Simulation-Based Engineering Lab
University of Wisconsin-Madison

Technical Report TR–2017–06

Networking Architecture for Distributed Vehicle Simulation in the CAVE Project

Dylan Hatch

December 11, 2017

Abstract

This technical report outlines the second iteration of the design of the server and client communication interface used to facilitate the distribution of physical vehicle simulation in the CAVE project. A previous technical report [1] outlines an earlier version of the CAVE server and client communication interface.

Keywords: Autonomous Vehicles, Networking, UDP, TCP.

Contents

1 Introduction **3**

2 Server Design **3**

2.1 Network Communication 4
2.2 Message Handling 4
2.3 The World Object 4

3 Client Design **5**

4 Network Handler Design **5**

4.1 The Client Network Handler 6
4.2 The Server Network Handler 6
4.3 Message Parsing, Serialization, and Format 6

1 Introduction

The Connected Autonomous Vehicle Emulator (CAVE) project aims to create a framework for the simulation of autonomous vehicles in a virtual world that is distributed across a network of participating computers. This goal is achieved by having client simulations connect to a central CAVE Server, which facilitates the distribution of state information of each vehicle to all other clients by maintaining the state of the virtual world and handling clients' requests for updates. Network communication is done with the Boost.Asio library, using a combination of TCP and UDP. Serialization of vehicle state information is done using Google Protocol Buffers (protobuf). Figure 1 provides a visualization of the client and server networking in the context of the rest of the CAVE Project.

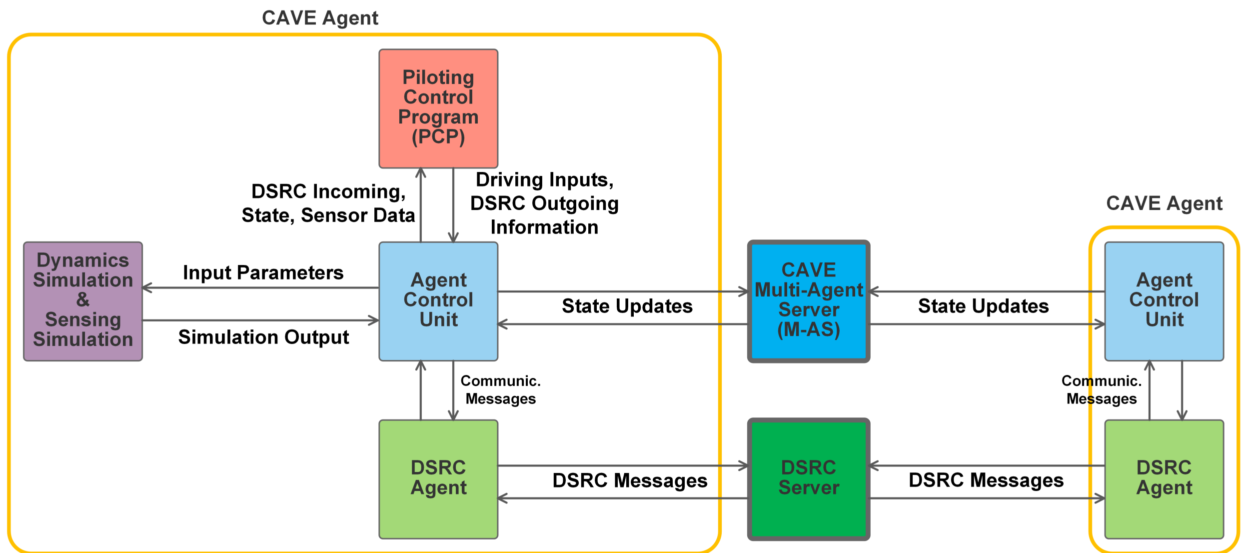

Figure 1: An overview of the network layout of the CAVE Project.

2 Server Design

The Cave server consists of three concurrent components: handling network communication, processing messages, and updating the virtual world state. Network communication is done using the ChNetworkHandler, which will be discussed in greater detail later, and uses two threads to enable the sending and receiving of messages at virtually any time. The server is designed for any number of threads to be assigned to the task of processing messages, giving it the ability to scale better as the number of cores it has access to increases. After the messages have been processed, the states in the world object are updated by one thread in order to maintain thread-safety.

2.1 Network Communication

The server uses a combination of TCP and UDP for network communication. For initial connection to new clients, the server accepts connections on a TCP socket. Once connected, the client is expected to send a connection request message. If the server cannot accept the connection, then a decline message is sent and the socket is closed. If the server is capable of accepting the connection, it will respond with a message indicating that it accepted the connection, and then send the client its connection number. After the client's connection number has been successfully established, the client and the server are free to continue to communicate over TCP. Information sent can include virtual world data, vehicle specifications, starting location, etc.

After all TCP communication has been concluded, the connection is closed and all further communication is done over UDP. The client is expected to use UDP for all state updates of vehicles and any other world objects. The server is capable of sending both individual state updates to any client, as well as state updates of the entire world or any subset of it.

2.2 Message Handling

The message handling threads are used to parse incoming messages, evaluate their sanity, and submit expressions to a queue that update the states of world elements. While parsing is done within the ChNetworkHandler class, it is still done by the handler threads in order to keep the network communication threads from doing things unrelated to communication. Once a message is parsed, the handler threads check that the message is of the same type as the preexisting world object, and that the update has a newer time-stamp. These checks are done using protobuf's introspective functions. These functions essentially allow the handler threads to handle any protobuf message without needing to know its type, so long as the message has certain fields, like time-stamp and identification number. Once these checks are done, the message is handed off to the world update thread.

2.3 The World Object

The World object stores all information about the server's connected clients and their associated world elements. Upon the initial client connection, the connection number of the new client is added to a set of numbers in the world. After the switch to UDP occurs, the connection number of incoming messages is compared with elements of this set. If an incoming message has a connection number in this set, the connection number is removed from the set and an insertion is made to a mapping of connection numbers to client UDP endpoint information. This process is what completes the "handshake" from TCP to UDP. At this point, the client is considered to be registered in the world. If further element state updates or new elements are sent, this registration is verified by checking for its entry in the endpoint map. If a message is received that doesn't have a corresponding entry in either the the connection number set or the endpoint map, then it is thrown out, thereby enforcing the TCP to UDP handshake process.

The current states of the actual world elements are stored in the form of a mapping from connection number and identification number to protobuf message. For this reason, any element is uniquely identified by the combination of its connection number and identification number. After a message has been verified to correspond to a valid connection number and endpoint, the user of the world object (the message handler) is given a handle to a endpoint-profile structure, which stores the receiving endpoint of a client, as well as the number of elements associated with that client, and iterators to the first and last elements of the client in the element map. The endpoint-profile enables the user to modify the elements within the world object associated with that profile's corresponding client. In order for the elements to be updated in a thread-safe manner, the message handler threads push a lambda expression to a queue, which upon being popped is executed by the world updating thread.

3 Client Design

The client contains at least two concurrent components: the main simulation loop and handling network communication. The main simulation loop is handled by Chrono, or any other physics simulation engine, and the network communication is once again handled by an instance of the ChNetworkHandler. Before the simulation loop starts, a TCP connection is set up with the server to register a connection number with the server and receive any simulation initialization information, such as weather or vehicle starting position. Then, in the simulation loop, all element updates are handed off to the network handler to be sent to the server over UDP. In other words, protobuf messages representing the current states of all elements belonging to the client are created and packed into a buffer to be sent to the server. At this stage, the loop should also check the network handler for any new messages from the server, and update the states of all external element representations.

4 Network Handler Design

Both the client and the server use a subclass of the ChNetworkHandler for network communication. This allows for a minimal amount of code to be written and tested. The base behavior of the ChNetworkHandler involves a sender thread and a listener thread, both of which use a mutex-protected UDP socket for communication. These threads can be created by a function call any time after the constructor has finished executing, and are destroyed by the ChNetworkHandler's destructor.

The listener thread obtains a lock on the mutex and receives UDP packets on the socket, and then pushes them to a queue of messages, ready for the user to receive. If there isn't anything to be received from the socket at the time of the receive call, then the call immediately returns and the lock is released. The receive call will be repeated in this fashion until a packet can be received on the socket.

The sender thread pops a message from an outgoing queue of messages, obtains a lock on the mutex, and then sends it on the socket. If there are no messages waiting to be sent,

then the thread will sleep on the pop call in the queue until a message is pushed.

4.1 The Client Network Handler

The client side uses ChClientHandler, which inherits from ChNetworkHandler. The constructor of the client handler adds the necessary TCP communication for initial connection to the server. This is done by accepting a function as an argument, which is called at the time of connection with the purpose of handling any TCP communication before the hand off to UDP.

Before being pushed to the outgoing queue, each message is serialized to a buffer, such that the sending thread only sends on the socket, and wastes no time serializing. The listening thread, however, takes time to parse each buffer it receives into a protobuf message before it pushes the message to the incoming queue. The client handler is currently designed to send and receive messages from the server, but can be easily modified to allow for peer to peer communication.

4.2 The Server Network Handler

The server uses the ChServerHandler, which also inherits from ChNetworkHandler. The ChServerHandler constructor instantiates a thread designed to handle the TCP connection to all new clients. This "acceptor" thread listens on an accepting TCP socket until it receives a new TCP connection to a potential client, at which point a socket to handle the new client is created. This thread then uses the new socket to evaluate a request to connect. If coherent responses are received, and if the server is capable of adding a new connection, the client receives a connection number, which is added to the world object. At this point, additional TCP communication necessary for the initialization of the client simulation can take place. Once all TCP communication has concluded, the handling socket is closed and the thread resumes listening on the accepting socket for the next incoming TCP connection.

In order to minimize packet reception and sending time, messages are serialized to buffers before being pushed to the outgoing queue, and are not parsed until popped from the incoming queue. This means the sending thread is only used to send messages, and sleeps on the outgoing queue pop call if there are no messages waiting to be sent, and the receiving thread is only used to receive incoming messages, and spins on a non-blocking receive call on the socket when there are no messages to be received. The server handler is currently designed to be able to send to and receive from any UDP endpoint, with the evaluation of the legitimacy of all messages left to the handling threads once their parsing has been done successfully.

4.3 Message Parsing, Serialization, and Format

The use of protobuf allows for a robust storage system, in which instances of the abstract Message class can be stored in a single data structure as representations of the states of

any type of world element. This is a much-preferred alternative to using separate data structures to store different types of world element because it allows for the removal or addition of element types to the CAVE Project infrastructure with absolutely no change to the storage system of the world object.

The actual structure of a CAVE protobuf message is a series of fields, many of which are meta data fields that all CAVE message types will have in common. These common fields include the connection number of the sending client, the local identification number, and the time-stamp fields that describe the sending time of the message. These fields are accessed by the server to ensure messages are valid when they become updates in the World object. The remaining non-meta data fields of a message are unique to that message type, and should describe the position and orientation information of a specific type of world element. More specifically, the message format for a vehicle contains information describing the position and orientation of the chassis, and each of its wheels. When other message types are added, they will contain similar fields.

In order to be sent, these messages are serialized to a buffer using the protobuf library. The first byte of the buffer is used to describe the type of the message so that the receiving end can successfully parse it. Messages can be sent as individual UDP packets, or in a packet with other messages as a repeated field. This allows individual messages or groups of any size of messages to be sent with ease by the client and the server.

References

[1] D. Hatch. Networking architecture for the distributed simulation of manned and autonomous vehicles. Technical Report TR-2016-06: `http://sbel.wisc.edu/documents/TR-2016-06.pdf`, Simulation-Based Engineering Laboratory, University of Wisconsin-Madison, 2016.