UMEÅ UNIVERSITY

STREAMING DATA MODELS FOR DISTRIBUTED PHYSICS SIMULATION WORKFLOWS

Martin Vikdahl

Master Thesis, 30 credits
Supervisor: Paolo Bientinesi
External supervisor: Nils Hjelte
Civilingenjörsprogrammet i teknisk datavetenskap
2020

Abstract

This project explores the possibility of lowering the barrier of entry for integrating a physics engine into distributed organization-specific pipelines by providing an interface for communicating over the network between domain-specific tools. The approach uses an event-driven interface, both for transferring simulation models incrementally as event streams and for suggesting modifications of the models.

The proposed architecture uses a technique for storing and managing different versions of the simulation models that roughly aligns with the concept of event sourcing and allowed for communicating updates to models by only sending information about what had changed since the older version. The architecture also has a simple dependency management system between models that takes versioning into account by causally ordering dependencies. The solution allows for multiple simultaneous client users which could support connecting collaborative editing and visualization tools.

By implementing a prototype of the architecture it was concluded that the format could encode models into a compact stream of small, autonomous event messages, that could be used to replicate the original structure on the receiving end, but it was difficult to make a good quantitative evaluation without access to a large collection of representative example models, because the size distributions depended on the usage.

Acknowledgements

I would like to thank everyone at Algoryx Simulation AB for letting me do my thesis project with them, in particular my advisor Nils Hjelte who has given me directions and support throughout the entire project and Martin Nilsson who provided feedback and helped proof-read this report. I also want to thank my advisor at Umeå University, Paolo Bientinesi, for offering advise along the way.

Contents

List of Figures

1 Introduction

This report deals with exploring the possibility of lowering the barrier of entry to integrate a physics engine into a distributed domain-specific pipeline.

The simulation-related data that is communicated in such a pipeline is the information required to describe the scene that the simulation acts upon. It could include positions, velocities, shapes, masses, joints, shafts, gears, pumps, engines and so on. This information is all organized and stored as simulation models. In a distributed context, there might be several services that need access to viewing and updating these models.

The solution that is evaluated has an event-driven architecture that uses event messages to communicate over a network. The technique used for storing and managing different versions of the simulation models is a version of event sourcing, which is explained in Section 2.3, but with relatively low-level events. The main focus of this report lies on the integration of a tool for modeling simulation components for Algoryx's physics engine AGX Dynamics. The particular tool integration examined is specialized on modeling actuator simulations, e.g. drivetrain or hydraulics.

1.1 Motivation

Accurate physics simulations have many application areas across different industries. Physics engines such as Algoryx's physics engine AGX Dynamics must therefore be able to integrate well in many organization-specific pipelines. This includes working together with different tools. Such tools could for example be used for setting up the simulation scenario or visualize the simulation when it is running. The challenge is creating a tool that interacts in a modular fashion with the physics engine and can efficiently be integrated with the rest of the simulation environment. A way to lower the barrier for doing so could be to provide interfaces that are weakly coupled with the physics engine and does not rely on the engine developers to provide an explicit interface for every specific use case.

This thesis specifically focuses on web-based tools and how those could more easily communicate with a physics engine. The format best suited for communicating the necessary information over the network might not be the most intuitive format for the tool to use when speaking to the physics engine. Algoryx already had a structured tree-based format in development which is a part of something called "Brick." That format could be used to interface with tools, but sending data using that format over the network would be sub-optimal when the data rate or amount is limited, especially for a real time application and when dealing with large scenes.

This project aims to explore the viability of event-driven interfaces for communicating simulation models over the network.

1.2 Problem Statement

This project aims to evaluate a method that uses an event-driven architecture to communicate between interactive web-based tools or applications and a physics engine. The purpose is integrating the tools and the engine into a distributed pipeline that involves real time physics simulation. A structured data model is exposed to the tool while the declarative, revision-based format is used behind the scenes for communicating over the network with the physics engine. The purpose for the declarative interface is mainly for a client and a server to communicate about a simulation model. This includes the server describing the state of the simulation models and incrementally over time sending messages that report changes that has happened to the models' state. It also includes the client proposing new changes to the model to the server, based on the actions taken by the tool user.

1.3 Problem Statement Clarifications

Here is an example on a typical workflow: The user tool displays some visual representation of the simulation model to the user based on the structured data. When the user performs an action in the tool that should change the model, such as setting some value, adding a component or connecting two components, a change request is automatically sent to the server. The server validates the change and reports the update to the client that requested it, but also to other clients that might use the same model. The revision-based format is the interface used to communicate information between the client and the server about the model.

The goal of the interface that this thesis project produces is to be powerful enough to be able to articulate the definitions for the components that the physics engine uses, down to describing the structure of a 3D vector or what constitutes a rigid body. But in terms of these underlying components, it should also be able to define models that describe the particular setup that the user wants to simulate.

The solution should also include some form of version handling using a low-level variant of event sourcing. It does not need to be fully featured, but should lay a foundation flexible enough to allow for the possibility of adding a more advanced version handling system in the future. The same is true for dependency management. Dependency management means allowing the data of one model to contain references to other models, but also includes how this system relates to the versioning system. These features would be important in order for this solution to be usable in a larger, more complex ecosystems. The system also needs to be scalable to large projects with many simulation models and multiple microservices (explained in Section 2.2) having shared access to the large pool of simulation models. To achieve scalability, a tool should be able to request only the subset of the data that it is interested in from the server.

To fulfill these requirements an event-based interface with an accompanying architecture was proposed. A prototype was implemented to evaluate the proposed method. Analysis and experiments with this prototype are used to investigate demands on data communication in terms of state consistency and network load.

The implementation is not fully featured, so the parts that were not explicitly prototyped are only analyzed theoretically. In practice the prototype is only used with Algoryx's hydraulics and drivetrain systems but the general principles should be applicable to other simulation systems as well.

1.4 Thesis Outline

Section 1.5 below contains descriptions of previous publications and commercial solutions that relates to this work. Chapter 2 is dedicated to background theory that aims to help the reader understand the rest of the thesis. This includes both a description of Algoryx's framework that this project works within, found in Section 2.1, and then explanations for important concepts related to the project in the following sections. Chapter 3 describes the proposed architecture; mostly focusing on the parts that are implemented in the prototype, but it also mentions parts of the architecture that is beyond the scope of the prototype. Chapter 4 presents quantitative results from running the prototype implementation and collecting metrics. Chapter 5 contains some theoretical analysis of the properties of the final system design and considers alternative solutions that could have been used, but for one reason or another, were not. Chapter 6 finally summarizes the conclusions reached in the report and highlights areas for future work to explore.

1.5 Related Work

Caroline Desprat, Jean-Pierre Jessel and Hervé Luga from the University of Toulouse have designed 3DEvent which is a framework that uses event sourcing for 3D web-based collaborative design [4]. It has many of the same challenges and goals as this project, aiming for a flexible, loosely-coupled and non-monolithic architecture that scales well. An immediately apparent similarity is the fact that it is dealing with a distributed event-driven architecture that utilizes event sourcing for web-based collaborative editing tools. It also takes advantage of the fact that the problem domain allows for efficient encoding of the data.

It employs peer-to-peer technology (P2P), unlike this paper which instead focuses on using a central synchronization point for validating and globally ordering the events. The solution in this paper mainly targets a low-latency environment. Although P2P is not in focus here it might still be a realistic extension and P2P alternatives are discussed briefly in Section 5.7. Another difference to this thesis is that 3DEvent was not developed with physics simulation in mind, but rather 3D geometry manipulation.

An updated version of that architecture was later proposed in a separate paper 2018 [3].

Andrzej Debski et al. [2] concluded in a 2017 article that CQRS architectures (explained in Section 2.5) with event sourcing is horizontally scalable and provide eventual consistency. They also mentioned that 'event versioning may be solved in multiple ways' and lacked a commonly accepted best practice. Although that article is not concerned with the same problem domain as this paper, it seems to have experienced positive results using similar techniques and architecture.

Dennis G. Brown et al. presented already 2004 [1] an event-based distribution system for a mobile augmented reality system called BARS that works for both virtual reality and mobile augmented reality. The paper claims that the method works well on unreliable network connections, is able to handle many types of users, works well for both low and high bandwidth application areas, and has sufficient flexibility [1].

Outside of academia, there has been some commercial projects that tackle similar problems as well. **NVIDIA Omniverse** is a non-monolithic collaborative platform from NVIDIA which has a lot of similarities to this thesis project. Omniverse is based on Pixar's universal scene description (USD) and has the goal to connect tools from different vendors together into one real-time platform focused on 3D content creation and this includes real-time integration of physics simulations with the PhysX physics engine. It also includes asset management, layers, and version control [13].

Not much information about Omniverse had become accessible to the public up until right before the end of the project. Thus, NVIDIA's solution has unfortunately not been available for comparison with our solution within the scope of this project.

Since this paper revolves around access to and modification of data, there are also parallels that can be drawn to some databases. **Event Store** is a database that is specifically made with event sourcing in mind [7].

Datomic is another similar database. In it, all stored data is immutable and it keeps the complete history over previous states [15, 10]. The fact that data stored in the database never expires is very convenient for caching. Caching and replication can be done automatically in peer servers [15]. The whole database can use a persistent data structure where reads can read the immutable data while writes only add new data without changing the old, thus writes can be performed without coordination with reads [10].

2 Background

2.1 Algoryx Simulation, AGX Dynamics and Brick

The thesis project is performed in cooperation with Algoryx Simulation AB, a company specialized in accurate engineering grade simulations, using the AGX Dynamics physics engine. AGX Dynamics is a professional multi-purpose physics engine developed by Algoryx Simulation. The company is increasingly focusing on the development of integrated, non-monolithic, engineering pipeline tools and end user applications. This is critical for taking advantage of simulation results in real-world use cases. It allows domain specialists and non-simulation experts to be included in this workflow.

Algoryx has many complementary modeling tools for mechanical systems. Working with CAD models is done using Algoryx Momentum. Modeling simulation runtime environments is done using AgxDynamics for Unity and AgxDynamics for Unreal. This is further complemented by additional customer specific tools and applications in a web-based environment.

AGX Dynamics has a file format for storing the state of a simulation. The format uses the file extension ".agx". That format is not impacted by this paper but is used as a contrasting example. It is a serialization of raw data where the structure is known beforehand and does not need to be encoded in the file.

Brick is the name given to a modeling format used by Algoryx as well as a few tools that make use of that modeling format. The idea for Brick is to be a front-end modeling interface for AGX Dynamics. The tools can perform marshaling and unmarshaling to/from Brick files among other uses. The tools and format are still under development.

This project explores the possibility to add an event-based streaming interface as part of Brick that can be used to more easily connect it to other tools.

2.1.1 The Structure of the Brick Format

The internal representation of Brick is divided into models. Models are trees that can store a wide variety of structured data. Each model can contain variables that have names, data types and values. They are like structure definitions that can optionally contain default values. Models can also inherit the content of other models and can then choose to override inherited values as well as adding new variables. Models can depend on other models either through member variables having other models as types or through inheritance which could lead to a potentially complex dependency graph.

In the future, models might also get custom constructors and associated functions (or at least function-like macros) and could at that point aptly be likened to class definitions in object oriented programming languages. However, instances of models can also have new variables added to them dynamically.

Brick will contain many models that represent specific data structures used in AGX Dynamics. These models could be read, and can be a part of the dependency graph, but should not be edited by tool users. Other models represent the users' simulation models and these are the ones that a user might want to edit.

Brick's structural model format can be stored to disk using a YAML-based format. That format is separate from the ".agx" file format.

2.2 Microservices

Microservices is the idea that an application can be made up from several independent smaller services instead of having a single large unit often called a "monolith". There are some problems with monolithic software that the microservice pattern is trying to address. Some difficulties are the fact that maintenance can be complicated, implementing new features can take a long time, scalability is sometimes poor, deployment of new features often impacts the availability of the entire system, albeit temporarily, and it is very difficult to make architectural changes to the application [14].

Each microservice should have a single responsibility and it can be deployed, scaled, and tested independently [11]. This way they can be part of a larger distributed system and delivered on demand.

2.3 Event Sourcing and Event-Driven Architecture

The idea of event sourcing is that the state of the application is stored as a series of ordered events. Each event represents a fact that captures a change to the system's state [2, 8]. This event log is the ground truth for the system. Applying a change to the state of the system is done by creating a new event that captures the change. Event sourcing is in theory "append-only" [8]. This means that all data is immutable and when the user wants to change something they simply add an event that represents that change (or a series of events depending on the nature of the change) to the event log.

The entire application state could be completely rebuilt from scratch by applying the events in the event log in the order they appear. All the past states of the system could be reconstructed by reading and applying the event log up to some specific point and the system could thus query previous states of the system as well as introducing retroactive changes by injecting event messages into the event log at an earlier point and recalculating the application state after that point as if the new event had occurred. The system could be queried as of specific past states and the series of events could be replayed.

Event sourcing is versatile and has many potential application areas. Some use cases of event sourcing is version control systems and accounting systems [8]. An advantage of event sourcing is that caches of the application state can be useful even after the state has changed, since the new state can be constructed by simply applying the new changes to the cache and snapshots of the application state can be created from the event log at any time without affecting or interrupting the running application itself [8].

The main advantage of event sourcing is the fact that past states can be reconstructed and that the complete event log is available at all times. Reconstructing older states does however come at a computational cost, especially if the event log is long. Some event sourced systems also allow for recreating past events by stepping backwards through the event stream, applying the inverse action to the events starting with the latest event. Event sourcing is good for building scalable systems, with an event-driven architecture [8].

Event-driven architecture loosely means that systems communicate via event messages. It is especially advantageous if the system is distributed and non-monolithic with few writers but many readers.

2.4 Domain-Driven Design

The term Domain-Driven Design (DDD) was coined by Eric Evans and is described in his book "Domain-Driven Design: Tackling Complexity in the Heart of Software" [6, 12].

DDD is an approach to software development that focuses the design around the processes and behavior of the domain [2, 6]. The design of the software should center around the domain that the software models and the mental model the developers use when designing and modeling the software should closely match the mental model domain experts use to reason about the domain [6].

2.5 CQRS

Command Query Responsibility Segregation (CQRS) means that queries should be separate from commands. Queries are immutable requests for data and do not have side effects, i.e. they do not change the state of the queried system. Commands are requests for changing the state [2].

CQRS works well with event sourcing and the two are often used together. When CQRS is used alongside event sourcing, the commands produces events that the read view has access to. A command might be rejected by the system in which case it has no effect.

Queries do not have side effects, they only return data. Updates can be read from several consumers and reads can be fully lock-free.

Because CQRS splits queries from commands, both sides can be optimized independently from each others, each focusing on the task it needs to to efficiently, which is reading and writing, respectively. It is possible to use different kinds of databases to interface with each and they can have independent scalability and availability for reading and writing.

CQRS is not a top-level architecture and can be applied on a limited part of a system where it is needed [12]. It allows for the introduction of eventual consistency which is good for low-latency systems and improves scalability, especially in systems where reads are more common than writes [12].

3 System Architecture

A design for the system architecture is elaborated in this chapter. A simple prototype was implemented in order to test the proposed architecture and to help highlight strengths and weaknesses with the design.

3.1 Prototype Scope and Delimitation

Some specific points that the prototype implementation has to include are the following:

- Defining a minimal set of stream operation primitives that can be used to (fully or partially) describe the construction and incremental mutation of the simulation models (in the structure of the existing Algoryx Brick modeling format). This is explained in Section 3.3.2.
- Decoding and encoding the stream into the structured Brick runtime AST representation, allowing the tool to use the structured model API and not explicitly manage the stream.
- Load model dependencies as separate streams and piece all this information together into a larger system. This is useful when the actuator system model is loaded as part of a larger simulation model, when e.g. including vehicle mechanics modeled in other tools.
- Allow for incremental changes to be suggested by a writer client and the change be validated and then communicated from the server to all listening clients.
- Calculate the sizes for the theoretical optimally encoded event messages in the stream so that this can be used in the analysis.
- The ability to compile to WebAssembly for use in the frontend application, so that the same implementation can be reused both in the front-end and the back-end.

The prototype implementation is limited to defining and editing the models which sets up the initial state that the simulation can start running from. The does not include streaming data when running the simulation although the implementation is designed in such a way that allows for future development of efficient streaming of runtime state using the same mechanisms.

3.2 Architecture Design

As described in Section 2.1.1, Brick's structured format is divided into models. The data of these models are converted into event streams and all further changes of the model are also recorded as new events, appended to the end of the stream. A single server should be responsible for handling which changes are accepted to be part of the event stream, but the stream itself can in theory be replicated to any number of clients, servers or databases and the local copies can be stored, ready to respond to future queries. This makes scaling out trivial and provides horizontal scalability properties, at the very least for reading. The event log could be the primary format that stores the models on the disk, possibly complimented with structural "snapshots".

Multiple tools should be able to connect to the network simultaneously, allowing real time collaboration. Complex conflict resolution during concurrent editing is out of scope for this project. For now, all write requests must synchronize in a single server which can simply reject changes that conflicts with the current state of the model.

The important thing is that the most recent versions are known to the server responsible for appending new events to the stream, so that no conflicting versions are created. Subscription can be used to keep all replicas up to date. When an update to one of the models occurs the new events can be transported to all nodes in the network that subscribes to that stream. That means that consistency is achieved eventually, but there is nothing that guarantees that the most recent version exists in all replicas at any given time.

In the receiving end of the event stream the stream is used to reconstruct the tree-based internal representation of the system which is kept track of in addition to storing the event log itself. If the model depends on other models and the client has not yet learned about those models, it requests those as well and receives those as separate streams. Before ever communicating with the server, the client would still have local knowledge about a base library of core models, so that those never needs to be sent explicitly.

Each model has have an accompanying version number that is included when the model is referred to. That way, the receiver knows if its local version of a model is out of date. In the prototype implementation the version number is simply the number of events in the stream. Since events are append-only, the number of events should be strictly increasing. However, it is very likely that this versioning system would not be the best idea for a production-ready system. This is discussed further in Sections 5.1 and 5.2.

In the client end, the resulting tree after decoding the event stream is what is exposed by the API that faces the rest of the user application. This API provides read-access to the structured data which allows for a simpler mental representation that the user application can reason about. However, instead of directly modifying the local copy of the tree in the client, the user requests changes from the server. In this system design, it has been decided that requests are to be encoded in the same format as events. Even so, these are only supposed to be seen as commands, operations or preliminary events and are not at that point yet part of the event stream.

The changes that the user requests in the tool are automatically encoded into preliminary events and these are sent to the server which then validates and synchronizes the new information, updates the event log and updates its own data tree. It then reports the changes to all listeners, which probably includes the client that requested the change. Only then is the tree representation in the client updated.

In Figure 1 is a visual depiction of the process described above. Remember that requests and queries/subscriptions are completely separate from one another. A subscriber to a model does not have to be the one the sends the change request. All subscribers get all event messages regardless of who requested it. A client that wants a version of the stream, but does not want to subscribe and get future updates can make a one-off query for a specific version. If

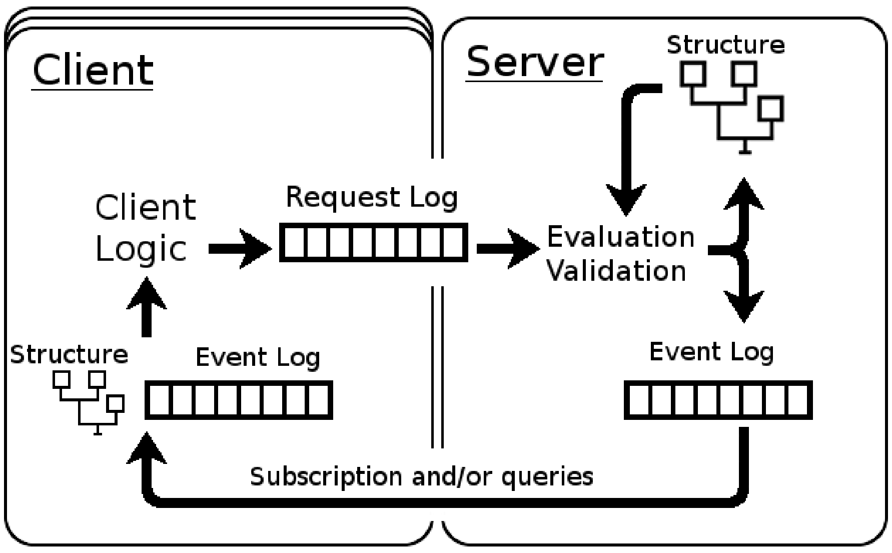


Figure 1: A visualization of the dataflow when editing models. The arrows show what affects what. A request in the client is sent to the server which evaluates the change and produces one or more event and updates its local structural representation of the system which is used in the validation process. When an even is added to the event log in the server, it is automatically sent to all subscribers of the model but copies of the event log can also be sent on request to any non-subscriber. When an event is received in a client, it is stored and the client's local tree structure is updated.

the client already has some version of the model locally, it can request to only get the event messages since that version.

Having the server responsible for reporting all changes means that the server decides the global ordering of events. This makes sure that consistency is obtained even if multiple simultaneous writers are allowed.

All connections between the server and the listener streams need to use TCP or some at least some method that ensures ordering and no data loss since every event is important and so is the order between them.

3.3 Event Message Design

There are many ways to design the event messages in the event log. Alternatives to the approach presented here are mentioned in Section 5.5.

The layout of the event messages used in the proposed architecture are as follows. The event log consists of discrete messages where each message describes one event. The event message starts with an event type ID which specifies what type of event the message represents. This, in turn, determines what data the rest of the message should contain. Both declaring type schemata and assigning values to an existing schema use the same format but the two use cases might use a different subset of the operations. A small data format is a priority to avoid data transfer becoming a bottleneck.

The format is a modeling format, but it is also used as a serialization format. Serialization means converting data into a serial stream of data that is suitable for transportation over a network. In this report it refers to the action of converting from the structured brick format into an event stream. Deserialization in the context of this report is the inverse action of serialization which restores it to the structured brick format. Of course, if the event log is the primary representation of the simulation models, then converting to that format from the structured format would be more rare than doing the inverse. Events that are added to an event stream would become a part of the serial stream before having an effect that is reflected in the internal structured format. Nonetheless, the act of interpreting an event message and affecting the data structures accordingly is still referred to as deserialization in this report.

3.3.1 Data Types

The exact set of data types is not that important for the theoretical evaluation of this approach, but it does nonetheless influence the outcome from the quantitative analysis and is therefore documented here. A set of very general and versatile data types has been chosen for this prototype. Below is a list of data types that the fields in the event messages can consist of. In Section 3.3.2, the event message layout is described in terms of these. It is worth noting that the fields were not encoded exactly as described here in the real prototype. Because the prototype had a limited development time and resources it is partially simplified. For example, paths are encoded as strings under the hood, rather than using the more compact format described in 3.3.5. But the encoding described here is the theoretical format which can realistically be implemented and the paper takes the differences into consideration when calculating and comparing the theoretical and actual demands on the data communication.

- `bool` A Boolean value
- `real` A double precision floating point value
- `utf-8_string` A null-terminated utf-8 string. The Brick format allows assigning expressions as values to variables. This might include referencing other variables or performing arithmetic operations. These expression values are currently also encoded as this kind of string when put into event messages rather than using some more complex encoding. This behavior is chosen due to simplicity for the prototype and not because it was determined to necessarily be the best solution.
- `ident_string` A string that can encode the character set that identifiers can consist of. Since the names for variables, namespaces and models are fully ASCII-compatible in this system, we do not need unicode support. A small optimization would be to use the bit that goes unused in the ASCII format as a terminator instead of a null character. That would save one byte per identifier string. Of course, if unicode support were desired, than this type of string could just as well be encoded identically to the utf-8 string.
- `s_varint` A signed variable size integer. How it is encoded is explained in Section 3.3.4.
- `u_varint` An unsigned variable size integer. How it is encoded is explained in Section 3.3.4.
- `path` A valid existing path in the tree structure of the data. How it is encoded is explained in Section 3.3.5.
- `scalar_param` Can be `s_varint`, `real`, `bool` or `utf-8_string`. Which it should be interpreted as must be deduced from the context.

- `param` Similar to `scalar_param`, but has even more potential values, such as being a reference (a `path` to a different variable) or being empty (zero bytes). Which it should be interpreted as must be deduced from the context. As explained in Section 3.3.3 there are four special constructors that does not implicitly know the type of the values they are creating, so the parameters for those four constructors includes the type ID that they represent.

- `param_variant` This is the same as `param` except it is used in contexts where the types for the four special constructors mentioned in Section 3.3.3 can be deduced, so the type IDs does not need to be included.

3.3.2 Message Types

The following list shows the message types that are present in the prototype implementation. These messages are not only used as events, but are both used as events in the event stream and as requests for new changes. They will generally be referred to as "events" or "operations" here, but should be understood as referring to either events or requests. For each message type in the list below, there is a short description followed by a visualization of the layout. The visualization shows how the message type is serialized in terms of the data types described in the list of section 3.3.1. The field called "opcode" which is present in every message type is the event type ID, unique for every type of event.

Note that the names of the operations as written here are not written in a format that necessarily fits either as events in an event log or as commands with proposed changes. A better event naming convention is discussed in Section 5.5.

- **variable** Create a new variable with a name and a type and give it a value based on a constructor and input parameters to that constructor.

opcode	location	name	constructor ID	value
`u_varint`	`path`	`ident_string`	`u_varint`	`param`

- **data** Sets the value of a variable based on a constructor. This operation does the same thing as the "variable" operation except creating and naming the variable. Instead, it assumes that the variable already exists.

opcode	location	constructor ID	value
`u_varint`	`path`	`u_varint`	`param_variant`

- **literal** Sets a value just like the "data" operation, but without specifying the constructor in the message. Instead the type is inferred from the specified location of the variable (which has to already exist beforehand) and only scalar types are supported (int, real, bool or string). This event type is a variant of the event type "data" but using one less `varint`. The reason why it cannot encode as many data types as "data" is that the type of the value has to be deducible from the location, but just knowing the type does not help to know what constructor to use if constructor support were fully implemented, and a value of a child type should be assignable to a variable of its parent type.

opcode	location	value
`u_varint`	`path`	`scalar_param`

- **delete** Remove a variable from the model.

opcode	location
`u_varint`	`path`

- **type-entry** Add a type to the type table and add the type's constructor(s) to the constructor table. As mentioned, constructors are not fully implemented, so each type has exactly one constructor (except generic types, those lack constructors, but the only generic type in the prototype is `List`). These tables keep track of the IDs for types and constructors used in all the other event messages. The type path is an `ident_string` rather than a `path` because making module paths part of the path encoding system described in Section 3.3.5 would add a lot of complexity and constraints to the design. There might be some clever way to compress this field, but for now it is represented by a string.

opcode	type paths	revision
`u_varint`	`ident_string`	`u_varint`

- **composite-type-entry** Just like type-entry, but for composite types i.e. specializations of generic/template types. There is only one generic type at the moment and that is List. So to create a list containing a specific data type you create a composite-type-entry that specifies the type that links the generic type List to the inner type. This should in the future work with other generic types as well, once that is implemented.

opcode	data type ID 1	data type ID 2
`u_varint`	`u_varint`	`u_varint`

- **origin** An operation message that appears exactly once per model and specifies whether or not the model inherits from another model or not and if so, which one. The type ID parameter is set to -1 if the model doesn't inherit, so the data type ID `varint` has to be signed.

opcode	data type ID
`u_varint`	`s_varint`

- **scope-push** This operation is usually paired with a "scope-pop" message and together they introduce statefulness; they affect how the messages between them are interpreted. Every operation between these two, (except operations that handle the creation of type entries,) operate within a specified scope. The scope is a path into somewhere in the model which means that all paths are relative to that place, rather than the model root. This is just used for size optimization. They allow event messages between them to not have to specify the full paths. The "scope-push"-operation could be seen as an operation that moves around a cursor or marker that remembers a stack trace of all the places it has been to and "scope-pop" could be seen as the operation that tells that marker to pop an element from the stack and return to that place.

opcode	location
`u_varint`	`path`

- **scope-pop** This operation is explained above.

opcode
`u_varint`

- **create** A combination of the "variable"-operation and "scope-push"-operation. Just like the "scope-push"-operation, it needs to be paired with a "scope-pop". This is an example of a higher level operation that is introduced to increase stream compaction. It is included in the prototype because it was anticipated that creating a variable, then immediately doing things inside that variable's scope, would be a common use case.

opcode	location	name	constructor ID	value
`u_varint`	`path`	`ident_string`	`u_varint`	`param`

3.3.3 Constructors and Parameters

For each stream, the encoder and decoder must keep track of which types the model depends on and an ID number is assigned to each one, and are stored in a table. A separate table stores all constructors for these types and these are assigned separate IDs, but the table links each constructor to the type it constructs.

The prototype has a very rudimentary constructor handling implemented. Each type only has one constructor. Constructors for scalar types takes the data type they are as an argument while the ones for model data types take no arguments and sets default values for all members.

The ability to create constructors that take arguments and perform special tasks is an interesting idea that could be implemented, but is not part of the prototype implementation nor is it defined more precisely in this architecture design. If this more complex constructor behavior were implemented it would probably also be combined with the ability to create custom functions (or at least function-like macros) and evoking them. The design of the parameter layout in the event messages would have to account for more complex arguments for this to work. More on this in Section 5.4.

That said, beyond making the event streams more compact at the cost of adding complexity to the system, having functions could also help with making the user's intent documented more explicitly in the event stream.

Four special constructors that are part of the prototype are the constructors for constructing references, expressions, null values and undefined values. They are special because they are not connected to specific types. The same constructor could in these cases create values of different types in different contexts. To figure out which type they are, there are two strategies. The first is to attach a type ID along with the constructor arguments. The second is to infer the type based on the variable it is targeting.

3.3.4 Encoding Numerical Values

The prototype implementation uses dynamically sized integers rather than specifying several data types representing different fixed-sized integers. More specifically, it uses variable length integers in the LEB128 format. The first bit per byte in the encoded number is dedicated to mark whether that byte is the final byte of the number or not and the remaining seven bits per byte encodes the actual value [9]. This has advantages and disadvantages. It is however very good when small values are common and it reduces complexity since it prevents overflow. It also guarantees to only be one byte when used as a unique identification token for sets of items fewer than 129. This means that smaller numbers take less space than larger ones.

3.3.5 Encoding Paths

A path defines a location in a model schema that contains some variable. These paths could be arbitrarily long since the data structures can be arbitrarily nested in the data tree. A path could be encoded as a string "`foo.bar.baz`" but this would be inefficient with respect to the encoding size. Because of this, the data members of a type structure should be ordered, (e.g. by their insertion order,) so that they could be accessed as a number. Then a path might become "`3.1.2`" rather than "`foo.bar.baz`". This example would be read as "the second member of the first member of the third member" and could be encoded pretty compactly. The ideal would be one variable length integer per node in the path, and each one of those would typically be a single byte in length, but one complication is that there needs to be an indicator for when the reader has reached the end of the path. The most simple method for doing so, without needing to add a byte-sized terminator character at the end of the path with some reserved bit-pattern, would be to dedicate one bit per variable length integer for indicating whether it represents the very final segment in the path or not (or alternatively, the very final byte in the entire path). This avoids having to insert more than one variable sized integer per segment in the path.

Variable length integers in the LEB128 format guarantees to only take up one byte when used as a unique identification token for sets of 128 items or less but they still can support larger sets than that. This makes variable size integers a good fit for this purpose since the number of members in a structure usually is less than that, but the format also allows a larger quantity if so would be necessary.

Reserving an extra bit per segment as described earlier would limit this to only fit 64 unique numbers in the first byte, rather than 128. Still, that is typically more than the number of members in a structure and the first $64 \times 128 = 8192$ members are still representable with at most 2 bytes each.

There are however ways to improve the encoding size slightly without too much modification. One observation is that among the two reserved bits in each byte, (the bit that marks whether or not this byte is the final byte in the variable size integer and the bit that marks whether or not this is the final segment in the path,) there are only three combinations that are used out of the four possible: either the byte is or is not the final in the variable size integer and if it is the final one it either is or is not the final integer in the path. There is no case where a byte that is not the final byte in an integer would be marked as the final byte of the path. So, recognizing this redundancy, it is possible to reduce the bit-pattern for one of these scenarios into only a single bit. Which scenario to minimize depends on what scenario is deemed the most beneficial to reduce, but in either case it is a very small optimization and might add more complexity that what it is worth.

Regardless of that issue, one consequence of using variable length integers to specify a member in the theoretical (yet unlikely) case where the data type has hundreds or thousands of members is that the most commonly accessed members can strategically be placed earlier in the ordering in order to get the most compact representation.

3.4 Read and Write Coordination

The proposed architecture would allow for multiple simultaneous writers, without locking write access, and handling conflicts by rejecting the conflicting command. However, events within a scope, (i.e. between a push and a pop of a scope,) must be seen as a transactional operation. It is necessary to prevent race conditions such as the case where someone sends a command message, but the scope is changed by a different user just before the message takes effect, resulting in it happening in the wrong scope. This is not implemented in the prototype.

Reads can be fully asynchronous.

3.5 Running the Simulation

The ability to run the simulation from the server is not included in the prototype implementation, but will still be accounted for in the design.

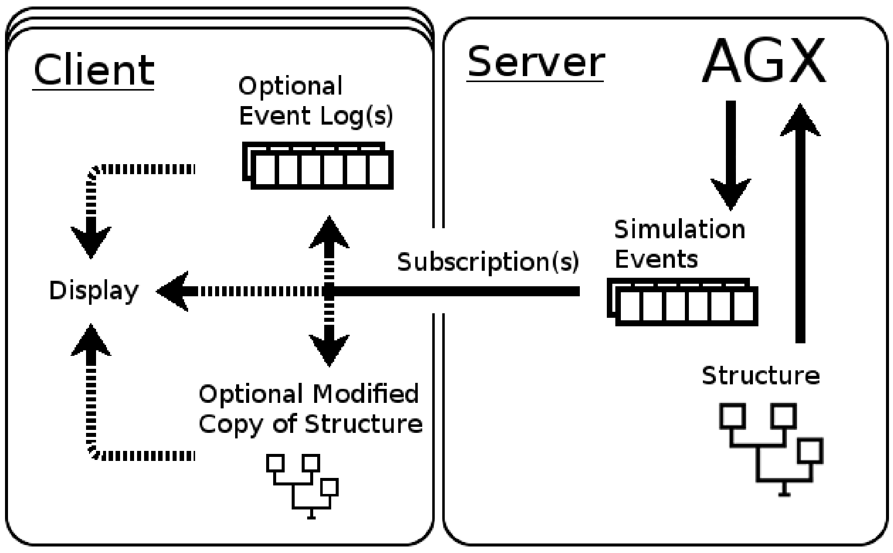


Figure 2: A visualization of the dataflow when a server is running a simulation. The arrows show what affects what. The model is feed into AGX Dynamics. The physics engine runs the simulation and outputs events when values are changed. These change notifications are filtered based on the client's preferences and the data is sent to the clients that subscribe to it. The client may or may not record the event log for future reference, otherwise the older events can be discarded. They may or may not also keep an updated copy of their structure. The processed information from the subscriptions then informs what information should be relayed to the user.

The simulation could run locally on the client's machine or on any other node of the network. Having a dedicated server that runs simulations might be good if several clients want to witness the same simulation play out in real-time.

If the simulation is performed on a server, the data needs to be streamed to the tool. These changes are not permanent modifications of the models, so they are not part of the event log, but they may still be represented as events. Putting different kinds of data on different channels would be preferable since it would allow specific tools to only subscribe to the channels that interest it. For example, a rigid body simulation tool would probably want to know the transformations for every object and that might be enough for some simple visualization tools. However, it is also possible that the collision points and velocities are interesting to some visualization tools so having the choice of which variables to observe allows for more flexible customization and avoids unnecessary network load. Since most values are updated every frame it is important to not transmit data in vain. It is conceivable to imagine a system where the amount of data transmitted from the server during real-time streaming of the physics simulation is minimized even further by allowing the client to specify update frequencies for each channel individually and also have the ability to request that for some channels changes are only reported when a value changes with a significant enough amount.

In Figure 2 is a visualization of what happens when a server is running a simulation. If the client chooses to save the simulation log they can later go back and step through the simulation, inspecting what happened, without involving the server.

3.6 Version Handling

A simple version handling system was implemented in the prototype, effectively implementing causal ordering between the models' different versions.

As an example of a scenario where version handling is necessary, consider a model `Foo` that contains a variable of type `Bar`. In the scenario, there are two active user applications involved, one that is editing `Foo` and one that is editing `Bar`. If the user of one client application changed the model `Bar` by deleting one of its variables, then this change can be synchronized with the server, but the other client that is working on `Foo` does not immediately know about the change and will at first still believe that `Bar` contains the deleted variable.

If that client then tries to set the value of the deleted member of `Bar` in its local instance of `Bar` within `Foo`, then that might seem to work locally, but if the operations from both of the clients are allowed by the server to become part of the event streams, then any future client that tries to reconstruct `Foo` from its event stream will have to know that an older version of `Bar` should be used, otherwise it will not arrive at the same result.

The easiest method for solving this is to simply include a version number for each dependency when they are added, and there could be some method for `Foo` to update any dependency at any time the client chooses. By including version numbers in the event stream when referring to other models it is possible to reconstruct a model tree from an event stream by keeping the appropriate versions of referenced models in memory. There are some problems with this, however.

If `Foo` version 1 depends on `Bar` version 1 which in turn depends on `Baz` version 1 and `Foo` version 2 then gets an additional dependency `Fubar` version 1 that in turn depend on `Baz` version 2, then `Foo` must keep both version 1 and 2 of `Baz` in memory at the same time. An illustration of this scenario is found in Figure 3. This might be an okay approach, but it does add complexity and support for having multiple versions in memory at the same time is not part of the prototype implementation, but it is discussed further in Section 5.1.

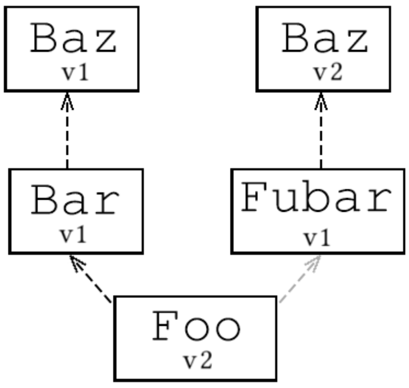


Figure 3: A visualization of an example dependency hierarchy where two versions of the same model appears. The arrow in a lighter shade of gray was added in version 2 of `Foo` which gave cause of the potential conflict. This is only a problem if two versions of a model is not allowed in the same client at the same time.

In the prototype, every client or server always only has one version of a dependency in memory at any given time. It is then necessary to find a solution that can work with that limitation.

Since all decisions are made on a single server, that server is the reference for what the latest versions are. Clients are not automatically notified about changes to a model unless they are explicitly subscribed to that model's event stream. If they are, then they will eventually receive all the changes to that model. The client and the server has one type table per stream that keeps track of which models the stream depends on and which version those models are presumed to be.

Recall that one property of event sourcing is that the model can be reconstructed perfectly from only knowing the event stream(s). If the stream has one dependency, but the client then updates the version of that dependency locally, or if the server has a newer version, then subsequent changes to the model could be ambiguous. If the client makes a request to change any part of the model that would produce different results depending on the version of some dependency, then the server checks if the latest version of that dependency is in the type table for the stream. If the version in the table is older, a message urging an update is inserted into the model's event stream before the requested change can be considered. This is, for now, done with another "type-table" message with the new version number. Doing that overrides any older version present in the table.

When a newer version is needed in a client but the client already has an older version of the dependency in memory, it only requests the changes to the event log since the version that it already has knowledge of from the server, rather than receiving the entire log and get redundant information.

The prototype does not handle the scenario where a stream requests an older version of a model than the one currently in memory and thus the prototype does not handle the modified version of the problematic scenario mentioned earlier, where `Fubar` had an older version of `Baz` than the version that `Bar` had, rather than having a newer one. That is an important functionality to have in a production-ready version of the system, since reconstructing any structure from its event stream might involve temporarily using older versions of models early on in the stream. Once again, alternative solutions are discussed in Section 5.1.

4 Results

By implementing and using the prototype, it was concluded that the format proposed could be used to serialize and deserialize the Brick runtime AST, per the definition of serialization and deserialization described in Section 3.3. This conclusion was reached by performing the sequence encode-decode-encode and checking that the two encoded streams were identical. Correctness tests were also performed by connecting several clients that subscribed to the same model and see that changes suggested by one client propagated to the rest. Tests were also performed to see that the simple dependency handling system worked for cases that the prototype was built to handle.

It is difficult to test the performance of the event streaming architecture since it would greatly depend on how the interface is used in a real world context. Nonetheless, a few examples were created in order to test the prototype implementation.

4.1 Field Size Comparison

In Figure 4 is a breakdown of the event stream for an example model called "`Examples.Vehicle.DriveTrain.DriveTrainWithDifferential`". It was created by taking all event messages needed to encode the entire model and taking the sum of all fields on all the event messages divided into categories. This gives a quick overview of what kind of data the bulk of the event stream consists of.

The summary consists of event messages from both the stream to the model itself, but also all streams of the models that the first model depends on. In other words, it includes all data necessary to create the model from scratch, without prior knowledge about any data types other than the primary data types such as scalar types and the generic list type.

In real-world usage, it would be more common for the client to already have local knowledge about a base library of core models and so only high-level models would have to be transferred.

In Figure 4, the slices labeled "opcodes," "locations," "names," "values" and "type paths" represents the size of the fields with those same names in the event messages, (but in singular,) as they are described in Section 3.3.2. The slice labeled "other" represents the rest of the data in the the event messages. This includes the type and constructor IDs as well as the revision of type-entries.

Two versions of the breakdown was made. One where location paths was encoded using the path encoding described in Section 3.3.5 and one using ASCII strings as path encoding. The latter has a larger size of the locations, but the values increase slightly in size as well. That is because the value field can be a reference to another variable, which is then represented by a path to that model.

One thing that sticks out is that the type paths take up a proportionally large portion of the total size. The slice labeled "type paths" only represents the size of the field "type path" in the event message type "type-entry". The reason why it represents such a large portion of the

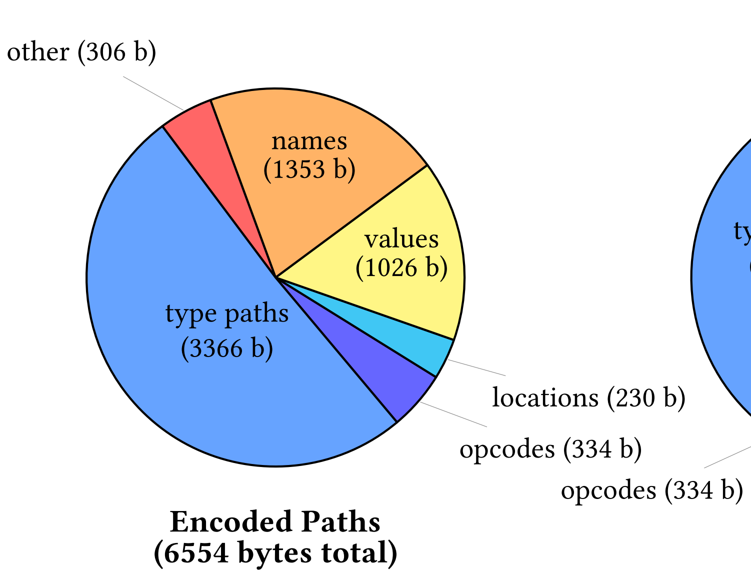


Figure 4: Analysis of which kinds of data the set of streams needed to construct the model "`Examples.Vehicle.DriveTrain.DriveTrainWithDifferential`" from scratch consists of, shown with and without path encoding.

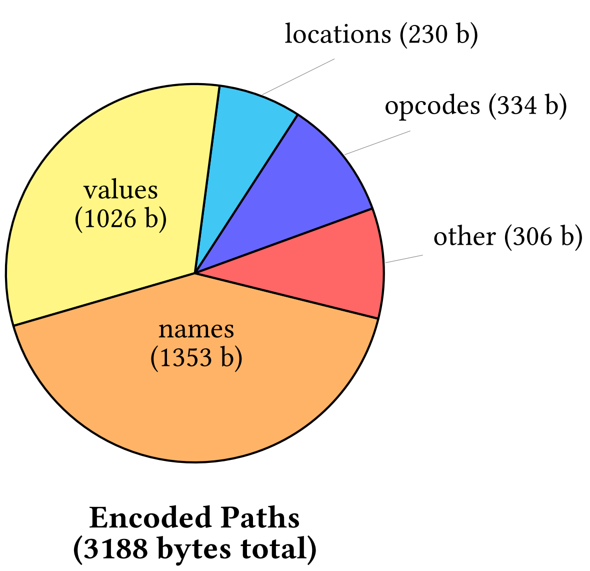
total size is that it is encoded as an ASCII string, rather than using the same encoding as the variable paths.

The problem of encoding type paths is more difficult than encoding variable paths since variables have well defined positions and orders within the model, clearly defined by the event stream. Since the types do not have a clear order and is not part of any event stream it would be much trickier to encode these. But since they represent such a large portion of the total size, trying to solve that issue might be a good idea.

Perhaps it can be solved by introducing a global index for each model that is stored in a table that is persistently managed by the server. Or perhaps there could be have a special kind of model that stores information about namespaces.

Note that the names of models in the example files can be pretty long in some cases. For example, "`Examples.Vehicle.DriveTrain.DriveTrainWithDifferential`" is the path of the top level model, and that string is 54 characters long. It is worth stressing that the set of models in the example, might not be representative of how an average model might look. If models had fewer unique dependencies, then fields of this type would be fewer and might have a smaller impact. Disregarding the type paths completely, Figure 4 would instead look like Figure 5 which shows that names and values are the most dominant field types, as well as locations if they are not encoded well.

Another important fact is that there would be more data transferred other than just the event streams. For example, when an event stream is read and a new dependency is introduced, if the client does not have a local copy of that model, or if the local copy is a too old version, then the client must make a request for obtaining the event stream for that model. The most notable field in that request message is, once again, the type path. Any other fields in the data request can be omitted from this analysis. The total size of all unique type paths in the dependency graph for "`Examples.Vehicle.DriveTrain.DriveTrainWithDifferential`" is 2233 bytes, but of course, there might be smarter ways to address types in this context as well, which would reduce this size. Figure 6 shows that type paths in the event stream and in model



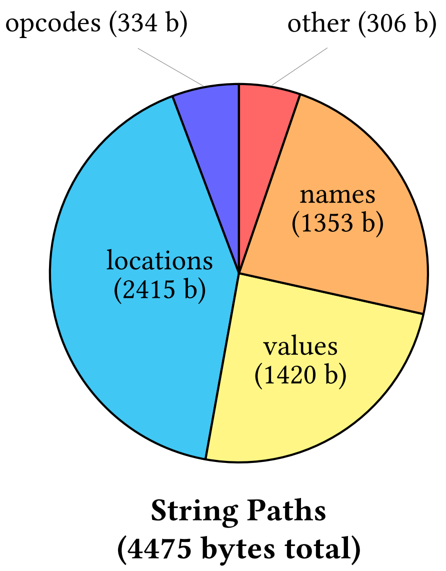


Figure 5: Analysis of which kinds of data the set of streams needed to construct the model "`Examples.Vehicle.DriveTrain.DriveTrainWithDifferential`" from scratch consists of, excluding the encoding of the type paths in type-entry event messages. The result is shown with and without path encoding.

requests together make up about 60% of the total data. There are many potential ways of reducing the size of this data.

Another note about the distributions between types of fields is that string fields become much rarer when instantiating and setting values in a structure with an already defined schema, rather than adding new variables. In Figure 7 (a) is a comparison of the data fields for a stream that defines a model with 100 new integer variables with set default values of one byte each. In contrast, Figure 7 (b) creates a single variable which is an instance of the model defined in (a) and then overrides all 100 integer values. In the latter case, the names do not need to be re-defined. It is also easy to imagine a scenario where the values were much larger than one byte each. If the values had instead been double-precision floating point numbers the value fields would have been 8 times as large and would have taken up 77.5% of the stream, all else being equal. That means that the data overhead that describes structure, rather than the actual raw data, would be significantly smaller, proportionally, than in the case of (a). The stream for (b) is dependent upon the stream (a), so (a) must be fetched before (b) can be constructed from its event stream. But this only needs to be done once per client. If several models depend on the same version of model (a) or older, no new information about (a) needs to be fetched from the server when those new models are reconstructed from its event stream.

4.2 Impact of Scopes

In Figure 8 is a comparison from three different sets of streams. The first, Figure 8 (a), being "`Examples.Vehicle.DriveTrain.DriveTrainWithDifferential`" showed very little difference between using and not using scopes. This is probably due to that model graph being divided into 50 models, each already operating within its own model scope. Adding scope-management within models did not make much of a difference, since most of the models probably only set values that are not nested.

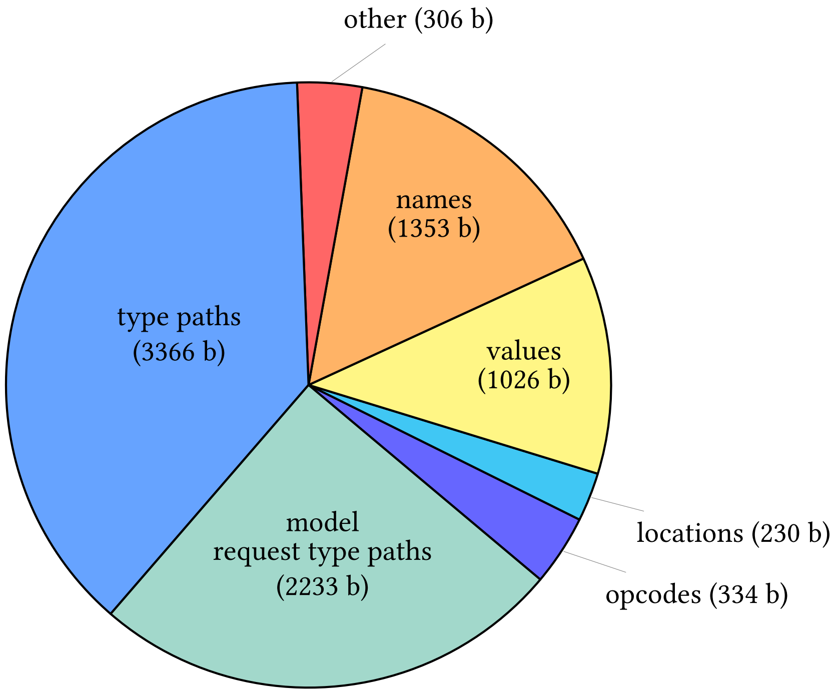


Figure 6: Analysis of which kinds of data the set of streams needed to construct the model "`Examples.Vehicle.DriveTrain.DriveTrainWithDifferential`" from scratch consists of, as well as the total size of all the type path strings needed for all the event stream requests.

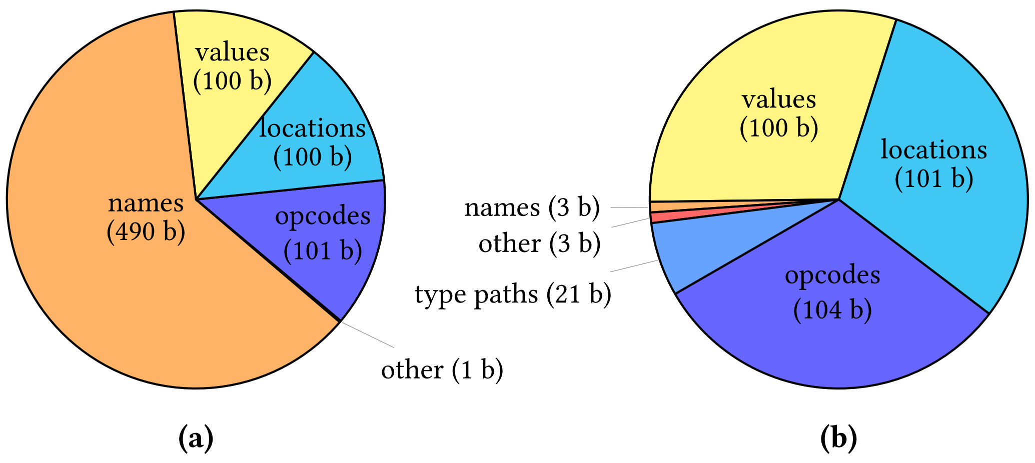


Figure 7: Analysis of which kinds of data the streams for two models consist of. Model (a) is a model with 100 integer variables with their values set, and (b) is a model that contains a single variable which is an instance of (a) but with all 100 integers overridden.

The second, Figure 8 (b), is the models used in Figure 7. It showed a slightly larger relative difference, since the two models had separate areas of responsibilities. One defined the structure and default values and one initialized the instance with particular values. That means that the value initialization model set the values within the scope of the other model. If such a separation between structure schemata and the instance with custom values is a common use case, then (b) might be a more realistic scenario. The difference might even be larger, if the schema was larger or more nested. To test an extreme case, a third case, Figure 8 (c), was also added. It has by far the largest contrast. It was created by placing a mode within a node within a node (etc.) 100 nodes of depth and then adding 100 variables within the most nested node. This is not the most realistic scenario, but it shows that scopes to have a large impact when pushed.

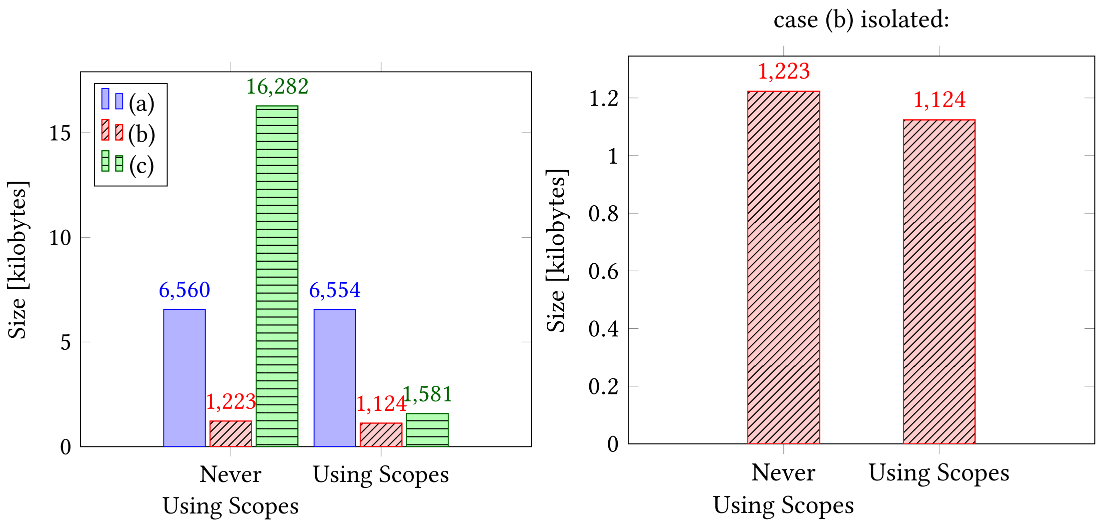


Figure 8: Analysis of the effects from using scopes.

4.3 Comparing the .AGX file format

Algoryx's ".agx" file format encodes the "`DriveTrainWithDifferentials`" model used in most of the examples of this chapter is 48 kB. The proposed event-based format encodes the same model graph with 6554 bytes. The comparison is not a completely fair one, since the two formats have different purposes. The ".agx" format is a serialization format, rather than a modeling format. It also stores all the values for every instance of a model even the that are assigned their default values. Another difference is that there is information stored in the event logs that is not included in an ".agx" file. The event logs encode the names and types of all variables and set both the default values and overridden values.

An event stream could be made to encode only the information present in an ".agx" file, if the other information existed in separate models that that stream depended on. The fact that it is easy to break an event log file into a base file and a patch file, where the latter only contains the updates from one version to another is also a benefit of the new format.

In a scenario where the model changes over time, these changes as well as the order they were made in would be traceable by reading the event log. In this comparison though, the model is just build up from scratch without any meaningful changes over time encoded. But it is still worth noting that the new format is smaller than the “.agx” format yet, which might be a testimony to its success at remaining compact.

5 Discussion

This chapter contains discussions about problems and limitations in the design and brings up some potential other approaches that could have been used instead. These approaches might in some cases trade one limitation for another and are not necessarily strictly better than the approach used. In other cases there are some obvious flaws in the prototype implementation that has to be fixed if a production ready system is desired.

5.1 Resolving Conflicting Dependencies

This section revolves around the problem where something is depending on conflicting versions of the same dependency in different parts of the dependency graph.

Such version conflicts can easily arise when several models are involved in the same dependency graph and can be updated independently. As mentioned in Section 3.6, there are problems with the approach to only have a single version of each model in memory at any given time, as is the case in the prototype. There are several ways to resolve this problem.

The first is to have several versions of a model in memory, but only as many as is required at the time. When a newer version that is not already in memory is required, the closest older version that is in memory is cloned. That clone is then updated to the new version by applying the relevant parts of the event stream.

If `vX` is the new version number and `instances` is the set of all instances of the target model that are stored locally, then this pseudocode describes the algorithm:

```
if(model with version vX is not in instances) {
    older ← instances.GetSubsetWhere(version < vX);
    if(older is empty) {
        modelX ← new Model(empty);
    } else {
        tmp ← older.GetInstanceWithHighestVersion();
        modelX ← new Model(clone of tmp);
    }
    modelX.UpdateTo(vX);
    instances.add(modelX);
}
return instances.GetInstanceWithVersion(vX);
```

Old versions of a model can be deleted when no other models depend on it.

This approach introduces some redundant calculations, but at least the client can reuse the event stream that it has already received, rather than fetching it again from the server.

Another idea for handling dependency conflicts is to keep every version of the model in memory all the time. This avoids having to reconstruct old versions that had already been constructed once, but it requires more memory. If every time a new event message is added counts as a separate version, this approach is almost certainly not feasible. In Section 5.2 are some ideas for other methods of assigning version numbers. This reduces the number of versions that needs to be taken into consideration by the dependency graph. Fewer versions at play means fewer versions that could need to be in memory at the same time which could make this approach more viable.

Yet another approach is to store all versions, but take advantage of the fact that most of the structure probably stays the same through several versions of the model. This can be done by using persistent data structures. Persistent data structures always preserve previous versions when modified. That way access can still be granted to those previous versions [5]. Persistent data structures can for example be implemented efficiently using hash array mapped tries (HAMT) which is done to implement persistent collections for the standard libraries of some programming languages such as Clojure and Scala [16].

5.2 Reducing Redundancy and Refining Version Handling

The event log could easily include redundant event messages, especially when the user is tweaking a value, changing it multiple times to different values in quick succession. It might be able to optimize it away by not sending every change to the server as soon as it happens, but instead wait until it becomes necessary.

This is similar to the approach chosen by one of the papers mentioned as related work for this paper [3]. In that architecture, the user would edit a "ghost" of the object until happy with the change at which point the change is committed [3]. That could work here as well, and could probably be used in some capacity in addition to any other suggestion brought forward in this section, but relying too heavily on this mechanic would loose some of the benefits with having multiple clients be able to see the changes in real time or close to real time.

Another problem with that solution emerges when the user wants to run the simulation on the server. If the user has unpublished local changes at that point, then the server would not know about them. Thus, that data would not be fed into the physics engine. To prevent that, sending the changes to the server becomes necessary. It is likely that the user wants to run the simulation ever so often while tweaking the values which means that the user is forced to push the changes to the event stream before every test run. Each intermediate value is then reflected in the version history, making the event history contain redundancies and subsequently, the process of recreating the model from the history has to perform redundant steps. This also increases the network load when sending models to other clients.

Another problem is that the version handling in the prototype design is dependent on the exact number of events in the event log at any given time. Since the log is append-only, this gives a unique ID-number for every version, but it also means that every message introduces a new version. This is a problem for several reasons. First off, it prevents the server from removing the redundant messages from the event stream since the exact number of messages is important. Of course, just changing the version history would be bad anyway, since it would remove information. It would ruin the undo-history and all the earlier versions would no longer be reconstructable, which is a requirement for resolving dependencies. Another problem with having every message produce a new version is that it amplifies the dependency conflict problem explained in Section 5.1 since there are more potential versions in the dependency graph that can conflict with each others.

There are several potential solutions that can solve all of these problems. The potential solution that appears strongest is to keep a faithful version of the event history while the user is editing the model, stored as a separate "unpublished" event log. In this log, the server can allow the user to undo and redo as much as they would like. This version could also be used when the user wants to run the simulation, but it would not be exposed to the dependency graph until the user is happy with the changes and decides to "publish" it to the real event log. At that point, instead of transferring the actual version history to the published event log, a new more compact version of the history from the last published version to the new one is generated and that more compact set of events is added to the event log. Not until then can the new model version be used as a dependency in other published models. The development event stream and the published event stream are basically two separate streams in this scenario. This circumvents the reconstructability problem by only promising that the published versions can be reconstructed, but still allows for undo-history in the development stream.

Other alternatives also exist such as having separate major versions where each can have their own minor version and each new major version changes the history that came before it into the most compact possible form when it is created. A third alternative solution could be to keep the version history as it is on the server, but create optimized versions of that history whenever a specific version of the model is requested by a client. The most obvious problem with both of those approaches is that a client that requests one version cannot infer older versions by reading parts of that stream. These two approaches also does not address the problem with storage redundancies on the server.

5.3 Demands Placed on the Physics Engine

As can be seen from Figure 2, the architecture from running the simulation requires the physics engine to emit events. This is one of the largest demands that this approach puts on the physics engine itself.

If running the simulation is ignored, as is the case for the prototype, than the purpose of all the event streams is to provide a graph that describes the initial state of the world. This data can then be converted into a series of commands that puts the physics engine into the correct initial state. This can be done externally without changing anything about the physics engine itself.

However, if the state of the world when running the simulation should be transmitted back to the client using the same event stream format, then there has to be some method for the physics engine to spit out data to the server that it can use to construct new event messages. This might not be a huge problem if the physics engine is developed alongside the event interface, or if there is some method to listen to changes and then manually convert them into events, but if the physics engine is an externally developed software and lacks functionality, then the developers of the event interface is limited to the functionality present in the physics engine.

5.4 Function, Constructor or Macro Arguments

The ability to group several operations together into a single operation and then evoking that operation instead of each operation separately could decrease network load as well. Since this thesis aims to create an interface that is general, it might be a good idea to allow the creation of such operation bundles by the user application. One specialization of such a functionality could be specialized constructors for a model. The constructors could take data as arguments to customize the outcome. Whether or not behavior such as flow control could be injected into such a constructor or whether it would only be a predetermined series of event messages with customized values remains to be decided. Exploring the possibilities of such functionality is out of scope for this thesis, but the idea might have potential and the ramifications of using it might be significant.

5.5 The Choice of Event Set

The set of operations used in this architecture for commands, as well as making up the log of events, shares many similarities to events in event sourcing. However, event sourcing is based on domain-driven design. To follow the principles of domain-driven design, the events should be modeled after the behavior of the domain. The current set of operations might be seen as too low-level, and too CRUD-like to be regarded as an ideal set for use in event sourcing. It is unknown if this choice of event set is an anti-pattern that may have any negative effect on the future extensibility or maintainability, or if it is a decent approach for the data-driven system it is a part of. It can be argued that the domain for the event sourcing is "data modeling" rather than the high level domains of the end users. With that in mind, perhaps the interface is at a suitable level of abstraction for that purpose. Further research might be needed to draw a definitive conclusion in that regard.

Regarding the naming of the event messages, it might have been a good idea to use more names that better reflect the fact that the operations are representations of completed events. This could have been realized as names such as "variable-created", "value-updated" and "dependency-added" instead of "variable", "data" and "type-entry". Furthermore, when used as requests they should be in imperative, i.e. "create-variable", "update-value" and "add-dependency". This could result in a more data-driven mental model as well as a more semantically correct description of what the messages represent.

Then there is the question about whether or not using small, discrete operation-like events is a good idea to begin with. An alternative could be to make each event message a partial graph that contains only the differences from the main model graph. That way, each modification can contain arbitrarily large changes, but still be able to specify only the subset of the data it is changing. Encoding what information to merge might take up as much space as the event information does or more, but it is possible that the mental model becomes more manageable or that some optimizations can be made. It is also possible that more optimizations can be made when each message is larger. Using this graph-based approach, fewer event messages would be necessary, and if a version handling system such as one of those mentioned in Section 5.2 are used, meaning that there is a need for calculating compact versions of the stream, then that might even allow for each version to be merged into a single message.

5.6 Evaluation of Proposed Architecture

It was concluded that the format proposed could be used to serialize and deserialize the Brick runtime AST. That means that the client gets full access to the Brick format. This tree format is very flexible and can define components of the physics engine which means that as long as the structures in the physics engine has Brick equivalents, it is possible to create tools that interacts with them. The drivetrain models were loaded into the client without being provided an explicit interface for working with drivetrain components.

A single server is responsible for deciding the global ordering of events. This makes sure that consistency is obtained even if multiple simultaneous writers are allowed. Thus, collaborative tools would be possible. A more robust conflict resolution system might be desirable for that use case, but that is out of scope for this report. Multiple readers of models is trivial since all model data in the committed model versions are immutable.

This also provides scalability by adding servers and multicasting new information among them.

There are aspects that are difficult to analyze the proposed architecture from within the scope of this project. Maintainability, fault tolerance and extensibility are some such aspects. How easy it is to develop and set up tools that communicate through this interface is also difficult to test without having more resources available. Even the aspects that were analyzed, would have produced more conclusive results if they were backed up by tests from a larger library of relevant models, something which was not available.

5.7 Peer to Peer

This thesis has mostly assumed a central synchronization server for the models through which all changes are validated and ordered to provide a consistent global ordering. This is a reasonable assumption for most use cases, but peer-to-peer synchronization might also be feasible.

There are some problems with trying to achieve this. First and foremost, the system is supposed to work on a set of models where the sum of the size of all models is large. When using a central server, the server is assumed to be large enough to hold every model. Write access to the system has a single point of failure, but regular backups can be made to prevent permanent data loss and multiple read-servers can prevent read-access downtime.

Creating a P2P solution has different problems that depend on which assumptions can be made.

If none of the peers in the peer-to-peer network are assumed to have a large enough storage capacity to reasonably be expected to be able to store every model, there is no one place where all models exist. Then it is harder to make complete backups. But a larger problem is consensus; if there is no central server that decides the global ordering, then how can consensus be reached about which changes are to be accepted by the system and in which order they should occur?

Another concerns is whether peers be assumed to run according to the specification; that none of them are corrupt or deliberately sabotaging.

Yet another concern is whether every node in the P2P network can be assumed to constantly have a near 100% uptime. If so, it would not matter if a model was only replicated in a few nodes of the network, since those nodes would always be able to pass on the model to any other node that requests it. It would also make consensus simpler, since every node could reasonably wait for a response from every other node.

There is no one solution that fits all needs.

6 Conclusions

This thesis paper has shown how an event-driven architecture in which physics simulation models can be serialized into a stream of events, could be used to coordinate the authoring of such models using editing tools provided as microservices.

The paper also highlight some difficulties that needs to be overcome when implementing such a system. Specifically dependency management and in particular handling versioning of dependencies.

Experiments showed that explicit strings made up the majority of the event message size. Experiments were also conducted to measure the effect of trying to limit the length of paths using scopes. The conclusion was that it had a greater effect on models that made changes to scopes that were deeply nested in the structure than to those that did not. This mostly happened in cases where schemata containing the structure was placed in models separate from the model that was setting the values for instances of that structure. This scoping system did however add some complexity to the system, introducing the requirement of treating some groups of events as a single transaction, but might also open up the possibility of creating complex function-like operations in the future.

The proposed architecture did not require locking and had really good horizontal scalability properties since all data is immutable, but only eventual consistency could be guaranteed on low-latency systems since updates would not be able to propagate instantly.

In the proposed architecture, models could depend on other models in a causal dependency graph, explicitly specified by the models themselves. A proposal where the event log was separated into a committed version and a work-in-progress version was suggested as a solution to avoid unreasonably long editing histories and too many versions to coordinate.

It was difficult to draw conclusions from the quantitative testing without having a good idea of how the interface would be used in practice. A large library of relevant models could have helped this analysis. Similarly, it is difficult to measure how successful the proposed architecture is without having an alternative implementations with corresponding data models to compare it to.

6.1 Future Work

Many potential topics for future work are brought up in Chapter 5. One examples of an interesting topic to explore is maintainability and extensibility of such a system, e.g. by looking at the possibility of adding, removing or changing some event types or make updates in general without breaking existing tool integration and models.

Another topic is to actually implementing one of the more robust version handling systems and using that versioning system together with a tool that makes use of it, to see how well it holds up does in practice.

Testing what impact functions or constructors would have on the system would also be a next step. These might also be able to introduce more high level events on the event stream and allow a more domain-driven approach, but might also introduce new problems such as introducing having to handle different versions of a function as well.

Commands are in the current design fire-and-forget. It might be a good idea to add some error feedback mechanisms.

More rigorous testing in general would be good to make. Unfortunately, we did not have time to to a proper tool integration and most tests were performed from a simple terminal interface. Proper end-to-end testing with multiple users would be valuable.

References

[1] Dennis G Brown, Simon J Julier, Yohan Baillot, Mark A Livingston, and Lawrence J Rosenblum. Event-based data distribution for mobile augmented reality and virtual environments. *Presence: Teleoperators & Virtual Environments*, 13(2):211–221, 2004.

[2] Andrzej Debski, Bartlomiej Szczepanik, Maciej Malawski, Stefan Spahr, and Dirk Muthig. In search for a scalable & reactive architecture of a cloud application: Cqrs and event sourcing case study. *IEEE Software*, 2017.

[3] Caroline Desprat, Benoît Caudesaygues, Hervé Luga, and Jean-Pierre Jessel. Loosely coupled approach for web-based collaborative 3d design. *11th ACM International Conference on Distributed Event-Based Systems (DEBS 2017), 19 June 2017 - 23 June 2017 (Barcelona, Spain).*, 2018.

[4] Caroline Desprat, Jean-Pierre Jessel, and Hervé Luga. 3devent: A framework using event-sourcing approach for 3d web-based collaborative design in p2p. In *Proceedings of the 21st International Conference on Web3D Technology*, Web3D '16, page 73–76, New York, NY, USA, 2016. Association for Computing Machinery.

[5] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, 1986.

[6] Eric Evans. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[7] Event Store. Event store - event sourcing database, 2020. avaliable online: https://eventstore.com (accessed 12/05/2020).

[8] Martin Fowler. Event sourcing. *Online, Dec*, 2005. avaliable online: https://martinfowler.com/eaaDev/EventSourcing.html (accessed 17/03/2020).

[9] Free Standards Group. Dwarf debugging information format specification version 3.0, December 2005. avaliable online: http://dwarfstd.org/doc/Dwarf3.pdf (accessed 13/05/2020).

[10] Alexander Kiel. Datomic-a functional database. 2013.

[11] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert. Microservices. *IEEE Software*, 35(3):96–100, 2018.

[12] Scott Millett and Nick Tune. *Patterns, principles, and practices of domain-driven design.* John Wiley & Sons, 2015.

[13] NVIDIA. Nvidia omniverse™, 2019. avaliable online: https://developer.nvidia.com/nvidia-omniverse (accessed 12/05/2020).

[14] Vinicius Feitosa Pacheco. *Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices.* Packt Publishing Ltd, 2018.

[15] Tobias Stauber. Immutable databases at the example of datomic. 2016.

[16] Michael J Steindorfer and Jurgen J Vinju. Optimizing hash-array mapped tries for fast and lean immutable jvm collections. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 783–800, 2015.

UMEÅ UNIVERSITY