is subscribed to both $m_1$ and $m_2$. This means that $A_i$ is subscribed to the origin and destination microcells. In this situation, $A_0$ only forwards a simple update message with the new position, and locally updates the new microcell location for the entity. In Figure 3.11, this situation is illustrated by the crossing the circle entity in $A_0$. Actor $A_i$ has both microcells 2 and 3, and thus receives only the update informing
the new position and microcell.

2. \textit{Ai is subscribed to }m_1. A_0 \text{ may send a simple microcell crossing update, since } Ai \text{ has a copy of the full entity. } Ai \text{ will forward the update to other interested actors, and will delete the entity. In Figure 3.11, this situation can be seen by crossing the diamond entity in } A_0. \text{ Actor } A_i \text{ is subscribed to microcell 3, and was being updated on its state before the crossing. As in the previous situation, } A_0 \text{ can send just the position and microcell update. Actor } A_i \text{ will notice the entity is being crossed to a microcell outside of its subscription set, and will delete the diamond entity.}

3. \textit{Ai is subscribed to }m_2. A_i \text{ does not have a previous copy of the entity, since it was not subscribed to } m_1. A_0 \text{ must encode enough information for } A_i \text{ to build the entity locally. This situation is shown in Figure 3.11 by the triangle entity crossing, done by actor } A_0. \text{ Actor } A_i \text{ was not subscribed to microcell 1, so the crossing will generate a new entity. } A_0 \text{ must send all the data necessary for } A_i \text{ to rebuild the triangle entity in microcell 2.}

![Figure 3.11: An example to demonstrate the 3 entity crossing situations. In the example, actor } A_0 \text{ originates the crossing of the triangle, circle, and diamond entities, sending to actor } A_i.\]

The above list pertains to the possible situations of the receiving actor. The sending actor } A_0 \text{ is subscribed to } m_1 \text{ by definition, but not necessarily to } m_2. \text{ If } A_0 \text{ is not subscribed to } m_2, A_0 \text{ sends the relevant updates to } A_i, \text{ and deletes the entity locally.}
3.2.5 Evaluation

The first objective of our evaluation is to assess whether DSG-M is capable of dividing and distributing the workload execution by space. We expect the distributed workload execution to yield similar results to the non-distributed execution. Additionally, we expect that overhead of inter-simulator communication is minimized so that the load of distributed simulators is close to the same load on a single machine, should such a powerful single machine exist.

In conducting the experiments, it became apparent that the simulation was affected by border effects, such as simulation collisions near the border and entity crossings. As such, the second objective is to test whether passive microcells reduce the border effects. Passive microcells should reduce the number of large entity crossings and provide updates of objects near the borders for improved collision detection, yielding improved results over simulations with active-only microcell subscriptions.

Our experiment consist of simulations of a real device, the galton box [68]. A galton box is a board with pegs equally spaced in rows. The number of pegs per row increase from top to bottom by 1 per row, forming a triangular-shaped board of pegs. The simulated galton box can be seen in Figure 3.12.

![Figure 3.12: On the left, the original drawing for a galton box [68]. On the right, the simulated galton boxes for the experiment. The boxes at the top drop the balls.](image)

The simulation workload is generated by 4 galton boxes set in the world, each 93 rows high,
128 meters wide. The bins are 1.2 meters wide each, and balls are of 1 meter diameter. At the top of each box there are 27 droppers, disposed in 3 rows of 9. Each dropper creates a ball right below it every $t$ seconds, where $t$ is the period.

Balls dropped at the top collide with pegs, and have a 50% chance to drop to the right or to the left peg in the lower row. The end result is, when the balls are collected at the bottom, the distribution of balls in bins will be a binomial distribution. With a large enough number of balls, the binomial distribution approximates a normal distribution.

In terms of the evaluation, a normal distribution of balls is the measure of correct expected behavior. For purposes of evaluating DSG-M, we look into three metrics: (1) the error of the measured distribution of balls with respect to the expected distribution, (2) the average CPU load of the physics actors, and (3) the time it takes on average for balls to fall.

The first metric captures the precision of the simulation results. In pre-experiments, we identified two major sources of error, which we now control for: overwhelmed physics actor due to load, and inconsistency derived from space partitioning. The second and third metric captures the load placed on the simulator. By combining all three metrics, we can evaluate changes in the load of the simulators while controlling for the preciseness of the simulation’s expected result. Furthermore, we identify the source of the error in precision, and test solutions to reduce it.

**Experiment Description**

The experiments run on a DSG-M implementation on top of the OpenSimulator platform, using a star topology. There are multiple actors on each experiment, of 4 different types:

- **Root actor**: The root actor, also known as persistence actor, does not perform any updates in the experiments. It is dedicated to coordinate actors, persist data, and
forward messages. There is only one root actor in all experiments.

- **Physics actor**: The physics actor is responsible for creating physics updates, such as calculating positions and velocity based on forces and collisions. Depending on the experiment, there might be one or two physics actors present. The physics subscriptions varies per experiment.

- **Script actor**: The script actor is responsible for executing scripted actions. In the experiments, the script actor produces the balls at the top of the galton box on a regular interval period. In DSG only we use one script actor. In DSG-M, we use two script actors, covering two galton boxes each.

- **Client actor**: The client actor handles human observers. There is one client actor in every experiment, and one client connected to it. The client actor will not generate updates, but will receive every update from other actors.

All experiments use a microcell size of 4x4 meters. Each dropper releases 350 balls with a variable interval over the course of the experiment, for a total of 37,800 balls. Experiments were performed with dedicated desktops on a local area network. Each actor runs on a different desktop. The desktops are Intel Core i7-2600 CPU @ 3.40GHz, 16 GB RAM, and 1Gbps Ethernet controllers. Desktops run Ubuntu 12.10, and the application run with mono 3.2.8.

The experiments are divided into 2 parts. The first part of the experiments shows the effects of increasing load on results and physics behavior. The second part of the experiment measures the impact in performance and error of distributing load with microcell partitions on DSG.
Part 1: Non-partitioned Simulation

For this part, we use standard DSG, without microcell partitioning, and gradually increase the rate the droppers create balls to drop on the galton box. We test the resulting distribution and load measurements for periods of 12, 9, and 6 seconds. The topology for this part can be seen in Figure 3.13.

Figure 3.13: Topology and subscriptions of actors in the first part of the experiment, and in Scenario 1 of the second part. Each horizontal group of bars represent a galton box. Microcell grid not to scale.

Results Table 3.1 shows the results for the first part of the experiment. The period is the time interval between each drop. The average load is measured as CPU percentage, where 100% CPU corresponds to one CPU core being fully used. Our machines had 8 cores, so the maximum possible CPU load is 800%. The fall duration is the time the ball takes between creation and colliding with the ground, and RMSE stands for Root-Mean-Square Error (RMSE), the square root of the mean squared error. RMSE uses the difference between the expected number of balls for a bin and the measured value in the simulation, for all 96 possible bins.
Table 3.1: Part 1: DSG without partitioning, load measurements. **AL:** Average load (CPU%), **PA:** Physics actor, **RA:** Root Actor, **FD:** Average fall duration of balls in seconds, **AB:** Average number of balls on the galton box, **RMSE:** Root-Mean error.

<table>
<thead>
<tr>
<th>Period</th>
<th>AL:PA</th>
<th>AL:RA</th>
<th>FD(s)</th>
<th>AB</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>244%</td>
<td>208%</td>
<td>125</td>
<td>1100</td>
<td>25.15</td>
</tr>
<tr>
<td>9</td>
<td>269%</td>
<td>204%</td>
<td>127</td>
<td>1479</td>
<td>26.25</td>
</tr>
<tr>
<td>6</td>
<td>317%</td>
<td>189%</td>
<td>295</td>
<td>4507</td>
<td>60.03</td>
</tr>
</tbody>
</table>

**Discussion**  This part of the experiment measures how physics load is handled by the simulator when the number of physical entities increases. Observing the results in Table 3.1, from a period of 12 seconds to 9, the CPU load increases, but the fall duration remains nearly the same. This indicates that the load was increased, but the physics actor was capable of processing the updates in adequate time. However, from 9 seconds to 6, the fall duration increases by a factor of 2.32. The average number of balls also increases by a factor of 3.04. The CPU load increased only by a factor of 1.18. The increased fall duration and average number of balls are evidence that the physics actor is overwhelmed, and is dilating time. Dilation of time occurs when the simulation can no longer run at real-time speed, and the simulated world runs slower compared to real-world clock time. The average CPU increased to the limit of the simulator’s parallel processing capability, and can no longer process the scene in time.

With the results of Table 3.1, we can also make a correlation between load and error. From 12 seconds to 9, the increase in the error is small, but still present. From 9 seconds to 6, the error is more than double. We also see that the error becomes much larger when the simulator no longer has hardware resources to manage the workload, and resorts to dilating time as result. Consequently, *we show that increased load is associated with increased error in the end distribution.*
Part 2: Space Partitioned Simulation

The second part of the experiment is to measure the improvement of microcells over DSG. We have determined that increasing the load on the physics actor to the point where it is overwhelmed causes the fall duration and error to increase significantly. We fix the dropper period at 6 balls per second, where the load exceeds one physics actor. We divided the second part into 3 scenarios:

1. **No space partitioning**: This is the base comparison scenario, extracted from part 1. Figure 3.13 represents the subscriptions in this scenario.

2. **Active subscriptions**: Two physics actors partition the Scene in half, splitting all 4 galton boxes. Figure 3.14 (left) shows the subscriptions of both actors in this scenario.

3. **Active and passive subscriptions**: Two physics actors partition the Scene, but each physics actor has one additional column of passive microcell subscriptions. Figure 3.14 (right) shows the subscriptions of both actors in this scenario.

The first scenario serves as the base scenario for comparing the different scene partitioning approaches to load balancing. The second scenario determines the benefits, if any, of dividing the physics load in two different processes partitioned by space. Finally, the third scenario adds passive microcell subscriptions at the border of the partition, which we hoped would improve the behavioral error.

**Results**  Table 3.2 shows the results of part 2 of the experiments. The average load and memory usage for each physics actor is shown. All the other measures are the same as in the first part.

We used no partitioning with a period of 12 seconds as the base scenario to generate the expected normal distribution. Figure 3.15 shows the resulting and expected distribution of
Figure 3.14: Topology and subscriptions of actors in Scenarios 2 and 3 of the second part of the experiment. Each horizontal group of bars represent a galton box. Microcell grid not to scale.

balls for Scenarios 1, 2, and 3. The sum of all the measured and expected values of the normal curves match the number of balls in all three Figures.

Table 3.2: Part 2: DSG-M with space partitioning, at a period of 6 seconds per ball. **AL**: Average load (CPU%), **M**: Process memory in megabytes, **PA1** and **PA2**: Physics actor 1 and 2, **RA**: Root actor, **FD**: Average fall duration of balls in seconds, **AB**: Sum of PA1 and PA2 average number of balls, **RMSE**: Root-Mean error.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>AL:PA1</th>
<th>AL:PA2</th>
<th>M:PA1</th>
<th>M:PA2</th>
<th>AL:RA</th>
<th>FD(s)</th>
<th>AB</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>317%</td>
<td>—</td>
<td>1910 MB</td>
<td>—</td>
<td>189%</td>
<td>295</td>
<td>4507</td>
<td>60.03</td>
</tr>
<tr>
<td>2</td>
<td>304%</td>
<td>246%</td>
<td>765 MB</td>
<td>719 MB</td>
<td>348%</td>
<td>265</td>
<td>1624</td>
<td>175.88</td>
</tr>
<tr>
<td>3</td>
<td>305%</td>
<td>279%</td>
<td>885 MB</td>
<td>900 MB</td>
<td>305%</td>
<td>126</td>
<td>3286</td>
<td>91.75</td>
</tr>
</tbody>
</table>

**Discussion** In the second part of the experiment, we measure the effects of microcell partitioning on DSG. We have two goals: first, we wish to verify the improvement in scalability by checking that the load was divided with microcell partitioning. Second, we wish to assess whether the behavioral error of Scenario 2, where we use only active subscriptions, can be reduced by adding a passive subscription to microcells at the border.

We recall that the average load at a 6 second period for Scenario 1 was already overwhelmed,
Figure 3.15: Expected and resulting distributions for Scenario 1 (no partitioning), Scenario 2 (active subscriptions only), and Scenario 3 (passive subscriptions included).

As indicated by a longer fall duration and a higher error, as seen in Table 3.1. Figure 3.15 shows the resulting distribution of the 3 scenarios of this experiment. In Table 3.2, the average loads for Scenario 2 and 3 are similar to Scenario 1, showing that in all scenarios, CPU resource is scarce. By looking at the fall duration however, we find evidence supporting our expectation for our first goal. Scenario 1 had a 295 seconds fall duration average. By partitioning the space into two, with active subscriptions, Scenario 2 had a decrease in the fall duration to 265 seconds. Scenario 3 decreased the fall duration even further to 126 seconds. Both partitioned scenarios were successful in reducing the fall duration, and hence, in improving load balancing of the overwhelmed physics actor on Scenario 1.

In addition to a high CPU load, Scenario 1 allocated a considerable amount of memory, 1.9 gigabytes. In Scenarios 2 and 3, each physics actor used less than half of the memory of the single physics actor in Scenario 1. The use of microcell partitioning in Scenarios 2 and 3 divided the memory usage over the single partition in Scenario 1.

As for distribution errors, Table 3.2 shows Scenario 2 with a high RMSE. Objects near the border suffer imprecision when being crossed to another actor. By looking at distributions for Scenario 2, we see this imprecision as spike in the distribution near the border. By using passive microcells, that error is attenuated significantly, with a reduction in the RMSE to nearly half. For Scenario 3, we see a bimodal distribution closer to the expected curve.
With a significant reduction in the error, we show that passive microcells improved microcell partitioning by reducing the error in the expected distribution.

### 3.2.6 Conclusion

We have looked into load balancing for virtual worlds, focusing on two dimensions: Scene and operation partitioning. In Scene partitioning, we partition the data and the state. In operation partitioning, we disaggregate the behaviors of virtual worlds.

In previous work on the DSG architecture, the separation of virtual world operations for load balancing was shown to be very effective. DSG-M is designed to use the microcell fine-grained Scene partitioning method with DSG’s distributed system with disaggregated operations. DSG-M effectively maintains the advantages of operation partitioning while adding the advantages of space-partitioning, with little loss of precision, typically at border crossings.

In the evaluation, we demonstrate our load balancing technique through the use of active and passive subscriptions through a physics experiment. DSG-M successfully balanced an overwhelmed physics actor by partitioning the space in two, and attenuated the errors caused by the partitioning through the use of passive subscriptions. DSG-M proved successful in executing a workload that was infeasible with operation partitioning alone, while maintaining acceptable accuracy.

### 3.3 Conclusions of design experiments

This chapter has presented two approaches to address two concerns: network scalability and state synchronization. RCAT focused on creating a separation of concerns between the data
synchronization layer, the application logic processing, and the client networking, addressing
the latter known $O(N^2)$ state propagation problem. The data layer of RCAT provided a
state synchronization mechanism that was efficient, but required a considerable amount of
effort from the simulation developer. To use RCAT’s data layer, the developer must design
and implement a mapper that will maintain data ownership across the multiple application
servers. The mapper is responsible for defining queries (such as space-partitioning filters)
and for handling object relocation. Even though the object manager provided some relief
in transferring objects between application servers, developing the mapper requires a fair
knowledge of architectural organization and distributed systems awareness.

The DSG architecture uses a much simpler state synchronization mechanism: maintaining
all state constantly synchronized. The toll of constantly updating other simulators is high
but feasible, as demonstrated in evaluations [113] and real-world usage of DSG for large
scale simulations [123]. Unfortunately, while DSG-M successfully improved performance of
DSG by adding space partitioning to actors, it also added an extra layer of complexity. The
illusion that all state was local can no longer be tolerated in DSG-M: actors must be fully
aware that the state is distributed and be capable of handling state ownership, crossings,
and non-availability of simulation data locally.

Despite DSG-M being successful in improving load balancing for distributed simulations, the
additional complexity of no longer having universally shared data introduced new complica-
tions. The object definitions in DSG-M were exactly the same across all actors. This meant
that a virtual object simultaneously had methods and properties for physics, for scripting,
and for client management aspects. These objects already span thousands of lines in Open-
Simulator – the virtual world platform DSG is based on – due to similar reasons, a mixture
of different aspects overpopulating the same object.

In software engineering, it is common to use object-oriented programming (OOP) inheritance
to specialize objects. However, inheritance is used to specialize only the object type, but not
the object instance. In DIS, there is an underlying assumption that some remote objects share identity, even though they are not the same from a programming language standpoint (i.e. pointers to the same memory area). For instance, a vehicle in the physics actor of DSG also need to exist in the script actor, and state updates made to one, should be reflected to the other. However, this entanglement is artificial: the DIS architecture is responsible for creating this illusion of identity. If that is the case, why must type definitions be the same on both actors? Intuitively, each actor should only be concerned with a subset of properties and methods of the object that may or may not overlap.

Using the same type definitions across all actors inevitably would lead to bloating of properties and methods. Addressing the complexity of handling shared object definitions became the next topic of our research, in the design of the Collaborative Aspect-oriented DIS architecture (CADIS). CADIS is designed to encapsulate only relevant properties and methods in objects, while maintaining shared state synchronization seen in DSG-M. Furthermore, CADIS also encapsulates the concept of object collections: a type definition can group a collection of objects based on a certain characteristics that can be defined both statically and at runtime. The example used throughout the next chapter is a simulation with cars and pedestrians. Car and pedestrian are direct matches to OOP types, but a more interesting and useful type in DIS could be of endangered pedestrians. An endangered pedestrian is a pedestrian who will be run over by a car in the next steps of the simulation. Being able to define a group of objects with characteristics that remain entangled with its counter-parts (e.g. moving an endangered pedestrian should move the original pedestrian instance) is central to the design of CADIS. The next chapter describes the CADIS architecture and how it successfully isolates concerns of different aspects of the simulation.
Chapter 4

CADIS Architecture: Scaling Scope

4.1 Introduction

In software engineering, libraries play an essential role in pushing the boundaries of software development. Complex functionality that typically requires deep domain knowledge are easily reduced to understanding much simpler library interfaces – commonly known as APIs (Application Program Interfaces) – allowing software developers to focus and thrive uniquely in their own area of expertise. The latest advancements in software integration are Web APIs, that allow Web applications to integrate other independently developed applications and services in the Web. The end result is a rich collaborative environment for software development, where independent developers join efforts to build much higher complexity software.

The simulation domain is still behind in this aspect of software integration. Apart from the plethora of standards and commercial off the shelf (COTS) simulation platforms, integrating independently developed simulations may also require a great deal of documentation and technical specifications that often result in re-implementation of existing simulations. One
of the causes is the lack of separation between simulation logic and modeling. Some existing standards have successfully created such separation, as for instance the most widely used distributed simulation standard: the High Level Architecture (HLA) [87]. However, HLA and distributed simulations are widely considered too difficult to use for most simulation practitioners. As pointed out in an industry survey by Boer et al. [20], standards such as HLA can be overwhelming to simulation practitioners due to its large set of services and functionalities. Additionally, synchronization is treated at a lower level of abstraction, which has led to several research initiatives to create higher level abstractions (see Section 4.3 for examples). It is not that HLA is an inadequate solution, but rather it is missing an important abstraction layer that is commonplace in the universe of software development.

In a previous experience integrating real-time simulations [180], it became apparent that integrating the data models of independent simulations was difficult for two reasons: first, a simulated entity shared between simulations became bloated, as it collected the behaviors and properties that each simulation was responsible for processing. Second, most recent software development is done using object-oriented paradigms, and standards like HLA offer no support for matching events and object updates to programming language objects. Our answer to both concerns is CADIS (Collaborative Architecture for Distributed Interactive Simulations). CADIS focuses on distributed real-time simulations, but several parts of CADIS can also be applied to distributed simulation in general.

4.2 Overview of the CADIS Approach

In the introduction, we discussed how our previous research led us to two major challenges in integrating simulations: bloated and complex data models and mapping of events and updates to objects in object-oriented programming (OOP) languages. We have designed two frameworks that work together to address these challenges. The first is programming
model framework called **Predicate Collection Classes (PCC)**, which enables objects to be reclassified during runtime while maintaining an identity entanglement with its original class. A simple example – used throughout this article – is a Car and a ActiveCar objects. A Car represents a common vehicle driven in cities. An ActiveCar is a Car object whose velocity property is bigger than 0. PCC maintains an entanglement between an object and its runtime derivatives based on a primary key, thus allowing objects to be defined specifically to the need of a particular simulation. This means modifying properties of ActiveCar imply the same modifications in the corresponding Car object. We discuss PCC in Section 4.4.

The second is a framework called **Spacetime** that drives the simulation forward while maintaining PCC objects updated and classified on regular time intervals. Listing 4.1 shows an example of two different simulations. TrafficSimulation generates cars and updates their position over time. PedestrianSimulation generates pedestrians that walk in opposite direction to those cars. The goal of this example is to show how CADIS handles interaction between two simulations that might have been developed independently.

The architecture of CADIS can be seen Figure 4.1. The **Simulation** is the simulation developed by the user. **Frame** is a native-language library that the simulation uses to access shared objects of the distributed simulation. The **store** is responsible for synchronizing state across all simulations and maintain predicate class objects updated. The store and frame are both implemented as part of the spacetime framework, which in turn uses the pcc framework.

The main rationale and features of CADIS are listed in the follow observations, based on the example simulation source-code of Listing 4.1.

**PCC:**

- **Predicate Collection Classes:** PCC allows predicate class derivations of OOP types
class TrafficSimulation(IApplication):
    TICKS_BETWEEN_CARS = 10
    def initialize(self):
        for i in xrange(2):
            self.frame.add(Car())
        self.ticks = 0
        self.cars = self.frame.get(Car)
    def update(self):
        if self.ticks % self.TICKS_BETWEEN_CARS == 0:
            inactives = self.frame.get(InactiveCar)
            if inactives != None and len(inactives) > 0:
                inactives[0].start();
            for car in self.frame.get(ActiveCar):
                car.move()
        self.ticks += 1

@Producer(Pedestrian)
@Setter(Car, Pedestrian)
@GetterSetter(StoppedPedestrian, Walker, CarAndPedestrianNearby)
class PedestrianSimulation(IApplication):
    TICKS_BETWEEN_PEDESTRIANS = 10
    def initialize(self):
        for i in xrange(5):
            self.frame.add(Pedestrian())
        self.pedestrians = self.frame.get(Pedestrian)
        self.ticks = 0
    def update(self):
        if self.ticks % self.TICKS_BETWEEN_PEDESTRIANS == 0:
            inactives = self.frame.get(StoppedPedestrian)
            if inactives != None and len(inactives) > 0:
                inactives[0].move();
            endangereds = self.frame.get(CarAndPedestrianNearby)
            for car_ped in endangereds:
                car_ped.move()
            for pedestrian in self.frame.get(Walker):
                pedestrian.move()
        self.ticks += 1

Listing 4.1: Car and Pedestrian simulations in CADIS.
that represent collections of objects. Using the recurring example, an ActiveCar is a collection of Car objects whose velocity are not null or 0. Other derived types used in this example are InactiveCar (Car objects with no velocity), StoppedPedestrian (Pedestrian objects in initial position), Walker (Pedestrian objects not in initial position), and CarAndPedestrianNearby (Car near Pedestrian objects).

- **Relational Algebra:** In PCC, relational algebra – similar to the one used in SQL languages – can be used in PCC types. A clear example is CarAndPedestrianNearby, which is JOIN operation on Car and Pedestrian, querying for cars that are near pedestrians, as seen in Listing 8.2 lines 38–54. Other allowed operations are subsets (like in ActiveCar), projections, parameterized collections (queries with parameters), and joins. More details in Section 4.4.3.

- **Derivative Type Construction:** PCC is used to represent collections of objects that may enable different behaviors than its relatives. A PCC type can add or remove properties and functions of the original type, constructing a class that has solely the necessary functionality for simulating its behavior. For instance, in Listing 4.1, lines 40 and 42, the same call to move() is made for a pedestrian, but in line 40 it will move the pedestrian to safety, whereas in line 42, it will move the pedestrian towards its normal direction.
Spacetime:

- **Spacetime Framework**: Simulations are called by the Spacetime framework in regular intervals. These calls are made to the simulation’s update function (Listing 4.1: lines 12 and 33).

- **Native-Language Library**: Simulations have access to data through Frame, the native-language library that handles the synchronization of objects between simulations. The `self.frame` variable used throughout the example is a reference to the Frame library, which is passed in the constructor of `IApplication` simulation classes (omitted in the listing).

- **Frame API**: offers two vital functions to the simulation: adding new objects (lines 8 and 29) and retrieving existing objects (Listing 4.1: lines 10, 14, 17, 30, 35, 38, and 41).

- **Interest Management**: Interest management in CADIS is declarative. Listing 4.1, lines 1–2 and 21–23, describe respectively the interest management declarations of TrafficSimulation and PedestrianSimulation. In this example, TrafficSimulation produces Car objects, and subscribes to updates for InactiveCar and ActiveCar objects. The PedestrianSimulation produces Pedestrian, updates Car and Pedestrian, and fetches and updates StoppedPedestrian, Walker, and CarAndPedestrianNearby.

- **Data Store**: Spacetime allows objects to be stored in different locations (called stores). In line 1 of Listing 4.1, Car is being fetched specifically from localhost. Others will default to an address specified in a configuration file.

Listing 4.1 provided a simple example to demonstrate the main features of CADIS. The next sections will cover in detail PCC (Section 4.4) and Spacetime (Section 4.5), which form the architecture of CADIS, followed by a performance evaluation with a more complex
urban simulation scenario (Section 3.2.5), and a usability study of teaching computer science students to use develop a collaborative distributed simulation with CADIS (Section 5.4).

4.3 Related Work

4.3.1 Predicate Classes

Some of the inspiration for PCCs, including the name, comes from predicate classes [34]. The idea behind predicate classes is for objects to be dynamically classified, taking new/different behavior as they change state. Objects that satisfy predicates specified in predicate classes automatically become instances of those classes; when they stop satisfying those predicates, they stop being instances of those classes. This is very similar to PCCs, but there are some important differences.

The major difference is that PCCs, as the name indicates, pertain to collections of objects, rather than to individual objects. This difference changes the focus and the capabilities of the basic idea substantially. In the case of simple predicate classes, the programmer simply states the predicate to be satisfied (e.g. a car whose velocity is zero), but there are no handles for collections of objects that satisfy those predicates at any point in time. The PCCs’ focus on collections not only exposes these handles but also enables the full expression power of relational operations on collections, such as subsetting, projection, cross product, etc. Simple predicate classes express implicit subsets only: subsets, because predicates on field values constrain the state of the parent objects; implicit, because there is no handle for that subset.

More importantly, one of the goals underlying simple predicate classes is to always ensure the satisfiability of the predicate. For example, given a normal class buffer and a predicate class
empty buffer, if the last element is removed from a buffer object, that object immediately\(^1\) becomes an instance of the empty buffer predicate class; similarly, if an element is added to an empty buffer object, that object immediately stops being an instance of the empty buffer predicate class. Classification is always consistent with the state of the objects. That is not a goal of PCCs. PCCs classify the objects at some point in time, at which point the predicate is guaranteed to be satisfied; but once included in the collection, the state of the objects may later change, possibly becoming inconsistent with the predicate that placed them in the data frame. That is not just acceptable: it is a desired semantics. PCCs are meant to hold a fixed collection of objects whose state can change. Take, for example, the case of a collection of non-empty buffer objects; if during subsequent processing all elements are removed from a given buffer object in that collection, we still want that buffer object to be in the collection, even though it is empty; we don’t want it to suddenly disappear because it became empty. So the semantics of predicates in PCCs is quite different from that of predicates in simple predicate classes: always true (in the case of simple predicate classes) vs. true at the time of data frame creation (in the case of PCCs).

The relaxation of satisfiability is also what makes it possible to implement relational operations in practice, not just subsetting. Unlike subsetting, that looks only at internal state of the objects, joins (cross product) and parameterization pertain to combinations of objects. Take for example, a join between cars and their owners. If at some point in the framed computation the car ownership changes from one person to another (or to no one), we would need to search combinations of a car and persons again to check whether the resulting join object satisfies the predicate. Strict satisfiability of predicates for objects involved in join operations, as well as parameterizations, would be prohibitive to implement.

\(^1\)“Immediately” here includes lazyness, i.e. not necessarily instantly but as soon as classification is needed.
4.3.2 Other Class-Instance Associations

Besides predicate classes, the OOP literature presents a considerable number of ideas aimed at making the instance-class relationships more flexible. We describe some of them here, and how they relate to PCCs.

Fickle [52] includes another idea for dynamic object reclassification that is not based on predicates, but on explicit reclassification by the programmer. The Fickle language provides a construct to reclassify instances of special "state" classes that can be used by programmers. These special classes, however, cannot be used as types for fields of parameters, as that would violate type safety. The main difference between PCCs and this older work is, again, the focus on collections rather than on individual instances. Additionally the Fickle reclassification construct is not declarative but imperative in nature. In contrast, PCCs are defined using declarations (the predicates).

First introduced in Flavors by Moon [130], and then in CLOS, mixins (abstract subclasses) are a powerful way of combining object behavior, as they can be applied to existing superclasses in order to create a related family of modified classes. Bracha and Cook introduced mixin-based inheritance [23], a form of class inheritance formulated as a composition of mixins. Mixin layers [161] are a form of decomposition that targets the encapsulation of class collaborations: each class encapsulates several roles, where each role embodies a separate aspect of the class's behavior. A cooperating suite of roles is called a collaboration. Mixins are only vaguely related to PCCs in that they make reuse of behavior more flexible than inheritance, allowing objects to be given several different roles. But the classification is still static, meaning that it is established before any instances are created. In contrast, PCCs serve to reclassify objects at runtime.

Bertino and Guerrini proposed a technique that allows objects to belong simultaneously to multiple [most specific] classes [13]. The motivation was data modeling situations in
which a single instance (e.g. a person) is naturally associated with multiple classes (e.g. student, and female). Although similar to multiple inheritance, the technique proposed in that paper aimed at avoiding the proliferation of subclasses that are simple combinations of other classes. This work built on the idea of mixin-based inheritance [23], and it predates traits [137, 153]. Traits are another way of reusing behavior. PCC instances do not have traits, but rather they are instances associated with a single class, that take the state from existing objects.

Finally, virtual classes [55,122], dependent classes [70], and generics [24,127] are mechanisms to parameterize classes. That work is vaguely related to PCCs in that it is particularly useful for collection classes such as lists, sets, etc. But the purpose of parameterized classes is quite different from that of predicate [collection] classes: the former targets the generalization of type definitions (types parameterized on other types), whereas the latter targets the association between instances and their classes.

4.4 Predicate Collection Classes

Predicate Collection Classes is a programming model based on the concept of collections of objects. While OOP has traditionally been concerned with abstract data type relationships (e.g. object $A$ is related to object $B$), PCC provides the tools to build relational operations between collections of objects (e.g. collection $C_A$ of objects of type $C$ behave in a certain way). It is vital to delineate the motivation behind PCC in simulations. This chapter will use the previous example of car and pedestrian simulations to illustrate both the existing concerns of modeling simulation entities as abstract data types, and to show how PCC can address these concerns.
Motivation

The main motivation for PCC is to handle the explosive complexity of data models when different categories of simulations interact to represent the same simulated entities. A car, for instance, can start being modeled as an object with no physical dimension, containing only a position and a velocity. As the model becomes more complex, it starts to add dimensions (i.e. length, width, height) and acceleration. Then, what moves the car can be modeled as an engine, axes, and pistons, followed by an electrical system, interconnected by a Controller Area Network protocol. Cars running in wet roads need to be simulated differently, so does heavier cars, and cars with anti-brake systems and electronic stability controls. In the end, the abstract data type Car will contain hundreds of properties that define multiple aspects of the car, whether they are dynamic (e.g. car on a wet road, has electronic stability) or static (e.g. electrical and mechanical subsystems).

For static aspects of a type, OOP is helpful in allowing an object to reference other objects (e.g. a Car references a Mechanical and Electrical data types). However, if aspects overlap – as they usually do in the real-world – it is difficult to organize an abstract data type to properly represent different aspects of the same entity. In the car example, both electrical and mechanical subsystems interact with overlapping properties of the car that are not entirely clear. In a hybrid vehicle, for instance, the electrical subsystem directly interacts with mechanical parts. What would be ideal for modeling this car was to fetch different abstract data types for MechanicalCar and ElectricalCar, that would contain solely the relevant properties and behaviors for this type. This is not possible with traditional OOP implementations, but there have been solutions that address such concern in research, such as mixins (see Section 4.3).

Dynamic aspects are non-existent in most implementations of OOP languages, possibly due to performance concerns. A dynamic aspect is an aspect that depends on runtime values, like
in the ActiveCar example, which relies on velocity values during runtime. Types that rely on dynamic aspects to evaluate are called dynamic types. Every time a dynamic type is used, its predicate must be evaluated. In a normal source-code execution, this could potentially slow down computation by many orders of magnitude, as with each statement in the source-code, a dynamic type could be changed. In time discrete simulations, however, there is often time for dynamic types to be resolved between events or time steps. As discussed in the car example, one of the major benefits of dynamic typing is modularity. The conditions for a collection of objects to be of a dynamic type are hidden in the type definition, while the semantics of the dynamic type can often be easily understood by humans. Our initial example of CarAndPedestrianNearby is simple for a human to understand, while the algorithm to resolve whether a car and a pedestrian pair should belong to this collection may not be trivial.

With improved modularity from information hiding, it becomes possible to reuse simulations that were independently developed. CADIS objective is to achieve easier interoperability and reuse of simulations than exists today. Even with standards like HLA, it is still difficult to reuse simulations as much of the logic driving the models is buried within the simulation code. By better representing abstract data types in a manner that is easily understood by human-beings, we aim to simplify the process of integrating simulations developed by domain-experts in distinct fields.

This chapter will cover PCC from the standpoint of simulations, but for the details of the programming model grammar, we recommend reading our original article on PCC. We divide PCC in two separate functionality groups: object reclassification (Section 4.4.2) and relational algebraic operation (Section 4.4.3).
4.4.2 Object Reclassification

Object reclassification is what allows CADIS to subtype simulated entities into its multiple aspect definitions. To avoid performance bottlenecks, PCC requires explicit execution points for creation of dynamic types. Listing 4.2 shows an example of how ActiveCar objects can be created and used.

```python
while (True):
    foreach aCar in pcc.create(ActiveCar, cars):
        aCar.Move()
```

Listing 4.2: Creation of ActiveCar objects using pcc.

A create call needs a universe of existing objects to resolve new types. In the example, the cars variables is assumed to be a universal collection of all existing cars, similar to the universe concept in mathematical set theory. When create is executed, classification is evaluated as if the universe is in a frozen state. After classification is performed, modifying objects do not trigger a new reclassification. Thus, if a modification caused an object to longer belong to that collection, it will remain in the collection until a new classification process begins. In the same example, if instead of moving the car stopped, it would remain as part of the result collection of create(ActiveCar), until a new create function was called. This behavior is by design, to avoid concurrency issues and unexpected behavior when iterating over a collection of objects and modifying their state.

Another consideration is whether PCC types are created by copy or by reference. By default, PCC types are created by reference. This implies that all related types share the same properties, which are referred as dimensions in PCC. Modifying a dimension would immediately cause all other related types containing the same dimension to also see an update. This may have unexpected consequences, particularly in multithreaded environment, but better reflects the purpose of object reclassification: to maintain an entanglement with related types. The other alternative is to create a copy of the objects in the universe, and
then reclassify the types. However, in order to achieve the desired effect of entanglement, any changes done to the copies would have to be merged back. If opting to use PCC with pass-by-copy, the default is not to merge back changes. The merging back on pass-by-copy can be achieved with the use of dataframes.

A dataframe in PCC provides a context for the creation and execution of PCC types. It was previously mentioned that creating PCC types requires a universe of existing objects to evaluate from. The dataframe plays the role of organizing that universe: objects can be added and removed from dataframes, and a dataframe can be locked when being used. Listing 4.3 shows an example of using dataframes. The with clause is used to create a context: inside of the with clause, all objects are copies of the objects that were in the dataframe. Modifications done to objects within the with clause have no external effect until the with clause is over. When the execution exits the with clause, PCC gathers all the modifications done within that context and merges back the changes to the original objects in the dataframe.

```python
def MoveActiveCars(cars):
    # Moves active cars every 300 ms
    while True:
        with dataframe(lock) as df:
            acs = df.add(ActiveCar, cars)
            for car in acs:
                car.Move()
            sleep(0.3)
```

Listing 4.3: Moving an ActiveCar.

In this snippet, a dataframe is used to fetch all ActiveCar objects, then call the Move() function for each ActiveCar, all within a loop of 300 ms intervals. The difference between this example the one in Listing 4.2 is twofold: first, the code within the with clause is using copies of the original objects in the dataframe; second, the dataframe itself uses a lock to prevent other threads of execution from modifying the dataframe while ActiveCar objects are being processed. If instead a lock is not used and multiple threads enter with dataframe
sections, they are – from a logical clock standpoint – considered simultaneous interactions with the state and there is no correct order of execution for the end result. Merging the state back to the dataframe is guaranteed to be thread safe by the PCC framework, but resulting modifications to objects could be merged in a non-wallclock-time order.

For distributed simulations, only the pass-by-copy with no locks can be used, since memory references and locks cannot be shared across a network. Spacetime uses a similar concept as dataframes to maintain network-based state synchronization, called a store. More details on Spacetime stores can be seen in Section 4.5.4.

### 4.4.3 Algebraic Operations

Predicate classes in PCC enables abstract data types to represent relationships besides inheritance. Borrowing from relational database concepts, data types in PCC can be treated like tables. A predicate class has dimensions in the same way a table has columns. A predicate class representing a collection of objects – like the ActiveCar presented in previous examples – is the equivalent of selecting multiple rows of a table. Additionally, every object in the PCC framework must have a primary unique identifier, commonly known as primary key in relational databases.

As PCC and relational databases share the same semantics, it is possible to adapt the same powerful relational algebra used for relational databases in PCC. In this section we will present the set of operations that are currently supported by the PCC framework.

**Subset**

A subset is a collection of objects of a data type that conform to a predicate. In SQL, it is the equivalent of a query with a WHERE statement. An ActiveCar in SQL would be:
SELECT * FROM cars WHERE velocity > 0

A subset is entangled with its superset data type. The implication is that modifying a subset object should also modify the superset object sharing the same identity. Another consequence is that subsets share identity with its superset relative. In the our example, modifying the position of an ActiveCar will also update the position of Car, and active_car == car will return true if both objects have the same primary key identifier.

Projection

A projection is a collection of objects of a data type with only a selected number of dimensions. Projections are used to construct predicate classes that hide unnecessary properties. In SQL, it is the equivalent of choosing rows in a SELECT statement, as in

SELECT velocity FROM cars

A predicate class equivalent could be named CarVelocity, and would contain only the primary key and the velocity dimension of Car objects. As with subsets, projections are also entangled with its related data type.

Cross-Product (Join)

Join defines a collection of pairs of objects from two data types that conform to a predicate. From our CarAndPedestrianNearby example, it is the equivalent to a SQL join call:

SELECT * FROM cars JOIN pedestrians
ON pkey WHERE abs(cars.pos - ped.pos) < 10
Join types are different from projection and subsets with regards to identity. The collection of pairs of objects do not share an identity entanglement with either of its related types. Thus, a `car_near_pedestrian == car` and `car_near_pedestrian == pedestrian` would return false. Modifying an object typed as a join does not modify its related types, but since a join contains a reference to both objects belonging to the join, modifying those references will naturally modify the related objects. In pseudo-code, a `car_near_pedestrian.car.position = 0` will modify the position dimension of the car in the `car_near_pedestrian` join object.

A consequence of the join’s lack of shared identity is that a join type may have additional properties besides its reference to the types being joined. Because of this characteristic, joins cannot simply be built from the data of types it joins. Where subsets and joins need only a list of primary keys to recreate a collection, a join may need additional information. This difference, and its impact in the performance of spacetime, is discussed Chapter 5.

### 4.5 Spacetime

The Spacetime framework is named as such for introducing the notion of time to PCC. In the PCC framework, the evaluation of predicates happen at specified points in the source-code execution, such as `create()` and at the `with` clause. Spacetime defines regular intervals of dynamic type evaluation with a 3 step loop: pull, update, and push. The pull phase requests updated objects, the update phase runs the simulation code, and the push phase merges back changes done to objects. It is equivalent to the `with` clause used in PCC: the `with` statement is the pull, the code within the `with` clause is the update, and leaving the `with` clause is the push.

This section covers the Spacetime framework in detail. Section 4.5.1 discusses the interest management capabilities of Spacetime. Section 4.5.2 describes the time flow design of space-
time. Section 4.5.4 presents the store, responsible for maintaining PCC objects up-to-date and to merge back modifications done by simulations. Finally, Section 4.6 discusses the performance concerns and optimizations of Spacetime that make CADIS a viable real-time distributed simulation platform.

4.5.1 Interest Management

Section 4.4 showed how objects can be created, deleted, and updated using the PCC framework. The responsibility of spacetime is to maintain a mapping of what simulations produce, delete, and update. This mapping is often referred to as interest management, and is typically represented by publish-subscribe pattern, where some simulations declare themselves as producers of data, and others as subscribers. Spacetime uses the same analogy, but with one major difference: interest management in spacetime is declarative, not imperative.

In HLA, for instance, simulations are expected to inform the runtime-infrastructure (RTI) of what objects and properties it intends to publish and subscribe during execution. It is commonly the first step of an HLA simulation (called federates) to inform the RTI of its publish and subscribe desires before starting the simulation. In Spacetime, the underlying assumption is that publishing and subscribing mapping is not a runtime decision, but rather one that is implicitly defined in the purpose of the simulation and the data types it uses. For example, a TrafficSimulation will always be responsible for producing cars, no matter what happens to the runtime execution. Naturally, this is not a universal truth: any simulation could – by design – change its publish-subscribe patterns if the developer wishes it so. However, CADIS is an aspect-oriented architecture, enforcing that simulations should always fulfill an aspect of the distributed simulation. And an aspect should have pre-determined requirements and outputs of data type.

Instead of publish-subscribe, our spacetime framework uses the terms producer-getter-setter.
to specify interest management needs. Interest management declarations are performed as annotations to the simulation, as seen in example Listing 4.1, lines 1–2 and 21–23. The available interest management declarations in Spacetime are:

- **Producer** *(types)* The simulation creates objects of the specified types.

- **Getter** *(types)* The simulation is interested in creation and updates of objects of the specified types.

- **Setter** *(types)* The simulation updates objects of the specified types.

- **GetterSetter** *(types)* The combination of Getter and Setter declarations.

- **Tracker** *(types)* The simulation is solely interested in creation and deletion (not updates) of objects of the specified types.

To improve scalability, spacetime interest management declarations can specify the host data store that a particular type is associated with, avoiding centralization of data requests. Listing 4.1, line 1, shows an example of a producer statement that specifies the data store as being localhost. Further details about stores can be seen in Section 4.5.4.

### 4.5.2 Time Flow

Spacetime operates with a 3 step loop: pull, update, and push. These steps are necessary to create established points in time where state is considered frozen, otherwise it would be unfeasible to perform consistent relational algebra on a changing dataset. Relational databases often lock tables in transactions to avoid the exact same issue.

The implication for spacetime is that simulations must be time-discrete: execution can only occur at certain points in time. For simulation, this translates to time-discrete simulations.
steps. Currently, spacetime only supports fixed time-intervals, where every interval of clock-time $t$ the 3 operations are performed. However, it should be possible to support event-driven simulations, with the caveat that every event triggers a push and pull. With lookahead, every time the lookahead number of events have been reached, would trigger a push and pull. For a fast paced event system this could turn out to be unfeasible due to severe low performance of the simulation. For the purpose of this article and the current implementation and design of spacetime, only fixed-time interval simulations will be considered. Fixed-interval simulations are particularly appropriate for real-time human-in-the-loop simulations like virtual environments.

The fixed-interval of execution is configurable in spacetime, and does not need to be the same for all simulations. If a fixed-interval simulations take too long to process, it may overrun its allotted time. Spacetime does not interrupt the execution, but can trigger a callback to inform the allotted time is over, while providing the actual time interval since the last call to the simulation’s update function. This way a simulation can be aware that a non-fixed-interval amount of time has passed, and it should take it into account to process the next call.

### 4.5.3 Frame

Frame is the client-side library for CADIS, allowing simulations to fetch data types, and abstracting most of the inner-mechanics of PCC and Spacetime. The frame library is part of the spacetime project, and is currently available in Python and in limited functionality in C#. The frame API has 2 essential operations: `attach` and `get`.

The attach call is used to attach the user’s simulation to frame. This is a necessary step, so frame can call the simulation’s update method at regular intervals. Frame also calls initialize before the simulation starts, and shutdown when its done. All these methods are specified
in the interface IApplication, used in Listing 4.1, lines 3 and 24.

The get call allows the simulation to fetch collections of objects. The direct use of get will fetch all objects of a certain data type, like fetching all ActiveCar objects in Listing 4.1 line 17. Other uses of get is for fetching a specific object (by passing its unique id), and for fetching modified objects since last update, returning objects that are new, modified, or deleted since last update. This is particularly useful in simulations, since state changes are often triggers for actions. Typically, this would be an event, but in CADIS the use of events is discouraged. Instead, CADIS encourages the use of state change as an event. For example, when using an event like CarStartedMoving, the simulation can instead be a getter for ActiveCar, and get objects that are new since last update.

4.5.4 Store

Objects in Spacetime are tracked by a an implementation of a dataframe called store. The store – like the dataframe presented earlier – is responsible for keeping track of PCC objects during the execution. However, unlike the dataframe, the store uses the PCC framework to transparently hide mechanics of predicate classes from the developer. Dataframes assume a shared memory environment, and make use of in-memory locks and object references to fulfill simulation data requests. In contrast, the stores assume a distributed simulation environment, using pass-by-copy to fulfill data requests and handles merging back state updates. Requests and responses follows the paradigm used in the Web: a RESTful API that processes a collection of verbs for handling remote resources. The permitted verbs are a subset of the ones found in HTTP (Hypertext Transfer Protocol): PUT, GET, POST, DELETE.

With the expectation that requests may be performed over a network, the store maintains a type interest mapping per simulation. Each simulation must register with the store before
starting its pull/update/push process. The registration will send to the store which types
the simulation will be fetching and updating (see Section 4.5.1 for more details). A pull
becomes a GET on a data type, for instance: GET /TrafficSimulation/Car. A push
is similar: if a new object is added, a PUT operation could be PUT /TrafficSimulation/
Car/1 { [car_data] }. Updating an existing object would be a post: POST /TrafficSimula-
tion/Car/1 { [update_map] }. Deleting an object would be simply DELETE /TrafficSimula-
tion/Car/1. While this approach is an elegant RESTful way of handling
data requests, it does not perform well under the well known HTTP protocol over TCP/IP,
since multiple requests could mean opening multiple socket connections. Thus, POST also
handles a collection of new, updated, and deleted objects of multiple types so that a single
request can be made in lieu of several.

Both the pull and push requests requires care when handling requests from multiple simula-
tions. However, a global lock per request is not necessary. Instead locks are done per object
type, only while Spacetime calls the PCC framework to evaluate data types.

4.6 Performance

The data types expressiveness provided by PCC will naturally have a negative impact in the
performance of CADIS as a simulation platform. In particular, moving from the memory
model of PCC to the networked infrastructure of spacetime requires a great deal of optimiza-
tions to reduce the amount of data that needs to be exchanged between the simulation and
the store. In PCC, the default configuration is passing objects by reference. This is done
very fast, since only pointers need to be copied over to and from the simulation. Further-
more, the data entanglement of dependent types occurs naturally, since all related objects
hold pointers to data structures in the same memory location. In contrast, passing objects
through a network requires creating a serialized version of the entire object, passing it over
a network, then deserializing and rebuilding the object on the other side. Passing entire objects at every time step between simulation and store will not scale.

This Section presents a series of optimizations to spacetime and PCC with the goal of reducing the total processing time (i.e. push, simulation update, and pull) of networked simulations in CADIS.

4.6.1 Pushing only modifications

The simplest optimization is having simulations push only modifications to shared objects. Such an approach is not as simple as it appears, since PCC dependent types have entangled data structures. To achieve this functionality, the PCC dataframe has to be divided in two parts: a client and a server version.

In the client version, the dataframe is responsible for intercepting changes to object properties and recording them. These changes are stored as simple dictionaries of types to primary keys, where each primary key maps to another dictionary of dimension to value. Listing 4.2 shows an example of a Car object that had its Velocity dimension modified. If the modification is done on a dependent type (e.g. ActiveCar), the related set type (Car) is the one marked in the set of changes, which are collected during the update phase of the simulation. In the push phase, these changes are sent to the server.

In the server, objects are stored in a similar dictionary of type to object ID, and object ID to serialized object. Notice that the object is not rebuilt on the server, but rather stored in its serialized version received from the client. This skips another source of heavy computational workload – serializing and deserializing data to and from objects. The server is solely responsible for calculating dependent types, checking the predicates and creating the object collections for each request. When the server receives a pull request that includes
a dependent type, the server uses the newly updated information to create the respective
collection of objects of that type. Each object that resulted in a positive result of the
predicate gets added to a list to be sent to the client. After all the list of base and dependent
types are calculated, a dictionary of types to list of objects is created and sent back to the
client.

In this optimization, the amount of data sent is greatly reduced, since only modifications to
set types are passed from simulation to store. However, the pull still fetches all objects from
the store.

4.6.2 Pulling only modifications

The next optimization is for simulations to pull only the modifications from the store. While
conceptually similar to the push scenario, the pull requires that the server is constantly
aware of what objects each simulation pulled at the last request. The server keeps a cache
(i.e. a simple dictionary of type to state update) for each connected simulation. When a
simulation pushes changes, the server applies the changes to its object map and adds these
modifications to the caches of all other simulations. When a simulation requests a pull,
it pulls all the aggregated modifications sent by each client. When the pull is finished, the
server clears that cache so it may keep track of new modifications made by other simulations.

This approach of pushing and pulling only object modifications greatly reduces the amount
of exchanged data between simulation and store. However, pulling dependent types presents
a challenge. Since dependent types are calculated at every pull, maintaining a record of
changes is not only a matter of aggregating changes from simulations. Thus, at this level of
optimization, dependent types are still pulled as entire list of objects.
4.6.3 Pulling dependent type modifications

Dependent types represent collections of objects, and have a different semantic than set types. Take the example of ActiveCar: a Car may be an ActiveCar in one time step and be an InactiveCar in the next, since its Velocity can change at any time. However, the data required to build an ActiveCar object is the same data required to build a Car. Thus, projection, subset, parameter, and join data types can be simply represented as a collection of primary keys. In the Car example, ActiveCar can simply be a list of IDs (e.g. [1,2,3]), whereas Car is an actual object with dimensions (e.g. Car.X, Car.Y). This optimization takes advantage of this feature to further reduce the amount of data sent over the network in a pull.

There is one complication to handling objects this way – a client may not declare interest in the set type of a dependent type. For instance, the client may be interested in ActiveCar but no in Car. Since relational types apply to other relation types, it is also possible that only some dimensions may be missing (e.g. subset of a projection). In these scenarios, sending the primary keys of Car objects would not be enough to build an ActiveCar object in the client side.

In order to pull only the necessary data, the way objects are stored in both the server and client had to be changed. Instead of using simple dictionaries that hold modifications made by simulations, a new dataframe implementation was developed, that operates in 3 modes: master, client, and cache. The client mode is used for the simulations, while the server uses both a master and cache. A master dataframe has one cache per simulation. When updates are received, they are applied to the store object maps, but also passed to each cache dataframe of each simulation. During this process, the dataframe determines how much data does the cache need to have to build the requested object. In the Car scenario, the cache dataframe would check whether the client has the dimensions necessary to build
an ActiveCar. If any dimensions are missing, they are added to the cache as well. When the client pulls, the cache serializes all the necessary dimensions and their values, and send it to the client. The client now has exactly enough information to create the dependent type.

In order to avoid an explosion of state, the dataframe caches hold references to the actual data stored in the object map. This way multiple caches may have a reference to the same dimension (e.g. ActiveCar.Velocity and Car.Velocity), but only one data value is held in memory.

4.7 Concluding Remarks

The use of PCC dependent types for distributed simulations shows a great promise in information hiding, through encapsulation of behavior using runtime state. However, as noted in Section 4.6, there is a great deal of complication in maintaining state synchronized over a network, and the entanglement of data in dependent types makes this synchronization even more of a challenge. Even with the proposed performance improvements, one should expect a performance degradation when compared to clean network protocols that send only the bare minimum data.

The next chapter is entirely dedicated to demonstrating that, with the performance optimizations seen in Section 4.6, CADIS is perfectly capable of running complex distributed simulations in real-time. First, a benchmark of each operation is shown, comparing the improvements made in each of the performances described previously. Then, we demonstrate that CADIS is a feasible solution in the actual scenario used throughout the dissertation – an urban simulation. Finally, the value of using CADIS over lower-level DIS protocols is shown in a usability study, where graduate students with little knowledge of simulations successfully developed a distributed interactive traffic simulation during 5 weeks of class.
Chapter 5

Evaluation of CADIS

Evaluating architectures for DIS systems is a challenging task, as the goals of such systems are often unclear. Some requirements are well known, such as available, partitioned, and consistent synchronized data between simulators. However, as those goals are conflicting, it is difficult to determine whether the architecture supports the right amount of each requirement for any class of simulations.

To tackle this challenge, the evaluation of CADIS is divided in 4 parts. The first part is a benchmark (Section 5.1) whose goal is to determine at the lowest level of functionality how scalable the framework is and how well it performs. The purpose of these benchmarks is to inform us, as well as simulation developers, what to expect when using the powerful predicate collection features offered in CADIS. The second part of the evaluation is feasibility (Section 5.2). The feasibility evaluation demonstrates that CADIS can perform a real-world collaborative simulation with sufficient performance to be useful. However, the feasibility analysis is used and built by CADIS developers, falling short in demonstrating its usefulness for simulation developers in general. The third part is an experiment in integrating HLA with CADIS (Section 5.3), determining whether CADIS can complement the existing features
of HLA with more expressive data models. The fourth and final evaluation is a usability study (Section 5.4), demonstrating how efficiently can software developers with little to no simulation background design a collaborative distributed simulation.

5.1 Benchmarks

The benchmark evaluation built for CADIS is, conceptually, the equivalent of a unit test in software engineering. The goal of the benchmark is to measure components and features of the CADIS architecture individually, with a cleanroom experiment approach. Specifically, CADIS offers powerful relational algebra operations that improve expressiveness of data types, but often incur a performance penalty of repeatedly reclassifying dynamic types at runtime. The benchmark evaluation in CADIS is a series of isolated experiments that measures the processing time of each simulation step in producing, updating, and pulling each PCC relational operation. For each operation, there are two graphs corresponding to two simulations: a producer and consumer. The producer (i.e. publisher) will generate updates of PCC objects that the consumer (i.e. subscriber) will retrieve. To provide a realistic simulation workload, 1000 objects are used as a reference for every scenario.

As mentioned in Chapter 4, when PCC was moved from a memory-based storage to a networked store, many of the previously used state synchronization techniques were no longer viable. For each benchmark shown in this section, there will be 4 different lines representing cumulative performance updates that enabled CADIS to be feasible in a networked environment. The first line is called baseline, and essentially follows the in-memory paradigm used originally for PCC – all relevant objects are reclassified, made available to the simulation for updates, then pushed back to memory. The second line and first improvement is called diff.push, which pushes back to memory only the changes as opposed to all objects. The third line is called diff.pushpull, and instead of pulling all the objects for the simulation
to use, it pulls only the modifications since last push. However, dependent types in PCC are not actual objects, but collections, and in this mode every pull requires recalculation and pulling all dependent types. Finally, the fourth line is called dataframe and represents our best-effort optimization of CADIS, allowing dependent types to also only report changes since last tick. A detailed explanation of how each of these performance updates were implemented can be seen in the Performance Section of Chapter 4.

The benchmark’s implementation starts sets up two simulations in CADIS: a producer and a consumer. Each simulation accepts two optional parameters: initialize hook and update hook. This way, each benchmark scenario may optionally define its own initialize and update methods for the producer and consumer. All the data models for the benchmark scenarios and the simulations source-codes are available in the appendix, Sections 8.2 and 8.3. The appendix also has tables and bar graphs (in Section 8.4) with measured values for push, pull, and update processing time, and the amount of data sent and received for the producer and consumer of each test.

5.1.1 PCC set type

In this test, the producer initializes 1000 objects of BaseSet – a base PCC data type with 11 properties (Listing 8.3). The consumer skips the initialization phase, as its role in the benchmark is only to be kept updated. During the simulation, nothing else is done in the producer and the consumer. Figure 5.1 shows the results of this experiment.

For the consumer, baseline and diff.push pull all 1000 objects at every tick, resulting in a constant processing delay of 157 ms on average (Table 8.18) throughout the experiment. With diff.pushpull and dataframe, data is only pulled if there are modifications since last push, which means during the entire experiment no object data is exchanged, reducing pull to around 6 ms. The producer baseline has a similar issue, having to push all objects at
Figure 5.1: Total processing time (push, pull, and update) for producer and consumer in PCC set type scenario.

every tick, adding an average of 519 ms delay (Table 8.17). But all other modes only push data that changed during the update, meaning the producer does not exchange any object data during the entire experiment.

5.1.2 Subset

Subsets in PCC are a collection of objects of a data type that conform to a predicate. The ActiveCar example is a subset of Car, where the predicate is velocity different than 0. For the benchmark, a new type is defined based on BaseSet, called SubsetHalf (Listing 8.4). The SubsetHalf is a BaseSet object whose Number property is divisible by 2. Each BaseSet object is set with incremental values for Number, meaning that in the 1000 objects created, each object has a number from 0 to 999. Thus, a list of 500 SubsetHalf objects should be available at each time step. Figure 5.2 shows the results for the 4 modes.
With base objects, modifications are easy to detect – if a property changed, let the store know, and the store in turn will forward that change to other simulations. However, PCC types are different: whether an object belongs to a collection or not is defined at the store during reclassification as opposed to a modification caused by the client. Thus, for the first 3 optimizations, the consumer pulls every object calculated for the projection. In the dataframe optimization, PCC objects are sent only if added to the previous calculated collection. If an object is no longer part of the collection in the next time step, that information is also sent. Thus, the dataframe line shows a great performance improvement, since no objects were exchanged throughout the simulation. Similarly to the set type results, objects were never modified, created, or updated, so no data is passed by the producer. The only exception being the baseline scenario, where all objects are pushed at every time step.

When comparing PCC set types to subsets, it is noticeable that even though the amount of data transferred decreases from 545 KB to 272KB, the processing time increases from 162 ms to 367 ms (Tables 8.18 and 8.20). This counter-intuitive result is the overhead of having to process PCC dependent types at run-time. In the baseline, diff_push, and diff_pushpull modes, all data in the store is stored as JSON. Thus, requesting all objects is simply transferring a list of JSON dictionaries already in memory. In dependent types

Figure 5.2: Total processing time (push, pull, and update) for producer and consumer in subset scenario.
like subsets, the server needed to create all objects from their JSON, run the query for each object, and only then return the results. In the latest dataframe, dependent types may still need to be recalculated when pulling, but data is no longer stored as JSON. The performance improvement is significant, dropping from an average of 358 ms to only 11 ms. This significant reduction in processing time shows that the delay of recreating the objects to run the queries were much higher than the query execution time.

5.1.3 Projection

Projections allow data types to be trimmed down, removing properties of a PCC data type that is not interesting to the simulation. Projections are likely the most used operation in relational databases, allowing queries to retrieve only a selected number of columns. To test the projection feature, the type BaseSetProjection (Listing 8.5) keeps only two properties from the original BaseSet type: the ID and Name. Results can be seen in Figure 5.3.

![Figure 5.3: Total processing time (push, pull, and update) for producer and consumer in projection scenario.](image)

The results for projection are similar to the ones seen in subset. Because projection is a dependent type, all objects need to be created and then sent to the client for the first three modes. In dataframe, objects do not have to be created and almost no data is transferred, since there new objects are not introduced during the simulation.
5.1.4 Join

A join operation behaves differently from subsets. In Chapter 4, we discussed how a join data type does not share identity with its related types, whereas subset do. For the benchmark, a join data type called JoinHalf (Listing 8.6) is defined. A JoinHalf has a predicate that returns true when the two objects passed as arguments are the same and divisible by two. Thus, in the experiments, the query will run for 1 million times and result in 500 pairs of objects. Figure 5.4 presents the results of running the join benchmark experiment.

A join will return a pair of objects that matched the predicate, and possibly additional properties of the join type. For this reason, when optimizing joins, it is not possible to simply send two primary keys from the store and rebuild it on the client, like it is done in subset. Furthermore, detecting changes becomes non-trivial, as changes to a join collection (i.e. added or removed from the collection) depend on both the pair of objects and the properties of the join type being different. Thus, joins – even in the optimized dataframe benchmark – does not detect changes in the collection. Figure 5.4 reflects this characteristic, with the pull operation of the consumer being higher than in subset, despite the join resulting in the same collection at every time step. However – as opposed to the baseline – the dataframe optimization does not send the serialized pair of objects, but rather only their primary keys and any additional properties. This simple optimization reduces the amount of data being passed from 553 KB to 284 KB (Table 8.24). Added to no longer storing objects as JSON, the total processing time for the consumer reduces from 621 ms in diff_pushpull to 291 ms in dataframe.
In the consumer scenario, both baseline and diff.push are the same as in Section 5.1.1.
every BaseSet object is pulled at every time step. In the other two modes only the changes are pulled, so a much smaller processing time can be observed as a result. For the producer, the first two modes also reflect set types in Section 5.1.1. The diff_pushpull and dataframe have much smaller processing times, but dataframe is slightly higher due to an increased message size – from 57 KB to 114 KB (Table 8.25) in the dataframe optimization implementation (see Performance in Chapter 4). The same increase is observed in the consumer as well (Table 8.26). The new format used by dataframe is larger and more verbose, but allows a hierarchical organization of PCC data types. All dependent types are assigned a group, typically a PCC base set type. Grouping objects this way allows for easy propagation of modifications to related dependent types.

5.1.6 Creation

The creation scenario adds new objects over time, totaling 1000 objects by the end of the 500 time steps. Results are seen in Figure 5.6.

![Figure 5.6: Total processing time (push, pull, and update) for producer and consumer in creation scenario.](image)

The consumer sees an increasing number of objects sent at each time step for baseline and diff_push, as in these modes all objects are pulled. In diff_pushpull and dataframe
only new objects are pulled, meaning only 2 objects are sent at each time step, resulting in a much lower processing time – from around 90 ms to 10 ms (Table 8.28). As for the producer the same situation applies, except only baseline sends all objects, where other modes only send objects that were not sent before. Thus baseline increases linearly over time, where other modes are constant, sending only 2 objects per time step. It is also visible in the Figure 5.6 that the producer’s processing time grows a lot faster than the consumer’s. While the amount of data exchanged is the same, the serialization of objects in the producer takes a significant amount of time. It is not entirely clear how much more serialization can be improved – dimensions of PCC objects behave like graphs, and traversing graphs to serialize data is a costly computational effort. It remains a future task to study alternatives to reduce serialization processing time by either improving graph traversal and serialization algorithms or by restricting data types and object references as dimensions. As in other scenarios dataframe messages are more verbose, leading to an increase in processing data transferred – from 1KB in diff_push and diff_pushpull to 4KB in dataframe.

5.1.7 Summary of Benchmark Experiments

Both PCC and spacetime frameworks are fairly complex software libraries that work closely together to implement the CADIS architecture efficiently. Adding the complexity of distributed simulations in general, it is a challenging task to keep all parts working at maximum efficiency – small changes to either framework can propagate into large processing delay bottlenecks. Through the benchmark executions, we have found three major sources of processing delays: a) serializing and deserializing updates and objects; b) amount of data transferred between client and store; and c) execution of long queries for dependent types.

The first two sources were predominant in the benchmark tests, and were successfully mitigated in the latest implementation of CADIS, the dataframe mode. The diff_push and
**diff.pushpull** were simpler and immediate solutions to prevent set types to be transferred too often, but failed to address dependent types. Furthermore, opting to store objects in JSON resulted in a high deserialization delay bottleneck when converting JSON to object for query executions in the store. In *dataframe*, modifications to dependent type collections are quickly collected and distributed with minimal data transfer, and objects are no longer saved as JSON. The downside of *dataframe* is a more verbose message specification that explicitly determines each individual dimensions modifications (i.e. added, modified, or removed) and the assignment of data types to groups to determine relationship between types. This relationship is useful on the store to propagate updates, but increases verbosity in the message protocol. We are currently working on reducing the message size by moving from JSON to more efficient serializers.

The speed of queries remains an essential problem of CADIS: if queries are too long to execute, they inflict delays. From the results seen in the join scenario in Section 5.1.4, the execution of the query itself is not time demanding. Even though the join benchmark ran the query 1,000,000 times (compared to 1000 in subset), the difference in processing time is attributed to serialization and data transfer concerns. One future work to improve query execution time is to allow trigger properties – only when a trigger property is modified does a query execute again. This way, if objects are not changing, we can assume the result of their predicates in the query will remain the same.

In summary, the benchmarks have helped expose the bottlenecks in PCC and spacetime that have been, for the most part, mitigated with the our best-effort work in *dataframe*. The benchmarks will remain a tool to keep both frameworks within acceptable performance boundaries as the libraries evolve with time.
5.2 Feasibility

The guiding use case for designing CADIS was the urban simulation scenario. The simulation of a city involves many different aspects and stakeholders, such as traffic management, city zoning, power, water, and weather. The cities themselves are organized in departments (e.g. Department of Motor Vehicle, Department of Health, Planning and Zoning) with multiple stakeholders holding expertise in many different domains. And yet most of these departments work independently, not only geographically, but also in knowledge-domain and operation.

Our research question behind urban simulation is: can multiple domain experts in different aspects of a city collaborate in one highly complex urban simulation? Our evaluation covers an example urban simulation scenario with 3 simulation aspects: citizen, traffic, and virtual environment. The citizen simulation is responsible for simulating citizen behavior with respect to traffic: going to work, going shopping, going to a coffee shop, and going back home. The traffic simulation coordinates with the citizen simulation to create vehicles and drive them from source to destination in the city. Finally, the virtual environment serves two purposes: an easy 3D visualization of the city’s traffic and a way for a human to interact with the simulation (though for this evaluation, only the visualization purpose was used).

The implementation of this scenario – called mobdat – was originally done by one of our collaborators, Mic Bowman. To evaluate CADIS, the original event routing system in mobdat was adapted to use spacetime. The citizen simulator is called SocialConnector, the traffic simulator is called SumoConnector, and the virtual environment simulator is called OpenSimConnector. The SumoConnector is an adapter module between the open-source traffic simulation known as Simulation of Urban Mobility (SUMO) [98] and CADIS. The OpenSimConnector is an adapter module between an open-source virtual environment platform called OpenSimulator [138] and CADIS. The SocialConnector is an original implementation done by Bowman. The logical organization of mobdat in CADIS can be seen in Figure 5.7.
The SocialConnector creates vehicles, the SumoConnector forwards position and velocity updates, and the OpenSimConnector fetches the vehicles for visualization.

The experiment runs 20 minutes of simulation at 200 ms per step, adding to 6000 steps. Steps correspond to 2 seconds of simulation time, simulation a total of 3 hours and 20 minutes, starting at 7 AM in the simulation wall-clock time. That time is chosen for having the most traffic, as most citizens head out to work between 7 to 10 AM.

The evaluation consists of 4 scenarios. The event-based scenario is the original implementation, using an event router in shared memory. The unoptimized scenario uses a simplistic spacetime/pcc implementation, equivalent to the baseline mode in Section 5.1. The optimized scenario implements the \texttt{diff.pushpull} performance optimizations as discussed in Section 5.1, due to the \texttt{dataframe} changes not being implemented at the time of this evaluation. Finally, the remote scenario will separate the simulations and the data store by a local area network.
5.2.1 Results

Figure 5.8 shows processing time for each of the simulators running the 4 versions of mobdat. Table 5.1 shows the means for each version and for each simulation. Numbers for the Un-optimized CADIS were particularly bad, showing that the same approach used for shared memory needs to be improved for network distribution. The means of the processing time in the optimized version were orders of magnitude lower, and within the 200 milliseconds time step interval. However, the original Event-based mobdat – as expected – performs better than the Optimized CADIS. The difference in processing time between Event-based and Optimized executions of the SumoConnector and SocialConnector were fairly comparable, but for the OpenSimConnector there was a factor of 3 difference between the means. This difference was inevitable with the current implementation. Analyzing profiling information from the optimized execution, we still found that roughly 39% of the busy processing time was spent in the HTTP library used, particularly on the method for HTTP requests. It is, however, still possible to improve these numbers if a different network stack solution were to be used, possibly UDP/IP instead of HTTP over TCP/IP. Finally, the Remote execution was quite efficient, having only a small difference in processing time compared to optimized CADIS. The overhead of the network stack made little difference over loopback (i.e. localhost communication only) or real sockets over a LAN.

Table 5.1: Processing time in milliseconds for each version of mobdat execution in the experiment. E.B.: Event-based.

<table>
<thead>
<tr>
<th></th>
<th>E.B.</th>
<th>Unoptimized</th>
<th>Optimized</th>
<th>Remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>SumoConnector</td>
<td>70.53</td>
<td>2395.00</td>
<td>91.72</td>
<td>99.26</td>
</tr>
<tr>
<td>OpensimConnector</td>
<td>23.80</td>
<td>328.17</td>
<td>71.94</td>
<td>83.10</td>
</tr>
<tr>
<td>SocialConnector</td>
<td>0.51</td>
<td>174.35</td>
<td>7.04</td>
<td>11.58</td>
</tr>
</tbody>
</table>

There are noticeable performance differences between Figures 5.8a, 5.8b, and 5.8c, caused by the different level of responsibility each simulation has. The SumoConnector (Figure 5.8a) has the task of both querying SUMO for updates and of pushing updates to CADIS, making
Figure 5.8: Performance comparison between the original event-based mobdat with the CADIS version. The **y-axis** is processing time in milliseconds for each clock event. The **x-axis** is minutes of wall-clock experiment time.

It is the heaviest simulation in processing time. The OpenSimConnector (Figure 5.8b) is the second heaviest load, having to both receive constant updates to Vehicle objects and forward them to OpenSimulator. The SocialConnector (Figure 5.8c), in contrast, has only two tasks: create Vehicles based on citizen’s schedules and track Vehicle deletions, used for tracking citizen’s arrivals and prepare their next trip.

The overwhelming difference between the **Unoptimized** and **Optimized** versions of CADIS are also noteworthy. Table 5.2 shows the processing time of each CADIS version broken down into the 3 parts of each time step: *push*, *pull*, and *update*. The **Unoptimized** run was entirely non-feasible: in all 3 simulations the processing time exceeded the fixed time step interval of 200 milliseconds at some point. The major culprit for this underwhelming performance is handling *push* and *pull* operations by copying of entire objects. Particularly for the SumoConnector, a *push* operation took an average of 2222 milliseconds, while a *pull* averaged at 105 milliseconds. The large difference between push and pull is simple: pulls were performed per type, and pushes per object. A pull of a type retrieves a JSON list of all objects of that type in the Store. Pushes would send one object at a time to the server, greatly increasing the load on the network stack. For a memory-based Store, it is necessary to insert one a time, but introducing networking means bulk messages will avoid
needless execution of the network stack. The Optimized CADIS – the equivalent of the diff_pushpull scenario in Section 5.1 – improved communication between Frame and the Store by only communicating changes since last tick, and by only sending and receiving the changed properties. Finally, the Remote version showed a 5-10 millisecond difference for pushing and pulling operations, demonstrating that CADIS can feasibly integrate simulations over a network without much of an overhead with respect to a local execution.


<table>
<thead>
<tr>
<th></th>
<th>C.push</th>
<th>C.pull</th>
<th>C.updt</th>
<th>O.push</th>
<th>O.pull</th>
<th>O.updt</th>
<th>R.push</th>
<th>R.pull</th>
<th>R.updt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sumo</td>
<td>2222.58</td>
<td>105.12</td>
<td>67.23</td>
<td>40.93</td>
<td>5.45</td>
<td>45.27</td>
<td>44.78</td>
<td>8.57</td>
<td>45.83</td>
</tr>
<tr>
<td>OS</td>
<td>0.01</td>
<td>318.66</td>
<td>9.42</td>
<td>0.01</td>
<td>53.22</td>
<td>18.65</td>
<td>0.02</td>
<td>64.14</td>
<td>18.87</td>
</tr>
<tr>
<td>Social</td>
<td>0.98</td>
<td>173.22</td>
<td>0.09</td>
<td>0.83</td>
<td>5.95</td>
<td>0.20</td>
<td>1.27</td>
<td>10.03</td>
<td>0.21</td>
</tr>
</tbody>
</table>

5.3 CADIS and HLA

The most widely adopted distributed simulation framework for collaborative simulations is HLA. The difficulty in adopting HLA has motivated our design of CADIS, but CADIS is not a replacement of HLA. Rather, CADIS is as a high level abstraction – using the PCC programming model in lieu of the well known publish-subscribe of properties and interactions (i.e. events) found in HLA. As such, HLA and CADIS can operate together, despite not being clear whether it should. In order to explore this possibility, we decided to build a CADIS and HLA integration, where HLA performs the data synchronization through property updates, while CADIS offers its API to the simulation and converts data updates from the RTI into PCC data types.

The integration is divided in two parts: the HLACo

backend. The challenge in this integration is that HLA is only available (i.e. stable and functional) in Java and C++, while CADIS’s current implementation is in Python. To integrate both platforms, we used Jython – a Java implementation of the Python library. However, Jython has two issues that make this integration difficult. First, it takes a large toll in performance, as essentially Jython uses Java to implement Python. Second, Jython interprets dictionaries as a different type called stringmap, which invalidated the mechanisms used in spacetime to build PCC data types.

The HLAConnector was built and tested using the mobdat experiment in Section 5.2, replacing the event-based router by HLAConnector. The performance was too low for significant measurements, but the integration worked – the mobdat simulation ran as expected. Unfortunately, running the CADISConnector is currently not possible due to the limitations of Jython’s dictionary. We have built the CADISConnector nonetheless, which can be found in the mobdat source-code repository, as an exercise of how this integration can be achieved. It remains as future work to implement the Java versions of spacetime and pcc frameworks to effectively test the integration of HLA and CADIS and determine whether using the existing structure of HLA is beneficial to CADIS.

5.4 Usability Study: Distributed Simulation Course

As made evident in Boer et al. industry survey, developing distributed simulations is a difficult task, even for simulation practitioners. It can be an even more daunting task if given to computer scientists with little to no knowledge in the field of simulation. The CADIS architecture proposes to make the development of distributed simulations easier, particularly with independent groups of developers that have little opportunity to coordinate. To evaluate the effectiveness of CADIS with real-world developers, we have used CADIS as part of a graduate-level class at the University of California - Irvine.
The course – called Distributed Interactive Simulations – was 10 weeks long, had 9 students enrolled, and met once a week for 2 hours and 50 minutes. The first 5 weeks were workshops in simulations and distributed simulations, where students had the opportunity to read and discuss some significant papers in the field and had a small warm-up project. For this first project, students were allowed to work in pairs and were expected to write a small traffic simulation – containing two roads and a single intersection – with collision detection. All students succeeded in the readings, discussions, and the warm-up project.

In the last 5 weeks, students were introduced to CADIS, in the form of the spacetime/pcc framework implementation. The project was to create multiple fleets of cars over an existing simulation of a city. Each group picked a type of fleet to build (e.g. ride sharing, amazon delivery) and had the task of simulating routes with their own vehicles, while avoiding collisions with their own vehicles and the vehicles simulated by others.

### 5.4.1 City Simulation

This usability study uses a simulation of the city of Uppsala in Sweden, done by a company named Encitra [116]. My advisor (Cristina Lopes) is responsible for the development of the virtual environment platform the simulation is built on – OpenSimulator. Having built the OpenSimConnector for the evaluation in Section 3.2.5, it was fairly easy to port a similar solution to use the more complex Uppsala simulation. The Uppsala simulation took care of visualizing both the city and the vehicles simulated in it.

Another vital task for the usability study is path finding. A ride-share service, for instance, is expected to roam around the city’s roads, to pick up passengers, and to take them to a new location. The city was modeled using OpenStreetMaps, and is represented as an XML file indicating node locations that form the roads and intersections. We decided that there was not enough time to have the students build a routing system (nor was it the intent of
the class), so we built a simulation service that takes requests of source and destination, and returns a list of X and Y pairs that will form the path between them.

Figure 5.9 shows the logical organization of the usability study. We developed the City Simulation and Route Simulation, and students contributed with multiple instances of Fleet Sim – simulations responsible for driving their own fleet of vehicles in the city’s traffic.

5.4.2 Route Simulation

The route simulation service uses a route planning library in Python called PyRoute, found in the OpenStreetMaps website. We decided to build a CADIS application to handle routing for the simulations the students were to build. It was an interesting exercise to build a request/response service in a data-oriented architecture. Listing 5.1 shows a simplified version of the source-code for the route simulation service. The original can be seen in https://github.com/Mondego/spacetime/tree/master/python/applications/nodesim.

As with all Spacetime simulations, the first task is to declare the simulation’s interests in PCC types. Lines 1–4 shows that the RouteSimulation produces and deletes roads and business and residential nodes. These were extracted from OpenStreetMaps, where we assigned nodes
as businesses or residential based on the area of the city they were in. More relevant to its function, RouteSimulation produces Routes and tracks (i.e. receives updates for creation and deletion) and deletes RouteRequests. A RouteRequest has 3 dimensions: name, source and destination. Name is used to identify the route that is being created, so it can be fetched later. Source and destination are self explanatory. Route contains only one dimension, a list of X,Y pairs containing all the points that will take a vehicle from source to destination.

If a simulation wishes to use the RouteSimulation service, it needs to produce RouteRequest and track Route. When tracking Route, all new routes will be returned, including ones that were not requested by the simulation receiving it. There are two ways to specify a route. One – used in class – is simply to parse through all Route objects and find the one that matches the name used in the RouteRequest. Another approach would be to use subsets, and create a new PCC type whose predicate matches the name used for the RouteRequest.

With a routing service and virtual environment platform available to the students, all that was left was to build their own simulations and add it to the world. In the last 5 weeks, students would come to class and have time to work on their simulations. In the last 2 classes, we integrated all the simulations in CADIS. In the next sections, we present the students’ impressions in learning and developing distributed simulation with CADIS.
1 @Producer(BusinessNode, ResidentialNode, Route, Road)
2 @Tracker(RouteRequest)
3 @Deleter(RouteRequest, BusinessNode, ResidentialNode, Route, Road)
4 class RouteSimulation(IApplication):
5     def initialize(self):
6         self.pyrouter = PyRouter()
7         # Initialize roads, business and residential nodes
8
9     def update(self):
10        for req in self.frame.get_new(RouteRequest):
11            waypoint_list = self.pyrouter.doRouteAsLL(req.Source)
12            pccroute = Route(waypoint_list)
13            self.frame.add(pccroute)
14            self.frame.delete(req)
15
16     # Datamodel
17     @pcc_set
18 class RouteRequest(object):
19        @dimension(str)
20        def Source(self):
21            return self._Source
22        @dimension(str)
23        def Destination(self):
24            return self._Destination
25        @dimension(str)
26        def Name(self):
27            return self._Name
28
29     @pcc_set
30 class Route(object):
31        @dimension(list)
32        def Path(self):
33            return self._Path

Listing 5.1: Route simulation and data model source-code
5.4.3 Student Survey and Interviews

In the last week of class, 2 groups of students were interviewed: the first group had 2 students and the other had only 1. These interviews served to achieve an in-depth understanding of how students with little experience in simulation perceive the process of writing distributed simulations. The answers to these questions served to create the anonymous survey, which was distributed online a week after the last class. The survey had 13 questions, 8 of which were multiple choice and 5 were open-ended. The questions collected information on their experience in developing software and specifically on developing simulations. Then it covered questions of using spacetime and pcc, including using the algebraic operations as modeling tools. When not specified, scales used in the survey were Likert scales (1-5).

It is important to note that the survey had only 6 responses out of 9 students. Hence, this usability study is an exploratory exercise, not providing any significant results from an statistical standpoint. The participating students were graduate students in the Donald Bren School of Information and Computer Science. Most were on a Masters program, with considerable programming experience. While most students reported to have considerable experience or be an expert in software development (83%), all students reported to have only some or no experience (50% each) in simulations and distributed simulations.

When asked about how challenging simulations were, the majority (66%) considered simulations to be of normal difficulty (3 on the scale), while the rest considered it to be challenging (4 on the scale). The same questions for distributed simulations had 100% responses as challenging. For collaborative simulations, 5 answered challenging and 1 answered very challenging (5 on the scale).

The majority of students agreed or strongly agreed that pcc and spacetime made it easy to develop and distribute a simulation, reduced communication between stakeholders, made it easy to maintain a consistent data model, and that the framework was easy to understand
and develop for. Particularly on the latter two questions, there was less of an agreement (2 answers were neutral, meaning 3 on the scale). This was reinforced later that the framework still needs to be clearer in some aspects. With regards to independent development, students collectively agreed that there was little to no need for communication with other stakeholders when developing and deploying their fleet simulation on the common CADIS server.

Finally, students were asked how difficult was to understand algebraic operations in pcc and interest management and the update loop in spacetime. Most students considered subsets easy to understand, but found joins and parameterized subsets (i.e. subset that have queries with arguments) to be difficult. The concept of projects was a split opinion. In spacetime, students found the push/update/pull loop easy to understand, but were divided in the interest management concept.

In the open-ended questions, 3 students mentioned that the main advantage of CADIS was abstracting away state synchronization, with one answer particularly mentioning the interest management solution as responsible for such a clean abstraction. When asked about challenges, each answer was different, including debugging, understanding the conceptual model, and finding no challenge at all. When asked about improvements, answers also varied, and included improving the power of expression, detailed error messages, and concrete and clear expectations for the users. All students found CADIS a valuable tool for the development of collaborative simulation.

In the interview, students were generally excited with the platform. One student said: “Not having to worry about networking and database helped developed simulations like OOP software”. A student expressed how they did not think they would be able to achieve the goal of connecting everyone’s simulation in the end. One student mentioned that it was not too hard to understand the concepts, but teaching a class on the topic would have made it a lot easier. For the future, the first group mentioned they would like to see advanced query features available in relational databases for CADIS.
5.4.4 Findings

The survey and interviews were limited in population size, but provided great initial insights into CADIS. We had confirmation from reasonably experience software developers that the proposed interfaces and paradigm matched to their existing knowledge of relational algebra. Still, the conceptual use of relational algebra as OOP types was still not as smooth as we expected them to be. One interesting remark was a group mentioning they were not immediately aware that queries and predicates were executed on the remote data store, and not locally. Observing students in class, it suddenly became apparent that most students faced the same confusion. And it is easy to see why: not often do we write source-code that mixes local and remote code in the same abstract data type.

It was a particularly exciting achievement for the students and for us, the instructors. At the start of the class, the goal of having students who have little to no experience in developing simulations design a distributed simulation with multiple fleets of vehicle seemed daunting. Specially since students had only 5 weeks to learn and use CADIS appropriately. The fact that this became possible shows a great deal of promise for CADIS and for the instruction of students in the field of distributed simulations, a fairly neglected discipline.

The biggest challenge in this usability study was coordinating the data types the students were developing. Each group developed their own data type to represent their fleet of vehicles. When we integrated all data models in the simulation during class, there was no simple way for students to run their simulations, debug issues, and upload a fixed version of their data model while the simulation was running. Stopping the simulation would have meant that each update of each group of students would disrupt the execution of others.

We adopted a temporary solution that allowed data models to be shared in an automated way. By connecting a shared github repository with the CADIS store, updates to data types could be automatically pulled and reloaded. Students pushed changes to their data types
in the repository during class and the execution of all other simulations would be intact – reflecting the new changes immediately. This solution served its immediate purpose in class, but cannot be generalized for collaborative simulations – a more permanent solution needs to eventually be designed and implemented. Furthermore, This experience raised another question: what if other simulations relied on certain behaviors of a data model that is suddenly changed or gone? Evolution of data models is vital, but not yet handled in the current design decisions of CADIS.

In summary, the most important finding is showing CADIS is feasible to learn and use in a short period of time. Section 3.2.5 proved CADIS is feasible in terms of performance, while this usability study showed groups of developers with a reasonable computer science can backgrounds can collaborate to build a highly complex simulation, where each only contributes with a small part of the expertise. We have found some significant challenges in organizing shared data models, for which we have conceptual solutions (see Section 5.5.3) to be realized in future work.

5.5 Research Questions

Chapter 1 presented 3 guiding research questions for the investigating the effectiveness of CADIS in fulfilling the thesis statement. The questions were:

1. How does CADIS compare to traditional ways of load-partitioning simulations?

2. How does CADIS compare to similar architectural efforts such as HLA?

3. How effective is CADIS in supporting independent, but collaborative development of different aspects of simulations?

The four-part evaluation of CADIS presented in this Chapter can adequately answer our
5.5.1 How does CADIS compare to traditional ways of load partitioning simulations?

CADIS is a generalist architecture, like HLA and RCAT, that allows flexible partitioning strategies. The traditional load-partitioning strategy is object partitioning, where objects are assigned to simulators based on a filter, such as the location of the object. Object-partitioning is not an effective load-partitioning strategy for collaborative simulations, where simulators are responsible for different aspects of the simulation and must share access to the same objects. CADIS is also different from HLA and RCAT, where the burden of partitioning the workload falls on the simulation developers. Thanks to the PCC programming model, distribution is inherently a part of the data model. Simulations express interest in different aspects of the same entities, and are given access and notified of updates of data types that are specifically relevant to the simulation.

In terms of performance, our evaluation exposes multiple aspects of CADIS performance in running DIS simulations. The benchmarks in Section 5.1 presented base values of how fast CADIS can process algebraic operations and maintain a simulation synchronized with the store. These benchmarks isolate individual simulation execution, so it is not confounded with partitioning strategies and inter-simulation dependencies that are often simulation specific concerns. The feasibility study in Section 5.2 complements the benchmark results, by analyzing a use-case of CADIS in urban simulation. The feasibility study – in contrast to the benchmarks – is focused on the relationship between traditional event-based systems used in simulations to CADIS.

The results are encouraging – CADIS has a performance penalty, as expected, but performs well enough to be used in real-world applications. It is difficult to draw a parallel between
different evaluations of different DIS architectures (we discuss these problems in Chapter 6). But RCAT, DSG-M, and CADIS have all shown to process a simulation step under 100 ms with around 1000 simulated entities. All three experiments show that the amount of data transferred is the bottle-neck of DIS – the reason why the benchmark and feasibility experiments of CADIS focus on optimizations that reduce data transfer. Hence, in terms of scaling in size, CADIS performs as well as previous design experiments.

Scaling in scope is the focus of CADIS, and the usability study in Section 5.4 has shown potential as a paradigm for collaborative development. Students who were unfamiliar with traditional simulations easily understood and developed a complex distributed simulation using CADIS’ relational algebra operations to model their vehicles. Simulations interacted with each other, but were developed independently – students had no knowledge of how other students developed their vehicles. Additionally, vehicles were visualized through OpenSimulator without any additional effort on the part of simulation developers.

To summarize, load-partitioning performance in CADIS is comparable to previous solutions that were focused in scaling in size, but adds more value in its novel scalable scope features. Data transfer over the network remains overwhelmingly the cause of processing time delays and an essential problem of DIS systems. Where CADIS comes out successful is in its ability to easily scale in scope – an improvement to load-balancing that is traditionally neglected by existing load-balancing approaches in DIS.

5.5.2 How does CADIS compare to similar architectural efforts such as HLA?

As discussed in the previous answer to research question 1, CADIS and HLA share a generalist load-partitioning strategy, but have different approaches to managing shared state synchronization. CADIS is strictly data-oriented: only object updates are exchanged between
simulations, as opposed to the ubiquitous use of events for inter-simulation communication. Objects in HLA are data structures with inheritance – different from classes and objects used in OOP. CADIS uses objects in a similar manner to OOP, but adds additional functionalities through the PCC programming model that enables data types to be reclassified at runtime.

While CADIS provides highly expressive data models, it currently lacks the maturity of a well established architecture like HLA. HLA, for instance, provides logical ordering of updates and allows permissions of simulations on individual object properties. Hence, in Chapter 4, we investigated whether HLA and CADIS could work together – with CADIS using HLA as an advanced state synchronization protocol, while converting HLA’s object updates into PCC data types.

We wrote a working HLA-compatible version of mobdat (as described in Section 5.3). The event-based system of mobdat was replaced for an HLA-based event distribution. The source-code is available in https://github.com/arthur00/mobdat/tree/hla. The resulting simulation was tested and worked successfully, however, the toll of running python on top of Java was very large, and the simulation only performed well up to around 40 vehicles. This exercise was not intended to show performance comparisons, but rather demonstrate that simulation data distribution models are compatible. The next step was to add a CADIS connector, that would transform updates received from the RTI in HLA into PCC types. Unfortunately, Jython and PCC became ultimately incompatible due to low-level language implementation restrictions.

In terms of scalable scope, CADIS affords the same independent development capabilities of HLA, by separating data model from the simulation logic. What CADIS offers in addition to HLA is the capability of modeling object at a high level abstraction and the capability of modeling collections of objects through PCC. This further separates models from logic, by embedding the logic that defines a collection of objects as part of a data type. Using our traditional example, HLA’s definition of an endangered pedestrian would have to be defined
as an entirely different object, and maintained by one of the participating simulations. In CADIS, the endangered pedestrian is entirely in the data model. The down-side is that HLA’s RTI has only the task of routing messages, whereas CADIS needs to process queries and maintain data types synchronized. However, Sections 5.1 and 5.2 have shown that CADIS can perform these high level functions with acceptable loss of performance.

The concept of CADIS using HLA as a data distribution mechanism can potentially bring several advantages. HLA is a finished product, in terms of security, language-independent data type formalization, and other features that can fulfill or guide CADIS in its own development.

5.5.3 How effective is CADIS in supporting independent, but collaborative development of different aspects of simulations?

Supporting independent collaborative development is key to scalability in scope – the missing piece to a scalable DIS architecture. Our case-study in Section 5.4 has shown that CADIS is easy to understand, even by developers with no expertise in simulations. Students were capable of creating interactive simulations that worked together to create a rich traffic environment under an independent development process – no knowledge of the development of other simulations was necessary. With the visualization setup by the instructors, any car added to the system was appropriately visualized in the virtual environment of OpenSimulator. Typically, adding a graphical viewer for 3D models is a complex project in itself. Thanks to the PCC relational operations, it was possible for new data models to be added and simulated without changing the graphical viewer source-code. Furthermore, we implemented a data-model update detection and reloading mechanism that did not require halting the running DIS – updates in the data model were taken in at runtime. Incremental updates to data-models are entirely non-disruptive, creating a collaborative simulation environment
where an increase in scope causes no disruption. It remains to be studied if it is possible to achieve the same non-disruptive workflow with destructive data model updates, such as removing properties of a data type.

The biggest challenge of CADIS proved to be sharing the data models between participants, made evident in the usability study. In class, we had the source-code repository of the data model hooked to the CADIS through a Python script as temporary solution. As a permanent solution, we envision a package manager similar to the one used in Python called *pip* ([https://pypi.python.org/pypi/pip](https://pypi.python.org/pypi/pip)). The package manager would allow publishing of new data models and installation of remote data models locally.

Another concern with shared data models is evolution: how should changes to the data models affect the data type definitions. Our conceptual solution to this problem is to adopt the same set of rules as Google’s data interchange format called Protocol Buffers ([https://developers.google.com/protocol-buffers/](https://developers.google.com/protocol-buffers/)). Additions to an existing data type, such as adding new dimensions, requires no update to the name of the type, as it does not break existing functionality. However, if changes to an existing data type causes removal of dimensions or modification of behavior, developers should create a new data type, which can be as simple as adding a version to the data type name (e.g. `ActiveCar_1.0`).

### 5.6 Conclusion

We have presented a novel approach to distributed simulations, with the intent of reducing the entrance barrier and improve collaboration between simulations. The design philosophy of CADIS is to be simple to develop for, with transparent data sharing in native source-code. The combination of PCC – a highly expressive programming model for establishing relationship between objects – and spacetime – driving the simulation and abstracting the
distribution – has shown to be an effective way of developing distributed simulations.

In the experiments, CADIS was shown to be feasible and performant enough for a realistic simulation workload. While CADIS does face some – albeit acceptable – performance degradation, there is still room for improvement by tackling overhead in the network stack used for communication between the store and frame. In the usability study, we also shown how CADIS is feasible as a tool for computer scientists with some experience to easily tackle complex simulation integration tasks.

Our future work on CADIS is to improve performance and scalability, support more languages, and add new capabilities to our data model definitions such as permissions, language-agnostic definitions with source-code generation, and a package management systems for sharing models.
Chapter 6

Evaluation and Testing of DIS

The design and evaluation of DIS architectures in Chapters 3, 4, and 5 presented many fundamental questions regarding both the goals of the architecture and whether they were achieved after implemented. This valuable experience is seldom found in literature – DIS systems vary greatly in purpose and requirements, so existing architectures and frameworks tend to miss the larger picture of designing and evaluating DIS systems in general. Separating the architecture from the simulation that is intended to run on it is a difficult but necessary task for two reasons: a) so the developer understands the limitations of the framework; and b) so we understand the design decisions made, and the resulting trade-offs that will impact simulation execution. This chapter acts as post-mortem bird's-eye view analysis of DIS architectures and distributed simulation requirements, and is intended for software architects to comprehend the challenges in the field of DIS systems.
6.1 Introduction

In software systems, the first measure of a successful design is the fulfillment of functional requirements. However, functional correctness is not enough; non-functional properties, i.e. the operational characteristics, are equally important for the success of software systems. This dissertation has covered many topics in designing and evaluating DIS architectures, but DIS is only part of a broader realm of applications that have high availability, consistency, and distribution requirements. These systems are part of the broader collection of Distributed Real-Time (DRT) applications. In DRT applications, non-functional requirements are often a critical part of the overall function of those systems and need to be taken into consideration from the early stages of design; neglecting non-functional requirements can possibly render the software useless. For instance, many social applications and online games, such as Google Hangouts or Second Life, are naturally distributed, must perform in real-time and must be resilient to failures. If the response time between components of these systems is above a certain threshold, or if the components fail systematically, these systems become unusable.

Designing for distributed components and real-time responsiveness is challenging, as these two requirements often hinder each other. Distributed systems partition 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 responsiveness when designing a DRT application.

But designing DRT systems is not the only difficult aspect of these systems. Evaluating the success of those designs is also a non-trivial task. It is often impractical, and clearly unwise, to evaluate and test a DRT application as it is deployed in production. It is impractical because the operation may require hundreds to thousands of machines and users, and it is unwise
because the application may not be functioning correctly or may not be operating at an acceptable level. It is then necessary to develop experiments and metrics that can be expected to perform similarly to the production deployment. Yet assumptions and abstractions of test deployments, such as unlimited bandwidth, no jitter, and no thread context-switching costs, can be made carelessly, resulting in unachievable performance in production. Furthermore, choosing and interpreting the metrics that demonstrate correctness and performance of a design also requires careful consideration.

In spite of these difficulties, DRT systems become necessary when a combination of properties from distributed systems and real-time is required. The variety of DRT systems these days is very wide, so this chapter focuses on one type of DRT system that we have more experience with: Distributed Virtual Environments (DVE). DVEs are DRTs that connect multiple users instantly with each other and with a shared virtual space over a network. These environments have a broad range of uses; applications range from shared observation of simulation of real world physics, to games, to creating interactive platforms where users can share experiences, engage in communication, and even modify the virtual environment as they see fit. Examples of DVE applications include World of Warcraft, Second Life, Google Hangouts, shared online editors, advanced instant messaging systems (e.g. Slack), among many others.

This chapter presents an analysis of our experience with designing, testing and evaluating a DVE, OpenSimulator [138]. In doing that work, we have come across several challenges that, although not new, illustrate very well the kinds of challenges that are present in the development of DVEs. As such, the contribution of this paper is twofold: (1) it provides a couple of concrete design and profiling scenarios that are representative of a large spectrum of situations in the development of DVEs; and (2) it reflects on those experiences, placing them in an historical perspective of DVE and DRT research over the years, showing that these challenges are intrinsic and quite interesting as research topics.

The remainder of this paper is organized as follows. Section 6.2 presents the context for
the case studies and their analysis. Sections 6.3 and 6.4 present the two experiment case studies and the lessons learned in each. Section 6.5 places those observations in an historical perspective. Finally, 6.7 offers some departing thoughts.

6.2 Context and Prior Work

6.2.1 DVEs and OpenSimulator

The design of DVEs tends to fall into two camps: peer-to-peer [64, 93, 119] and client-server architectures [49]. Although peer-to-peer DVEs are very popular in research, most commercial DVEs are done in a client-server architectural style: users connect to a single server[-side], responsible for maintaining rules, generating reactions, and broadcasting updates to all users. The reasons for the industry preference fall beyond the scope of this paper, but security and privacy are some of the major concerns. Small DVEs are able to serve all those functions from a single server. However, as the number of shared objects and users increases, single servers are bottlenecks [6, 17, 25, 178]. The responsiveness of these environments is highly dependent on the number of real-time events that need to be distributed, and those events are highly dependent on users’ actions; in other words, performance of the system, as a whole, is highly application-specific.

OpenSimulator [138] is an open-source virtual environment framework that uses the same protocol as Second Life; it is a clean-room reimplementation of the server-side of Second Life that is able to use the unmodified Second Life client. Comparable DVE open-source distributed simulator implementations are OpenWonderland [92] and Meru [86], but these are less popular than OpenSimulator. Over the past 8 years, we have been contributing to the development of OpenSimulator.\footnote{The third author is one of the main core developers of OpenSimulator, and the other two authors have}
tions for alternative architectures for scalability, and in performance assessments of various scenarios that seem to be problematic in real world usage of OpenSimulator.

Many scalability approaches for virtual environments involve space partitioning techniques. In earlier space partitioning methods [33, 187], space is partitioned in fixed-size large areas of space, sometimes referred as regions or worlds. Due to constraints dictated by the client-server protocol, OpenSimulator inherited that architecture from Second Life itself. In OpenSimulator, like in Second Life, the world is divided in blocks of 256 meters squared, and each region is simulated on a different simulator server. A novel and more flexible approach is space partitioning through microcells [45], which are indivisible small areas of space that can be grouped to form custom shaped partitions that better adapt to load. But even specialized space partitioning methods alone were shown to be insufficient under certain conditions of load [112]. Many other load partition schemes can be designed.

The **Distributed Scene Graph (DSG)** is a client-server architecture for decoupling Scene and operations [102,112,113] in OpenSimulator. 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. DSG uses an eventually-consistent timestamp-based synchronization protocol for resolving updates between simulators. The clocks are synchronized using the Network Protocol Time (NTP) service, and the update with the highest timestamp is applied on every simulator. For more details on the consistency model used in DSG, see Liu et al. [114].

In **DSG with Microcells (DSG-M)** [180] we redesigned DSG to push scalability further, by allowing simultaneous decoupling of operations and space through microcell par-
titions [45]. 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 worst-case scenario and nearly double for a perfectly partitioned space scenario (i.e. no inter-partition communication).

The case studies in this chapter pertain to our experiments with the design and evaluation of DGS-M, and to the evaluation of specific problematic situations in unmodified OpenSimulator.

6.2.2 Six Dimensions of Concern

The evaluation of the design, and the systematic testing of any DVE require the existence of well defined metrics for establishing acceptable behavior. In a previous paper [179], we formulated six concerns that capture important tradeoffs of DRT systems: correctness, fault tolerance, parallelism, time sensitivity, consistency, and overhead costs. As such, in our OpenSimulator work, we have used metrics in all of these dimensions of concern.

The research community has long identified these, or variations of these, as major concerns for these systems (e.g. [43,133,154,164]). We will give a more in-depth historical perspective on these issues in Section 6.5. In order to ground our case studies, we give just a brief description of each of these dimensions of concern.

- **Correctness.** In a traditional algorithmic perspective, when an algorithm is correct, execution will produce correct results repeatedly. External factors, such as the operating environment, are abstracted away. However, even when ignoring the inevitability of application defects, most DRT systems are not deterministic because the operating...
environment is a fundamental piece of these systems. Hardware and physical devices such as CPU, memory, and networking can influence computation results due to exhaustion of resources or heterogeneity of hardware. Additionally, interactive systems, such as DVEs, process user input continually, and don’t necessarily produce a final output. Determining the behavior to be correct requires more malleability, room for imprecision and, many times, a fair amount of ingenuity.

- **Fault tolerance.** Fault tolerance is a system’s ability to survive failures, and is a highly desired property of distributed systems. Distributing computation adds more hardware and networking, increasing the chance of a single component failing. As a distributed system grows, so does chance of failure. Through fault tolerant design and 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.

- **Parallelism and Scalability.** Parallelism enables computation to be partitioned and executed in parallel. Partitioning computation may require coordination, which 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. Distributed systems have networked parallelism, where processes are executed in different hardware, connected through a network. 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.

- **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.

- **Consistency.** The consistency property determines how each participating node of
the distributed system maintains shared state. Shared state can be always consistent, eventually consistent, or allow for inconsistency. Enforcing consistent states for every node at every point in time would require strong consistency algorithms that may break real-time requirements. Many 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.

• **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.

Many of these concerns are hard to measure, as they are often application-specific and are also correlated with multiple hardware measurements such as CPU, memory, and latency. Evaluations of DRT systems must account for multiple external factors, such as hardware, network, and operating system. Furthermore, many DRT applications cannot, or should not, be tested in production; thus, controlled experiments must be used to mimic real-world usage.
6.3 Case Study 1: DSG-M

This section presents a study of the evaluation of DSG-M, an extension of DSG, which, in turn, is an extension of OpenSimulator. In designing DSG-M we wanted to know whether, in practice, it was “better” or “worse” than DSG.\(^2\) We describe the rationale behind the experiments and metrics, and conclude the section with observations regarding the challenges we encountered.

6.3.1 Objective

In terms of the six dimensions of concern presented in the previous section, DSG-M was designed to be parallel/scalable above all else, much more than the basic OpenSimulator and the DSG extension; correctness and time sensitivity were secondary, but important, concerns. Thus, the objective was to measure precisely the systems’ performance along those dimensions. In other words, the evaluation goal was to assess how much more scalable than DSG DSG-M was under acceptable correctness and time sensitivity intervals. The secondary objectives were to identify processing bottlenecks of the distributed simulation and to estimate the computing power required per simulated entity.

6.3.2 Experiment

After much consideration about how to measure behavior that could both push the limits of the simulators and be “correct,” we designed a physics-based experiment; we chose to simulate a device called Galton box [68]. Figure 3.12 in Chapter 3 shows both the original sketch of the device, and the simulated Galton box.

\(^2\)We had previously presented some results of DSG [102] comparing it to unmodified OpenSimulator.
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. With a large enough number of balls, the distribution becomes normal. Second, we can drop tens of thousands of balls, in order to obtain a statistically significant and repeatable result. Third, to increase load we simply drop the balls at a faster rate. 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.

6.3.3 Setup

The evaluation consisted of comparing the results of the experiments on two system designs:

A. Non-partitioned: One simulator responsible for handling the entire physics workload. This is the baseline case, corresponding to DSG.

B. Two Partitions: Two simulators shared the workload of computing physics by splitting the region in half. The split separates each of the Galton Boxes in half, as shown by the orange lines in Figure 3.12. As there are 3 rows of droppers being split, one partition will contain one row, while the other will contain 2. This setup had two sub-cases, active and passive subscriptions, details of which are out of scope for this article. For more information, refer to the original paper [180].

To generate enough load to overwhelm a single simulator process, we used 4 Galton boxes of \( n = 93 \) levels, with 27 droppers each (3 rows of 9). All droppers are at the exact same height and the 3 rows of each Galton box are aligned across all Galton boxes. Droppers create balls at an experiment-defined period of \( t \) seconds per ball. Each dropper drops 350
balls per experiment, at a configurable rate $t$. Each row is offset by a space larger than the diameter of the ball, so there are nearly no collisions between balls. By decreasing $t$, balls are generated faster and simulation load is increased.

Considering the modified Galton box has 3 rows of droppers, each row will generate a binomial distribution with a different average and same standard deviation. Figure 6.1 shows the expected theoretical distribution of an experiment with 37,800 dropped balls. The Figure shows the 3 expected distributions and their overlapping total.

![Figure 6.1: Example distribution of 3 rows of droppers and the combined distribution for 37800 dropped balls. Notice that there are 93 levels, but 96 buckets, due to the overlapping of binomial distributions.](image)

Each simulator runs by itself on a dedicated desktop, connected by a Local Area Network. The desktops are Intel Core i7-2600 CPU @ 3.40 GHz, 16GB RAM, and 1Gbps Ethernet connections. The operating system is Ubuntu 12.10, and the simulators run on mono 3.2.8.
6.3.4 Metrics and Results

The concerns we were interested in evaluating were correctness, represented as ball distribution in buckets, time sensitivity, by verifying whether or not the simulation runs in real-time, and scalability, represented as the balance of load and performance, with and without space partitioning.

The metric for correctness is ball distribution per bucket. To compare the results with the baseline, we used root mean square error (RMSE).

For time sensitivity, we first opted for CPU as a measure of performance. If the simulator is overwhelmed (high CPU), the simulation will slow down, and real-time behavior will be lost. Later on, we changed this metric to ball drop interval: the time a ball takes from creation at the top to destruction on the floor. This value is of $124.82 \pm 1.42$ seconds on normal conditions for DSG (i.e. non-overwhelmed simulator). The reason for this change will be discussed in the next subsection. Other metrics collected were number of messages exchanged for each simulator, number of messages in the queue to be sent, and network bandwidth.

The scalability metric is simply the number of balls being simulated. The more we can simulate, the better we can scale. 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.

Figure 6.2 shows a summary of the 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,
Figure 6.2: Number of balls (left), average interval between creation and collection (right), and CPU usage (right) over wall-clock time in 24 hour format. 

(a) and (b): Experiment A: one physics simulator; (c) and (d): Experiment B: two physics simulators (1 and 2), dividing region in half, but with 2/3 of the balls in one simulator and 1/3 in the other.

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.

6.3.5 Observations

What follows is a list of the most relevant reflections from these experiments that are relevant for DVE research.

1. Correctness.
The experiments were designed to assess the differences in *scalability* between DSG and DSG-M under acceptable intervals of *correctness* and *time sensitivity*. We knew what kinds of design changes we wanted to try in DSG-M regarding scalability, and we also knew how to measure time sensitivity; but we needed some concept of correctness, and that was surprisingly not trivial. Correctness in a DVE can be seen under two perspectives, namely: (1) From an algorithmic perspective, functional correctness is the primary goal; if the DVE has physics, for example, 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, etc.; and (2) From a user perspective, all that matters is how believable or immersive the virtual environment is; incorrect behavior that cannot be noticed by people is tolerated.

In designing the experiments, we faced the question of whether to assess correctness from the user perspective or from the system perspective. A user-facing experiment would, in many ways, be more meaningful, but doing user studies is a time-consuming effort that requires either a large number of independent subjects or a large time commitment on the part of a few. Plus, it is quite hard to design meaningful perception metrics without a deep understanding of the human visual system. Algorithmic experiments are much easier to implement and measure. Physical simulations, in particular, have properties that make them 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. We ended up doing the Galton Box simulation, a form of assessing correctness of a physics simulation adopted from experiments by close collaborators [4].

However, it is important to be aware that algorithmic correctness is not the same as user-level correctness, and this is an important distinction that designers of DVEs need to take into consideration.
2. **Choice of baseline.**

As mentioned before, the metric for *correctness* for this experiment was the distribution of balls in bins. We derived the theoretical predictions for that number of balls, and started by using this prediction as the baseline; the initial approach was to compare *correctness* between DSG-M and DSG under heavy workloads by comparing the empirical results of both with the expected theoretical values. Trial runs showed a distribution that resembled a normal curve, but further analysis showed that the standard deviation of the distribution in the experiment in a non-stressing scenario was higher than in theory. In an effort to understand the deviation from theory, we realized the issue did not lie with DSG or DSG-M, but rather with the physics simulation of OpenSimulator itself: the physics engine was not precise enough to match the theoretical expected distribution of balls to bins. In other words, we assumed that the physical simulation under normal operating conditions matched the theory, but that was not the case.

The solution was to replace the theoretical baseline with an empirical one. Specifically, the new baseline came from measuring the results of DSG under a non-stressing workload.

Relying on a theoretical prediction as baseline to compare two architectures may seem like a perfectly reasonable approach, but it is naive. In general, in DVEs it is hard to abstract away the influence of the operating environment, which often distorts theoretical predictions. Comparisons of alternative designs must always be done with empirical baselines.

3. **Interpretation of metrics.**

Maintaining real-time behavior (i.e. *time sensitivity*) through the experiments is essential in order to validate *scalability* results. However, operating system metrics such as CPU, memory, and bandwidth, may not represent *time sensitivity* appropriately, and, if used, may lead to misinterpretations. In order to interpret them correctly, it is essential to understand the internals of the software being assessed.
For example, it is tempting to assume CPU load is a measure of processing load and that, eventually, an overwhelmed program will reach 100% CPU. However, modern CPUs are multicore, and many applications these days are multi-threaded. In OpenSimulator, there are two long-lived main threads (one of them being physics simulation), and one additional thread per connected user. Some of these threads may spawn several short-lived threads as they process events from various sources; physics, however, does not spawn threads. Hence, even when the physics simulation is overwhelmed, not all cores are used; as a result, the maximum CPU% on an 8-core machine (as in our case) was never reached. In Figures 6.2b and 6.2d, CPU usage tops at around 40%, independent of how much the physics workload was increased. This result only makes sense when there is a deep familiarity with the code of OpenSimulator, specifically its concurrency model, which we briefly described here.

In the case of OpenSimulator and the extensions studied, the CPU numbers reflected a combination of computing tasks, some of which were from physics, some of which were not. As such, CPU usage did not quite capture the performance issue we were looking to measure. In order to isolate and measure the processing time allocated to physics, we needed another metric; we used the time the balls take from creation at the top to destruction on the floor. When physics is overwhelmed, this interval increases.

In general, operating system metrics may not capture what is important to measure. When those generic metrics are used, they need to be interpreted according to in-depth knowledge of the software, or they may be misleading. In many cases, application-specific metrics become necessary.

4. Dependent variables and masking.

In both DSG and DSG-M, ball creation was done in the script simulator, and ball deletion was done in the physics simulator(s). By design, the script simulator was never overloaded on any experiment, and dropped balls at a constant speed. The physics simulator(s) eventually
became overwhelmed with the number of balls, and slowed down all physics operations, including object deletion. A slower simulator affected time sensitivity: the simulation started running slower than real-time. This can be seen in Figures 6.2a and 6.2b; these Figures show an overwhelmed physics simulator in DSG taking increasingly longer time to delete the balls (Figure 6.2b), which results in an increasing number of balls staying in the scene to be physically simulated (Figure 6.2a) – a loop that produces super-linear growth of both interval and number of balls, until the simulator crashes.

In a non-distributed, single-threaded architecture this would not have happened. Instead, a slow down of physics (and deletion of balls) would also slow down the scripts (and creation of balls), resulting in a constant number of balls to be physically simulated, even when the simulator would be overloaded.

Interestingly, we observed a constant number of balls in the scenes for the DSG-M experiment (Figure 6.2c), where there were two physics simulators, both of them overloaded (Figure 6.2d). At first, we interpreted this as a strongly positive result for DSG-M: it looked like DSG-M was capable of handling the load under stress much better than DSG. But this result was puzzling: how could the experiment for two overloaded simulators show such a different result from that with just one overloaded simulator, considering that in both cases the data clearly showed an increase in the interval between creation and deletion? Where were those “zombie” balls being processed?

The culprit behind this puzzling result was the network. In DSG-M, the two physics simulators exchange many balls that are moving at their border, and that creates a much higher network traffic than in DSG. Figure 6.3 shows messages from the dispatcher component to one of the physics simulator being increasingly queued as the physics simulation unfolds. These messages correspond to exchange of balls (from the other physics simulator) as well as creation of balls (from the script simulator). The growing queue size meant that new balls added to the Galton box took longer to arrive, resulting in less balls to simulate. But as
the network got progressively worse, so did the mean time between creation and deletion. Inadvertently, the overhead networking cost was masking the scalability results, leading us to believe that DSG-M was much better than what it was in reality.

These situations, unfortunately, are not uncommon when dealing with DVEs. The activation of certain behaviors may result in a complex cascade of effects, some of which may mask others, leading to erroneous conclusions.

5. **Third-party defects.**

Often, the operating environment of DRTs includes a variety of third-party components, and these can be problematic. In our case, when we ran experiments under very high load, we ran into a problem that made the simulators crash at the end of the experiment. We assumed there was a bug in OpenSimulator or in DSG / DSG-M, and spent many hours trying to find it. Eventually, we concluded the bug was in the memory management of the Mono framework [1]. By recompiling a more recent version of Mono with an extra flag, and
using the right garbage collector, the simulators stopped crashing.

These situations, unfortunately, are also not uncommon. Most software, these days, stands on the shoulders of a very large number of 3\textsuperscript{rd}-party frameworks and libraries, over which there is very little control. When a failure happens, there is a large number of potential culprits, not just in one’s own code, but in the code of the entire operating environment.

### 6.4 Case Study 2: Login Procedure

This case study concerns a systematic profiling to assess and control the impact of user login on OpenSimulator server performance. OpenSimulator users and developers had observed that user login was a heavy activity that suffered from lag, but the cause of lag was unknown. Like in the previous Case Study, we describe the rationale behind the experiments and metrics, and conclude the section with observations regarding the challenges encountered.

#### 6.4.1 Objective

The main objective of this study was to isolate the cause(s) of perceived lagging (time sensitivity), and to mitigate, or even improve, OpenSimulator once those causes were identified. In this case, the users’ perception of lag correlated very strongly with the operating system’s CPU metric, as high CPU load was always observed upon user’s login. This, in turn, undermined the ability for OpenSimulator to scale to a large number of users during simultaneous logins, as the server became too busy to be able to process the login requests within an acceptable time frame.

The secondary objective was to develop test scenarios related to users login that could be ran automatically, and ensure that future changes to OpenSimulator would preserve an
acceptable behavior upon users login on the part of the simulator.

To login to an OpenSimulator simulation server, a user enters a server URI and credentials into a client viewer. Once the login is authenticated, the user’s client needs to download a large volume of virtual entities so that its local virtual environment state is consistent with the server’s state. This means that the user entering the system demands a high volume of resources from the simulation server, thus limiting bandwidth resources available to other logged in clients. The client caches many of the assets between sessions, and this somewhat lowers the login load upon subsequent logins. We were interested in the login procedure in the worst case scenario, i.e. the first time the user logs in to a certain virtual space that they have never visited before. Figure 6.4 shows privileged processor time (roughly, CPU%) accumulated in one example login into OpenSimulator. Three stages of login are identified in the figure: (1) an initial load spike, (2) content retrieval, and (3) return to steady-state levels. The underlying reasons for the initial performance spike and content retrieval load were unknown.
Table 6.1: Login Experiment Configurations

<table>
<thead>
<tr>
<th>Factor</th>
<th>Configuration</th>
<th>Size 1</th>
<th>File Contents 2</th>
<th>Graphics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avatar Weight</td>
<td>Light Avatar</td>
<td>0.33 MB</td>
<td>136 items</td>
<td>Figure 6.5a</td>
</tr>
<tr>
<td></td>
<td>Heavy Avatar</td>
<td>1.1 MB</td>
<td>183 items</td>
<td>Figure 6.5b</td>
</tr>
<tr>
<td>Inventory Size</td>
<td>Light Inventory</td>
<td>0 MB</td>
<td>0 additional folders and items</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Heavy Inventory</td>
<td>20.6 MB</td>
<td>8,977 folders with 31,986 items</td>
<td></td>
</tr>
<tr>
<td>Scene Complexity</td>
<td>Light Scene</td>
<td>0.038 MB</td>
<td>2 scene objects, 2 assets</td>
<td>Figure 6.5c</td>
</tr>
<tr>
<td></td>
<td>Heavy Scene</td>
<td>185.4 MB</td>
<td>238 scene objects, 1171 assets</td>
<td>Figure 6.5d</td>
</tr>
</tbody>
</table>

1. All inventory and scene formats gzip compressed
2. Scene contents computed with oarinfo.py utility [39]

6.4.2 Experiment

Figure 6.5: Avatar and scene configurations from login study. (a) Ruth, light-weight baseline avatar (left); (b) Alien, heavy-weight avatar; (c) Light scene; (d) Heavy scene [136].

Our in-depth knowledge of OpenSimulator indicated that three factors might impact server performance during login, namely:

1. **Avatar Weight**: The complexity of a user’s appearance, including textures, skins, and scripts.

2. **Inventory Size**: Inventories are file-folder structures containing virtual objects that
belong to the user.

3. **3D Scene Complexity**: The objects, scripts, textures, and meshes contained in an OpenSimulator region.

As such, we designed an experiment meant to measure the effect of each of these factors, independently, towards CPU load. Table 6.1 summarizes the experimental configurations, with a “light” and a “heavy” configuration for each factor. Figures 6.5a and 6.5b show the light and heavy avatars; figures 6.5c and 6.5d show the light and heavy scenes. The experiment consisted in measuring CPU load upon one user’s login under the 8 scenarios resulting from the complete combination of configurations, i.e. light avatar + light inventory + light scene, light avatar + light inventory + heavy scene, etc.

Possible compounding factors were eliminated. First, the client’s cache was always cleared in between experiments. Second, given previous work concerning performance impact of avatars in OpenSimulator [67], we know that the avatar’s actions in the world have a measurable impact on server load, as measured by CPU; for example, standing vs. seating have different performance profiles, because a seated avatar is removed from the physics simulation; walking vs. standing also have different performance profiles; etc. In this study, the avatars remain standing and do not interact with any object on the scene, so that confounding processing load is minimized.

6.4.3 **Setup**

OpenSimulator was configured in a “grid” configuration, where the space simulation server is separated from the central resource server that serves the login request as well as many resources stored in a database. As such, the set up for the experiments consisted of three networked components: (1) the client; (2) the space simulation server; and (3) the central
resource server. The architecture of OpenSimulator is such that the space simulation server always proxies the access to backend resources, such as inventory and textures – i.e. the client never accesses the central server directly, except for the initial login request.

A single avatar was logged into the region server using the Singularity Viewer [160], an open source client for OpenSimulator. The viewer was configured with high graphics quality and a large draw distance to simulate a full view of the virtual scene. Each experiment run lasted for 600 seconds (10 minutes), starting from the time at which the region received a login request for the user. This time was chosen because most inventory configurations loaded within 10 minutes. Five experiment runs were performed for each combination of avatar weight, inventory size, and scene complexity — 40 runs in total. The built-in OpenSimulator logging and monitoring utilities were used to record data.

The experiments were conducted on wired machines on the UC Irvine network. Test avatars were logged in to the OpenSimulator server from a laptop with Intel i5-2520M CPU (2.50GHz), 4 GB of RAM, and an Intel integrated graphics card. The OpenSimulator simulation servers in this study were hosted on a Dell machine with an Intel i5-4670 CPU (3.40 GHz) with 4 cores and 8 GB of RAM. The machine ran the Ubuntu 12.04 LTS operating system. Monitoring and statistics logging occurred on this machine.

### 6.4.4 Metrics and Results

The primary metric was accumulated privileged processor time used by the simulation server. This metric is computed within OpenSimulator’s monitoring component, and it measures CPU usage by the server process over time. Two metrics were later collected for debugging purposes: the quantity of inventory folder requests received by the server, as well as the quantity of HTTP packet requests received by the server.
Figure 6.6: Simulation server privileged processor time, comparing light inventory (left) and heavy inventory (right) configurations.

After conducting five tests for each combination of login configurations (6.5 hours of combined tests), it appeared that the size of the user inventory had the highest impact on performance load. Figure 6.6 shows average load measurements between configurations with light inventories and heavy inventories. We observed that configurations with heavy inventories resulted in many server requests for nested inventory folders. The impact of the avatar complexity seemed to be negligible. The scene complexity had some effect, but not as much as inventory size. We also observed a puzzling performance profiles in two of the experimental configurations that are discussed next.

In order to further study the performance issue with inventory, we made additional experiments where we added a fourth component to the setup, specifically a dedicated inventory server. This server was configured to handle all inventory folder requests directly from the client, meaning that inventory retrieval was no longer proxied by the simulator server. The goal of this additional experiment was to verify whether removing inventory service altogether from the simulation server would bring CPU usage to acceptable levels at the simulator, or whether there was something more complex going on. We conducted all experimental runs again with the added component in the experimental setup.

Indeed, adding a dedicated server for inventory retrieval reduced server load to levels comparable to those obtained for light inventories. Figure 6.7 shows this additional result. This
Figure 6.7: Simulation server privileged processor time with a dedicated simulation server for inventory, comparing light inventory (left) and heavy inventory (right) configurations.

meant that inventory servicing at the simulation server was the sole cause of the observed high CPU usage. That, in turn, gave us the necessary confidence to start solving the problem by looking at the code that handled inventory servicing at the simulator. We found very problematic code, changed it, and eventually fixed all these issues with initial inventory downloads.

### 6.4.5 Observations

1. **Time sensitivity.**

The experiment was designed to profile a specific user activity (login) that consistently showed lag for users. “Lag” is an informal term used in DVEs that denotes situations where the interaction with the environment *feels* slower than expected. As such, by definition, lag is a perceptual phenomenon; it may correlate with system-level metrics in complex ways, or not at all.

In this case, there was a very strong correlation between the lag felt by users and CPU usage at the simulation server. Clearly, the code executed at login was making the CPU busy. By focusing on measuring CPU and, eventually, decrease its usage during login, we hoped to decrease the lag felt by users. This was a hope that might or might not come to fruition, as
lag and CPU usage are not the same thing.

After these profiling experiments, we improved the login code of OpenSimulator considerably, reducing CPU usage to a small fraction of what was measured in these experiments. Fortunately, these improvements resulted in a considerable reduction of lag too. We measured this reduction in lag qualitatively, by releasing the fixes made to OpenSimulator to the community and requesting feedback from them.\(^3\)

Similarly to observations made regarding correctness of simulation in Case Study 1, the user experience is the most important aspect DVEs, and that is, ultimately, what needs to be measured. However, not only user experience is hard to measure, but it becomes impractical to measure it \textit{while} developing these systems. For example, it would have been highly ineffective to ask independent subjects to check the lag after every important code commit that seemed to reduce the CPU usage. System-level metrics are much easier to measure, but they might or might not correlate with the observable effects that matter to users.


This study exposed two puzzling performance profiles, all related to heavy inventory configurations – see Figure 6.6, right bar chart. One of them pertained to the configuration heavy inventory + heavy avatar + light scene. Those experiments had a very wide variation in CPU usage, as seen in the error line (second bar from left). The second one pertained to the configurations heavy inventory + light avatar: the experiments with the light scene (first bar from the left) showed higher CPU usage than the experiments with the heavy scene (third bar from the left); this was counterintuitive.

After measuring a couple of other internal quantities, we concluded that both of these situations could only be explained by the existence of bugs in the code of OpenSimulator.

\(^3\)See http://opensimulator.org/mantis/view.php?id=7564
However, these weren’t functional defects related to correctness of behavior, as the function performed by the simulator was essentially correct – inventory was downloaded by the client, eventually. These were defects affecting the non-functional properties of OpenSimulator. In one case, the CPU load was very unpredictable; in the other, something was making the CPU unreasonably busy on light scenes when compared to heavy scenes.

Non-functional defects are much harder to deal with than functional defects. First, a specification usually does not exist upfront, not even an upfront expectation of correct behavior; “I know something is wrong when I see it,” seems to be the main approach to identifying these issues. For example, we can define upfront the inventory download feature, as that is a fundamental part of the login procedure, but it is much harder to define upfront the non-functional property related to variance in CPU usage of inventory download, because it is one of a possibly unbounded list of non-functional behaviors. Second, non-functional defects may show up only when certain conditions are met, making them very hard to reproduce. For example, while we were fixing the inventory download issue, it became apparent that the distance between the simulation server and the central server had a significant impact on this defect, something that caused a fair amount of confusion; for a certain OpenSimulator grid whose central server is hosted in a US data center, some users in Europe experienced issues that users in the US could not reproduce.4 Finally, once the non-functional defects become apparent, it is much harder to develop regression tests for non-functional properties, such as those exposed by these two puzzling performance profiles.

While all software is affected by non-functional properties, DVEs, and DRTs in general, are particularly exposed to them, as the existence of distributed components and the expectation of real-time interaction pose difficult challenges in terms of identifying non-functional defects, reproducing them, and making sure they do not come back once they are fixed.

3. Masking, again.

4See, for example, http://opensimulator.org/mantis/view.php?id=7567
Distributed non-deterministic interaction between components may lead to masking of known and unknown incorrect behaviors. We encountered this already in Case Study 1, when queues acted as buffers of the balls and spared CPU from having the expected load on the physics simulators. Here, too, we observed masking of incorrect [non-functional] behavior when we moved the inventory service to a separate server. By doing that, we eliminated the abnormalities described above, but the defects were still there in the code. They just became invisible.

As we fixed the code, it was clear that the API of the inventory service was highly inefficient. When the calls were on the same component (such as in the case of a dedicated server) those inefficiencies were not noticeable; however, when the calls came from a component on the network, those inefficiencies became visible, and produced the results we measured in these experiments.


The profiling of any software requires its execution, as well as the triggering of specific inputs. Our profiling experiments were labour-intensive: for each of the 45 measurements we had to start 3 components (4, in the case of the dedicated inventory server) and then login a user manually. This was clearly not ideal, but we had no other option: we didn’t know where the cause of the high CPU was, and it could partly be the graphical client, Singularity – a very large and complex piece of software that we treat as a black box and for which there is no headless version.

Once we were confident that the cause was a non-functional bug in OpenSimulator, we then replaced the graphical client with a very simple headless client that only downloaded the inventory. This allowed us to reproduce the high CPU usage at the simulator without having to deal with the graphical client. But the process of starting and stopping components was still manual. Also, we have not been able to add any regression tests regarding this issue,
as that would require a framework for distributed testing of non-functional defects that OpenSimulator does not have.

It would be desirable to develop a framework for automatic testing of these specific distributed components, particularly tuned for testing application-specific non-functional behaviors, but that is challenging goal. To the best of our knowledge, that does not exist.

6.5 Historical Perspective

This section examines the challenges of DVEs in greater depth, and offers a broader historical context for the observations we made about our experiments. Many of the challenges we encountered in these two case studies have been analyzed in the literature. The goal of this historical perspective section is to argue that the difficulties in designing, testing and evaluating DVEs are inherent, and not just the consequence of inept engineering. New ideas for addressing them are needed.

6.5.1 On Design and Evaluation

We discuss the unique characteristics of DVEs that impact their design and evaluation.

On Correctness

Since the early days of computing, correctness has had a well-established definition: an algorithm, or a system, is correct if it honors its specification (see, e.g. [78,145]). Functional correctness pertains to the input-output behavior of the algorithm or the system. This commentary focuses on functional correctness, but we use the word “correctness” for brevity.

The size and complexity of what can be proven correct has been growing at a steady
pace [145], and it is conceivable that in the future extremely complex systems like OpenSimulator could be formally specified and verified; we are still a long way from that. It is not our intention to cover the impressive progress of formal verification techniques of recent years, including for real-time and distributed systems [69, 100, 115]. Instead, we want to discuss the definition of correctness provided above, how it interferes with other design concerns of DVEs, and how researchers and developers have been coping with those interferences.

In discussing Case Study 1, we mentioned that correctness of a DVE can be seen under two perspectives: the user and the function itself. In the case of a physics simulation, like in our experiments, it is desirable that it is as realistic as possible – ideally simulating exactly physics in the real world. But such goal carries with it a heavy demand on computing power, and it becomes unachievable in practice. In order to keep the simulation’s performance under control, developers of physics simulation engines simply make better-performing numerical approximations of real physics everywhere they can (see e.g. [11]). In doing so, the simulation deviates from the correct behavior. As observed in Case Study 1, when simulating a Galton Box, the physics engine in OpenSimulator produces a result that does not match the correct functional behavior dictated by the laws of physics in the real world, which led us to having to use an empirical baseline.

Physics is not the only aspect of DVEs that is subject to correctness-degrading approximations. Performance and responsiveness of these systems usually take precedence over correctness. This gives rise to a different, and more recent set of metrics for assessing correctness, those that are based on human perception [42, 110, 149, 192]. The basic premise of this line of work is this: if humans cannot tell the difference between an exactly correct behavior and an incorrect behavior that requires less computing resources, then the latter is preferred. This makes up for substantially different DRT systems than those traditionally envisioned in DRT research.

On Time Sensitivity and Consistency
In Case Study 2, we mentioned that users of OpenSimulator reported lag upon certain logins. In OpenSimulator, lag is usually felt in terms of physics – e.g. walking (of self and others) is not smooth, collision detection is slower than expected – and in terms of the environment’s response to their inputs – e.g. clicking on an object to trigger some effect produces that effect much later than expected. The workload generated by graphics rendering and event propagation can lead to latencies over 3 times larger than video streaming [151].

Real-time systems need to perform operations in a time sensitive manner. Sha et al. [155] give a comprehensive overview of the history of the major developments in real-time scheduling, including for distributed systems; we refer the reader to that interesting paper for the historical perspective on dealing with time sensitivity in computing systems. DVEs, in particular, are designed under the expectation that their responsiveness matches, to some approximation, the speed of interactions in the real world [165]. However, unlike hardware control DRT systems such as anti-lock brakes or the control of a rocket, DVEs are interactive systems ultimately used by people; as such, and as mentioned before for correctness, it is important to include the human perceptual system as a parameter of the design and assessment. This inclusion has two consequences for design: compensation for delays and variance, and opportunity for optimizations.

On the one hand, the network introduces latency and jitter. This has been known for a long time in the engineering of DVEs; well-known solutions to these problems include efficient server placement algorithms [167] and several prediction techniques such as the centuries’ old dead reckoning [163] applied to these environments [81, 141]. These techniques help in improving responsiveness and in preserving the illusion of consistency among the distributed components, even if the system is not exactly consistent.\(^5\)

On the other hand, the perceptual effects of latency and jitter have been studied more recently in the research literature with the goal towards devising optimizations that improve

\(^5\)See e.g. [7] for a good overview of latency compensation techniques.
the perceived responsiveness of DVEs [37, 48, 80, 147].

Just like for correctness, this line of work in perceptual metrics is very important for DVEs, as many more opportunities for optimization will likely be found that will make these systems more scalable. However, equally important is the mapping of such metrics to system-level metrics, as user studies carry an unbearable overhead during development of the system.

**On Scaling and Overhead Costs**

Jim Waldo stated [185]: “Online games and virtual worlds have familiar scaling requirements, but don’t be fooled: everything you know is wrong.” This is an overstatement, but it is true that the focus on user experience in DVEs requires us to rethink many of the concepts we took for granted regarding DRTs.

A large virtual environment is usually associated with a large workload. If the virtual environment has thousands of users and objects, computing the result of each interaction at every time step is unfeasible within the limited time frame required for reasonable interactivity, usually in the hundreds of milliseconds. DVEs typically partition this workload into multiple simulators by virtual space, with simulators being responsible for unique areas of the virtual environment. This idea can be traced back to Locales [10] and DIVE [63], and it has been the main architectural approach to scalability of massive multi-user environments.

Many improvements and variations of this idea have been proposed over the years. For example, microcells of custom size and shape add flexibility to adapt the load among machines dynamically [45]. A push-pull framework can be used to filter and reduce the number of messages exchanged between partitions [128]. Another variation, sharding, is a form of space partitioning where different users connect to different copies of parts of the space.\(^6\)

---

\(^6\)Unfortunately, the origin of the word *sharding*, and corresponding technique, seems to have been lost in history. It likely came from the game Ultima Online launched in 1997, which may have been the first one to provide multiple copies of game spaces for different groups of users.
Load partitioning among servers is, therefore, the only way a virtual environment can scale. However, space is not the only aspect of these environments that can be partitioned. The DSG architecture, for example, partitions the workload by functionality, such as physics and script execution [102,112,113]. Project Darkstar [185] divides the load by task triggered by user input. Kiwano [47] divides the world by space, and groups the load generated by the users’ input by their proximity in space. Meru [85] partitions the load by both space and content (3D objects).

The art of designing scalable DVEs lies in finding the right load partitions for the purpose for which the DVE is being designed. Load partitions carry overhead costs in terms of coordination among servers. In a “good” partition design, the system will scale horizontally, i.e. more load can be handled by adding more servers in as linear a relation as possible; in a “bad” partition design the benefits of load distribution will be dwarfed by the communication overhead among servers. For example, in our worst-case scenario experiments in Case Study 1, only a 15% improvement was observed when dividing the space in two; nearly 85% of the computation was being used for the overhead of synchronizing the simulators. The overhead was mostly due to object migration causing numerous messages related to the creation and deletion of tens of thousands of objects.

Finding the appropriate load partitions for a DVE requires a considerable amount of experimentation, of the kinds we did for DSG-M. Ideas that look good on paper often fall short when placed into an actual system. Scalability in DVEs is still very much a topic of research. One of the confounding factors in this research area is the absence of common objectives among the different systems. They all claim to scale, but the applications for which they are being designed, and therefore the metrics they use to assess scalability, are all very different.

In general, the volume of concurrent user interactions and the complexity of the shared artifacts are the two most important factors governing approaches to scalability [49,96,99, 113,159]. Given that different applications have very different demands of user-user and user-
environment interactions, it follows that common metrics are hard to find. The development of DVEs has been primarily driven by commercial interests and realized by skillful engineers. Research in these systems, so far, has been fairly ad-hoc. Calls for making it more systematic are now starting to appear [159].

Giving a positive spin to Waldo’s observation, there is a fair amount of work to be done in categorizing dimensions of scalability in DVEs, and in producing benchmarks and metrics that can be used to compare different solutions for the same categories of problems.

On Faults, Fault-Tolerance and Fault Prevention

The concept of fault tolerance emerged a long time ago, and for a while, it remained associated with hardware design [143]. As software became more pervasive and important, those same ideas were adopted for software systems [148]. For a long time, software fault-tolerance meant almost exclusively the existence and management of “stand-by sparing” components; it eventually grew to encompass a much larger scope of concepts and techniques, such as fault detection and fault models (see, e.g. [43,88]). Distributed systems, in particular, are fundamentally unreliable [133]. Our case studies do not illustrate the traditional concepts of fault tolerance very well, those where possible faults are known in advance and mitigated in some way; but the effect of non-functional, unanticipated software faults was prevalent in our observations.

The existence of these unanticipated faults has been discussed in the literature for a long time. For example, as early as 1990 Lee and Anderson wrote in their textbook [104]:

"While the anticipation of faults has been successful in the past for hardware, the present trend towards very large scale integration is already casting doubt on the validity of the assumption that all component failure modes are known.[...]
It follows that unanticipated faults are likely to arise in hardware systems, and
will certainly require tolerance in high reliability applications.

However, there is a much more important and insidious reason for the occurrence of unanticipated situations. If there are faults in the design of a system then the effects of such faults will be unanticipated (and unanticipatable!) [...] While design faults may have been uncommon in hardware systems, the only type of fault that can be introduced into a non-physical system, such as a system implemented in software, is a design fault [...] Applications of the fault prevention techniques that have been successful for exposing design faults in hardware systems have only met with limited success when applied to software.”

This text is as valid today as it was in 1990. Since then, a great deal of progress has been made in capturing functional behavior of software. The area of software testing, for example, has seen an enormous growth (more on this later on). More recently, the area of formal specification and verification has also gained considerable attention.

Unfortunately, very little progress has been made in formally identifying and capturing non-functional behaviors, such as those we described, to the point of preventing non-functional defects from occurring. Additionally, very little progress has been made in formally identifying and capturing operational expectations from 3\textsuperscript{rd}-party components to the point of preventing defects in these components from causing our own systems to fail. DVEs, and DRTs in general, are particularly exposed to these two kinds of hard software engineering problems, as they tend to rely on a large stack of 3\textsuperscript{rd}-party components, and a considerable portion of their existence is determined by how they do the things they do (i.e. operation rather than function).

A lot more needs to be done in addressing the unavoidable existence of design faults, especially in what concerns non-functional aspects of the designs.
6.5.2 On Testing

When discussing the non-functional defect studied in Case Study 2, we mentioned that one unfulfilled goal of that work was to write some sort of regression test that would ensure that specific unreliable behavior with inventory download will not come back again. Regression tests are standard practice in software development, and OpenSimulator has hundreds of them. Unfortunately, when it comes to identifying and testing the occurrence of known non-functional defects, especially in distributed systems, the research literature is scarce.

The research field of software testing can be traced back to the early 1970s (for a good overview, we refer the reader to [14]). Most testing research and development has focused on functional, non-distributed testing. Once described as a “lost world of debugging and testing” [75], distributed systems always lagged behind relevant developments in testing and debugging. Unfortunately, the situation has not improved much: while the area of testing has seen enormous progress in the last 30 years [140], distributed systems testing is still lagging behind non-distributed systems testing. Unit testing frameworks, code coverage frameworks, fault injection and all sorts of test input generators have made the transition from research to practice, but up to today, we continue to see research literature refer to testing of distributed systems as “a challenge” [31, 79].

The challenges in distributed real-time systems testing were first clearly formulated by Schütz in a paper that is still as relevant today as it was in 1994 [154]. This work identified six fundamental issues in DRT testing: organization of test phases; observability of DRT systems; reproducibility to cope with non-deterministic behavior; splitting testing between development hosts and target systems, environment simulation to support real-time correctness, and representativity of realistic inputs. Schütz’s paper is still a solid blueprint not just for the challenges of testing DRT, but it also inspires potential solutions for those challenges.

Indeed, in recent years there has been some progress in distributed systems testing [31, 44,
The most recent advances in software testing for distributed systems include model checking (explored in depth in [95,109]) and capture-replay testing [91], which can be traced back to Tsai’s work in non-intrusive real-time system testing and debugging [176].

We note, however, that our faulty inventory download behavior does not fit well in Schütz’s framework, specifically in what concerns reproducibility. We know that part of the inventory bug was non-deterministic; its non-determinism was not about the function itself (there was no functional defect) but about CPU usage. Furthermore, even though the defect was non-deterministic, it is possible to describe it using a statistical specification: part of the bug was that it produced a wide variation in CPU usage between independent login sessions. Clearly, if that happens again after our fixes, it means that there is a regression. Schütz’s description of the reproducibility concern did not take into account the existence of these kinds of tests that require launching the execution several times and observing the statistics of some metric. We believe that repeated experimentation is an important aspect of testing DVEs and DRTs in general, especially when it comes to testing non-functional properties.

This leads us to the most relevant literature for the testing problems we describe in our case studies: testing non-functional properties of software. A paper by Weyuker et al. published in 2000 notices the almost complete lack of literature related to performance testing at the time [188], contrasted with how important the problem was/is in industrial projects. They then describe a case study within AT&T, explaining how performance tests were specified. This included having to collect data in order to establish realistic workloads for the tests – similarly to our profiling work and the setup of the tests. They also make this insightful observation:

“It is essential to recognize that designing test suites for performance testing is fundamentally different from designing test suites for functionality testing. When doing functionality testing, the actual values of parameter settings is of
utmost importance. For performance testing, in contrast, the primary concerns are the workloads and frequencies of inputs, with the particular values of inputs frequently being of little importance.”

Since then, there have been a few more papers on the topic. Denaro et al. [46] propose deriving and running performance tests from early architectural decisions, even before the software is written. Their assumption is that “[...] performance measurements in the presence of the stubs are good enough approximations of the actual performance of the final application.” They provide some preliminary evidence regarding one use case of one architecture in a J2EE tutorial, but this assumption is difficult to accept in general.

In a study of unit testing non-functional properties of distributed systems [83], Hill et al. observe that many non-functional DRT properties, not just performance, are often relegated to system integration stages of software development, and that conventional system execution modeling tools do not provide the necessary support for unit testing non-functional requirements. They then propose their approach to testing non-functional properties, which is based on mining logs.

As Weyuker et al. noticed in 2000, the lack of research literature on the subject does not reflect the importance of the subject in industry, especially for Web applications. That importance can be seen in the existence of several developer-oriented books on the subject, e.g. [54,129]. These books tend to focus on the practicalities and tools of performance testing, and do not offer suggestions for how to advance the state of the art.

6.6 Lessons Applied in CADIS

This chapter has provided a broad understanding of the challenges in evaluating and testing DVE. We have uncovered how little consistency there is in evaluating DRT systems in general,
and how it has affected our own experiences in designing DIS architectures. The broader study and lessons presented in this chapter ultimately culminated in the evaluation work for CADIS in Chapter 4. This section makes the parallel between lessons learned and lessons applied within the context of designing, testing, and evaluating CADIS.

6.6.1 Objective

Similarly to both case studies in this chapter, CADIS is focused on scalability – the capability of the architecture to adapt to new requirements of size and scope [169]. However, CADIS has a different design goal than our previous DVE case studies. While in the case studies the goal was to improve the simulation size (i.e. number of participants and simulated entities), CADIS is concerned with improving the scope – the capability of adding functionally distinct simulations to the system. OpenSimulator assumes a singular development group for the platform, thus restricting new functionality scope to scripting when adding new simulations. DSG-M, by contrast, encourages the development of new actors to increase the scope of the simulation. However, DSG-M also abides by the internal architecture and data structures of OpenSimulator, as for instance following the LLUDP networking API for communication. Additionally, since our contribution to DSG-M was its space partitioning solution, an emphasis on evaluating scalability of size, performance, and correctness was preferred over its original scalable scope design goal.

CADIS imposes no restrictions to the architecture of each participating simulation. There is no standard object definition nor previously existing simulation services. The only protocols and data type definitions in CADIS exist to provide state synchronization and dependent type functionality. Understanding how CADIS fits within the six dimensions of concerns helped us understand what needs to be evaluated. But that in turn presents another challenge: how

\cite{169}OpenSimulator also allows region modules that give access to internal representations, but even those are limited to following an internal architecture and data types.
to evaluate a system in its capability of increasing scope?

6.6.2 Choosing the baseline

In the first case study, choosing the correct baseline scenario was one of the complications of designing an evaluation. Instead of adhering to the theoretical expected results that were not achievable with OpenSimulator, DSG-M was compared to its DSG counterpart. Setting the gold standard to be a known empirical measurement allowed DSG-M differences to be highlighted.

The evaluation in CADIS followed the same principles – setting an empirical gold standard that highlights what our framework does differently. In the feasibility evaluation, the urban simulation mobdat was used as the gold standard for a DIS. The same mobdat simulation was used on CADIS, replacing only the event delivery mechanism. The original mobdat was exclusively event-based, and the CADIS version was exclusively data-oriented, using PCC data types synchronized by spacetime. The result was an evaluation that focused on what CADIS does differently, eliminating common factors such as programming language, architectural design of the simulation, and networking protocols used.

6.6.3 Feasibility vs Isolated Evaluation

Our experience in the second case study hinted that the mobdat evaluation study was not enough to demonstrate CADIS’s capability of scaling. The objective of the second case study was to map individual conditions of the system (i.e. inventory size, avatar weight, and scene complexity) to their effects in terms of relevant metrics. In the mobdat evaluation, like in DSG-M galton box, that information is lost, since there are too many variables to determine the impact of the individual parts of the architectural design. The feasibility
study demonstrated the capability of achieving our objective – an urban simulation – but did not provide an understanding of the impact of each design decision in relation to that goal. A good evaluation should show that each architectural design decisions contribute to its design objective.

Thus, we built a benchmarking platform for CADIS that resembles the evaluation work done in case study 2. The purpose of the benchmarking platform is to demonstrate that PCC types enables scalability in size as well as in scope. The benchmarks measure the cost of each PCC operation in isolation. These results are vital so a simulation developer understands the trade-off between the benefits of PCC modularity and the cost in recalculating and transmitting new data types.

6.6.4 Metrics

In both case studies, hardware metrics like CPU and memory were initially chosen as metrics for evaluation. In the first case study, this proved to be a problem: the CPU load was an and confounding metric that provided a mixture of symptoms. In the second case study, another issue derived from confounding CPU measurement – an unexpected high load of the CPU in a scenario that did not have a such a high workload. In both complications, we referred to these issues as masking of dependent variables: when multiple symptoms affect the same metric, rendering the measurements useless in terms of explaining the desired phenomenon.

It is temping to measure performance in terms of hardware metrics, they are easier to relate, simple to measure, and are apparently correlated with workload. Previous case studies, however, have shown us that application-specific measurements are superior in isolating certain workload behaviors. In the first case study, the time-to-drop a ball in the galton box from top to bottom was preferred, as it relayed exactly the increase in system load due to the corresponding increase in the workload.
The spacetime framework of the CADIS architecture has a convenient simulation loop with 3 operations: pulling new data, simulating a step, then pushing state modifications back to the spacetime store. As each simulation need to stay within its time step deadline, the most representative metric for measuring workload is how much time is taken to perform each of these operations, and most importantly, the sum of them. In the mobdat experiment, the time spent on each of the operations was the chosen metric for measuring performance. After a brief explanation of the responsibilities and data requirements of each of the simulations, it was possible to see the impacts of the CADIS architectural design decisions for each simulation based on their individual workload. The social simulation, for instance, was mostly idle, since it pulled no data types and only produced new cars with low frequency. Meanwhile, the sumo and opensim simulations had the heaviest workloads, as vehicles had to be updated at every tick. The time spent in push and pull perfectly reflected this dynamic: sumo had long push times and opensim had long pull times.

For the same reasons, the benchmark in CADIS used push, update, and pull elapsed times as a metric. Understand the dynamics between sending large amounts of data, processing queries with large number of objects, and serializing/deserializing objects is the objective of the benchmark. Time spent in pushing, updating, and pulling objects in isolated test cases provided values that accurately describe the resulting workload of each individual pcc operation and the effects of creating and objects over time.

6.7 Conclusion

Distributed Virtual Environments (DVE) are Distributed Real-Time (DRT) systems designed with the general goal of connecting multiple users over the Internet instantly with each other and with a shared virtual space. DVEs inherit some of the intrinsic difficulties of DRT systems, such as the overhead of distribution and the overarching importance of
responsiveness and performance. They also have unique characteristics that make them different from traditional DRT systems. Specifically, a strong focus on user experience, and the quality of that experience, requires a re-evaluation of some of the concepts in traditional DRTs.

This chapter first presented two case studies describing design and profiling work previously done by us on one DVE, OpenSimulator. OpenSimulator is an open-source virtual environment framework that uses the same protocol as Second Life, and, as such, supports Second Life-like environments. The first case study focused on assessing whether a specific design idea we had was beneficial or not. The second case study focused on profiling a specific activity (user login) that OpenSimulator users had reported to suffer from poor responsiveness (i.e. lag), with the hope of fixing it, and then making sure it would not regress with future changes to OpenSimulator. In doing both of these pieces of work, we encountered many challenges related to metrics and their interpretation, baselines, dependent variables and masking of defects, non-functional defects, and automation. We described and discussed these challenges for each case study.

In order to place our observations in a broader context, and to show that they represent foundational challenges in DVEs, we then presented an historical perspective on the design and testing of DVEs and DRTs, focusing on the pain points encountered during our experimental work.

We believe our experience with the development of OpenSimulator, and the placement of that experience in a broader context, shed some light into the open challenges of DVEs, and the kinds of problems that are worth solving.

Finally, we placed all lessons learned with previous evaluation design experiments within the context of evaluating CADIS. The experiences described in this chapter were invaluable tools to reach an evaluation that demonstrates CADIS features and costs in isolation, allowing a
simulation developer to fully understand the consequences of designing simulations within the CADIS frameworks of PCC and spacetime. This chapter contributes to the general practice of evaluating DRT systems, presenting lessons learned from 2 valuable post-mortem case studies followed by lessons applied to our novel architecture presented in Chapter 4.
Chapter 7

Conclusion and Future Work

The computing power advances in the last decades have enabled classes of applications that could only be dreamed of in the early days of computer science. Larger memories, faster CPUs, and increased network bandwidths with lower latencies were essential advances to the networked world of applications that we currently experience. It is the duty of academic research to look in the future for applications that will contribute to progress, even if in the current state, their feasibility may still be limited in size and scope.

This dissertation presented a novel architecture for building collaborative simulations called CADIS – Collaborative Aspect-oriented DIS. After a brief introduction summarizing the goals of CADIS, Chapter 2 presented a large body of research that have for the most part disconnected from each other. Different research communities, like simulation, military, virtual environments, and GIScience are all interested in applications of collaborative simulations, and have produced many interesting approaches and architectures that we have built upon to design CADIS. In Chapter 3, the design of two previously developed DIS architectures – RCAT and DSG-M – are exposed, highlighting the strengths and weaknesses of design decisions in each. The knowledge gained in building RCAT and DSG-M were vital to
design CADIS’ novel programming model, the Predicate Collection Classes (PCC). Chapter 4 presents PCC as a new paradigm that empowers traditional OOP solutions. CADIS is detailed as a combination of two frameworks: PCC and Spacetime. PCC regulates how predicate collection classes behave and an API to use it, while Spacetime synchronizes PCC objects between multiple participants.

Evaluating the effectiveness of a DIS architecture is difficult, since the requirements vary depending on the simulation developers. Chapter 5 tackles this challenge by providing three different evaluations: a benchmark for PCC synchronization, a feasibility analysis, and a case study with graduate students. The feasibility study showed how CADIS can be applied to an urban simulation and demonstrated its capability of performing in real-time with a scenario that is similar to the real-world. The benchmark evaluation provides individual tests for each relational algebra operation offered in PCC. Such cleanroom benchmark results allow developers to understand the performance of CADIS with regard to each relational operations, like join and subsets, independent of the simulation workload. Finally, a case study is presented where graduate students without prior experience in simulations worked in a collaborative traffic simulation using CADIS. Students were able to write their traffic simulations and see them working in class in only 5 weeks of work. Considering the complexity of using and understanding distributed simulations, that shows how CADIS can greatly decrease the technical complexity of design and development of DIS.

Chapter 6 is an overview at designing and evaluating DIS architectures. There are many dimensions of software requirements that are commonly desired for DIS systems. DIS architectures must first understand these dimensions and their trade-offs, so it may be possible to evaluate whether an approach is successful. The chapter concludes demonstrating how lessons learned in previous DIS architecture design applied to CADIS.

There are still a great deal of work to make CADIS a stable platform for widespread use. One of the vital points for future work is the use of a programming language independent
PCC data type specification. Similar DIS architectures use XML and JSON – language independent data exchange formats – to achieve platform independence. PCC definitions may require formats that are more powerful than existing data exchange formats, particularly when defining queries and predicates. Using a combination of SQL and JSON could be a solution, but it remains to be evaluated whether such a simple approach is both effective and sufficient. Another vital future development is a universal way to deploy and fetch data types. A good solution is a package manager, similar to ones existing in major programming languages like Maven (https://maven.apache.org/) for Java or pip (https://pypi.python.org/pypi/pip) for Python. However, this raises new questions regarding how should PCC data type definitions evolve. Versioning is a simple and popular approach, but creates dependency issues that are known to be hard to solve in existing package managers. The goal is for PCC data types to be as easily accessible as packages are in Python and Java.

Other known concern for future work is security: secure communication channels between simulations and the store and authentication. The existing CADIS implementation requires registration, so authentication is simply a matter of adding login and password or public-private key exchange. Communications are currently done in clear-text HTTP connections, which can easily be reverted to SSL. However, other security concerns may still exist and should be studied before CADIS can be widely used as a DIS platform.

Even though collaborative simulations are still a novel idea with a small number of existing practical applications, the concept of real-time interconnected applications collaborating to produce a complex results by uniting the expertise of multiple knowledge domains provides a promising future for science. The urban simulation example used throughout this dissertation could be the future of smart cities, where traffic, parking, zoning, health, and many other concerns can not only provide real-time information, but interact to improve life in cities. Other applications are likely to emerge from Internet of Things (IoT), as information
synchronization and exchange between nodes in IoT is still a widely popular topic of research.
Bibliography


[77] H. F. Gudjonsson. The Server Technology of {EVE} {O}nline: How to Cope With 300,000 Players on One Server. Slides from a {GDC} talk, CCP, 2008.


Microsoft. Internet Information Services (IIS).


Chapter 8

Appendix

8.1 RCAT Tables

Table 8.1: Proxy Linker Connector Operations: Client API

<table>
<thead>
<tr>
<th>Operation*</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_message_to_server</td>
<td>Takes a message argument and sends to the server identified by serverid. If no server is passed, sends to any available server. Returns true if succeed.</td>
</tr>
<tr>
<td>Arguments:</td>
<td>message, serverid</td>
</tr>
<tr>
<td>Return:</td>
<td>success</td>
</tr>
<tr>
<td>broadcast_servers</td>
<td>Broadcasts message to all servers (message). Returns true if succeed.</td>
</tr>
<tr>
<td>Arguments:</td>
<td>message</td>
</tr>
<tr>
<td>Return:</td>
<td>success</td>
</tr>
<tr>
<td>move_client</td>
<td>Informs the application handler to send messages from user identified by userid to server identified by serverid. Returns true if succeed.</td>
</tr>
<tr>
<td>Arguments:</td>
<td>userid, serverid</td>
</tr>
<tr>
<td>Return:</td>
<td>success</td>
</tr>
<tr>
<td>get_server_id</td>
<td>Returns the server id of the server receiving messages from (i.e. “sticky” to) this userid. Returns null if no server is assigned to user.</td>
</tr>
<tr>
<td>Arguments:</td>
<td>userid</td>
</tr>
<tr>
<td>Return:</td>
<td>serverid</td>
</tr>
</tbody>
</table>

* Arguments in italic are optional.
Table 8.2: Proxy Linker Connector Operations: Application API

<table>
<thead>
<tr>
<th>Operation*</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_message_to_client</td>
<td>Takes a message argument and sends to the client identified by clientid. If no client is passed, sends broadcasts to all clients. Returns <em>true</em> if succeeded.</td>
</tr>
<tr>
<td>*Arguments: message, clientid</td>
<td></td>
</tr>
<tr>
<td>*Return: success</td>
<td></td>
</tr>
<tr>
<td>list_clients</td>
<td>Returns a list of all user ids from the client handler.</td>
</tr>
<tr>
<td>*Arguments: []</td>
<td></td>
</tr>
<tr>
<td>*Return: list_userids</td>
<td></td>
</tr>
</tbody>
</table>

* Arguments in *italic* are optional.
### Table 8.3: Application Connector Operations

<table>
<thead>
<tr>
<th>Event*</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{<strong>NU:userid,</strong> &lt;br&gt;<strong>SS:serverid</strong>}</td>
<td>New user identified by userid has connected. Store the user’s id and the proxy instance the message came from in a client location table. Forward the message to all application servers. If sticky routing is used, SS contains the serverid picked to treat messages from this user.</td>
</tr>
<tr>
<td>{<strong>UD:userid</strong>}</td>
<td>User identified by userid has disconnected. Remove user id entry in the client location table. Forward the message to the application.</td>
</tr>
<tr>
<td>{<strong>NS:serverid</strong>}</td>
<td>New server identified by serverid has joined the system. Store server id in the server location table.</td>
</tr>
<tr>
<td>{<strong>SD:serverid</strong>}</td>
<td>Server identified by serverid has disconnected from the system. Remove entry from server location table.</td>
</tr>
<tr>
<td>{<strong>:<em>:</em></strong>}</td>
<td>Application specific message from another application server. Forward message to application.</td>
</tr>
</tbody>
</table>

* Arguments in *italic* are optional.

### Table 8.4: Application Connector Operations

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{<strong>M:message, U:userid</strong>}</td>
<td>New client request message sent by userid, should be delivered to the application for treatment.</td>
</tr>
</tbody>
</table>
Table 8.5: Application Connector Operations

<table>
<thead>
<tr>
<th>Event*</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{M: message, U: user_list}</td>
<td>Application message intended for all the users in the user_list variable. If no users are specified, the application connector understands it as a broadcast.</td>
</tr>
</tbody>
</table>

* Arguments in italic are optional.
<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{REG:serverid, RT:boolean}</td>
<td>Registers an application server instance identified by serverid with all proxy instances. If RT is true, the proxy must broadcast a NS message to all application servers with the serverid of the newly registered instance.</td>
</tr>
<tr>
<td>{LUR:true}</td>
<td>Request for the list of users from a proxy instance.</td>
</tr>
<tr>
<td>{LSR:true}</td>
<td>Request for the list of all application and service servers from a proxy instance. For recovery purposes, since application connectors maintain track of all servers.</td>
</tr>
<tr>
<td>{BC:msg}</td>
<td>Request proxy instance to broadcast the message (as identified by variable msg) to all other application servers.</td>
</tr>
<tr>
<td>{FW:{M:msg, SID:serverid}}</td>
<td>Request proxy instance to forward the message (as identified by variable msg) to application or service server identified by serverid.</td>
</tr>
<tr>
<td>{MU:{SID:serverid, U:userid}}</td>
<td>Routing request: move client requests identified by userid to the server identified by serverid.</td>
</tr>
<tr>
<td>{CR:custom_route }</td>
<td>Request proxy instance to execute custom routing script (passed on the variable custom_route). Currently disabled.</td>
</tr>
</tbody>
</table>

* Arguments in italic are optional.
Table 8.7: Object Manager Request Methods

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET uri</td>
<td>Similar to HTTP. Requests a representation of the resource identified by an uri.</td>
</tr>
<tr>
<td>POST uri</td>
<td>Sends a request to update a resource identified by uri.</td>
</tr>
<tr>
<td>DELETE uri</td>
<td>Similar to HTTP, deletes a resource.</td>
</tr>
<tr>
<td>PUT uri</td>
<td>Similar to HTTP. Creates a new resource.</td>
</tr>
<tr>
<td>RELOCATE uri_origin uri_destination</td>
<td>Requests that the resource located in uri_origin be relocated to uri_destination. May only relocate resources owned by destination, otherwise returns an error message.</td>
</tr>
<tr>
<td>SUBSCRIBE uri_object uri_subscribe</td>
<td>Requests subscription to a resource identified by uri_object. When the object manager receives a subscription, it becomes responsible to forward every update of the resource to uri_subscribe location until the subscription is cancelled or the object is deleted or transferred. The origin server may also stream updates using the same channel. If the object is deleted or transferred, inform the origin server with a redirect message.</td>
</tr>
<tr>
<td>Operation</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>get</strong></td>
<td>Generates the URI and sends GET request: GET location/objectid. If location is local, get object from persistent media if not already in cache, return cached value.</td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>Generates the URI and sends PUT request: PUT server/objectid. The server identified by location keeps the object in cache. Returns true if operation was successful.</td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>Generates the URI and sends DELETE request: DELETE server/objectid. If the server identified by location had the object in cache, delete it from cache as well. Returns true if operation was successful.</td>
</tr>
<tr>
<td><strong>relocate</strong></td>
<td>Generates the URI and sends RELOCATE request: RELOCATE old_location/object. Object manager on server identified by old_location looks up data in cache. If it is cached, update location of object through persistence manager to new_location and return latest state of object. If not cached, return error for object not found.</td>
</tr>
<tr>
<td><strong>subscribe</strong></td>
<td>Generates the URI and sends SUBSCRIBE request: SUBSCRIBE location/objectid. Server identified by location looks for the object in cache. If not cached, returns error. If cached, returns true for success and initiates a connection to subscribe_location using the chosen protocol, UDP or TCP.</td>
</tr>
</tbody>
</table>
Table 8.9: Jigsaw Piece Data Structure

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>uuid/string</td>
<td>Uniquely identifies a piece. It is also the unique identifier of the object Piece, required for the RCAT data layer.</td>
</tr>
<tr>
<td>bound</td>
<td>boolean</td>
<td>If true, the piece is bound to the board in the correct place. If false, the piece is free to move.</td>
</tr>
<tr>
<td>X,Y</td>
<td>integer</td>
<td>Position of piece in a 2D board, pixel-size based.</td>
</tr>
<tr>
<td>C,R</td>
<td>integer</td>
<td>Column and row where the piece belongs.</td>
</tr>
<tr>
<td>lock</td>
<td>uuid/string</td>
<td>Determines who has the lock on the piece. If the piece is not locked, lock is an empty string.</td>
</tr>
</tbody>
</table>

Table 8.10: Jigsaw Operations: Client

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{pm: [pid, X, Y] }</td>
<td>Piece movement: Informs server that the user is picking up a piece identified by pid and moving it. This event will lock the piece to this user.</td>
</tr>
<tr>
<td>{pd: [pid, X, Y, b] }</td>
<td>Piece drop: Informs server that user is dropping a piece identified by pid in the position X,Y. If b is set to true, the piece has been placed in the correct position and should be bound. Otherwise, release user’s lock to the piece.</td>
</tr>
</tbody>
</table>
### Table 8.11: Jigsaw Configuration Message

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{image_settings: {[imgurl,width,height]} }</code></td>
<td>Image settings. URL for fetching, and width and height of image.</td>
</tr>
<tr>
<td><code>{board: [width, height, minscale, maxscale] }</code></td>
<td>Board (canvas) settings. Width, height, minimum and maximum zoom allowed. Controlling zoom is important for controlling interest management.</td>
</tr>
<tr>
<td><code>{pieces: list_pieces }</code></td>
<td>List of all piece objects. If interest management is on, only the pieces in the client’s frustrum.</td>
</tr>
<tr>
<td><code>{connected_clients: list_connected_clients }</code></td>
<td>List of connected clients with their IDs and usernames. Used for score keeping.</td>
</tr>
<tr>
<td><code>{scores: clientScores }</code></td>
<td>List of scores for all clients. Each score is a username/score pair.</td>
</tr>
</tbody>
</table>
Table 8.12: Jigsaw Mapper

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_piece_movement</td>
<td>Updates a piece identified by pid. Updates the piece’s X and Y position, and sets the lock of the piece to the user identified by userid. Returns true on success if no interest management is in place, or the list of users interested in the piece otherwise.</td>
</tr>
<tr>
<td><strong>Arguments:</strong> pid, X, Y, userid</td>
<td></td>
</tr>
<tr>
<td><strong>Return:</strong> list_users or success</td>
<td></td>
</tr>
<tr>
<td>set_piece_bound</td>
<td>Sets the piece identified by pid to bound. Returns true on success.</td>
</tr>
<tr>
<td><strong>Arguments:</strong> pid</td>
<td></td>
</tr>
<tr>
<td><strong>Return:</strong> success</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.13: Virtual Environment Operations: Client

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{am: [X, Y] }</td>
<td>Informs server that the avatar belonging to the client has moved to position X,Y in cartesian coordinate system.</td>
</tr>
<tr>
<td>{om: [objid, X, Y] }</td>
<td>Object movement: Informs server that the avatar belonging to the client wishes to move the object identified by objid to position X,Y in cartesian coordinate system.</td>
</tr>
</tbody>
</table>
### Table 8.14: Virtual Environment Operations: Mapper

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_avatar_new_position</td>
<td>A request for moving the avatar of the user identified by userid to the new</td>
</tr>
<tr>
<td></td>
<td>coordinates, X and Y.</td>
</tr>
<tr>
<td>Arguments: userid,X,Y</td>
<td></td>
</tr>
<tr>
<td>Returns: success</td>
<td></td>
</tr>
<tr>
<td>set_object_new_position</td>
<td>A request for moving an object identified by objectid to the new coordinates,</td>
</tr>
<tr>
<td></td>
<td>X and Y.</td>
</tr>
<tr>
<td>Arguments: objectid,X,Y</td>
<td></td>
</tr>
<tr>
<td>Returns: success</td>
<td></td>
</tr>
</tbody>
</table>

### Table 8.15: Feasibility Evaluation: Message Types

<table>
<thead>
<tr>
<th>Message Types</th>
<th>Entry</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Connection</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Disconnect</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Position</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>AllUsers</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>255</td>
</tr>
</tbody>
</table>
### Table 8.16: Feasibility Evaluation: Messages

<table>
<thead>
<tr>
<th>Message Type / Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any / Error / {Type: 255, Description: desc}</td>
<td>An error has occurred. Return error description in variable desc.</td>
</tr>
<tr>
<td>Proxy-Server / Connect / {Type: 0, User: userid}</td>
<td>Informs application server that user identified by userid has connected.</td>
</tr>
<tr>
<td>Proxy-Server / Disconnect / {Type: 1, User: userid}</td>
<td>Informs application server that user identified by userid has disconnected.</td>
</tr>
<tr>
<td>Proxy-Server / Position / {Type: 2, User: userid, Position: pos, TimeStamp: ts}</td>
<td>Message setting the position of user identified by userid to the position in pos. The variable pos is an X,Y pair. Timestamp is used to measure round-trip latency and to improve performance, discarding old messages, if they reach the proxy after a newer message.</td>
</tr>
<tr>
<td>Server-Proxy / AllUsers / {Type: 3, User: userid, Position: pos, TimeStamp: ts}</td>
<td>Application server reply to setting a new position. Creates a broadcast type of message (AllUsers) and send it to the proxies, so they can broadcast it to all users. Timestamp ts is the same, carried from Proxy-Server Position message.</td>
</tr>
</tbody>
</table>
8.2 PCC and Spacetime Datamodels

```python
@pcc_set
class Car(object):
    @primarykey(str)
def ID(self):
        return self._ID
    @dimension(Vector3)
def Position(self):
        return self._Position
    @dimension(Vector3)
def Velocity(self):
        return self._Velocity

@subset(Car)
class ActiveCar(Car):
    @staticmethod
def query(cars):
        return [c for c in cars if ActiveCar._predicate_(c)]
    @staticmethod
def predicate_(c):
        return c.Velocity != Vector3(0, 0, 0)
def move(self):
    # End of ride
    if (self.Position.X >= self.FINAL_POSITION
        or self.Position.Y >= self.FINAL_POSITION):
        self.stop()

def stop(self):
    self.Position = Vector3(0, 0, 0)
    self.Velocity = Vector3(0, 0, 0)

@subset(Car)
class InactiveCar(Car.Class()):
    @staticmethod
def query_(cars):
        return [c for c in cars if InactiveCar._predicate_(c)]
    @staticmethod
def predicate_(c):
        return c.Position == Vector3(0, 0, 0)
def start(self):
    self.Velocity = Vector3(self.SPEED, 0, 0)
```

Listing 8.1: Car and related types for example traffic simulation
class Pedestrian(object):
    INITIAL_POSITION = 500
    @primarykey(str)
    def ID(self):
        return self._ID
    @dimension(int)
    def X(self):
        return self._X
    @dimension(int)
    def Y(self):
        return self._Y

    def move(self):
        self.X -= self.SPEED
        if self.X <= 0:
            self.stop()

    def stop(self):
        self.X = self.INITIAL_POSITION
        self.Y = 0

    def setPosition(self, x):
        self.X = x;

@subset(Pedestrian)
class Walker(Pedestrian):
    @staticmethod
    def query(peds):
        return [p for p in peds if Walker.predicate(p)]

    @staticmethod
    def predicate(p):
        return p.X != Pedestrian.INITIAL_POSITION

@subset(Pedestrian)
class StoppedPedestrian(Pedestrian):
    @staticmethod
    def query(peds):
        return [p for p in peds if StoppedPedestrian.predicate(p)]

    @staticmethod
    def predicate(p):
        return p.X == Pedestrian.INITIAL_POSITION

@join(Walker, ActiveCar)
class CarAndPedestrianNearby(object):
    @dimension(Car)
    def car(self):
        return self._car

    @dimension(Pedestrian)
    def pedestrian(self):
        return self._ped

    @staticmethod
def query(peds, cars):
    return [(p, c) for p in peds for c in cars if CarAndPedestrianNearby.predicate(p, c)]

@staticmethod
def predicate(p, c):

def move(self):
    self.pedestrian.Y += 50;

Listing 8.2: Pedestrian and related type definitions for example traffic simulation
class BaseSet(object):
    @primarykey(str)
    def ID(self):
        return self._ID
    @ID.setter
    def ID(self, value):
        self._ID = value

    @dimension(str)
    def Name(self):
        return self._Name
    @Name.setter
    def Name(self, value):
        self._Name = value

    @dimension(int)
    def Number(self):
        return self._Number
    @Number.setter
    def Number(self, value):
        self._Number = value

    @dimension(list)
    def List(self):
        return self._List
    @List.setter
    def List(self, value):
        self._List = value

    @dimension(dict)
    def Dictionary(self):
        return self._Dictionary
    @Dictionary.setter
    def Dictionary(self, value):
        self._Dictionary = value

    @dimension(str)
    def Property1(self):
        return self._Property1
    @Property1.setter
    def Property1(self, value):
        self._Property1 = value

    @dimension(str)
    def Property2(self):
        return self._Property2
    @Property2.setter
    def Property2(self, value):
        self._Property2 = value
```
@dimension(str)
def Property3(self):
    return self._Property3
@property3.setter
def Property3(self, value):
    self._Property3 = value

@dimension(str)
def Property4(self):
    return self._Property4
@property4.setter
def Property4(self, value):
    self._Property4 = value

@dimension(str)
def Property5(self):
    return self._Property5
@property5.setter
def Property5(self, value):
    self._Property5 = value

@dimension(str)
def Property6(self):
    return self._Property6
@property6.setter
def Property6(self, value):
    self._Property6 = value

def __init__(self, num):
    self.ID = None
    self.Name = ""
    self.Number = num
    self.List = [i for i in xrange(20)]
    #self.Set = set([i for i in xrange(20)])
    self.Dictionary = {str(k) : k for k in xrange(20)}
    self.Property1 = "Property_1"
    self.Property2 = "Property_2"
    self.Property3 = "Property_3"
    self.Property4 = "Property_4"
    self.Property5 = "Property_5"
```

Listing 8.3: Definition of data type BaseSet used in benchmarks.
@subset (BaseSet)

class SubsetHalf(BaseSet.Class):
    @staticmethod
    def __query__(base):
        return [o for o in base if SubsetHalf.__predicate__(o)]

    @staticmethod
    def __predicate__(o):
        return o.Number % 2 == 0

Listing 8.4: Definition of data type SubsetHalf used in benchmarks. Returns half of all BaseSet objects.

@projection (BaseSet, BaseSet.ID, BaseSet.Name)
class BaseSetProjection(object):
    @property
    def DecoratedName(self):
        return "∗∗" + self.Name + "∗∗"

Listing 8.5: Definition of data type BaseSetProjection used in benchmarks. Projects ID and Name dimensions only.

@join (BaseSet, BaseSet)
class JoinHalf(object):
    @primarykey (str)
    def ID(self):
        return self._ID

    @ID.setter
    def ID(self, value):
        self._ID = value

    @dimension (BaseSet)
    def b1(self):
        return self._b1

    @b1.setter
    def b1(self, value):
        self._b1 = value

    @dimension (BaseSet)
    def b2(self):
        return self._b2

    @b2.setter
    def b2(self, value):
        self._b2 = value

    def __init__(self, b1, b2):
        self.b1 = b1
        self.b2 = b2

241
```python
@staticmethod
def query(b1s, b2s):
    return [(b1, b2) for b1 in b1s for b2 in b2s if JoinHalf.predicate(b1, b2)]

@staticmethod
def predicate(b1, b2):
    if b1 == b2:
        return b1.Number % 2 == 0
```

Listing 8.6: Definition of data type JoinHalf used in benchmarks. Returns half of the total number of BaseSet objects as pairs.
8.3 CADIS Benchmark Simulation Scenarios

```python
# Benchmark Producer declaration
BP_PRODUCER = [BaseSet]

# Benchmark Consumer declaration
BC_TRACKER = [BaseSet]

builtin_dict = globals()['__builtins__']
builtin_dict['BP_PRODUCER'] = BP_PRODUCER
builtin_dict['BT_TRACKER'] = BT_TRACKER

# Initialize method for producer
def initialize_producer(sim):
    frame = sim.frame
    for i in xrange(sim.instances):
        frame.add(BaseSet(i))

# Initialize method for consumer
def initialize_consumer(sim):
    pass

# Update method for producer
def update_producer(sim):
    pass

# Update method for consumer
def update_consumer(sim):
    pass
```

Listing 8.7: Source-code of PCC set benchmark scenario.
# Benchmark Producer declaration

BP_PRODUCER = [BaseSet]

# Benchmark Consumer declaration

BC_TRACKER = [SubsetHalf]

builtin_dict = globals()["__builtins__"]
builtin_dict["BP_PRODUCER"] = BP_PRODUCER
builtin_dict["BT_TRACKER"] = BT_TRACKER

# Initialize method for producer

def initialize_producer(sim):
    frame = sim.frame
    for i in xrange(sim.instances):
        frame.add(BaseSet(i))

# Initialize method for consumer

def initialize_consumer(sim):
    pass

# Update method for producer

def update_producer(sim):
    pass

# Update method for consumer

def update_consumer(sim):
    pass

Listing 8.8: Source-code of PCC subset benchmark scenario.
# Benchmark Producer declaration
BP_PRODUCER = [BaseSet]

# Benchmark Consumer declaration
BC_TRACKER = [BaseSetProjection]

builtin_dict = globals()['__builtins__']
builtin_dict['BP_PRODUCER'] = BP_PRODUCER
builtin_dict['BT_TRACKER'] = BT_TRACKER

# Initialize method for producer
def initialize_producer(sim):
    frame = sim.frame
    for i in xrange(sim.instances):
        frame.add(BaseSet(i))

# Initialize method for consumer
def initialize_consumer(sim):
    pass

# Update method for producer
def update_producer(sim):
    pass

# Update method for consumer
def update_consumer(sim):
    pass

Listing 8.9: Source-code of PCC projection benchmark scenario.
# Benchmark Producer declaration
BP_PRODUCER = [BaseSet]

# Benchmark Consumer declaration
BC_TRACKER = [JoinHalf]

builtin_dict = globals()['_builtins_']
builtin_dict['BP_PRODUCER'] = BP_PRODUCER
builtin_dict['BT_TRACKER'] = BT_TRACKER

# Initialize method for producer
def initialize_producer(sim):
    frame = sim.frame
    for i in xrange(sim.instances):
        frame.add(BaseSet(i))

# Initialize method for consumer
def initialize_consumer(sim):
    pass

# Update method for producer
def update_producer(sim):
    pass

# Update method for consumer
def update_consumer(sim):
    pass

Listing 8.10: Source-code of PCC join benchmark scenario.
# Benchmark Producer declaration
BP_PRODUCER = [BaseSet]
BP_SETTER = [BaseSet]

# Benchmark Consumer declaration
BC_GETTER = [BaseSet]

builtin_dict = globals()['_builtins_']
builtin_dict['BP_PRODUCER'] = BP_PRODUCER
builtin_dict['BT_GETTER'] = BT_GETTER

# Initialize method for producer
def initialize_producer(sim):
    frame = sim.frame
    for i in xrange(sim.instances):
        frame.add(BaseSet(i))

# Initialize method for consumer
def initialize_consumer(sim):
    pass

# Update method for producer
def update_producer(sim):
    for o in sim.frame.get(BaseSet):
        o.Number += 1

# Update method for consumer
def update_consumer(sim):
    pass

Listing 8.11: Source-code of PCC update benchmark scenario.
# Benchmark Producer declaration

BP_PRODUCER = [BaseSet]

# Benchmark Consumer declaration

BC_TRACKER = [BaseSet]

builtิน_dict = globals()['_builtins_']
builtิน_dict['BP_PRODUCER'] = BP_PRODUCER
builtิน_dict['BC_TRACKER'] = BC_TRACKER

# Initialize method for producer

def initialize_producer(sim):
    pass

# Initialize method for consumer

def initialize_consumer(sim):
    pass

# Update method for producer

def update_producer(sim):
    for i in xrange(sim.instances):
        sim.frame.add(BaseSet(1))

# Update method for consumer

def update_consumer(sim):
    pass

Listing 8.12: Source-code of PCC creation benchmark scenario.
8.4 CADIS Tables and Graphs

Tables show average values for processing time in push, pull, and update operations, and average data sent and received. Units are \textit{milliseconds} for push, pull, update, and total, and \textit{bytes} for sent and received.

Table 8.17: Results for producer of test PCC set

<table>
<thead>
<tr>
<th>mode</th>
<th>push</th>
<th>pull</th>
<th>update</th>
<th>total</th>
<th>sent</th>
<th>received</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>519.64</td>
<td>6.14</td>
<td>0.04</td>
<td>525.92</td>
<td>544,940.00</td>
<td>3.00</td>
</tr>
<tr>
<td>DIFF_PUSH</td>
<td>0.07</td>
<td>6.41</td>
<td>0.04</td>
<td>6.61</td>
<td>0.00</td>
<td>3.00</td>
</tr>
<tr>
<td>DIFF_PUSHPULL</td>
<td>0.06</td>
<td>6.22</td>
<td>0.04</td>
<td>6.39</td>
<td>0.00</td>
<td>42.00</td>
</tr>
<tr>
<td>DATAFRAME</td>
<td>5.78</td>
<td>5.21</td>
<td>0.04</td>
<td>11.11</td>
<td>47.00</td>
<td>10.00</td>
</tr>
</tbody>
</table>

Table 8.18: Results for consumer of test PCC set

<table>
<thead>
<tr>
<th>mode</th>
<th>push</th>
<th>pull</th>
<th>update</th>
<th>total</th>
<th>sent</th>
<th>received</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>4.66</td>
<td>157.35</td>
<td>0.06</td>
<td>162.15</td>
<td>39.00</td>
<td>543,805.44</td>
</tr>
<tr>
<td>DIFF_PUSH</td>
<td>0.03</td>
<td>161.93</td>
<td>0.06</td>
<td>162.10</td>
<td>0.00</td>
<td>542,706.87</td>
</tr>
<tr>
<td>DIFF_PUSHPULL</td>
<td>0.06</td>
<td>6.29</td>
<td>0.08</td>
<td>6.50</td>
<td>0.00</td>
<td>42.00</td>
</tr>
<tr>
<td>DATAFRAME</td>
<td>5.81</td>
<td>5.24</td>
<td>0.09</td>
<td>11.23</td>
<td>47.00</td>
<td>10.00</td>
</tr>
</tbody>
</table>

Table 8.19: Results for producer of test subset

<table>
<thead>
<tr>
<th>mode</th>
<th>push</th>
<th>pull</th>
<th>update</th>
<th>total</th>
<th>sent</th>
<th>received</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>515.45</td>
<td>6.10</td>
<td>0.04</td>
<td>521.68</td>
<td>544,940.00</td>
<td>3.00</td>
</tr>
<tr>
<td>DIFF_PUSH</td>
<td>0.06</td>
<td>6.11</td>
<td>0.04</td>
<td>6.30</td>
<td>0.00</td>
<td>3.00</td>
</tr>
<tr>
<td>DIFF_PUSHPULL</td>
<td>0.06</td>
<td>6.30</td>
<td>0.04</td>
<td>6.48</td>
<td>0.00</td>
<td>42.00</td>
</tr>
<tr>
<td>DATAFRAME</td>
<td>5.80</td>
<td>5.24</td>
<td>0.04</td>
<td>11.15</td>
<td>47.00</td>
<td>10.00</td>
</tr>
</tbody>
</table>

Table 8.20: Results for consumer of test subset

<table>
<thead>
<tr>
<th>mode</th>
<th>push</th>
<th>pull</th>
<th>update</th>
<th>total</th>
<th>sent</th>
<th>received</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>5.34</td>
<td>361.45</td>
<td>0.05</td>
<td>366.92</td>
<td>39.00</td>
<td>271,912.72</td>
</tr>
<tr>
<td>DIFF_PUSH</td>
<td>0.27</td>
<td>367.36</td>
<td>0.05</td>
<td>367.74</td>
<td>0.00</td>
<td>271,912.72</td>
</tr>
<tr>
<td>DIFF_PUSHPULL</td>
<td>0.04</td>
<td>358.23</td>
<td>0.05</td>
<td>358.36</td>
<td>0.00</td>
<td>271,967.72</td>
</tr>
<tr>
<td>DATAFRAME</td>
<td>5.80</td>
<td>5.27</td>
<td>0.08</td>
<td>11.24</td>
<td>47.00</td>
<td>10.00</td>
</tr>
</tbody>
</table>
Table 8.21: Results for producer of test projection

<table>
<thead>
<tr>
<th>mode</th>
<th>push</th>
<th>pull</th>
<th>update</th>
<th>total</th>
<th>sent</th>
<th>received</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>518.60</td>
<td>6.11</td>
<td>0.04</td>
<td>524.85</td>
<td>544,940.00</td>
<td>3.00</td>
</tr>
<tr>
<td>DIFF_PUSH</td>
<td>0.06</td>
<td>5.94</td>
<td>0.04</td>
<td>6.13</td>
<td>0.00</td>
<td>3.00</td>
</tr>
<tr>
<td>DIFF_PUSHPULL</td>
<td>0.06</td>
<td>6.24</td>
<td>0.04</td>
<td>6.41</td>
<td>0.00</td>
<td>42.00</td>
</tr>
<tr>
<td>DATAFRAME</td>
<td>5.81</td>
<td>5.24</td>
<td>0.04</td>
<td>11.17</td>
<td>47.00</td>
<td>10.00</td>
</tr>
</tbody>
</table>

Table 8.22: Results for consumer of test projection

<table>
<thead>
<tr>
<th>mode</th>
<th>push</th>
<th>pull</th>
<th>update</th>
<th>total</th>
<th>sent</th>
<th>received</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>4.38</td>
<td>282.12</td>
<td>0.05</td>
<td>286.63</td>
<td>39.00</td>
<td>99,822.39</td>
</tr>
<tr>
<td>DIFF_PUSH</td>
<td>0.12</td>
<td>286.42</td>
<td>0.05</td>
<td>286.65</td>
<td>0.00</td>
<td>99,822.39</td>
</tr>
<tr>
<td>DIFF_PUSHPULL</td>
<td>0.04</td>
<td>286.59</td>
<td>0.04</td>
<td>286.73</td>
<td>0.00</td>
<td>99,884.39</td>
</tr>
<tr>
<td>DATAFRAME</td>
<td>5.81</td>
<td>5.27</td>
<td>0.09</td>
<td>11.25</td>
<td>47.00</td>
<td>10.00</td>
</tr>
</tbody>
</table>

Table 8.23: Results for producer of test join

<table>
<thead>
<tr>
<th>mode</th>
<th>push</th>
<th>pull</th>
<th>update</th>
<th>total</th>
<th>sent</th>
<th>received</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>514.36</td>
<td>6.02</td>
<td>0.04</td>
<td>520.52</td>
<td>544,940.00</td>
<td>3.00</td>
</tr>
<tr>
<td>DIFF_PUSH</td>
<td>0.06</td>
<td>5.69</td>
<td>0.03</td>
<td>5.87</td>
<td>0.00</td>
<td>3.00</td>
</tr>
<tr>
<td>DIFF_PUSHPULL</td>
<td>0.06</td>
<td>5.80</td>
<td>0.03</td>
<td>5.96</td>
<td>0.00</td>
<td>42.00</td>
</tr>
<tr>
<td>DATAFRAME</td>
<td>5.77</td>
<td>5.47</td>
<td>0.04</td>
<td>11.36</td>
<td>47.00</td>
<td>10.00</td>
</tr>
</tbody>
</table>

Table 8.24: Results for consumer of test join

<table>
<thead>
<tr>
<th>mode</th>
<th>push</th>
<th>pull</th>
<th>update</th>
<th>total</th>
<th>sent</th>
<th>received</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>4.76</td>
<td>632.83</td>
<td>0.05</td>
<td>637.74</td>
<td>39.00</td>
<td>553,786.27</td>
</tr>
<tr>
<td>DIFF_PUSH</td>
<td>0.58</td>
<td>633.25</td>
<td>0.05</td>
<td>633.96</td>
<td>0.00</td>
<td>553,786.27</td>
</tr>
<tr>
<td>DIFF_PUSHPULL</td>
<td>0.04</td>
<td>620.95</td>
<td>0.05</td>
<td>621.10</td>
<td>0.00</td>
<td>553,839.27</td>
</tr>
<tr>
<td>DATAFRAME</td>
<td>5.04</td>
<td>285.85</td>
<td>0.08</td>
<td>291.03</td>
<td>47.00</td>
<td>284,959.34</td>
</tr>
</tbody>
</table>
Table 8.25: Results for producer of test update

<table>
<thead>
<tr>
<th>mode</th>
<th>push</th>
<th>pull</th>
<th>update</th>
<th>total</th>
<th>sent</th>
<th>received</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>506.32</td>
<td>6.15</td>
<td>16.31</td>
<td>528.88</td>
<td>545,289.85</td>
<td>3.00</td>
</tr>
<tr>
<td>DIFF_PUSH</td>
<td>16.93</td>
<td>6.33</td>
<td>17.09</td>
<td>40.44</td>
<td>57,325.71</td>
<td>3.00</td>
</tr>
<tr>
<td>DIFF_PUSHPULL</td>
<td>13.48</td>
<td>5.95</td>
<td>16.69</td>
<td>36.19</td>
<td>57,326.45</td>
<td>42.00</td>
</tr>
<tr>
<td>DATAFRAME</td>
<td>44.79</td>
<td>5.42</td>
<td>32.58</td>
<td>82.87</td>
<td>114,297.82</td>
<td>10.00</td>
</tr>
</tbody>
</table>

Table 8.26: Results for consumer of test update

<table>
<thead>
<tr>
<th>mode</th>
<th>push</th>
<th>pull</th>
<th>update</th>
<th>total</th>
<th>sent</th>
<th>received</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>4.73</td>
<td>157.34</td>
<td>0.06</td>
<td>162.23</td>
<td>39.00</td>
<td>544,155.52</td>
</tr>
<tr>
<td>DIFF_PUSH</td>
<td>0.04</td>
<td>159.87</td>
<td>0.06</td>
<td>160.04</td>
<td>0.00</td>
<td>543,055.39</td>
</tr>
<tr>
<td>DIFF_PUSHPULL</td>
<td>0.09</td>
<td>37.77</td>
<td>0.05</td>
<td>37.96</td>
<td>0.00</td>
<td>57,142.67</td>
</tr>
<tr>
<td>DATAFRAME</td>
<td>4.75</td>
<td>48.27</td>
<td>0.06</td>
<td>53.14</td>
<td>47.00</td>
<td>113,803.59</td>
</tr>
</tbody>
</table>

Table 8.27: Results for producer of test creation

<table>
<thead>
<tr>
<th>mode</th>
<th>push</th>
<th>pull</th>
<th>update</th>
<th>total</th>
<th>sent</th>
<th>received</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>262.96</td>
<td>7.08</td>
<td>2.28</td>
<td>272.42</td>
<td>269,788.27</td>
<td>3.00</td>
</tr>
<tr>
<td>DIFF_PUSH</td>
<td>6.27</td>
<td>6.79</td>
<td>2.70</td>
<td>15.86</td>
<td>1,171.00</td>
<td>3.00</td>
</tr>
<tr>
<td>DIFF_PUSHPULL</td>
<td>6.10</td>
<td>8.29</td>
<td>2.63</td>
<td>17.09</td>
<td>1,171.00</td>
<td>42.00</td>
</tr>
<tr>
<td>DATAFRAME</td>
<td>11.48</td>
<td>6.84</td>
<td>3.17</td>
<td>21.58</td>
<td>4,792.00</td>
<td>10.00</td>
</tr>
</tbody>
</table>

Table 8.28: Results for consumer of test creation

<table>
<thead>
<tr>
<th>mode</th>
<th>push</th>
<th>pull</th>
<th>update</th>
<th>total</th>
<th>sent</th>
<th>received</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>4.44</td>
<td>86.64</td>
<td>0.05</td>
<td>91.21</td>
<td>39.00</td>
<td>269,902.52</td>
</tr>
<tr>
<td>DIFF_PUSH</td>
<td>0.03</td>
<td>86.58</td>
<td>0.05</td>
<td>86.72</td>
<td>0.00</td>
<td>268,157.28</td>
</tr>
<tr>
<td>DIFF_PUSHPULL</td>
<td>0.06</td>
<td>9.38</td>
<td>0.07</td>
<td>9.58</td>
<td>0.00</td>
<td>1,136.79</td>
</tr>
<tr>
<td>DATAFRAME</td>
<td>8.90</td>
<td>9.33</td>
<td>0.29</td>
<td>18.61</td>
<td>47.00</td>
<td>4,735.87</td>
</tr>
</tbody>
</table>