# CARL – Developers Guide

v. 2023-February

Asmus Eilks

April 24, 2023

### Introduction

This Document serves as an introduction to working with the CARL App and its surrounding systems. CARL is designed to allow multi-user studies in augmented reality, including optional Opti-Track tracked objects, across a network.
Some knowledge on part of the reader of how to use the Unity-Engine, Visual Studio and the Microsoft HoloLens is assumed. This document is split into three parts:
Firstly, a quick guide for experimenters that just want to use the existing system as-is to conduct experiments with as little work as possible.
Secondly, a more in-depth guide of the systems capabilities and how they can be extended, for people that want to built on top of the existing system.
Thirdly, an explanation of how the core parts of the application are implemented, and why these implementation decisions were made, for developers who would like to improve the core system in one way or another.

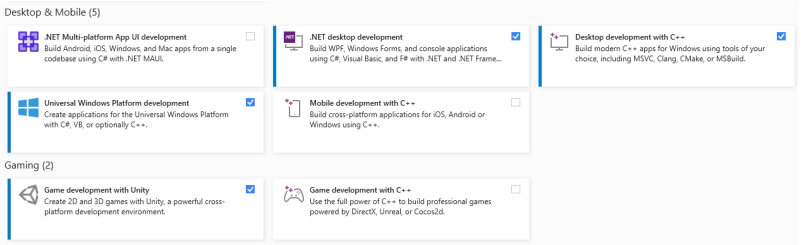
# Contents

# 1 Quick Start Guide

## 1.1 Required Frameworks

CARL is a unity-project, built with Unity 2021.3.17f1. To avoid potential incompatibilities, I recommend using the same version. Download links for its installation can be found in the Unity-Archive. In addition, I have used Visual Studio 2019 as code editor and deploy pipeline, therefore this guide will reference its functionality, it can be installed as part of the Unity-Installation. Make sure to install the packages for .Net Desktop Development, Desktop Development with C++, Universal Windows Platform Development and Game Development with Unity (see fig 1).

Figure 1: VS Installer Packages




For better debugging capabilities, you may also want to install the Windows Mixed Reality Toolkit.

### 1.1.1 For Optitrack

To include an optitrack system, the Optitrack-Motive Application is required. I have used Motive 2017.

## 1.2 Building and executing the existing application without changes

To build the existing application as a Standalone Application on the Hololens, first open the Unity Project, go to File > Build Settings, make sure that the target Platform is Universal Windows Platform and that the following Properties are set the same as fig 2
You can also do a Debug Build if you want to attach a Debugger for any reason, however this comes at a significant cost in performance. Then click "Build".
After the build has finished, navigate to your build folder, and open the LabLinkListen.sln in Visual Studio. Connect your Hololens via USB, make sure it is turned on and unlocked, then select ARM64, Release (or Debug, if you want to make a Debug Build) and Device, then click on Device to build & deploy to that device (see fig 3).

Figure 2: Unity Build Settings

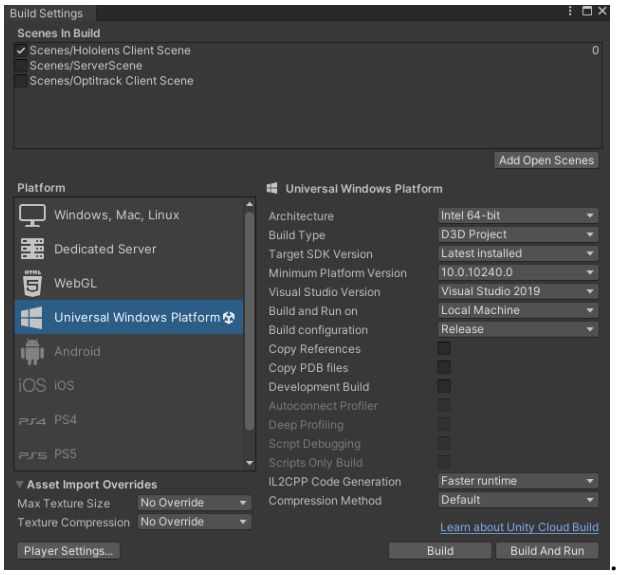



Figure 3: VS Build Settings


In the Unity Editor, open the "Server Scene" in Assets > Scenes and click Play. The application will automatically open a Server on port 6666, so make sure that one isn't

blocked, or change it on the NetworkManager prefab. On the hololens, start the LabLinkListen application, enter the server's IP address with the virtual Keyboard, and click Connect. If the connection was successful, the UI in the Hololens disappears, and a Player should become visible in the Unity Editor.

### 1.2.1 Synchronizing Position

If you have several Hololenses connected, you will need to synchronize their positions. To do so, print a QR-Code containing the word "origin" and place it somewhere in your play area. You can see whether the QR-Code is recognized by the app through the appearance of a coordinate system around the QR-Code. This QR Code serves as the reference origin-point for each hololens - the positions of the other players and virtual objects is synchronized relative to their respective origin points.

### 1.2.2 Adding an optitrack system

If you want to add real objects tracked by an Optitrack-system, first, set up a Rigidbody in motive where you want your origin, and ensure that it has Streaming ID 1. Copy the Unity Project and open the copy in a new Editor Window. For each object you want to track, add an additional Rigidbody, and set up its streaming ID. Per default, the IDs 31-36 are used, how to adjust this will be explained in 1.4. Switch to the "Optitrack Scene" and start it. Click on "connect", and on "Start Spawning", in the editor window, you can see a simple debug window informing you about the server connection and the positions of the tracked objects (again, relative to the origin). You should then be able to see virtual representations of the tracked objects on all connected HoloLenses.

## 1.3 Adding additional objects

To add a new object, add the objects model to the Assets > FBX Assets Folder. Then copy the Synchronized Object Parent in Assets > Synchronized Objects and change the mesh of the "Synchronized Main Mesh" child object to your new model. Be sure to update the mesh-colliders mesh as well, Rename the copy as you see fit, and store it as a new prefab. Next, open the Server Scene, find the "Synchronized Object Spawner" and click on Reload Synchronized Objects, your object(s) should now appear in the list of buttons. Then, go to the Netcode Holder Prefab, and add your new objects to the List of Network Objects on the NetworkManager Component of the NetcodeHolder. Save your changes, and make sure to rebuild & redeploy for any HoloLenses used for the project. You should now be able to spawn your new object across the network and see its movement synchronized between the clients.

### 1.3.1 Adding complex Objects

The system supports synchronization of entire object trees using only one script. Simply add children to an object with a Synchronized Object Parent Script attached, and the position & rotation of all children will be synchronized. This is useful if objects need to preserve references to one another.

## 1.4 Adding additional Optitrack Objects

To add a new object to be tracked by the Optitrack system, first add the mesh as you would any other synchronized object (see 1.3). Then, set up a Rigidbody for the object in motive. Go to the Optitrack scene and find the "Client Optitrack" Gameobject. Add your new GameObject to the "Tracked object spawnables list", and make sure that the "Key" property in that list is set to the same streaming ID the object's Rigidbody has in motive.
Note: The streaming ID 1 is permanently reserved for the origin object, and only positive integers can be valid streaming IDs.

## 1.5 Data Logging

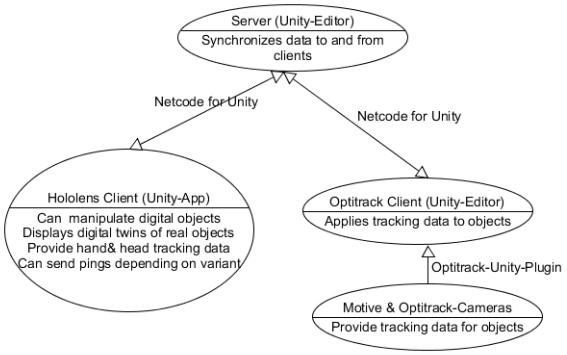
The Server and the Optitrack client automatically open up an LSL-Stream on startup, which can be recorded with the LSL-LabRecorder. In this stream, the Server writes any Log-Messages it generates, and the Optitrack client writes the positions of any tracked objects 5 times per second.
Hololens-Clients need an LSL-Bridge to write to the LabRecorder. They connect (by default) to the same IP-Address as the Server, using the Ports 8877 & 8878 respectively. This can be configured on the Server, by altering the Addresses/Ports on the MetaDataHolder-Object. Clients push the poses of all Hand-Joints and their head 5 times per second, as well as any additional Events (Grabbing&Releasing an Object, Connecting/Disconnecting from the Server, Pings).

# 2 Extending the system

## 2.1 Network & Device Layout

CARL uses a strict Server-Client architecture, where one Server hosts and synchronizes many clients. It was built on top of the Transport Layer of the "Netcode for GameObjects" pre-release in early 2022. The system uses the existing Network Manager to establish connections between clients and server, but does all other Network-Communication via Named Messages, to allow for a high amount of control what exactly is sent when. Its important to note that clients cannot directly communicate to each other, instead messages have to be sent to the server which then forwards it to other clients.

Figure 4: Network Structure




The Server is intended to run inside the Unity Editor, as it has no in-Scene UI. Extensions to the HoloLens-Clients may need to be compiled into UWP or IL2CPP to run, extension to an Optitrack-Client are compiled to a desktop windows machine, and should therefore not have any specific requirements.

## 2.2 Needed Everywhere: Adding additional Messages

Adding almost anything to CARL will require new message Senders/Handlers. Custom Messages are packages of arbitrary data that are identified by a string, and sent between a server and a client. These are used to transfer data and commands between server and clients. To enable different functionalities, I've used a few patterns across the project, and encourage further additions to the project to follow the same patterns, as this keeps the code readable and understandable.
On its most basic, a message is sent using the CustomMessagingManager.SendNamedMessage function. This function requires three parameters:

1. string | messageName: The message Name by which this message is identified. This should be a public const string, whose variable name ends in Key, and is defined in the same class that sends and/or handles this message.

2. ulong | clientID: The ID of the receiving client. If the message is Server-bound, use NetworkManager.ServerClientId.

3. FastBufferWriter | messageStream: A stream containing the messages payload. This should be created right before sending the message, in a using-statement, so its disposed immediately afterwards. To create the FastBufferWriter, simply use its constructor. Use a starting size of 1, a temporary allocator, and a max size equal or larger to the size of the data you want to send. FastBufferWriters can take any primitive data, which is written into them using the WriteValueSafe function. Serializing objects to primitive data will have to be done manually before sending the message.

In total, a message being sent might look like this:

```
public class SampleClass{
    public const string SampleMessageKey = "SampleMessage";
    public void SendSampleMessage(string samplePayload){
        using FastBufferWriter writer = new(1,
            Unity.Collections.Allocator.Temp, 64000){
        writer.WriteValueSafe(samplePayload);
        NetworkManager.Singleton.CustomMessagingManager.SendNamedMessage
            (SampleMessageKey, NetworkManager.ServerClientId, writer);
        }
    }
}
```

Listing 1: Sample Message Sending

The handler for the message must take exactly two parameters: a ulong, to which the senders ID is written, and a FastBufferReader which contains the payload. A function with this signature can then be registered to the CustomMessagingManager using the RegisterNamedMessageHandler function. This should be done a coroutine, using a bool to make sure the handler is not registered more than once. The reason this cannot be reliably done in Start or Awake is that the CustomMessagingManager object is only created when the Server or Client is started, which is often not the case at object creation. The interface IMessageHandler provides appropriate Signatures.

```
public class SampleHandlerClass : MonoBehaviour, IMessageHandler{
    bool msgHandlerSet;
    public bool MsgHandlerSet { get => msgHandlerSet; }
```

```
    public IEnumerator RegisterMsgHandlers()
    {
        while (!MsgHandlerSet)
        {
            if (networkManager.CustomMessagingManager != null)
            {
                networkManager.CustomMessagingManager.RegisterNamedMessageHandler
                    (SampleClass.SampleMessageKey, SampleMessageHandler);
                msgHandlerSet = true;
            }
            yield return null;
        }
    }
    public void SampleMessageHandler(ulong senderID, FastBufferReader
        payload){
         payload.ReadValueSafe(out string message);
         Debug.Log(message);
         }
    }
}
```

Listing 2: Sample Message Handling

To share a message to all other clients, I've usually used a foreach-loop iterating over the clientIDs stored in NetworkManager.ConnectedClientsIds, and send the message to each of them aside the sender. If you want to send a message to all clients, including the Sender, you can also use the SendNamedMessageToAll function.

## 2.3 Adding new Client-Types

To add a new type of client, create a new Scene, add a NetcodeHolder, Synchronized Object Spawner and Device Identifier to it. The Netcode Holder contains all functionality needed to connect to the server, the Synchronized Object Spawner handles spawning and synchronization of objects, and the Device Identifier provides the Server with information about this client and receives info about other clients. Extend the CustomDeviceType enum found in the MetaDataHolder class. Now you can implement your clients functionality inside the project.

## 2.4 Logging additional Datastreams

Adding additional data or datastreams may be necessary for various experiments. If the data is collected on a Desktop-Windows-Machine, it can easily be exposed to LSL by creating a new StreamOutlet object, and pushing the data using streamOut.pushSample(). The LSL_Server script can be references a sample implementation for this functionality. It becomes a little more complicated if a new data stream needs to be introduced to the HoloLens-Devices, as they require a bridge-script to reach LSL. On the Unity side, introduce a new ChannelKey to the BridgeConnectionManager script. You can send arbitrary data to that stream using the Send() function. However, for the bridge to pass on the data, the stream needs to be also registered in the holo_lsl_start_experiment_server.py python script. There, create a new StreamInfo object, analog to the existing Event & HL Tracking Outlets. It is important the stream has the same name as the ChannelKey, and is registered with that Key in the outlets-dictionary of the bridge-script.

```
UNITY
public class BridgeConnectionManager : MonoBehaviour{
    [...]
    public const string SampleChannelKey = "new_sample_stream";
    [...]
}
public class SampleStreamSender : MonoBehaviour {
    public void Update(){
        BridgeConnectionManager.Instance.Send(
        "This is a sample message",
        BridgeConnectionManager.SampleChannelKey)
    }
}
PYTHON
[...]
Sample_Stream_outlet = StreamInfo(name='new_sample_stream',
                                  type='sample_stream',
                                  channel_count=1,
                                  nominal_srate=lsl.IRREGULAR_RATE,
                                  channel_format=lsl.cf_string,
                                  source_id='sample_Data')
[...]
outlets = {
    new_sample_stream: StreamOutlet(Sample_Stream_Outlet, chunk_size=1,
       max_buffered=3600) [...]
}
[...]
```

Listing 3: Sample LSL-Bridge Stream implementation

# 3 Current Implementation

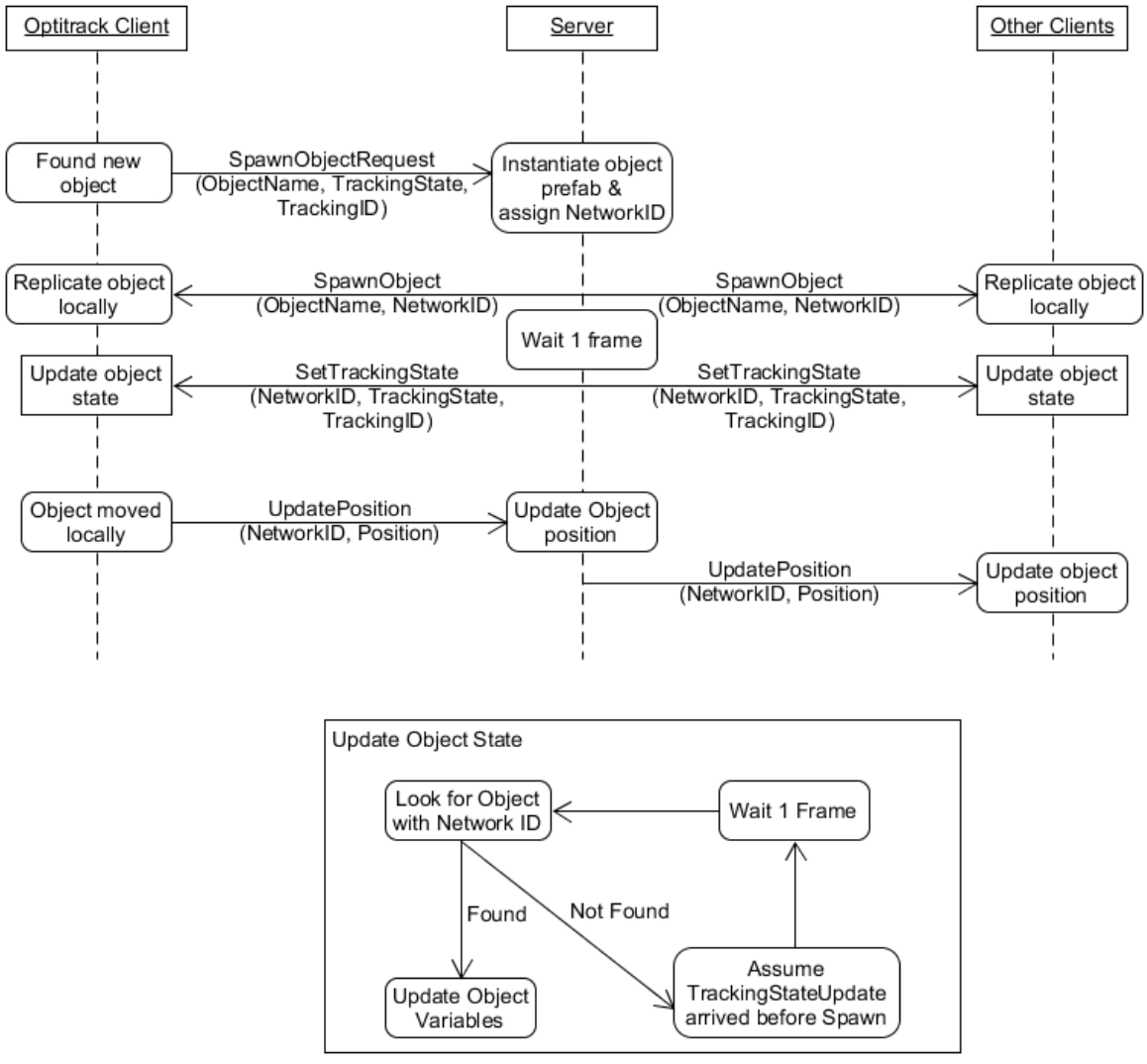
## 3.1 The Life of a Synchronized Object

All synchronized objects have to be spawned at runtime by the server. Right after a synchronized object is spawned, it is given an Owner. This is always the client that called the spawn command. Only the owner can modify an object. The object is also given a unique ID by which it can be identified across the network. Finally, the object is also assigned a trackingState. This marks objects tracked by a constant tracking source, such as the optitrack system, so that their position can not be overwritten by other clients/sources. All of these changes/assignments are then propagated across the network. Now our object is accessible by clients. Every update, the object checks if it has been moved or rotated beyond a certain threshold. If it was, it notifies the SynchronizedObjectManager, which propagates the new pose across the network. That way, objects are always kept in sync. If the object is controlled by a single, consistent tracking source, this is all the synchronization needed. However, if multiple users can interact with the object at any time, ownership needs to be transferred accordingly, so whoever is currently manipulating the object propagates their changes to the other users. As currently the only clients that can do this are Hololens-Clients, this is handled by the ObjectManipulator, which calls GetOwnership when manipulation begins. Other potentially manipulating clients or client-types need to do the same when they start manipulating the object.

### 3.1.1 Special Case: Optitrack tracked objects

With the optitrack system, as well as presumably most other tracking systems, another problem arises: The Optitrack client gets an ID from Motive, which corresponds to an object to be tracked. It then sends a spawn command to the server, and the object is spawned. However, the optitrack client can not simply assign the ID from Motive to the newly spawned object, as it does not have a reference, and it is not guaranteed that the next object spawned is the result of the Optitrack Clients spawn command, since other clients may also be spawning objects at the same time. To solve this issue, the Tracking ID is sent with the Spawn command, and the Server sends out an update containing the Network-ID of the spawned object and the tracking-ID after spawning the object. This way, Optitrack clients can link up the object with their ID. This tracking-ID assignment is also propagated to other clients, which then locally block interaction with the tracked object.

Figure 5: Optitrack Spawning & Sync



## 3.2 QR-Detection & Spatial Sync

For two users to interact in the same relative space, as well as synchronizing the Optitrack space to the HoloLens, some sort of origin in real space is needed. As mentioned in 1.2.1, ive used QR codes for that purpose. All objects that are supposed to be placed in real space, become attached to the Qr-Detected-Origin object on the HoloLens-Clients, and the attached OriginSeeker script then positions the Origin-Object to the location of the last recognized QR-Code that contains the word "Origin". To detect this QR code, Microsofts MixedReality.QR library is used, following this guide by Joost van Schaik.

## 3.3 Optitrack-Implementation

The optitrack implementation uses the optitrack-unity plugin to get data from the Motive Software. The OptitrackObjectSpawner script regularly polls the OptitrackStreamingClient about the data it received, spawning or updating objects if an ID of a motive object matches the ID of an object defined in the TrackedObjectSpawnables list. Details about how objects are spawned and linked to the motive ID are described in 3.1.1. Objects can be given a rotational & positional offset in the TrackedObjectSpawnables list, allowing a developer to adjust a consistently incorrectly positioned/rotated object in the editor. However, the same can be done in Motive, using more intuitive arrow/sphere gizmos, so i recommend adjusting objects in Motive to fix consistent inaccuracies.

## 3.4 AR-Interaction

Interaction with purely virtual objects is implemented using the ObjectManipulator and NearInteractionGrabbable scripts from the MRTK without any noteworthy changes. The only addition is the InteractionHighlighter script, which colors an object red while its being held. This is done by registering the functions to the Object Manipulators Manipulation-Started/Ended events, and further functionality could easily be added the same way.