Simulation-Based Engineering Lab
University of Wisconsin-Madison

Technical Report TR-2016-06

Networking Architecture for the Distributed Simulation of Manned and Autonomous Vehicles

Dylan Hatch

September 11, 2016

Abstract

This technical report outlines the design and implementation of the server and client interface that will be used to facilitate communication for the distributed simulation of vehicles among many hosts connected over a network. The client will be compatible with the `Chrono::Vehicle` module for vehicle and physics simulation, and is designed to be robust enough to work with other physics engines as well.

Keywords: Autonomous Vehicles, Networking

Contents

1 Introduction

The Chrono Autonomous Vehicle Project aims to provide users with a high-fidelity, many vehicle simulation environment for the testing of autonomous vehicle software. This goal is achieved through the connection of multiple client hosts, each providing position and orientation information of a manned or autonomous vehicle, pedestrian, or other world object. These clients will be connected via a server, which will facilitate the communication of necessary information between clients. The server is intended to do as little work as possible, providing only communication and leaving all simulation work to be done by the clients.

2 Server Design

The server has a typical concurrent design. The server has a main listening thread that listens on a port for incoming connections from new clients and accepts them. After accepting a connection with a client, the listening thread then creates a new thread dedicated to handling the connection. The connection-handling thread waits for incoming messages from the client, and responds by sending the necessary information about other clients' vehicles and world objects. This process continues until the connection is closed by either the server or the client, in which case the thread will end. There is also a heartbeat thread, used to produce a constant heartbeat that is broadcast to all clients.

World information is maintained and stored using an instance of a `World` class. This class stores information about world objects by organizing them into smaller sections of the world. Each section contains at least one tree-map, which is used to map a unique identification number to each world object. In order to make write operations to the `World` thread-safe, connection-handling threads push a lambda expression of each write operation to a mutex-protected queue, while another dedicated thread continuously pulls from the queue and executes each lambda expression in the order in which they were pushed.

```
while(true) {
    if (!worldQueue.empty()){
        worldQueue.front()(); // Executes the next expression
        worldQueue.pop();
    }
}
```

This allows for all write operations to be performed on one thread, ensuring multiple write operations cannot occur simultaneously. When the message is added to a map within the `World`, it is done as a shared pointer in order to reduce the copying time, and to allow for future abstraction of the messages that are stored. When reading from the `World`, the connection handling thread receives a copy of the map of the requested section. This ensures the map the thread has access to will not be written to during the sending process.

3 Client Design

All communication with the server in any client program is conducted with one or more instance of the `ChClient` class. Each instance of the `ChClient` is designed to represent one primary vehicle or world object, over which the controlling host has ownership. The `ChClient` encapsulates all necessary communication with the server, and can be made to do so both with and without the use of multiple threads.

In a serial fashion, the `ChClient` can send and receive messages that are passed to it. This is useful in situations where individual messages that are sent or received must be modified or used sequentially before or after the sending or receiving of the message. It must be noted that significant blocking will occur when the `ChClient` is waiting to receive a message.

```
// Blocks until message has been serialized and sent over the socket
void sendMessage(std::shared_ptr<google::protobuf::Message> message);

// Blocks until message has been received from the socket and parsed
void receiveMessage(std::shared_ptr<google::protobuf::Message> message);
```

The other method of sending and receiving messages using the `ChClient` object is doing so asynchronously. When asynchronously sending and receiving, the `ChClient` object creates its own separate thread, and continuously sends or receives messages until the connection is closed or the `ChClient` is destroyed. In the case of sending, a reference to a message will be passed to the `ChClient`, which will then be read by the `ChClient` before each send operation. In the case of receiving, a reference to a tree-map object is passed to the `ChClient`. The map will be a mapping of identification numbers to world objects. Upon each receive, the `ChClient` updates the received message in the map, or inserts a new entry if the message identification number does not yet have an object assigned to it in the map. No other threads should write to the map after the `ChClient` has begun asynchronously receiving.

```
// Returns immediately and uses the socket to receive from server on another
// thread
void asyncListen(std::map<int, std::shared_ptr<google::protobuf::Message>>&
serverMessages);

// Returns immediately and uses the socket to send to server on another
// thread
void asyncSend(std::shared_ptr<google::protobuf::Message> message);
```

In the interest of maintaining simulation speed, the asynchronous method of receiving messages is particularly effective.

4 Communication Flow and Protocol

The following is a description of the technical aspects behind the process of converting, serializing, sending, receiving, and parsing of world objects.

4.1 Structure of Message Flow

Upon initial connection, the server sends the client its connection number, which corresponds to the identification number of the primary world object with which the client is associated. After the connection number has been sent, the majority of communication on the server end happens in a receive-reply pattern, in which the server receives a request from the client, and responds accordingly with one or more messages pertaining to the current state of the `World` object. Before any message is sent, either by the client or the server, a one byte message code is sent beforehand to identify the nature of the message. In most cases, the message code describes the type of world object being sent (vehicle, pedestrian, etc). Otherwise, the message code is essentially the message itself, used to coordinate actions between the client and the server. The `ChronoMessages::VehicleMessage` is currently only message type used, although more will be added.

After sending all the messages of a requested section, the server then sends a message code indicating that the last of the messages has been sent. This is essential for the client, which uses this information to determine which objects are no longer in the world and need to be removed. This message code can also be used to deduce the number of world objects sent after one request.

In addition to message codes denoting message types for requests and responses, there is also a message code used to indicate a heartbeat. The heartbeat is sent by the server to all clients at a constant rate to ensure the clients all simulate time at the same rate. The server uses the heartbeat to determine what time stamp the messages from the clients should have, and the clients use the heartbeat to adjust their simulation step sizes such that they remain synchronized with the server. If the server detects that a client is not simulating time at the appropriate rate, or that the client isn't sending messages frequently enough, the server may disconnect the client.

4.2 Network I/O

Both the `ChClient` class and the server connect to the network with the use of the Boost.Asio library. Among other things, Boost.Asio provides object-oriented TCP and UDP socket support. Due to its stability and stream-based nature, the current design of the server and `ChClient` makes use of `tcp::socket`s for all message passing. In future versions, hardware restrictions may call for the use of `udp::socket`s in order to make the server scale better with the number of clients. With the current TCP design, the number of threads the server creates is linearly dependent on the number of clients connected to it. Therefore, when the number of clients connected to the server exceeds the number of logical cores in the server's system, the server will begin to run less efficiently. This means that the speed of the server will be significantly affected by the limits of the system on which it is running. The packet-

based nature of the udp::sockets allows for the possibility of a constant number of threads receiving and sending messages to and from all clients, in order to keep from exceeding the optimal number of threads that should be run on any given system. The ability to handle all clients using any number of threads will not reduce the work load of sending and receiving messages, but it will remove the added time that the system would have otherwise spent switching between threads.

Boost.Asio sockets also provide functionality for asynchronous sending and receiving of messages. The current design of the server and ChClient does not use this functionality, but the ChClient has it's own asynchronous capabilities that cater specifically to the needs of the Chrono Autonomous Vehicle Project. This allows for a more streamlined and intuitive design, as well as a more efficient connection handling process.

4.3 Message Parsing and Serialization

World object messages are serialized to and parsed from a stream using Google Protocol Buffers, or protobuf. Custom protobuf message classes generated with the protobuf compiler allow for world object information to be efficiently and reliably sent over the socket. In addition, the custom messages implement the google::protobuf::Message abstract class, allowing section maps of messages within the server and client to contain any type of world message, while maintaining the serialization and parsing functionality that any protobuf message has.

The current message format for a ChronoMessages::VehicleMessage describes the, identification number of the message, the ChTime of the corresponding vehicle, the time stamp of the messages creation, the vehicle speed, the chassis position and orientation, and the positions and orientations of each wheel. Position vectors are represented with a nested ChronoMessages::VehicleMessage_MVector message, which contains x, y, and z coordinates in the form of doubles. Orientation quaternions are represented with a nested ChronoMessages::VerhicleMessage_MQuaternion message, which contains e0, e1, e2, and e3 components that are also stored as doubles.

Serialized on the wire, the ChronoMessages::VehicleMessage takes up exactly 361 bytes. In the future, a more robust form of the vehicle message should exist in which the vehicle may have any number of wheels, which are stored in a map within the message. In this case, the byte size would not be deterministic, and the size of each vehicle message would need to be sent before the message itself is sent.

Protobuf messages for pedestrians and other world objects can be added in the future with little to no structural change to the design of the client and the server. With the inclusion of a unique message code to identify each world object message, additional code can be easily added to handle new message types.

4.4 Object-Message Conversion

When sending the client's vehicle message, conversion from `ChVehicle` to `ChronoMessages::VehicleMessage` is relatively straight forward, with position fields withing the message being set to their corresponding components within the vehicle. However, conversion from `ChronoMessages::VehicleMessage` to a representation of other vehicles in the client's world is more complicated, as the current `ChVehicle` does not allow its position and orientation to be changed externally. To mend this, a new `SeverVehicle` class has been created to represent other vehicles. The `ServerVehicle` contains all the necessary body representations of the chassis and wheels, but complex physics information is not stored. In the future, `ChVehicle` may be modified to allow external changes to its position and orientation, making physics information easier to interpret.