New York University Abu Dhabi

ENGR-UH 4020 Senior Design Capstone Project II

Spring 2020

Design of a Haptic-Audio-Visual Tele-Dental Training Simulation

Final Report

10 May 2020

Ken Iiyoshi (ki573@nyu.edu)
Mahrukh Tauseef (mt3312@nyu.edu)
Ruth Gebremedhin (rgg282@nyu.edu)

Capstone Mentor:
Dr. Mohamad Eid
Capstone Instructors:
Dr. Pradeep George
Dr. Ramesh Jagannathan

Abstract

Over the past two decades, high-speed communication technologies have revolutionized applications of Tactile Internet (TI) by allowing low-latency data transfer. This has led to the emergence of haptic-based medical simulations that have numerous technical and ethical advantages in medical training. Since dental training is a highly haptic task, tactile internet has the potential to improve the current dental training techniques by allowing communication of motor skills as haptic media. This project designs and implements a detailed, virtual-reality based simulation of a periodontal procedure using haptic technology and a realistic 3D model of the oral cavity. A communication system that allows low-latency transmission of haptic and audiovisual data over a network was developed. The local-hosted communication network performed with minimal overhead; an average delay of 0.62 ms and jitter of 0.53 ms for haptic data, and round trip time of less than 30 ms for audio-visual data. This system enables effective supervised training over a physical distance and is more interactive than commonly used non-haptic computer simulations.

1 Project Management

Management methods such as Work Breakdown Structure (WBS), Design Structure Matrix (DSM), Critical Path Method (CPM), and Gantt Chart were used to plan and schedule the design and implementation process of this project. Each method is discussed in the sections below.

1.1 Work Breakdown Structure (WBS)

A work breakdown structure assists project planning by dividing projects into a series of tasks and sub-tasks [27]. See Table 1 below for the WBS of this project.

Table 1: Project Work Breakdown Structure

Haptic-Audio-Visual Tele-Dental Training Project		Duration	Planned Dates	
Primary tasks	Sub-tasks	(days)	start	End
	0.1 Begin Project	1	9/1/19	9/1/19
1.0 Determine dental needs	1.1 Weekly meetings with dentists	220	11/1/19	5/10/20
	1.2 Review regulatory requirements	20	10/2/19	10/22/19
	1.3 Research alternative solutions	20	10/2/19	10/22/19
	1.4 Create a hierarchical list of dental needs	40	11/1/19	12/8/19
	1.5 Revise problem statement	40	11/1/19	12/8/19
2.0 Generate Concepts	2.1 Functionally decompose the project	21	9/14/19	10/2/19
	2.2 Research for code	49	9/1/19	10/14/19
	2.3 Generate concepts	28	10/14/19	11/8/19
	2.4 Select promising concept(s)	28	11/1/19	11/29/19
3.0 Begin Detailed Design	3.1 Perform detailed analysis of concepts	28	10/13/19	11/7/19
	3.2 Perform simulations	28	11/7/19	12/8/19
	3.3 Material selection/availability	7	12/1/19	12/8/19
	3.4 Component selection/availability	7	12/1/19	12/8/19
	3.5 3D Tongue Modelling	17	1/3/20	1/28/20
4.0 Build Prototype	4.1 Purchase materials	42	12/8/19	1/28/20
	4.2 Machine/manufacture components	21	1/28/20	2/21/20
	4.3 Assemble prototype	35	2/14/20	3/14/20
5.0 Test Prototype	5.1 Develop testing protocol	35	3/14/20	4/14/20
	5.2 Perform tests	21	4/14/04	5/1/20
6.0 Documentation&Report	6.1 Fall mid-term report	14	9/28/19	10/13/19
	6.2 Fall proposal and presentation	14	11/23/19	12/8/19
	6.3 Spring mid-report/ppt	14	3/2/20	3/15/20
	6.4 Final presentation and report	14	4/28/20	5/10/20
7.0 End Project	7.0 End Project	1	5/10/20	5/10/20

1.2 Design Structure Matrix (DSM)

A design structure matrix helps organize the task order for finishing a project. It identifies inputs required for each task. For example, the weekly meetings with dentists feed into literature review, product research, and listing of dental needs priorities [27]. See Table 2 below for the DSM of this project.

Table 2: Project Design Structure Matrix

	0.1	1.1	1.2	1.3	1.4	1.5	2.1	2.2	2.3	2.4	3.1	3.2	3.3	3.4	3.5	4.1	4.2	4.3	5.1	5.2	6.1	6.2	6.3	6.4	7
0.1 Begin Project	0.1																								
1.1 Dentists meetings	x	1.1	x	x																					
1.2 Literature review	x	x	1.2	x																					
1.3 Research products	x	x	x	1.3																					
1.4 Prioritize dental needs	x	x	x	x	1.4																				
1.5 Problem statement	x	x	x	x	x	1.5																			
2.1 Decompose project						x	2.1																		
2.2 Research alternatives						x	x	2.2																	
2.3 Generate concepts						x		x	2.3																
2.4 Select concepts									x	2.4															
3.1 Concepts analysis										x	3.1	x	x	x											
3.2 Perform simulations											x	3.2	x	x											
3.3 Material selection											x	x	3.3	x											
3.4 Component selection											x	x	x	3.4											
3.5 3D tongue modelling								x		x	x	x	x	x	3.5										
4.1 Purchase materials													x	x		4.1									
4.2 Assemble devices															x	x	4.2								
4.3 Assemble prototype																	x	4.3							
5.1 Develop test protocol						x													5.1						
5.2 Perform tests																		x	x	5.2					
6.1 Mid report 1						x															6.1				
6.2 Proposal/presentation										x											x	6.2			
6.3 Mid-report 2																		x				x	6.3		
6.4 Report/presentation																							x	6.4	
7.0 End project																								x	7

1.3 Simplified Critical Path Method (CPM)

Critical path method aids in identifying bottlenecks in a project schedule. In particular, it can identify the extent to which each task can be delayed without delaying the project. It also identifies high risk tasks that cannot be delayed without extending the project completion date [27]. A simple/abstracted critical path of this project, which was constructed based on the primary tasks from the WBS in Table 1, is shown in Figure 1. See Figure 25 in the Appendix (Section 12) for a detailed CPM, which was constructed based on the DSM in Table 2.

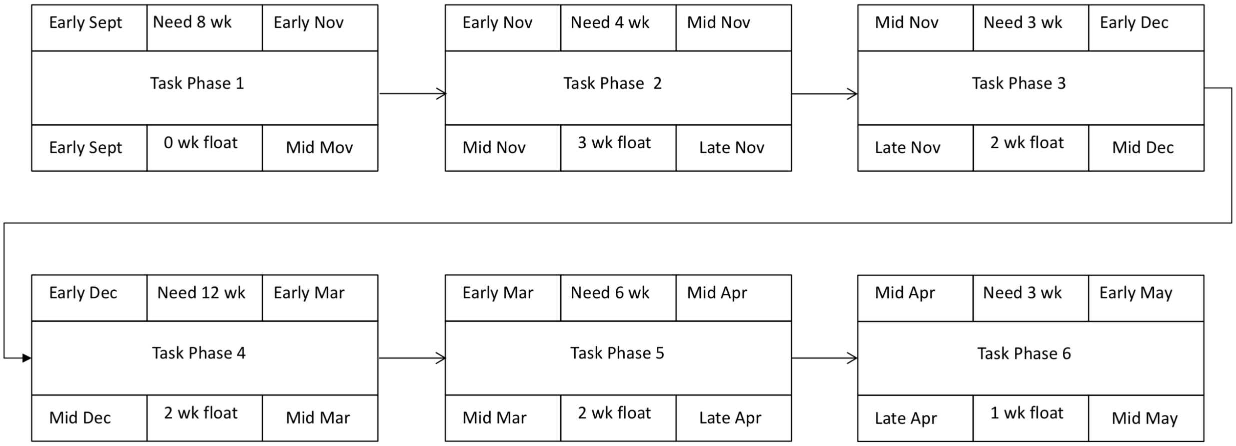


Figure 1: Simplified Project Critical Path

1.4 Gantt Chart

A Gantt Chart is effective for project monitoring. It can correlate tasks with duration time, integrate sub-tasks having separate scheduling charts, and visually represent high level assessment of project progress [27]. See Figure 2 below for the Gantt Chart that was used in this project.

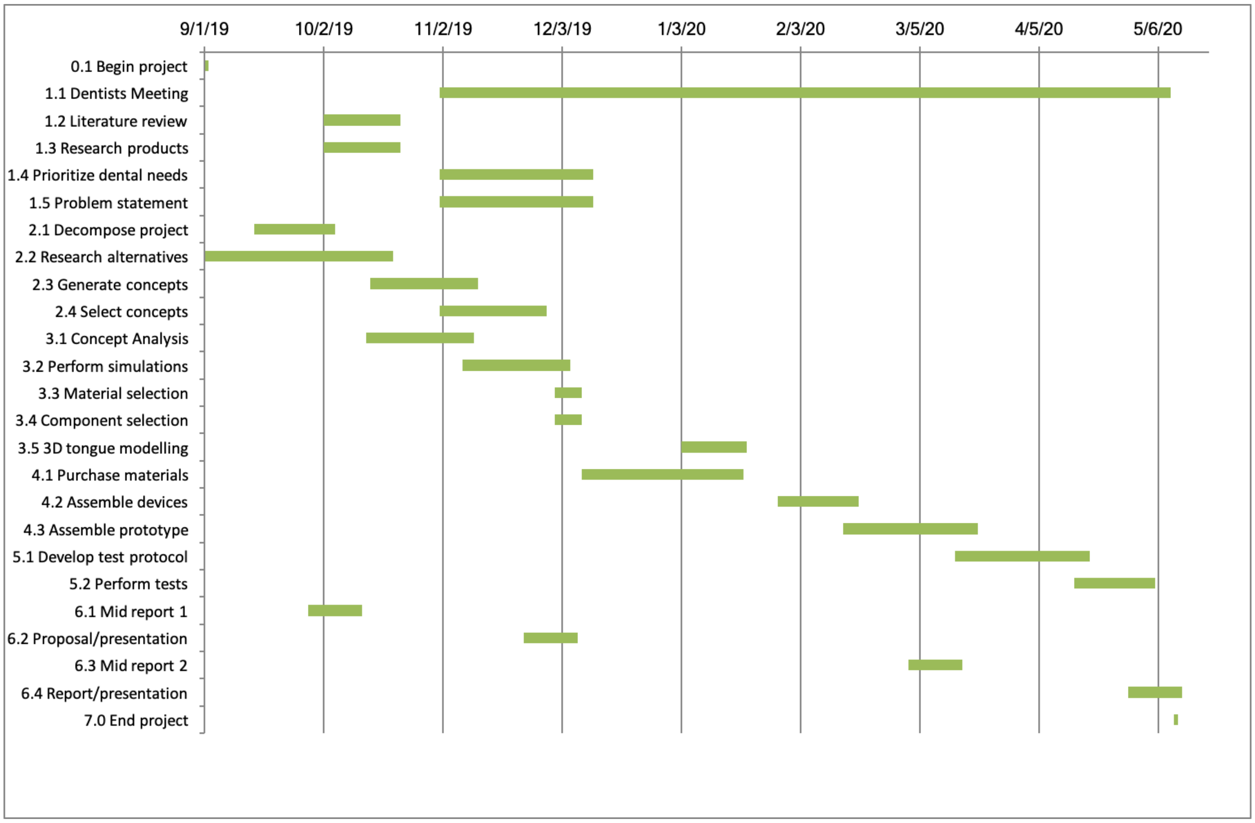


Figure 2: Project Gantt Chart

1.5 Changes made to Project Management

As discussed in Section 9, due to local lock down policies enacted in response to COVID-19, access to the hardware for dental simulation application was restricted. This limited the development of the project application and impacted meetings with dentists.

2 Problem Definition

2.1 Problem Analysis

1) Who has the problem?

   Dental professionals and students face this problem when they try to convey and understand the proper way of probing teeth.

2) What does the problem seem to be?

   With recent improvements in network communication technologies, tele-operation systems can now host medical simulations under sufficient stability and reliability. However, medical operations, specifically dental probing, are highly dependent on tactile feedback which is generally not available while tele-operating.

3) What are the available resources?

   The resources are haptic devices such as *Novint Falcon* and *Geomagic Touch*, libraries for virtual haptic experiences such as CHAI3D and Oculus VR, and networking software such as WebRTC and NS3.

4) When does the problem occur? Under what circumstances?

   This problem occurs when a there is a physical distance between a dental instructor and student.

5) Where does the problem occur?

   This problem occurs in dental facilities and medical schools.

6) Why does the problem occur?

   This problem occurs because most medical tele-operation systems, specifically dental tele-operation systems, do not have a touch feedback.

7) How does the problem occur?

   This problem occurs when a dental professional wants to instruct a dental student over a distance. A dental professional is able to only convey audio-visual data with currently available technology. However, dental probing is a highly tactile task in which touch feedback is crucial for a proper diagnosis.

2.2 Problem Clarification: Black-Box Modeling

The black-box model in Figure 3 was constructed based on the problem analysis.

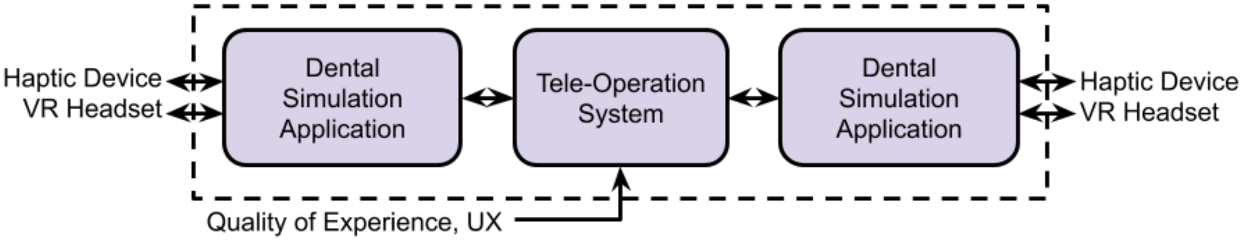

Haptic Device
VR Headset
Dental Simulation Application
Tele-Operation System
Dental Simulation Application
Haptic Device
VR Headset
Quality of Experience, UX

Figure 3: Simple Black Box Model of the Tele-Operated Dental Simulation

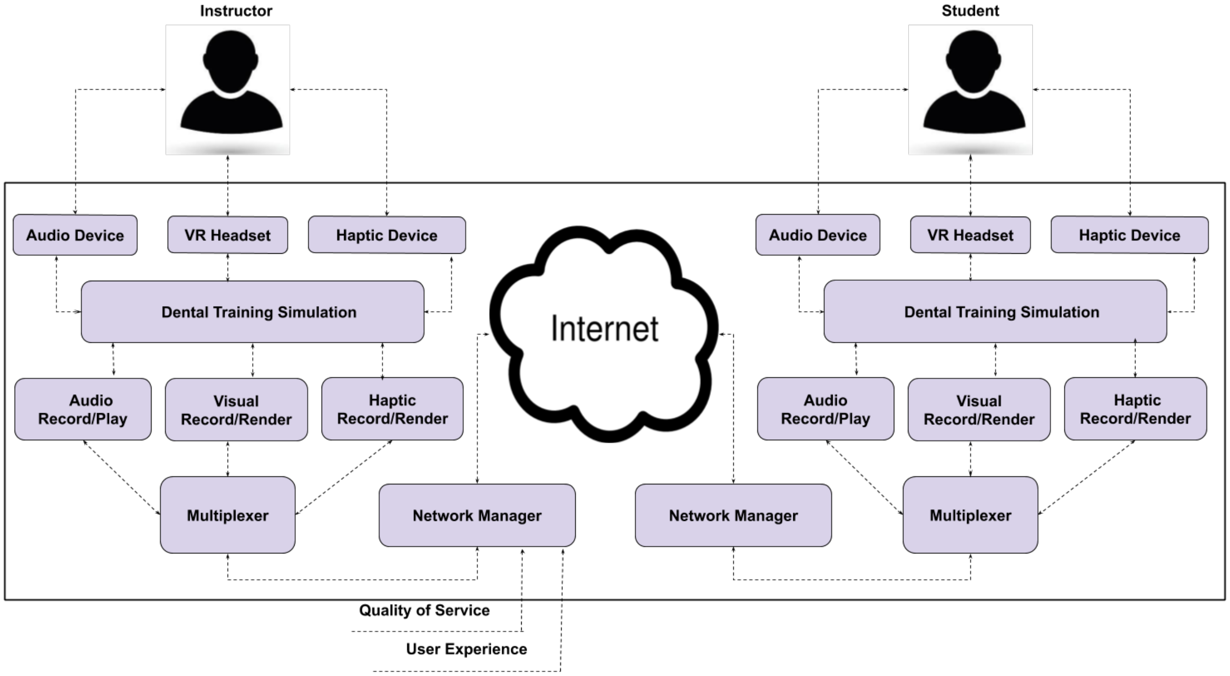


Figure 4: Detailed Open Box Model of the Tele-Operated Dental Simulation

As can be seen from Figure 3, the input and output of the system is haptic-audio-visual (HAV) data stream. The main goal of the system is to communicate HAV data through a network according to the specifications required by the Quality of Experience. This is further illustrated in Figure 4. The dental professional first probes and interacts with the simulated dental model in which their hand movement and fine motor skills are recorded in three modes; namely audio, video and haptic. This data is then transmitted through a network to the dental student's setup. The haptic devices in the student's setup serve as actuators where the teacher's fine motor skills are replayed. The student is also be able to receive audio-visual feedback. This is all be done in real-time (see section 2.4.1 for specifications of what constitutes real-time in each modality) so that the system gives the same experience as a dental teacher physically interacting and guiding the movements of their student.

2.3 Problem Statement

Recent advancements in technology have enabled us to communicate through auditory and visual modalities with minimum delays. However, most communication systems lack one of the most important modalities for human communication– touch.

Touch plays an important role in forming the perception of physical environments. Particularly in medical settings, surgeons use their sense of touch to differentiate between different tissues, perceive pressure, and recognize blood vessels. In the dental field, touch and pressure feedback is used to recognize unhealthy gums and teeth.

Traditionally, the training to become a medical professional, such as a dentist, is given in person. The teacher (dental professional) has to physically show their students how to hold the dental probe, how much pressure to apply on the gums, and how much pressure a healthy tooth can withstand. These tasks require fine motor skills and high sensitivity to touch. Thus, it is clear that dental training over long distance requires touch to mimic real learning environments. Hence, a capstone project that enables real-time, long-distance, networked HAV dental training was proposed.

In proposing this design project, it was recognized that a networked system introduces additional constraints. In a tele-operated system, end users should feel and perform as if there was no distance between them. This is ensured by making the end-to-end delay introduced by the network well below the delay that is perceived by humans. Hence, a maximum delay of 70 ms for haptic data, 200 ms for visual data, and 400 ms for auditory data must be ensured. Additionally, the jitter (packet delay variation) introduced by the network must be less than 20 ms for haptic data to make sure it is not perceived by human end users.

The proposed design project is also constrained by the available resources. In this project, the use of haptic devices that provide a single point of interaction for haptic feedback was proposed despite the fact that a touch sensation is produced by multi point interaction. See section 2.4.1 for more details on the technical constraints.

2.4 Design Constraints

2.4.1 Technical Constraints

1) Human Perception of Communication Delay [18]

   a) Maximum delay of 70 ms for haptic data.

   b) Maximum delay of 200 ms for visual data.

   c) Maximum delay of 400 ms for auditory data.

   d) Maximum jitter (packet delay variation) of 20 ms for haptic data.

2) Dexterity of Interaction provided by Haptic Hardware

   a) No Torque Feedback provided

   b) 6 Degrees of Freedom

   c) Single Point of Interaction for haptic feedback

2.4.2 Non-Technical Constraints

1) Cost: Maximum cost - $ 3000 (estimated from the price of two *Geomagic Touch* devices $1200x2, one *Novint Falcon* $200, one *Oculus Rift* $400, one Leap Motion Controller $150:- a total of $3150 exactly)[7][11].

2) Safety: The haptic devices must never be unstable as an unstable force feedback is a safety hazard.

3) Portability: The entire system setup must be mobile so that it can be easily integrated in multiple dental institutions.

4) Maintenance: The haptic devices require a specific expertise to maintain and debug them in-case of a malfunction.

3 Conceptualization

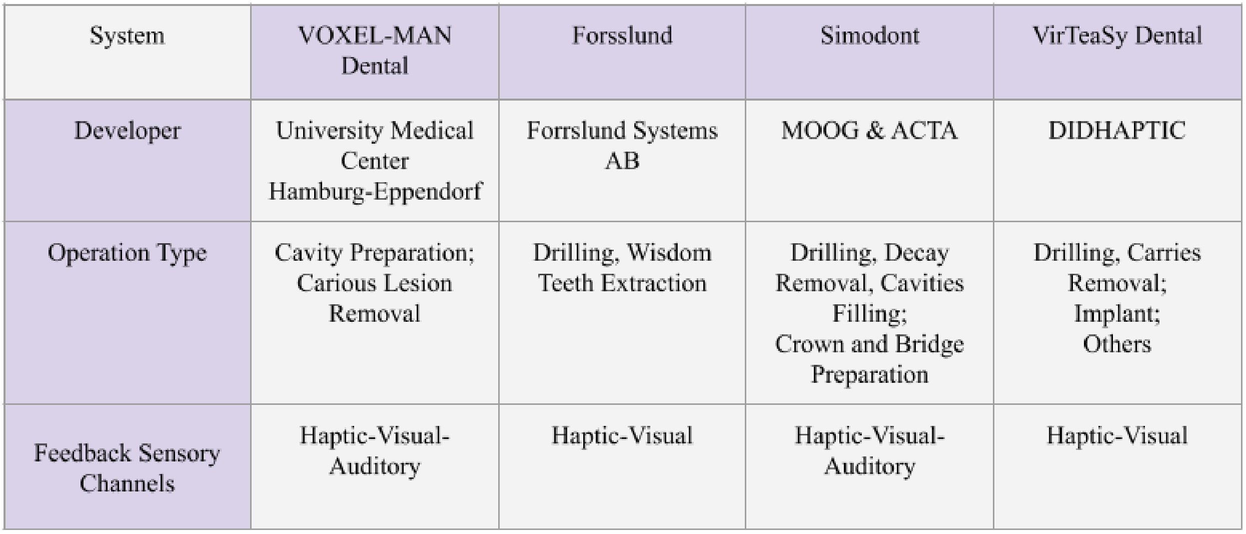
3.1 Background Research

Medical simulation has become an essential component of medical training as it offers solutions to various ethical and accessibility challenges [17]. Medical training on human subjects presents an ethical dilemma since it can put the subjects in danger [23]. Additionally, training on human subjects increases the cost, as well as the duration, of many medical procedures [32]. As the concern about the safety of human subjects grows, there is a increasing need for accurate, life-like medical simulation systems that offer medical students the necessary experience before they perform on human subjects. The first accurate mannequin simulation model, Harvey Mannequin, replicated the human anatomy and its functions [32]. It recreated many of the physical aspects of cardiology examinations, including palpitation, auscultation and electrocardiography [32].

This project focuses on dental operations, specifically the periodontal procedure. The periodontal procedure is used to diagnose periodontal diseases that are caused by calcified plaque and bacteria [36]. These diseases lead to inflammation of the space between the tooth and the surrounding tissues [36]. The areas with affected tissues are called periodontal pockets. A periodontal probe is used to find the depth of the space between the tooth and the gingival sulcus to detect any signs of periodontal pockets. The depth of these pockets is measured by the markings at the end of the periodontal probe. Dental students are trained to recognize the depth of a pocket by sensing the interaction between the markings on the probe and the tissues/gums surrounding the pocket.

A dental simulation lab has now become an essential component of dental education and training centers. These labs include physical 3D models (i.e. typodonts [30]) as well as advanced 3D modeling softwares and systems for dental training. Low-cost typodonts have limited physical properties (i.e. texture, stiffness, etc.) that make them less realistic [35]. However, lifelike typodonts are costly [35]. Due to these limitations, there is an increased interest in interactive software-based 3D models and virtual reality [35]. One of the earliest designs of Virtual Reality Dental Training (VRDT) was built during the late nineties and it provided training on cavity preparation [31].

The advancement of haptic technology has made it possible for developers to integrate haptic feedback to VRDT. The term *haptic* refers to two types of perception; tactile and kinesthetic perception [17]. The word *tactile* is defined as "human perception of touch" [17]. This means that *haptic* refers to the feeling of touch as well as the perception of torque, force, velocity, etc. [17]. Haptic technology has numerous applications in medical simulation, telesurgery, remote disaster management, etc. Haptic feedback has been integrated to several commercial dental simulators. Figure 5 lists the specifications of these simulators as recorded by a survey published in the European Journal of Dental Education in 2015 [35]. This figure highlights the increasing interest in the integration of haptic-audio-visual sensory channels within dental simulators.



System	VOXEL-MAN Dental	Forsslund	Simodont	VirTeaSy Dental
Developer	University Medical Center Hamburg-Eppendorf	Forrslund Systems AB	MOOG & ACTA	DIDHAPTIC

Operation Type	Cavity Preparation;
	Carious Lesion Removal
		Drilling, Wisdom Teeth Extraction	Drilling, Decay Removal, Cavities Filling;
			Crown and Bridge Preparation
				Drilling, Carries Removal;
				Implant;
				Others

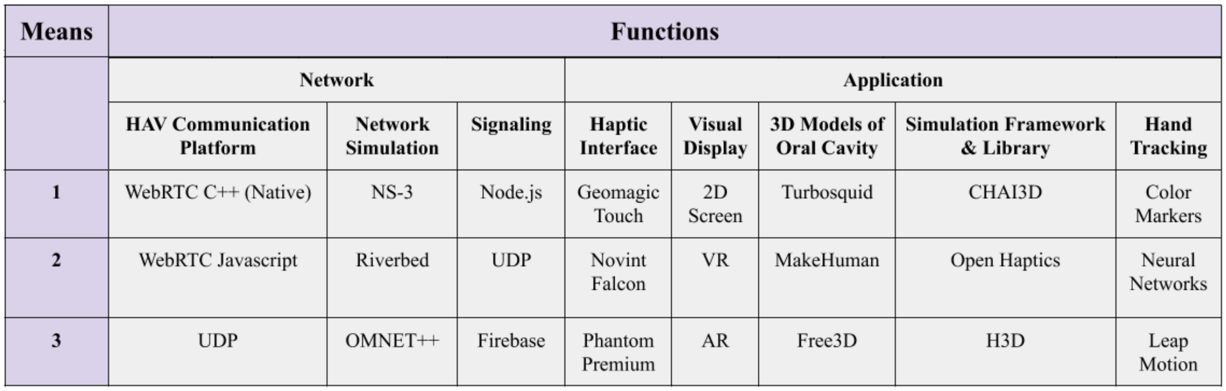
Feedback Sensory Channels	Haptic-Visual-Auditory	Haptic-Visual	Haptic-Visual-Auditory	Haptic-Visual

Figure 5: List of Commercial Dental Simulators and their Specifications. Adapted from [35]

Similar to the commercial simulators listed in Figure 5, this design project aims to develop a dental simulator that performs an interactive periodontal procedure training. The main difference between the design proposed in this project and those identified in Figure 5 is that this design allows dental simulation over the Internet, specifically the Tactile Internet(TI). Tactile Internet(TI) can be understood as the ability to communicate human perception of touch via the Internet. This technology heavily relies on ultra-reliable low-latency communication (URLLC) networks [34]. The emergence of innovative internet technologies such as 5G Internet has enabled the TI, revolutionizing the versatility of applications that focus on communicating touch. As such, this project proposes haptic-audio-visual simulation of the periodontal procedure communicated over the Tactile Internet.

3.2 Concept Generation: Morphological Chart

After a thorough literary research, the project was broken down into sub-problems. A morphological chart, which is a visualization tool for listing implementation methods and techniques for a set of proposed tasks/problems, was then generated (see Figure 6). The functions are categorized into those for the dental simulation application, and those for the HAV network, both of which are further described in the following sections.



Means	Functions							
	Network			Application				
	HAV Communication Platform	Network Simulation	Signaling	Haptic Interface	Visual Display	3D Models of Oral Cavity	Simulation Framework & Library	Hand Tracking
1	WebRTC C++ (Native)	NS-3	Node.js	Geomagic Touch	2D Screen	Turbosquid	CHAI3D	Color Markers
2	WebRTC Javascript	Riverbed	UDP	Novint Falcon	VR	MakeHuman	Open Haptics	Neural Networks
3	UDP	OMNET++	Firebase	Phantom Premium	AR	Free3D	H3D	Leap Motion

Figure 6: Morphological Chart

3.2.1 Network

- **HAV Communication Platform:** In order to communicate HAV data from the trainer to the trainee, a communication platform is needed. Three options listed below are considered for this problem.

  1) **WebRTC - Native C++ and JavaScript:** Web Real Time Communication (WebRTC), which was standardized through World Wide Web Consortium (W3C) and Internet Engineering Task Force (IETF) [26], is an open-source, web-based, real-time communication API (Application Programming Interface). WebRTC is designed to enable cross-platform, cross-browser, real-time multimedia communication between two nodes/peers. WebRTC is also designed for peer to peer (P2P) communication of multi-modal data, as opposed to the conventional server-client architecture, which minimizes network congestion. While the WebRTC API is mainly available in JavaScript, it can be also be developed using C++ for a native platform. WebRTC allows communication via User Datagram Protocol (UDP) as well as Transmission Control Protocol (TCP). UDP allows faster but unreliable communication whereas TCP is used for more reliable but slower communication.

In cases of real-time communication which require minimal delay, UDP is usually chosen over TCP.

   2) **UDP:** A real-time communication platform can be developed from scratch using UDP. UDP is generally used by APIs like WebRTC at a lower level to achieve a low delay communication.

- **Network Simulation** platforms can be used in order to evaluate performance under different network conditions. Such simulations allow the evaluation of the dental simulation's performance while varying the delay and jitter during communication. The considered simulation platforms are listed below.

   1) **Network Simulator 3 (NS-3)** is an open-source Linux-based platform that can be used to simulate different types of network connections (i.e. ethernet, Wi-Fi, etc.) to test the performance of a chosen communication platform [24].
   2) **Riverbed** [12] is a professional tool that is used to measure the performance of a communication system. This system can also test the performance of a system over 5G internet connection.
   3) **OMNET++** is one of the oldest open-source platforms that does not only work as a network simulator but it can also be used for "modeling of multiprocessors and and performance evaluation of complex software systems" [24].

- **Signaling:** Although WebRTC communicates multi-modal data between two peers without using a server, it needs a signaling server to coordinate communication between the peers. Before data communication starts, the peers coordinate by sending control metadata via the signaling server. The WebRTC API does not implement signaling platforms. Hence, the considered signaling servers are listed below.

   1) **Node.js Server (Localhost):** [19] Node.js is an open-source JavaScript runtime environment. A Node.js locally hosted server can be established using the Socket.IO Node.js module.
   2) **UDP** [1] is a protocol that can be used for signaling. UDP compromises on reliability in order to achieve low latency. Due to its low-level implementation, it can become very difficult to integrate a UDP signaling system to a communication platform.
   3) **Firebase** [2] is a web application development platform developed by Google. Firebase can be used to establish a signaling server but is no longer open-source.

3.2.2 Application

- **Haptic Interface** is used to record and actuate haptic data in the form of position, velocity and/or force data. In this project, the interaction of the dental tools (the probe and the mirror) with the 3D model of teeth, gums, tongue and cheeks is recorded and then actuated. Amongst the different haptic devices available in the market, three of them were considered: *Geomagic Touch* [11], *Novint Falcon* [7], and *Phantom Premium* [8]. All three devices can interact with 3D objects in a virtual graphical environment.

1) A ***Geomagic Touch*** consists of a stylus that has six degrees of freedom and its range of motion includes hand pivoting at wrist [11].
2) A ***Novint Falcon*** consists of a removable spherical grip that has three degrees of freedom [6]. Its work-space is smaller than that of the *Geomagic Touch*.
3) A ***Phantom Premium*** also consists of a stylus and has 6 degrees of freedom like the *Geomagic Touch*. In addition, a Phantom Premium also provides torque feedback.

- **Visual Display** is used for the display of 3D models and the setup can be achieved in three different ways.

  1) **2D visual display**, where the trainer and trainee see the set-up on a 2D screen, is commonly used in simulations.
  2) **Virtual Reality (VR)** environment allows the user to feel as if they are physically present in front of the set up. This makes the experience more immersive.
  3) **Augmented Reality (AR)** overlays or projects images and/or other digital features onto the real world environment and allows user interaction.

- **3D Models of Oral Cavity** can be made from scratch or retrieved from existing 3D model databases.

  1) **Turbosquid** [3] is a website that has high definition 3D models. Although the more realistic models are expensive, the website offers a variety of models with a variety of prices. The models are available in a variety of formats.
  2) **MakeHuman** [28] is an open source software that enables users to build 3D models of different human body parts from base meshes. It has high flexibility for making 3D models of the human anatomy specifically. These models can be exported in a variety of formats.
  3) **Free3D** [9] is a website that has a range of 3D models with different levels of complexities. This makes some of the models cost-effective. The models are available in a variety of formats.

- **Simulation Framework and Library**: Another important feature of this project is to enable the playback of HAV data once the simulation ends. Certain libraries have functions that can be used to achieve this feature. Some of the common libraries available are: CHAI-3D, Open Haptics and H3D.

  1) **CHAI3D** [4] is an open-source software that serves as a framework for computer haptics, visualization and interactive real-time simulation. It supports multiple haptic devices with different degrees of freedom. It can work with several libraries that can create interactive real-time VR simulations.
  2) **Open Haptics** [15] is a software that allows haptic interaction with 3D design applications. It can work alongside graphic libraries like OpenGL which creates high quality objects that can interact with haptic devices. Open Haptics only supports 3D System PHANTOM devices.

3) **H3D** [29] is an open-source, real-time development platform that uses platforms like OpenGL and X3D to create simulations with graphical and haptics integration. It also allows integration of multiple haptic devices.

- **Hand Tracking** records the position of the instructor's hand in the physical environment and maps it to the haptic interface device that serves as the finger support. This mechanism makes the simulation ergonomic and realistic by imitating how a dentist rests their ring finger on a patient's front teeth. Three different ways were considered to achieve this tracking.

  1) **Color Markers** can be tracked real-time through computer vision [25]. An RGB color model, which takes in video data from two webcams placed perpendicularly, can be used to deduce the position of a specific color in the field of view.

  2) **Neural Network** can be trained using images of hands in different orientation. This model can then be used to recognize hands and specific fingers in the physical environment.

  3) **Leap Motion** is a sensor that can track the position of hands and fingers. It uses infrared cameras and algorithms that enable it to accurately track hands in real-time and predict their position if they are occluded [5].

3.3 Concept Selection: Pugh Charts

Once the morphological chart was completed, Pugh charts were generated to select the best solution based on an evaluation criteria. For each problem in the morphological chart, each proposed solution was set as base (one at a time) while the remaining solutions were compared to the base. The following scoring system was used for comparison:

- -1 means inferior to the base
- 0 means similar to the base
- 1 means superior to the base

The sum of these scores was taken for all the Pugh Charts to select the best solution.

3.3.1 Network

- **HAV Communication Platform**

  Table 3 shows the Pugh Chart of HAV Communication Platform. The total score supported the selection of either WebRTC, Native C++ or JavaScript. The choice between C++ based WebRTC and the JavaScript based WebRTC depended on the trade between latency and the complexity of the program. Since, JavaScript based WebRTC seemed to comply with the minimal allowable latency for the communication of haptic-audio-visual data, it was chosen over C++ based WebRTC. This assessment was based on the following:

  1) Complexity: Is the platform least complicated to use?
  2) Latency: Is the delay of the communication minimum?
  3) Real-time: Is the communication real-time?
  4) Reliability: Is the communication reliable (with fewer packets loss?)

Table 3: Pugh Charts of HAV Communication Platform

Concept	Evaluation Criteria				
HAV Communication Platform	Complexity	Latency	Real-time	Reliability	Sum
WebRTC Native C++	base				
WebRTC JavaScript	+1	-1	0	0	0
UDP	-1	+1	0	-1	-1
WebRTC Native C++	-1	+1	0	0	0
WebRTC JavaScript	base				
UDP	-1	+1	0	-1	-1
WebRTC Native C++	+1	-1	0	+1	1
WebRTC JavaScript	+1	-1	0	+1	1
UDP	base				

- **Network Simulation**
  Table 4 shows the Pugh Chart of Network Simulation. The total score supported the selection of NS-3 or OMNET++. However, NS-3 was chosen over OMNET++ since it has more networking features and its simulation is faster. This assessment was based on the following:

  1) Cost: Is the technique most cost-effective?
  2) Complexity: Is the platform less complicated to use?
  3) Networking Features: Does the technique have the most features for network simulations (i.e. 4G and 5G simulation)?
  4) Time: Is the simulation fast enough?

Table 4: Pugh Charts of Network Simulation

Concept	Evaluation Criteria				
Network Simulation	Cost	Complexity	Networking Features	Time	Sum
NS-3	base				
Riverbed	-1	+1	+1	-1	0
OMNET++	0	+1	-1	-1	-1
NS-3	+1	-1	-1	+1	0
Riverbed	base				
OMNET++	+1	+1	-1	+1	2
NS-3	0	-1	+1	+1	1
Riverbed	-1	-1	+1	-1	-2
OMNET++	base				

- **Signaling**

  Table 5 shows the Pugh Chart of Signaling. The total score supported the selection of Node.js or UDP. Node.js was chosen over UDP because the reliability was prioritized. This assessment was based on the following:

  1) Cost: Is the tool cost-effective?
  2) Latency: Does the signaling allow end-to-end communication with minimum delay?
  3) Reliability: Is the tool reliable enough (i.e. has minimum packet loss)?
  4) Compatibility: Is the tool compatible with both C++ and JavaScript WebRTC implementations?

Table 5: Pugh Charts of Signaling

Concept	Evaluation Criteria				
Signaling	Cost	Latency	Reliability	Compatibility	Sum
Node.js	base				
UDP	0	+1	-1	0	0
Firebase	-1	0	0	-1	-2
Node.js	0	-1	+1	0	0
UDP	base				
Firebase	-1	-1	+1	-1	-2
Node.js	+1	0	0	+1	2
UDP	+1	+1	-1	+1	2
Firebase	base				

3.3.2 Application

- **Haptic Interface**

  Table 6 shows the Pugh Chart for Haptic Interface. The total score supported the selection of the *Phantom Premium*. However, a *Geomagic Touch* was chosen because the low cost was an advantage. The realism and data quality of a *Geomagic Touch*, although lower than that of a *Phantom Premium*, were good enough for the setup. This assessment was based on the following:

  1) Cost: The product with the minimum cost.
  2) Realism: The product that allows closest to real experience.
  3) Ease of use: The product that is more compatible with the Haptodont setup is preferred.
  4) Data Quality: The product with maximum data quality is preferred.

Table 6: Pugh Charts of Haptic Interface

Concept	Evaluation Criteria				
Haptic Interfaces	Cost	Realism	Ease of Use	Data Quality	Sum
Geomagic Touch	base				
Novint Falcon	+1	-1	-1	-1	-2
Phantom Premium	-1	+1	0	+1	1
Geomagic Touch	-1	+1	+1	+1	2
Novint Falcon	base				
Phantom Premium	-1	+1	+1	+1	2
Geomagic Touch	+1	-1	0	-1	-1
Novint Falcon	+1	-1	-1	-1	-2
Phantom Premium	base				

- **Visual Display**
  Table 7 shows the Pugh Chart for the Visual Display. The total score supported the selection of virtual reality. This assessment was based on the following:

  1) Cost: Is the method cost-effective?
  2) Realism: Does the method give a close to real experience for a dental procedure?
  3) Immersion: Is the method more immersive and less distractive?
  4) Flexibility: Is it flexible enough to give the user a complete control over the setup?

Table 7: Pugh Charts of Visual Display

Concept	Evaluation Criteria				
Visual Display	Cost	Realism	Immersion	Flexibility	Sum
2D Screen	base				
VR	-1	+1	+1	+1	2
AR	-1	+1	+1	+1	2
2D Screen	+1	-1	-1	-1	-2
VR	base				
AR	+1	-1	-1	-1	-2
2D Screen	+1	-1	-1	-1	-2
VR	-1	+1	+1	+1	2
AR	base				

- **3D Models of Oral Cavity**
  Table 8 shows the Pugh Chart for the sources available to obtain 3D models of tongue, cheeks, head, and teeth, etc. The total score supported the selection of Free3D. This assessment was based on the following:

  1) Cost: Is the model cost-effective?
  2) Realism: Does the technique give a close to real experience for a dental procedure?
  3) Complexity: Do the models have the appropriate complexity (i.e. have a suitable number of polygons) to fit the design?
  4) Flexibility: Is the format of the models flexible enough to be used with 3DS MAX?

Table 8: Pugh Charts of 3D Models of Oral Cavity

Concept	Evaluation Criteria				
3D Modeling of Oral Cavity	Cost	Realism	Complexity	Flexibility	Sum
MakeHuman	base				
TurboSquid	-1	+1	-1	-1	-2
Free3D	0	-1	+1	+1	1
MakeHuman	+1	-1	+1	+1	2
TurboSquid	base				
Free3D	+1	-1	+1	+1	2
MakeHuman	0	+1	-1	-1	-1
TurboSquid	-1	+1	-1	-1	-2
Free3D	base				

- **Simulation Framework and Libraries**
  Table 9 shows the Pugh chart for framework and library used to make the dental simulation application. The total score supported the selection of CHAI3D or H3D. Chai3D was chosen over H3D because of its compatibility with multiple 3D and 2D object filetypes for the development of the VR simulation. This assessment was based on the following:

  1) Accessibility: Is the platform open-source?
  2) Multimedia Support: Is the software compatible with multiple 3D and 2D object filetypes?
  3) Supporting Haptic Devices: Is the platform easily compatible with multiple haptic devices?
  4) Availability: Is there an existence of a community that supports the platform? Is the API actively updated and developed?

Table 9: Pugh Charts of Simulation Framework and Library

Concept	Evaluation Criteria				
Simulation Framework & Library	Accessibility	Multimedia Support	Supporting Haptic Devices	Availability	Sum
CHAI3D	base				
Open Haptics	-1	-1	-1	0	-3
H3D	0	-1	+1	0	0
CHAI3D	+1	+1	+1	0	3
Open Haptics	base				
H3D	+1	+1	+1	0	3
CHAI3D	0	+1	-1	0	0
Open Haptics	-1	-1	-1	0	-3
H3D	base				

- **Hand Tracking**
  Table 10 shows the Pugh Chart for the hand tracking. The total score supported the selection of either color based tracking algorithm or Leap Motion. Though they had the same score, Leap Motion was chosen over the color based tracking algorithm since high accuracy and low complexity were prioritized. This assessment was based on the following:

  1) Cost: Is the technique most cost-effective?
  2) Complexity: Is the platform least complicated to use?
  3) Latency: Does the technique introduce the shortest delay time?
  4) Accuracy: Is the tracking done accurately?

Table 10: Pugh Charts of Hand Tracking

Concept	Evaluation Criteria				
Hand Tracking	Cost	Complexity	Latency	Accuracy	Sum
Color Markers	base				
Neural Networks	0	-1	-1	+1	-1
Leap Motion	-1	+1	-1	+1	0
Color Markers	0	+1	+1	-1	1
Neural Networks	base				
Leap Motion	-1	+1	+1	0	1
Color Markers	+1	-1	+1	-1	0
Neural Networks	+1	-1	-1	0	-1
Leap Motion	base				

4 Simulation and Experimentation

4.1 Network

One of the key features of this design is its real-time implementation. Thus, it is important to make sure that the end to end delay is well below what is perceptible and within the set technical constraints. In particular, the constraint is the maximum allowable delay after which the latency becomes perceptible by the user (see Section 2.4.1 for further detail). WebRTC and NS-3 were used to evaluate and simulate end-to-end communication delay of the dental simulation, as described below.

4.1.1 Evaluation through WebRTC Browser

WebRTC's *getStats()* function is an easy method used to get the reading of network performance (i.e. delay, jitter, packet-loss, packets sent per second, bits sent per second). Figure 7 below shows the results obtained from the *getStats()* function while an example WebRTC application is running. Since this method allows to monitor parameters like delay, jitter and packet loss while the communication system is running, it gives real time feedback on which network conditions affect the parameters the most.

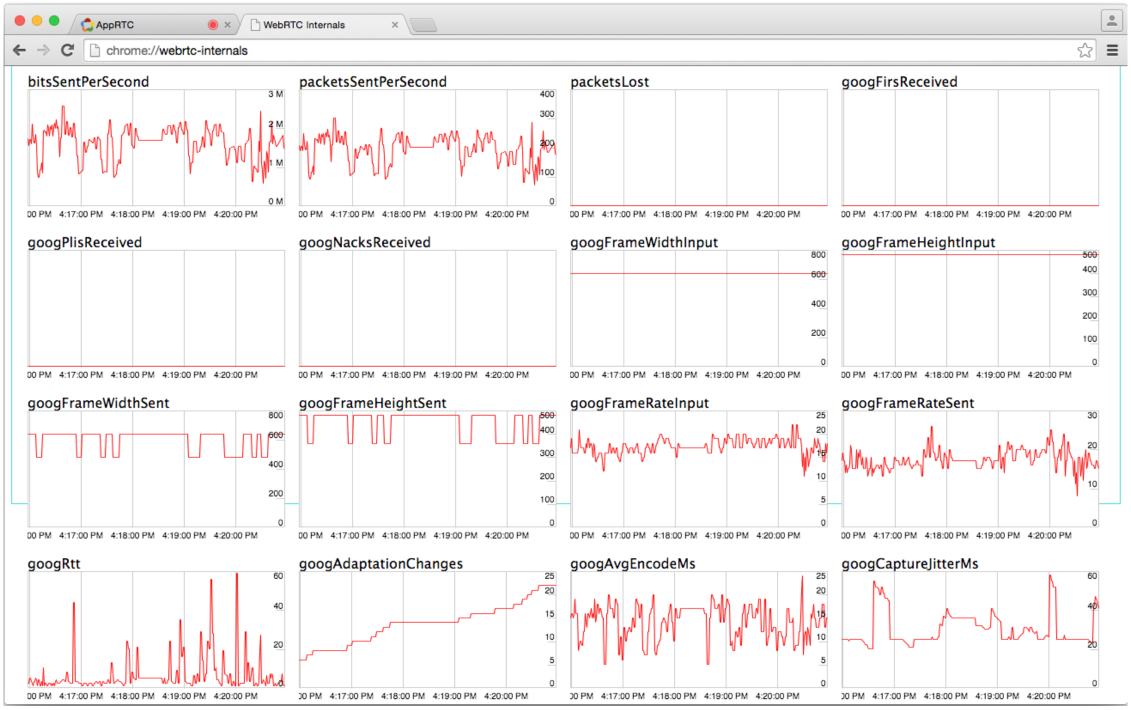


Figure 7: Real-time Plots generated from WebRTC *getStats()*

4.1.2 Simulation through NS-3

The HAV communication network performance can be simulated using NS-3; a network simulator that can introduce selected values of delay, jitter, and packet loss to test the performance of a network in different conditions. NS-3 allows users to simulate different types of connections such as Ethernet, 4G, 5G, etc. Thus, the designed HAV communication system can be tested under different network conditions to analyze its behavior in internet environments an end user might encounter. Although preliminary work simulating place holder dummy networks was done on NS-3, the designed HAV communication network was not simulated using NS-3 due to limited lab access introduced by COVID-19 restrictions (see Section 9 for further detail).

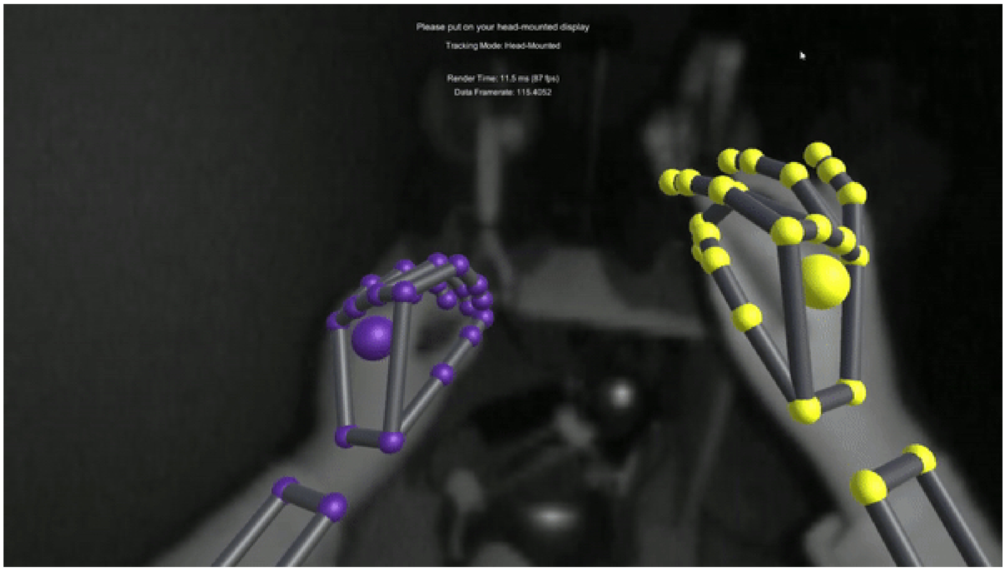
4.2 Application

4.2.1 Hand tracking

The goal here was to accurately track the hands in order to provide precise finger support. The parameters under test were:

- Position of the joints and knuckles
- Palm position
- Ring Finger position without the use of tools (No tool Occlusion)
- Ring Finger position while using tools (Possible tool Occlusion)

The native frame object obtained from Leap Motion (see Figure 8) was used to acquire the test data of the parameters listed above. Using this data, an iterative procedure was followed to find the optimal leap motion placement and orientation that ensures minimum occlusion and maximum data credibility. This means several experiments were run while changing the Leap Motion's position in order to get the optimum position that provides an accurate representation of the palm and ring finger in the environment.



Please put on your head-mounted display

Tracking Mode: Head-Mounted

Render Time: 11.5 ms (87 fps)

Data Framerate: 115.4052

Figure 8: Hand tracking using leap motion

4.2.2 Feedback from Dental Community

The dental training simulation, particularly the oral cavity model and the finger support, must be as realistic as possible. This was ensured by receiving periodic feedback from dental professors; primarily professors from NYU College of Dentistry (dental.nyu.edu).

In order to assess if the designed simulation accurately teaches the necessary skills, probing procedures were recorded as dental professors performed them on the oral cavity model. These recordings can then be used as a baseline to evaluate a dental student's performance. A survey can also be used to provide data regarding the ease of usage and learning experience of the students as they interact with the simulation. Though a dental probing procedure was recorded, the simulation was not tested with dental students and remains as a future plan.

5 Final Design

The complete design of this project is composed of two constituents; the HAV communication network, and the dental simulation end application.

5.1 Initial Network Design

Using the selected concepts from Section 3.3.1, the HAV communication network was designed using WebRTC and a locally hosted Node.js signaling server. Because both WebRTC and Node.js only include audio-visual communication for developmental flexibility, a haptic communication feature was added to create a haptic handshake protocol discussed in Section 5.1.1. Node.js signaling described in Section 5.1.2 is required to extend HAV communication from a localhost to multiple computer-based setup.

5.1.1 Haptic Handshake Protocol and Operation Design

HAV communication is initiated by a haptic handshake protocol shown in Figure 9. The protocol comprises of metadata communication and data communication, termed respectively as "Haptic Control State" and "Operation State".

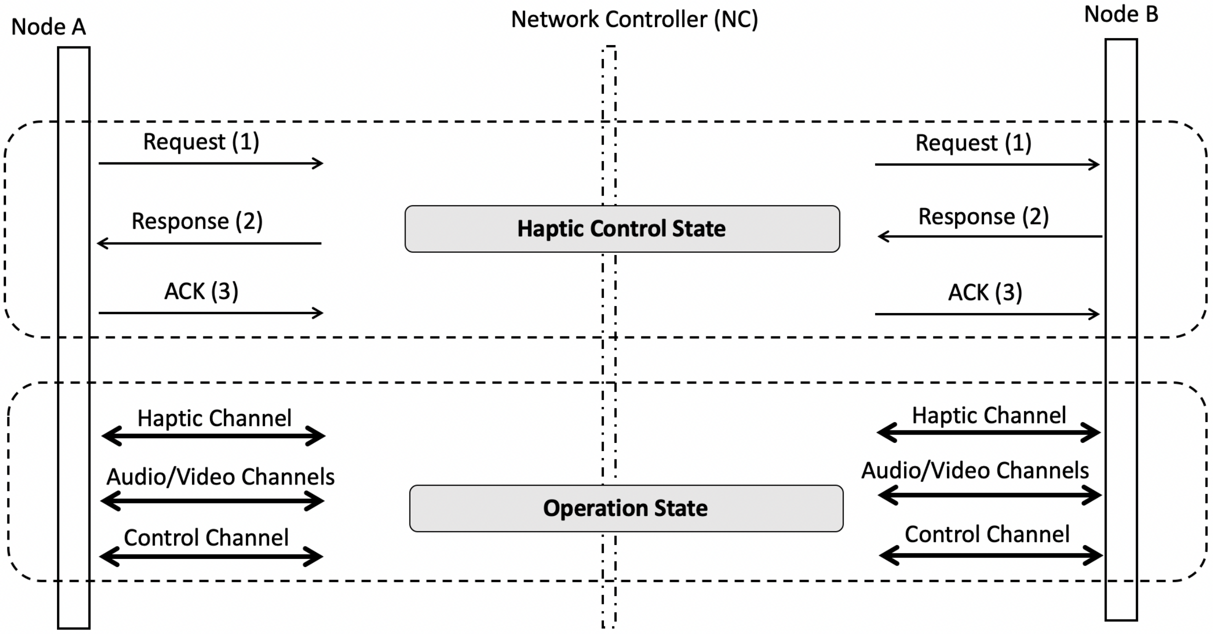


Figure 9: Schematic of the designed haptic handshake. 'Haptic Control State' involves a simple 3-way handshake consisting of request, response, and acknowledgement

In haptic control state, haptic devices exchange their metadata such as degree of freedom, and points of interaction. This is done by the local node 'A' sending a "request to connect" to remote node 'B'. The remote node then sends a response detailing its own specifications. Finally, the local node sends another message acknowledging the response and confirms the agreed upon specifications. This is followed by the operation state, commencing the bidirectional HAV data communication between the devices. In addition to data communication, the network keeps a control channel open in-case additional metadata need to be communicated during the operation. This enables the haptic devices to be switched to different models without the need to restart the communication.

The WebRTC-based implementation of the haptic handshake protocol is described in Section 7.1.4.

5.1.2 Signaling Server

As mentioned before, WebRTC requires a signaling server to establish communication between two peers. Hence, a server-client architecture was designed using the Socket.IO Node.js module. In this design, a Node.js application establishes a locally hosted server. Following this, the browser clients connect to the server and exchange their SDP (Session Description Protocol, a media description format for session announcement and invitation) in order to start the communication session. From here on, the communication is peer to peer (browser-to-browser) as shown in Figure 9.

5.2 Initial Application Design

The dental simulation was designed using a CHAI3D based oral cavity model and Leap Motion hand tracking for finger support. The experimental setup for the dental simulation, hereon referred to as Haptodont, is shown in Figure 10.

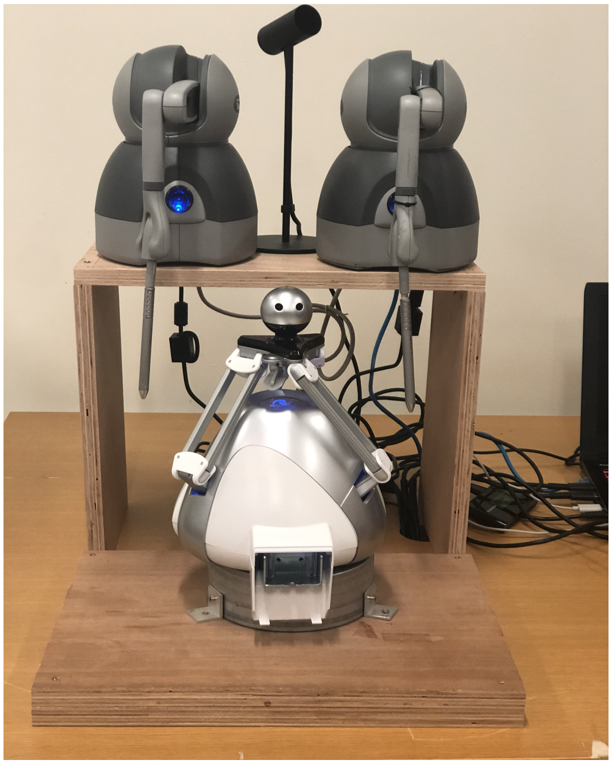


Figure 10: Haptodont Front View: Two *Geomagic Touch* haptic devices on top with *Oculus Rift Camera*, and *Novint Falcon* Finger Support on the Base.

5.2.1 3D Oral Cavity Model

A CHAI3D library-compatible oral cavity model was designed by integrating the ready-made models shown in Figure 11. The GEL library module in CHAI3D was used to introduce gel-like deformation of the cheeks in the oral cavity model. Hence, when a dental instructor or student interacts with the cheeks using the dental probe or mirror, they are able to see and feel the cheek stretch. This design makes the virtual simulation more realistic.

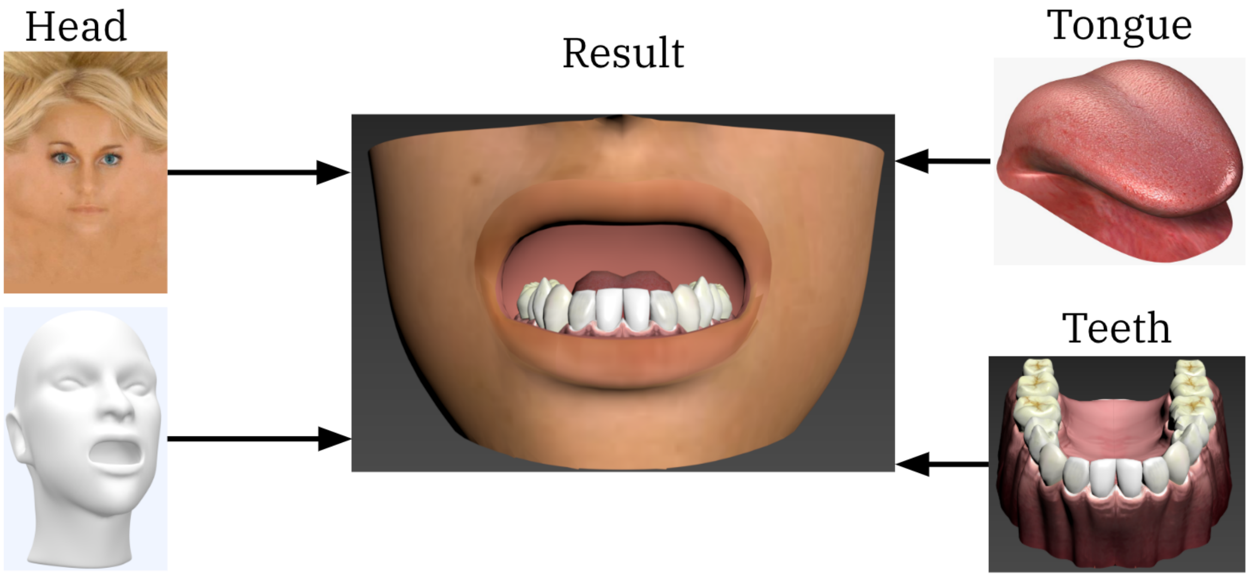


Figure 11: 3D Oral Cavity Model

5.2.2 Haptodont Application

The complete Haptodont application is composed of the 3D oral cavity model, dental probe and mirror model, and 3D buttons to change the orientation of the oral cavity model. The probe and mirror models (see Figure 12) are controlled by the styli of the *Geomagic Touch* devices. These models replicate the dental probe and mirror tools that are used by dentists. The mirror tool was modeled using a front camera to mimic a real mirror. The probe model has markings at its tip that indicate the depth of pockets in the gums. The oral cavity rests on a base that can be rotated using the interactive buttons in the simulation. This freedom of movement makes the Haptodont simulation ergonomic and natural.

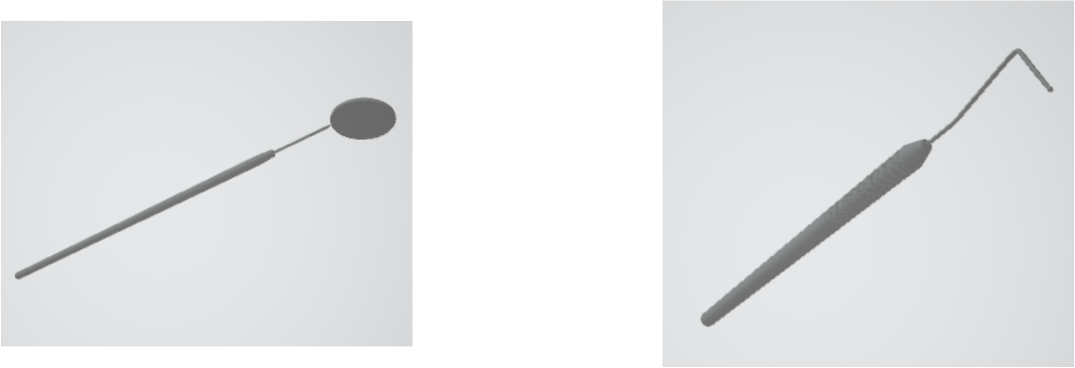


Figure 12: 3D model of mirror (left) and probe (right) used in the Haptodont simulation

5.2.3 Hand Tracking

A finger support was designed for a more realistic user-experience. This support simulates the dentists' technique of resting their ring finger on the chin or lower teeth of the patient. In order to determine the position of the ring finger within the Haptodont simulation, the user's hand was tracked using the Leap Motion device. Once this position was located, a *Novint Falcon* device was used to simulate finger support by providing haptic feedback at the last joint of the ring finger. The Leap Motion can track either hand of the user which means finger support can be provided whether the user is right or left handed.

5.2.4 Playback Integration

Haptic Playback is similar to video and audio playback whereby the student is able to replay the instructor's haptic interactions with the model. This gives the student the opportunity to learn the instructor's fine motor skills without the need for a one-to-one instruction session. Haptic Playback was implemented by recording the haptic packets in a file instead of sending them through the HAV communication network. This file can then be replayed as needed to observe the instructor's interaction with the Haptodont simulation.

5.3 Future Plan: Native WebRTC C++ integration

According to the Pugh Charts in Table 3, the choice made between the JavaScript (JS) based WebRTC platform and the Native C++ WebRTC platform comes down to the compromise between the complexity of using the API and the latency introduced by the API. One of the main issues with the current design of the JS based WebRTC network is the overhead caused by sending the haptic packets from a C++ environment to a JS environment. As further detailed in the implementation section 7.1, Websockets were used to transmit haptic data from C++ to JS environment. This introduced additional delays which could have been avoided with the use of the Native C++ WebRTC platform. However, the C++ platform was significantly more complex than the JS platform and hence it was decided to implement it in the future.

5.4 Design Review

With the integration of all the design elements mentioned above, the final design:

- provides a life-like virtual patient developed in immersive virtual reality.
- allows dental students to have realistic training experience on the Haptodont simulation.
- has a robust, low latency, and high reliability communication network that sends HAV data packets between two nodes.
- is easy-to-use and cost-effective (when compared to other simulators in the market).

6 Budget

The price of each device is listed below for reference. Note that the prices listed above were based on market price as of December 8, 2019. The Applied Interactive Media Lab had purchased these devices before the start of this capstone project.

- Two *Geomagic Touch* devices - \$1200x2 = \$2400 [10][11].
- One *Novint Falcon* device - \$200 [7].
- One *Oculus Rift* - \$400 [13]
- Two Leap Motion Controllers - \$150 [14] [22].

7 Implementation

7.1 Network: On Localhost

The implemented network includes several components: the haptic handshake, communication of data between haptic device and the C++ platform, the transfer of data between the C++ platform and JavaScript based communication platform, and the transmission of data within the JavaScript platform. The localhost implementation of the simulated HAV network is shown together in Figure 13. The sections below detail the current implementation of the HAV communication network.

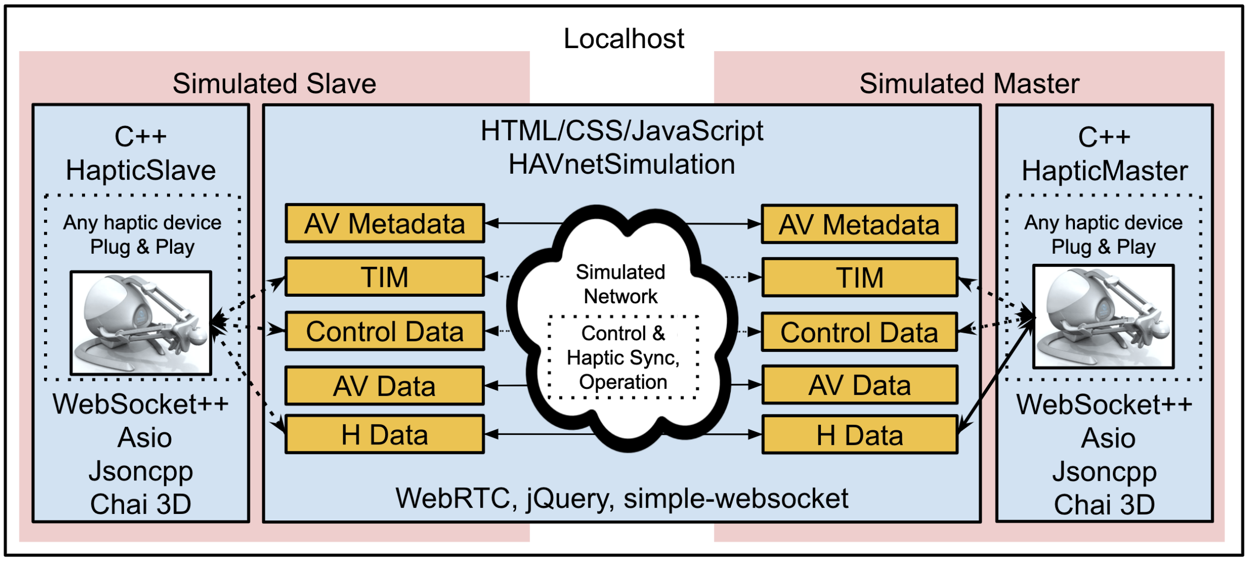


Figure 13: HAV Handshake Simulation Diagram

Ideally, the HAV network would be implemented in one program. However, the network was implemented using the JavaScript WebRTC API, while a C++ based CHAI3D application was used to access the haptic devices. Because of this, the implementation had to be divided into three programs: C++ HapticSlave, HTML/CSS/JavaScript HAVnetSimulation, and C++HapticMaster.

The HTML/CSS/JavaScript browser communicates haptic data between simulated slave and master. The haptic data is provided by the C++ programs that are interfaced to *Novint Falcon* haptic devices. The characteristics of haptic data, combined with AV data, is determined by AV metadata and TIM (Tactile Internet Metadata) which was communicated during the HAV handshaking phase. The attributes of the HAV data can also be renegotiated during operation phase by using the control data channel.

Several existing projects and libraries were merged to create a set of three programs. The browser consists of WebRTC and C++WebSocket Server Demo's client project. The client project consists of jQuery and simple-websocket. jQuery simplifies traditionally verbose JavaScript expressions while simple-websocket was used to receive websocket data. Both of the C++ programs (Master and Slave) consist of C++ WebSocket Server Demo's server project, Chai 3D, and Haptic Codec provided by Prof. Xiao Xu from TUM (shared only within AIM Lab). The server projects consists of WebSocket++, Asio, and Jsoncpp. WebSocket++ allows data to be sent on WebSocket via C++, Asio aids the networking process through Asynchronous Input/Outputs, and Jsoncpp allows creation and management of JSON objects in C++. Chai 3D was used to sense and actuate the *Novint Falcon* devices. Simplified version of HapticCodec was used to form haptic data packets that were then converted into stringified JSON via Jsoncpp and WebSocket++. The stringified JSON haptic data packets were then sent form the C++ Master/Slave environment to the JavaScript browser in real time using simple websockets. More implementation details and specific code snippets can be seen in the appendix (Section 12).

Once the haptic packets were sent from the C++ Master/Slave environment to the JS WebRTC environment, they were then communicated browser to browser using *RTCDataChannel*. First, a one-directional communication, where master haptic device sends position/velocity data and slave haptic device actuates, was implemented. The communication was then made bidirectional by sending force data from the slave device to the master device.

7.1.1 A WebRTC-based HAV Communication Model

Section 5.1.1 introduced a haptic handshake protocol subsuming WebRTC's existing AV communication. Figure 14 illustrates how this was implemented. Because of the WebRTC architecture, our implementation first sets up the AV side of handshake and communication, as denoted by the demarcated dashed lines. The haptic handshake and communication described in Section 5.1.1, shown in the lower right box, runs alongside with AV operation state. More implementation details and specific code snippets can be seen in the appendix (Section 12).

7.1.2 Haptic Handshake Protocol: Haptic Control State

The haptic control state model in Section 5.1.1 was implemented as shown in Figure 15.

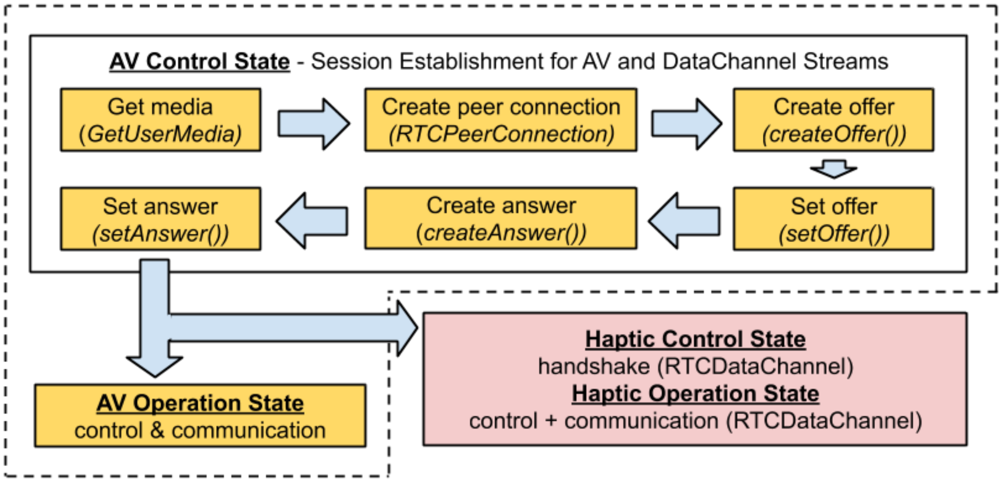

AV Control State - Session Establishment for AV and DataChannel Streams
Get media (GetUserMedia)
Create peer connection (RTCPeerConnection)
Create offer (createOffer())
Set offer (setOffer())
Create answer (createAnswer())
Set answer (setAnswer())
Haptic Control State
handshake (RTCDataChannel)
Haptic Operation State
control + communication (RTCDataChannel)
AV Operation State
control & communication

Figure 14: WebRTC-based HAV Communication Model

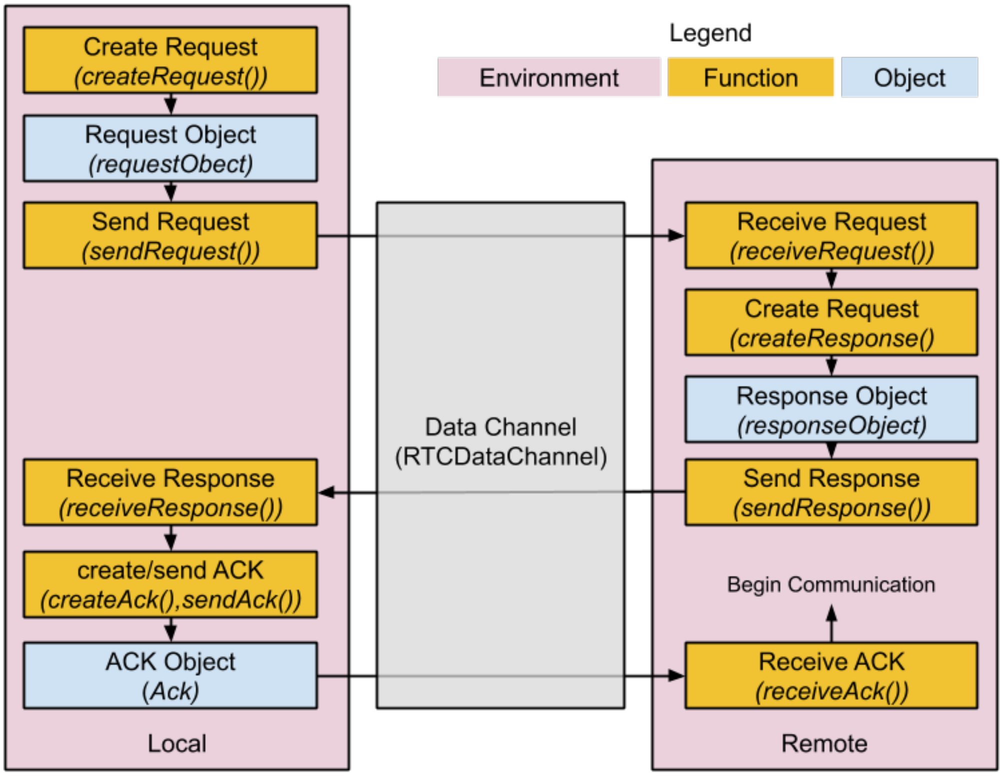

Legend
Environment
Function
Object
Create Request (createRequest())
Request Object (requestObect)
Send Request (sendRequest())
Receive Response (receiveResponse())
create/send ACK (createAck(),sendAck())
ACK Object (Ack)
Local
Data Channel (RTCDataChannel)
Receive Request (receiveRequest())
Create Request (createResponse()
Response Object (responseObject)
Send Response (sendResponse())
Begin Communication
Receive ACK (receiveAck())
Remote

Figure 15: Haptic Handshake Request/Response/ACK Model.

The *createRequest()* function creates a request object that contains local device-specific metadata such as maximum allowable latency, jitter, and degrees of freedom. The *sendRequest()* function sends the object to the remote device through *RTCDataChannel*. The remote device, based on its own metadata, modifies and sends the object back as a response object. Upon receival, the local device replies with an acknowledgement (or *Ack*) object, opening *RTCDataChannel* for HAV data communication.

7.1.3 Haptic Handshake Protocol: Request/Response/ACK Messages

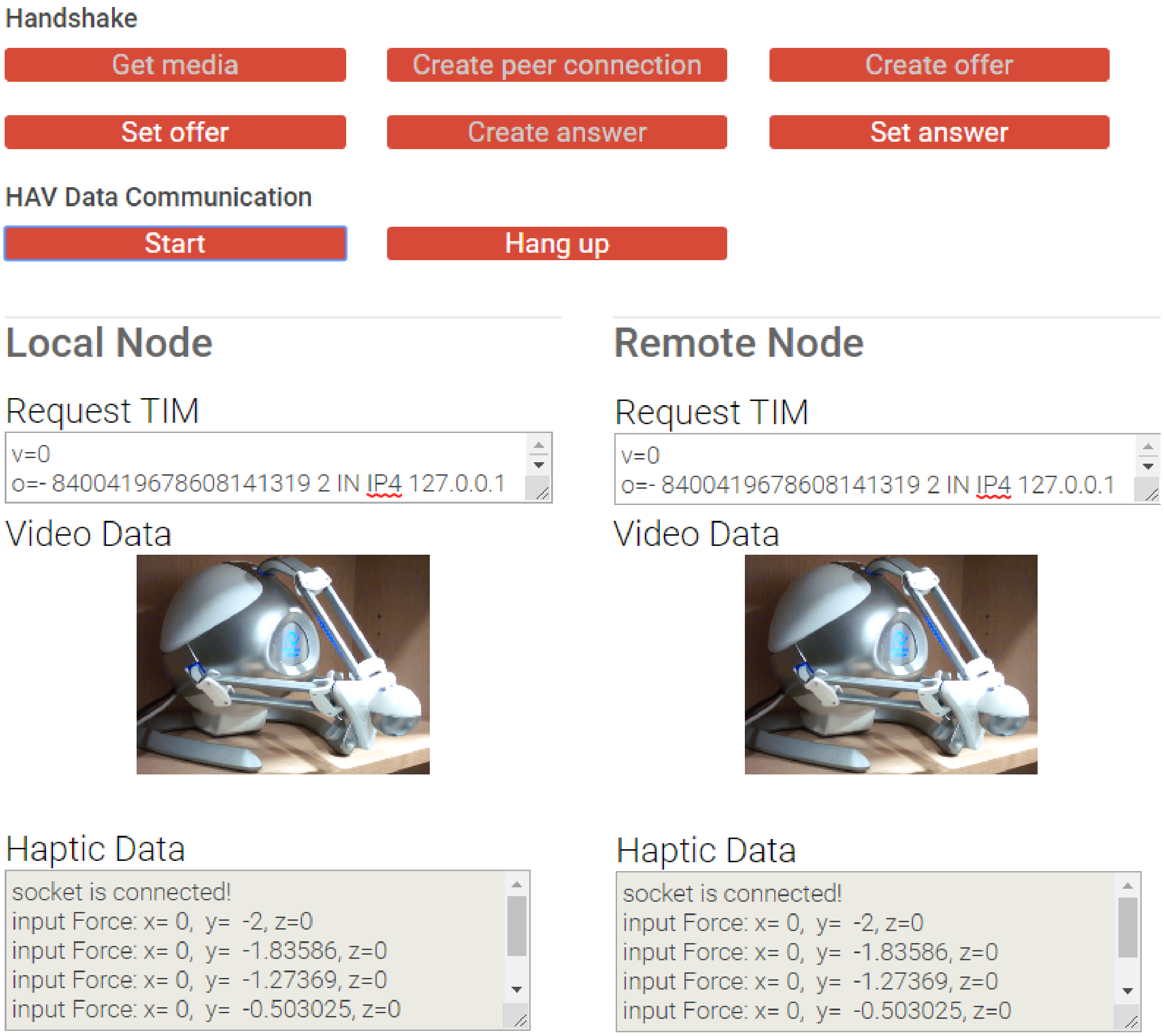
The request/response/Ack objects, shown in Figure 16, are based on IETF's text-based SDP for media session negotiation [20][33]. This allows easy and fast processing of the metadata.

```
{type: "request" or "response" or "ACK"
sessionDescription:
 "v=0
 o=- <timestamp> <sessionVersionCounter> IN IP4 <IPAddress>
 s=Haptic SDP
 i=SDP for Haptic Handshake
 t= 0 0
 a=<add attribute at the session level>"
mediaDescription:
 "m=haptic: <DeviceName> <portNumber> SCTP/DTLS HRTP 1
 i=Novint Falcon Haptic System
 a=QoS_hapLatency: <IntegerValue>
 a=QoS_hapJitter: <IntegerValue>
 a=QoS_hapReliability: <IntegerValue>
 a=UE_immersion: <Boolean 0 or1>
 a=UE_collabortation: <Boolean 0 or 1>
 a=UE_satisfaction: <Boolean 0 or 1>
 a=UE_presence:  <Boolean 0 or 1>
 a=Hap_Deadband: <Boolean 0 or 1>
 a=Hap_kinSampleRate: <IntegerValue>
 a=Hap_tacFequency: <IntegerValue>
 a=HapI_dof: <NaturalNumberValue>
 a=HapI_ws_x_y_z: <IntegerValueofx> <IntegerValueofy> <IntegerValueofz>
 a=HapI_fr_x_y_z: <IntegerValueofx> <IntegerValueofy> <IntegerValueofz>
 a=HapI_tr_x_y_z: <IntegerValueofx> <IntegerValueofy> <IntegerValueofz>
 a=UA_0001 (add custom attributes here...)"
CodecParams:
 "RecordSignals=0;                  // 0: Recording  off, 1: Recording on
 ForceDeadbandParameter=0.0;        // for force data reduction
 VelocityDeadbandParameter=0.0;     // for velocity data reduction
 PositionDeadbandParameter=0.0;     // for position data reduction
 ForceDelay=0;                      // Constant force network delay
 CommandDelay=0;                    // Command channel constant delay
 ControlMode=1;                     // 0: position, 1: velocity
 FlagVelocityKalmanFilter=0;        // 0: disabled, 1: enabled
 LocalIP=127.0.0.1;                 // local node
 RemoteIP=127.0.0.2;"               // remote node}
```

Figure 16: Template of Request/Response/ACK object inspired by the textual format of SDP.

7.1.4 Haptic Handshake Protocol: Browser Interface

The haptic handshake protocol is managed through a WebRTC-based browser menu, as shown in Figure 17. The functions described in Section 7.1.1 are controlled by corresponding buttons on the screen, and the request/response objects are shown below. The "Start" and "Hang Up" buttons initiates and terminates the HAV communication respectively.



Local Node — Request TIM:

```
v=0
o=- 8400419678608141319 2 IN IP4 127.0.0.1
```

Local Node — Haptic Data:

```
socket is connected!
input Force: x= 0,  y=  -2, z=0
input Force: x= 0,  y=  -1.83586, z=0
input Force: x= 0,  y=  -1.27369, z=0
input Force: x= 0,  y=  -0.503025, z=0
```

Remote Node — Request TIM:

```
v=0
o=- 8400419678608141319 2 IN IP4 127.0.0.1
```

Remote Node — Haptic Data:

```
socket is connected!
input Force: x= 0,  y=  -2, z=0
input Force: x= 0,  y=  -1.83586, z=0
input Force: x= 0,  y=  -1.27369, z=0
input Force: x= 0,  y=  -0.503025, z=0
```

Figure 17: GUI for Controlling HAV Handshake and Communication.

7.2 Issues faced during implementation

7.2.1 Application: Hand Tracking and Finger Support

Hand tracking was initially proposed to track the fingers and knuckles of the trainer's hand in order to obtain the position of the fingers in the environment. This position data would then be used by the finger support *Novint Falcon* device to simulate teeth supporting the fingers during a probing session. As mentioned in previous sections, a leap motion device was chosen as the hand tracking method.

The main issue faced during implementation of this procedure is occlusion. Accurate position of the joints and knuckles is needed for a successful performance of the finger support. However, tool occlusions (occlusions caused by the probing device) and self occlusions (occlusions caused by the hand itself, such as the palms and fingers) prevent the leap motion device from gaining accurate position information. To overcome this challenge, two leap motion devices were used in orthogonal fields of view to gain a better representation of the hand in the physical environment. Although this improved the performance of the hand tracking, it still didn't solve the problem of occlusions since no clear view of the target distal phalanges (bones at the tips of the fingers) was obtained. This uncertainty in the position of the target distal phalanges then caused instability of the finger support.

Tracking the fingers more accurately could be done using machine learning algorithms that make use of optimal camera/leap motion placement. However, implementing this was out of the scope of this design project. Hence, hand tracking was not implemented in this design project.

7.2.2 Application: Deformable Oral Cavity Model

The Chai 3D GEL library was explored to give the cheeks and tongues in the oral cavity model the attributes of gel-based materials. Although the library and its demo application were successfully integrated into the Haptodont application, further development seemed to have diminishing returns. This was due to the GEL library being computationally demanding and too complex to develop realistic deformation. Hence, this functionality was disabled in this implementation but can be turned on when needed.

7.3 Changes made during implementation

7.3.1 Network: Signaling Server-Client Communication Architecture

As discussed in section 5.1.2, the signaling server aids with session establishment by communicating SDP between the connected browsers. The architecture followed to implement this is shown in Figure 18. However, this architecture requires the use of an external server. Since the main goal of this project was to design a network while introducing minimal delays, the architecture seen in Figure 18 was unravelled and implemented on local host. This unravelled implementation can be seen in Figure 19. Here, three locally hosted servers are open simultaneously. The "Master Server" application interfaces with the haptic device, packetizes the haptic data (velocity and/or position), and sends the haptic packet to the "Master Client Browser" via a web-socket port (Port A). The "Slave Server" application is identical to the "Master Server" application, except that it sends force haptic packets and actuates velocity/position packets. When the packets reach the client browsers, they are recorded onto a file. This file is then read by the browsers ("Browser 1/2") on each node and the haptic packets are communicated peer-to-peer bidirectionally via *RTCDataChannel*. More implementation details and specific code snippets can be seen in the appendix (Section 12).

Ideally, "Browser 1" and "Browser 2" would be replaced by the Master and Slave client browsers. This would enable direct communication between the master and slave haptic devices without the need for record/play. However, this was not implemented as it required the use of an external signaling server that can establish a session between the "Master Client Browser" and the "Slave Client Browser".

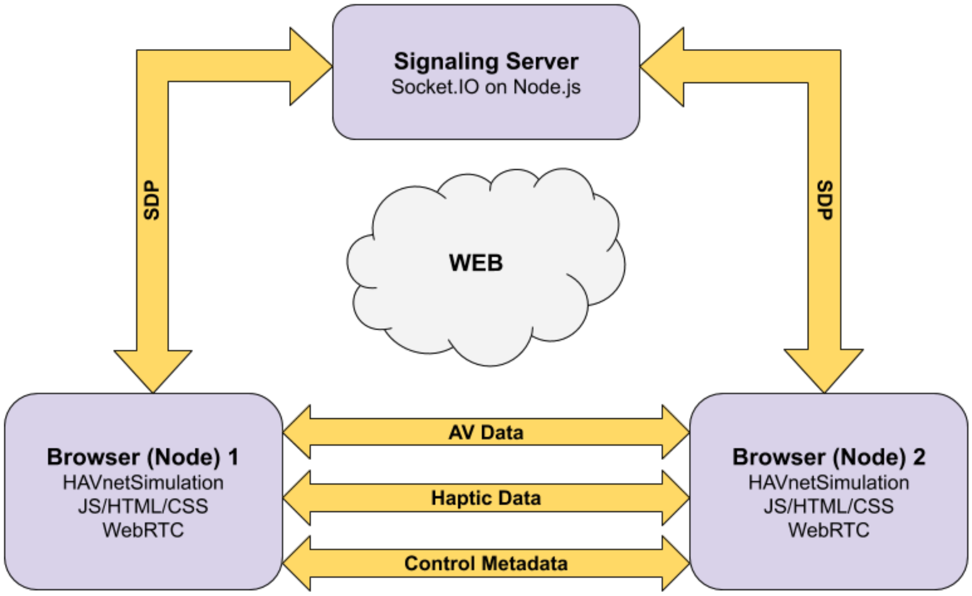


Figure 18: Signaling Network Architecture

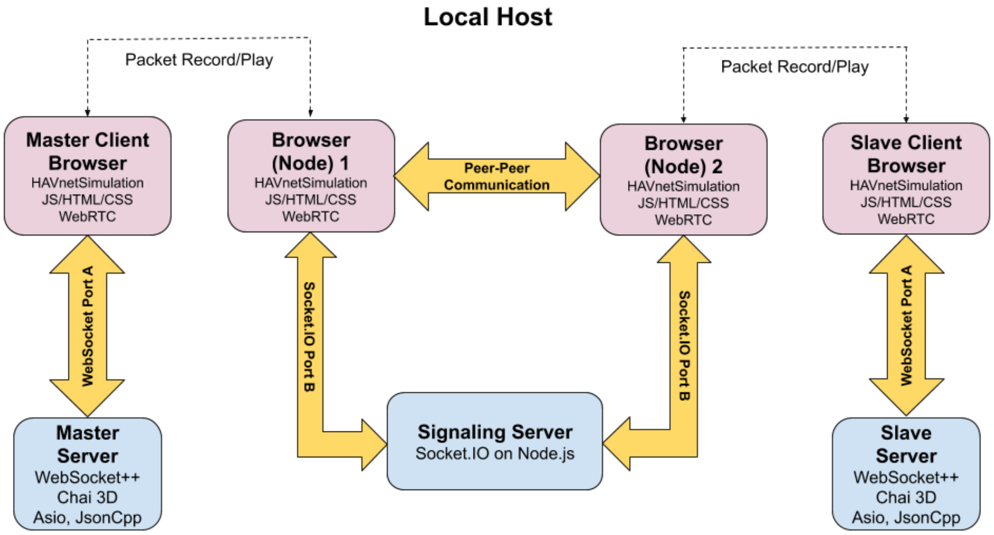


Figure 19: Unravelled Signaling Network Architecture

7.3.2 Application: Finger-Support

Since hand tracking for finger support was no longer being implemented, the means of obtaining finger position data changed. Instead of obtaining the position data from the Leap Motion tracking, the position of the fingers was deduced from the *Geomagic Touch* device that is used as a probing tool in the simulation. Although this method did not provide accurate position data, it still gave the orientation and relative position of the hand holding the probing device.

The finger support was then set to be 15 mm away from the tooth that is being probed in the physical space. A buffer of 5 mm was used to trigger the support; "move" command was dispatched only when the probe moves more than 5mm from its relative position or orientation. This implementation needs to be more realistic and dynamic in future iterations. For example, the fixed 15 mm distance in the physical environment does not allow flexibility when it comes to differing hand sizes.

7.4 Initial Results

Based on simple networking statistics, the mean latency introduced by the TIM handshake implementation was measured to be 47.25 ms, with 23.38 ms standard deviation. This is well below the threshold required by the technical constraints.

7.5 Video Demonstration of Completed Design

The following link demonstrates the Haptodont simulation with CHAI3D oral cavity model, *Oculus Rift*, and haptic devices in one setup : https://youtu.be/jstlJ6TwCEA

8 Contribution to IEEE 1918.1.1 Working Group (WG)

The IEEE 1918.1 working group (WG) aims to envision and standardize various modules crucial for the realization of the Tactile Internet (TI). Our capstone have been contributing to this effort as part of their TIM handshake and HAV data communication standardization effort. The haptic handshake protocol was presented to WG on Summer 2019, and then presented to the 2019 IEEE International Symposium on Haptic, Audio and Visual Environments and Games (HAVE) in October 2019 [21]. Figure 13 was presented to WG early March to update on the progress of HAV network implementation. This was the second time to progress was presented, the first one being on August 2019. WG members expressed high satisfaction in the progress, and are currently hoping to obtain a completed implementation by the end of May for validation. Documentation for the HAV network implementation is attached in the Appendix (Section 12).

9 Impact of COVID-19 on the Capstone Design Project

9.1 Network

9.1.1 Network Simulation

NS-3 is a Linux based Network Simulator. Due to limited access to the lab, a Linux machine with NS-3 installed was not acquired (in a timely manner). Hence, network simulation of the delay and jitter introduced by the internet was not conducted.

9.2 Application

9.2.1 Integration of Haptodont End-Application and Communication Network

Although the Haptodont end-application and the communication network for HAV data was completed, these two pieces were not integrated to each other due to limited access to the lab. The Haptodont setup is located in the lab and our work on that setup was completed before COVID-19 restrictions. Our work on the communication Network was completed after COVID-19 restrictions, but this implementation could not be integrated with the Haptodont due to limited access to the lab.

9.2.2 Application Testing

An important criteria for the design evaluation of the Haptodont application was to introduce the setup to the dental community and to collect feedback from the instructors, students and dental experts. The probing performance and task completion time of the Haptodont setup needed to be determined by comparing it to the student's performance on a real setup. However, due to COVID-19 restrictions and physical distancing, testing with the dental community could not be carried out.

10 Ethics

The following ethical considerations were taken into account while developing the proposed HAV Tele-Dental Training Simulation.

- Network Security - The system is highly reliant on the internet, which makes it prone to hacking attacks that could endanger end users. These can be alleviated by implementing end to end encryption. Some examples of security threats are:

  1) Sending unstable force feedback with the intent to harm end users.
  2) Hacking the system and tempering with the grading system.
  3) Illegally distributing an expert's recorded motor skills.

- Device Safety - Stability of the haptic devices must be ensured at all times as unstable force feedback could injure the end users. Stability was ensured by implementing a threshold for the maximum allowable force feedback.

- Privacy (Confidentiality) - Users must give an informed consent to have their audio, video and haptic interactions recorded. This can be implemented by asking the users to read and sign a release form if they agree to having their data recorded.

11 Design Evaluation

11.1 Criteria for Design Evaluation

1) Communication Network Performance [18]

   The network should not exhibit delays that are noticeable by the human end users. These delays are listed in section 2.4.1. Additionally, the maximum delay allowed by the network should be lower than the maximum delay perceived by humans so that some time is used as a buffer in-case of packet loss. The network used to test for these criteria should be 4G or higher generations.

   a) Total end to end delay of less than 50 ms for haptic data.
   b) Total end to end delay of less than 200 ms for visual data.
   c) Total end to end delay of less than 300 ms for auditory data.
   d) Jitter of haptic media (Packet Delay Variation) of less than 15 ms.

2) Expert Feedback

   a) Realism: A rating of more than 85% from dental professionals and field experts on how closely this system emulates real setups.
   b) Immersion: A rating of more than 85% from field experts.

3) Probing Accuracy

   a) Probing Performance: There should be no significant difference ($p < 0.05$) between the performance of students trained on this system versus those trained on a real setup.

4) Task Completion Time.

   a) There should be no significant difference ($p < 0.05$) between the time taken by students trained on this system versus those trained on a real setup while preforming identical procedures.

11.2 Results and Test Data

11.2.1 Network

WebRTC-internals was used to collect statistics about ongoing WebRTC sessions [16]. The round trip time (RTT) of the audio-visual data over a localhost was obtained from WebRTC-internals. In order to measure the delay and jitter in the communication of haptic

data, timestamps were added just before the hapitc data was sent from the local node and when it was received at the remote node. The mean delay was measured to be 0.62 ms and the jitter was found to be 0.53 ms.
Figure 20 shows the delay in the communication of a haptic packet via *RTCDataChannel*.

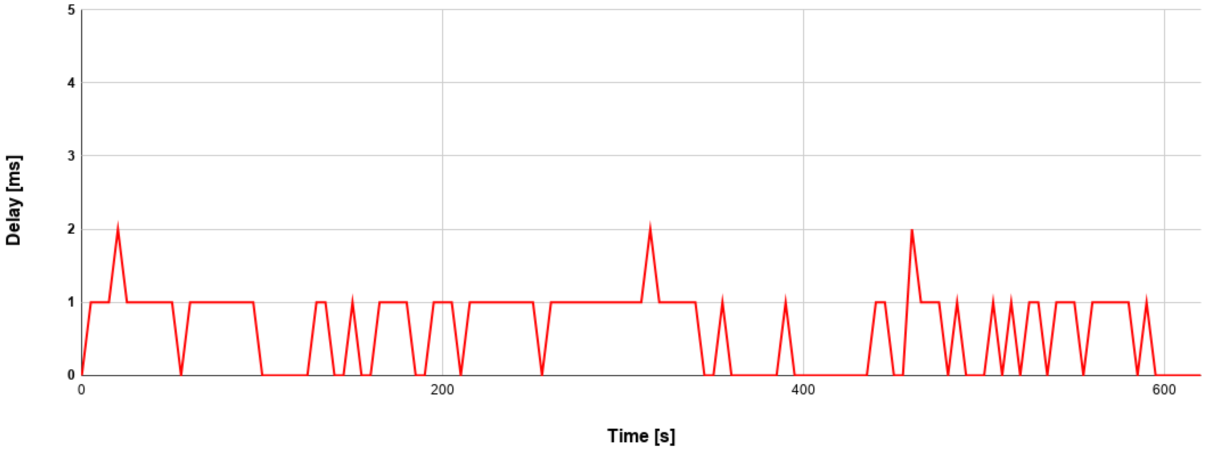


Figure 20: Delay in the communication of haptic packets over time

Figures 21 and 22 show the round trip time of the communication of audio-visual data using WebRTC.

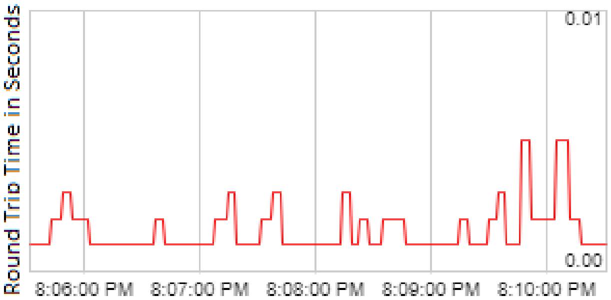


Figure 21: Round trip time of audio data communication

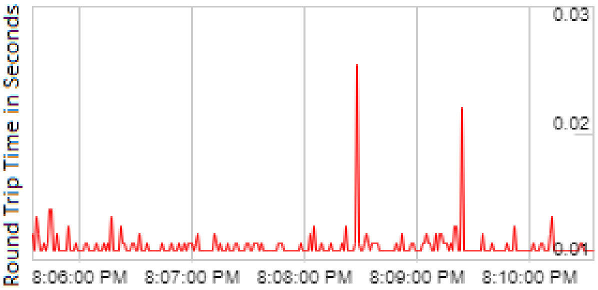


Figure 22: Round trip time of visual data communication

Figures 23 and 24 show packets and bytes (sent and received), jitter and packet loss for audio-visual data streamed using WebRTC. These statistics were collected from WebRTC-internals.

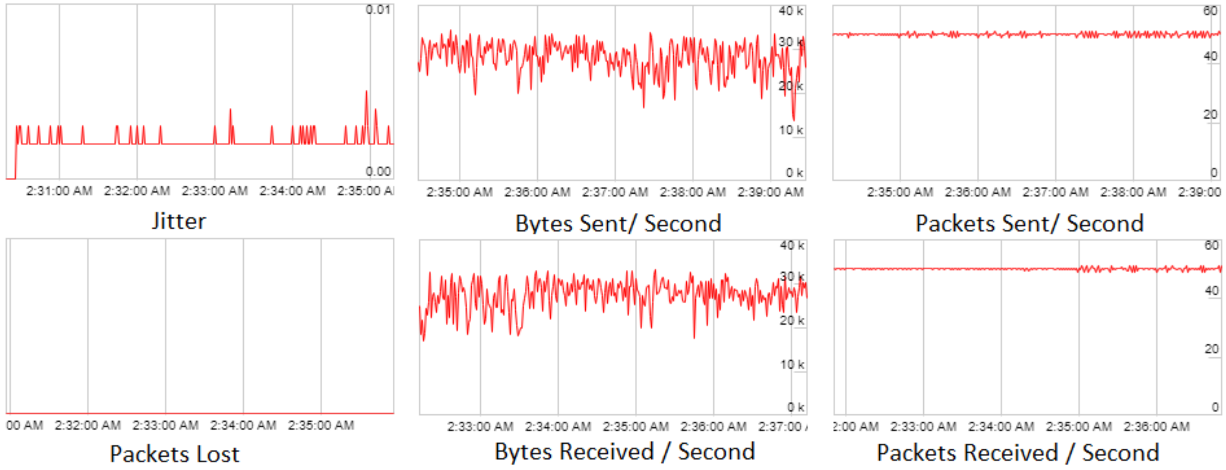


Figure 23: Packets and bytes sent and received (per second), jitter, and packet loss for audio data

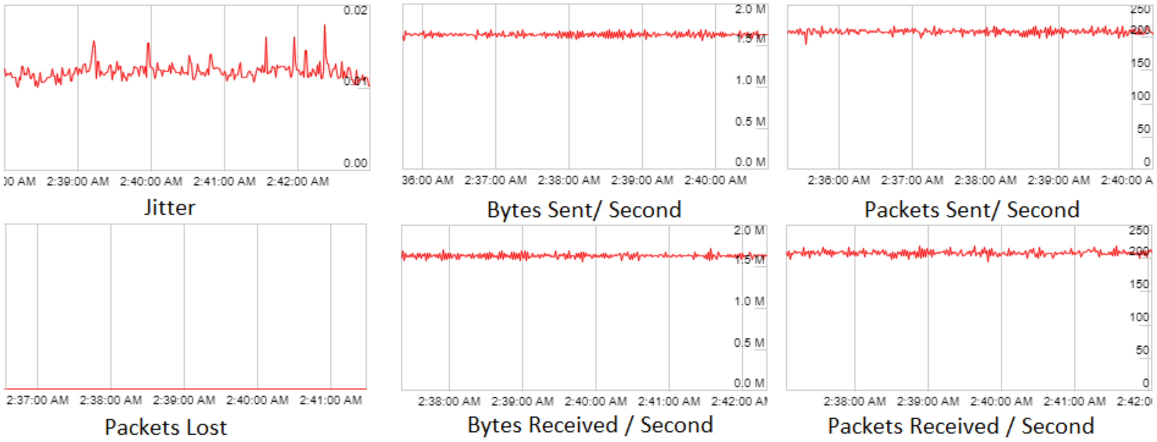


Figure 24: Packets and bytes sent and received (per second), jitter, and packet loss for video data)

In order to obtain the expert feedback, probing accuracy, and task completion time, the setup was to be introduced to the dental community. However, COVID-19 restrictions made it impossible to interact with and receive feedback from the dental community (i.e. students, professors, and experts). Refer to Section 9 for further details.

11.3 Discussion of Test Data

11.3.1 Network

- The measured average haptic delay was significantly smaller than the allowable haptic delay. The standard deviation of the delay accounted for haptic jitter that was much smaller than allowable jitter.

- Unlike audio-visual data, WebRTC-internals does not provide any statistics for the delay in communicating data using the *RTCDataChannels*. Thus, timestamps were used to find the delay and jitter in haptic data.

- The round trip time (RTT) is the time taken to send audio-visual data (from one end) and to receive an acknowledgement of this data when it is received on the other end. It can be deduced that end-to-end delay (time taken for the audio-visual data to be received on the other end) is half the round-trip-time.

- Figure 21 and 22 show that the delay of the data- when communicated over the localhost- was significantly smaller than the allowable end to end delay mentioned above.

- Figures 23 and 24 show that there was no packet loss and minimal jitter as the audio-visual data was communicated over a localhost.

- The received measurements ensured that this system can be plugged into real networks without introducing considerable overhead in delay.

References

[1] [Online]. Available: https://tools.ietf.org/rfc/rfc768.txt

[2] [Online]. Available: https://firebase.google.com/

[3] [Online]. Available: https://www.turbosquid.com/Search/Index.cfm?keyword=oral+cavity

[4] [Online]. Available: https://www.chai3d.org/

[5] [Online]. Available: https://learn.sparkfun.com/tutorials/leap-motion-teardown

[6] "Falcon specifications." [Online]. Available: https://hapticshouse.com/pages/falcon-specifications

[7] "Falcons." [Online]. Available: https://hapticshouse.com/collections/falcons

[8] "Features: 3d systems." [Online]. Available: https://www.3dsystems.com/haptics-devices/3d-systems-phantom-premium/features

[9] "Free 3d models." [Online]. Available: https://free3d.com/3d-models/3ds-max

[10] "Geo-magic touch website." [Online]. Available: https://www.3dsystems.com/haptics-devices/touch

[11] "Geomagic touch x - professional haptic device providing force feedback." [Online]. Available: https://www.or3d.co.uk/products/hardware/haptic-devices/geomagic-touch/

[12] "Maximize your digital performance gain a competitive edge." [Online]. Available: https://www.riverbed.com/mena/

[13] "Oculus rift s." [Online]. Available: https://www.oculus.com/rift-s/

[14] "Where to buy in the us." [Online]. Available: https://www.leapmotion.com/where-to-buy/us/

[15] "Openhaptics," Aug 2018. [Online]. Available: https://www.3dsystems.com/haptics-devices/openhaptics

[16] "webrtc-internals," Jan 2019. [Online]. Available: https://webrtcglossary.com/webrtc-internals/

[17] K. Antonakoglou, X. Xu, E. Steinbach, T. Mahmoodi, and M. Dohler, "Toward haptic communications over the 5g tactile internet," *IEEE Communications Surveys Tutorials*, vol. 20, no. 4, pp. 3034–3059, Fourthquarter 2018.

[18] M. Eid, J. Cha, and A. El Saddik, "Admux: An adaptive multiplexer for haptic–audio–visual data communication," *IEEE Transactions on Instrumentation and Measurement*, vol. 60, no. 1, pp. 21–31, 2010.

[19] N. Foundation. [Online]. Available: https://nodejs.org/en/

[20] M. Handley, V. Jacobson, C. Perkins, *et al.*, "Sdp: session description protocol," 1998.

[21] K. Iiyoshi, M. Tauseef, R. Gebremedhin, V. Gokhale, and M. Eid, "Towards standardization of haptic handshake for tactile internet: A webrtc-based implementation," in *2019 IEEE International Symposium on Haptic, Audio and Visual Environments and Games (HAVE)*, 2019, pp. 1–6.

[22] A. Industries, "Leap motion controller with sdk." [Online]. Available: https://www.adafruit.com/product/2106

[23] R. Jagsi and L. S. Lehmann, "The ethics of medical education," *Bmj*, vol. 329, no. 7461, pp. 332–334, 2004.

[24] A. Khana, S. Bilalb, and M. Othmana, "A performance comparison of network simulators for wireless networks," in *Proc. IEEE Int. Conf. on Control System, Computing and Engineering (ICCSCE)*, 2012.

[25] V. Kravtchenko, "Tracking color objects in real time," Ph.D. dissertation, University of British Columbia, 1999.

[26] S. Loreto and S. P. Romano, "Real-time communications in the web: Issues, achievements, and ongoing standardization efforts," *IEEE Internet Computing*, vol. 16, no. 5, pp. 68–73, Sep. 2012.

[27] O.-K. G. Ogot, Madara, *Engineering Design: A Practical Guide*, 2007.

[28] J. Palmius, J. Palmius, and J. Palmius. [Online]. Available: http://www.makehumancommunity.org/

[29] D. M. Popovici, F. G. Hamza-Lup, A. Seitan, and C. M. Bogdan, "Comparative study of apis and frameworks for haptic application development," in *2012 International Conference on Cyberworlds*. IEEE, 2012, pp. 37–44.

[30] I. H. Randoll and R. C. McShirley, "Typodont having removable teeth," Jan. 6 1981, uS Patent 4,242,812.

[31] J. F. Ranta and W. A. Aviles, "The virtual reality dental training system: simulating dental procedures for the purpose of training dental students using haptics," in *Proceedings of the fourth PHANTOM users group workshop*, vol. 4. November, 1999, pp. 67–71.

[32] K. R. Rosen, "The history of medical simulation," *Journal of critical care*, vol. 23, no. 2, pp. 157–166, 2008.

[33] J. Rosenberg and H. Schulzrinne, "An offer/answer model with session description protocol (sdp)," 2002.

[34] M. Series, "Imt vision–framework and overall objectives of the future development of imt for 2020 and beyond," *Recommendation ITU*, pp. 2083–0, 2015.

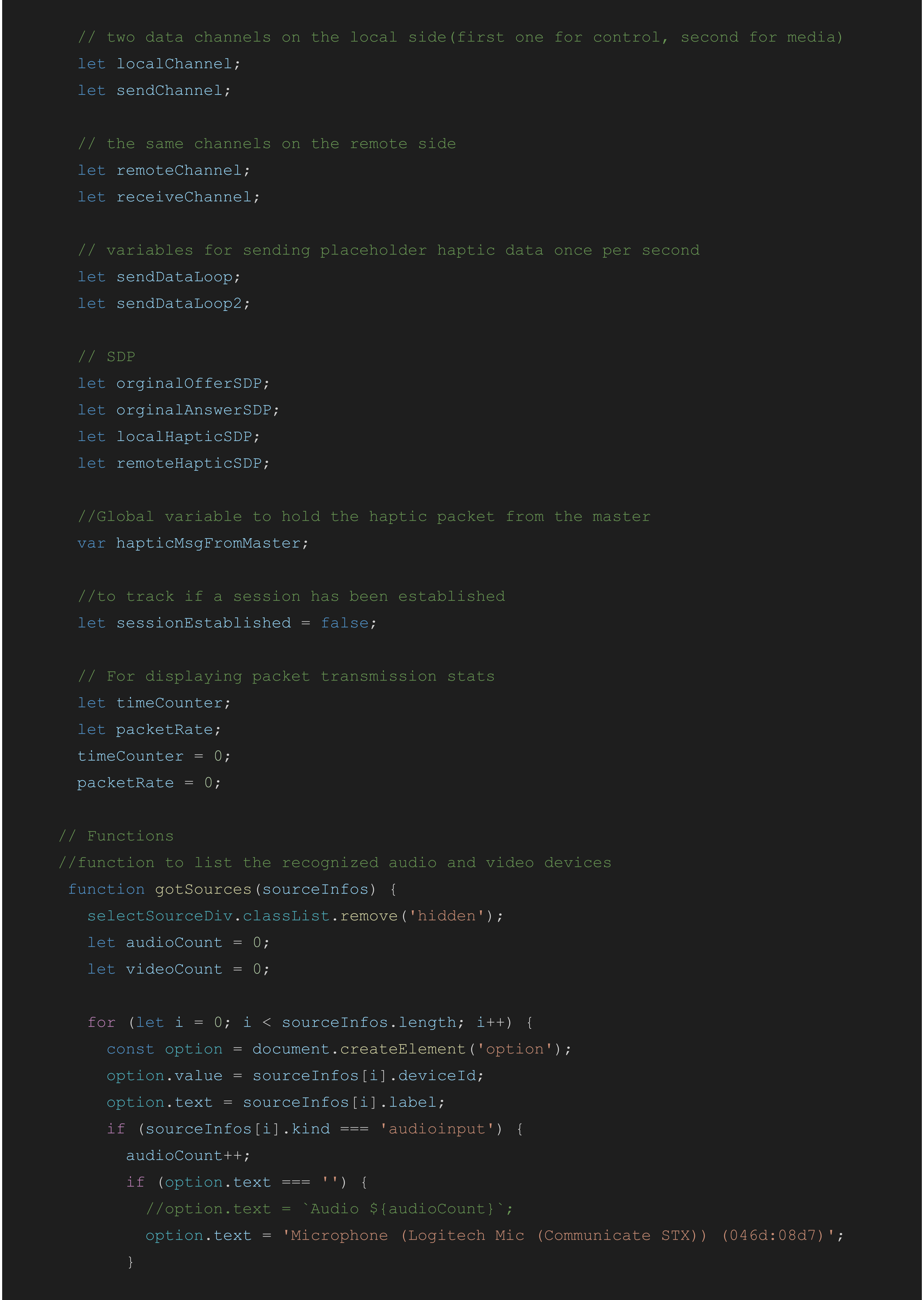
[35] D. Wang, T. Li, Y. Zhang, and J. Hou, "Survey on multisensory feedback virtual reality dental training systems," *European Journal of Dental Education*, vol. 20, no. 4, pp. 248–260, 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/eje.12173

[36] M. Weinberg, C. Westphal, S. Froum, M. Palat, and R. Schoor, *Comprehensive periodontics for the dental hygienist*. Pearson Higher Ed, 2014.

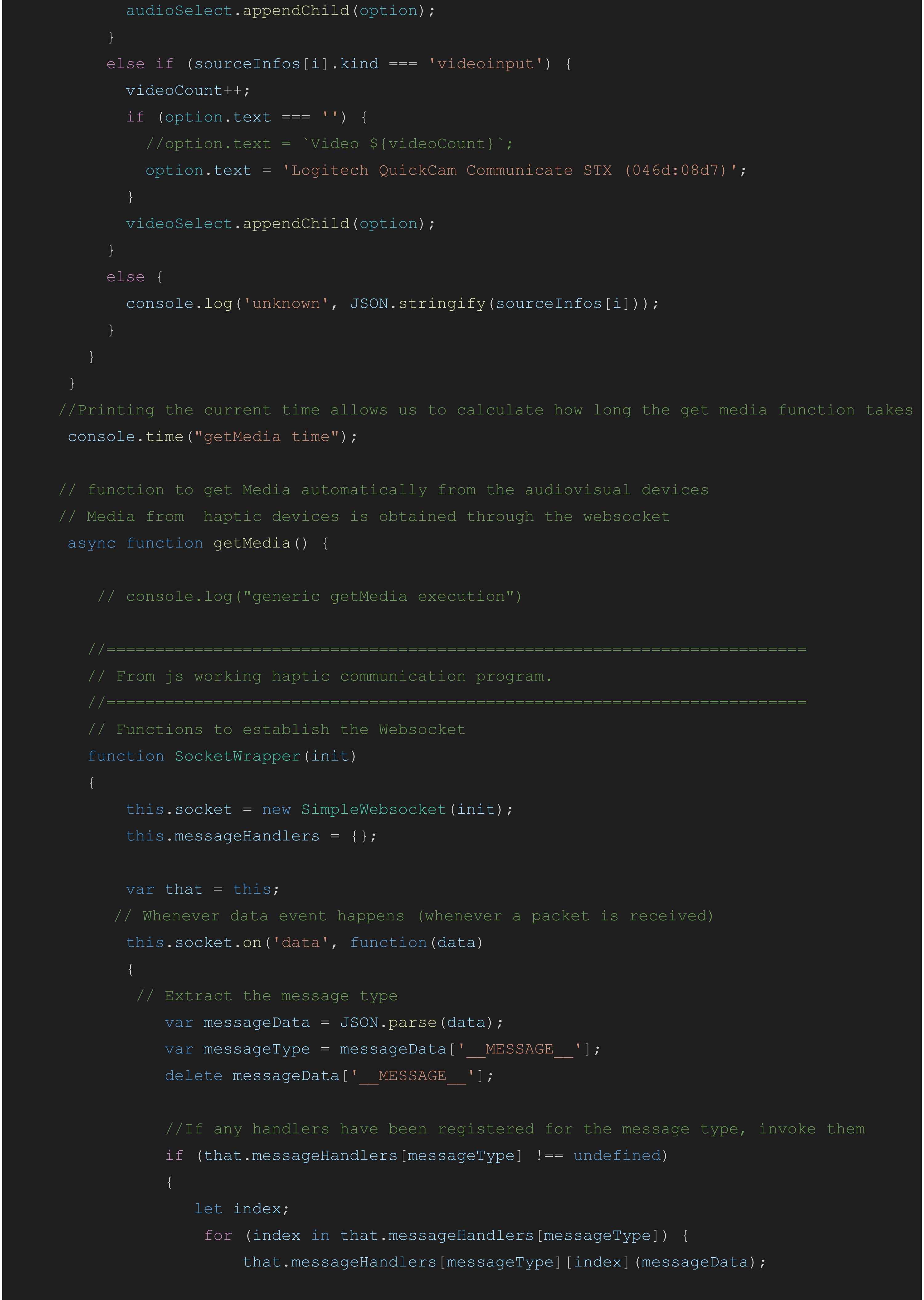
12 Appendix

12.1 Browser Code (WebRTC and Websockets)

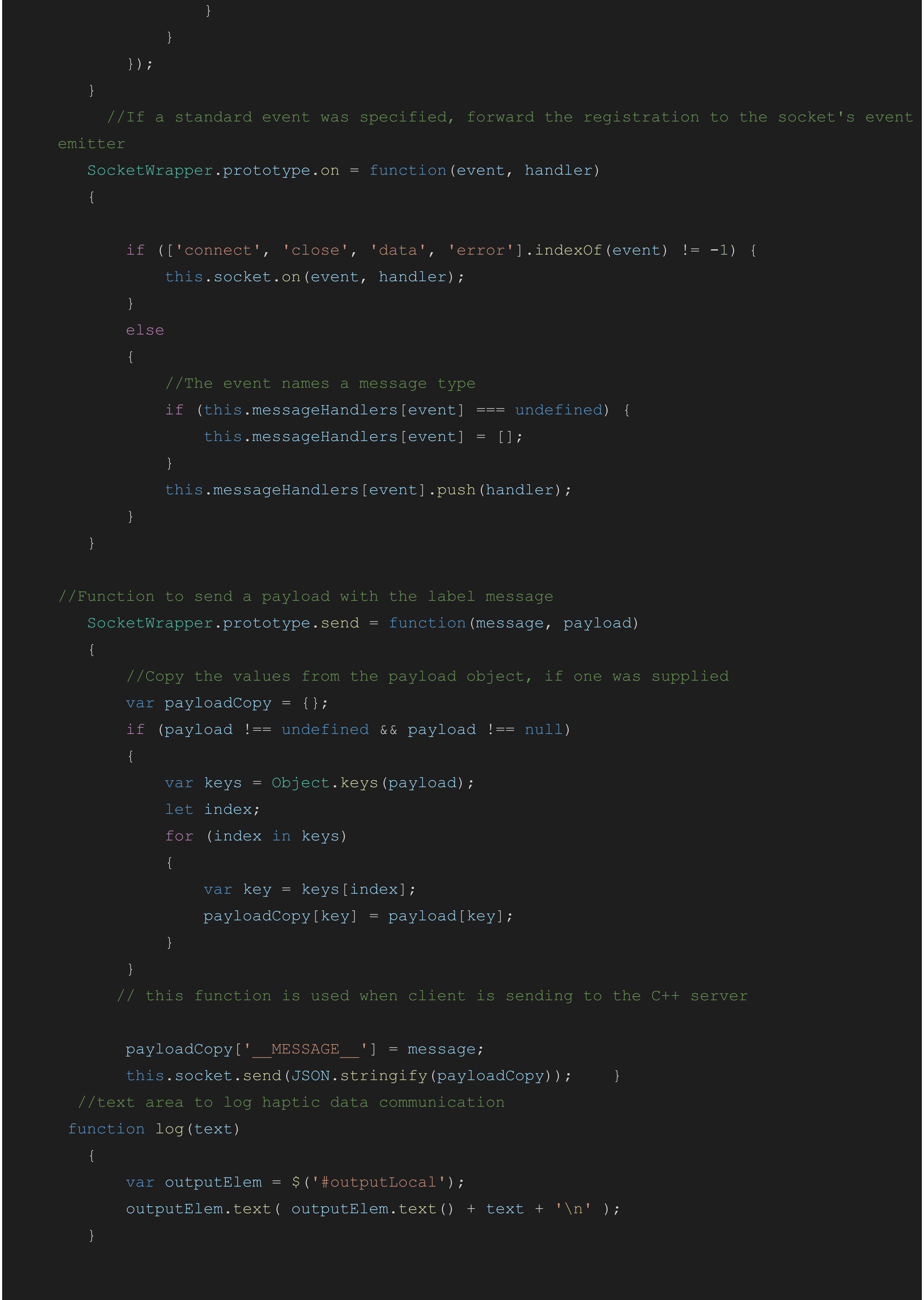
```
/*
*  This project was inspired by the Munge SDP sample example which can
*  be found on the WebRTC website. The copyright from that project can
*  be seen below.
      *  Copyright (c) 2015 The WebRTC project authors. All Rights Reserved.
      *
      *  Use of this source code is governed by a BSD-style license
      *  that can be found in the LICENSE file in the root of the source
      *  tree.
* Authors
   Ken Iiyoshi
   Mahrukh Tauseef
   Ruth Gebremedhin
* Date
   Sun, 10 May 2020
*/
//==========================================================================

// use strict mode. i.e. one cannot use undeclared variables.
'use strict';
document.addEventListener('DOMContentLoaded', init);

console.time("init time");
async function init() {

  // for hiding source selection when needed.
  const selectSourceDiv = document.querySelector('div#selectSource');

  // to choose which source of HAV to use.
  const audioSelect = document.querySelector('select#audioSrc');
  const videoSelect = document.querySelector('select#videoSrc');
  const hapticSelect = document.querySelector('select#hapticSrc');

  // to have options drop down
  hapticSelect.options[hapticSelect.options.length] = new Option('NovintFalcon');
  hapticSelect.options[hapticSelect.options.length] = new Option('GeoMagicTouch');

  console.log('DOMContentLoaded');
  try {
    const enumerateDevices = await navigator.mediaDevices.enumerateDevices();
    gotSources(enumerateDevices); // adds the devices discovered by enumerate Device to
the object for selecting which video and audio source to use.
  }
  catch (e) {
    console.log(e);
  }

  // Buttons to run the functions
  const getMediaButton = document.querySelector('button#getMedia');
  const createPeerConnectionButton =
document.querySelector('button#createPeerConnection');
  const createOfferButton = document.querySelector('button#createOffer');
  const setOfferButton = document.querySelector('button#setOffer');
  const createAnswerButton = document.querySelector('button#createAnswer');
  const setAnswerButton = document.querySelector('button#setAnswer');
  const hangupButton = document.querySelector('button#hangup');
  const handShakeButton = document.querySelector('button#StartHandshake');

  //Text areas
  const offerSdpTextarea = document.querySelector('div#local textarea');
  const answerSdpTextarea = document.querySelector('div#remote textarea');

  //Video areas
  const localVideo = document.querySelector('div#local video');
  const remoteVideo = document.querySelector('div#remote video');

  //Which functions to run when the buttons are pressed
  getMediaButton.onclick = getMedia;
  createPeerConnectionButton.onclick = createPeerConnection;
  createOfferButton.onclick = createOffer;
  setOfferButton.onclick = setOffer;
  createAnswerButton.onclick = createAnswer;
  setAnswerButton.onclick = setAnswer;
  hangupButton.onclick = hangup;
  handShakeButton.onclick = sendFromMe;

  //getMedia when the audio, video, haptic source is changed
//This ensures that a session is renegotiated if a device has changed as the metadata
would change

  audioSelect.onchange = videoSelect.onchange = hapticSelect.onchange = getMedia;

  //Constants to configure the two data channels
  //Since ordered is set to true by default, this means choosing
  //a reliable method of communication
  //For UDP set ordered to false and maxRetransmits to 0.
  //can control maxRetransmits or maxPacketLifeTime attributes but not both
  //const dataChannelOptions = { [[Ordered]], [[MaxPacketLifeTime]],
  //[[MaxRetransmits]], [[DataChannelProtocol]], [[DataChannelId]]};
  const dataChannelOptions = {Ordered: false, MaxRetransmits:0, negotiated: true, id:
0}; //for non-dynamic

  // Variables
  let localStream;
  let localPeerConnection;
  let remotePeerConnection;
```



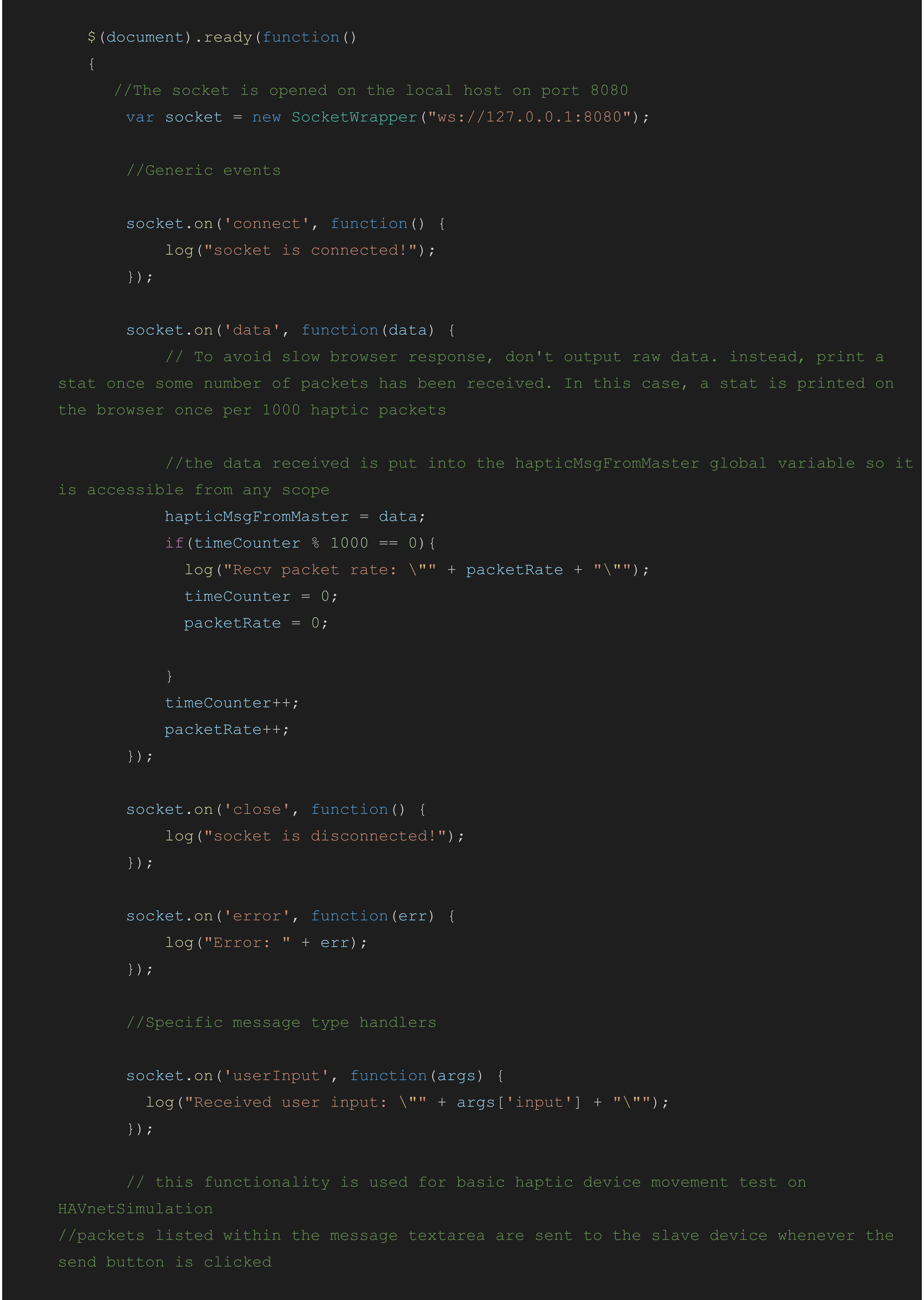
```
  // two data channels on the local side(first one for control, second for media)
  let localChannel;
  let sendChannel;

  // the same channels on the remote side
  let remoteChannel;
  let receiveChannel;

  // variables for sending placeholder haptic data once per second
  let sendDataLoop;
  let sendDataLoop2;

  // SDP
  let orginalOfferSDP;
  let orginalAnswerSDP;
  let localHapticSDP;
  let remoteHapticSDP;

  //Global variable to hold the haptic packet from the master
  var hapticMsgFromMaster;

  //to track if a session has been established
  let sessionEstablished = false;

  // For displaying packet transmission stats
  let timeCounter;
  let packetRate;
  timeCounter = 0;
  packetRate = 0;

// Functions
//function to list the recognized audio and video devices
 function gotSources(sourceInfos) {
   selectSourceDiv.classList.remove('hidden');
   let audioCount = 0;
   let videoCount = 0;

   for (let i = 0; i < sourceInfos.length; i++) {
     const option = document.createElement('option');
     option.value = sourceInfos[i].deviceId;
     option.text = sourceInfos[i].label;
     if (sourceInfos[i].kind === 'audioinput') {
       audioCount++;
       if (option.text === '') {
         //option.text = `Audio ${audioCount}`;
         option.text = 'Microphone (Logitech Mic (Communicate STX)) (046d:08d7)';
       }
```



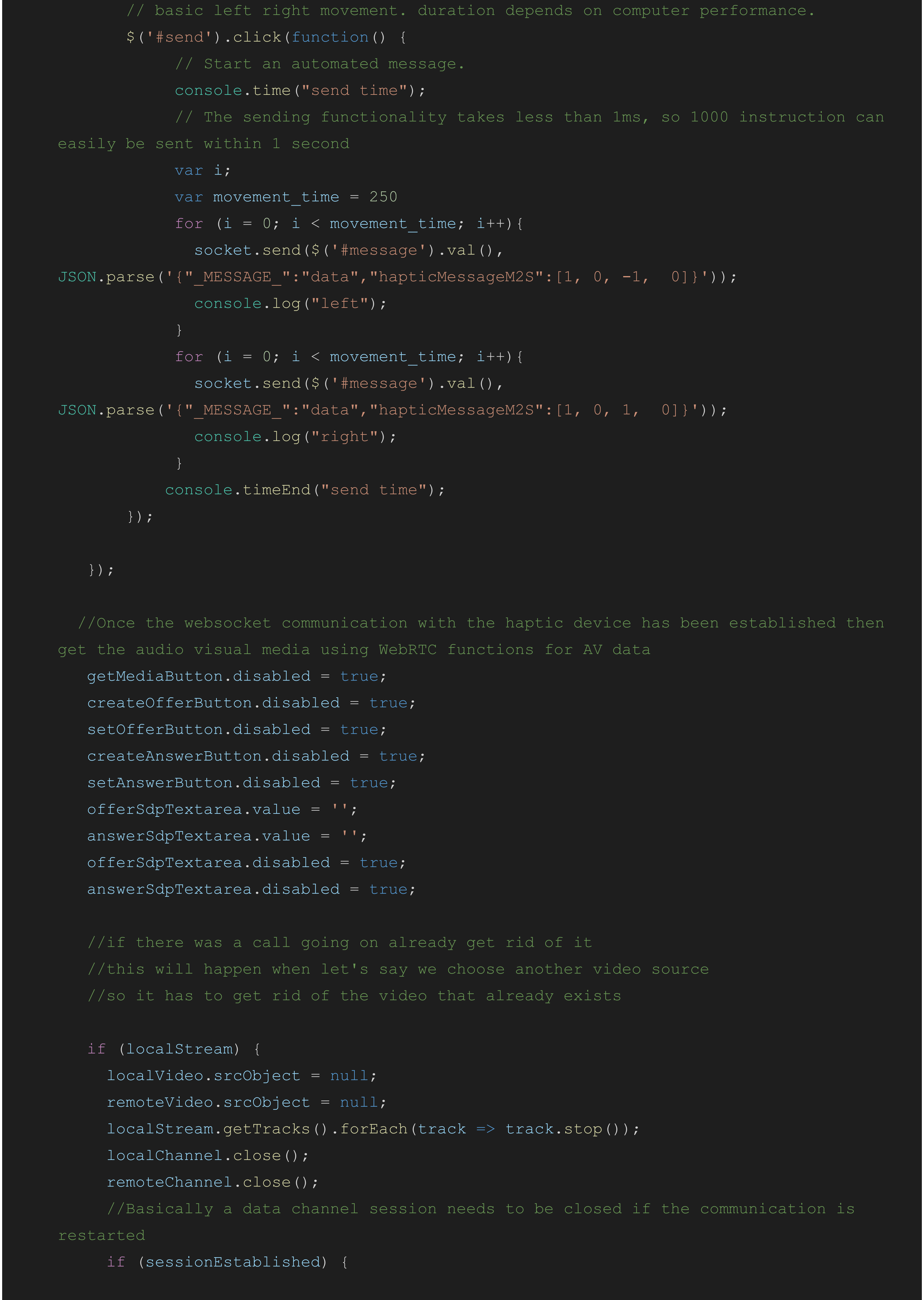
```
       audioSelect.appendChild(option);
     }
     else if (sourceInfos[i].kind === 'videoinput') {
       videoCount++;
       if (option.text === '') {
         //option.text = `Video ${videoCount}`;
         option.text = 'Logitech QuickCam Communicate STX (046d:08d7)';
       }
       videoSelect.appendChild(option);
     }
     else {
       console.log('unknown', JSON.stringify(sourceInfos[i]));
     }
   }
 }
//Printing the current time allows us to calculate how long the get media function takes
 console.time("getMedia time");

// function to get Media automatically from the audiovisual devices
// Media from  haptic devices is obtained through the websocket
 async function getMedia() {

    // console.log("generic getMedia execution")

   //========================================================================
   // From js working haptic communication program.
   //========================================================================
   // Functions to establish the Websocket
   function SocketWrapper(init)
   {
       this.socket = new SimpleWebsocket(init);
       this.messageHandlers = {};

       var that = this;
      // Whenever data event happens (whenever a packet is received)
       this.socket.on('data', function(data)
       {
        // Extract the message type
           var messageData = JSON.parse(data);
           var messageType = messageData['__MESSAGE__'];
           delete messageData['__MESSAGE__'];

           //If any handlers have been registered for the message type, invoke them
           if (that.messageHandlers[messageType] !== undefined)
           {
              let index;
               for (index in that.messageHandlers[messageType]) {
                   that.messageHandlers[messageType][index](messageData);
```



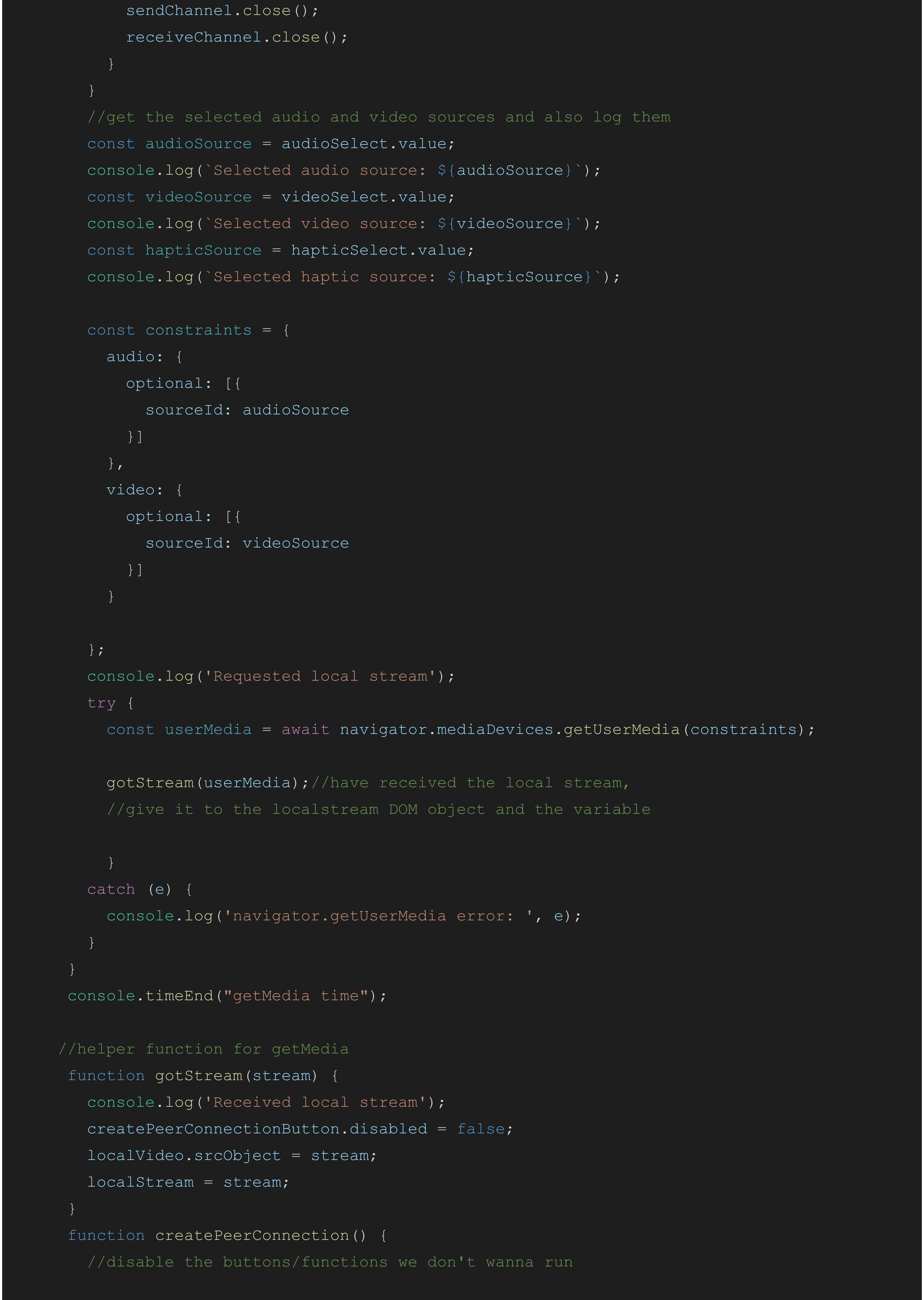
```
               }
           }
       });
   }
     //If a standard event was specified, forward the registration to the socket's event
emitter
   SocketWrapper.prototype.on = function(event, handler)
   {

       if (['connect', 'close', 'data', 'error'].indexOf(event) != -1) {
           this.socket.on(event, handler);
       }
       else
       {
           //The event names a message type
           if (this.messageHandlers[event] === undefined) {
               this.messageHandlers[event] = [];
           }
           this.messageHandlers[event].push(handler);
       }
   }

//Function to send a payload with the label message
   SocketWrapper.prototype.send = function(message, payload)
   {
       //Copy the values from the payload object, if one was supplied
       var payloadCopy = {};
       if (payload !== undefined && payload !== null)
       {
           var keys = Object.keys(payload);
           let index;
           for (index in keys)
           {
               var key = keys[index];
               payloadCopy[key] = payload[key];
           }
       }
      // this function is used when client is sending to the C++ server

       payloadCopy['__MESSAGE__'] = message;
       this.socket.send(JSON.stringify(payloadCopy));    }
  //text area to log haptic data communication
 function log(text)
   {
       var outputElem = $('#outputLocal');
       outputElem.text( outputElem.text() + text + '\n' );
   }
```



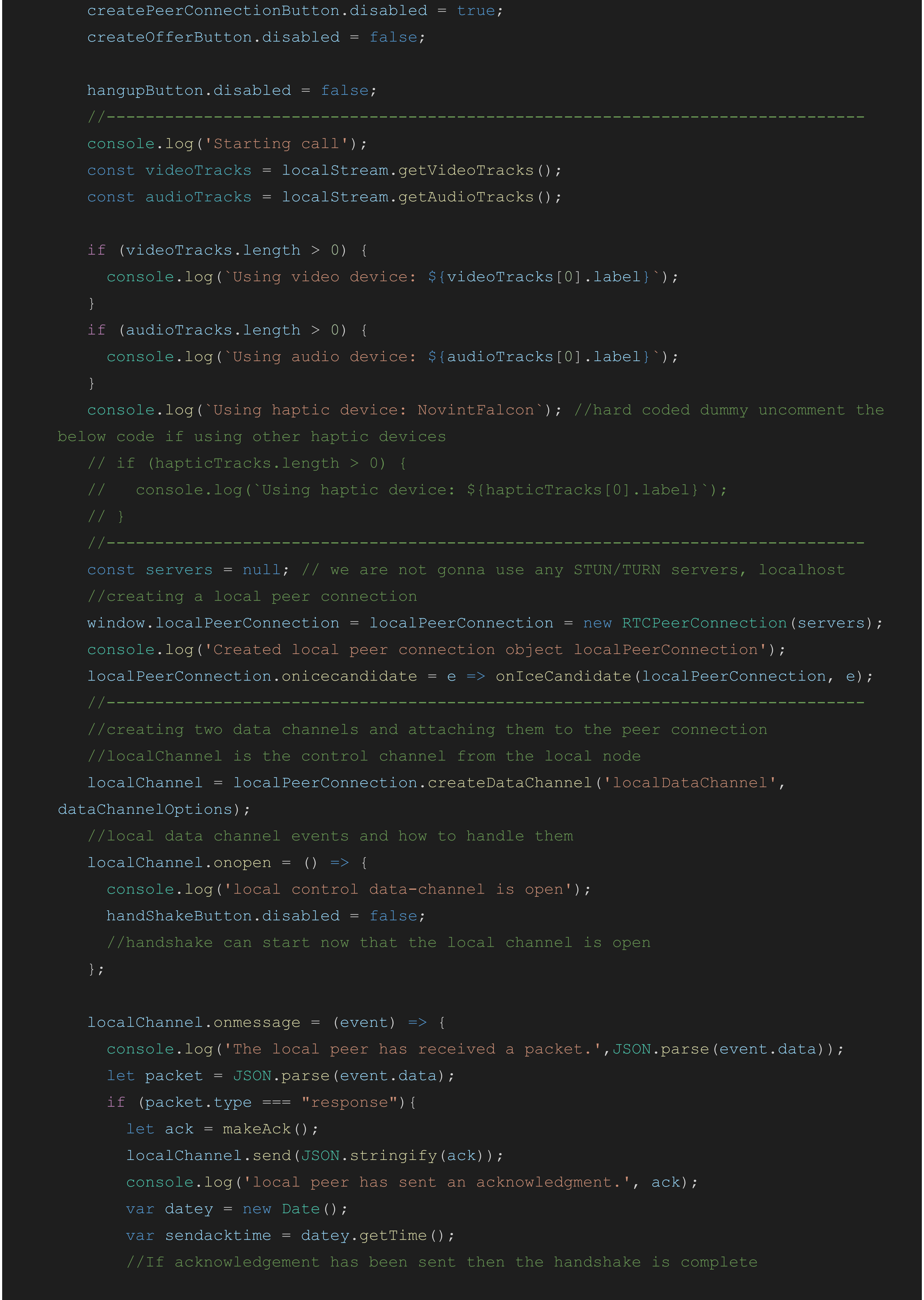
```
   $(document).ready(function()
   {
      //The socket is opened on the local host on port 8080
       var socket = new SocketWrapper("ws://127.0.0.1:8080");

       //Generic events

       socket.on('connect', function() {
           log("socket is connected!");
       });

       socket.on('data', function(data) {
           // To avoid slow browser response, don't output raw data. instead, print a
stat once some number of packets has been received. In this case, a stat is printed on
the browser once per 1000 haptic packets

           //the data received is put into the hapticMsgFromMaster global variable so it
is accessible from any scope
           hapticMsgFromMaster = data;
           if(timeCounter % 1000 == 0){
             log("Recv packet rate: \"" + packetRate + "\"");
             timeCounter = 0;
             packetRate = 0;

           }
           timeCounter++;
           packetRate++;
       });

       socket.on('close', function() {
           log("socket is disconnected!");
       });

       socket.on('error', function(err) {
           log("Error: " + err);
       });

       //Specific message type handlers

       socket.on('userInput', function(args) {
         log("Received user input: \"" + args['input'] + "\"");
       });

       // this functionality is used for basic haptic device movement test on
HAVnetSimulation
//packets listed within the message textarea are sent to the slave device whenever the
send button is clicked
```



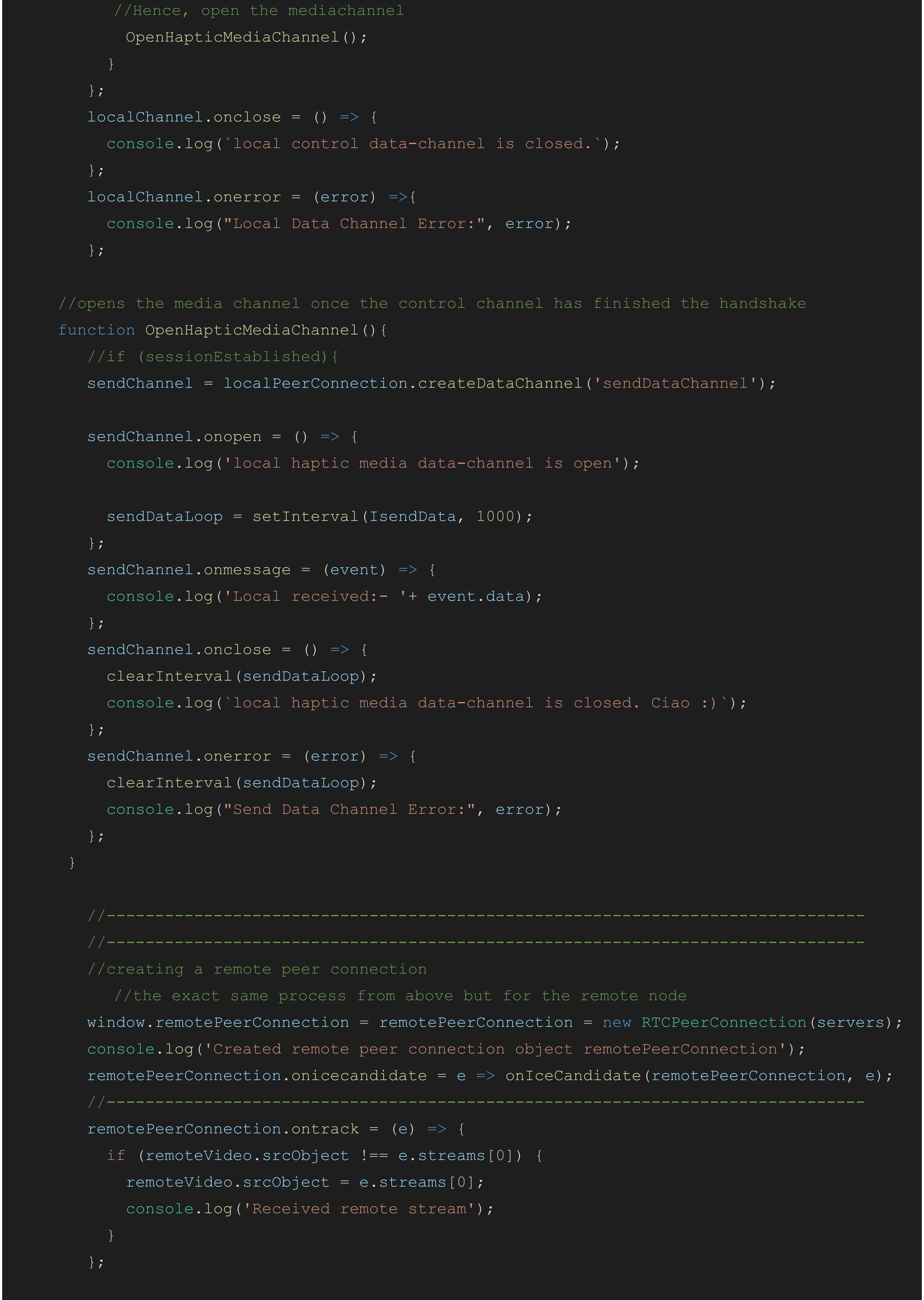
```
       // basic left right movement. duration depends on computer performance.
       $('#send').click(function() {
            // Start an automated message.
            console.time("send time");
            // The sending functionality takes less than 1ms, so 1000 instruction can
easily be sent within 1 second
            var i;
            var movement_time = 250
            for (i = 0; i < movement_time; i++){
              socket.send($('#message').val(),
JSON.parse('{"_MESSAGE_":"data","hapticMessageM2S":[1, 0, -1,  0]}'));
              console.log("left");
            }
            for (i = 0; i < movement_time; i++){
              socket.send($('#message').val(),
JSON.parse('{"_MESSAGE_":"data","hapticMessageM2S":[1, 0, 1,  0]}'));
              console.log("right");
            }
           console.timeEnd("send time");
       });

   });

  //Once the websocket communication with the haptic device has been established then
get the audio visual media using WebRTC functions for AV data
   getMediaButton.disabled = true;
   createOfferButton.disabled = true;
   setOfferButton.disabled = true;
   createAnswerButton.disabled = true;
   setAnswerButton.disabled = true;
   offerSdpTextarea.value = '';
   answerSdpTextarea.value = '';
   offerSdpTextarea.disabled = true;
   answerSdpTextarea.disabled = true;

   //if there was a call going on already get rid of it
   //this will happen when let's say we choose another video source
   //so it has to get rid of the video that already exists

   if (localStream) {
     localVideo.srcObject = null;
     remoteVideo.srcObject = null;
     localStream.getTracks().forEach(track => track.stop());
     localChannel.close();
     remoteChannel.close();
     //Basically a data channel session needs to be closed if the communication is
restarted
     if (sessionEstablished) {
```



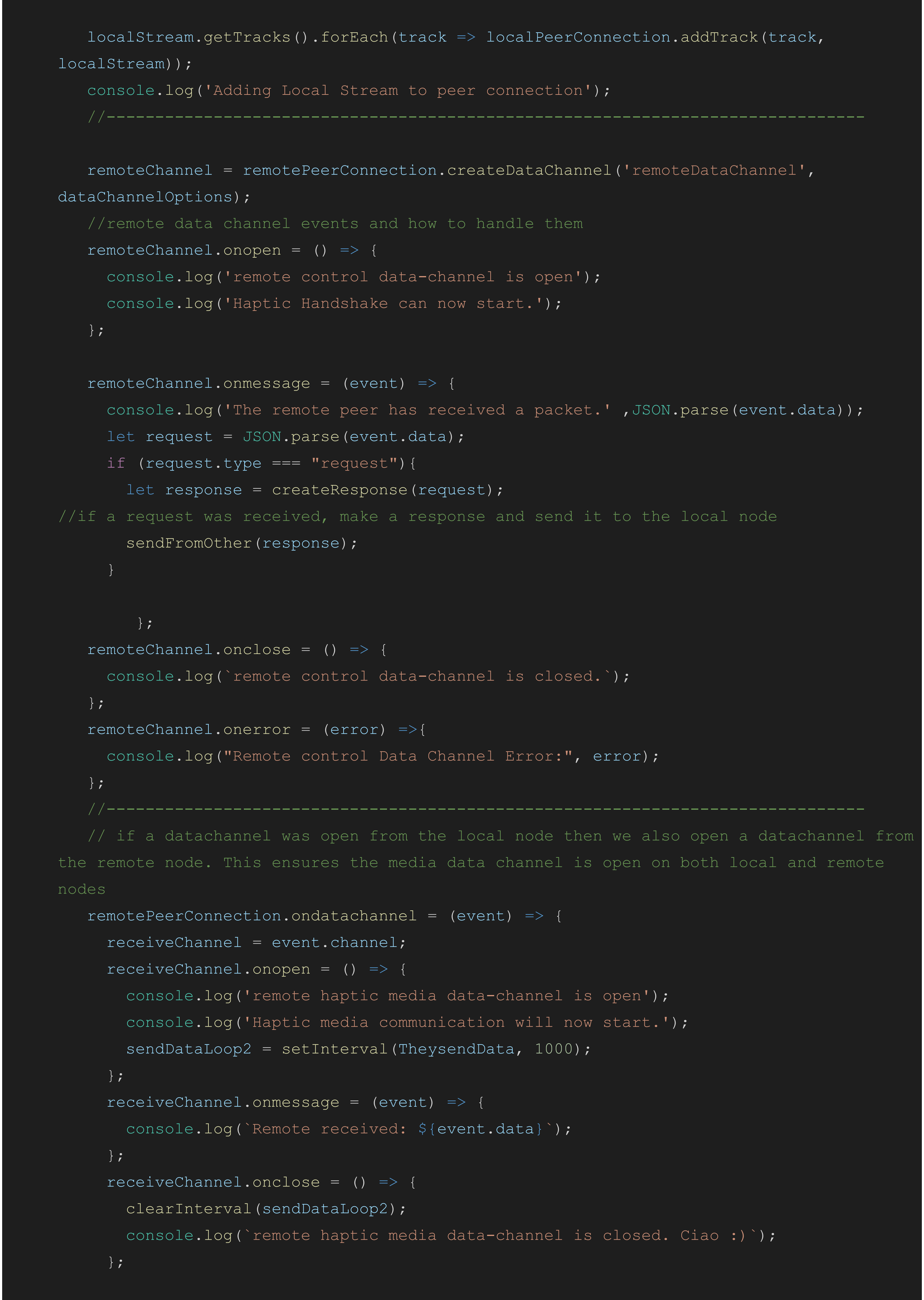
```
       sendChannel.close();
       receiveChannel.close();
     }
   }
   //get the selected audio and video sources and also log them
   const audioSource = audioSelect.value;
   console.log(`Selected audio source: ${audioSource}`);
   const videoSource = videoSelect.value;
   console.log(`Selected video source: ${videoSource}`);
   const hapticSource = hapticSelect.value;
   console.log(`Selected haptic source: ${hapticSource}`);

   const constraints = {
     audio: {
       optional: [{
         sourceId: audioSource
       }]
     },
     video: {
       optional: [{
         sourceId: videoSource
       }]
     }

   };
   console.log('Requested local stream');
   try {
     const userMedia = await navigator.mediaDevices.getUserMedia(constraints);

     gotStream(userMedia);//have received the local stream,
     //give it to the localstream DOM object and the variable

     }
   catch (e) {
     console.log('navigator.getUserMedia error: ', e);
   }
 }
 console.timeEnd("getMedia time");

//helper function for getMedia
 function gotStream(stream) {
   console.log('Received local stream');
   createPeerConnectionButton.disabled = false;
   localVideo.srcObject = stream;
   localStream = stream;
 }
 function createPeerConnection() {
   //disable the buttons/functions we don't wanna run
```



```
  createPeerConnectionButton.disabled = true;
  createOfferButton.disabled = false;

  hangupButton.disabled = false;
  //------------------------------------------------------------------------------
  console.log('Starting call');
  const videoTracks = localStream.getVideoTracks();
  const audioTracks = localStream.getAudioTracks();

  if (videoTracks.length > 0) {
    console.log(`Using video device: ${videoTracks[0].label}`);
  }
  if (audioTracks.length > 0) {
    console.log(`Using audio device: ${audioTracks[0].label}`);
  }
  console.log(`Using haptic device: NovintFalcon`); //hard coded dummy uncomment the
below code if using other haptic devices
  // if (hapticTracks.length > 0) {
  //   console.log(`Using haptic device: ${hapticTracks[0].label}`);
  // }
  //------------------------------------------------------------------------------
  const servers = null; // we are not gonna use any STUN/TURN servers, localhost
  //creating a local peer connection
  window.localPeerConnection = localPeerConnection = new RTCPeerConnection(servers);
  console.log('Created local peer connection object localPeerConnection');
  localPeerConnection.onicecandidate = e => onIceCandidate(localPeerConnection, e);
  //------------------------------------------------------------------------------
  //creating two data channels and attaching them to the peer connection
  //localChannel is the control channel from the local node
  localChannel = localPeerConnection.createDataChannel('localDataChannel',
dataChannelOptions);
  //local data channel events and how to handle them
  localChannel.onopen = () => {
    console.log('local control data-channel is open');
    handShakeButton.disabled = false;
    //handshake can start now that the local channel is open
  };

  localChannel.onmessage = (event) => {
    console.log('The local peer has received a packet.',JSON.parse(event.data));
    let packet = JSON.parse(event.data);
    if (packet.type === "response"){
      let ack = makeAck();
      localChannel.send(JSON.stringify(ack));
      console.log('local peer has sent an acknowledgment.', ack);
      var datey = new Date();
      var sendacktime = datey.getTime();
      //If acknowledgement has been sent then the handshake is complete
```

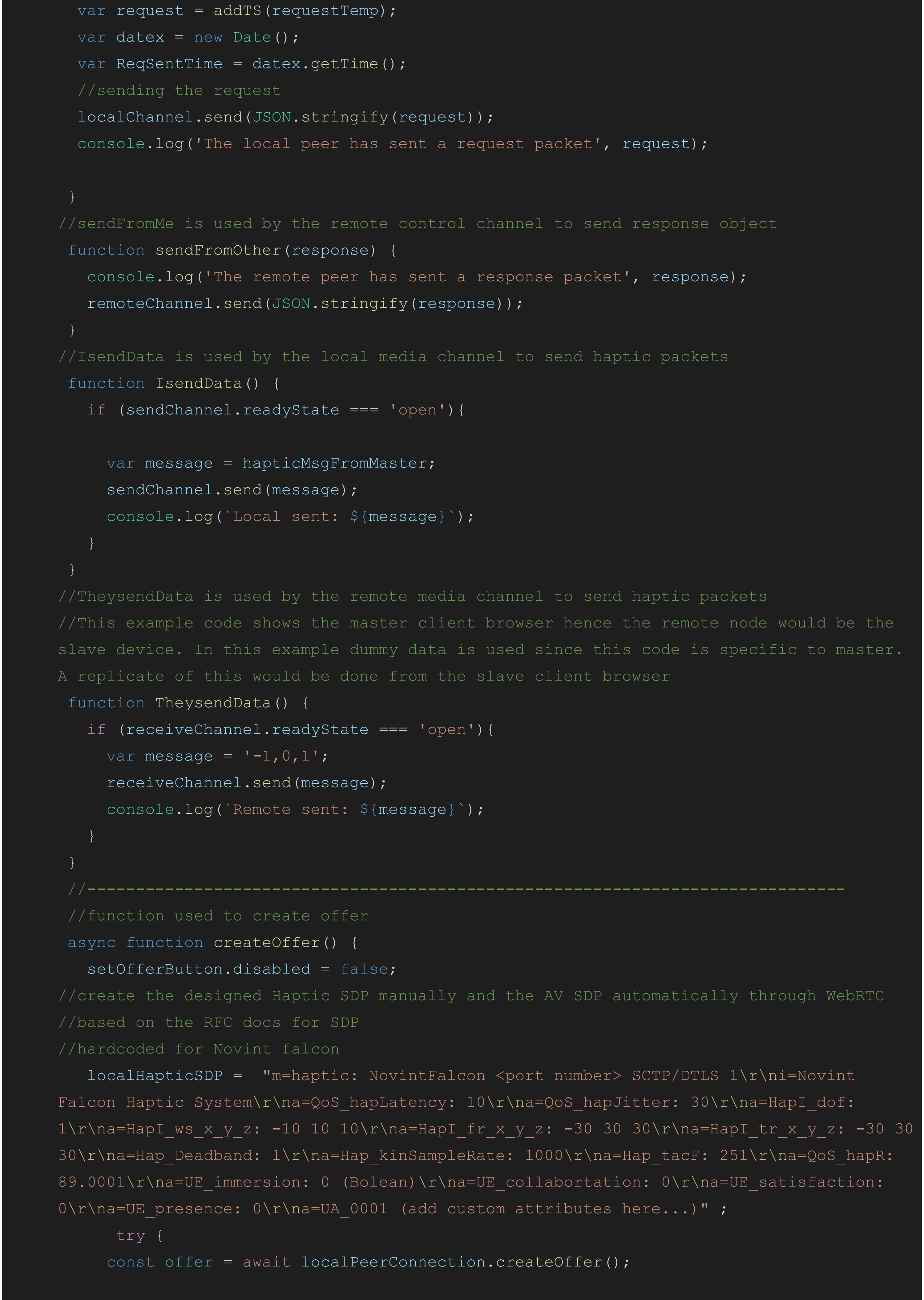


```
      //Hence, open the mediachannel
       OpenHapticMediaChannel();
     }
   };
   localChannel.onclose = () => {
     console.log(`local control data-channel is closed.`);
   };
   localChannel.onerror = (error) =>{
     console.log("Local Data Channel Error:", error);
   };

//opens the media channel once the control channel has finished the handshake
function OpenHapticMediaChannel(){
   //if (sessionEstablished){
   sendChannel = localPeerConnection.createDataChannel('sendDataChannel');

   sendChannel.onopen = () => {
     console.log('local haptic media data-channel is open');

     sendDataLoop = setInterval(IsendData, 1000);
   };
   sendChannel.onmessage = (event) => {
     console.log('Local received:- '+ event.data);
   };
   sendChannel.onclose = () => {
     clearInterval(sendDataLoop);
     console.log(`local haptic media data-channel is closed. Ciao :)`);
   };
   sendChannel.onerror = (error) => {
     clearInterval(sendDataLoop);
     console.log("Send Data Channel Error:", error);
   };
 }

   //--------------------------------------------------------------------------------
   //--------------------------------------------------------------------------------
   //creating a remote peer connection
      //the exact same process from above but for the remote node
   window.remotePeerConnection = remotePeerConnection = new RTCPeerConnection(servers);
   console.log('Created remote peer connection object remotePeerConnection');
   remotePeerConnection.onicecandidate = e => onIceCandidate(remotePeerConnection, e);
   //--------------------------------------------------------------------------------
   remotePeerConnection.ontrack = (e) => {
     if (remoteVideo.srcObject !== e.streams[0]) {
       remoteVideo.srcObject = e.streams[0];
       console.log('Received remote stream');
     }
   };
```

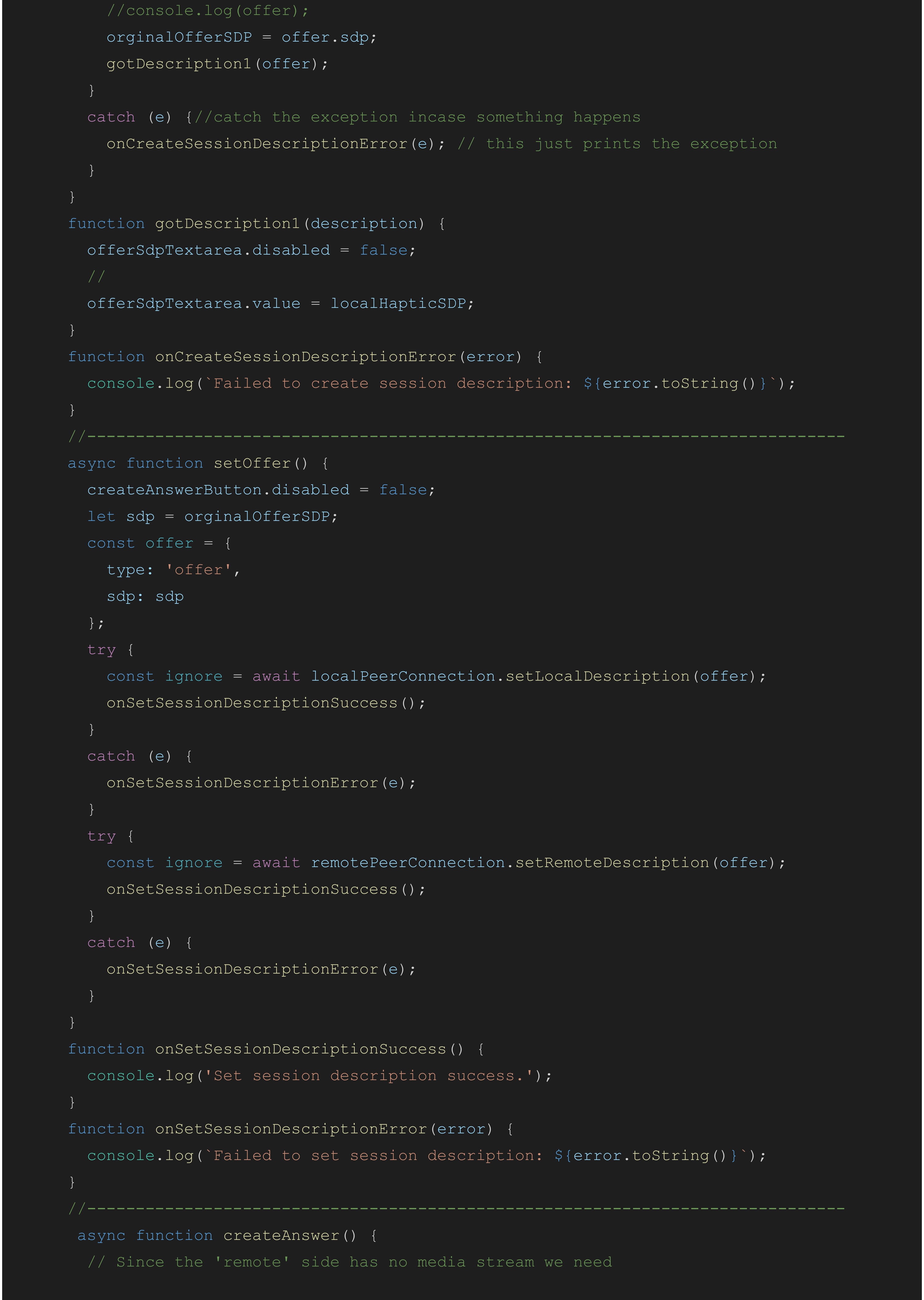


```
  localStream.getTracks().forEach(track => localPeerConnection.addTrack(track,
localStream));
  console.log('Adding Local Stream to peer connection');
  //-------------------------------------------------------------------------------------

  remoteChannel = remotePeerConnection.createDataChannel('remoteDataChannel',
dataChannelOptions);
  //remote data channel events and how to handle them
  remoteChannel.onopen = () => {
    console.log('remote control data-channel is open');
    console.log('Haptic Handshake can now start.');
  };

  remoteChannel.onmessage = (event) => {
    console.log('The remote peer has received a packet.' ,JSON.parse(event.data));
    let request = JSON.parse(event.data);
    if (request.type === "request"){
      let response = createResponse(request);
//if a request was received, make a response and send it to the local node
      sendFromOther(response);
    }


       };
  remoteChannel.onclose = () => {
    console.log(`remote control data-channel is closed.`);
  };
  remoteChannel.onerror = (error) =>{
    console.log("Remote control Data Channel Error:", error);
  };
  //-------------------------------------------------------------------------------------
  // if a datachannel was open from the local node then we also open a datachannel from
the remote node. This ensures the media data channel is open on both local and remote
nodes
  remotePeerConnection.ondatachannel = (event) => {
    receiveChannel = event.channel;
    receiveChannel.onopen = () => {
      console.log('remote haptic media data-channel is open');
      console.log('Haptic media communication will now start.');
      sendDataLoop2 = setInterval(TheysendData, 1000);
    };
    receiveChannel.onmessage = (event) => {
      console.log(`Remote received: ${event.data}`);
    };
    receiveChannel.onclose = () => {
      clearInterval(sendDataLoop2);
      console.log(`remote haptic media data-channel is closed. Ciao :)`);
    };
```

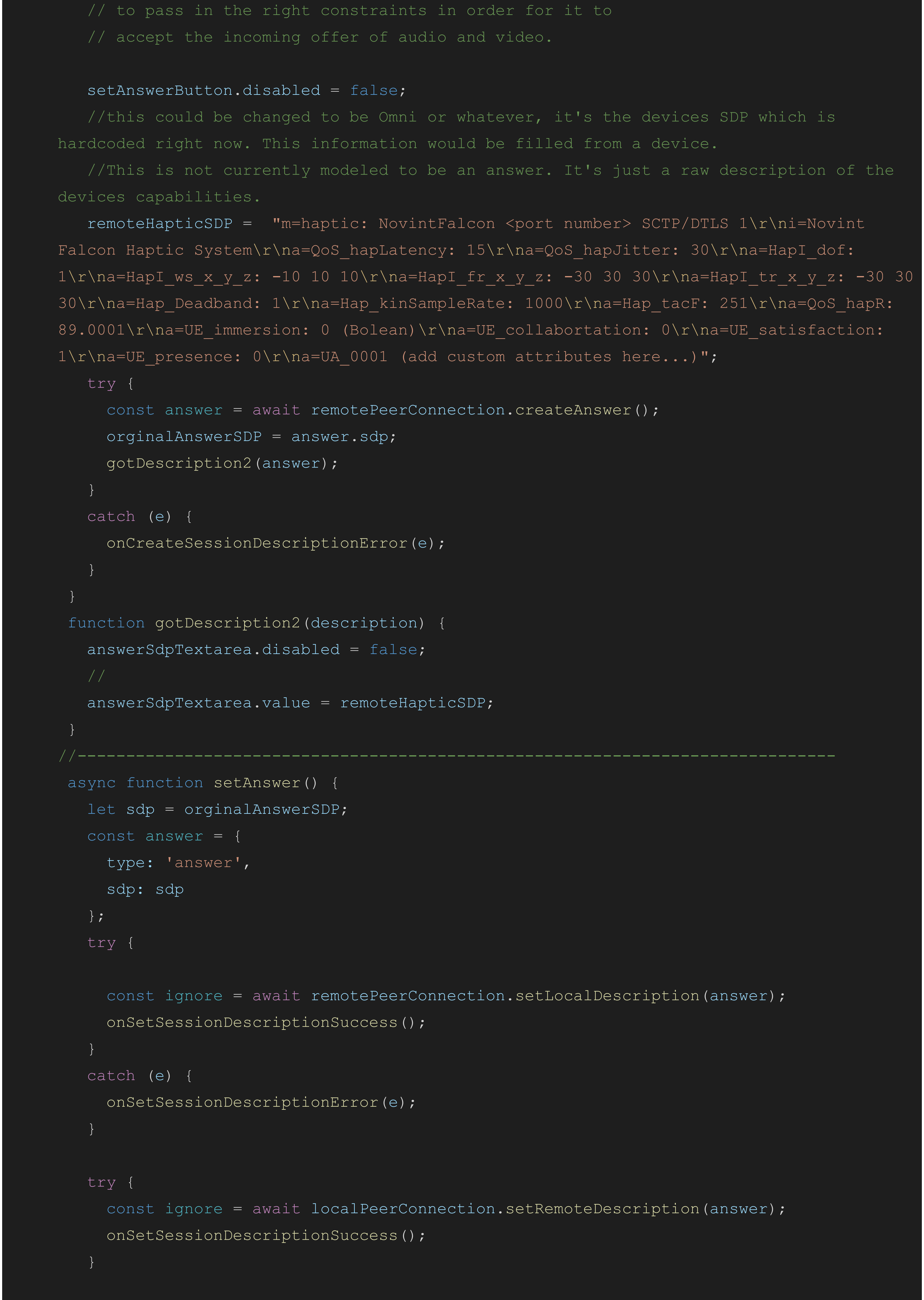
```
     receiveChannel.onerror = (error) => {
       clearInterval(sendDataLoop2);
       console.log("Receive Data Channel Error:", error);
     };
   };
 }
 //Dealing with ICE candidates
//connects the local and remote peer connection objects
 //----------------------------------------------------------------------------------
 function getOtherPc(pc) {
   return (pc === localPeerConnection) ? remotePeerConnection : localPeerConnection;
 }
 function getName(pc) {
   return (pc === localPeerConnection) ? 'localPeerConnection' : 'remotePeerConnection';
 }
 async function onIceCandidate(pc, event) {
   try {
     // eslint-disable-next-line no-unused-vars
     const ignore = await getOtherPc(pc).addIceCandidate(event.candidate);
     onAddIceCandidateSuccess(pc);
   }
   catch (e) {
     onAddIceCandidateError(pc, e);
   }
    }
 function onAddIceCandidateSuccess() {
   console.log('AddIceCandidate success.');
 }
 function onAddIceCandidateError(error) {
   console.log(`Failed to add Ice Candidate: ${error.toString()}`);
 }
 //----------------------------------------------------------------------------------
 //functions used to send messages from the two pairs of data channels

//sendFromMe is used by the local control channel to send request object
  function sendFromMe(){

  let requestTemp = {
    type: "request",
    sessionDescription: "v=0\r\no=- <sess-id add time stamp here> <sess-version add a
feature that counts everytime offer is created> IN IP4 127.0.0.1\r\ns=Haptic
SDP\r\ni=SDP for Haptic Handshake\r\nt= 0 0\r\na=<add attribute at the session
level>\r\n",
    mediaDescription: localHapticSDP,
    CodecParams: "file" //this can be replaced by codec parameters communicated from the
C++ environment
  }
```



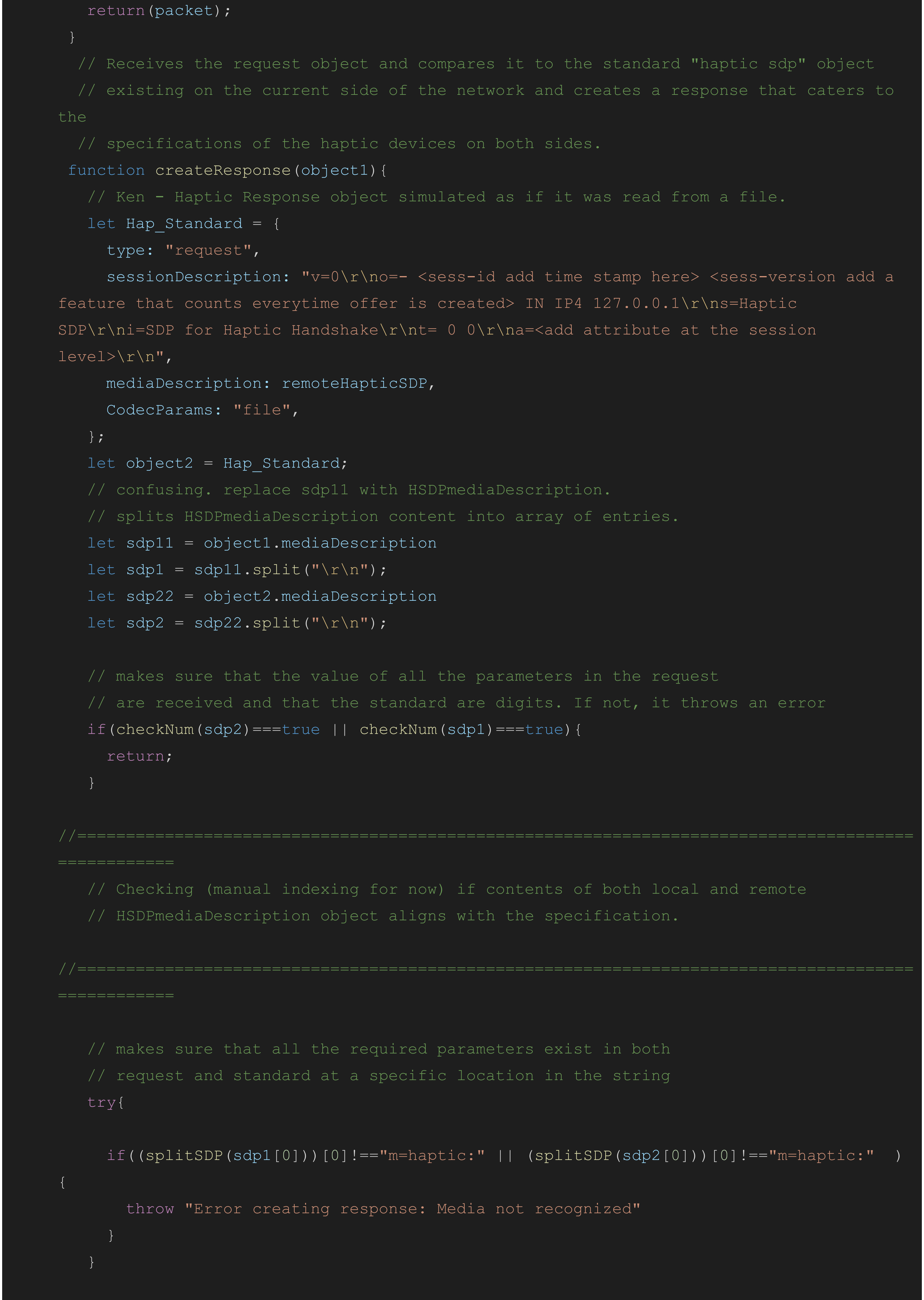
```
  var request = addTS(requestTemp);
  var datex = new Date();
  var ReqSentTime = datex.getTime();
  //sending the request
  localChannel.send(JSON.stringify(request));
  console.log('The local peer has sent a request packet', request);

 }
//sendFromMe is used by the remote control channel to send response object
 function sendFromOther(response) {
   console.log('The remote peer has sent a response packet', response);
   remoteChannel.send(JSON.stringify(response));
 }
//IsendData is used by the local media channel to send haptic packets
 function IsendData() {
   if (sendChannel.readyState === 'open'){

     var message = hapticMsgFromMaster;
     sendChannel.send(message);
     console.log(`Local sent: ${message}`);
   }
 }
//TheysendData is used by the remote media channel to send haptic packets
//This example code shows the master client browser hence the remote node would be the
slave device. In this example dummy data is used since this code is specific to master.
A replicate of this would be done from the slave client browser
 function TheysendData() {
   if (receiveChannel.readyState === 'open'){
     var message = '-1,0,1';
     receiveChannel.send(message);
     console.log(`Remote sent: ${message}`);
   }
 }
 //------------------------------------------------------------------------------
 //function used to create offer
 async function createOffer() {
   setOfferButton.disabled = false;
//create the designed Haptic SDP manually and the AV SDP automatically through WebRTC
//based on the RFC docs for SDP
//hardcoded for Novint falcon
   localHapticSDP =  "m=haptic: NovintFalcon <port number> SCTP/DTLS 1\r\ni=Novint
Falcon Haptic System\r\na=QoS_hapLatency: 10\r\na=QoS_hapJitter: 30\r\na=HapI_dof:
1\r\na=HapI_ws_x_y_z: -10 10 10\r\na=HapI_fr_x_y_z: -30 30 30\r\na=HapI_tr_x_y_z: -30 30
30\r\na=Hap_Deadband: 1\r\na=Hap_kinSampleRate: 1000\r\na=Hap_tacF: 251\r\na=QoS_hapR:
89.0001\r\na=UE_immersion: 0 (Bolean)\r\na=UE_collabortation: 0\r\na=UE_satisfaction:
0\r\na=UE_presence: 0\r\na=UA_0001 (add custom attributes here...)" ;
      try {
     const offer = await localPeerConnection.createOffer();
```



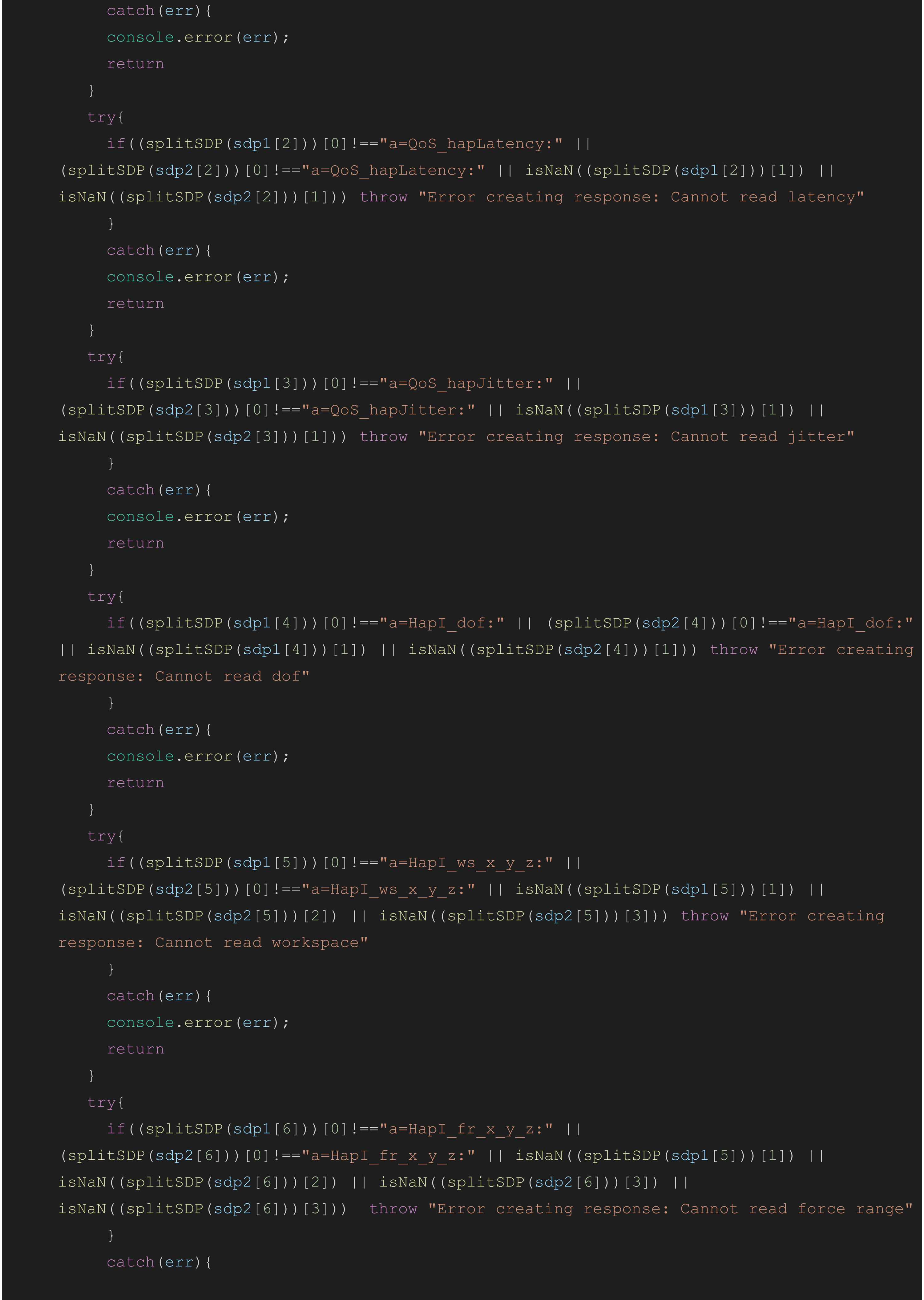
```
    //console.log(offer);
    orginalOfferSDP = offer.sdp;
    gotDescription1(offer);
  }
  catch (e) {//catch the exception incase something happens
    onCreateSessionDescriptionError(e); // this just prints the exception
  }
}
function gotDescription1(description) {
  offerSdpTextarea.disabled = false;
  //
  offerSdpTextarea.value = localHapticSDP;
}
function onCreateSessionDescriptionError(error) {
  console.log(`Failed to create session description: ${error.toString()}`);
}
//--------------------------------------------------------------------------------
async function setOffer() {
  createAnswerButton.disabled = false;
  let sdp = orginalOfferSDP;
  const offer = {
    type: 'offer',
    sdp: sdp
  };
  try {
    const ignore = await localPeerConnection.setLocalDescription(offer);
    onSetSessionDescriptionSuccess();
  }
  catch (e) {
    onSetSessionDescriptionError(e);
  }
  try {
    const ignore = await remotePeerConnection.setRemoteDescription(offer);
    onSetSessionDescriptionSuccess();
  }
  catch (e) {
    onSetSessionDescriptionError(e);
  }
}
function onSetSessionDescriptionSuccess() {
  console.log('Set session description success.');
}
function onSetSessionDescriptionError(error) {
  console.log(`Failed to set session description: ${error.toString()}`);
}
//--------------------------------------------------------------------------------
 async function createAnswer() {
  // Since the 'remote' side has no media stream we need
```



```
   // to pass in the right constraints in order for it to
   // accept the incoming offer of audio and video.

   setAnswerButton.disabled = false;
   //this could be changed to be Omni or whatever, it's the devices SDP which is
hardcoded right now. This information would be filled from a device.
   //This is not currently modeled to be an answer. It's just a raw description of the
devices capabilities.
   remoteHapticSDP =  "m=haptic: NovintFalcon <port number> SCTP/DTLS 1\r\ni=Novint
Falcon Haptic System\r\na=QoS_hapLatency: 15\r\na=QoS_hapJitter: 30\r\na=HapI_dof:
1\r\na=HapI_ws_x_y_z: -10 10 10\r\na=HapI_fr_x_y_z: -30 30 30\r\na=HapI_tr_x_y_z: -30 30
30\r\na=Hap_Deadband: 1\r\na=Hap_kinSampleRate: 1000\r\na=Hap_tacF: 251\r\na=QoS_hapR:
89.0001\r\na=UE_immersion: 0 (Bolean)\r\na=UE_collabortation: 0\r\na=UE_satisfaction:
1\r\na=UE_presence: 0\r\na=UA_0001 (add custom attributes here...)";
   try {
     const answer = await remotePeerConnection.createAnswer();
     orginalAnswerSDP = answer.sdp;
     gotDescription2(answer);
   }
   catch (e) {
     onCreateSessionDescriptionError(e);
   }
 }
 function gotDescription2(description) {
   answerSdpTextarea.disabled = false;
   //
   answerSdpTextarea.value = remoteHapticSDP;
 }
//--------------------------------------------------------------------------------
 async function setAnswer() {
   let sdp = orginalAnswerSDP;
   const answer = {
     type: 'answer',
     sdp: sdp
   };
   try {

     const ignore = await remotePeerConnection.setLocalDescription(answer);
     onSetSessionDescriptionSuccess();
   }
   catch (e) {
     onSetSessionDescriptionError(e);
   }

   try {
     const ignore = await localPeerConnection.setRemoteDescription(answer);
     onSetSessionDescriptionSuccess();
   }
   catch (e) {
     onSetSessionDescriptionError(e);
   }
 }
 //--------------------------------------------------------------------------------
 function hangup() {
   remoteVideo.srcObject = null;
   console.log('Ending call');
   localStream.getTracks().forEach(track => track.stop());

   sendChannel.close();
   receiveChannel.close();
   localChannel.close();
   remoteChannel.close();

   localPeerConnection.close();
   remotePeerConnection.close();
   offerSdpTextarea.value = '';
   answerSdpTextarea.value = '';
   offerSdpTextarea.disabled = true;
   answerSdpTextarea.disabled = true;
   getMediaButton.disabled = false;
   handShakeButton.disabled = true;
   createPeerConnectionButton.disabled = true;
   createOfferButton.disabled = true;
   setOfferButton.disabled = true;
   createAnswerButton.disabled = true;
   setAnswerButton.disabled = true;
   hangupButton.disabled = true;
   // localChannel = null;
   // remoteChannel = null;
   // sendChannel = null;
   // receiveChannel = null;
   // localPeerConnection = null;
   // remotePeerConnection = null;
 }

//----------------------------------------------------------------------------------------
-------------------------------
 // funtions to process and manipulate the Haptic SDP
 function addTS(packet){
   let sessionDesc = packet.sessionDescription;
   let sessionArray = sessionDesc.split("\r\n")
   let parseOrigin = sessionArray[1].split(" ")
   var date = new Date();
   parseOrigin[1] = date.getTime();
   sessionArray[1] = parseOrigin.join(" ");
   packet.sessionDescription = sessionArray.join("\r\n");
```

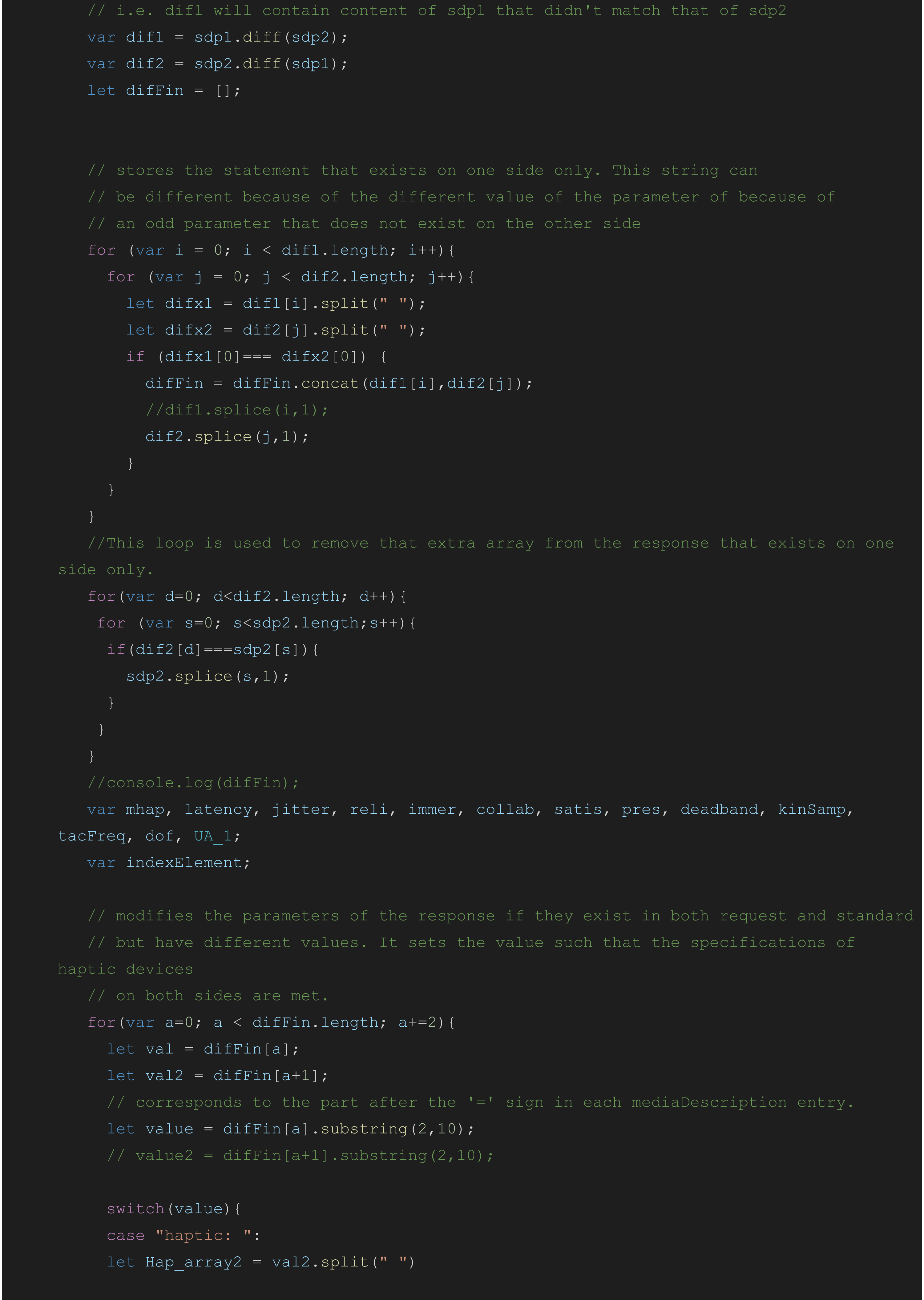


```
   return(packet);
 }
  // Receives the request object and compares it to the standard "haptic sdp" object
  // existing on the current side of the network and creates a response that caters to
the
  // specifications of the haptic devices on both sides.
 function createResponse(object1){
   // Ken - Haptic Response object simulated as if it was read from a file.
   let Hap_Standard = {
     type: "request",
     sessionDescription: "v=0\r\no=- <sess-id add time stamp here> <sess-version add a
feature that counts everytime offer is created> IN IP4 127.0.0.1\r\ns=Haptic
SDP\r\ni=SDP for Haptic Handshake\r\nt= 0 0\r\na=<add attribute at the session
level>\r\n",
     mediaDescription: remoteHapticSDP,
     CodecParams: "file",
   };
   let object2 = Hap_Standard;
   // confusing. replace sdp11 with HSDPmediaDescription.
   // splits HSDPmediaDescription content into array of entries.
   let sdp11 = object1.mediaDescription
   let sdp1 = sdp11.split("\r\n");
   let sdp22 = object2.mediaDescription
   let sdp2 = sdp22.split("\r\n");

   // makes sure that the value of all the parameters in the request
   // are received and that the standard are digits. If not, it throws an error
   if(checkNum(sdp2)===true || checkNum(sdp1)===true){
     return;
   }

//==========================================================================================
============
   // Checking (manual indexing for now) if contents of both local and remote
   // HSDPmediaDescription object aligns with the specification.

//==========================================================================================
============

   // makes sure that all the required parameters exist in both
   // request and standard at a specific location in the string
   try{

     if((splitSDP(sdp1[0]))[0]!=="m=haptic:" || (splitSDP(sdp2[0]))[0]!=="m=haptic:"  )
{
       throw "Error creating response: Media not recognized"
     }
   }
```

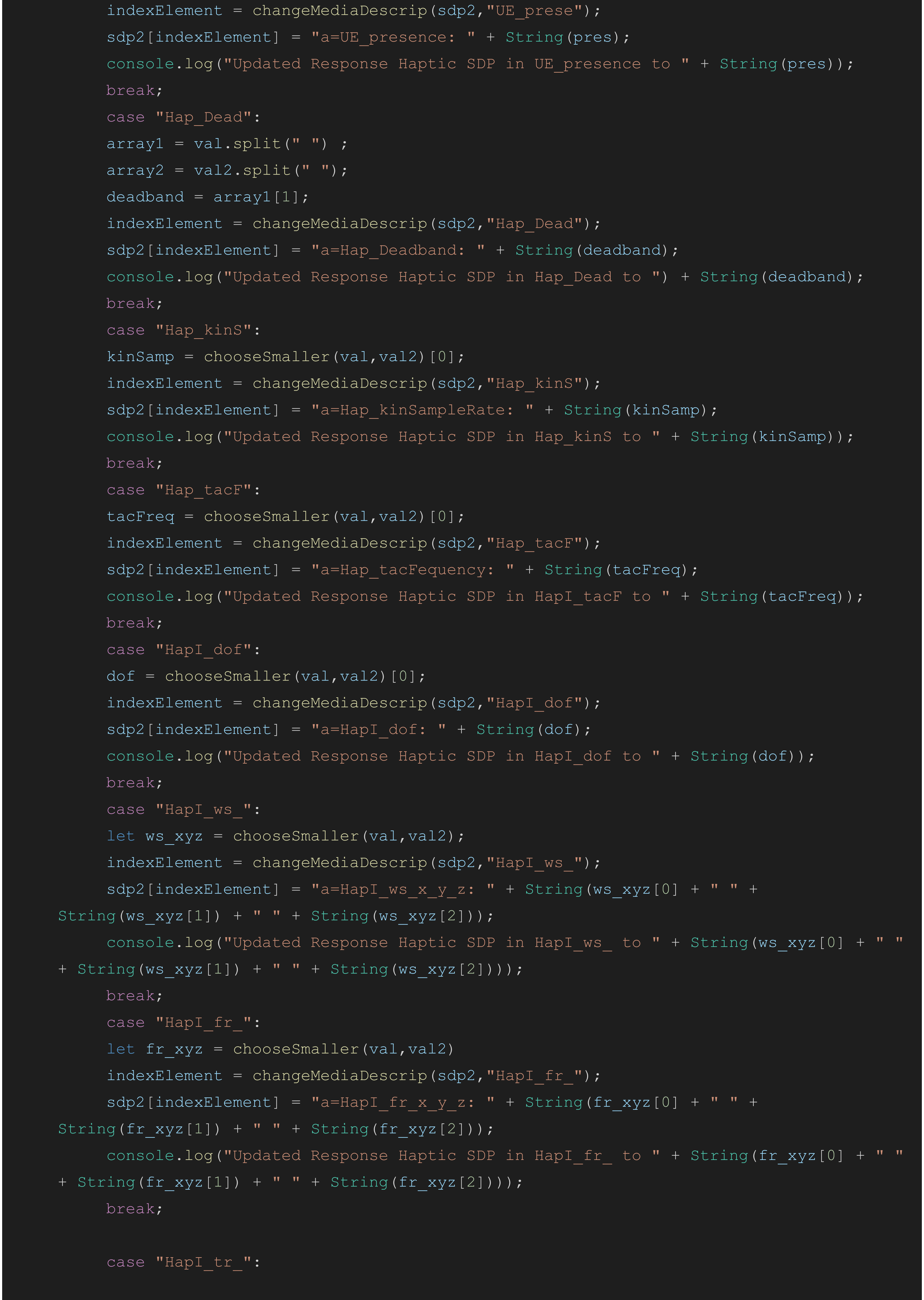


```
     catch(err){
     console.error(err);
     return
   }
   try{
     if((splitSDP(sdp1[2]))[0]!=="a=QoS_hapLatency:" ||
(splitSDP(sdp2[2]))[0]!=="a=QoS_hapLatency:" || isNaN((splitSDP(sdp1[2]))[1]) ||
isNaN((splitSDP(sdp2[2]))[1])) throw "Error creating response: Cannot read latency"
     }
     catch(err){
     console.error(err);
     return
   }
   try{
     if((splitSDP(sdp1[3]))[0]!=="a=QoS_hapJitter:" ||
(splitSDP(sdp2[3]))[0]!=="a=QoS_hapJitter:" || isNaN((splitSDP(sdp1[3]))[1]) ||
isNaN((splitSDP(sdp2[3]))[1])) throw "Error creating response: Cannot read jitter"
     }
     catch(err){
     console.error(err);
     return
   }
   try{
     if((splitSDP(sdp1[4]))[0]!=="a=HapI_dof:" || (splitSDP(sdp2[4]))[0]!=="a=HapI_dof:"
|| isNaN((splitSDP(sdp1[4]))[1]) || isNaN((splitSDP(sdp2[4]))[1])) throw "Error creating
response: Cannot read dof"
     }
     catch(err){
     console.error(err);
     return
   }
   try{
     if((splitSDP(sdp1[5]))[0]!=="a=HapI_ws_x_y_z:" ||
(splitSDP(sdp2[5]))[0]!=="a=HapI_ws_x_y_z:" || isNaN((splitSDP(sdp1[5]))[1]) ||
isNaN((splitSDP(sdp2[5]))[2]) || isNaN((splitSDP(sdp2[5]))[3])) throw "Error creating
response: Cannot read workspace"
     }
     catch(err){
     console.error(err);
     return
   }
   try{
     if((splitSDP(sdp1[6]))[0]!=="a=HapI_fr_x_y_z:" ||
(splitSDP(sdp2[6]))[0]!=="a=HapI_fr_x_y_z:" || isNaN((splitSDP(sdp1[5]))[1]) ||
isNaN((splitSDP(sdp2[6]))[2]) || isNaN((splitSDP(sdp2[6]))[3]) ||
isNaN((splitSDP(sdp2[6]))[3]))  throw "Error creating response: Cannot read force range"
     }
     catch(err){
```

```
     console.error(err);
     return
   }
   try{
     if((splitSDP(sdp1[7]))[0]!=="a=HapI_tr_x_y_z:" ||
(splitSDP(sdp2[7]))[0]!=="a=HapI_tr_x_y_z:" || isNaN((splitSDP(sdp1[7]))[1]) ||
isNaN((splitSDP(sdp2[7]))[2]) || isNaN((splitSDP(sdp2[7]))[3])) throw "Error creating
response: Cannot read torque range"
     }
     catch(err){
     console.error(err);
     return
   }
   try{
     if((splitSDP(sdp1[8]))[0]!=="a=Hap_Deadband:" ||
(splitSDP(sdp2[8]))[0]!=="a=Hap_Deadband:" || isNaN((splitSDP(sdp1[8]))[1]) ||
isNaN((splitSDP(sdp2[8]))[1])) throw "Error creating response: Cannot read deadband"
     }
     catch(err){
     console.error(err);
     return
   }
   try{
     if((splitSDP(sdp1[9]))[0]!=="a=Hap_kinSampleRate:" ||
(splitSDP(sdp2[9]))[0]!=="a=Hap_kinSampleRate:" || isNaN((splitSDP(sdp2[9]))[1]) ||
isNaN((splitSDP(sdp1[9]))[1])) throw "Error creating response: Cannot read kinetic
sample rate"
     }
     catch(err){
     console.error(err);
     return
   }
   try{
     if((splitSDP(sdp1[10]))[0]!=="a=Hap_tacF:" ||
(splitSDP(sdp2[10]))[0]!=="a=Hap_tacF:" || isNaN((splitSDP(sdp1[10]))[1]) ||
isNaN((splitSDP(sdp2[10]))[1])) throw "Error creating response: Cannot read tactile
frequency"
     }
     catch(err){
     console.error(err);
     return
   }

   // finds all parameters in the request or standard that do not have a match
   Array.prototype.diff = function(a) {
       return this.filter(function(i) {return a.indexOf(i) < 0;});
   };
```

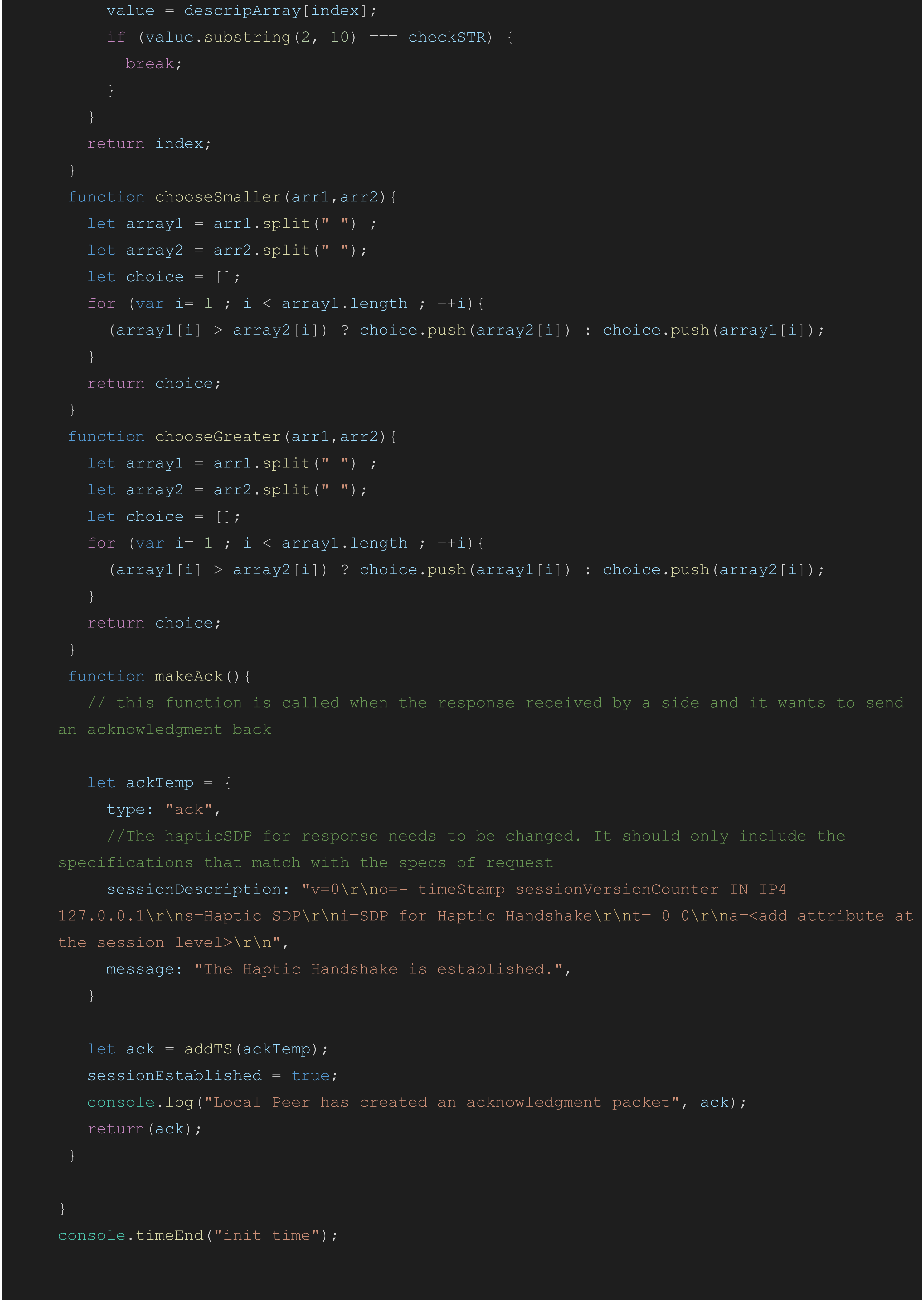


```
   // i.e. dif1 will contain content of sdp1 that didn't match that of sdp2
   var dif1 = sdp1.diff(sdp2);
   var dif2 = sdp2.diff(sdp1);
   let difFin = [];


   // stores the statement that exists on one side only. This string can
   // be different because of the different value of the parameter of because of
   // an odd parameter that does not exist on the other side
   for (var i = 0; i < dif1.length; i++){
     for (var j = 0; j < dif2.length; j++){
       let difx1 = dif1[i].split(" ");
       let difx2 = dif2[j].split(" ");
       if (difx1[0]=== difx2[0]) {
         difFin = difFin.concat(dif1[i],dif2[j]);
         //dif1.splice(i,1);
         dif2.splice(j,1);
       }
     }
   }
   //This loop is used to remove that extra array from the response that exists on one
side only.
   for(var d=0; d<dif2.length; d++){
    for (var s=0; s<sdp2.length;s++){
     if(dif2[d]===sdp2[s]){
       sdp2.splice(s,1);
     }
    }
   }
   //console.log(difFin);
   var mhap, latency, jitter, reli, immer, collab, satis, pres, deadband, kinSamp,
tacFreq, dof, UA_1;
   var indexElement;

   // modifies the parameters of the response if they exist in both request and standard
   // but have different values. It sets the value such that the specifications of
haptic devices
   // on both sides are met.
   for(var a=0; a < difFin.length; a+=2){
     let val = difFin[a];
     let val2 = difFin[a+1];
     // corresponds to the part after the '=' sign in each mediaDescription entry.
     let value = difFin[a].substring(2,10);
     // value2 = difFin[a+1].substring(2,10);

     switch(value){
     case "haptic: ":
     let Hap_array2 = val2.split(" ")
     // Finds the haptic device that exists on the requester's side
     console.log("Connecting to a " + String(Hap_array2[1]));
     break;
     case "QoS_hapL":
     latency = String(chooseGreater(val,val2)[0]);
     indexElement = changeMediaDescrip(sdp2,"QoS_hapL");
     sdp2[indexElement] = "a=QoS_hapLatency: " + latency;
     // Modifies the limit set for latency that works for haptic devices on both sides
     console.log("Updated Response Haptic SDP in QoS_hapLatency to " + latency);
     break;
     case "QoS_hapJ":
     jitter = String(chooseGreater(val,val2)[0]);
     indexElement = changeMediaDescrip(sdp2,"QoS_hapJ");
     sdp2[indexElement] = "a=QoS_hapJitter: " + jitter;
     console.log("Updated Response Haptic SDP in QoS_hapJitter to " + jitter);
     console.log(sdp2);
     break;
     case "QoS_hapR":
     reli = String(chooseSmaller(val,val2)[0]);
     indexElement = changeMediaDescrip(sdp2,"QoS_hapR");
     sdp2[indexElement] = "a=QoS_hapReliability: " + reli;
     console.log("Updated Response Haptic SDP in QoS_hapReliability to " + reli );
     break;
     case "UE_immer":

     immer = 0;      //this only happens when they are different, so response is set to
zero
     indexElement = changeMediaDescrip(sdp2,"UE_immer");
     sdp2[indexElement] = "a=UE_immersion " + String(immer);
     console.log("Updated Response Haptic SDP in UE_immersion to " + String(immer));
     break;
     case "UE_colla":

     collab = 0;
     indexElement = changeMediaDescrip(sdp2,"UE_colla");
     sdp2[indexElement] = "a=UE_collabortation: " + String(collab);
     console.log("Updated Response Haptic SDP in UE_colla to " + String(collab));
     break;
     case "UE_satis":

     satis = 0;
     indexElement = changeMediaDescrip(sdp2,"UE_satis");
     sdp2[indexElement] = "a=UE_satisfaction: " + String(satis);
     console.log("Updated Response Haptic SDP in UE_satisfaction to " + String(satis));
     break;
     case "UE_prese":

     pres = 0;
```



```
     indexElement = changeMediaDescrip(sdp2,"UE_prese");
     sdp2[indexElement] = "a=UE_presence: " + String(pres);
     console.log("Updated Response Haptic SDP in UE_presence to " + String(pres));
     break;
     case "Hap_Dead":
     array1 = val.split(" ") ;
     array2 = val2.split(" ");
     deadband = array1[1];
     indexElement = changeMediaDescrip(sdp2,"Hap_Dead");
     sdp2[indexElement] = "a=Hap_Deadband: " + String(deadband);
     console.log("Updated Response Haptic SDP in Hap_Dead to ") + String(deadband);
     break;
     case "Hap_kinS":
     kinSamp = chooseSmaller(val,val2)[0];
     indexElement = changeMediaDescrip(sdp2,"Hap_kinS");
     sdp2[indexElement] = "a=Hap_kinSampleRate: " + String(kinSamp);
     console.log("Updated Response Haptic SDP in Hap_kinS to " + String(kinSamp));
     break;
     case "Hap_tacF":
     tacFreq = chooseSmaller(val,val2)[0];
     indexElement = changeMediaDescrip(sdp2,"Hap_tacF");
     sdp2[indexElement] = "a=Hap_tacFequency: " + String(tacFreq);
     console.log("Updated Response Haptic SDP in HapI_tacF to " + String(tacFreq));
     break;
     case "HapI_dof":
     dof = chooseSmaller(val,val2)[0];
     indexElement = changeMediaDescrip(sdp2,"HapI_dof");
     sdp2[indexElement] = "a=HapI_dof: " + String(dof);
     console.log("Updated Response Haptic SDP in HapI_dof to " + String(dof));
     break;
     case "HapI_ws_":
     let ws_xyz = chooseSmaller(val,val2);
     indexElement = changeMediaDescrip(sdp2,"HapI_ws_");
     sdp2[indexElement] = "a=HapI_ws_x_y_z: " + String(ws_xyz[0] + " " +
String(ws_xyz[1]) + " " + String(ws_xyz[2]));
     console.log("Updated Response Haptic SDP in HapI_ws_ to " + String(ws_xyz[0] + " "
+ String(ws_xyz[1]) + " " + String(ws_xyz[2])));
     break;
     case "HapI_fr_":
     let fr_xyz = chooseSmaller(val,val2)
     indexElement = changeMediaDescrip(sdp2,"HapI_fr_");
     sdp2[indexElement] = "a=HapI_fr_x_y_z: " + String(fr_xyz[0] + " " +
String(fr_xyz[1]) + " " + String(fr_xyz[2]));
     console.log("Updated Response Haptic SDP in HapI_fr_ to " + String(fr_xyz[0] + " "
+ String(fr_xyz[1]) + " " + String(fr_xyz[2])));
     break;

     case "HapI_tr_":
```

```
     let tr_xyz = chooseSmaller(val,val2)
     indexElement = changeMediaDescrip(sdp2,"HapI_tr_");
     sdp2[indexElement] = "a=HapI_tr_x_y_z: " + String(tr_xyz[0] + " " +
String(tr_xyz[1]) + " " + String(tr_xyz[2]));
     console.log("Updated Response Haptic SDP in HapI_tr_ to " + String(tr_xyz[0] + " "
+ String(tr_xyz[1]) + " " + String(tr_xyz[2])));
     break;
     case "UA_0001 ":
     //This space is for the user to add custom attributes.
     break;
   }
   }

   let resp_med_desc = sdp2.join("\r\n");

   // creates response
   let response ={
     type: "response",
       sessionDescription: "v=0\r\no=- <sess-id add time stamp here> <sess-version add a
feature that counts everytime offer is created> IN IP4 127.0.0.1\r\ns=Haptic
SDP\r\ni=SDP for Haptic Handshake\r\nt= 0 0\r\na=<add attribute at the session
level>\r\n",
       mediaDescription: resp_med_desc,
       CodecParams: object1.CodecParams,
   }
   console.log("A response was created by remote peer", response);
   return response;
 }
 function splitSDP(array){
   let final = array.split(" ");
   return(final)
 }
 function checkNum(numArray){
   var val = false;
   for (var r=11;r<numArray.length-1;r++){
     if (isNaN(splitSDP(numArray[r])[1]))
     {
       val = true;
       console.error("Error: Value of " + String(splitSDP(numArray[r])[0]) + " is not a
number")
     }
   }
   return val;
 }
 function changeMediaDescrip(descripArray, checkSTR){  //Changes the Media Description
for the Response
   var index, result, value;
   for (index = 0; index < descripArray.length; ++index) {
```



```
     value = descripArray[index];
     if (value.substring(2, 10) === checkSTR) {
       break;
     }
   }
   return index;
 }
 function chooseSmaller(arr1,arr2){
   let array1 = arr1.split(" ") ;
   let array2 = arr2.split(" ");
   let choice = [];
   for (var i= 1 ; i < array1.length ; ++i){
     (array1[i] > array2[i]) ? choice.push(array2[i]) : choice.push(array1[i]);
   }
   return choice;
 }
 function chooseGreater(arr1,arr2){
   let array1 = arr1.split(" ") ;
   let array2 = arr2.split(" ");
   let choice = [];
   for (var i= 1 ; i < array1.length ; ++i){
     (array1[i] > array2[i]) ? choice.push(array1[i]) : choice.push(array2[i]);
   }
   return choice;
 }
 function makeAck(){
   // this function is called when the response received by a side and it wants to send an acknowledgment back

   let ackTemp = {
     type: "ack",
     //The hapticSDP for response needs to be changed. It should only include the specifications that match with the specs of request
     sessionDescription: "v=0\r\no=- timeStamp sessionVersionCounter IN IP4 127.0.0.1\r\ns=Haptic SDP\r\ni=SDP for Haptic Handshake\r\nt= 0 0\r\na=<add attribute at the session level>\r\n",
     message: "The Haptic Handshake is established.",
   }

   let ack = addTS(ackTemp);
   sessionEstablished = true;
   console.log("Local Peer has created an acknowledgment packet", ack);
   return(ack);
 }

}
console.timeEnd("init time");
```

12.2 C++ Master Code

```
/#include "commTool.h"
#include "config.h"
#include "HapticCommLib.h"

#include <string.h>
#include <stdlib.h>

#include "chai3d.h"
#include "CBullet.h"
using namespace chai3d;

//---------------------------------------------------------------------------
// Read Parameters from configuration file
//---------------------------------------------------------------------------
//set manually for now

ConfigFile cfg("cfg/config.cfg"); // get the configuration file
double VelocityDeadbandParameter =
cfg.getValueOfKey<double>("VelocityDeadbandParameter"); //deadband parameter for velcity
data reduction, 0.1 is the default value
double PositionDeadbandParameter =
cfg.getValueOfKey<double>("PositionDeadbandParameter"); //deadband parameter for position
data reduction, 0.1 is the default value
double CommandDelay = cfg.getValueOfKey<double>("CommandDelay");

std::string IP_master = cfg.getValueOfKey<std::string>("MasterIP");
std::string IP_slave = cfg.getValueOfKey<std::string>("SlaveIP");

DeadbandDataReduction* DBVelocity; // data reduction class for velocity samples
DeadbandDataReduction* DBPosition; // data reduction class for position samples

bool VelocityTransmitFlag = false; // true: deadband triger false: keep last recently transmitted
sample (ZoH)
bool PositionTransmitFlag = false; // true: deadband triger false: keep last recently transmitted
sample (ZoH)

// TDPA variable
double MasterForce[3] = { 0.0, 0.0, 0.0 };
double MasterVelocity[3] = { 0.0, 0.0, 0.0 }; // update 3 DoF master velocity sample (holds the
signal before deadband)
double MasterPosition[3] = { 0.0, 0.0, 0.0 }; // update 3 DoF master position sample (holds the
signal before deadband)



//---------------------------------------------------------------------------
// DECLARED VARIABLES
//---------------------------------------------------------------------------

// a haptic device handler
cHapticDeviceHandler* handler;
```

```
// a pointer to the current haptic device
cGenericHapticDevicePtr hapticDevice;

// a virtual tool representing the haptic device in the scene
cToolCursor* tool;

// a flag to indicate if the haptic simulation currently running
bool simulationRunning = false;

// a flag to indicate if the haptic simulation has terminated
bool simulationFinished = true;

// a frequency counter to measure the simulation haptic rate
cFrequencyCounter freqCounterHaptics;

// haptic thread
cThread* hapticsThread;


LARGE_INTEGER cpuFreq;
double delay = 0;
//------------------------------------------------------------------------------
// DECLARED FUNCTIONS
//------------------------------------------------------------------------------

// updateHaptics() takes server class as an argument
// this function contains the main haptics simulation loop
void updateHaptics(void);


// this function closes the application
void close(void);

//------------------------------------------------------------------------------
// DECLARED VARIABLES
//------------------------------------------------------------------------------

// a world that contains all objects of the virtual environment
cBulletWorld* world;

// a frequency counter to measure the simulation graphic rate
cFrequencyCounter freqCounterGraphics;

cHapticDeviceInfo Falcon = {
        C_HAPTIC_DEVICE_FALCON,
        "Novint Technologies",
        "Falcon",
        8.0,      // [N]
        0.0,      // [N*m]
        0.0,      // [N]
        3000.0,   // [N/m]
        0.0,      // [N*m/Rad]
        0.0,      // [N*m/Rad]
        20.0,     // [N/(m/s)]
        0.0,      // [N*m/(Rad/s)]
        0.0,      // [N*m/(Rad/s)]
        0.04,     // [m]
        cDegToRad(0.0),
        true,
        false,
        false,
        true,
        false,
        false,
        true,
        true
};



#include "WebsocketServer.h"
#include <iostream>
#include <thread>
#include <asio/io_service.hpp>

//The port number the WebSocket server listens on
#define PORT_NUMBER 8080

int main(int argc, char* argv[])
{
        //to keep static screen


        //---------------------------------------------------------------------------
        // INITIALIZATION
        //---------------------------------------------------------------------------
        std::cout << "-----------------------------------" << std::endl;
        std::cout << "Teleoperation" << std::endl;
        std::cout << "-----------------------------------" << std::endl;

        std::cout << "-----------------------------------" << std::endl;
        std::cout << "Master" << std::endl;
        std::cout << "-----------------------------------" << std::endl;


        // initialized deadband classes for force and velocity   IP_master.data() IP_slave.data()
        DBVelocity = new DeadbandDataReduction(VelocityDeadbandParameter);
        DBPosition = new DeadbandDataReduction(PositionDeadbandParameter);

        QueryPerformanceFrequency(&cpuFreq);
        std::cout << "CPU freq: " << (double)cpuFreq.QuadPart / 1000 << std::endl;
```

```
    //--------------------------------------------------------------------------
    // WORLD AND HAPTIC DEVICE
    //--------------------------------------------------------------------------

    // create a new world.
    world = new cBulletWorld();

    // create a haptic device handler
    handler = new cHapticDeviceHandler();

    // get a handle to the first haptic device
    handler->getDevice(hapticDevice, 0);

    // create a tool (cursor) and insert into the world
    tool = new cToolCursor(world);

    // connect the haptic device to the tool
    tool->setHapticDevice(hapticDevice);

    // start the haptic tool - necessary
    tool->start();


    //--------------------------------------------------------------------------
    // create message sender used to control delay and send message
    //--------------------------------------------------------------------------

    // create a thread which starts the main haptics rendering loop
    hapticsThread = new cThread();

    hapticsThread->start(updateHaptics, CTHREAD_PRIORITY_HAPTICS);

    // setup callback when application exits
    atexit(close);


    ////keeps program running until any key is pressed
    system("pause");


    return 0;
}

//==============================================================================
// functions from HapticMaster for the server
//==============================================================================

void close(void)
{
    // stop the simulation
    simulationRunning = false;

    // wait for haptics loop to terminate
    while (!simulationFinished) { cSleepMs(100); }

    // close haptic device
    tool->stop();

    // delete resources
    delete hapticsThread;
    delete world;
    delete handler;
}

void updateHaptics(void)
{

//=============================================================================
============
    // Code from websocket-server-demo

//=============================================================================
============

    //Create the event loop for the main thread, and the WebSocket server
    asio::io_service mainEventLoop;
    WebsocketServer server;

    // Websocket-server-demo uses lambda function in order to send message upon
connection, disconnection, messaging, and instantiating
    // serverThread and inputThread.

    //Register our network callbacks, ensuring the logic is run on the main thread's event loop


    server.connect([&mainEventLoop, &server](ClientConnection conn)
    {
        mainEventLoop.post([conn, &server]()
        {
            std::clog << "Connection opened. There are now " <<
server.numConnections() << " open connections." << std::endl;

            //Send a hello message to the client
            server.sendMessage(conn, "hello", Json::Value());
        });
    });
    server.disconnect([&mainEventLoop, &server](ClientConnection conn)
    {
        mainEventLoop.post([conn, &server]()
        {
```

```
                        std::clog << "Connection closed. There are now " << server.numConnections()
<< " open connections." << std::endl;
                });
        });
        server.message("message", [&mainEventLoop, &server](ClientConnection conn, const
Json::Value& args)
        {
                mainEventLoop.post([conn, args, &server]()
                {

                        std::clog << "message handler on the main thread: payload:" << std::endl;
                        std::cout << args << std::endl;

                });
        });

        //Start the networking thread
        std::thread serverThread([&server]() {
                server.run(PORT_NUMBER);
        });

        Start reading and writing haptic data from\to the master haptic device.
        std::thread inputThread([&server, &mainEventLoop]()
        {

                //====================================================================
                initializations
                simulationRunning = true;
                simulationFinished = false;

                int time_counter = 0;

                //====================================================================


//===================================================================================
====================

                //Ensures communication stat to be printed on a new line.
                bool firstprint = true;

                while (simulationRunning)
                {

                        time_counter++;

                        ///////////////////////////////////////////////////////////////
                        // READ HAPTIC DEVICE
                        ///////////////////////////////////////////////////////////////

                        // update position and orientation of tool
                        tool->updateFromDevice();

                        // read position
                        cVector3d position = tool->getDeviceLocalPos();
                        // read linear velocity
                        cVector3d linearVelocity;
                        hapticDevice->getLinearVelocity(linearVelocity);

                         // update the variables with the newly read values
                        MasterVelocity[0] = MasterVelocity[1] = MasterVelocity[2] = 0.0;
                        for (int i = 0; i < 3; i++)
                        {
                                MasterVelocity[i] = linearVelocity(i);
                                MasterPosition[i] = position(i);
                        }

                        if (time_counter < 100)
                                MasterVelocity[0] = MasterVelocity[1] = MasterVelocity[2] = 0.0;
                        // Apply deadband on velocity
                        DBVelocity->GetCurrentSample(MasterVelocity);
                        DBVelocity->ApplyZOHDeadband(MasterVelocity, &VelocityTransmitFlag);

#pragma region create message and send it
                        ////////////////////////////////////////////////////////////////////////
                        // create message to send
                        ////////////////////////////////////////////////////////////////////////

                        hapticMessageM2S msgM2S;
                        for (int i = 0; i < 3; i++) {
                                msgM2S.position[i] = MasterPosition[i];//modified by TDPA

                                msgM2S.linearVelocity[i] = MasterVelocity[i];//modified by TDPA
                        }

                        __int64 curtime;
                        QueryPerformanceCounter((LARGE_INTEGER*)& curtime);
                        msgM2S.timestamp = curtime;


//=============================================================================
==========

                        //Broadcast the input to all connected clients (is sent on the network thread)
                        Json::Value payload;

                        // there are 33 fields in hapticMessageM2S.


                        // sample stringified JSON object for testing websocketpp on slave-browser
system.
                        /
```

```
{"_MESSAGE_":"data","hapticMessageM2S":[4377508300697,-0.020581520991381136,-0.008066097465685023,0.040221096070821034,2.1244822771173601e-322,1.6714049593151536e-293,0.0,9.9051321773382024e-315,7.5261082099078563e-313,5.0057480953748278e-291,1.3559553104480538e-311,0.0,0.0,1.7111774058343956e-310,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0,0,0,0,0]}

                         */
                          // haptic packet Master to Slave creation
                         int i = 0;

                         payload["hapticMessageM2S"][0] = msgM2S.timestamp;
                         //index 1~3
                         for (int i = 0; i < 3; i++) {
                                 payload["hapticMessageM2S"][i + 1] = msgM2S.position[i];
                         }
                         //index 4~12
                         for (int i = 0; i < 9; i++) {
                                 payload["hapticMessageM2S"][i + 4] = msgM2S.rotation[i];
                         }
                         payload["hapticMessageM2S"][13] = msgM2S.gripperAngle;
                         //index 14~16
                         for (int i = 0; i < 3; i++) {
                                 payload["hapticMessageM2S"][i + 14] = msgM2S.linearVelocity[i];
                         }
                         //index 17~19
                         for (int i = 0; i < 3; i++) {
                                 payload["hapticMessageM2S"][i + 17] = msgM2S.angularVelocity[i];
                         }
                         payload["hapticMessageM2S"][20] = msgM2S.gripperAngularVelocity;
                         payload["hapticMessageM2S"][21] = msgM2S.userSwitches;
                         //index 22~24
                         for (int i = 0; i < 3; i++) {
                                 payload["hapticMessageM2S"][i + 22] = msgM2S.energy[i];
                         }
                         //index 25~27
                         for (int i = 0; i < 3; i++) {
                                 payload["hapticMessageM2S"][i + 25] = msgM2S.waveVariable[i];
                         }
                         payload["hapticMessageM2S"][28] = msgM2S.button0;
                         payload["hapticMessageM2S"][29] = msgM2S.button1;
                         payload["hapticMessageM2S"][30] = msgM2S.button2;
                         payload["hapticMessageM2S"][31] = msgM2S.button3;
                         payload["hapticMessageM2S"][32] = msgM2S.ATypeChange;




                         //send the haptic packet to all connected clients
                         server.broadcastMessage("data", payload);
```

```
//==============================================================================
==========

#pragma endregion

            //-------packet rate------------
            if (time_counter % 1000 == 0)
            {
                // Ensures communication stat to be printed on a new line.
                if (firstprint) {
                    std::cout << std::endl;
                    firstprint = false;
                }

            }

//==============================================================================
=================

    //Start the event loop for the main thread
    asio::io_service::work work(mainEventLoop);
    mainEventLoop.run();

    // exit haptics thread
    simulationFinished = true;
}
```

12.3 C++ Slave Code

```
#include "commTool.h"
#include "config.h"
#include "HapticCommLib.h"

//============================================================
// Exclusive to HapticSlave
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
//============================================================

#include <string.h>
#include <stdlib.h>

#include "chai3d.h"
#include "CBullet.h"
using namespace chai3d;

//-------------------------------------------------------------------------------
// Read Parameters from configuration file
//-------------------------------------------------------------------------------
//set manually for now

ConfigFile cfg("cfg/config.cfg"); // get the configuration file

//============================================================
// Exclusive to HapticSlave
double ForceDeadbandParameter = cfg.getValueOfKey<double>("ForceDeadbandParameter");
//deadband parameter for force data reduction, 0.1 is the default value
double ForceDelay = cfg.getValueOfKey<double>("ForceDelay");
int ControlMode = cfg.getValueOfKey<int>("ControlMode"); // 0: position control, 1:velocity
control
//============================================================


double VelocityDeadbandParameter =
cfg.getValueOfKey<double>("VelocityDeadbandParameter"); //deadband parameter for velcity
data reduction, 0.1 is the default value
double PositionDeadbandParameter =
cfg.getValueOfKey<double>("PositionDeadbandParameter"); //deadband parameter for position
data reduction, 0.1 is the default value
double CommandDelay = cfg.getValueOfKey<double>("CommandDelay");

std::string IP_master = cfg.getValueOfKey<std::string>("MasterIP");
std::string IP_slave = cfg.getValueOfKey<std::string>("SlaveIP");

//============================================================
// Exclusive to HapticSlave
DeadbandDataReduction* DBForce; // data reduction class for force samples
//============================================================
DeadbandDataReduction* DBVelocity; // data reduction class for velocity samples
DeadbandDataReduction* DBPosition; // data reduction class for position samples


//==========================================================================
// Exclusive to HapticSlave
bool ForceTransmitFlag = false; // true: deadband triger false: keep last recently transmitted sample (ZoH)
//==========================================================================

bool VelocityTransmitFlag = false; // true: deadband triger false: keep last recently transmitted sample (ZoH)
bool PositionTransmitFlag = false; // true: deadband triger false: keep last recently transmitted sample (ZoH)

// TDPA variable
double MasterForce[3] = { 0.0, 0.0, 0.0 };
double MasterVelocity[3] = { 0.0, 0.0, 0.0 }; // update 3 DoF master velocity sample (holds the signal before deadband)
double MasterPosition[3] = { 0.0, 0.0, 0.0 }; // update 3 DoF master position sample (holds the signal before deadband)


//==========================================================================
// Exclusive to HapticSlave
//---------------------------------------------------------------------------
// Teleoperation control
//---------------------------------------------------------------------------
double xsd[3] = { 0, 0, 0 };  //desired slave position
double xs[3] = { 0, 0, 0 };   //actual slave position
double fs_prev[3] = { 0,0,0 };  //slave force from last time stamp
double xs_err_prev[3] = { 0,0,0 }; //previous position error from last time stamp
double A = 0, B = 0, C = 0;   //control factor A, B, C. PD controller with IIR filter
//==========================================================================

//---------------------------------------------------------------------------
// DECLARED VARIABLES
//---------------------------------------------------------------------------

// a world that contains all objects of the virtual environment
cBulletWorld* world;

// a haptic device handler
cHapticDeviceHandler* handler;

// a pointer to the current haptic device
cGenericHapticDevicePtr hapticDevice;

// a virtual tool representing the haptic device in the scene
cToolCursor* tool;
```

```
// a flag to indicate if the haptic simulation currently running
bool simulationRunning = false;

// a flag to indicate if the haptic simulation has terminated
bool simulationFinished = true;

// a frequency counter to measure the simulation haptic rate
cFrequencyCounter freqCounterHaptics;

// a frequency counter to measure the simulation graphic rate
cFrequencyCounter freqCounterGraphics;

// haptic thread
cThread* hapticsThread;

LARGE_INTEGER cpuFreq;


cHapticDeviceInfo Falcon = {
        C_HAPTIC_DEVICE_FALCON,
        "Novint Technologies",
        "Falcon",
        8.0,      // [N]
        0.0,      // [N*m]
        0.0,      // [N]
        3000.0,   // [N/m]
        0.0,      // [N*m/Rad]
        0.0,      // [N*m/Rad]
        20.0,     // [N/(m/s)]
        0.0,      // [N*m/(Rad/s)]
        0.0,      // [N*m/(Rad/s)]
        0.04,     // [m]
        cDegToRad(0.0),
        true,
        false,
        false,
        true,
        false,
        false,
        true,
        true
};

bool startFlag = false;



//------------------------------------------------------------------------------
// DECLARED FUNCTIONS
//------------------------------------------------------------------------------
```

```
// this function contains the main haptics simulation loop
void updateHaptics(void);


// this function closes the application
void close(void);



//==============================================================
// Exclusive to HapticSlave

void deviceInit(double, double, double, cGenericHapticDevicePtr);

//----------------------------------------------------------------------
// DECLARED MACROS
//----------------------------------------------------------------------
// convert to resource path
#define RESOURCE_PATH(p)    (char*)((resourceRoot+string(p)).c_str())

//==============================================================
// Header files used by Websocketpp
//================================================================
#include "WebsocketServer.h"
#include <iostream>
#include <thread>
#include <asio/io_service.hpp>

//The port number the WebSocket server listens on
#define PORT_NUMBER 8080

int main()
{

    //----------------------------------------------------------------------
    // INITIALIZATION
    //----------------------------------------------------------------------
    std::cout << "---------------------------------" << std::endl;
    std::cout << "Teleoperation (Slave Side)" << std::endl;
    std::cout << "---------------------------------" << std::endl;


    // initialized deadband classes for force and velocity   IP_master.data() IP_slave.data()
    DBVelocity = new DeadbandDataReduction(VelocityDeadbandParameter);
    DBPosition = new DeadbandDataReduction(PositionDeadbandParameter);



    // Exclusive to HapticSlave
    DBForce = new DeadbandDataReduction(ForceDeadbandParameter);
    //----------------------------------------------------------------------
    QueryPerformanceFrequency(&cpuFreq);
    std::cout << "CPU freq: " << (double)cpuFreq.QuadPart / 1000 << std::endl;

    //-----------------------------------------------------------------------
    // WORLD AND HAPTIC DEVICE
    //-----------------------------------------------------------------------

    // create a new world.
    world = new cBulletWorld();

    // create a haptic device handler
    handler = new cHapticDeviceHandler();

    //Note that master device corresponds to 0 and slave device corresponds to 1
    handler->getDevice(hapticDevice, 1);

    //initialize haptic device
    deviceInit(0.0, 0.0, 0.0, hapticDevice);

    // create a tool (cursor) and insert into the world
    tool = new cToolCursor(world);

    // connect the haptic device to the tool
    tool->setHapticDevice(hapticDevice);

    // start the haptic tool - necessary
    tool->start();

    //-----------------------------------------------------------------------
    // create message sender used to control delay and send message. Then, simulation starts.
    //-----------------------------------------------------------------------

    // create a thread which starts the main haptics rendering loop
    hapticsThread = new cThread();


    hapticsThread->start(updateHaptics, CTHREAD_PRIORITY_HAPTICS);

    // setup callback when application exits
    atexit(close);

    // keeps program running until any key is pressed
    system("pause");


    return 0;
}

void close(void)
{
    // stop the simulation
```

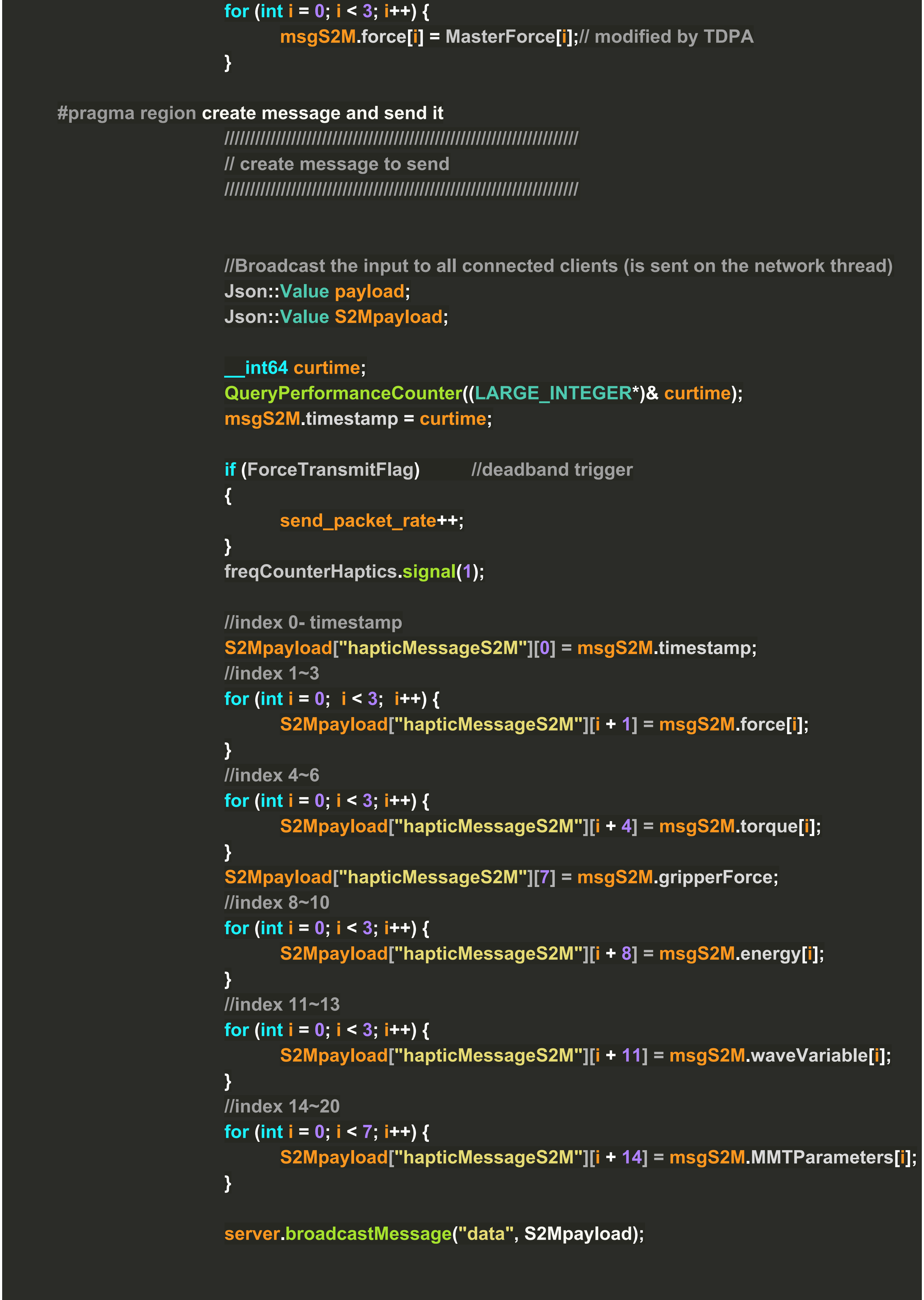
```
        simulationRunning = false;

        // wait for graphics and haptics loops to terminate
        while (!simulationFinished) { cSleepMs(100); }

        // close haptic device
        tool->stop();

        // delete resources
        delete hapticsThread;
        delete world;
        delete handler;
}

void updateHaptics(void)
{

         //Create the event loop for the main thread, and the WebSocket server
        asio::io_service mainEventLoop;
        WebsocketServer server;

        server.connect([&mainEventLoop, &server](ClientConnection conn)
        {
                mainEventLoop.post([conn, &server]()
                {
                        std::clog << "Connection opened. There are now " <<
server.numConnections() << " open connections." << std::endl;

                        //Send a hello message to the client
                        server.sendMessage(conn, "hello", Json::Value());
                });
        });
        server.disconnect([&mainEventLoop, &server](ClientConnection conn)
        {
                mainEventLoop.post([conn, &server]()
                {
                        std::clog << "Connection closed. There are now " << server.numConnections()
<< " open connections." << std::endl;
                });
        });
        server.message("message", [&mainEventLoop, &server](ClientConnection conn, const
Json::Value& args)
        {
                int time_counter = 0;

                mainEventLoop.post([conn, args, &server, &time_counter]()
                {

                        if (!startFlag)
                        {
                                        std::cout << "first commands received! " << std::endl;
```

```
                    startFlag = true;

                    cVector3d tempPos;
                    hapticDevice->getPosition(tempPos);
                    for (int i = 0; i < 3; ++i)
                    {
                        xs[i] = xsd[i] = tempPos(i);
                    }

            }

            hapticMessageM2S msgM2S;

            //need a converter from const Json:: Value to double
            //first  Json:: Value to string
            //string to double
            std::string::size_type sz;

            //index 0- timestamp
            msgM2S.timestamp = std::stod( args["hapticMessageM2S"][0].asString(), &sz);

            // for now the parameters that are not used are not being read
            // only the linear velocity and position are read

            //read the linear velocity from the received packet into the msgM2S structure
            for (int i = 0; i < 3; i++) {
                msgM2S.linearVelocity[i] = std::stod(args["hapticMessageM2S"][i +
1].asString(), &sz);
            }

            // for now the parameters that are not used are not being read
            // only the linear velocity and position are read



            // read linear velocity into a chai3d vector
            cVector3d linearVelocity(msgM2S.linearVelocity[0], msgM2S.linearVelocity[1],
msgM2S.linearVelocity[2]);

            // saving linear velocity info to MasterVelocity
            memcpy(MasterVelocity, msgM2S.linearVelocity, 3 * sizeof(double));

            //declaring slave force initially set to zero.
            cVector3d input_force(0, 0, 0);




        /////////////////////////////////////////////////////////////////////
        // COMPUTE AND APPLY teleoperation control FORCES
        /////////////////////////////////////////////////////////////////////
            cVector3d tempPos;
```

```
hapticDevice->getPosition(tempPos);
xs[0] = tempPos.x();
xs[1] = tempPos.y();
xs[2] = tempPos.z();

xsd[0] += 0.001 * MasterVelocity[0];
xsd[1] += 0.001 * MasterVelocity[1];
xsd[2] += 0.001 * MasterVelocity[2];


// PD controller for calculating force
double xs_err[3] = { xsd[0] - xs[0],  xsd[1] - xs[1], xsd[2] - xs[2] };   //compute slave force


// max force limited output to -1 to 2.
double abs_val = 2; // max 8
double weak_y_force = A * xs_err[1] - B * xs_err_prev[1] - C * fs_prev[1];
if (weak_y_force > abs_val) {
        weak_y_force = abs_val;
}
if (weak_y_force  < -abs_val) {
        weak_y_force = -abs_val;
}

double weak_z_force = A * xs_err[2] - B * xs_err_prev[2] - C * fs_prev[2];
if (weak_z_force > abs_val) {
        weak_z_force = abs_val;
}
if (weak_z_force < -abs_val) {
        weak_z_force = -abs_val;
}
input_force.set(
        0,
        weak_y_force,
        0

);

for (int i = 0; i < 3; ++i)
{
        xs_err_prev[i] = xs_err[i];
        fs_prev[i] = input_force(i);
}





/////////////////////////////////////////////////////////////////////
// APPLY FORCE TO THE DEVICE AND ENSURE 1ms SAMPLING RATE
/////////////////////////////////////////////////////////////////////
```

```
            cVector3d tempForce(input_force);
            if (tempForce.length() > 20.0)
            {
                tempForce = tempForce * 8.0 / tempForce.length();
            }


            std::cout << " input Force: x= " << tempForce(0) << ",  y=  " << tempForce(1) <<
", z=" << tempForce(2) << std::endl;
            tool->setDeviceLocalForce(tempForce);
            tool->applyToDevice();

        });

    });

    //Start the networking thread
    std::thread serverThread([&server]() {
        server.run(PORT_NUMBER);
    });


    // Approach taken by the slave main.cpp is
    //          1, Send Forces back to master device so haptic feedback to the master
    //          2, (Done) Convert the received position/velocity data to become force and
apply to device

    // temporarily adding variables to lamda list
    std::thread inputThread([&server, &mainEventLoop](){



        simulationRunning = true;
        simulationFinished = false;

        int send_packet_rate = 0;
        //==================================================================


        // Ken - for sending foce data. random values put in to not confuse with the
erroneous 0, 0,0 being printed on console
        cVector3d read_force(0, 0, 0); //slave force

        // this test function doesn't work either, unless without out getPosition() below.
        cVector3d linearVelocity;
        cVector3d testpos;

        double cur_f_x = 0;
        double cur_f_y = 0;
        double cur_f_z = 0;
```

```
		// Ken  - Essentially HapticMaster's updateHaptics() functions with server project's input transmission feature inserted into part of overall transmission.
		while (simulationRunning)
		{

			/////////////////////////////////////////////////////////////////////
			// get force and apply deadband approach
			/////////////////////////////////////////////////////////////////////


			 hapticDevice->getLinearVelocity(linearVelocity);


			 hapticDevice->getPosition(testpos);


			 hapticDevice->getForce(read_force);

			 if (read_force(0) != 0 && read_force(1) != 0 && read_force(2) != 0 && cur_f_x != read_force(0) && cur_f_y != read_force(1) && cur_f_z != read_force(2) ) {

				 cur_f_x = read_force(0);
				 cur_f_y = read_force(1);
				 cur_f_z = read_force(2);
			 }



			 cVector3d F_debug(read_force);

			 MasterForce[0] = -1 * F_debug.x();
			 MasterForce[1] = -1 * F_debug.y();
			 MasterForce[2] = -1 * F_debug.z();

			 // Slave side: Perceptual deadband data reduction is applied
			 DBForce->GetCurrentSample(MasterForce); // pass the current sample for DB data reduction
			 DBForce->ApplyZOHDeadband(MasterForce, &ForceTransmitFlag); // apply DB data reduction

			// Slave side: Perceptual deadband data reduction is applied
			DBForce->GetCurrentSample(MasterForce); // pass the current sample for DB data reduction
			DBForce->ApplyZOHDeadband(MasterForce, &ForceTransmitFlag); // apply DB data reduction

			/////////////////////////////////////////////////////////////////////
			// Send Forces
			/////////////////////////////////////////////////////////////////////
			hapticMessageS2M msgS2M;
```



```
            for (int i = 0; i < 3; i++) {
                msgS2M.force[i] = MasterForce[i];// modified by TDPA
            }

#pragma region create message and send it
            //////////////////////////////////////////////////////////////////////
            // create message to send
            //////////////////////////////////////////////////////////////////////



            //Broadcast the input to all connected clients (is sent on the network thread)
            Json::Value payload;
            Json::Value S2Mpayload;

            __int64 curtime;
            QueryPerformanceCounter((LARGE_INTEGER*)& curtime);
            msgS2M.timestamp = curtime;

            if (ForceTransmitFlag)        //deadband trigger
            {
                send_packet_rate++;
            }
            freqCounterHaptics.signal(1);

            //index 0- timestamp
            S2Mpayload["hapticMessageS2M"][0] = msgS2M.timestamp;
            //index 1~3
            for (int i = 0;  i < 3;  i++) {
                S2Mpayload["hapticMessageS2M"][i + 1] = msgS2M.force[i];
            }
            //index 4~6
            for (int i = 0; i < 3; i++) {
                S2Mpayload["hapticMessageS2M"][i + 4] = msgS2M.torque[i];
            }
            S2Mpayload["hapticMessageS2M"][7] = msgS2M.gripperForce;
            //index 8~10
            for (int i = 0; i < 3; i++) {
                S2Mpayload["hapticMessageS2M"][i + 8] = msgS2M.energy[i];
            }
            //index 11~13
            for (int i = 0; i < 3; i++) {
                S2Mpayload["hapticMessageS2M"][i + 11] = msgS2M.waveVariable[i];
            }
            //index 14~20
            for (int i = 0; i < 7; i++) {
                S2Mpayload["hapticMessageS2M"][i + 14] = msgS2M.MMTParameters[i];
            }

            server.broadcastMessage("data", S2Mpayload);
```

```
//===================================================================================
==========

#pragma endregion

                }

        });

//===================================================================================
=================

        //Start the event loop for the main thread
        asio::io_service::work work(mainEventLoop);
        mainEventLoop.run();

        // exit haptics thread
        simulationFinished = true;
}


Function to calculate the gains (z domain gains) A, B, C from (s domain gains) K, Ke
Using Tustin's approximation

s domain transfer function :    F = (K*error + Ke*error_dot)/(tau * s + 1)
 // with current arguments, the parameter results in A= 2619.05,  B=  2142.86, and C=-0.52381.

where,          K - Proportional Gain
Ke - Derivative Gain
tau - Low pass filter parameter
*/
void reCalcGains(double& K, double& Ke, double& A, double& B, double& C) {
        double tau = 0.0016;
        double T = 0.001;
        A = (2 * Ke + K * T) / (2 * tau + 1 * T);
        B = -1 * (-2 * Ke + K * T) / (2 * tau + 1 * T);
        C = (-2 * tau + 1 * T) / (2 * tau + 1 * T);
}

/*
initialize parameters from the starting position of the device
*/
void deviceInit(double x, double y, double z, cGenericHapticDevicePtr device)
{
        std::cout << "deviceInit() received! " << std::endl;

        cVector3d pos;
        device->getPosition(pos);
        for (int i = 0; i < 3; ++i)
        {
            xs[i] = pos(i);
            xsd[i] = pos(i);
        }

        double K = 1000, Ke = 5;

        reCalcGains(K, Ke, A, B, C);
        std::cout << " parameters: A= " << A << ",  B=  " << B << ", C=" << C << std::endl;
    }
```

12.4 Signaling Code

```
/*
 *  This project was inspired by the Munge SDP sample example which can
 *  be found on the WebRTC website. The copy right from that project can
 *  be seen below.
 *  Copyright (c) 2015 The WebRTC project authors. All Rights Reserved.
 *  Use of this source code is governed by a BSD-style license
 *  that can be found in the LICENSE file in the root of the source tree.
 * Authors
    Ken Iiyoshi
    Mahrukh Tausseef
    Ruth Gebremedhin
 * Date
    Sunday, May 10, 2020
 */

var isChannelReady = false;
var isInitiator = false;
var isStarted = false;
var room = 'foo';

// ************Begin Socket*****************
var socket = io.connect();

if (room !== '') {
  socket.emit('create or join', room); //sends to the server
  console.log('Attempted to create or  join room', room);
}

socket.on('created', function(room) {
  console.log('Created room ' + room);
  isInitiator = true;
});

socket.on('full', function(room) {
  console.log('Room ' + room + ' is full');
});

socket.on('join', function (room){
  console.log('Another peer made a request to join room ' + room);
  console.log('This peer is the initiator of room ' + room + '!');
  isChannelReady = true;
});

socket.on('joined', function(room) {
  console.log('joined: ' + room);
  isChannelReady = true;
});

socket.on('log', function(array) {
  console.log.apply(console, array);
});
```

```
////////////////////////////////////////////////

function sendMessage(message) {
  console.log('Client sending message: ', message);
  socket.emit('message', message);
}

// This client receives a message
socket.on('message', function(message) {
  console.log('Client received message:', message);
  if (message === 'got user media') {
    check_creatPeerConnectoin();
  } else if (message.type === 'offer') {
    if (!isInitiator && !isStarted) {
       check_creatPeerConnectoin();

      createAnswerButton.disabled = false;
      setAnswerButton.disabled = false;
      createOfferButton.disabled = true;
      setOfferButton.disabled = true;
    }
    localPeerConnection.setRemoteDescription
    (new RTCSessionDescription(message));

    createAnswerButton.onclick = createAnswer;
    setAnswerButton.onclick = setAnswer;

  } else if (message.type === 'answer' && isStarted) {
    localPeerConnection.setRemoteDescription
    (new RTCSessionDescription(message));
  } else if (message.type === 'candidate' && isStarted) {
    var candidate = new RTCIceCandidate({
      sdpMLineIndex: message.label,
      candidate: message.candidate
    });
    localPeerConnection.addIceCandidate(candidate);
  }
    else if (message === 'bye' && isStarted) {
    hangup();
  }
  });

//******************End Socket***************
//checks if the signaling was successful and creates the RTCPeerConnection
//object if it was successful
function check_creatPeerConnectoin() {
  console.log('>>>>>>> check_creatPeerConnectoin() ', isStarted, localStream,
  isChannelReady);
  if (!isStarted && typeof localStream !== 'undefined' && isChannelReady) {
    console.log('>>>>>> creating peer connection');
    createPeerConnection();
    localStream.getTracks().forEach(track =>
    localPeerConnection.addTrack(track, localStream));
    console.log('Adding Local Stream to peer connection');
    isStarted = true;
    console.log('isInitiator', isInitiator);

    if (isInitiator) {
      createOfferButton.onclick = createOffer;
      setOfferButton.onclick = setOffer;
    }
  }
}
```

12.5 HAV Network Documentation

The goal of this documentation is to instruct how to run the WebRTC-based localhost simulation of the WG haptic handshake, as well as to provide a high-level understanding of the how the simulation works. The documentation starts from the following page.

Localhost WG Haptic Handshake Simulation Documentation ver 2

Project: Group 04 (Design of a haptic wearable for Tele-operation) - NYU Abu Dhabi
Version: 1 - Updated on 2020 May 08 at 17:28 by Ken Iiyoshi

Description

The goal of this documentation is to instruct how to run the WebRTC-based localhost simulation of the WG haptic handshake, as well as to provide a high-level understanding of the how the simulation works.

Overview of this Documentation

- The goal of this documentation is to instruct how to run the WebRTC-based localhost simulation of the WG haptic handshake, as well as to provide a high-level understanding of the how the simulation works. This complements the low-level documentation that already exists for libraries that make up the simulation.
- For understanding/modifying the code further, the following skills would be useful:
    - Basics of socket programming for both JavaScript and C++ based section.
    - OOP in C++.
    - Understanding threads for haptic devices in C++.
    - Asynchronous programming for JavaScript code.
    - Basics of HTML, on how the content of the browser's graphical user interfaces (GUI) with JavaScript-based functionalities.
    - Basics of CSS, on how layout/color/formatting works on the GUI.

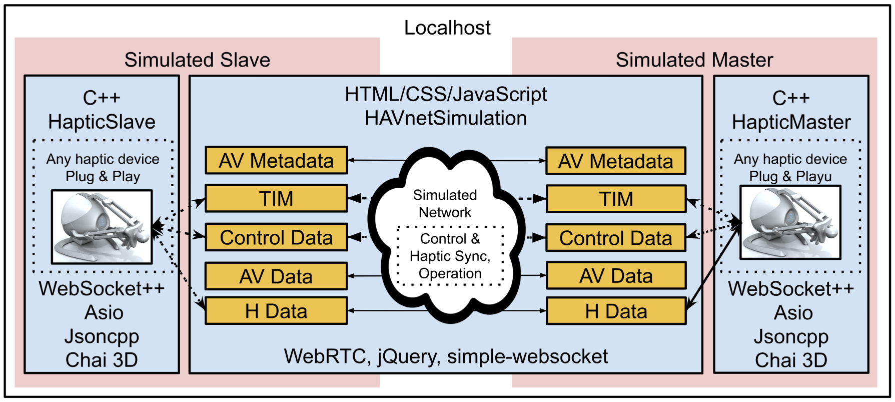
Overview of Implemented HAV Network

Aside from developing the haptodont application, our capstone focuses on creating a network that allows HAV communication between haptodont systems. As a first step, I implemented a locally hosted HAV network simulation. Ideally, this should be done in one program. However, because WebRTC is being used to simulate HAV network and because haptic data can only be accessed through a C++ based program, the simulation had to be divided into three programs: C++ HapticSlave, HTML/CSS/JavaScript HAVnetSimulation, and C++ HapticMaster, which are shown as separate blue blocks in Figure 1. The HTML/CSS/JavaScript browser communicates between simulated slave and master, including haptic data which is provided by the C++ programs that are interfaced to Novint Falcon haptic devices. The nature of haptic data, combined with AV data, are determined by AV metadata and TIM during HAV handshaking phase, and by control data during operation phase.

> Several existing projects and libraries are being merged to create a set of three programs. The browser consists of WebRTC and C++ WebSocket Server Demo's client project. The client project consists of jQuery and simple-websocket. jQuery simplifies traditionally verbose JavaScript expressions while simple-websocket is used to receive websocket data.

The C++ programs both consist of C++ WebSocket Server Demo's server project, Chai 3D, and Haptic Codec provided by Xiao Xu from TUM (shared only within AIM Lab). The server projects consists of WebSocket++, Asio, and Jsoncpp. WebSocket++ allows data to be sent on WebSocket via C++, Asio aids the networking process through ASynchronous Input/Outputs, and Jsoncpp allows creation and management of JSON objects in C++. Chai 3D is being used to sense and actuate the Novint Falcon devices. Simplified version of Haptic Codec is being used to form haptic objects that is converted into stringified JSON via Jsoncpp and WebSocket++, as well as formulate realtime statistics and processing of haptic data communication.

Most of the work during this J-term was put into enabling haptic data transmission between the simulated master's browser and C++ programs, which are indicated red borders. As a next step, haptic data will be passed through WebRTC's RTCDataChannel to complete HAV operation, which is indicated in blue arrows and text. Each project will be discussed in their own sections below.



Existing features are in solid black lines, while those that are yet to be completed are in dotted lines.

Figure 1. Model of Localhost WG Haptic Handshake Simulation

Accessing and Running Files

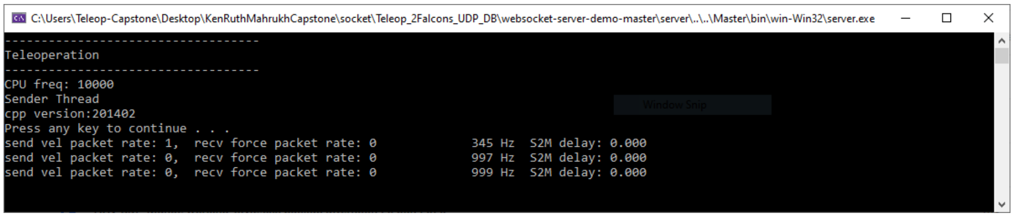
Figure 1, as well as backup files for the entire project, can be accessed from this capstone team drive folder (view only): https://drive.google.com/drive/folders/1BxGKWgv7EUzhzJoHQZNoPdE15y7R6_0O?usp=sharing

The current implementation can communicate haptic data within the simulated master. To run this implementation, run the three projects in the following order. Note that this project was compiled in VS2015, so using any other version would require recompilation of the entire project.

1. HapticSlave: socket > Teleop_2Falcons_UDP_DBn > Slave > Haptic.sln

- set **HapticSlave** as a start up project.
- Not necessary for now, as haptic communication between the slave and browser is not implemented yet.
- The console pop up here does not print any communication rates until haptic connection is established with master via the browser.

2. HapticMaster: socket > Teleop_2Falcons_UDP_DB > Master > Haptic.sln

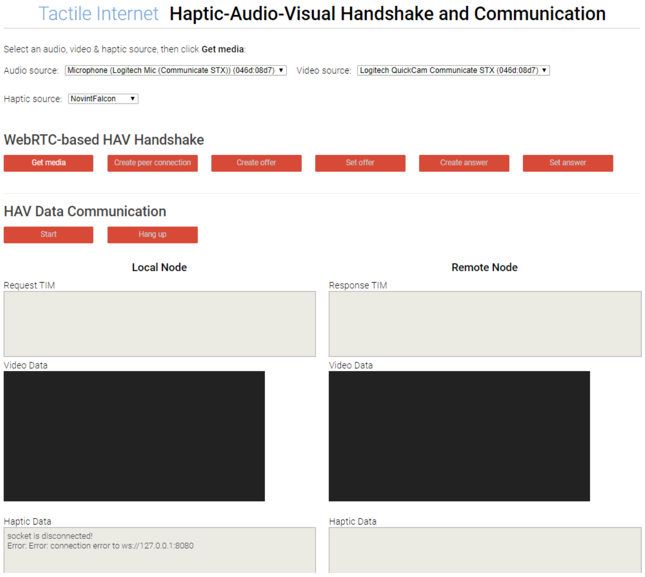
- set **server** as a start up project.
- A console should pop up and start printing packet communication rates once it detects the Novint Falcon device.
- 

C:\Users\Teleop-Capstone\Desktop\KenRuthMahrukhCapstone\socket\Teleop_2Falcons_UDP_DB\websocket-server-demo-master\server\..\..\Master\bin\win-Win32\server.exe

```
-----------------------------------
Teleoperation
-----------------------------------
CPU freq: 10000
Sender Thread
cpp version:201402
Press any key to continue . . .
send vel packet rate: 1,  recv force packet rate: 0              345 Hz  S2M delay: 0.000
send vel packet rate: 0,  recv force packet rate: 0              997 Hz  S2M delay: 0.000
send vel packet rate: 0,  recv force packet rate: 0              999 Hz  S2M delay: 0.000
```

- Notice that the parent folder and start up project for Haptic Master is different from HapticSlave. The context behind this will be discussed later.

3. HAVnetSimulation: socket > HAVnetSimulation.html.

- The browser's interface is similar to that of WebRTC's Munge SDP sample program, which demonstrates the AV handshake implemented via WebRTC. HAVnetSimulation added haptic features to the SDP Offer/Answer blocks, renaming it as Request/Response TIM. It also added a set of blocks for viewing haptic data stream in raw form.
- 

Tactile Internet Haptic-Audio-Visual Handshake and Communication

Select an audio, video & haptic source, then click **Get media**:

Audio source: Microphone (Logitech Mic (Communicate STX)) (046d:08d7) Video source: Logitech QuickCam Communicate STX (046d:08d7)

Haptic source: NovintFalcon

WebRTC-based HAV Handshake

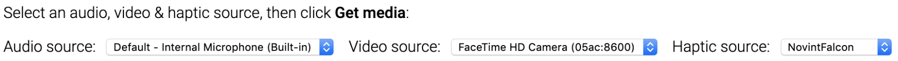
HAV Data Communication

socket is disconnected!
Error: Error: connection error to ws://127.0.0.1:8080

- Currently, loading the browser immediately starts haptic data communication between HapticMaster. Very large set of data is displayed on the local node column as well as on HapticMaster's console window, both of which usually must be stopped by halting communication from HapticMaster's side. This is to prevent unresponsiveness due to need in printing large amount of data within the browser.
- The goal within this system is to, after completing HAV handshake and clicking the start button, have the slave and master Novint Falcon devices move together.
- As HAVnetSimulation involves communication of control parameters and data for not only haptic, but also audial and visual data, the description below will cover the key functions for all three media.

Input selection

Audio, video, and haptic data source can all be manually selected before starting connections.

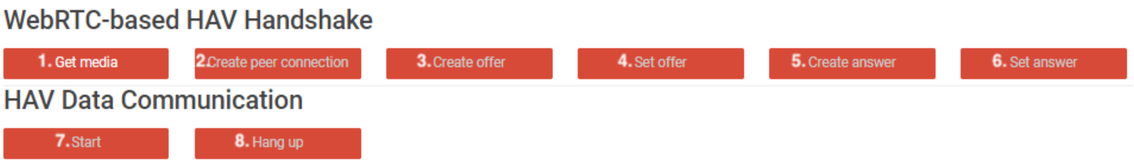


Select an audio, video & haptic source, then click **Get media**:

Audio source: Default - Internal Microphone (Built-in) Video source: FaceTime HD Camera (05ac:8600) Haptic source: NovintFalcon

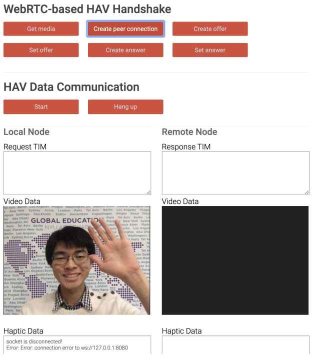
Buttons

Although HAVnetSimulation can automatically begin HAV communication process, the process has been divided into input selection and several on-screen buttons for the purpose of user debugging. The order of button press is shown in the annotated diagram of the buttons, followed by explanation of key features for each button:



1. Get media

Pressing the "Get media" button will activate the user's webcam, outputting video data on the browser. The browser will then highlight the next button to press, which is "Create peer connection", as shown below. Note that the textbox under "Haptic Data" will show the error will appear only if the haptic device is not connected to the localhost. Console logs are present.



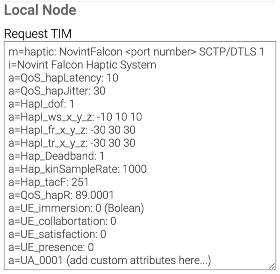
```
socket is disconnected!
Error: Error: connection error to ws://127.0.0.1:8080
```

2. Create peer connection

GUI–wise, no change occurs. As with the first step though, console logs are present.

3. Create offer

This will populate the "Request TIM" text box with the HSDPmediaDescription object. Console logs are present.



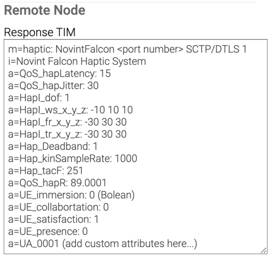
```
m=haptic: NovintFalcon <port number> SCTP/DTLS 1
i=Novint Falcon Haptic System
a=QoS_hapLatency: 10
a=QoS_hapJitter: 30
a=HapI_dof: 1
a=HapI_ws_x_y_z: -10 10 10
a=HapI_fr_x_y_z: -30 30 30
a=HapI_tr_x_y_z: -30 30 30
a=Hap_Deadband: 1
a=Hap_kinSampleRate: 1000
a=Hap_tacF: 251
a=QoS_hapR: 89.0001
a=UE_immersion: 0 (Bolean)
a=UE_collabortation: 0
a=UE_satisfaction: 0
a=UE_presence: 0
a=UA_0001 (add custom attributes here...)
```

4. Set offer

GUI–wise, no change occurs. As with the first three step though, console logs are present.

5. Create answer

This will populate the "Respond TIM" text box with the HSDPmediaDescription object. Notice that the text box content is slightly different from that of "Request TIM" on line 3. This difference is processed and processed programmatically by the time handshake starts. Console logs are present.



```
m=haptic: NovintFalcon <port number> SCTP/DTLS 1
i=Novint Falcon Haptic System
a=QoS_hapLatency: 15
a=QoS_hapJitter: 30
a=HapI_dof: 1
a=HapI_ws_x_y_z: -10 10 10
a=HapI_fr_x_y_z: -30 30 30
a=HapI_tr_x_y_z: -30 30 30
a=Hap_Deadband: 1
a=Hap_kinSampleRate: 1000
a=Hap_tacF: 251
a=QoS_hapR: 89.0001
a=UE_immersion: 0 (Bolean)
a=UE_collabortation: 0
a=UE_satisfaction: 1
a=UE_presence: 0
a=UA_0001 (add custom attributes here...)
```

6. Set answer

This will activate the user's webcam on the simulated remote end, outputting video data on the browser. Console logs are present.

Video Data

Video Data

7. Start

Nothing happens at the moment, since as of now, HAV communication is programmed to start right after "Set answer" is pressed.

8. Hang up

Disconnects and shutdown any communication.

Program for Debugging

One may want to know if a problem is caused by C++ or JavaScript. In this case, HapticSlave and HapticMaster can communicate through JavaScript server simulation that is strictly for localhost (i.e. not extendable to multi-computer network simulation due to JavaScript browsers not supporting UDP). To run the demo with this server, go to and run socket > JavaScript > js_server (open this folder in VS Code, and run main.js via its built-in terminal. i.e. press Ctrl+F5 to do this). This should immediately begin the haptic communication. i.e. moving either of the falcon device will result in mirrored movement on the other device. Notice that the haptic communication is robust to termination of js_server. i.e. the communication goes on as long as js_server is rerun. During downtime, one of the device can be moved, while the other one is stationary. Note that the same phenomenon occurs when terminating HapticMaster. However, terminating HapticSlave will irreversibly brake the connection. Also, UDP message from C++ devices seem to be sent only when the falcon devices move.

Key Features in HapticMaster

HapticMaster was built by inserting the master side of the Haptic codec into C++ WebSocket Server Demo's server project. This was done in order to utilize functionalities from both projects without creating any external symbolic errors (i.e. inserting the server project into the haptic codec introduced these errors). As a result, the start up project for the Haptic solution that contains HapticMaster is not the HapticMaster Project, but actually the server project which is located in socket > Teleop_2Falcons_UDP-DB > websocket-server-demo-master > server > server.vcxproj.

The main function for server.vcxproj is in server.cpp. The code is around 600 lines long, most of which is from a Haptic Codec and the remainder is the server project from C++ WebSocket Server Demo. Most importantly Haptic Codec, in addition to the main function setting up a queue used for haptic communication, spawns a thread for sending and receiving haptic packets through the queue. While the main function is around 100 few lines of code, the thread is around 300 lines of code put under a function called `updateHaptics()`. The following subsection documents key features of `updateHaptics()` function in more detail. Note that HapticMaster's code is still in the process of documentation and reorganization.

updateHaptics()

First, the function declares `asio::io_service` object as `mainEventLoop` and `WebsocketServer` object as `server`. They are used in most of the functions that are invoked within the `updateHaptics()` function. One of these functions is serverThread(), which starts the networking thread on a designated port number (8080 for now). The next function, `inputThread()`, starts reading and writing haptic data from\to the master haptic device. Finally, the function starts the event loop for the main thread through `asio::io_service::work` `work()` and `mainEventLoop.run()` function. The `updateHaptics()` function keeps running until the user terminates the main function through any keypress.

inputThread()

This WebSocket++-based function wraps the Haptic Codec code responsible for sending and receiving haptic packets through the queue. Through this wrapper, the C++ program can send haptic data to the browser, as opposed to the terminal-based JavaScript UDP server.

A set of sub-functions runs in the function's 200 lines long `while` loop during the simulation. The loop first reads from the master haptic device. Then, it creates a message to be sent, pushing it into "send queues", preparing to send it through sender thread. The loop then checks "receive queues". Finally, the received data is applied to the master device for actuation.

For now, the haptic codec packet named `hapticMessageM2S` is converted into `Json:Value payload` object so that `server.broadcastMessage("data", payload)` function can stringify and send `payload` with Jsoncpp functions. This conversion was hardcoded. The goal is to automate this conversion in order to accommodate various haptic devices.

Key Features in HapticSlave

Most of the features in HapticSlave are similar to that of HapticMaster. However, it sends haptic data as force, as opposed to velocity in HapticMaster. Also, its message packet name is `hapticMessageS2M`, not `hapticMessageM2S`.

Note that HapticSlave and HapticMaster follows the structure of a 2 channel teleoperation system. This means that no additional force sensors are equipped on both master and slave haptic devices. The master sends designed velocity and position to the slave. On the slave side, a PD controller generates the control force `fs_prev` to drive the motion of slave. At the same time, the slave control force `fs_prev` is transmitted back to the master and displayed to the user. So there is no environment force or user-inputted force. If the user input force or environment contact force is needed, there needs to be additional force sensors on both sides and the teleoperation becomes a 4-channel system.

Based on the 2 channel teleoperation, `hapticDevice->getForce()` on the slave side returns the last control force `fs_prev`, which is the force sent back to the master according to the deadband coder. The user inputted movement is the same as the motion read by the `hapticDevice->getPosition()` and `hapticDevice->getLinearVelocity()` functions, since we assume that the user is tightly holding the device.

`hapticDevice->getPosition(tempPos)` is used along with MasterVelocity for the PD controller in `input_force.set()`, which is sent to the master, enabling the bidirectional communication without force sensors. This is why HpaticMaster side sends velocity and the HapticSlave sends force; since that is necessary for a haptic device that can only actuate based on force data and measure position/velocity data.

Key Features in HAVnetSimulation

HAVnetSimulation was built on top of HTML, CSS, and JavaScript files from WebRTC's Munge SDP Sample program by adding HTML and JavaScript code from C++ WebSocket Server Demo's client project. The documentation aims to not only explain the key features of Munge SDP, but also those from WebSocket Server Demo, and most importantly haptic-communication related code unique to HAVnetSimulation.

HTML & CSS-based browser Graphical User Interface (GUI)

The code inside `<head>` tags assigns metadata, browser window title/icon, and addresses for fonts, CSS file, and JavaScript files.

The code inside `<body>` tags creates the on-screen features seen on the GUI, and makes additional JavaScript file importations. The features are title text, drop-down boxes for selecting HAV sources, buttons for setting HAV WebRTC communication, buttons for initiating the handshake, text boxes for displaying TIM and haptic data, and display boxes for video data. Each of these features are contained in `<div>` tags with corresponding ids. The CSS file will refer to these ids for formatting and styling.

Notice that the HTML code interfaces JavaScript through its features on the browser screen. The CSS code simply decorates these features. This means that in the context of HAVnetSimulation, HTML and CSS-based sections pertain only to the presentation aspect of it's HAV communication system.

JavaScript-based HAV data communication

From main.js, the browser first executes `document.addEventListener( 'DOMContentLoaded' , init )`. This simply means that it will run the `init()` function once the browser loaded. The rest of main.js is written inside this function. The duration for running this function is measured and outputted toconsole log accordingly. In fact, JavaScript console logs is used extensively, which complements the debugging process. Note that as long as one follows procedure in the documentation, the console log will out put no error or warnings.

The const declarations in the beginning of the `init()` function connects the HTML-based features to JavaScript. Immediately after are assignments of functions that run when the HTML-based features are interacted with. Then, more variables are declared to be used for WebRTC-based processes.

Multiple functions can occur simultaneously after one another due to asynchronous programming.

Receiving HAV Data from HapticMaster

HAVnetSimulation receives haptic data from HapticMaster using its `SocketWrapper` class, an existing implementation from WebSocket Server Demo. The browser first instantiates the `SocketWrapper` class named `socket` under a designated port number, `"ws://127.0.0.1:8080"`. `ws` stands for WebSockets. Port 8080 is typically used for a personally hosted (localhost) web server. A function from `socket` will then connect with HapticMaster. Only after this, another function can obtain haptic data from HapticMaster, stored into a variable named `hapticMsgFromMaster`.

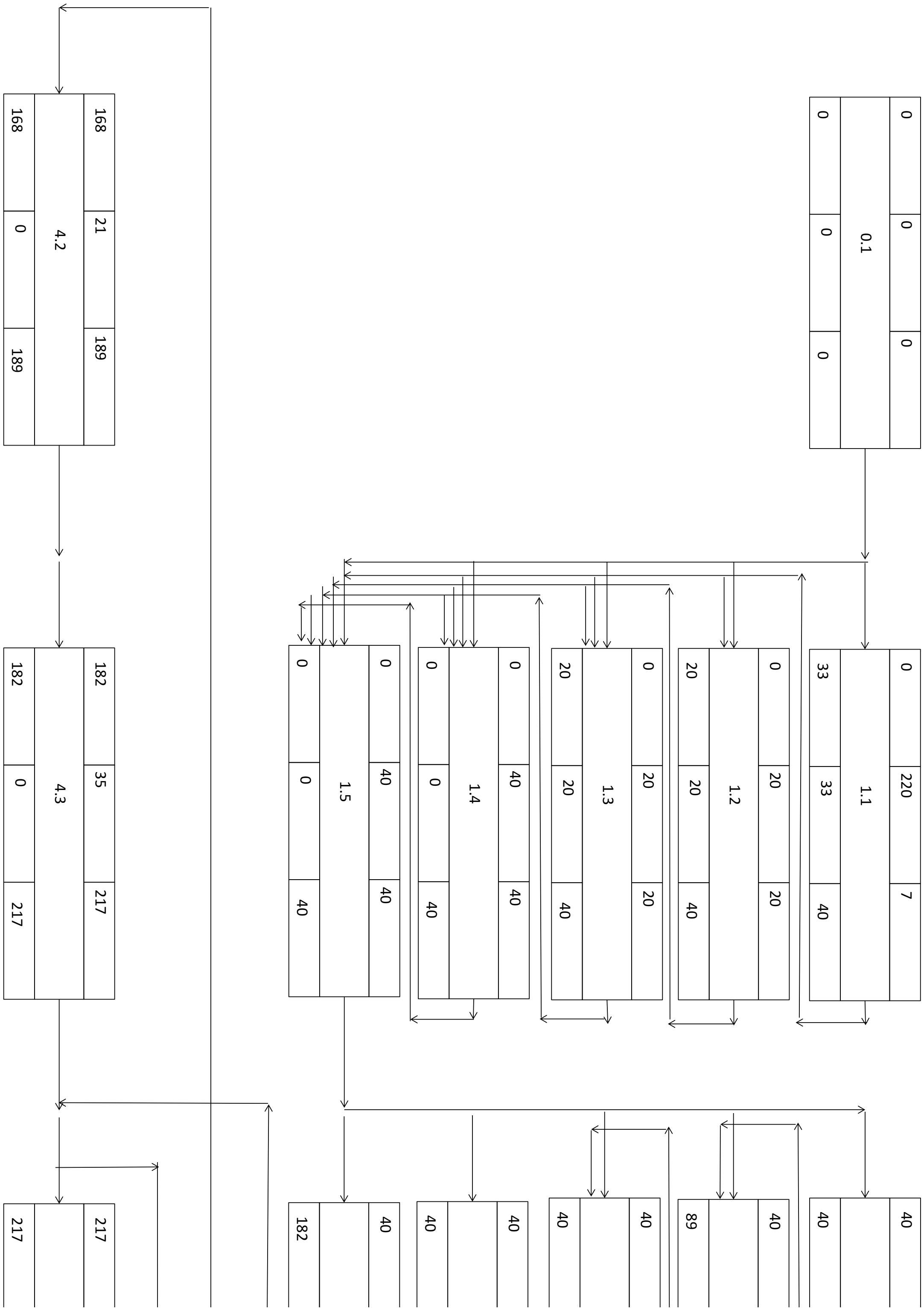
AV Data is obtained by WebRTC API, which is implemented separately from the haptic data-obtaining code.

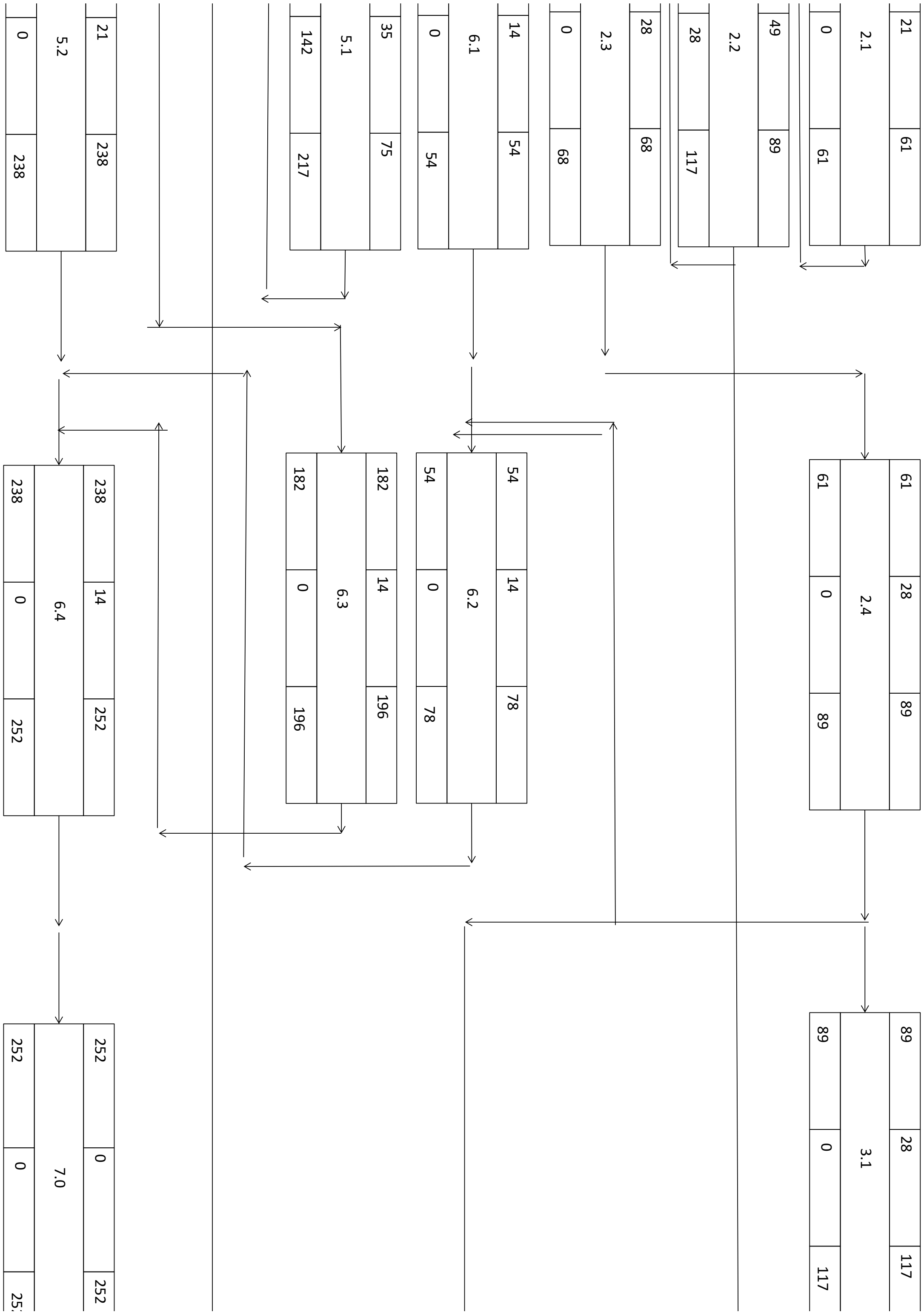
Sending HAV Data from the simulated Master to simulated Slave

The haptic data stored in `hapticMsgFromMaster` is sent through a channel through a custom-defined function `IsendData()`.

AV data is handled by WebRTC API.

12.6 Detailed CPM





2.1
21
61
0
61
2.2
49
89
28
117
2.3
28
68
0
68
6.1
14
54
0
54
5.1
35
75
142
217
5.2
21
238
0
238
2.4
61
28
89
61
0
89
6.2
54
14
78
54
0
78
6.3
182
14
196
182
0
196
6.4
238
14
252
238
0
252
3.1
89
28
117
89
0
117
7.0
252
0
252
252
0
25

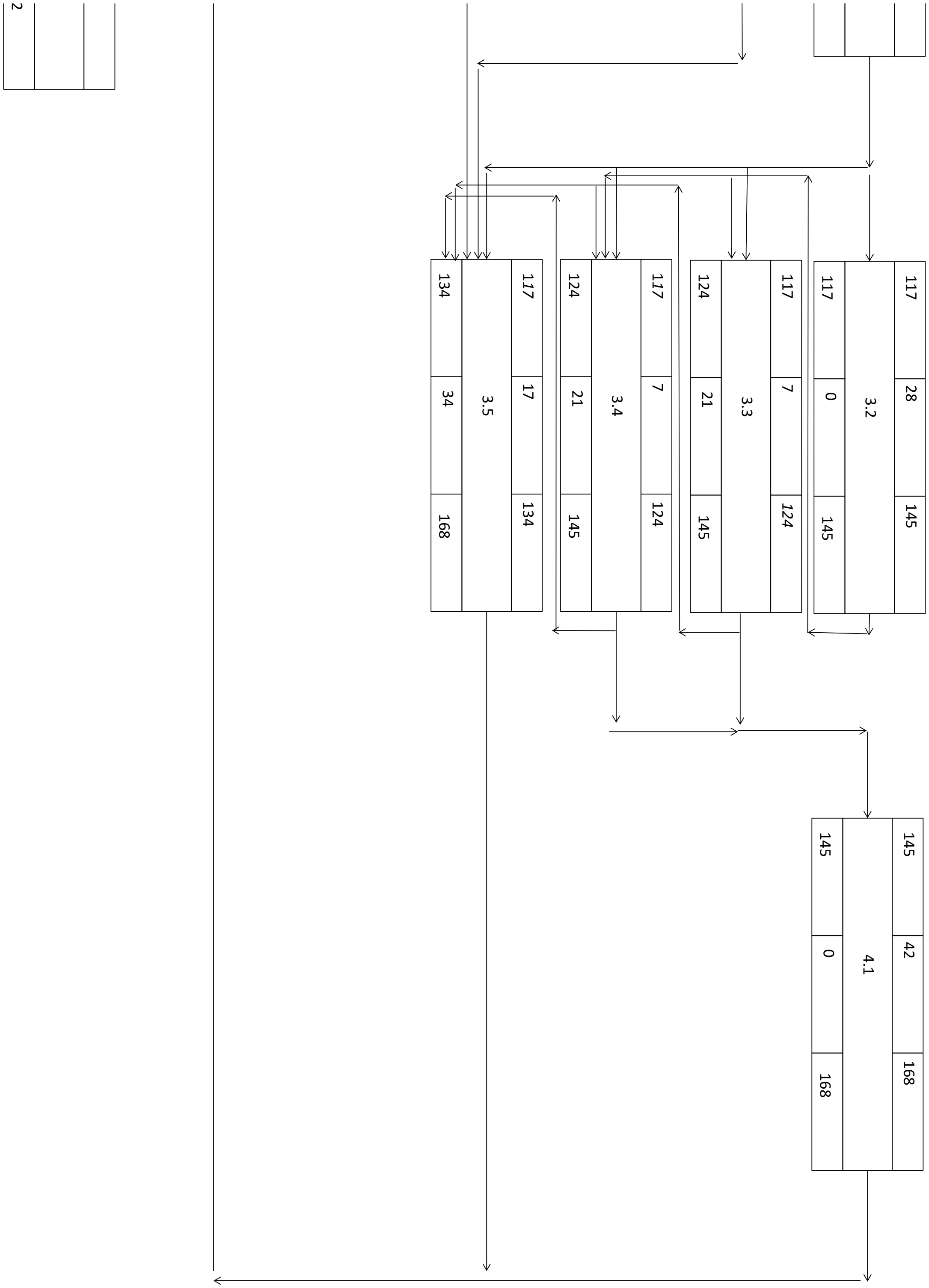


Figure 25: Detailed Project Critical Path Method. The critical path here, in order, is 0.1, 1.5, 2.3, 2.4, 3.1, 3.4, 4.1, 4.2, 4.3, 5.2, 6.4, and 7.0. Note that the numbers are in days.