NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

GRADUATE PROGRAM

MSc THESIS

Adaptive unmanned vehicle autopiloting using WebRTC vid-eo analysis

Apostolos I. Modas

Supervisor: **Athanasia Alonistioti,** Assistant Professor

ATHENS

JULY 2017

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Προσαρμοστικός αυτόματος πιλότος μη επανδρωμένου ο-χήματος με χρήση WebRTC ανάλυσης βίντεο

Απόστολος Η. Μόδας

Επιβλέπουσα: **Αθανασία Αλωνιστιώτη,** Επίκουρη Καθηγήτρια

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2017

MSc THESIS

Adaptive unmanned vehicle autopiloting using WebRTC video analysis

Apostolos I. Modas
A.M.: M1463

July 2017

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Προσαρμοστικός αυτόματος πιλότος μη επανδρωμένου οχήματος με χρήση WebRTC ανάλυσης βίντεο

Απόστολος Η. Μόδας
Α.Μ.: Μ1463

Ιούλιος 2017

ABSTRACT

We exploit the new features provided by WebRTC in terms of interoperability and state-of-the-art real-time communications, in order to develop a system for piloting unmanned vehicles using video analysis. Specifically, we define a topology where a ROS-based vehicle transmits its video using WebRTC to an intermediate server, who in turn relays it to a client. The server takes advantage of the OpenCV library and applies video analysis, with respect to a selected task (i.e. face detection) defined by the client. The corresponding commands are transmitted to the vehicle, resulting in an automatically driven unmanned vehicle. The client monitors the vehicle's movement and can dynamically change the selected use case; that is, either change slightly its operation (i.e. from human tracking to children tracking) or enable an entirely new core philosophy (i.e. to fire detection) by sending the appropriate requests to the server. Upon reception of these requests, the server utilizes the corresponding OpenCV functionalities to serve the new task, and sends the new piloting commands to the vehicle, forcing the system to adopt a new autopiloting mode. This communication between the vehicle, the server and the client is established using SIP/SDP and orchestrated via a WebSocket server that serves as a Signaling Server, the media are transferred through SRTP/UDP, and the commands are carried via the WebRTC Data Channel over SCTP. We explain and describe how to combine all of these heterogeneous components (WebRTC – OpenCV – ROS), in order to compose a web-based infrastructure for autopiloting ROS-based vehicles upon a specific use case. Finally, the results prove our concept, meaning a horizontal infrastructure that (a) consists of a modular architecture, (b) provides the necessary components for machine-to-machine communication, (c) uses state-of-the-art technologies, (d) allows a developer to implement her own logic vertically, and (e) provides IoT with a solution that can be easily exploited in numerous ways.

SUBJECT AREA: Real-Time Communications, Computer Vision, Internet of Things

KEYWORDS: WebRTC, OpenCV, web infrastructure, autopiloting, unmanned vehicles, ROS, IoT, technical instrumentation

ΠΕΡΙΛΗΨΗ

Εκμεταλλευόμαστε τις νέες δυνατότητες που παρέχονται από το WebRTC, υπό την έννοια της διαλειτουργικότητας και των τελευταίας γενιάς επικοινωνιών σε πραγματικό χρόνο, προκειμένου να αναπτύξουμε ένα σύστημα για το πιλοτάρισμα μη επανδρωμένων οχημάτων χρησιμοποιώντας ανάλυση βίντεο. Συγκεκριμένα, ορίζουμε μια τοπολογία όπου ένα ROS όχημα μεταδίδει βίντεο μέσω WebRTC προς έναν ενδιάμεσο εξυπηρετητή, ο οποίος με τη σειρά του το μεταβιβάζει σε έναν πελάτη. Ο εξυπηρετητής εκμεταλλεύεται τη βιβλιοθήκη OpenCV και εφαρμόζει ανάλυση βίντεο, με τέτοιο τρόπο ώστε να εξυπηρετήσει ένα επιλεγμένο από τον πελάτη σενάριο. Οι αντίστοιχες εντολές μεταδίδονται στο όχημα, με αποτέλεσμα να έχουμε ένα αυτόματα οδηγούμενο όχημα. Ο πελάτης παρακολουθεί την πορεία του οχήματος και μπορεί να αλλάξει δυναμικά το επιλεγμένο σενάριο – αυτό σημαίνει είτε να αλλάξει ελαφρώς τη λειτουργία του (π.χ. από παρακολούθηση ανθρώπων σε παρακολούθηση παιδιών) είτε να ενεργοποιήσει μια εντελώς διαφορετική φιλοσοφία λειτουργίας – στέλνοντας τα κατάλληλα αιτήματα στον εξυπηρετητή. Μόλις ο εξυπηρετητής λάβει αυτά τα αιτήματα, χρησιμοποιεί τις αντίστοιχες λειτουργίες το OpenCV για να εξυπηρετήσει το νέο σενάριο, και στέλνει τις νέες εντολές οδήγησης στο όχημα, αναγκάζοντας το σύστημα να υιοθετήσει μια νέα λειτουργία αυτόματου πιλότου. Η επικοινωνία μεταξύ του οχήματος, του εξυπηρετητή και του πελάτη εδραιώνεται μέσω των SIP/SDP και ενορχηστρώνεται μέσω ενός WebSocket εξυπηρετητή που επιτελεί το ρόλο του Signaling Server, ενώ οι εντολές μεταφέρονται μέσω του WebRTC Data Channel πάνω από το SCTP. Περιγράφουμε και αναλύουμε το πώς όλα αυτά τα ετερογενή συστατικά (WebRTC – OpenCV – ROS) συνδυάζονται για τη δημιουργία μιας δικτυακής υποδομής, για το αυτόματο πιλοτάρισμα ROS οχημάτων σύμφωνα με ένα συγκεκριμένο σενάριο χρήσης. Τέλος, τα αποτελέσματα αποδεικνύουν την ιδέα μας, δηλαδή μια οριζόντια υποδομή που (α) αποτελείται από μια ευέλικτη/αρθρωτή αρχιτεκτονική, (β) παρέχει τα απαραίτητα στοιχεία για την μηχανή-σε-μηχανή επικοινωνία, (γ) χρησιμοποιεί τελευταίας γενιάς τεχνολογίες, (δ) επιτρέπει σε έναν προγραμματιστή να εφαρμόσει τη δική του λογική κατακόρυφα σε βάθος και (ε) παρέχει στον τομέα του IoT μια λύση που μπορεί εύκολα να αξιοποιηθεί με πολλούς τρόπους.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Επικοινωνίες Πραγματικού Χρόνου, Μηχανική Όραση, Διαδίκτυο των Πραγμάτων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: WebRTC, OpenCV, web υποδομή, αυτόματος πιλότος, μη επανδρωμένα οχήματα, ROS, IoT, τεχνική ενορχήστρωση

CONTENTS

LIST OF IMAGES

LIST OF TABLES

PREFACE

The current Master Thesis was pursued in Athens from September 2016 until July 2017. It is a mandatory requirement for the graduation from the Master Program at the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens.

1. INTRODUCTION

Audio and video applications have become a basic part of everyday life, since they cover a wide range of services such as communication, entertainment, news/information, medicine, sports etc. Depending on the service, there are different ways in which someone can transmit and manipulate multimedia data, each of which requires a different approach. In this chapter, we will make a small introduction to general features related to real-time multimedia over the Internet, Google's new communication "standard" named WebRTC [1], and to the rapidly growing field of Internet of Things – IoT [2], combined with the well-known computer vision library, OpenCV [3]. These introductory sections are also the pre requirements for an in-depth understanding of our implementation.

1.1 Real-time Audio and Video Communication Over the Internet

Real-time audio and video communications over the Internet have high and strong requirements in terms of bandwidth limitations, delays, and receiving quality, but several difficulties need to overcome to meet these requirements. Although Internet provides the ability to transfer data over long distances at high speed, its bandwidth is always limited, and it presents many random fluctuations that have major impact on the delay and the quality of the data being transferred. Thus, the identification of the appropriate protocols and techniques for dealing with bandwidth limitations, for reducing both the delay and packet loss, and for preserving the quality at the receiving end is necessary.

1.1.1 Protocols

For real-time media transmission, there are a set of specific protocols that are used, which provide the appropriate mechanisms to meet or to handle the aforementioned requirements. Regarding some of the most used protocol, on the transport layer, the User Datagram Protocol – UDP [4] is adopted, whereas on the application layer, the Real-time Transport Protocol – RTP [5] is the standard protocol. In cases where security is mandatory, the Secure Real-time Transport Protocol – SRTP [6] is preferred. Both RTP and SRTP are used in conjunction with their sister protocols, Real-time Transport Control Protocol – RTCP [5] and Secure Real-time Transport Protocol – SRTCP [6], which provide control information for an RTP or an SRTP session respectively. Finally, on top of RTP/SRTP, the Session Initiation Protocol – SIP [7] together with the Session Description Protocol – SDP [8] are used, which provide the appropriate mechanisms for signaling and media description.

1.1.2 Barriers

The most common problem when trying to establish a Peer-to-Peer (P2P) connection is the presence of Network Address Translators – NAT [9], which operate as a network firewall, protecting the IP of the machine/device, preventing other machines to detect it. In order to surpass this barrier, a variety of NAT Traversal [10] and Hole Punching [11] techniques and standards have been developed, but they come with a lot of restrictions and complexity in use. Another very usual problem is the lack of interoperability; many applications are hardware and device dependent, which means that an additional application (i.e. a software) or a plugin must be present in order to setup a communication between peers.

1.2 WebRTC

WebRTC is new technology that simplifies and solves all the aforementioned requirements and barriers, enabling a fast, secure and simple real-time P2P communication infrastructure. Its main advantage is that it provides the functionalities for P2P communication on the Web, without the need of a software or a plugin. Furthermore, it is device independent since it can work on any device that has access to a browser, implements state-of-the-art audio and video codecs, and finally, uses the Interactive Connectivity Establishment – ICE [12] technique for NAT Traversal and Hole Punching, in order to find ways for two peers to talk each other as directly as possible. WebRTC is further explained in Chapter 2.

1.3 Internet of Things and Unmanned Vehicles

Concerning the field of IoT, it is worth mentioning the importance of multimedia as more and more applications are using audio and image/video sensors for measurements. For instance, unmanned vehicles is an IoT category that makes use of a large set of sensors, with the camera being the primary one, since it can be exploited by many sectors such as health, sports, agri-food, security etc. However, could someone use WebRTC for streaming the video as well as the sensor data in real-time from the vehicle to our application?

1.3.1 Image and Video Processing – OpenCV

Finally, our application must analyze and process the video and image data provided by the vehicles, in order to extract the corresponding information. These procedures can be accomplished with the use of OpenCV, a library that has many applications in computer vision and robotics. However, assuming that an unmanned vehicle is also a robot, could someone use the video provided by WebRTC as the main input to OpenCV?

1.4 Conclusions

Considering all of the above – meaning the real-time protocols, the codecs, the requirements and the barriers, the WebRTC, the unmanned vehicles, and the OpenCV – we will combine them properly, in order to develop an infrastructure that fulfills all of these requirements, and provides IoT with a Machine-to-Machine (M2M) communication solution, that exploits state-of-the-art technologies along a wide range of applications. The rest of the dissertation is organized as follows; in Chapter 2, an overview of WebRTC and its protocols, as well as a description of the offered API's is provided. The OpenCV library, including its features and its web exposure is analyzed in Chapter 3, whereas the Robot Operating System and the turtle simulator through the web are described in Chapter 4. In Chapter 5, we exploit the knowledge of the aforementioned Chapters to provide an in-depth analysis of our implemented infrastructure. Finally, Chapter 6 shows the results and the metrics of our system, while the future work is discussed in Chapter 7.

2. REAL-TIME COMMUNICATIONS THROUGH THE WEB – WEBRTC

WebRTC is a collection of communication protocols and Application Programming Interfaces – APIs that enable Real-Time Communication – RTC over P2P connections, allowing web browsers not only to request resources from backend servers, but also real-time information from browsers of other users. This enables applications such as video conferencing, file transfer, chat, or desktop sharing without the need of either internal or external plugins. Google released it on May 2011; it is supported by Mozilla and Opera amongst others, and is being standardized by the World Wide Web Consortium – W3C and the Internet Engineering Task Force – IETF.

2.1 State-of-the-art

WebRTC is a state-of-the-art technology since it came with a revolutionary approach on Web-based RTC, setting the ground for becoming the main standard for real-time communications.

2.1.1 Browser-to-Browser

The strongest aspect of WebRTC is the enrichment of browsers with RTC functionalities, without the need of internal or external plugins. Before that, in order for two peers to communicate over the browser, they had to use either a specific software or a specific plugin at both ends, which was Operating System – OS, browser, and device dependent, and responsible for providing the appropriate RTC methods and infrastructure.

2.1.2 Interoperability – Adapter.js

What is more, WebRTC is not only OS and device independent – requires only a browser – but also browser independent. This is feasible due to adapter.js [13], a shim to insulate applications from specification changes and prefix differences along different browsers (i.e. Chrome-to-Firefox, Chrome-to-Opera etc.).

2.1.3 Audio and Video Codecs

In addition, WebRTC uses some state-of-the-art audio and video codecs, which provide high compression, while keeping a balanced tradeoff in terms of information loss and network limitations. By now, it supports G.711, G.722, iSAC, iLBC, and Opus audio codecs [14, 15, 16, 17, and 18 respectively], while the supported video codecs are H.264 AVC, VP8, and VP9 [19, 20, and 21 respectively]. It is worth noting that the combination of Opus and VP9 is the WebM video file format [22], while the VP9 video codec is the one comparable to well-known HEVC [23], and the ancestor of AV1 [24].

2.1.4 Interactive Connectivity Establishment

Finally, WebRTC provides the appropriate frameworks and mechanisms for Hole Punching and NAT Traversal, by using the ICE technique. ICE is used in computer networking to find ways for two computers to talk to each other as directly as possible in P2P networking, and provides a framework with which a communicating peer may discover and communicate its public IP address so that other peers can reach it. Session

Traversal Utilities for NAT – STUN [25] is a standardized protocol for such address discovery (including NAT classification), while Traversal Using Relays around NAT – TURN [26] places a third-party server to relay messages between two clients when direct media traffic between peers is not allowed by a firewall.

2.2 Protocol Stack

Like every real-time P2P communication, WebRTC requires a set of communication protocols for exchanging media and data, as well as for orchestrating the session. Whereas the orchestration is usually managed by protocols and mechanisms that are integrated into the application (i.e. the SIP/SDP procedures), WebRTC uses an external entity for the signaling between peers called Signaling Server, which is not part of the project's implementation, but its presence is mandatory. Therefore, a WebRTC session consists of two main entities; the Signaling Server, and the set of RTC functionalities and APIs that are provided by the browser. The protocol stack used by WebRTC is depicted in Figure 1, where the left sub-stack refers to the Signaling Server, while the right one to the provided APIs.

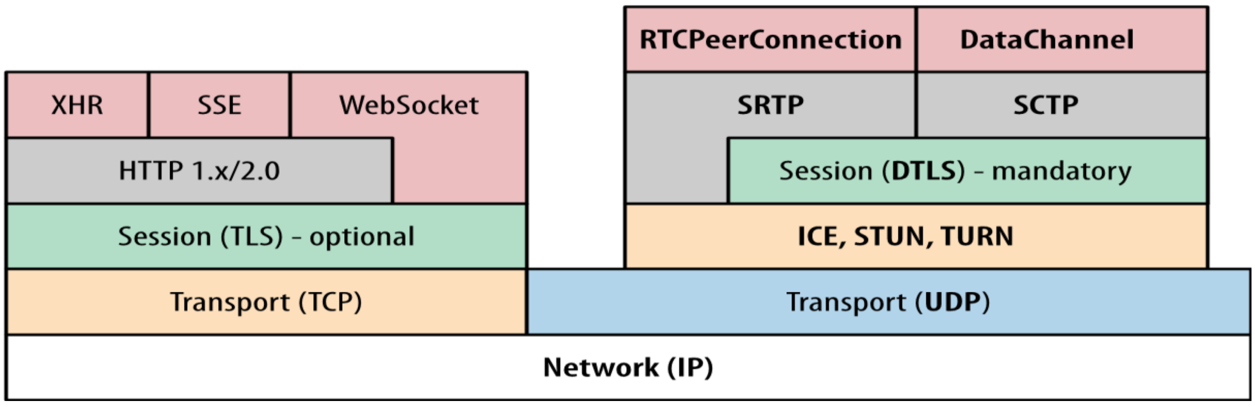


Figure 1: The Protocol Stack of WebRTC

In the following sub-sections, a brief description of the protocols used by WebRTC is outlined.

2.2.1 UDP

Unlike all other browser communication, WebRTC transports its data over UDP. The requirement for timeliness over reliability is the primary reason why the UDP protocol is a preferred transport for delivery of real-time data; it offers no promises on reliability or order of the data, and delivers each packet to the application the moment it arrives. In effect, it is a thin wrapper around the best-effort delivery model offered by the IP layer of the network stacks [27].

However, an application also needs mechanisms to traverse the many layers of NATs and firewalls, negotiate the parameters for each stream, provide encryption of user data, implement congestion and flow control etc. Thus, while UDP is the foundation for real-time communication in the browser, in order to meet all the requirements of WebRTC, the browser also needs a large supporting cast of application protocols and methods.

2.2.2 TCP

In the case of the transported messages of the Signaling Server, the Transmission Control Protocol – TCP is [28] preferred. The requirement for reliability over timeless is the primary reason why the TCP protocol is a preferred transport for delivery of these messages; it offers reliability, in-order data, and delivers each packet to the application after a complete error checking.

2.2.3 ICE, STUN, and TURN

WebRTC, as described in 2.1.4, utilizes the ICE technique, which makes use of the STUN protocol and its extension, TURN. ICE provides a general framework for describing the available candidates [29], and exploits the STUN and TURN protocols to reach the other peer behind the firewalls (Hole Punching and NAT Traversal).

STUN is a standardized set of methods, including a network protocol, for traversal of NAT gateways in real-time communication applications. It is used by other protocols (i.e. ICE, SIP), providing a tool for hosts to discover the presence of a NAT, and to discover the mapped, usually public, IP address and port number that the NAT has allocated for the application’s TCP and UDP flows to remote hosts. The protocol requires assistance from a third-party network server (STUN Server) located on the opposing, public side of the NAT, usually the public Internet.

TURN is a protocol that assists in traversal of NATs or firewalls for multimedia applications. It is most useful for clients on networks masqueraded by symmetric NAT devices. TURN does not aid in running servers on well-known ports in the private network through a NAT, while it supports the connection of a user behind NAT to only single peer. The topologies for STUN and TURN usages are depicted in Figure 2 and Figure 3 respectively.

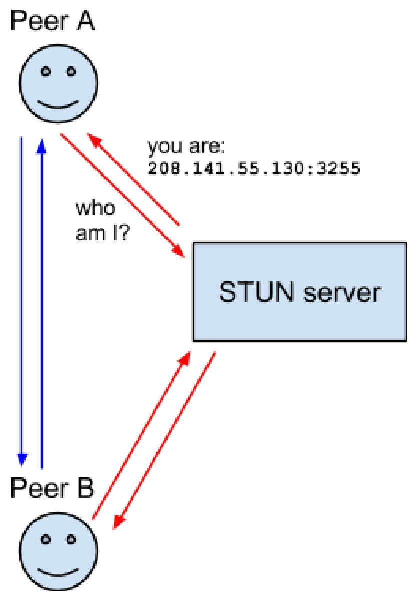


Figure 2: Use of STUN Server for NAT Traversal

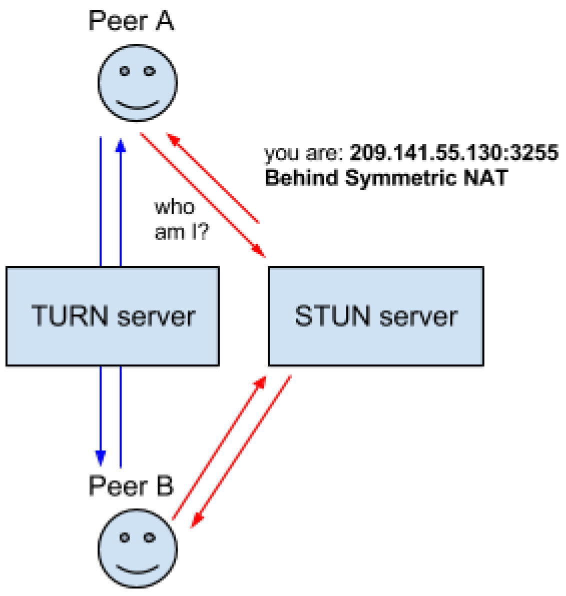


Figure 3: Use of STUN/TURN Servers for NAT Traversal

2.2.4 TLS and DTLS

WebRTC encrypts information for the Data Channels, by using the Datagram Transport Layer Security – DTLS [30] protocol, which is built into all browsers that support WebRTC, and is one protocol consistency used in web browsers, email, and VoIP platforms to encrypt information; the built-in nature also means that no prior setup is required before use. As with other encryption protocols, it is designed to prevent eavesdropping and information tampering. DTLS itself is modelled upon the stream-oriented Transport Layer Security – TLS [31] protocol, which offers full encryption with asymmetric cryptography methods, data authentication, and message authentication. As DTLS is a derivative of Secure Socket Layer – SSL [32] protocol, all data is known to be as secure as using any standard SSL-based connection. In fact, WebRTC data can be secured via any standard SSL-based connection on the web, offering end-to-end encryption between peers with almost any server arrangement.

2.2.5 RTP/RTCP and SRTP/SRTCP

The basic application protocol for real-time communications is the RTP along with RTCP. However, it does not have any built-in security mechanisms, and thus provides no protection or confidentiality of transmitted data, and so the use of external mechanism that provide encryption is mandatory.

The use of unencrypted RTP is explicitly forbidden by the WebRTC specification. For this reason, it utilizes SRTP/SRTCP for the encryption of media streams, rather than DTLS, because SRTP is a lighter-weight option. The specification requires that any compliant WebRTC implementation must support RTP/SAVPF [33], which is built on top of RTP/SAVP [34]. However, the actual SRTP key exchange is initially performed end-to-end with DTLS-SRTP [34], allowing for the detection of any man-in-the-middle (MiTM) attack.

2.2.6 SCTP

WebRTC Data Channels deliver data using the SCTP over DTLS protocols [35], where SCTP is the Stream Control Transmission Protocol [36]. STCP as a protocol can be seen as a hybrid of UDP and TCP, as shown in Table 1; it is connection and message oriented, offers optional reliability and ordering, and provides flow and congestion control. The SCTP is usually implemented on an application level.

Table 1: The SCTP protocol as a UDP-TCP hybrid

	TCP	UDP	SCTP
Reliability	Reliable	Unreliable	Configurable
Delivery	Ordered	Unordered	Configurable
Transmission	Byte-oriented	Message-oriented	Message-oriented
Flow control	Yes	No	Yes
Congestion control	Yes	No	Yes

2.2.7 SIP/SDP

Finally, WebRTC uses the SIP and SDP protocols for the orchestration of the communication between the peers. SIP is a communications protocol for signaling and controlling multimedia communication sessions. It defines the messages that are sent between endpoints, which govern establishment, termination and other essential elements of a call. It can be used for creating, modifying, and terminating sessions consisting of one or more media streams. SIP works in conjunction with several other protocols that specify and carry the media session. Media type and parameter negotiation and setup is performed with the SDP, which is carried as a payload in a SIP message and provides a format for describing streaming media parameters. These protocols are based on an Offer/Answer model, and are implemented in the WebRTC application, but they are transferred via the Signaling Server. The use of SIP in conjunction with SDP is shown in Figure 4.

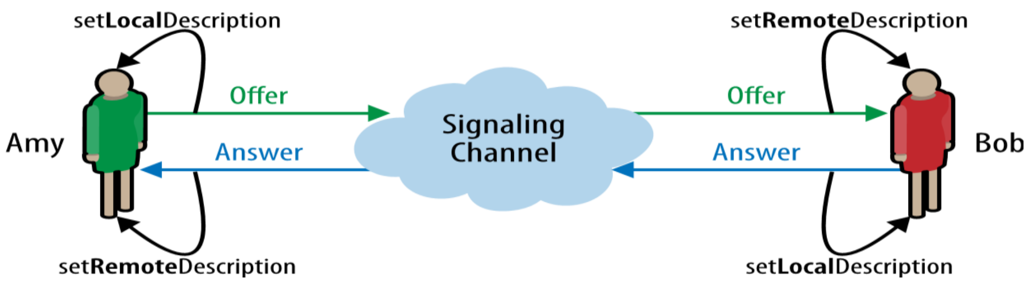


Figure 4: SIP/SDP Offer/Answer between peers

2.3 WebRTC APIs

WebRTC is a collection of standards, protocols, and JavaScript APIs, the combination of which enables peer-to-peer audio, video, and data sharing between browsers (peers). Instead of relying on third-party plug-ins or proprietary software, WebRTC turns real-time communication into a standard feature that any web application can leverage via a simple JavaScript API.

Delivering rich, high-quality, RTC applications such as audio and video teleconferencing and peer-to-peer data exchange requires a lot of new functionality in the browser: audio and video processing capabilities, new application APIs, and support for half a dozen new network protocols. Thankfully, the browser abstracts most of this complexity behind three primary APIs, which expose the native project's functionalities to the web and simplify the setup of the WebRTC session:

1. MediaStream API [37]
2. RTCDataChannel API [38]
3. RTCPeerConnection API [38]

2.3.1 MediaStream API

The MediaStream API represents synchronized streams of media (i.e., a stream taken from camera and microphone input has synchronized video and audio tracks), and is responsible for requesting and processing audio and video streams from the platform. Each MediaStream object (Figure 5) has an input, which might be a media stream from a device or a file, an output that can be manipulated in many ways (i.e. display video, play sound, send to other peer, record to file etc.), and consists of one or more tracks (i.e. an audio and a video track). Finally, the main method for requesting the streams is the getUserMedia, which takes three parameters:

1. Some constraints, which are applied to the media (i.e. fps, video dimensions, audio packet time etc.).
2. A success callback, which is passed a MediaStream object.
3. A failure callback, which is passed an error object.

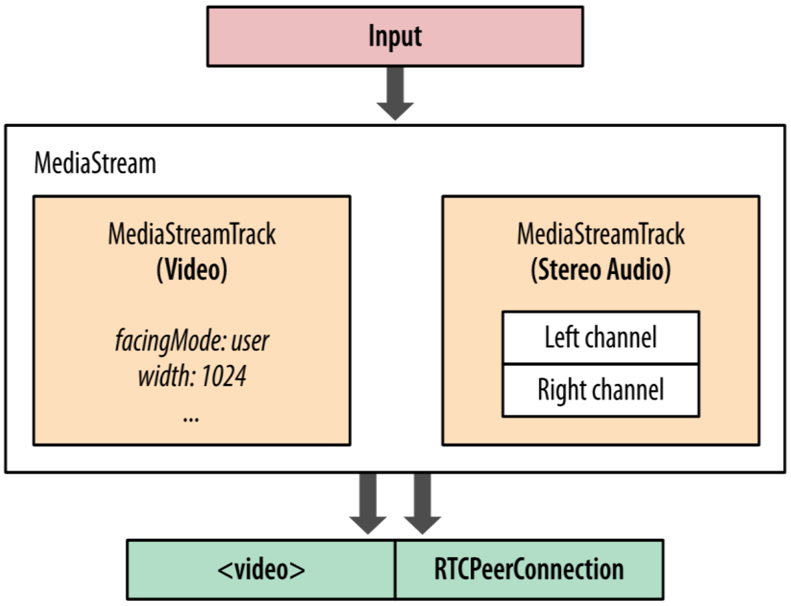


Figure 5: The MediaStream Object

2.3.2 RTCDataChannel API

Apart from audio and video, WebRTC supports real-time communication for other types of data. The RTCDataChannel API enables P2P exchange of arbitrary data over SCTP, with low latency and high throughput. What is more, it can be used as the main signaling channel, for the case of a signaling server. Finally, it has some properties that are quite powerful and flexible:

1. Maximum number of Data Channels: 65534 (theoretically)
2. Maximum capacity: Maximum Transfer Unit (MTU)
3. Delivery types: in-order, out-of-order, (un)reliable
4. Channel priorities
5. Multiplexing of independent channels
6. Message oriented API for fragmentations and assemblies
7. Flow and congestion control mechanisms
8. Confidentiality and integrity of transferred data

2.3.3 RTCPeerConnection API

The last and most important API of WebRTC is the RTCPeerConnection, since it is responsible for managing the full life cycle of each P2P connection. The PeerConnection object (Figure 6):

1. Manages the Ice workflow (Figure 7) for NAT Traversal, and trickles and registers the local and remote ICE candidates respectively
2. Sends automatic STUN keepalives to ensure the ICE procedure
3. Keeps track of both the local and remote streams
4. Triggers an automatic stream renegotiation using SDP Offer/Answer
5. Provides all the methods for Offer/Answer, connection's current state, ICE candidates etc.

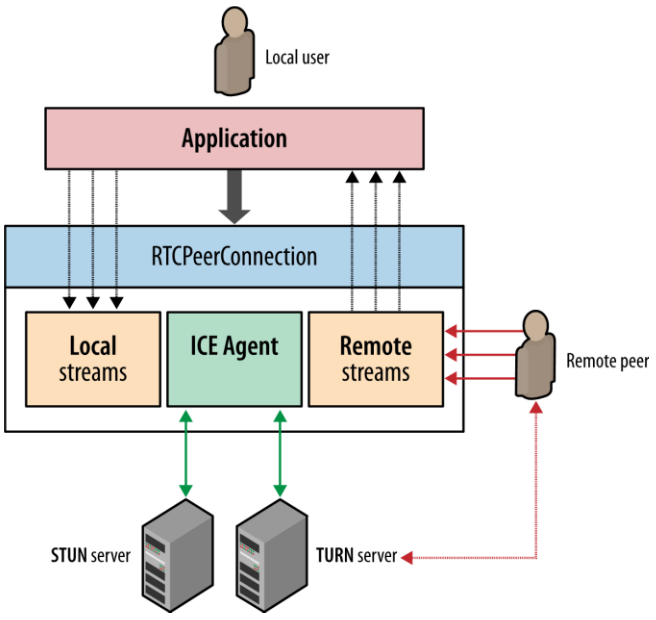


Figure 6: The PeerConnection Object

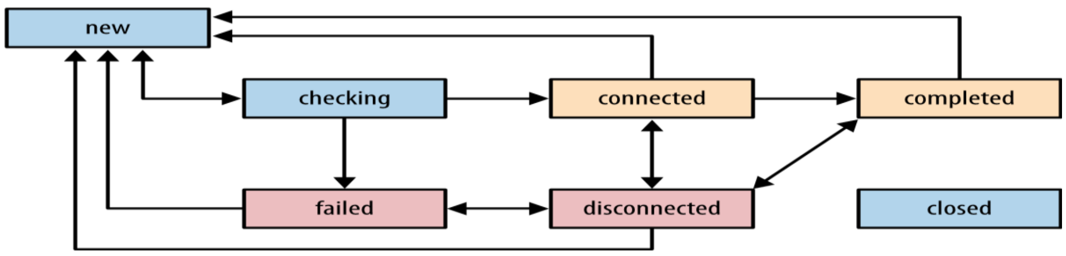


Figure 7: ICE agent connectivity states and transitions (ICE workflow)

2.4 Signaling Server

Signaling plays an important role in WebRTC but is not standardized, because it does not need to be for enabling interoperability between browsers; it is effectively a matter between the web browser and the web server. It has four main roles [39]:

1. Negotiation of media capabilities and settings.

2. Identification and authentication of participants in a session.
3. Controlling the media session, indicating progress, changing and terminating the session.
4. Glare resolution, when both sides of a session try to establish or change a session at the same time.

2.4.1 Topology

The most common scenario for establishing a WebRTC session is likely to be where both browsers are running the same WebRTC web application. This produces the WebRTC “Triangle”, shown in Figure 8 [40]. This arrangement is called triangle due to the shape of the signaling and media or data flows between the three elements.

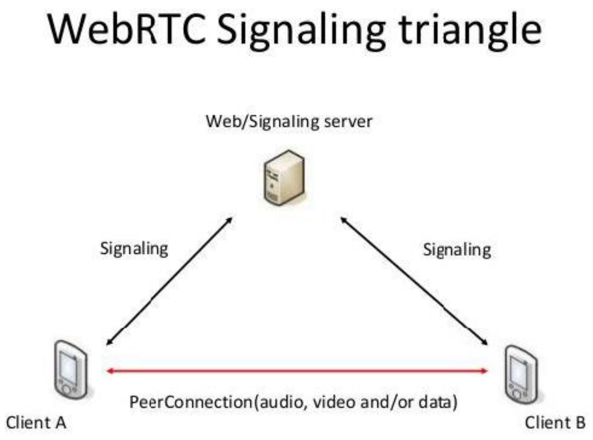

WebRTC Signaling triangle
Web/Signaling server
Signaling
Signaling
PeerConnection(audio, video and/or data)
Client A
Client B

Figure 8: The triangle topology

Figure 9 [41] shows the WebRTC Trapezoid [42] based on the SIP Trapezoid. The two web servers are shown communicating using a standard signaling protocol (i.e. SIP) or Jingle [43]. Note that in these more complicated cases, the media may not flow directly between the two browsers, but may go through media relays and other elements.

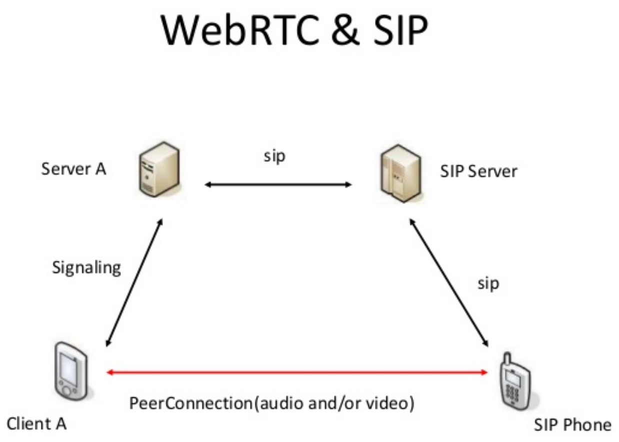


Figure 9: The trapezoid topology

2.4.2 Signaling Transport

WebRTC requires a bidirectional signaling channel between the two browsers, and thus three transports are commonly used for WebRTC signaling:

1. HTTP [44]: sending information to the server is straightforward, using an XML HTTP Request – XHR [45]. In the opposite direction, receiving information asynchronously from the server is trickier, and a number of techniques have been developed over the years known as Asynchronous JavaScript And XML – AJAX. The use of HTTP for signaling is often referred to as Representational State Transfer – REST [46] or RESTful signaling.
2. Data Channel [47]: The Data Channel, once established between two browsers, provides a direct, low latency connection, which makes it suitable for signaling transport. Since the initial establishment of a Data Channel requires a separate signaling mechanism, the Data Channel alone cannot be used for all WebRTC signaling. However, it can be used to handle all signaling after it is set up, including all the signaling for the audio and video media over the Peer Connection.
3. WebSockets [48]: WebSocket [49] transport, allows a browser to open a bidirectional connection to a server; the connection begins as an HTTP request, but then upgrades to a WebSocket. In order for a WebSocket server to be reachable, it must have a public IP address and be running an HTTP server. Note that this means it is not possible to open a WebSocket directly with another browser, since browsers implement HTTP user agent functionality and not HTTP server functionality. So, the WebSocket server is still needed to relay between two web clients using WebSockets.

2.4.3 Signaling Protocol

The choice of signaling protocol for WebRTC is an important one, and not necessarily tied to the choice of signaling transport. A developer may choose to create her own proprietary signaling protocol, use a standard signaling protocol such as SIP or Jingle, or use a library that abstracts away the details of the signaling protocol [50].

In our implementation, we chose to use the WebSocket Proxy approach [Burnet 86] with a triangle topology. A WebSocket proxy used for WebRTC signaling, would use a server which has a public IP address and is reachable by both browsers establishing the Peer Connection. Each browser opens an independent WebSocket connection with the same server, and the server bridges the connections, proxying information from one to another. Since JavaScript does not support DNS lookups, the WebSocket server will need to be provided by the web server as an IP address and port number. Finally, when information is received from a WebSocket (i.e. from a peer’s WebSocket), it is relayed to one particular connection that has the other browser; some sort of session/connection ID is used for this.

3. COMPUTER VISION – OPENCV

OpenCV is an open source computer vision and machine learning software library (written in C++), which was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. It has more than 2500 optimized algorithms, which include a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. Finally, it provides C++, C, Python, Java and MATLAB interfaces, and is supported – among others – by Google, Microsoft, Intel, and IBM. The OpenCV library is used in our implementation for the real-time processing of the vehicle's video.

3.1 Face Detection

OpenCV's functionalities are divided into modules, including Image Processing, Video Analysis, Object Detection, Object Tracking etc. We focus on an aspect of the Object Detection module, and specifically on the case of Face Detection using Haar feature-based cascade classifiers [52], in order to adopt it to the implementation .

Object Detection using Haar feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones [53]. It is a machine learning based approach where a cascade function is trained from a set of positive and negative images, and then it is used to detect objects in other images.

Initially, the algorithm needs a set of positive (images of faces) and negative images (images without faces) to train the classifier. Then the Haar-feature filters are used (Figure 10) to extract features from it, which are just like convolutional kernels. Each feature is a single value obtained by subtracting sum of pixels under white rectangle from sum of pixels under black rectangle.

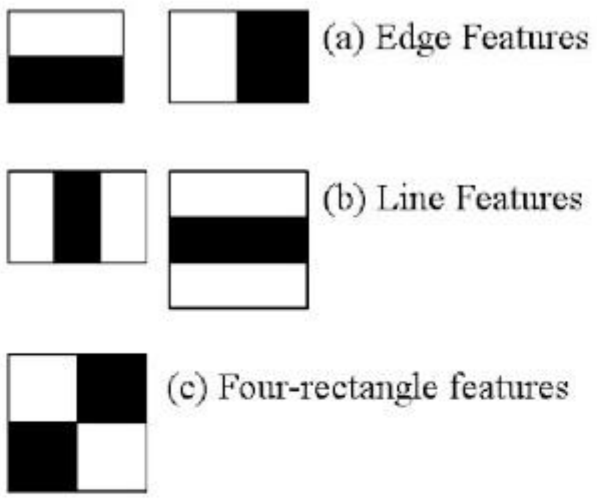


Figure 10: Haar-feature filters

All possible sizes and locations of each kernel are used to calculate the features, where a 24x24 window results over 160000 features. For each feature calculation, the sum of pixels under white and black rectangles is computed through the integral images. But among these features, most of them are irrelevant. For the best-features selection (out of ~160000), the Adaboost algorithm [54] is used. For each feature, Adaboost finds the best threshold that will classify the faces into positive and negative. Regarding the errors of misclassification, it selects the features with the minimum error rate, which means that they best classify the face and non-face images. The final classifier is a weighted sum of these “weak” classifiers, and the final setup had around 6000 features.

So, the procedure requires to divide the image into 24x24 windows, to apply 6000 features to each one, and to check if it is a face or not. Due to the high complexity of this procedure, they introduced the concept of cascade of classifiers, where instead of applying all 6000 features on a window, group the features into different stages of classifiers, and apply one-by-one; if a window fails the first stage, discard it, else, apply the second stage of features and continue the process. The window that passes all stages is a face region. For the record, Viola’s and Jones’ cascade of classifiers consisted of 38 stages (with 1, 10, 25, 25, and 50 features in first five stages), where ~10 features are evaluated per window.

OpenCV contains many pre-trained classifiers for face, eyes, smile, etc., stored in XML files. The results of the face and eyes XML classifiers on an image are shown in Figure 11 [52].

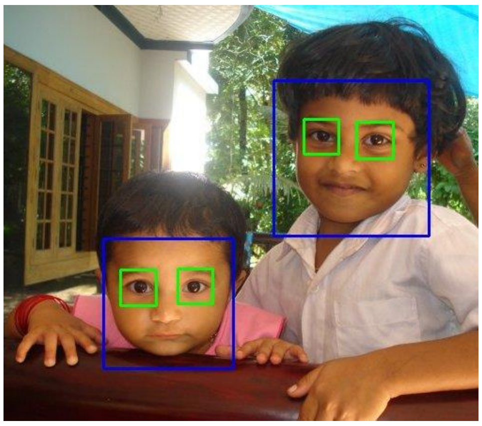


Figure 11: OpenCV results for face and eyes detection

3.2 Web OpenCV

Although OpenCV is the most common library for computer vision, it was not built for web applications. Despite the fact that it provides Java and Python interfaces – someone could build some sort of web app – there is no way to use it in HTML5 with JavaScript. At this point, it is sensible to make a small reference to Emscripten [55].

Emscripten is a source-to-source compiler, developed by Alon Zakai. It runs as a back end to the LLVM [56] compiler, and produces a highly optimized subset of JavaScript code known as asm.js [57]. This allows applications and libraries originally designed to run as standard executables (i.e. in C/C++) to be integrated into web applications (Figure 12).

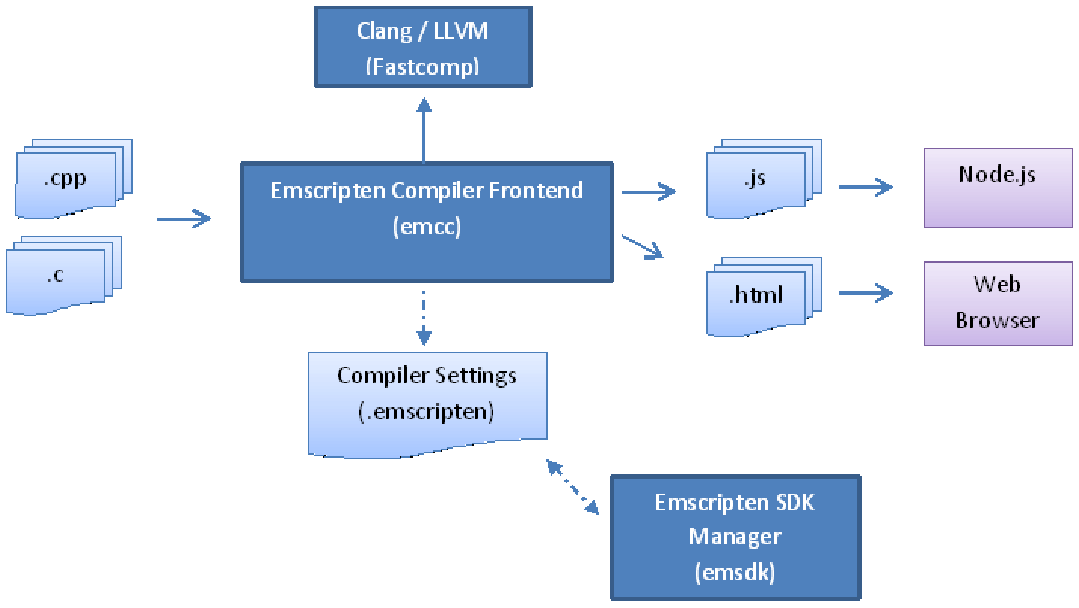

Clang / LLVM (Fastcomp)
.cpp
.c
Emscripten Compiler Frontend (emcc)
.js
.html
Node.js
Web Browser
Compiler Settings (.emscripten)
Emscripten SDK Manager (emsdk)

Figure 12: The Emscripten toolchain

Finally, asm.js can be compiled by browsers ahead of time, meaning that the compiled programs can run much faster than those traditionally written in JavaScript, and specifically close to native speed.

3.2.1 OpenCV.js

So, is it possible to use Emscripten to integrate into our web implementation the OpenCV library? The answer came from the University of Irvine, which, in cooperation with Intel, created the OpenCV.js [58], which takes advantage of Emscripten and asm.js to expose the native library to the web (Figure 13).

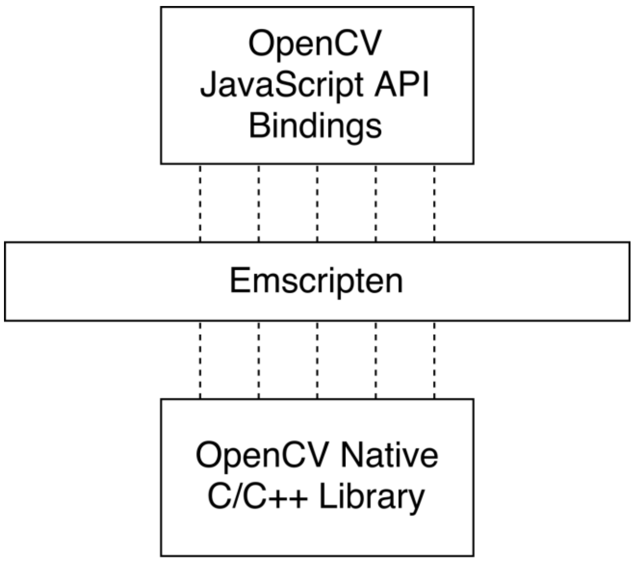


Figure 13: Use of Emscripten to expose the OpenCV native library to the Web

OpenCV.js extends the OpenCV language binding by providing a JavaScript interface. It allows emerging web applications with multimedia processing to benefit from the wide variety of vision functions available in OpenCV. It has several thrusts:

1. High performance: allow OpenCV.js to be used in demanding applications, e.g. video processing. The OpenCV.js is based on asm.js specification, and near native performance is obtained on most modern browsers. It also supports SIMD.js to take advantage of processor's vector (SIMD) processing capabilities.
2. Coverage: Currently, more than 50 classes and 800 functions from libraries including core, image processing, video processing, image codecs, machine learning are already supported. This subset is comparable to Python OpenCV.
3. API Correspondence: the JavaScript API is close to the original OpenCV API, thus making it easier to write/port applications.

3.2.2 HTML5 Canvas Element

One question remains: since we can access the OpenCV executables from the Web, can we also give as input a live video stream for processing? The answer is no, because HTML5 does not yet provide access to the video bitstream. However, there is a solution: HTML5 Canvas element.

Upon receiving a video stream, we use the Video element in order to manipulate it (display, record, relay etc). We can take advantage of the Canvas element, by taking a snapshot of the Video element, and converting it to a Canvas one. From the canvas, we can easily convert the data to an Image, and thus we can actually have a Frame of the video, which can easily be processed by OpenCV.js.

4. ROBOT OPERATING SYSTEM – ROS

The final component of our implementation is the Robot Operating System – ROS [59], which is used to simulate the unmanned vehicle. ROS is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions, that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

4.1 ROS Architecture

The ROS architecture has been designed and divided into three sections or levels of concepts [60]:

1. The Filesystem Level
2. The Computational Graph Level
3. The Community Level

In the first level, a group of concepts is used to explain how ROS is internally formed, the folder structure, and the minimal files that it needs to work. In the second level, the communication between processes and systems takes place, while in the third level, the tools and concepts to share knowledge, algorithms, and code from any developer is explained.

4.1.1 The Filesystem Level

Similar to an operating system, a ROS program is divided into folders, and these folders have some files that describe their functionalities [61]:

- Packages: packages form the atomic level of ROS. A package has the minimum structure and content to create a program within ROS. It may have ROS runtime processes (nodes), configuration files, and so on.
- Manifests: manifests provide information about a package, license information, dependencies, compiler flags, and so on. They are managed with a file called manifests.xml.
- Stacks: when someone gathers several packages with some functionality, she will obtain a stack. In ROS, there exists a lot of these stacks with different uses, for example, the navigation stack.
- Stack manifests: stack manifests (stack.xml) provide data about a stack, including its license information and its dependencies on other stacks.
- Message (msg) types: a message is the information that a process sends to other processes. ROS has a lot of standard type of messages.
- Service (srv) types: service descriptions define the request and response data structures for services in ROS.

The Filesystem Level is depicted in Figure 14.

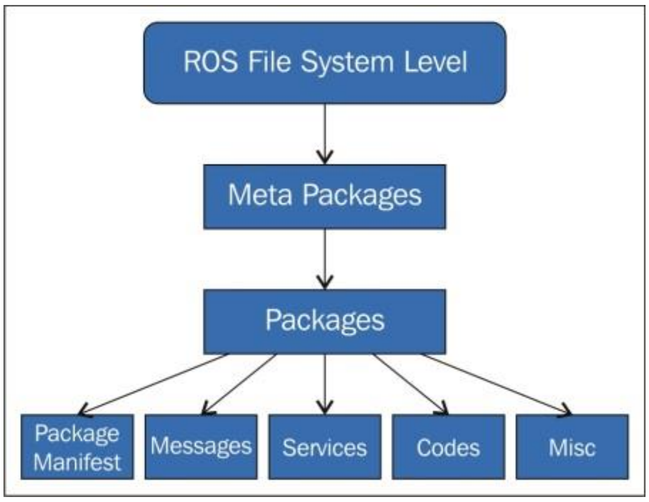


Figure 14: The ROS Filesystem Level

4.1.2 The Computational Graph Level

ROS creates a network where all the processes are connected. Any node in the system can access this network, interact with other nodes, see the information that they are sending, and transmit data to the network [62]:

1. Nodes: nodes are processes where computation is done. If someone wants to have a process that can interact with other nodes, she needs to create a node with this process to connect it to the ROS network. Usually, a system will have many nodes to control different functions.
2. Master: The Master provides name registration and lookup for the rest of the nodes. If it is not in the user’s system, she cannot communicate with nodes, services, messages, and others, but it is possible to have it in a computer where nodes work in other computers.
3. Parameter Server: The Parameter Server gives the possibility to have data stored using keys in a central location. With this parameter, it is possible to configure nodes while it is running or to change the working of the nodes.
4. Messages: Nodes communicate with each other through messages. A message contains data that sends information to other nodes. ROS has many types of messages, and someone can develop her own type of message using standard messages.
5. Topics: Each message must have a name to be routed by the ROS network. When a node is sending data, we say that the node is publishing a topic. Nodes can receive topics from other nodes simply by subscribing to the topic. A node can subscribe to a topic, and it isn't necessary that the node that is publishing this topic should exist. This permits to decouple the production of the consumption.
6. Services: When someone publishes topics, she is sending data in a many-to-many fashion, but when she needs a request or an answer from a node, she cannot do it with topics. The services give us the possibility to interact with nodes. Also, services must have a unique name. When a node has a service, all the nodes can communicate with it, thanks to ROS client libraries.

7. Bags: Bags are a format to save and play back the ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

The Computational Graph Level is depicted in Figure 15.

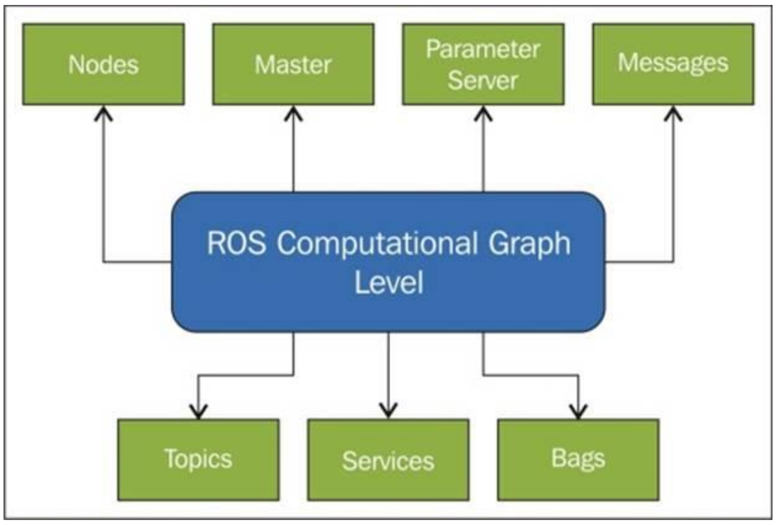


Figure 15: The ROS Computational Graph Level

4.1.3 The Community Level

The ROS Community level concepts are ROS resources that enable separate communities to exchange software and knowledge. These resources include [63]:

1. Distributions: ROS distributions are collections of versioned stacks that someone can install. ROS distributions play a similar role to Linux distributions. They make it easier to install a collection of software, and they also maintain consistent versions across a set of software.
2. Repositories: ROS relies on a federated network of code repositories, where different institutions can develop and release their own robot software components.
3. The ROS Wiki: The ROS Wiki is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute their own documentation, provide corrections or updates, write tutorials, and more.
4. Mailing lists: The ROS user-mailing list is the primary communication channel about new updates to ROS as well as a forum to ask questions about the ROS software.

4.2 Turtlesim

The TurtleBot [64] is a low-cost, personal robot kit with open source software that integrates in Microsoft’s Kinect, Yujin Robot’s Kobuki, and others, and can be combined with ROS. ROS provides the Turtlesim, a simulator that uses the turtlesim_node for teaching ROS concepts to TurtleBot.

4.2.1 Messages and Topics

Regarding the messages used in the simulator, the geometry_msg/Twist is the most common, which expresses the simulated turtle's velocity into linear and angular.

As for the subscribed topics, the turtleX/cmd_vel is used, which sets the linear angular velocity of turtleX that needs to be executed; note here that since these quantities express velocity, the turtle will execute the command within 1 sec.

Finally, the public topic is the turteleX/Pose, which informs the application about the coordinates (x, y), the angle θ, the linear, and the angular velocity of the turtle at that moment.

An example of the Turtlesim is depicted in Figure 16.

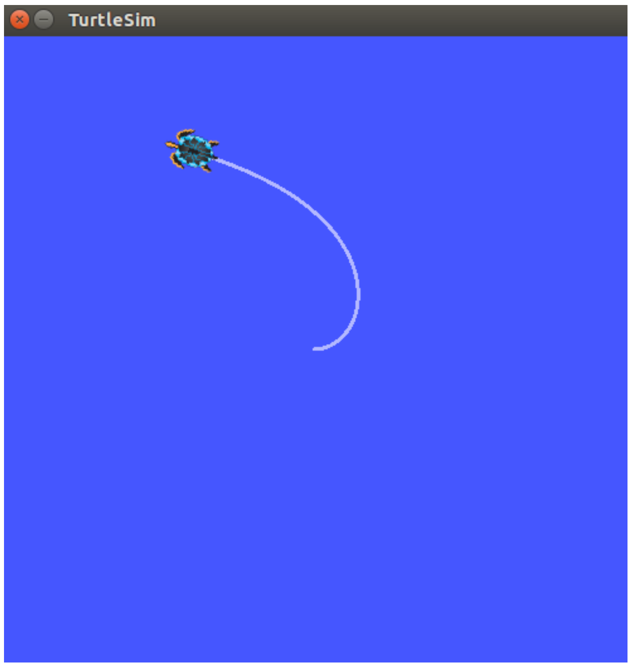


Figure 16: A Turtlesim example

4.3 ROS Web

Having explained the ROS and its functionalities, there is still on missing part to make sure that we can adopt ROS to our implementation: is there a way to interact with ROS from the browser? Fortunately, ROS provides the Rosbridge suite [65], which provides a JSON API to ROS functionality for non-ROS programs. There are a variety of front ends that interface with rosbridge, including a WebSocket server for web browsers to interact with. Rosbridge_suite is a meta-package containing rosbridge, various front end packages for rosbridge like a WebSocket package, and helper packages.

4.3.1 Rosbridge suite

Rosbridge consists of two parts:

1. The Rosbridge Protocol: a specification for sending JSON based commands to ROS, which is programming language and transport agnostic. The idea is that any language or transport that can send JSON can talk the rosbridge protocol

and interact with ROS. The protocol covers subscribing and publishing topics, service calls, getting and setting params, and even compressing messages and more.

2. The Rosbridge Implementation: the rosbridge suite package is a collection of packages that implement the rosbridge protocol, and provides a WebSocket transport layer. The packages include:
    a. The rosbridge_library: the core rosbridge package. It is responsible for taking JSON string and sending the commands to ROS and vice versa.
    b. rosapi: makes certain ROS actions accessible via service calls that are normally reserved for ROS client libraries.
    c. rosbridge_server: while rosbridge_library provides the JSON ←→ ROS conversion, it leaves the transport layer to others. Rosbridge_server provides a WebSocket connection so browsers can "talk rosbridge". Roslibjs [66] is a JavaScript library for the browser that can talk to ROS via rosbridge_server.

4.3.2 Roslibjs

Roslibjs is the core JavaScript library for interacting with ROS from the browser, and is developed as part of the Robot Web Tools effort [67]. It uses WebSockets to connect with rosbridge and provides publishing, subscribing, service calls, actionlib, TF, URDF parsing, and other essential ROS functionality. Roslibjs provides an API with all the appropriate JavaScript methods and objects for connecting to the rosbridge_server, and building JSON actions. Finally, roslib either exists locally or can be fetched as a pre-built executable script from a CDN server (Figure 17).

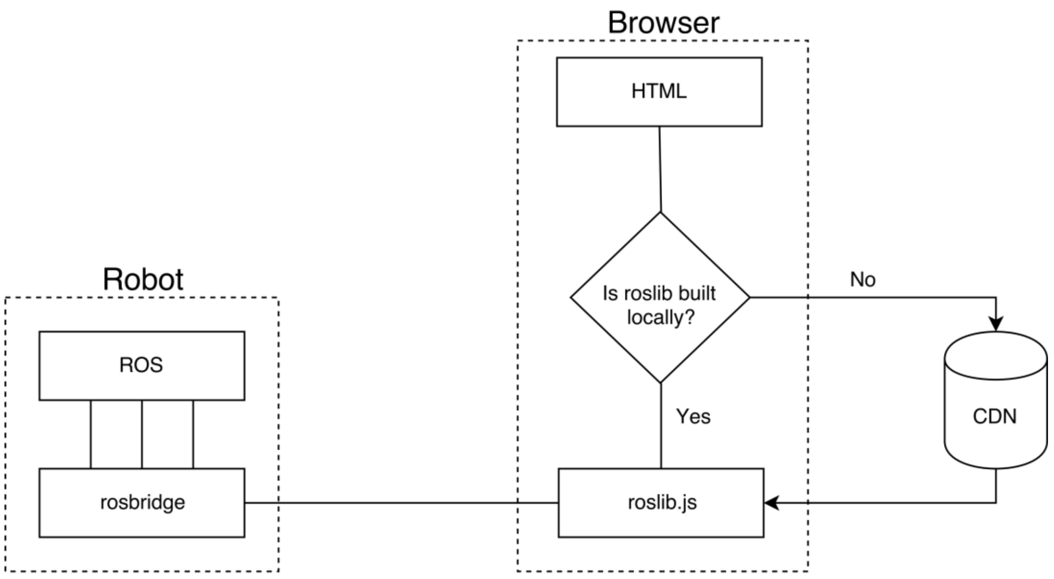


Figure 17: Interacting with ROS from the Browser using roslib and rosbridge

5. ROS-BASED VEHICLE AUTOPILOTING USING WEBRTC VIDEO ANALYSIS THROUGH OPENCV.JS

Having fully explained and described the three core components of our implementation (WebRTC – OpenCV – ROS), we will combine them properly to compose a web-based infrastructure for autopiloting ROS vehicles upon a specific use-case, by analyzing through OpenCV the video exchanged with WebRTC.

The idea is to build a horizontal infrastructure that:

1. Consists of a modular architecture
2. Provides the necessary components for machine-to-machine communication
3. Uses state-of-the-art technologies (i.e. WebRTC)
4. Allows a developer to freely implement her own logic vertically (i.e. her own OpenCV functions, autopiloting handling, use-cases etc.)
5. Provides IoT with a solution that can be easily adopted to many use-cases, and be exploited in numerous ways.

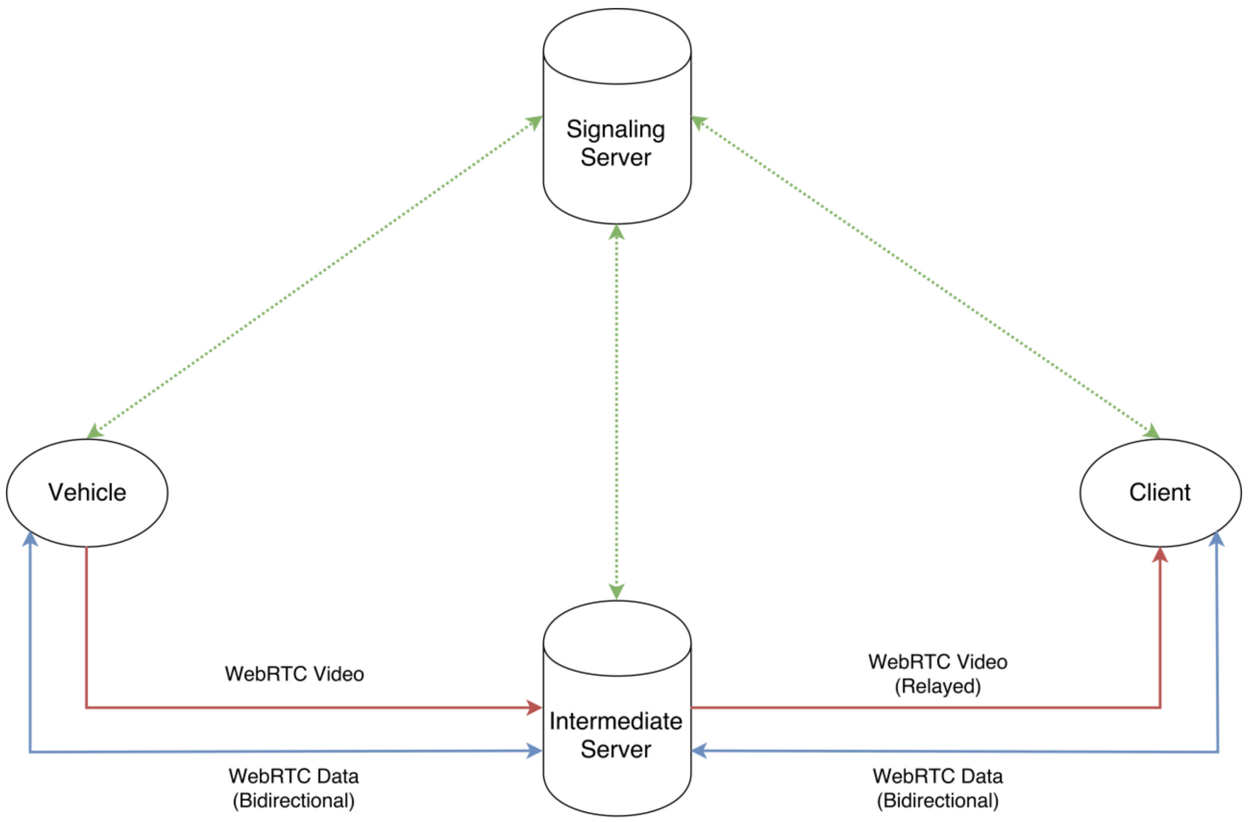


Figure 18: The core idea of the infrastructure

An overview of this idea is shown in Figure 18:

1. A Vehicle transmits its real-time video feed to an Intermediate Server using WebRTC.
2. The Intermediate Server relays the Vehicle’s video using WebRTC, and exposes its available operations using the Data Channel to a Client.

3. The Client monitors the Vehicle's vision and route, and selects the desirable operation (corresponding to a use-case, i.e. Face Detection) by informing properly the Intermediate Server through the Data Channel.
4. The Intermediate Server applies the corresponding OpenCV operations on the Vehicle's video, in order to server the selected use-case. After that, sends the appropriate commands to the Vehicle in order to pilot it.
5. What is more, the Client can dynamically choose a new operation (i.e. a new use-case), enabling the system to adopt to this new condition for autopiloting properly the Vehicle.
6. Finally, a Signaling Server orchestrates the whole communication between the Vehicle, the Intermediate Server, and the Client.

5.1 Architecture

Having in mind the aforementioned idea, let us get a little deeper by presenting the architecture of our infrastructure (Figure 19), and explaining how all these heterogeneous components fit together.

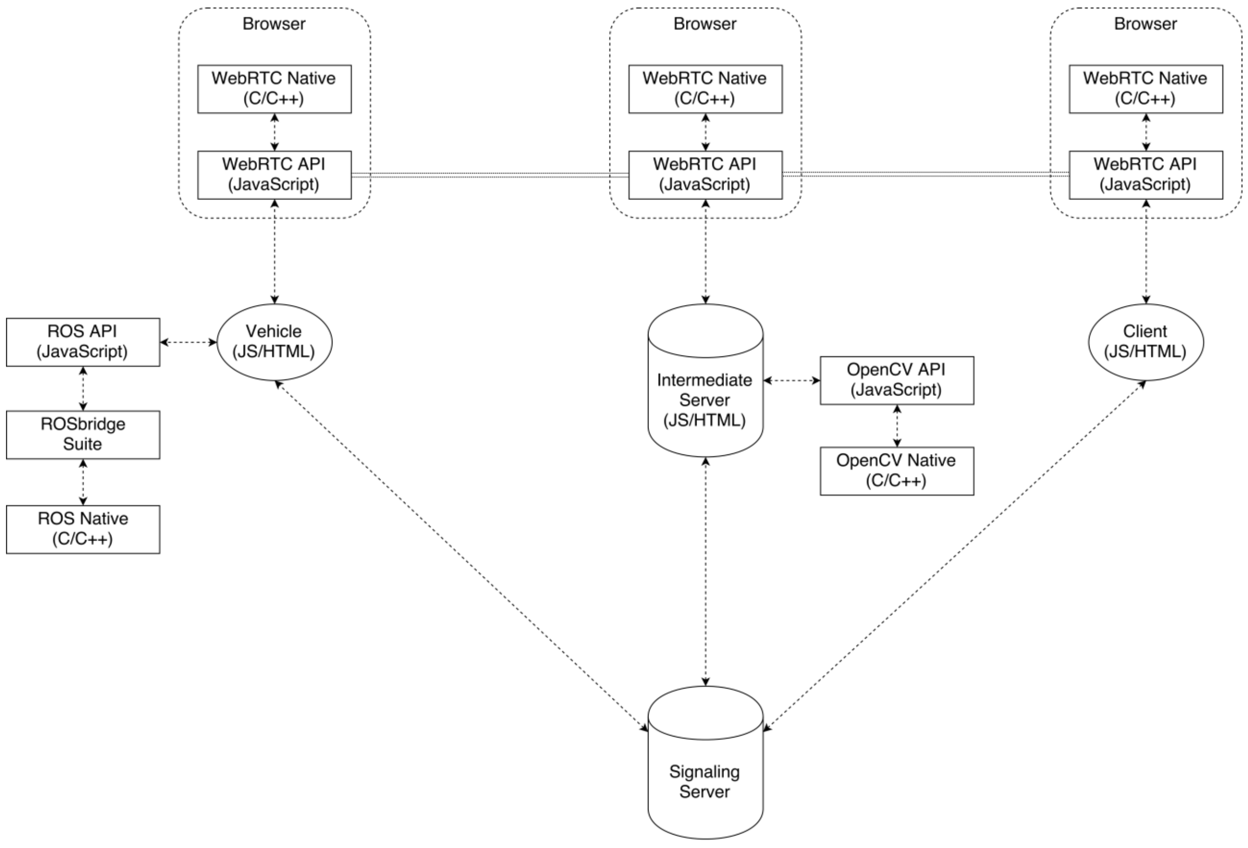


Figure 19: The architecture of the infrastructure

Starting from top, the communication between the individuals is achieved through the browser, enabling the exchange of multimedia and data using the WebRTC. Studying the underlying level, we implement three components as services, which communicate with each other by exploiting the WebRTC APIs:

1. Vehicle: consists of an HTML and a JavaScript file, and is responsible for delivering the appropriate commands to the ROS-based vehicle. Specifically, it uses the roslib API to connect to the rosbridge server and to construct the appropriate commands for interacting with ROS.
2. Intermediate Server: consists of an HTML and a JavaScript file, and is responsible for applying the appropriate techniques using OpenCV.js, which binds with the OpenCV executables. What is more, the Intermediate Server is responsible for servicing both the Vehicle and the Client.
3. Client: consists of an HTML and a JavaScript file, and is responsible for selecting the desired operation of the system, while monitoring the vehicle's vision and route. What is more, it can dynamically change the operation philosophy of the Intermediate Server, and can either start or stop the whole piloting on demand.
4. Signaling Server: consists of a JavaScript file, and is responsible for orchestrating the communication between the individuals. It handles and relays properly the transmitted messages, using WebSockets.

Having a deeper knowledge about the infrastructure, let us gradually analyze and explain in depth the implementation and the operation of each of the aforementioned entities.

5.2 Signalling Server

The first component to analyze is the Signaling Server. It consists of a JavaScript file that creates a WebSocket server for serving all the other individuals. Its role is to manage and properly relay a set of messages to the peers, in order to enable the communication through WebRTC. In the following sub-sections, the messages of this set are described.

5.2.1 Case: Login

Before establishing a WebRTC session, each individual must connect to the signaling server. Thus, the “Login” message requests to connect to the signaling server, and carries all the information required for the identification of each peer (i.e. name, IP, port etc.). The signaling server keeps a list of all the connected users, and responds with a success message.

5.2.2 Case: Offer

This message contains the SDP Offer of a peer A that wants to establish a connection with peer B. In our case, the peer A is either the Vehicle or the Client, whereas peer B is always the Intermediate Server. Thus, upon receipt of this message, the signaling server determines the name of the source (Vehicle or Client), as well as the destination name (Intermediate Server), and properly relays a message to the Intermediate Server, that includes the SDP Offer and the name of the source.

5.2.3 Case: Answer

This message contains the SDP Answer of a peer B to a peer A that wants to establish a connection. In our case, this is the Intermediate Server's Answer to the Vehicle's/Client's Offer. Thus, upon receipt of this message, the signaling server determines the name of the source (Intermediate Server), as well as the destination name (Vehicle or Client), and properly relays a message to the destination peer, that includes the SDP Answer and the name of the Intermediate Server.

5.2.4 Case: Candidate

This message contains the trickled ICE Candidates of peer A to peer B. The possible pair of peers can be {Intermediate Server, Vehicle}, and {Intermediate Server, Client}. Thus, upon receipt of this message, the signaling server determines the source and destination name, and properly relays a message to the destination peer, that includes the ICE Candidates and the name of the source.

5.2.5 Case: Leave

A peer that wants to leave the session sends this message. Thus, the signaling server relays it to all the other connected participants, in order for them to handle this event properly. Finally, the signaling server closes the connection with the gone peer.

5.2.6 Case: Start

This message comes from the Client's side and is meant for the Intermediate Server, in order to demand the start of the autopilot. The signaling server properly relays it to the Intermediate Server, declaring that the source is the Client.

5.2.7 Case: Stop

This message comes from the Client's side and is meant for the Intermediate Server, in order to demand the stop of the autopilot. The signaling server properly relays it to the Intermediate Server, declaring that the source is the Client.

5.2.8 Case: Get Operations

This message comes from the Client's side and is meant for the Intermediate Server, requesting the set of available operations. The signaling server properly relays it to the Intermediate Server, declaring that the source is the Client.

5.3 Client

The next component to analyze is the Client entity. It consists of an HTML file that provides a User Interface – UI, and a JavaScript file that contains the functionality behind the UI. Its role is to monitor the Vehicle's video, to log the received messages and data, to set or change the operation of the Intermediate Server, and to Start or Stop the autopiloting. In the following sub-sections, we will analyze each part of the Client's workflow diagram, depicted in Figure 20.

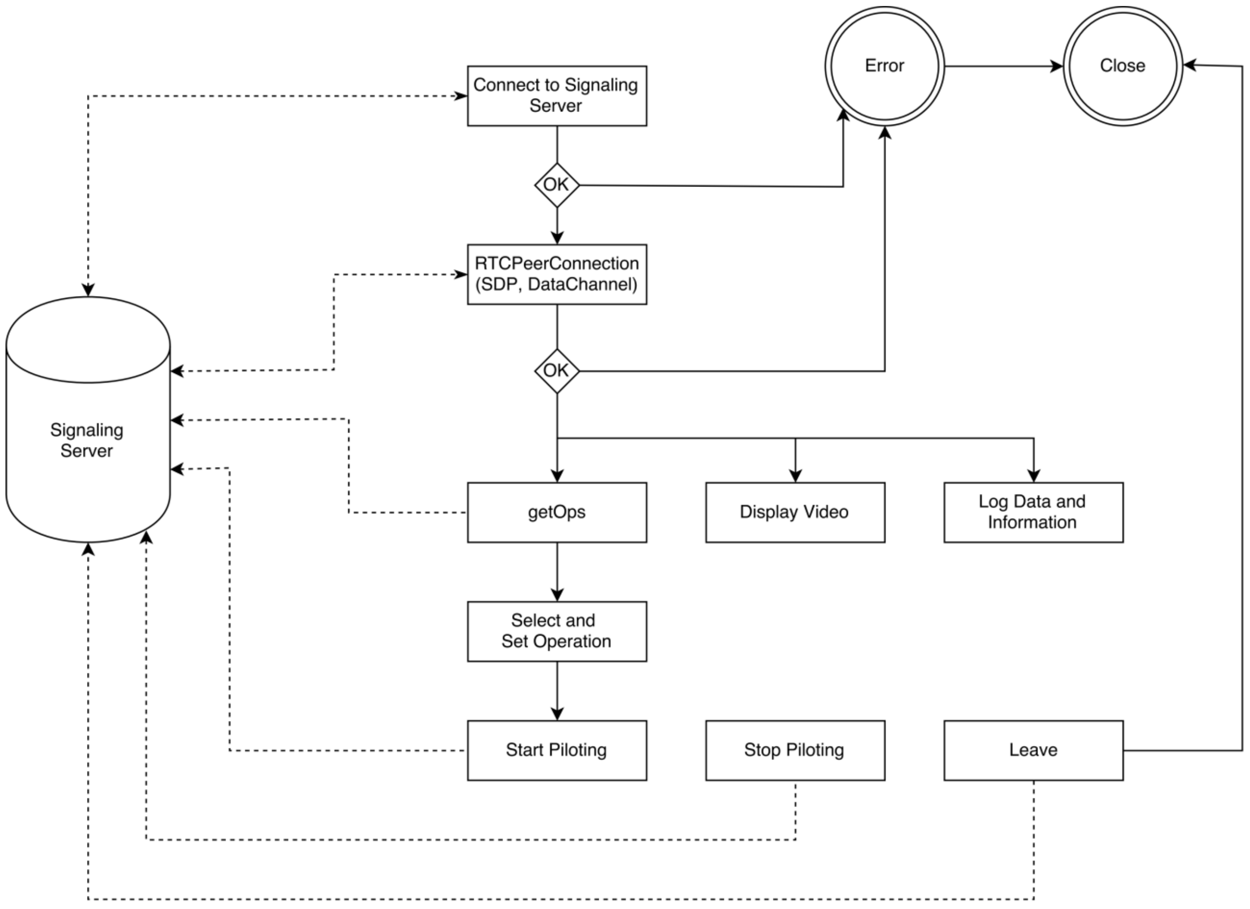


Figure 20: Client's Workflow

5.3.1 Connect to Signaling Server

As already mentioned, before establishing a WebRTC session the Client must connect to the signaling server, by sending the “Login” message. If the signaling server rejects the request, then the Client goes to an Error state, clears all the resources, and exits the program. Otherwise, continues to the PeerConnection state.

5.3.2 PeerConnection

At this state, the Client tries to establish a P2P connection with the Intermediate Server. Specifically, creates an SDP Offer, sets the Data Channel, defines the STUN url, and creates a PeerConnection, which requires only to receive video. Automatically, the SDP Offer and the ICE candidates are transferred through the signaling server, and the Offer/Answer procedure takes place. Finally, if everything goes smoothly and the PeerConnection is established, the peers start to exchange media and data, and the system is ready to go to the next states.

5.3.3 getOps – Display Video – Log Data and Information

After having successfully established a PeerConnection with the Intermediate Server, the Client automatically sends the “getOps” message through the signaling server, requesting the available operations form the Intermediate Server.At the same time, the Client receives the Vehicle’s video stream, as well as data from the Intermediate Server through the Data Channel. Therefore, this video is set as the srcObject of the Video Element in order to display it, while the data received are logged at a text area.

5.3.4 Select and Set Operation

The Intermediate Server’s response to the “getOps” message is transferred via the Data Channel. Upon reception, the Client properly parses the data to determine the available operations, and creates a drop-down menu with them. The user can select from the UI the operation she prefers, and inform properly the Intermediate Server through the Data Channel, by pressing the corresponding button.

5.3.5 Start and Stop Piloting

After selecting and setting the preferred operation, the user may choose to start the autopiloting, by sending the corresponding “Start” message through the signaling server; in a similar way, she may choose to stop the it, by sending the “Stop” message

5.3.6 Leave

At any time during the session, the Client may choose to leave. By sending the corresponding “Leave” message, she makes sure that all other individuals will be properly informed, and her session is terminated.

5.3.7 UI

The aforementioned procedures are “hidden” behind the UI depicted in Figure 21. The Connect button corresponds to the connection with the signaling server, the Join Room to the PeerConnection, while the rest buttons correspond to what is written on their labels. Finally, the black frame is the video area, while the white textbox is the log area.

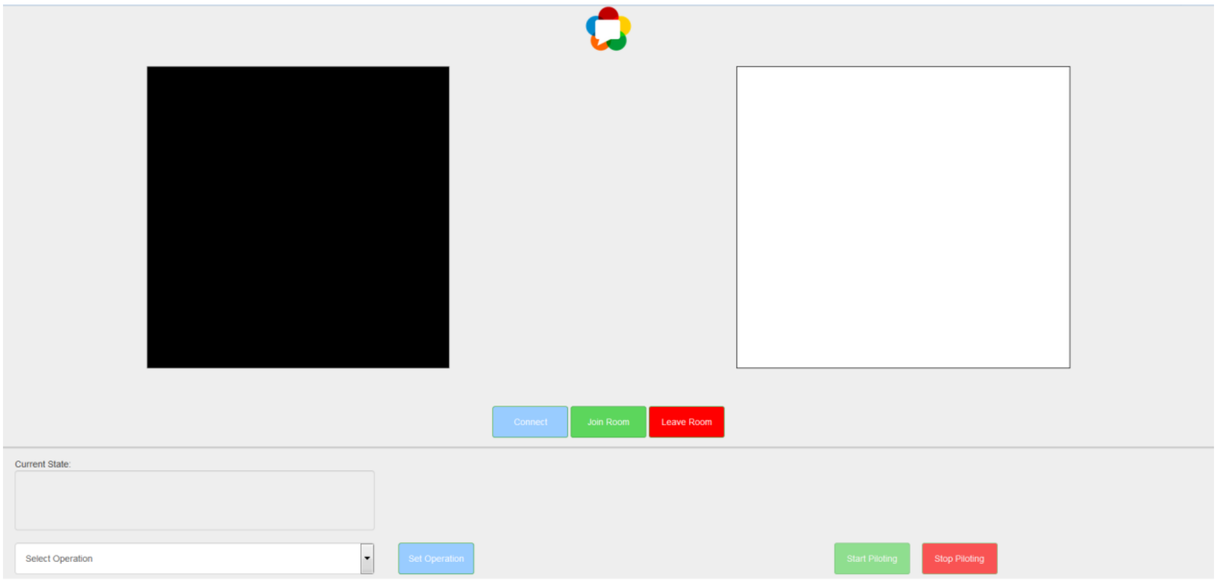


Figure 21: Client's UI

5.4 Intermediate Server

The third component to analyze is the Intermediate Server. It consists of an HTML file that provides a UI, and a JavaScript file that contains the functionality behind of the UI. Furthermore, the OpenCV.js bindings and APIs are also in a subdirectory, in order to have access to the OpenCV executables. Its role is to control the whole operation, to analyze the Vehicle's video, to relay it to the Client, to receive commands from the Client, and to send commands to the Vehicle. In the following sub-sections, we will analyze each part of the Intermediate Server's workflow diagram, depicted in Figure 22.

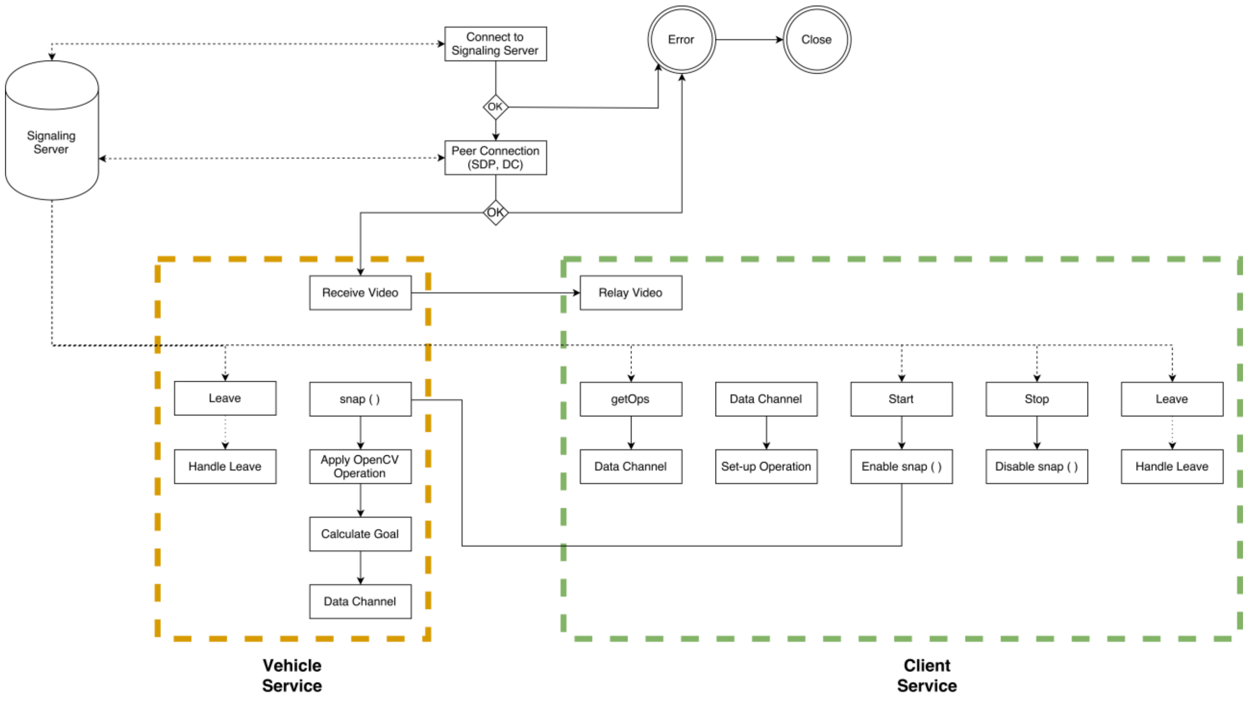


Figure 22: Intermediate Server's Workflow

5.4.1 Connect to Signaling Server

As already mentioned, before establishing a WebRTC session the Intermediate Server must connect to the signaling server, by sending the “Login” message. If the signaling server rejects the request, then the Intermediate Server goes to an Error state, clears all the resources, and exits the program. Otherwise, continues to the PeerConnection state.

5.4.2 PeerConnection

At this state, the Intermediate Server tries to establish a P2P connection with the other individuals. Specifically, creates an SDP Offer, sets the Data Channel, defines the STUN url, and creates a PeerConnection, which requires only to receive video. However, its status remains “Idle”, since it is the first to join the session, and it is thus waiting for incoming connections. Upon receipt of an SDP Offer, automatically, the SDP Answer and the ICE candidates are transferred through the signaling server, and the Offer/Answer procedure takes place. Finally, if everything goes smoothly and the PeerConnection is established, the peers start to exchange media and data, and the system is ready to go to the next states.

At this point, since the Intermediate Server may create a PeerConnection with either the Vehicle or the Client, it will have to operate in a corresponding way. Thus, as shown in Figure 22, we discern two service cases: the Client Service, and the Vehicle Service.

5.5 Intermediate Server – Client Service

The Client Service is responsible for interacting with the Client. This include the relay of the Vehicle’s video, as well as the exchange of information regarding the operations and the piloting.

5.5.1 Relay Video

Assuming that the Vehicle is already connected and streaming, the Intermediate relays the Vehicle’s video to the Client, by adding the incoming stream to its PeerConnection (with the Client) media object.

5.5.2 getOps – Expose operations

Upon receipt of the “getOps” message from the Signaling Server, the Intermediate Server send via the Data Channel to the Client all the available operations in a JSON format.

5.5.3 Data Chanel – Set-up Operations

When the Client sets the preferred operations, this is done through the Data Channel. Thus, when the Intermediate Server receives the corresponding data, it parses it, and enables the appropriate OpenCV operations for the selected use-case, by assigning the corresponding functionality to a global function.

5.5.4 Start – Enable Snapshots

Whenever the Intermediate Server receives the "Start" message from the Signaling Server, it immediately enable the snap function, which is responsible for the whole piloting operation. The snap function is part of the Vehicle Service, and will be explained in 5.6.

5.5.5 Stop – Disable Snapshots

Whenever the Intermediate Server receives the "Stop" message from the Signaling Server, it immediately disable the aforementioned snap function.

5.5.6 Leave

Upon receipt of this message, the Intermediate Server disconnects form the Client and terminates any operation with the Vehicle.

5.6 Intermediate Server – Vehicle Service

The Vehicle Service is responsible for interacting with the Vehicle entity. This include the reception of the video stream, and the exchange of data that corresponds to the calculated goal, which the Vehicle will transform/translate into ROS commands.

5.6.1 Receive Video

The Vehicle's incoming video stream is passed to the srcObject element, in order to manipulate it, and is relayed to the Client.

5.6.2 Snapshot function

As mentioned in the Client Service, upon the receipt of the "Start" message, the snap function is enabled. This is done through the JavaScript timing event "setInterval", which actually calls the snap function periodically. This function is responsible for taking a snapshot of the video, to forward it to OpenCV, to calculate the goal from the processing results, and finally to send it via the Data Channel to the Vehicle. The period of the snap function is determined based on the needs of the piloting, but the minimum snapshot period cannot be less than 1/fps.

However, we define a period of ~1500 milliseconds. Let us explain:

- Let the selected use-case be the Face Detection. We obtained some metrics and found out that from the time the function is called until the time the goal is sent via the Data Channel, the total duration is about 800 ms.
- Now, remember that we are going to test our system on a Turtlesim, where the execution of one command takes exactly 1000 ms, since it corresponds to velocity (note that we do not use the Pose topic, which enables us to interrupt the turtle's movement with accuracy). Considering some communication and synchronization delays, if the turtle receives a command at time *t* then it will be able to execute the next command at time *t + 1000 + delays ≈ t + 1100 ms*.
- So, if we obtain a snapshot at time *t*, then the turtle will receive it at time *t + 800* and will be ready to execute the next command at time *t + 800 + 1100 = t + 1900*,

meaning almost two seconds later. If this delay propagates in time, it is easy to understand that the turtle will “always be late” if the snapshot period is quite short.

- With some experiments and some synchronization at the Vehicle’s procedures (see 5.7.7), we determined that a sampling period of ~1500 ms is good enough for a smoother piloting.

The OpenCV operation that we are testing is the Face Detection case. In order to achieve this, we convert a video snapshot to Canvas element and get the image data. We convert this data to a Mat array, and we apply the face cascade classifier that OpenCV provides in an XML format. If any face is detected, we enclose it in a rectangle.

The next thing to do is to calculate the Goal. We define as the target of the turtle to be the central point of the face’s rectangle. The use-case is a little tricky; consider that the camera is somewhere above and is looking at a terrain, and we want to detect something specific that is moving on it. We consider that the object for detection is our face, and thus we reduce it to a single point. This point (rectangle’s central point) is the object on the terrain, and we need the turtle to rotate properly in order to stare at it.

Finally, the calculated coordinates of the central point (x_m, y_m) are sent via the Data Channel to the Vehicle.

5.6.3 Leave

Upon receipt of this message, the Intermediate Server terminates any operation with the Vehicle, and disconnects from it.

5.7 Vehicle

The final component to analyze is the Vehicle. It consists of an HTML file that provides a UI, and a JavaScript file that contains the functionality behind of the UI. Its role is to connect to the rosbridge_server for interacting with ROS, to stream its video to the Intermediate Server, and to calculate and send the appropriate commands to the simulator. In the following sub-sections, we will analyze each part of the Vehicle’s workflow diagram, depicted in Figure 23.

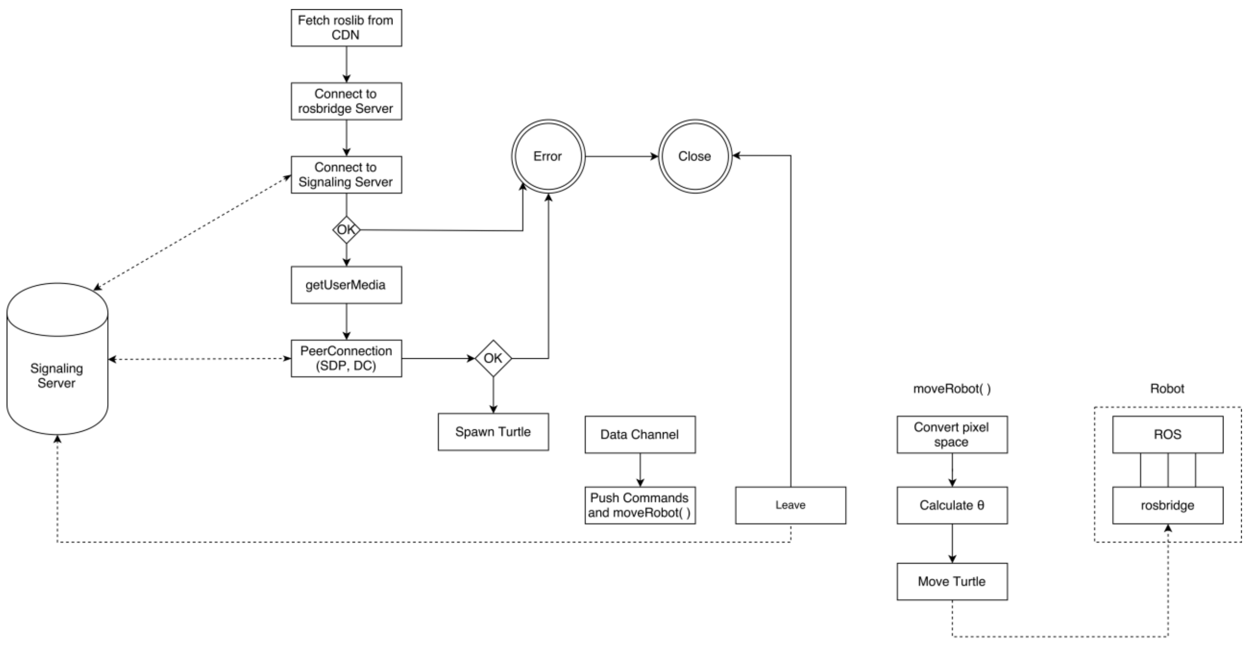


Figure 23: Vehicle's Workflow

5.7.1 Fetch roslib

The first thing to do is to fetch the pre-built version of roslib.js from a CDN, in order to use the APIs and the functionalities to interact with ROS.

5.7.2 Connect to rosbridge Server

Then, we connect to the reosbridge_server by creating a new Ros object, which establishes a WebSocket connection with the reosbridge_server. Note here that the rosbridge_server and the turtlesim must be running on the ROS device to enable the connection. Finally, after setting up the connection, we can exchange data and interact with the simulator

5.7.3 Connect to Signaling Server

As already mentioned, before establishing a WebRTC session the Vehicle must connect to the signaling server, by sending the “Login” message. If the signaling server rejects the request, then the Intermediate Server goes to an Error state, clears all the resources, and exits the program. Otherwise, continues to the PeerConnection state.

5.7.4 getUserMedia

After successfully connecting to the Signaling Server, the Vehicle calls the getUserMedia method of the MediaStream API in order to get access to the camera’s video stream. This video stream will be added on the PeerConnection object, in order to be transferred to the Intermediate Server.

5.7.5 PeerConnection

At this state, the Vehicle tries to establish a P2P connection with the Intermediate Server. Specifically, creates an SDP Offer, sets the Data Channel, defines the STUN url, and creates a PeerConnection, which the video stream. Automatically, the SDP Offer and the ICE candidates are transferred through the signaling server, and the Offer/Answer procedure takes place. Finally, if everything goes smoothly and the PeerConnection is established, the peers start to exchange media and data, and the system is ready to go to the next state.

5.7.6 Spawn Turtle

At this state, the Vehicle publishes a Twist message to the Turtlesim, in order to set the default orientation of the turtle at π/2 rads.

5.7.7 Move Robot

The calculated goal for the turtle is received from the Data Channel. When the first goal arrives, the Vehicle calls the moveRobot function. Then, creates a new timing event through the setInterval method, in order to periodically call the moveRobot function; this period, as explained before, is set to 1100 ms. After the first one, the rest of the goals are stored in a variable, in order to synchronize the turtle with the latest face position. The moveRobot is responsible for constructing the appropriate commands based on the turtle's current position and the goal:

At first, the goal point must be transformed properly in order to match the space of the turtle's terrain (Figure 24). More specifically, the turtle's space is smaller than the video's pixel space.

Let:

$$\left(t_{x_c}, t_{y_c}\right)$$

be the central point of the turtle's terrain (actually that is the spawn coordinates of the turtle).

Let:

$$(x_c,\, y_c) = \left(\frac{W}{2},\, \frac{H}{2}\right)$$

be the video's central pixel, where W and H are the Width and Height respectively.

Finally, let:

$$(x_m, y_m)$$

be the rectangle's central point.

Then, the following transformation expresses the coordinates of the rectangle's central point on the turtle's space:

$$\left(x_{m_t},\, y_{m_t}\right) = \left(x_m * \frac{t_{x_c}}{x_c},\, y_m * \frac{t_{y_c}}{y_c}\right)$$

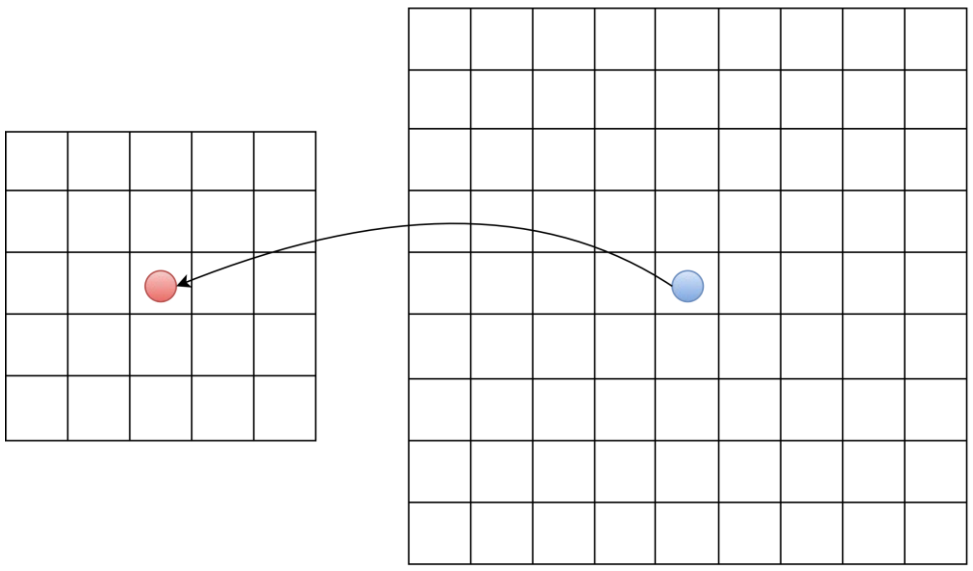


Figure 24: Pixel space to Turtlesim space transformation

After successfully transforming the coordinates of the central point, the moveRobot function must calculate the angles that the turtle must steer in order to stare at the goal point. First of all, the angle between the turtle's position and the central point is calculated:

$$\theta = \tan^{-1}\left(\frac{y_{m_t} - y_{t_{current}}}{x_{m_t} - x_{t_{current}}}\right)$$

Let:

$$\theta_t$$

be the turtle's current angle.

The absolute angle difference $|\theta_t - \theta|$ is calculated, and the following conditions provide the radians the turtle must steer in order to stare at the goal:

$$|\theta_t - \theta| \leq 2\pi \Rightarrow \begin{cases} ang = |\theta_t - \theta|, & \theta_t < \theta \\ ang = -\,|\theta_t - \theta|, & \theta_t \geq \theta \end{cases}$$

$$|\theta_t - \theta| > 2\pi \Rightarrow \begin{cases} ang = -\theta_t - (2\pi - \theta), & \theta_t < \theta \\ ang = 2\pi - \theta_t + \theta, & \theta_t \geq \theta \end{cases}$$

Finally, the calculated "ang" is set as the angular of z-axis (angular velocity) in the Twist message, which is published to the turtle (command).

5.7.8 Leave

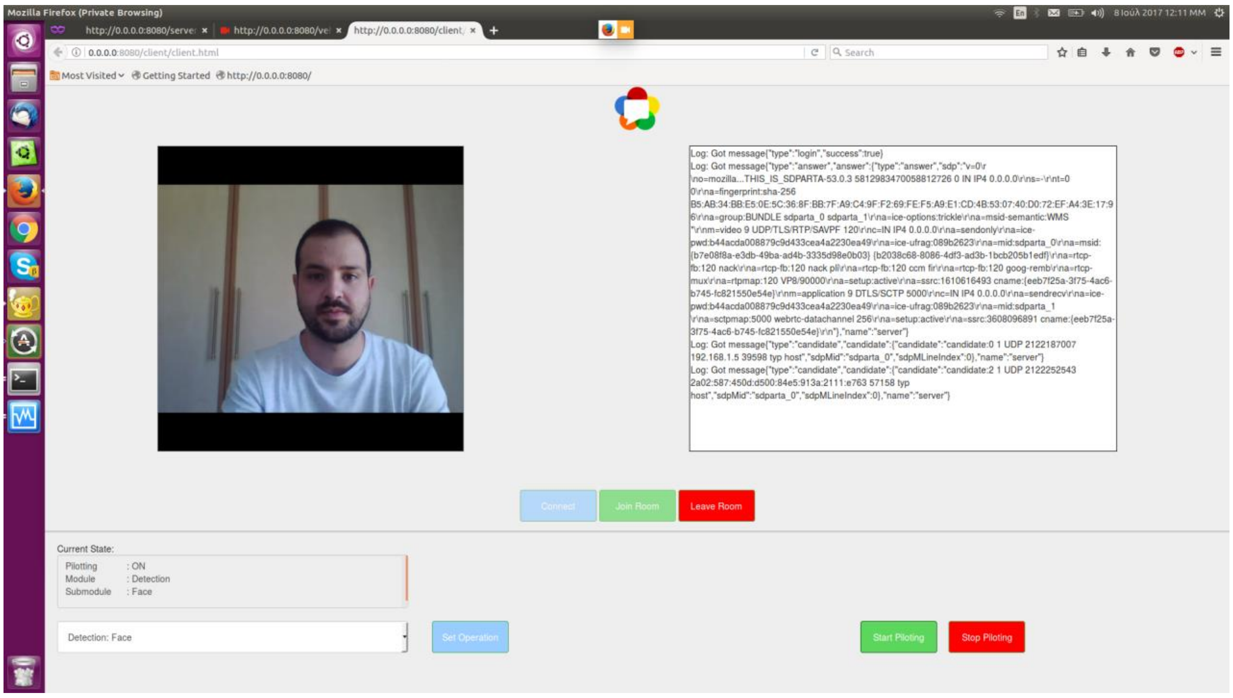
Finally, when we want to disconnect the Vehicle from the system, the “Leave” message is sent through the Signaling Server, in order to inform properly the other individuals.

6. RESULTS

In the following sub-sections, we will try to prove our concept of this technical instrumentation, and to provide some metrics about the processing delay of the autopiloting procedure.

6.1 WebRTC

First, we will prove that the real-time communication is achieved through WebRTC. Figure 25 shows the Vehicle's video that is successfully relayed to the Client, while Figure 26 depicts the data transferred to the Vehicle through the Data Channel. Finally, Figure 27 shows how the getUserMedia method requires access to the video stream from the Laptop's camera.

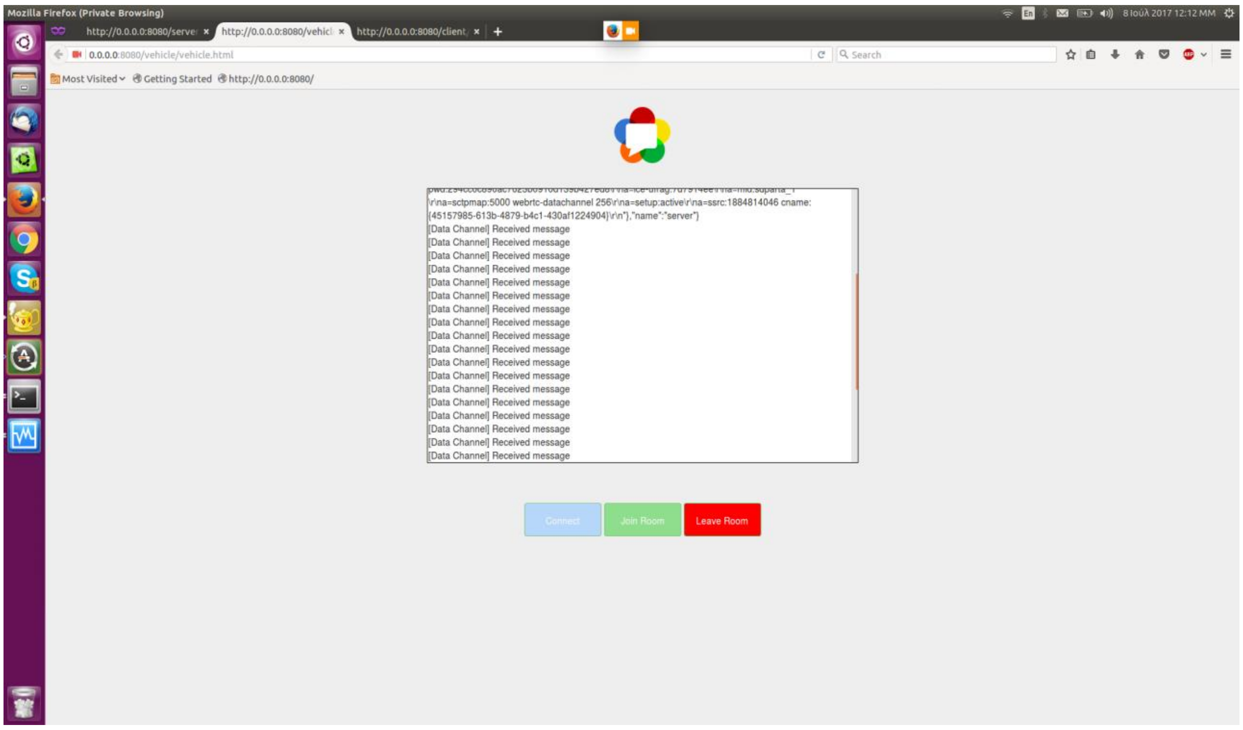

```
Log: Got message{"type":"login","success":true}
Log: Got message{"type":"answer","answer":{"type":"answer","sdp":"v=0\r
\no=mozilla...THIS_IS_SDPARTA-53.0.3 5812983470058812726 0 IN IP4 0.0.0.0\r\ns=-\r\nt=0
0\r\na=fingerprint:sha-256
B5:AB:34:BB:E5:0E:5C:36:8F:BB:7F:A9:C4:9F:F2:69:FE:F5:A9:E1:CD:4B:53:07:40:D0:72:EF:A4:3E:17:9
6\r\na=group:BUNDLE sdparta_0 sdparta_1\r\na=ice-options:trickle\r\na=msid-semantic:WMS
*\r\nm=video 9 UDP/TLS/RTP/SAVPF 120\r\nc=IN IP4 0.0.0.0\r\na=sendonly\r\na=ice-
pwd:b44acda008879c9d433cea4a2230ea49\r\na=ice-ufrag:089b2623\r\na=mid:sdparta_0\r\na=msid:
{b7e08f8a-e3db-49ba-ad4b-3335d98e0b03} {b2038c68-8086-4df3-ad3b-1bcb205b1edf}\r\na=rtcp-
fb:120 nack\r\na=rtcp-fb:120 nack pli\r\na=rtcp-fb:120 ccm fir\r\na=rtcp-fb:120 goog-remb\r\na=rtcp-
mux\r\na=rtpmap:120 VP8/90000\r\na=setup:active\r\na=ssrc:1610616493 cname:{eeb7f25a-3f75-4ac6-
b745-fc821550e54e}\r\nm=application 9 DTLS/SCTP 5000\r\nc=IN IP4 0.0.0.0\r\na=sendrecv\r\na=ice-
pwd:b44acda008879c9d433cea4a2230ea49\r\na=ice-ufrag:089b2623\r\na=mid:sdparta_1
\r\na=sctpmap:5000 webrtc-datachannel 256\r\na=setup:active\r\na=ssrc:3608096891 cname:{eeb7f25a-
3f75-4ac6-b745-fc821550e54e}\r\n"},"name":"server"}
Log: Got message{"type":"candidate","candidate":{"candidate":"candidate:0 1 UDP 2122187007
192.168.1.5 39598 typ host","sdpMid":"sdparta_0","sdpMLineIndex":0},"name":"server"}
Log: Got message{"type":"candidate","candidate":{"candidate":"candidate:2 1 UDP 2122252543
2a02:587:450d:d500:84e5:913a:2111:e763 57158 typ
host","sdpMid":"sdparta_0","sdpMLineIndex":0},"name":"server"}
```

Current State:

Piloting : ON
Module : Detection
Submodule : Face

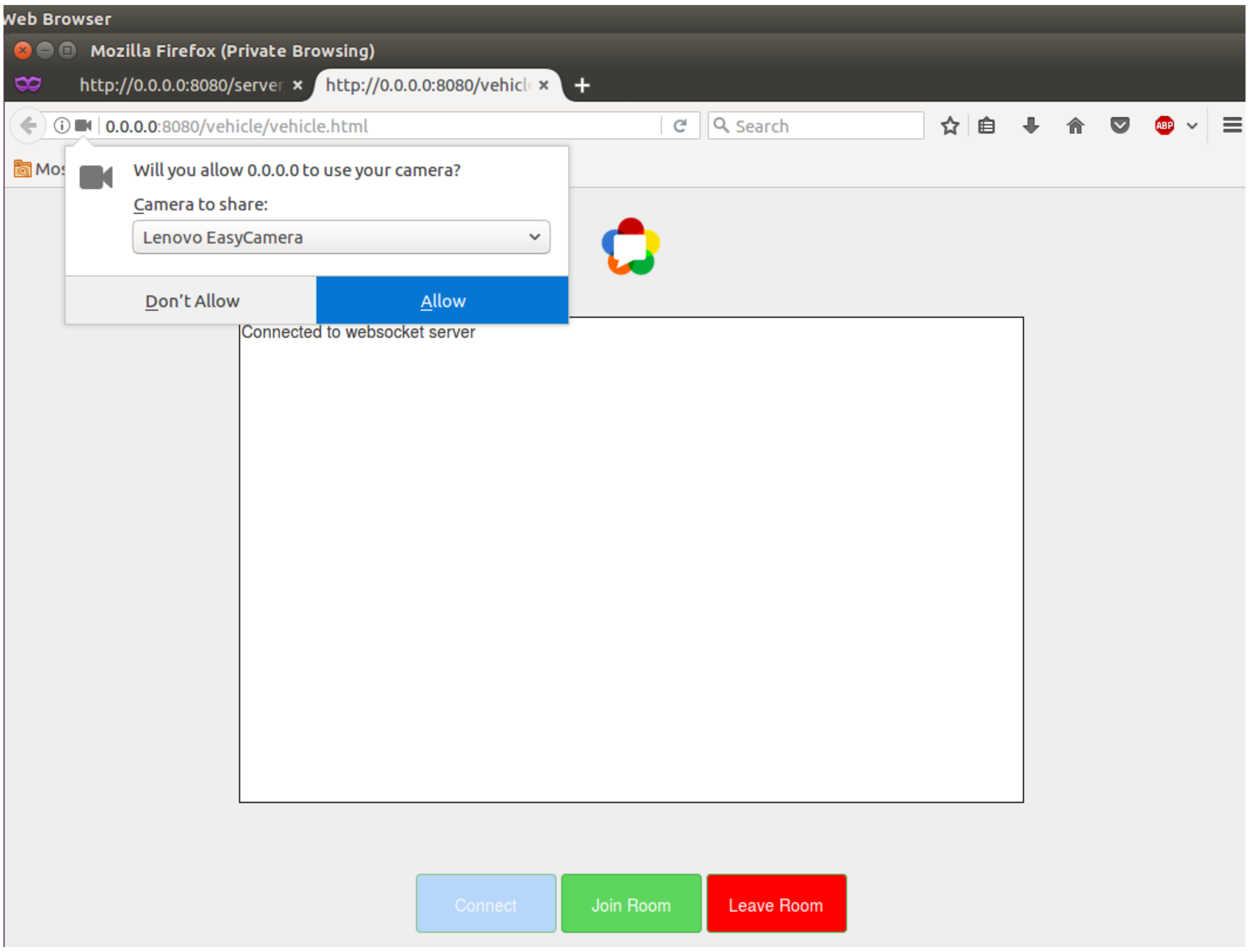
Detection: Face

Figure 25: Proof of concept: WebRTC Video



```
\r\na=sctpmap:5000 webrtc-datachannel 256\r\na=setup:active\r\na=ssrc:1884814046 cname:
{45157985-613b-4879-b4c1-430af1224904}\r\n"},"name":"server"}
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
[Data Channel] Received message
```

Figure 26: Proof of concept: WebRTC Data Channel



Will you allow 0.0.0.0 to use your camera?

Camera to share: Lenovo EasyCamera

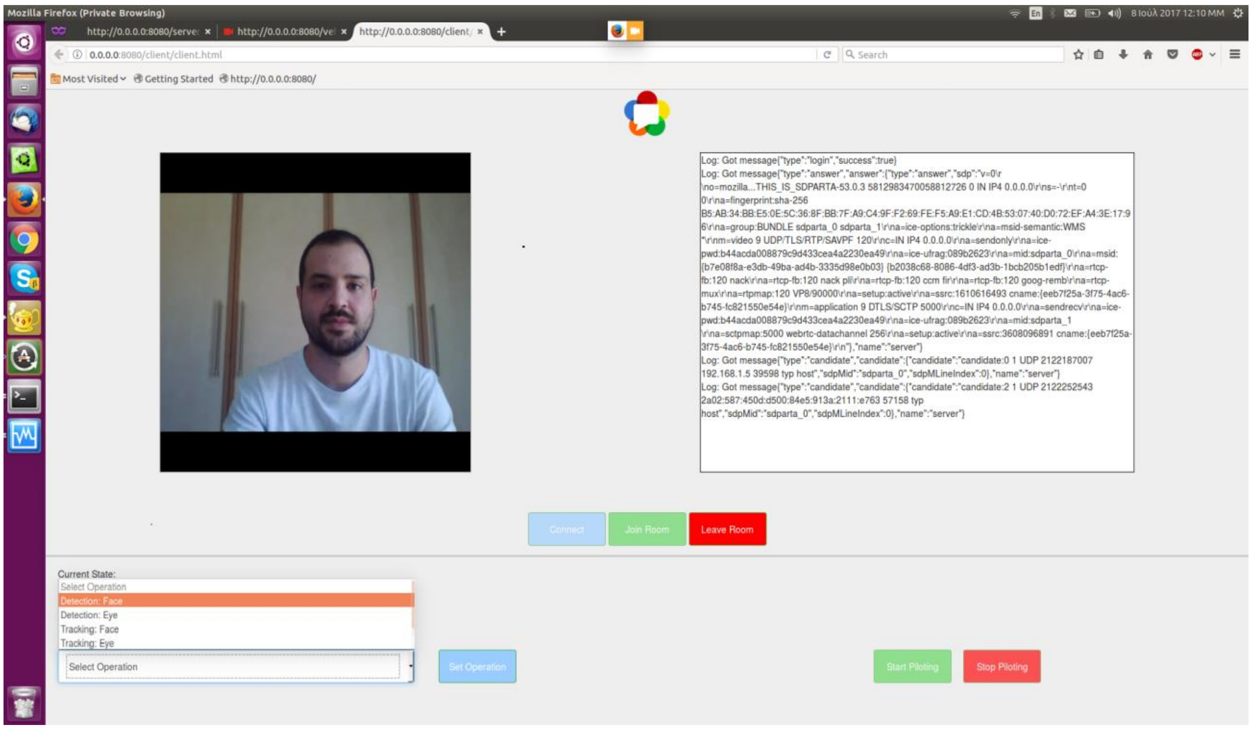
```
Connected to websocket server
```

Web Browser
Mozilla Firefox (Private Browsing)
http://0.0.0.0:8080/server
http://0.0.0.0:8080/vehicl
0.0.0.0:8080/vehicle/vehicle.html
Search
Don't Allow
Allow
Connect
Join Room
Leave Room

Figure 27: Proof of concept: getUserMedia method

6.2 Adaptive

Figure 28 shows how the Client can dynamically select a new operation, forcing the system to adapt to this new use-case.



```
Log: Got message{"type":"login","success":true}
Log: Got message{"type":"answer","answer":{"type":"answer","sdp":"v=0\r
\no=mozilla...THIS_IS_SDPARTA-53.0.3 5812983470058812726 0 IN IP4 0.0.0.0\r\ns=-\r\nt=0
0\r\na=fingerprint:sha-256
B5:AB:34:BB:E5:0E:5C:36:8F:BB:7F:A9:C4:9F:F2:69:FE:F5:A9:E1:CD:4B:53:07:40:D0:72:EF:A4:3E:17:9
6\r\na=group:BUNDLE sdparta_0 sdparta_1\r\na=ice-options:trickle\r\na=msid-semantic:WMS
*\r\nm=video 9 UDP/TLS/RTP/SAVPF 120\r\nc=IN IP4 0.0.0.0\r\na=sendonly\r\na=ice-
pwd:b44acda008879c9d433cea4a2230ea49\r\na=ice-ufrag:089b2623\r\na=mid:sdparta_0\r\na=msid:
{b7e08f8a-e3db-49ba-ad4b-3335d98e0b03} {b2038c68-8086-4df3-ad3b-1bcb205b1edf}\r\na=rtcp-
fb:120 nack\r\na=rtcp-fb:120 nack pli\r\na=rtcp-fb:120 ccm fir\r\na=rtcp-fb:120 goog-remb\r\na=rtcp-
mux\r\na=rtpmap:120 VP8/90000\r\na=setup:active\r\na=ssrc:1610616493 cname:{eeb7f25a-3f75-4ac6-
b745-fc821550e54e}\r\nm=application 9 DTLS/SCTP 5000\r\nc=IN IP4 0.0.0.0\r\na=sendrecv\r\na=ice-
pwd:b44acda008879c9d433cea4a2230ea49\r\na=ice-ufrag:089b2623\r\na=mid:sdparta_1
\r\na=sctpmap:5000 webrtc-datachannel 256\r\na=setup:active\r\na=ssrc:3608096891 cname:{eeb7f25a-
3f75-4ac6-b745-fc821550e54e}\r\n"},"name":"server"}
Log: Got message{"type":"candidate","candidate":{"candidate":"candidate:0 1 UDP 2122187007
192.168.1.5 39598 typ host","sdpMid":"sdparta_0","sdpMLineIndex":0},"name":"server"}
Log: Got message{"type":"candidate","candidate":{"candidate":"candidate:2 1 UDP 2122252543
2a02:587:450d:d500:84e5:913a:2111:e763 57158 typ
host","sdpMid":"sdparta_0","sdpMLineIndex":0},"name":"server"}
```

Figure 28: Proof of concept: Adaptive

6.3 OpenCV.js

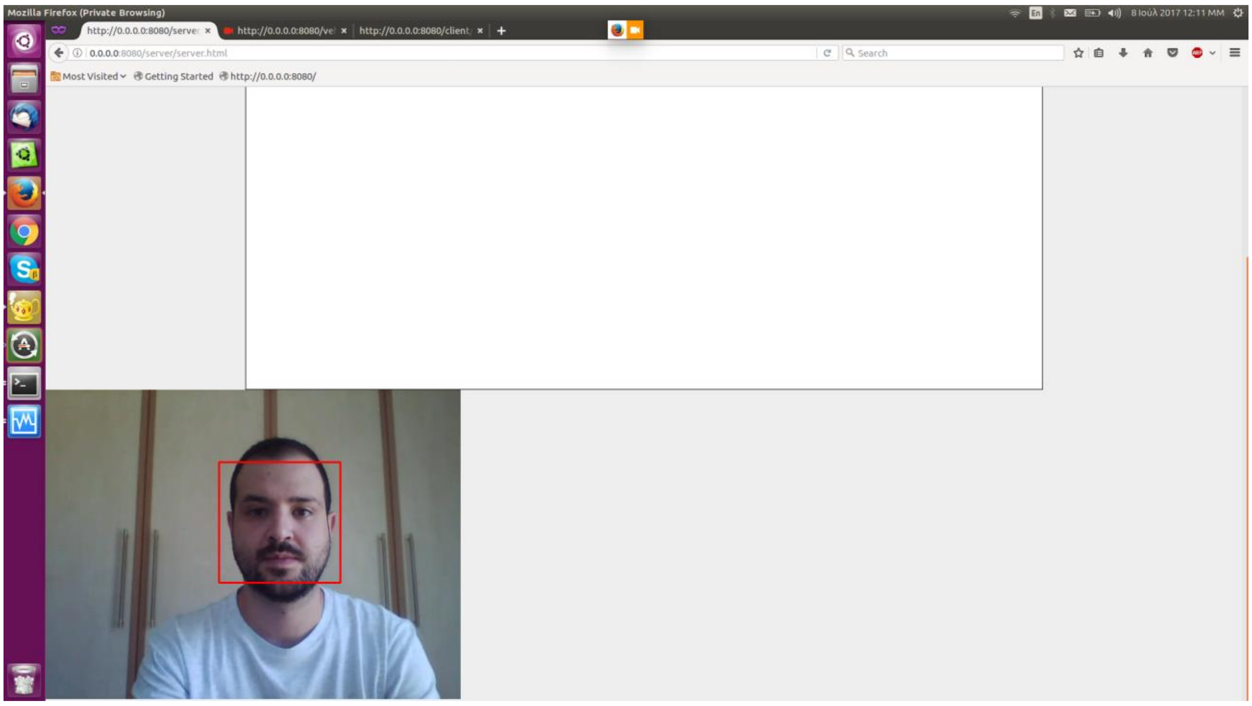


Figure 29: Proof of concept: OpenCV.js

Figure 29 shows the results of the Face Detection method. As we can see, OpenCV.js successfully recognizes the face area and encloses it into a rectangle

6.4 ROS-based Vehicle

Figure 30 depicts the autopiloting of the simulated turtle. As we can see, at first, the detected face is at the upper-left corner, and the turtle is staring at it. The, we make a 180 degree move, and the turtle successfully changes its orientation to the correct view.

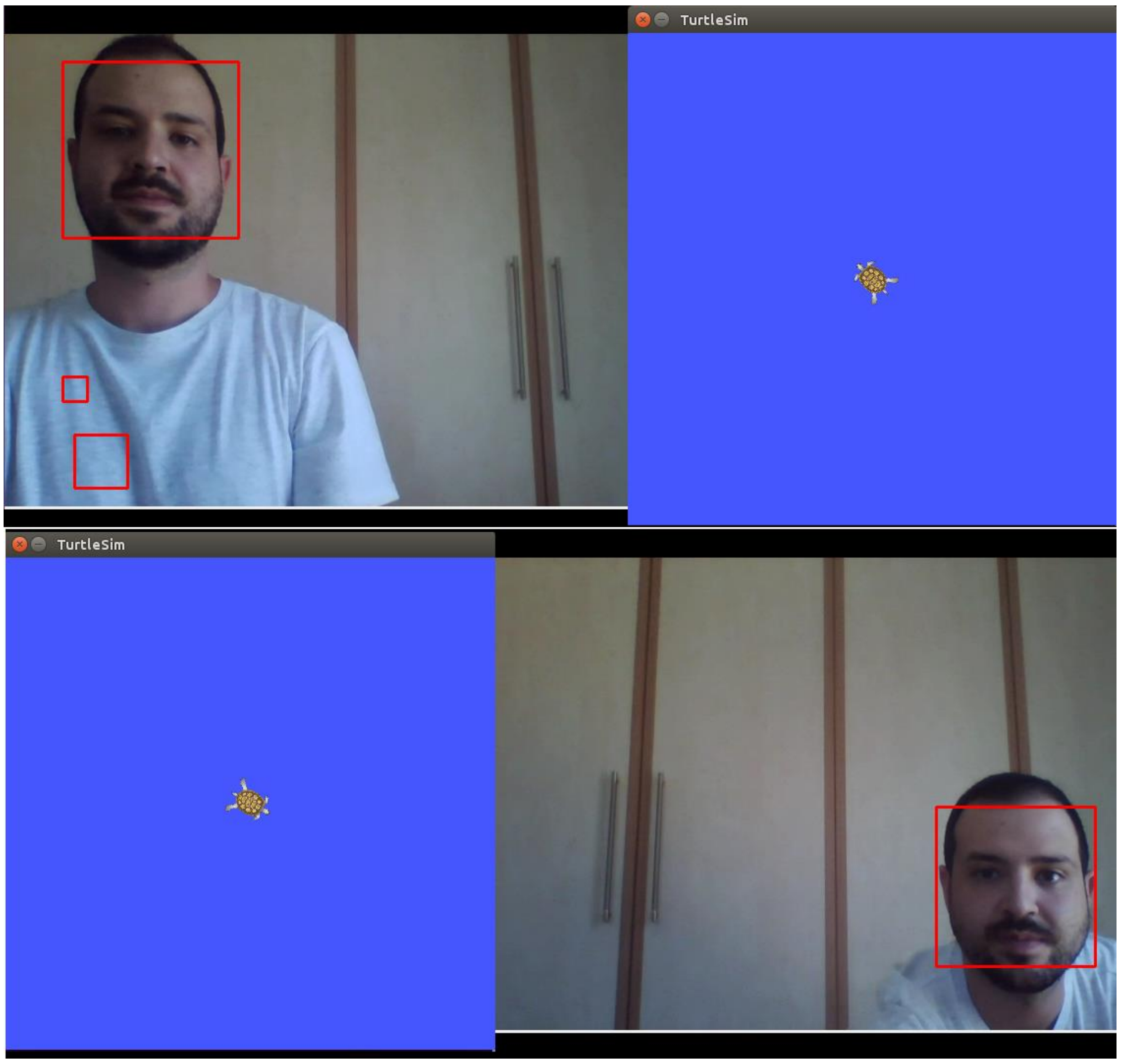


Figure 30: Proof of concept: ROS autopiloting

6.5 Metrics

Figure 31 depicts the processing delay from the time of the snapshot, until the time of the transmission of the calculated goal. The mean values is the red line on the graph, which is approximately 683 ms, while the standard deviation is about 196 ms. These measurements result due to changes on the image, the number of detected objects per image, and because no optimization techniques have been applied.

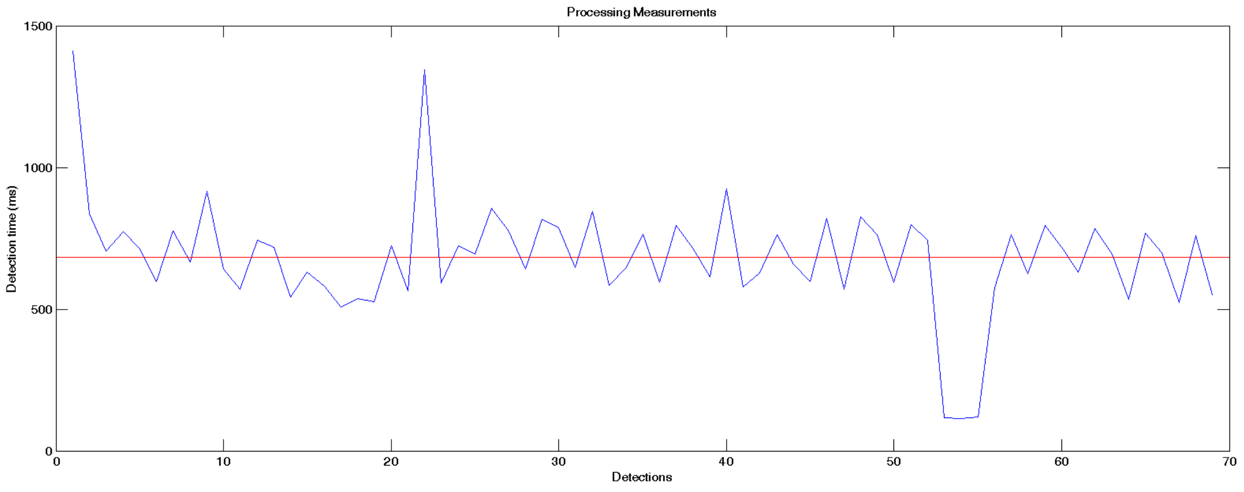


Figure 31: Processing delay

7. CONCLUSIONS AND FUTURE WORK

In this implementation we explained and described how to combine all of these heterogeneous components (WebRTC – OpenCV – ROS), in order to compose a web-based infrastructure for autopiloting ROS-based vehicles upon a specific use case. We developed a horizontal infrastructure that consists of a modular architecture, which provides the necessary components for machine-to-machine communication, uses state-of-the-art technologies (i.e. WebRTC), allows a developer to freely implement her own logic vertically (i.e. her own OpenCV functions, autopiloting handling, use-cases etc), and provides IoT with a solution that can be easily exploited in numerous ways.

The next step is to extend this infrastructure to a native project; since all of these components are written in C++, we can properly combine them in order to gain the maximum speed and efficiency. Furthermore, since many robotic applications pair some data along with the video, we should find a way to interleave the video with the Data Channel [68]. Regarding the metrics, we should test our system to real network conditions with packet losses, delays, and congestions, in order to measure the impact on the system (i.e. bandwidth, synchronization etc.). Finally, we should test the scalability of our system, in order to enable some advanced uses cases like a fleet management, car-to-car communications, and others.

ABBREVIATIONS – ACRONYMS

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
AVC	Advanced Video Coding
CDN	Content Delivery Network
DNS	Domain Name System
DTLS	Datagram transport Layer Security
FPS	Frames per Second
HEVC	High Efficient Video Coding
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
LLVM	Low Level Virtual Machine
M2M	Machine-to-Machine
MiTM	Man in The Middle
MTU	Maximum Transfer Unit
NAT	Network Address Translation
OpenCV	Open Computer Vision
OS	Operating System
P2P	Peer-to-Peer
REST	Representational State Transfer
ROS	Robot Operating System
RTC	Real Time Communications
RTCP	Real-time Transport Control Protocol
RTP	Real-time Transport Protocol
SAVP	Secure Audio Video Profile
SAVPF	Secure Audio Video Profile with Feedback
SCTP	Stream Control Transmission Protocol
SDP	Session Description Protocol

SIMD	Single Instruction, Multiple Data
SIP	Session Initiation Protocol
SRTCP	Secure Real-time Transport Control Protocol
SRTP	Secure Real-time Transport Protocol
SSL	Secure Sockets Layer
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TF	Transmission Frame
TLS	Transport Layer Security
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
UI	User Interface
URDF	Unified Robot Description Format
URL	Uniform Resource Locator
VoIP	Voice over IP
W3C	World Wide Web Consortium
WebRTC	Web Real Time Communnications
XHR	XMLHttpRequest
XML	eXtensible Markup Language

REFERENCES

[1] C. Jennings, D. Burnett, A. Bergkvist, T. Brandstetter, A. Narayanan, and B. Aboba, “WebRTC 1.0: Real-time Communication Between Browsers”, W3C Working Draft.
[2] “Internet of Things”, IEEE.
[3] “Open Computer Vision Library” [Online]. Available: http://opencv.org/
[4] J. Postel, “User Datagram Protocol”, RFC768, 1980.
[5] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, “RTP: A Transport Protocol for Real-Time Applications”, RFC3550, 2003.
[6] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman, “The Secure Real-time Transport Protocol (SRTP)”, RFC3711, 2004.
[7] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, “SIP: Session Initiation Protocol”, RFC3261, 2002.
[8] M. Handley, V. Jacobson, C. Perkins, “SDP: Session Description Protocol”, RFC4566, 2006.
[9] P. Srisuresh, and K. Egevang, “Traditional IP Network Address Translator (Traditional NAT)”, RFC1631, 2001.
[10] T. Kivinen, B. Swander, A. Huttunen, and V. Volpe, “Negotiation of NAT-Traversal in the IKE”, RFC3947, 2005.
[11] P. Srisuresh, B. Ford, and D. Kegel, “State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs)”, RFC5128, 2008.
[12] J. Rosenberg, “Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols”, RFC5245, 2010.
[13] adapter.js: https://github.com/webrtc/adapter.
[14] International Telecommunications Union, "Pulse code modulation (PCM) of voice frequencies", ITU-T Recommendation G.711, November 1988.
[15] International Telecommunications Union, "Low-complexity coding at 24 and 32 kbit/s for hands-free operation in systems with low frame loss", ITU-T Recommendation G.722.1, 2005.
[16] GIPS / Google, "iSAC reference implementation". Available at https://chromium.googlesource.com/external/webrtc/+/master/webrtc/modules/audio_coding/codecs/isac/
[17] Andersen, S., Duric, A., Astrom, H., Hagen, R., Kleijn, W., and J. Linden, "Internet Low Bit Rate Codec (iLBC)", RFC 3951, December 2004.
[18] JM. Valin, K. Vos, and T. Terriberry, “Definition of the Opus Audio Codec”, RC6716, September 2012.
[19] “Advanced Video Coding for Generic Audiovisual Services”, ITU-T Rec. H.264 and ISO/IEC 14496-10 (MPEG-4 AVC), Version 22: Feb. 2014.
[20] J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, and Y. Xu, “VP8 Data Format and Decoding Guide”, RFC6386, November 2011.
[21] A. Grange, P. de Rivaz, J. Hunt, “VP9 Bitstream & Decoding Process Specification”, Google 2012.
[22] “The WebM Project: WebM Container Guidelines”, April 2014.
[23] ISO/IEC, "Information technology -- High efficiency coding and media delivery in heterogeneous environments -- Part 2: High efficiency video coding", ISO/IEC 23008-2, 2013.
[24] “AV1 Bitstream & Decoding Process Specification”: https://aomedia.googlesource.com/av1-spec/
[25] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, “Session Traversal Utilities for NAT (STUN)”, RFC5389, October 2008.
[26] R. Mahy, P. Matthews, J. Rosenberg, “Traversal Using Relays around NAT (TURN): Relay extensions to Session Traversal Utilities for NAT (STUN)”, RFC5766, April 2010.
[27] ISO, "OSI Routeing Framework", ISO/TR 9575, 1989.
[28] Information Science Institute - University of Southern California, “Transmission Control Protocol”, RFC793, September 1981.
[29] J. Rosenberg, A. Keranen, B. B. Lowekamp, and A. B. Roach, “TCP Candidates with Interactive Connectivity Establishment (ICE)”, RFC6544, March 2012.
[30] E. Rescorla, and N. Modadugu, “Datagram Transport Layer Security Version 1.2”, RFC6347, January 2012.
[31] T. Dierks, and E Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2”, RFC5246, August 2008.
[32] A. Freier, P. Karlton, and P. Kocher, “The Secure Sockets Layer (SSL) Protocol Version 3.0”, RFC6101, August 2011.
[33] J. Ott, and E. Carrara, “Extended Secure RP profile for Real-time Transport Control Protocol (RTCP)-based Feedback (RTP/SAVPF)”, RFC5124, February 2008.
[34] D. McGrew, and E. Rescorla, “Datagram transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)”, RFC5764, May 2010.

[35] R. Jesup, S. Loreto, and M. Tuexen, “WebRTC Data Channels draft-ietf-rtcweb-data-channel-13.txt”, Internet-Draft, January 2015.
[36] R. Stewart, “Stream Control Transmission Protocol”, RFC4960, September 2007.
[37] D. C. Burnett, A. Bergkvist, C. Jennings, A. Narayanan, and B. Aboda, “Media Capture and Streams”, W3C Candidate Recommendation, 19 May 2016.
[38] A. Bergkvist, D. C. Burnett, C. Jennings, and A, Narayanan, “WebRTC 1.0: Real-time Communication Between Browsers”, W3C Working Draft, 10 September 2013.
[39] Alan B. Johnston, and Daniel C. Burnett, “WebRTC: APIs and Protocols of the HTML5 Real-Time Web”, Third Ed., Digital Codex LLC, March 2014, p60.
[40] Alan B. Johnston, and Daniel C. Burnett, “WebRTC: APIs and Protocols of the HTML5 Real-Time Web”, Third Ed., Digital Codex LLC, March 2014, p5.
[41] Alan B. Johnston, and Daniel C. Burnett, “WebRTC: APIs and Protocols of the HTML5 Real-Time Web”, Third Ed., Digital Codex LLC, March 2014, p6.
[42] H. Alvestrand, “Overview: Real Time Protocols for Browser-based Applications”, Internet Draft, March 2017.
[43] Ludwig, S., Beda, J., Saint-Andre, P., McQueen, R., Egan, S., and J. Hildebrand, "Jingle", XSF XEP 0166, June 2007.
[44] Alan B. Johnston, and Daniel C. Burnett, “WebRTC: APIs and Protocols of the HTML5 Real-Time Web”, Third Ed., Digital Codex LLC, March 2014, p63.
[45] van Kesteren, A., Ed., "XMLHttpRequest", W3C Candidate Recommendation CR-XMLHttpRequest-20100803, August 2010.
[46] Fielding, R., "Architectural Styles and the Design of Network-based Software Architectures", 2000.
[47] Alan B. Johnston, and Daniel C. Burnett, “WebRTC: APIs and Protocols of the HTML5 Real-Time Web”, Third Ed., Digital Codex LLC, March 2014, p65.
[48] Alan B. Johnston, and Daniel C. Burnett, “WebRTC: APIs and Protocols of the HTML5 Real-Time Web”, Third Ed., Digital Codex LLC, March 2014, p64.
[49] I. Fette, and A. Melnikov, “The WebSocket Protocol”, RFC6455, December 2011.
[50] Alan B. Johnston, and Daniel C. Burnett, “WebRTC: APIs and Protocols of the HTML5 Real-Time Web”, Third Ed., Digital Codex LLC, March 2014, p67.
[51] Alan B. Johnston, and Daniel C. Burnett, “WebRTC: APIs and Protocols of the HTML5 Real-Time Web”, Third Ed., Digital Codex LLC, March 2014, p70.
[52] OpenCV.org. Available at:http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html
[53] P. Viola, M. Jones, “Rapid object detection using a boosted cascade of simple features,” in Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001.
[54] Yoav Freund and Robert E. Schapire, “A decision-theoretic generalization of on-line learning and an application to boosting”, Journal of Computer and System Sciences, 55(1):119–139, August 1997.
[55] Emscripten. Available at: http://kripken.github.io/emscripten-site
[56] LLVM. Available at: https://llvm.org/
[57] asm.js. Available at: http://asmjs.org/
[58] OpenCV.js. Available at: http://www.ics.uci.edu/~sysarch/projects/OpenCV.html
[59] ROS. Available at: http://www.ros.org/
[60] Aaron Martinez, and Enrique Fernandez, “Learning ROS for Robotics Programming”, Third Ed., Packt Publishing Ltd., September 2013, p25.
[61] Aaron Martinez, and Enrique Fernandez, “Learning ROS for Robotics Programming”, Third Ed., Packt Publishing Ltd., September 2013, p26.
[62] Aaron Martinez, and Enrique Fernandez, “Learning ROS for Robotics Programming”, Third Ed., Packt Publishing Ltd., September 2013, p32-33.
[63] Aaron Martinez, and Enrique Fernandez, “Learning ROS for Robotics Programming”, Third Ed., Packt Publishing Ltd., September 2013, p39.
[64] TurtleBot. http://www.turtlebot.com/.
[65] Rosbridge Suite. Available at: http://wiki.ros.org/rosbridge_suite.
[66] Roslibjs. Available at: http://wiki.ros.org/roslibjs
[67] Robot Web Tools. http://robotwebtools.org/
[68] M. Petit-Huguenin, and G. Salgueiro, “Multiplexing Scheme Updates for Secure Real-time Transport Protocol (SRTP) Extension for Datagram Transport Layer Security (DTLS)”, RFC7983, September 2016.