Proceedings of the 2016 Winter Simulation Conference
T. M. K. Roeder, P. I. Frazier, R. Szechtman, E. Zhou, T. Huschka, and S. E. Chick, eds.

CADIS: ASPECT-ORIENTED ARCHITECTURE FOR COLLABORATIVE MODELING AND SIMULATION

Arthur Valadares,
Cristina V. Lopes,
Rohan Achar

Donald Bren School of ICS
University of California, Irvine
Irvine, CA 92697-3425, USA

Mic Bowman

Intel Labs, Intel Corporation
2111 NE 25th Ave
Hillsboro, OR, USA

ABSTRACT

The development of large and complex simulated models often requires teams to collaborate. One approach is to break a large model into independently developed partial models that, when combined, capture the overall behavior. However, maintaining consistent world state across independently developed simulations is a challenge.

In this paper, we introduce the Collaborative Aspect-Oriented Distributed Interactive Simulation (CADIS) architecture and development platform. CADIS embodies a new paradigm for integrating independently developed time-discrete partial models and simulations, focusing on transparently maintaining synchronized shared state. Data is pulled and instantiated in the beginning of each time step, and pushed at the end of each time step. An urban simulation is used to demonstrate CADIS capabilities and performance. We show how simple optimizations can bring the performance of the framework to acceptable levels, making CADIS a viable modeling and simulation methodology supporting separation of concerns.

1 INTRODUCTION

Simulations are powerful tools for understanding and predicting physical systems for a plethora of disciplines – engineering, sociology, and logistics to name a few. Simulated models are representations of real-world physical systems, where often there is not a clear separation of domains of knowledge. A complex, high fidelity simulation for traffic engineering, for instance, is the combination of social factors, logistics, natural conditions (e.g. weather), engineering, and others. Achieving a high fidelity model of a physical system requires deep knowledge of how each part of the system operates and interacts with the other parts. The required expertise for complex simulations is both deep and broad.

An approach to handle the large diversity of domain knowledge in simulations is to abstract away expertise by using simpler models. A traffic engineering group, for instance, might use simpler models for simulating social factors, whereas a sociology group may simplify the models of traffic engineering to favor studying how a person reacts to traffic. Unfortunately, the resulting model of either group would not be a high fidelity representation of the real-world, as simpler models could cause the findings of a study to be imprecise or even incorrect. This scenario raises a concern for how can groups of simulation practitioners who are separated by their domain knowledge – and sometimes also geographically – build a simulation collaboratively. We refer to this practice as **collaborative simulations**: simulations that are built with collaborators of different expertise or that integrate existing simulations from different domains.

This paper explores a novel approach to the collaborative development of simulations, by introducing a distributed simulation framework called CADIS (Collaborative Architecture for Distributed Interactive

Simulations). CADIS uses an object-oriented approach for managing shared state and data structures, and is designed to easily integrate independently developed simulations sharing the same entities. Three major design decisions makes CADIS unique: 1) shared state synchronization is declarative, in the form of type annotations; 2) shared state is imported into the partial simulations as instances of locally-defined classes using predicates on data; and 3) a central data store transparently manages shared state, acting as a database.

2 RELATED WORK

Distributed simulation frameworks provide standards that facilitate the development of simulations running in distinct physical processes interconnected by a network. However, distributed simulations are known to be difficult to use (Boer, Bruin, and Verbraeck 2006), and almost exclusively adopted in the military domain. The High Level Architecture (HLA) (IEEE Standards Association 2000) is possibly the most current and widely used standard for distributed simulations. Designed and developed as an initiative from the Department of Defense (DoD), HLA has been a successful standard in integrating multiple simulations from independent military contractors and in some academical ventures. 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* (Gianni et al. 2011), 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. (Tang et al. 2010) proposes a layered architecture with a Resource Management Federation (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 (Topçu and Oğuztüzün 2013) 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 Section 3.2), where simulation events are represented as object updates.

Other 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., combining a formal modeling framework called Discrete Event System Specification (DEVS) (Zeigler et al. 1996) with a distributed object standards called Common Object Broker Architecture (CORBA) (Orfali and Harkey 1998). The resulting framework is called DEVS/CORBA (Zeigler, Kim, and Buckley 1999).

Another proposed approach to distributed simulation framework focused on supply chain management is GRIDS (Generic Runtime Infrastructure for Distributed Simulation). GRIDS (Taylor, Saville, and Sudra 1999) 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.

3 CADIS

CADIS is designed to achieve one main objective: easy integration of independently developed simulations. It is not designed to be a replacement of existing solutions such as HLA, but rather rethink distributed simulation design in order to decrease the technical complexity for integrating simulations. The goal is to show that by designing simulations with a particular set of design principles, integration becomes simple and distribution transparency can be easily achieved.

The architecture of CADIS is similar to HLA, and can be seen in Figure 1. CADIS is composed of 3 main components: the **Simulation**, a library called **Frame**, and the **Store**. The Simulation component is the independently developed simulation, and assumed to be initially designed as non-distributed. For easy integration to CADIS, the simulation should follow (or be adapted to) the object-oriented development style of CADIS by encapsulating all simulation entities as abstract data types. Types should then be annotated so CADIS knows which types and attributes need to be synchronized with other simulations. In the following sections, the guiding design principles of CADIS are explained in detail.

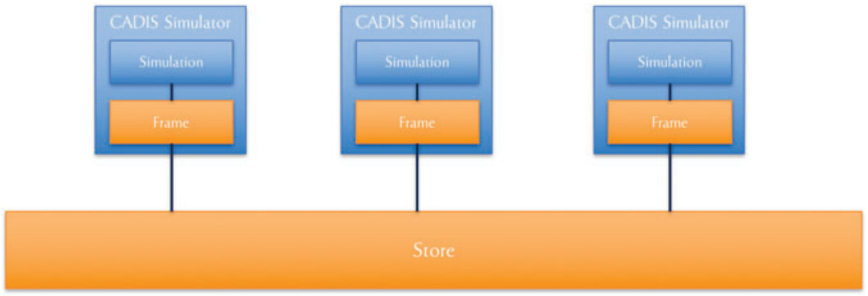


Figure 1: An architectural view of CADIS.

3.1 Layered Architecture and Interest Management

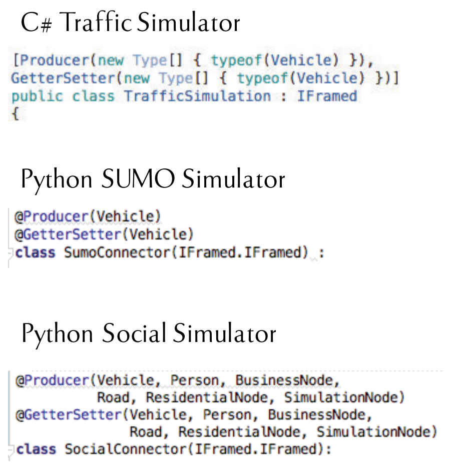
The architecture of CADIS is layered, meaning software components on each layer can only communicate with layers below or above it. For CADIS, this means a **Simulation** can only communicate with the **Frame** library and **Frame** communicates with both the **Store** and the **Simulation**.

A Simulation must implement an API from Frame that enforces 3 methods: *initialize*, *update*, and *shutdown*. *Initialize* is called by CADIS so simulations have a chance to prepare before the first clock event is distributed. It is also used so that simulations can add the initial load of shared objects to CADIS, so that CADIS may distribute these objects before the actual simulation begins. *Update* is the method called by CADIS per clock event, and *shutdown* is called before the distributed simulation is shut down.

Simulations must annotate the shared types that will be produced and consumed, as in Figure 2a. Simulations can be annotated as *Producer*, *Tracker*, *Getter*, or *GetterSetter* of a list of types. A *Producer(types)* annotation means the simulation will produce objects of the referenced types for CADIS. A *Tracker(types)* annotation indicates that the simulation wants to be informed of new and deleted objects of the referenced types. The *Getter(types)* annotation indicates the simulation's interest in receiving updates to property modifications for those object types (including new and deleted), and a *GetterSetter(types)* indicates the simulation will be pushing updates to the refeferenced types. The last 3 annotations are hierarchical: a *GetterSetter* implies a *Getter* which, in turn, implies a *Tracker*.

Frame is a native language library that the Simulation imports to seamlessly maintain synchronized shared state. Frame performs 3 operations at every clock event: *update*, *push*, and *pull*. *Update* calls the Simulation's update function: a function passed to Frame to be executed at every clock event. During the update phase, Frame collects all modifications done to setter objects – objects of types included in *GetterSetter*. In *push*, Frame sends all modifications collected in the *update* phase to the Store. Finally, *pull* retrieves from the Store the most current version of observed objects – objects of types included in *Tracker*, *Getter*, or *GetterSetter*. The Frame library is relatively simple to implement, having already been implemented in both C# and Python, currently available open-source at http://github.com/arthur00/mobdat.

During the update phase, the Simulation has access to 4 methods in the Frame API: *add*, *delete*, *get*, and *attach*. *Add* takes a new object instance to be shared across simulations. *Get* retrieves shared objects of a given type from the cache of objects maintained by Frame. *Delete* allows objects to be removed from the distributed simulation. *Attach* will attach the simulation to the Frame library, allowing Frame to interpret type annotations and to call the simulation's update method every tick. All 4 operations are performed locally: no communication with the Store is done during the update phase. Objects are fetched from the cache of the last *pull*, and modifications, additions, and deletions are sent in the *push* phase only.


C# Traffic Simulator

```
[Producer(new Type[] { typeof(Vehicle) }),
GetterSetter(new Type[] { typeof(Vehicle) })]
public class TrafficSimulation : IFramed
{
```

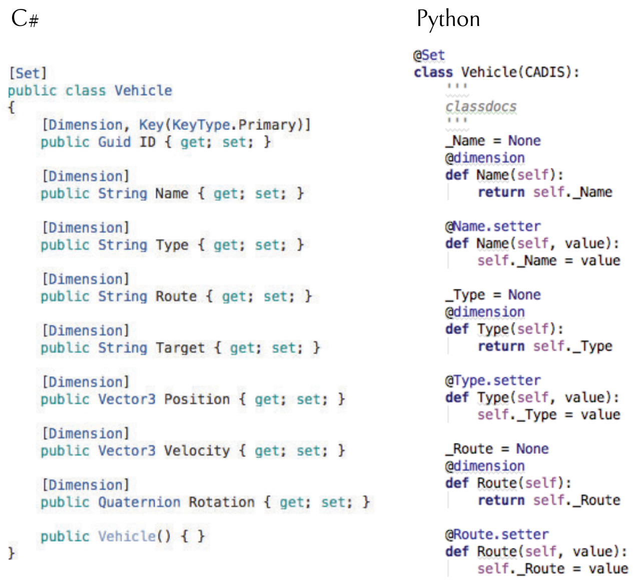
Python SUMO Simulator

```
@Producer(Vehicle)
@GetterSetter(Vehicle)
class SumoConnector(IFramed.IFramed) :
```

Python Social Simulator

```
@Producer(Vehicle, Person, BusinessNode,
          Road, ResidentialNode, SimulationNode)
@GetterSetter(Vehicle, Person, BusinessNode,
              Road, ResidentialNode, SimulationNode)
class SocialConnector(IFramed.IFramed):
```

(a) Simulators annotated with type interests.


C#

```
[Set]
public class Vehicle
{
    [Dimension, Key(KeyType.Primary)]
    public Guid ID { get; set; }

    [Dimension]
    public String Name { get; set; }

    [Dimension]
    public String Type { get; set; }

    [Dimension]
    public String Route { get; set; }

    [Dimension]
    public String Target { get; set; }

    [Dimension]
    public Vector3 Position { get; set; }

    [Dimension]
    public Vector3 Velocity { get; set; }

    [Dimension]
    public Quaternion Rotation { get; set; }

    public Vehicle() { }
}
```

Python

```
@Set
class Vehicle(CADIS):
    '''
    classdocs
    '''
    _Name = None
    @dimension
    def Name(self):
        return self._Name

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

    _Type = None
    @dimension
    def Type(self):
        return self._Type

    @Type.setter
    def Type(self, value):
        self._Type = value

    _Route = None
    @dimension
    def Route(self):
        return self._Route

    @Route.setter
    def Route(self, value):
        self._Route = value
```

(b) A Vehicle object type, annotated for CADIS in C# and Python.

Figure 2: Examples of annotation of types and simulations in CADIS.

The last layer is the **Store**, whose responsibility is to act similarly to a centralized database for shared state. The Store fulfills 2 major requests from Frame: *push* and *pull*. A *push* request contains updates to objects from the simulators, which are merged to the Store's copy of the data. *Pull* requests the latest updated objects observed by a simulation. If a pull request is made for a dependent type (see Section 3.3), the query is executed and the results are sent back to the simulation.

The Store can be implemented using many different backend approaches. Currently, there are two implemented versions of the Store: the **SimpleStore** and the **RemoteStore**. The SimpleStore is an entirely memory based implementation, using dictionary data structures. Using the SimpleStore directly currently restricts the simulations to being non-distributed and non-parallel. The RemoteStore separates the Store logic in two parts: a client and a server. On the client part, the RemoteStore is responsible for converting all Store API requests into JSON and send it over HTTP to a web server. On the server side, a web server imports a real Store – like the SimpleStore – decoding the requests and calling the respective Store methods.

3.2 Type Annotations

A *Set* is the fundamental shared type in CADIS. Types shared with other simulations must be annotated as a *Set* or one of the following *Set* relationships: *Projection*, *Is-a*, *Subset*, or *Join*. A *Projection* is a representation of an existing *Set* containing only a predefined sub-collection of properties. A *Is-a* is also a representation of an existing *Set*, but allows extending a *Set* with new properties. A *Subset* is a collection of objects of one *Set* type that conform to a given query. A *Join* is a *Set* constructed with a query using two *Sets*. *Subsets* and *Joins* are dependent collection classes: a data type whose extent depends on run-time values.

Shared attributes of CADIS types are marked as *Dimension*. Dimensions apply only to *properties*, a common paradigm in programming languages that hide direct access to a variable through *getter* and *setter* methods. Other annotations to properties include *primarykey* and *foreignkey*. A property annotated as *primarykey* will serve as the identifier of object instances across CADIS. Since CADIS allows different object types to refer to the same object instance, the primary key is used match if two object instances of different types refer to the same entity. A *foreignkey* property identifies the property as a pointer to another object, a common practice in Object-Oriented Programming (OOP) languages. Two examples of a type annotation can be seen in Figure 2b: the left is in C# and the right is in Python. Annotations are supported in most programming languages, and are interpreted by a language-specific Frame library.

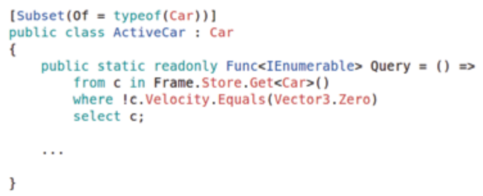
In OOP, inheritance is an important tool for maintaining type definitions small. To illustrate, imagine a type **Car**, containing position and velocity attributes. To make a car that drives in traffic, a new type can be defined, called **TrafficCar**, that inherits from Car and extend it to contain new methods and properties for handling traffic driving, like a *MakeLeftTurn()* method or a *SpeedLimit* property. In HLA, inheritance of FOM objects is possible, in a similar manner of OOP. However, in both OOP and HLA, inheritance is not recognized as identity. This means that two related objects – a Car and a TrafficCar, for instance – are not identified as the same instance. Additionally, since inheritance enforces that all properties of the parent are also available to the children, inheritance can potentially lead to objects bloated with properties and methods. In the context of simulations, it would be interesting to have both a Car and a TrafficCar representation of the same simulated entity instance, and that related types can pick and choose the properties of the car it finds relevant to know about.

There are two *Set* relationships in CADIS with the purpose of modeling static aspects of *Set* entities: *Projection* and *Is-a*. *Projection* acts like a filter, similar to selecting columns in a SQL **SELECT** statement. A *Projection* does not allow additional properties: it is a filtered version of the type projected from. By contrast, the *Is-a* expresses a relationship that extends an existing *Set*. In our example, a TrafficCar would be annotated as *Is-a(Car)*. This annotation infers that a TrafficCar must share an identity (i.e. same *primarykey*) to a corresponding Car object. This way, a simulation fetching *Car* will retrieve all *TrafficCar* objects without having to know any of the methods or properties that were added to *TrafficCar*. This approach allows for a scalable design where previously developed simulations do not need to be aware of future type definitions.

The last two *Set* relationships – *Subset* and *Join* – are used for aspects with dynamic qualities. *Subset* acts like a **SELECT** query with a **WHERE** statement, retrieving a collection of objects with a certain run-time characteristic (see Section 3.3). A *Join* allows new types to be formed by combining other types with a query. If a simulation is interested, for instance, in pedestrians near cars, a *Join* type could match the positions of pedestrians to cars, creating a *PedestrianAndCar* type that represents the collection of all pedestrians near cars.

3.3 Dependent Collection Classes

The relational nature of types in CADIS also enables the use of dependent collection classes: classes whose instances are automatically defined by a given query. Dependent collection classes draw some inspiration from dependent type theory (Martin-Löf 1986) and from the concept of predicate classes (Chambers 1993), both of which are beyond the scope of this paper. The best way of understanding dependent collection classes is with an example of an **ActiveCar** type, representing a vehicle whose speed property is not zero. A definition of an ActiveCar can be seen in Figure 3.



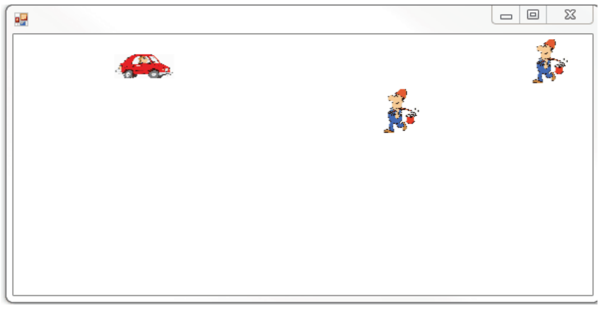
```
[Subset(Of = typeof(Car))]
public class ActiveCar : Car
{
    public static readonly Func<IEnumerable> Query = () =>
        from c in Frame.Store.Get<Car>()
        where !c.Velocity.Equals(Vector3.Zero)
        select c;

    ...

}
```

Figure 3: Class definition of ActiveCar type.

This is a particularly useful tool for establishing a dynamic form of publish-subscribe, similar to the Data Distribution Management service of HLA. Another example – one that we have developed and tested – is the **PedestrianInDanger** type, seen in Figure 4. Suppose the pedestrian simulator wishes to know if a pedestrian will be run over by a car. Instead of polling all cars and matching against position location at every tick, CADIS allows the definition of an PedestrianInDanger, which simulators can declaratively show interest in. CADIS will execute the PedestrianInDanger query at every tick, and return objects who match the query with type PedestrianInDanger. These dependent collection classes with queries are referred to as a *SubSet*, as can be seen in the annotation on the right side of Figure 4. Like *Projection* and *Is-a*, *Subsets* shares identity with the *Set* it relates to. In this example, a *PedestrianInDanger* is the same instance as a *Pedestrian* with the same unique identifier.



```
@SubSet(Pedestrian)
class PedestrianInDanger(Pedestrian):
    def distance(self, p1, p2):
        return abs(self.p1.X - self.p2.X);

    @staticmethod
    def query():
        result = []
        for p in Frame.Store.get(Pedestrian):
            for c in Frame.Store.get(Car):
                if abs(c.Position.X - p.X) < 130 and c.Position.Y == p.Y:
                    result.append(p)
        return result

    def move(self):
        logger.debug("[Pedestrian]: {0} avoiding collision!".format(self.ID));
        self.Y += 50;
```

Figure 4: Type declaration for a pedestrian in danger of being hit by a moving car.

Dependent collection classes can also be combined with *Projection* and *Is-a*. A dependent collection class can, for instance, extend or discard properties of the type it is related to. A PedestrianInDanger type can have only its position, or it can add or override a method originally in Pedestrian. In Figure 4, the *move* method is overridden, so that move in this context means moving in the y-axis to escape being hit by the car.

3.4 Time Flow

CADIS is a time-stepped discrete event simulation framework, meaning it is driven by a clock that ticks in fixed intervals. The primary purpose of CADIS design is to serve as a Distributed Interactive Simulation framework, so it has been designed to run in real-time or scaled real-time. Thus, CADIS does not currently offer time management services: all simulations execute in fixed wall-clock intervals that can be different per simulation. A visualization tool, for instance, may run at a higher rate of updates than other simulators to provide a smoother view of moving objects. Since CADIS uses the **Store** as a central dispatcher, there is no misordering of events. Data modifications performed in the same tick have no guarantees of order, as they are concurrent from a logical clock standpoint. This eventually-consistent approach is used, for instance, in OpenSimulator (OpenSimulator 2015), where synchronization conflicts are inevitable, but speed (i.e. low latency and fast-paced updates) is prioritized over an absolute ordering of events.

If a simulation becomes overwhelmed and cannot perform in real-time, there will be an increased interval between ticks. The **Frame** library will warn the user that the simulation update loop is slower than the requested fixed-interval, but will perform the same operations (i.e. *pull*, *update*, *push*) as fast as possible without sleeping. This way, the executing simulation will still receive updated information but at longer interval time-steps. Additionally, the actual interval of time passed since last tick is made available by **Frame**. Ordinarily, this value would match the fixed-interval. If the simulation exceeds the fixed-time interval, it can check how much time has actually passed so it may adjust its simulation loop to account for the increased delay.

It should be noted that a time-stepped, real-time, zero look-ahead is not a limitation of CADIS design, but rather a trade-off design decision. It is part of our future work to build CADIS on top of the infrastructure of HLA to take advantage of its powerful time flow and conflict resolution mechanics. Nonetheless, for the purpose of this paper, CADIS is focused only in time-stepped real-time simulations.

3.5 Data-Centric

Discrete simulations often rely on **events**: messages describing an occurrence that may cause the state of the simulation to change over time. The event-based messaging system is asynchronous, providing a separation between senders and receivers. Events are powerful solutions for scaling discrete simulations, however they can hinder scalability if not implemented appropriately. One example is *event-chaining*, where an event can fire other events and possibly cause the loss of a deterministic path of execution that can be retraced or even result in an event explosion.

CADIS opts to remove the use of traditional events – and its potential risks – by taking a data-centric approach to sharing state. The data-centric philosophy focuses on the state of simulated entities, instead of loose entity-related events. The data-centric approach of CADIS serves two major purposes: first, it improves transparency of distributed state by automating data synchronization in well-defined steps, which in turn provides a clear definition of when data has been updated without the need for imperative requests. Second, it flattens the structure of chained events. When all events are expressed in terms of data modification, the focus of an event moves from its semantic meaning – which can be hard to understand in chained events – to its symptom: how it affects the simulation entities. In our early observations, a data-centric design process has shown to be easily compatible with event-based approaches (for example, in our evaluation in Section 5). However it remains an open research question to determine if such a conversion applies to the universe of applications in modeling and simulation.

4 PERFORMANCE CONSIDERATIONS

So far CADIS has been discussed from the standpoint of shared memory. The operations of pushing and pulling have assumed retrieving and inserting objects in memory heap, which can be achieved efficiently in modern day computers. CADIS is a distributed architecture, so performance is an issue. Specially, the conceptual model is that all data needed in each partial simulation is pulled from the Store in the beginning of each time step, and pushed onto the Store at the end of each time step. This model, although very simple, can be prohibitively expensive in terms of runtime performance. Therefore, improvements are necessary.

The most important performance improvement concerns data copying between the central Store and each local Frame. With a memory sharing paradigm, the Frames and the Store can simply pass object references for pushing and pulling. If instead the communication between Frame and the Store occurs over a network stack, serialization/deserialization becomes necessary. As such, it is critical that passing of objects be limited to bundled and minimally-sized messages.

In the current version of CADIS, local Frames keep track of any changes performed by the simulation during the update phase. Those changes are saved as key-value pairs of property names and values. When the push phase arrives, the **RemoteStore** pushes those key-value pairs to the Store.

When the Store receives the key-value pairs of modifications, it applies the changes, then adds them to queues corresponding to connected simulations. When a simulation pulls, the Store simply pushes this queue of updates to the simulation. The Frame library receives these updates and applies them to the local buffered version of the objects.

Another improvement on data exchange is on *Subset* pulls. Since a *Subset* will always refer to a *Set*, results of a *Subset* query will retrieve a list of unique identifiers, instead of a list of objects. Frame is responsible for converting the identifiers into actual objects of the type requested, and deliver it to the simulation transparently. JSON is used as a serialization medium over the network stack.

The centralized architecture of CADIS around the Store is a natural concern to scalability. To address this bottleneck, the latest version allows the Store to be distributed in multiple instances, potentially improving scalability by decentralizing the Store service. This feature is particularly useful when there are groups of objects that do not share any relationship, thus having no need to be located in the same Store. A common example is the use of space-partitioning interest management. Following along our previous examples, *TrafficCar* objects in Los Angeles have no need to share the same Store as *TrafficCar* in Tokyo.

Additionally, unrelated *Set* objects can also be stored apart, as for instance a *TrafficCar* and a *Window* (i.e. representation of a window of a house Storing these objects separately would assume cars in traffic and windows of houses share no relationship. This feature has only recently been developed and is not part of this study.

5 EVALUATION

The inspiration for developing CADIS came from the difficult task of building an urban simulation. There are many different aspects to simulating a city, as for instance, traffic engineering, weather, social interactions, and public transportation. In an actual city, there are just as many stakeholders in the form different government departments, which all have studied and possibly built high fidelity model and simulations on their own. CADIS is our response to uniting all these efforts in a singular urban simulation scenario.

For evaluating CADIS, we use an open-source project called **mobdat** – developed by Mic Bowman, one of our co-authors. Mobdat integrates 3 distinct simulators to produce a small-scale urban simulation. The first is a traffic simulator called Simulation of Urban Mobility (SUMO) (Krajzewicz et al. 2012). The second is a widely used virtual environment simulator called OpenSimulator. The third is a social simulator that drives the citizens of the city to take actions according to a pre-defined schedule, part of the mobdat project. All 3 of these simulators are open-sourced and available online. Figure 5 shows the visualization of the urban simulation from an OpenSimulator viewer.

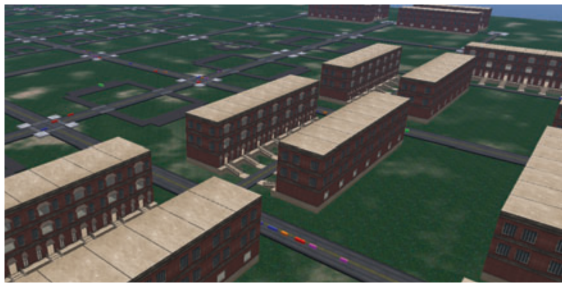


Figure 5: The mobdat urban simulation, seen from an OpenSimulator viewer.

The **mobdat** project and its 3 simulators were converted to CADIS for the evaluation. There were 3 major changes required. First, all simulation entity types were annotated in the style of CADIS. Second, all calls to publishing events were deleted. Third, all event-handlers are called from the update loop of the simulation. The update loop checks with **Frame** if there are new, modified, or deleted objects of relevant types and triggers their respective event-handlers.

The evaluation will compare CADIS to the original implementation of mobdat. We have argued the advantages of CADIS representation of types in Section 3, but it is expected that CADIS is less performant than a pure event-based simulation integration. This evaluation is meant to demonstrate CADIS feasibility and how it fares in comparison to the minimalist event-based approach. Furthermore, CADIS uses HTTP over sockets, and can operate as both parallel and distributed simulation framework. Mobdat uses shared objects, and can only perform in parallel in its current state.

5.1 Experiment

The experiment is an urban simulation focused on traffic, combining 3 interacting simulations to create the routine of a city based on citizens' daily schedules. The routines start with the social simulator – called **SocialConnector** – maintaining a daily schedule of activities for each citizen. Scheduled items in their agendas are sleeping, working, having coffee, lunch, dinner, and shopping. Regarding trips, citizens go to work and back on weekdays and may, depending on their particular preference probabilities, go out to lunch, dinner, coffee shop or shopping. Whenever a citizen decides to travel, the social connector determines the start and end locations for that trip and informs the traffic simulation connector for SUMO, called **SumoConnector**.

SumoConnector uses the trip information to create a vehicle with SUMO, corresponding to the traveling citizen's trip. SUMO will drive it from starting to ending point, also simulating interaction with other vehicles, traffic rules, and traffic lights. As SUMO guides the vehicle, changes in position and rotation are queried by the SumoConnector and forwarded to the virtual environment connector, **OpenSimConnector**.

OpenSimConnector acts as a connector for OpenSimulator, similarly to SumoConnector. The OpenSimConnector receives vehicle updates from SumoConnector, forwarding them to OpenSimulator for creation and update of vehicle instances in the virtual environment.

In the original event-based mobdat, communication between simulations are done through an event router object, responsible for maintaining a queue of events shared between processes. When a traveler is created, an event is put on the queue and delivered to SumoConnector. SumoConnector uses the traveler source and destination information to create a vehicle in SUMO. SumoConnector queries for vehicle updates at every tick, forwarding them as events to the OpenSimConnector.

In contrast, CADIS takes a data-centric synchronization approach. SocialConnector is a *Producer* and *Tracker* of *Vehicle* objects. Hence, when a traveler starts a trip, SocialConnector simply creates a new Vehicle object and adds it in Frame. SumoConnector is a *GetterSetter* for Vehicle objects, receiving the newly created objects from the Store during the next *pull*. The Vehicle object contains the start and end locations for the trip, which SumoConnector uses to create a vehicle in SUMO. When the vehicle starts moving, SumoConnector queries SUMO for updates, and simply update the velocity and position properties of the same Vehicle object retrieved at the pull operation. The OpenSimConnector is a *Getter* for *MovingVehicle*, a *Subset* of Vehicle containing a query for vehicles whose position are different than (0, 0, 0) – the initial value of newly created Vehicle objects. When SumoConnector modifies the position of a Vehicle object, OpenSimConnector receives it as a list of MovingVehicles every tick. Each MovingVehicle object is used to create and update a virtual vehicle object in the OpenSimulator virtual environment.

The simulation is executed for 20 minutes with 200ms per step, adding up to 6000 steps. Each step corresponds to 2 seconds in simulation time, adding to a total of 3 hours and 20 minutes of simulation execution. The simulation starts at 7 AM in the simulation time, used to trigger citizen's actions based on their daily schedules. 2000 travelers are picked at random from the pool of 21,312 citizens and used to generate trips, which at 7 AM are mostly from their homes to work, though some stop for coffee then head to work.

The experiment consists of 4 different executions. The first execution (**Event-based**) runs the original event-based mobdat in parallel, using shared memory. The second execution (**Unoptimized**) runs CADIS without performance improvements, using loopback sockets that separate the Store from Frame instances. The third execution (**Optimized**) adds several performance improvements discussed in Section 4, but also runs on loopback sockets. Finally, the fourth execution (**Remote**) uses the Optimized version of CADIS in a Local Area Network (LAN), with the Store running in one location and the 3 simulators in another.

5.2 Metrics

The purpose of the experiments is to show a realistic comparison between a conservative event-based approach of distributed simulation to CADIS. Thus, the metrics of the experiment are related to performance, in particular, processing time spent per step of the simulation. The event-based mobdat is not time-stepped, so measuring the processing time of a time step (i.e. time advance) handler routine will not yield the total processing time of the simulation for that step. Instead, we measure the sum of the time spent in all event handler functions called during each step. In CADIS, each step is the sum of the time spent in push, pull, and the simulator's update functions.

The CADIS adaptation of mobdat only modified the data fetching mechanism. Originally, each event handler received, for instance, a new vehicle or a vehicle update. When several vehicles were created or updated, the event handler was called multiple times. CADIS runs the same method, but instead fetches

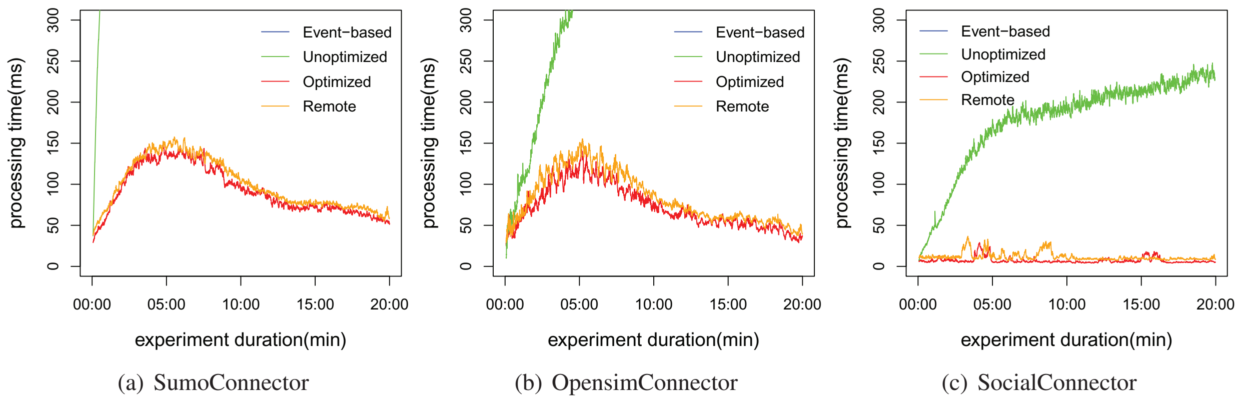


Figure 6: Performance comparison between the original event-based mobdat with the CADIS version. The **y-axis** is processing time in milliseconds for each clock event. The **x-axis** is minutes of wall-clock experiment time.

all updated and new vehicles with a call to the Frame library. The treatment for each individual object is exactly the same in the original and CADIS version of mobdat.

6 RESULTS

Figure 6 shows processing time for each of the simulators running the 4 versions of mobdat. Table 1 shows the means for each version and for each simulation. Numbers for the **Unoptimized** 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.

	E.B.	CADIS	Optimized	Remote
SumoConnector	70.53	2395.00	91.72	99.26
OpensimConnector	23.80	328.17	71.94	83.10
SocialConnector	0.51	174.35	7.04	11.58

Table 1: Processing time in milliseconds for each version of mobdat execution in the experiment. **E.B.**: Event-based.

There are noticeable performance differences between Figures 6a, 6b, and 6c, caused by the different level of responsibility each simulation has. The SumoConnector (Figure 6a) has the task of both querying SUMO for updates and of pushing updates to CADIS, making it the heaviest simulation in processing time. The OpenSimConnector (Figure 6b) is the second heaviest load, having to both receive constant updates to Vehicle objects and forward them to OpenSimulator. The SocialConnector (Figure 6c), 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 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 – as discussed in Section 4 – 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.

	C.push	C.pull	C.update	O.push	O.pull	O.update	R.push	R.pull	R.update
Sumo	2222.58	105.12	67.23	40.93	5.45	45.27	44.78	8.57	45.83
OS	0.01	318.66	9.42	0.01	53.22	18.65	0.02	64.14	18.87
Social	0.98	173.22	0.09	0.83	5.95	0.20	1.27	10.03	0.21

Table 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.

7 CONCLUSION AND FUTURE WORK

We have presented a novel approach to distributed simulations, with the intent of reducing the entrance barrier and improve collaboration between simulations. The CADIS framework explores the object-oriented paradigm with a distributed systems view, offering new ways to handle shared state as shared objects. The design philosophy of CADIS is to be simple to develop for, with transparent data sharing in native source-code. 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.

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. Our next study will include evaluation of the distributed data store mentioned in Section 4 which should greatly improve scalability.

REFERENCES

Boer, C., A. Bruin, and A. Verbraeck. 2006, dec. "Distributed Simulation in Industry - A Survey Part 2 - Experts on Distributed Simulation". In *Proceedings of the 2006 Winter Simulation Conference*, 1061–1068: IEEE.

Chambers, C. 1993. "Predicate classes". In *ECOOP'93-Object-Oriented Programming*, 268–296. Springer.

Gianni, D., A. D'Ambrogio, and G. Iazeolla. 2011, September. "A Software Architecture to Ease the Development of Distributed Simulation Systems". *Simulation* 87 (9): 819–836.

IEEE Standards Association 2000. "1516-2000 IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)-Framework and Rules".

Krajzewicz, D., J. Erdmann, M. Behrisch, and L. Bieker. 2012, dec. "Recent Development and Applications of SUMO - Simulation of Urban MObility". *International Journal On Advances in Systems and Measurements* 5 (3&4): 128–138.

Martin-Löf, P. 1986. "Intuitionistic type theory". *The Journal of Symbolic Logic* 51 (04): 1075–1076.

OpenSimulator 2015. "OpenSimulator". http://opensimulator.org.

Orfali, R., and D. Harkey. 1998. *Client/Server Programming with Java and CORBA (2Nd Ed.)*. New York, NY, USA: John Wiley & Sons, Inc.

Tang, S., T. Xiao, and W. Fan. 2010. "A collaborative platform for complex product design with an extended HLA integration architecture". *Simulation Modelling Practice and Theory* 18 (8): 1048–1068. Modeling and Simulation for Complex System Development.

Taylor, S. J. R., J. Saville, and R. Sudra. 1999. "Developing interest management techniques in distributed interactive simulation using Java". In *Simulation Conference Proceedings, 1999 Winter*, Volume 1, 518–523.

Topçu, O., and H. Oğuztüzün. 2013. "Layered simulation architecture: A practical approach". *Simulation Modelling Practice and Theory* 32:1–14.

Zeigler, B., D. K. D. Kim, and S. J. Buckley. 1999. "Distributed supply chain simulation in a DEVS/CORBA execution\nenvironment". In *Proceedings of the 1999 Winter Simulation Conference*, Volume 2, 1333–1340.

Zeigler, B. P., Y. Moon, D. Kim, and J. G. Kim. 1996, jan. "DEVS-C++: a high performance modelling and simulation environment". In *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on* ,, Volume 1, 350–359.

AUTHOR BIOGRAPHIES

ARTHUR VALADARES is a PhD Informatics candidate in the Bren School of Information and Computer Sciences at the University of California, Irvine. His research is centered in scaling virtual environments to accommodate thousands of users and virtual objects. He received his M.S. at University of California, Irvine, and his B.S. at Universidade Estadual de Campinas in Brazil. His e-mail address is avaladar@ics.uci.edu.

CRISTINA V. LOPES is a Professor of Informatics in the Bren School of Information and Computer Sciences at the University of California, Irvine. Her research focuses on software engineering for large-scale data and systems. She has a PhD from Northeastern University, and M.S. and B.S. degrees from Instituto Superior Tecnico in Portugal. Her email address is lopes@ics.uci.edu.

ROHAN ACHAR is a PhD software engineering student in the Bren School of Information and Computer Sciences at the University of California, Irvine. His research is on software engineering and programming languages. He received his M.S at University of California, Irvine, and his B. Tech at National Institute of Technology Karnataka in India. His e-mail address is rachar@uci.edu.

MIC BOWMAN is a principal engineer in Intel Labs and leads the Distributed Ledgers research project investigating algorithms and system architectures for scalable, blockchain-based distributed ledgers. He received his BS from the University of Montana, and his MS and PhD in Computer Science from the University of Arizona. He joined Intel Architecture Lab in 1999. While at Intel, he developed personal information retrieval applications, context-based communication systems, middleware services for mobile applications, and scalable virtual environments. In addition, he led the team that built and deployed the first version of PlanetLab, a global testbed for networking research. His email mic.bowman@intel.com.