EAI Endorsed Transactions
on Mobile Communications and Applications
Research Article EAI.EU

Ubi-Interact: A modular approach to connecting systems

Sandro Weber[1,*], Marian Ludwig[1], Gudrun Klinker[1]

[1]Technische Universität München, Institut für Informatik - I16, Boltzmannstraße 3, 85748 Garching bei München, Germany

Abstract

Ubi-Interact is a framework for interactive applications combining individual systems and devices distributed over a network. Specification and implementation of such applications should be modular, extendable and reusable. Performance, re-usability of once established capabilities and easy integration of devices are main objectives. It relies on extendable common data formats. Edge computing capabilities allow to analyze and transform this data. These computing modules can also manage an arbitrary number of devices with similar capabilities interchangeably. The framework is ready to be used. Nodes for CSharp, Javascript, C++ and Java exist and features are continuously expanded. Ubi-Interact lets users build real-time systems with modular components that can separate system behaviour from base API calls, leaving the user free to explore and maintain a combination of software and hardware each running in their native environment.

Received on 20 April 2021; accepted on 04 July 2021; published on 14 July 2021

Keywords: Human-Computer Interaction, Distributed Applications, Reactive Systems, Edge Computing, Mixed Reality, Internet of Things, Machine Learning, Robotics

doi:10.4108/eai.14-7-2021.170291

1. Introduction

In recent years systems for virtual and mixed reality applications have undoubtedly become mainstream and affordable. Yet there's still constant research and development with new form factors, capabilities of hardware and ways of interaction. This brings with it a constantly changing environment of new standards, APIs, SDKs and interfaces. Meanwhile companies like Neuralink aim to elevate the link between humans and machines to a whole different level [1].

At the same time the number of small individual systems and devices in the fields of IoT and Industry 4.0 are also constantly rising with a myriad of sensors & functionality and the expressed goal of inter-connecting them and performing analysis and behaviour based on their data. This is reflected in the concept of digital twins [2] [3]. To keep interactions with these systems on an acceptable timescale, the necessity of edge computing for interactive applications has been emphasized by Fraga-Lamas et al. [4]. The issue of scalability has been expressed by Navab [5].

The question is how can we effectively explore the space of possibilities, not only of humans interaction with these systems but also the individual system components interacting with each other? How would we navigate a world filled with such devices? How do we interact with them? What are the systematic requirements? Norouzi et al. [6] give an overview of what future possibilities might arise from merging the fields of augmented reality, the internet of things and intelligent virtual agents. Ubi-Interact is designed to explore these possibilities.

2. Motivation

The initial motivation to build a system like Ubi-Interact was the desire to use and explore devices like sEMG & EEG devices, smart devices, virtual & augmented reality hardware, IoT devices, etc. for HCI in 3D mixed reality environments and applications. The individual device capabilities and data should be analysed and processed in combination with each other - possibly using machine learning approaches

*Corresponding author. Email: webers@in.tum.de

- to investigate their combined potential for novel interaction methods and investigate if new HCI patterns could emerge from this combination.

Another push in the direction of a networking system solution came from the discontinuation of the networking API UNet for the popular game-engine Unity3D. The widespread use of Unity3D to develop mixed-reality applications made it necessary to find alternatives. A system that would - at the same time - make it possible to extend Unity3D applications with capabilities not native to the Unity ecosystem (e.g. modern machine learning solutions) was intriguing.

Yet another interest is the development of motion controls for full-body virtual re-embodiment avatars [7]. In previous work these have been implemented within the Neurorobotics Platform [8] and therefore closely intertwined with ROS [9] & Gazebo [10] and the provided physics engines. For a more general purpose solution it would be interesting to see if control strategies could be implemented that rely on only the most basic body tracking data as input and direct calls with force vectors/tensors to any physics engine as output. The development of systems like [11] by Matthes et al. and [12] by Tieck et al. could also be application areas.

So the basic idea was to have an open and extendable networking ecosystem with enough performance to support HCI tasks. This opened several questions:

- What development platform should be chosen?

  For the wide range of devices as well as the library integration, we need to connect with a variety of development languages, environments & SDKs. For networking, the system will obviously be heavy in asynchronous I/O tasks, so we need an environment that is strong in this regard.

- How can we easily integrate with other established solutions? How can we keep established solutions running over time, re-use them and build upon them in the future?

  It should be easy to integrate with other existing frameworks and libraries. We want to support and integrate with web technologies and IoT interfaces as well as popular libraries like TensorFlow [13] for machine learning, OpenCV [14] for computer vision, ROS [9] for robotics, etc.

  It should be easy for people to incorporate existing solutions into their environment, build upon and extend them. They should not have to worry about or be restricted in their choice of which environment to develop in. New and upcoming developments in hardware as well as implemented solutions in the form of available libraries should be quick and easy to adapt to.

- How can we benefit more from focused or small-scale individual research work and integrate them with each other? How can we produce systems evolving beyond a prototype stage or at least keep solutions surviving at that stage? How can we compare results with other findings and bodies of work?

  Especially in the environments of university and research, fluctuation of people and topics is higher. Often enough the work of a temporary project provides good solutions and results but is then kept on hold and hard to pick up on by the next person after a certain amount of time. This hampers progress.

  Or the results of individual works is hard to compare against each other. Updated and changing environments bring the additional overhead of making prior solutions run again in the new environments if not regularly touched upon and kept up-to-date constantly.

  Reproducibility of results and comparisons between them are core to research. We'd like to improve scalability in this regard.

- How can we quickly prototype systems? How can we produce modular device interfaces so we can re-use and re-combine them?

3. Goals

From the questions following the initial motivation, the goals behind *Ubi-Interact* can be summed up as follows:

- Easy development of distributed network applications

  We want to allow easy development of distributed network applications. We want to be able to combine and connect different systems and environments, exchange data and explore emergent possibilities for HCI when using this data in combination with each other.

- Make it easy to incorporate new devices into the system

  We want to be able to easily adapt to new hardware in the future. New devices should be easily connected to the existing system, extending the capabilities. Furthermore, the system should allow devices to be integrated "loosely" - connecting and disconnecting while the system is running without the system breaking or making assumptions on what devices will be used at runtime.

- Make I/O devices interchange-able, decouple system behaviour/interactions from specific devices used

Where possible, devices should be interchangeable based on the similarity of data they provide. If for example two devices both provide IMU data, we want to be able to use any of them for the context of an application where accelerometer + gyroscope data is part of the interaction. Ideally, it should be possible to combine several individual parts and devices to emulate or rapidly prototype the concept of a fully developed device. One could imagine hot-glueing together parts like an IMU, a touchscreen and a camera to emulate the possibilities of a smartphone.

- Make developed code reusable between different applications and development environments

  In addition to the point above, we want to minimize the cost of re-implementing working code when switching to different devices or developing new applications. The possibility to write "blackbox" functionality should be an integral part of the system architecture. Within these blackboxes, it should be possible to rely on existing libraries as much as possible.

- Good performance in the context of HCI

  Of course it is essential to have the system perform on a timescale that makes interactive tasks viable. The best possible performance should always be a main point of consideration.

To clarify, the goal of Ubi-Interact is not to provide connectivity to different devices on a hardware or driver level. Its purpose is to integrate the devices into a bigger system, given there's access to the devices' capabilities, services and/or data interfaces. If a device is required to have for example a bluetooth connection and certain drivers installed that might be available only for certain platforms, then the Ubi-Interact client needs to run on the platform with the bluetooth connection and drivers. The work on the side of Ubi-Interact ends once there is a Ubi-Interact node for a certain platform or operating system. Any platform with such a node can then expose the devices capabilities to the distributed system. If on the other hand for example an IoT device already provides access through open network interfaces like RESTful API, a central modular process can be established as a communication and status manager for this device - again exposing its capabilities to the wider system.

4. Related Work

4.1. Messaging

One of the core questions for a network framework is how to specify message formats and (de-)serialize messages. There are multiple solutions to this problem, each with their own advantages and disadvantages. Formats like JSON are self-descriptive and do not rely on prior knowledge of schemas while libraries like Google's Protocol Buffers [15], Google's Flatbuffers [16] or Cap'n Proto [17] aim to improve performance, relying on schema files compiled for target languages. Work by Popić et al. [18] and Biswal et al. [19] made performance comparisons for Flatbuffers, Protocol Buffers, JSON and BSON while Flatbuffers is also actively used by mobile game engines - a use-case that is definitely intensive in interaction.

For its messaging layer, Ubi-Interact relies on the proven capabilities of Protocol Buffers, possibly switching to libraries like Flatbuffers or Cap'n Proto for yet increased computation performance in the future.

4.2. ROS

The Robot Operating System [9] [20] has become the de-facto standard for robotics applications. One of its core strengths is the topic-based publish-subscribe distribution of messages between ROS nodes and the possibility to integrate modular packages executed within nodes. It is mainly focused on C++ and Python but offers bindings to several other languages.

While ROS already provides a lot of the functionality targeted here, ultimately the goal was to have a more lightweight system with minimal assumptions about what environment it is used in and what other components might become part of the system in the future or how they can connect.

4.3. Web Services, Cloud Computing and Edge Computing

The concept of web services & cloud computing offers an unprecedented flexibility and availability of computing power. Especially in the case of interactive applications it might not be viable to rely on remote services though. As remarked by Fraga-Lamas et al. [4] the concept of edge computing is more viable. Additionally, due to concerns over security and profiling of personal data users might want to keep their interaction data like voice commands, gestures or even EEG locally and not send it into the cloud.

4.4. Ubitrack

The development of a ubiquitous tracking framework by Pustka et al. [21] and Huber et al. [22] has been another inspiration for the development of Ubi-Interact. As the focus of this work is different from Ubitrack it does not seek to replace it but rather leave the possibility to work in conjunction with it in the future. Many of its strategies to handle time-critical computation and merging of tracking data can inspire the future development of Ubi-Interact.

4.5. Virtual-Reality Peripheral Network (VRPN)

Taylor et al. developed the Virtual-Reality Peripheral Network [23] as a server-client architecture providing device-independent and network-transparent interfaces to VR hardware.

To achieve device-independency, it features layered abstraction interfaces exposing common capabilities of device types like buttons, tracking and others. Within VRPN, these device data interfaces can also be layered on top of each other to transform base data into higher-level functionality and abstractions, a concept that is reflected by *Processing Modules* within Ubi-Interact (see chapter 5.5).

In comparison, Ubi-Interact tries to achieve the same abstract handling of devices by providing common data formats and the concept of separating devices into components or treating separate components as one abstract device. The VRPN feature of storing and replaying device data could be handled generically with a *Processing Module* too as they have access to the filesystem. The goal of Ubi-Interact is a more general integration of arbitrary system instead of focussing on VR periphery alone. A goal is to support multiple data stream / processing paradigms for *Processing Modules* (e.g. synchronized, event-triggered, asynchronous, ...) to give the user the freedom to configure with the execution strategy most fitting to the context of the data flow.

4.6. Node-RED

Node-RED is a tool aimed at connection hardware, APIs and online services. It offers low-code programming for event-driven, distributed applications and is based on NodeJS. Several works have successfully shown development of IoT-based applications with the help of Node-RED [24][25][26].

Ubi-Interact shares the core philosophy of connecting different processes and wiring together interactivity through modular components. Node-RED calls these components Nodes, in Ubi-Interact they are named *Processing Modules*. Node-RED also features graphical editors for the data flow and an online library for easy access and public sharing - two mechanisms which should further inspire the work on Ubi-Interact.

As a distinctive feature, while the NodeJS module ecosystem utilized in Node-RED offers widespread support and bindings for a lot of systems already Ubi-Interact aims to provide a collection of nodes for any language and environment that users choose to target.

5. System Architecture and Features

The basic structure of Ubi-Interact is a client-server architecture. Client nodes are the I/O data producing and consuming endpoints and therefore established on the I/O devices' systems, communicating data via the publish-subscribe pattern on identifiable topic URIs. The server or master node acts as a centralized data buffer for the device topics, data distributor and optional data analyzer & processor.

The server has been implemented in NodeJS for its architectural strength in asynchronous I/O handling and the available support of targeted external libraries. Client nodes are of course implemented for the target system. So far there are clients for C# / Unity3D, C++, Java as well as Web & NodeJS.

5.1. Message Schemas and Base Dependency

Before diving deeper into the description of the architecture, it is essential to talk about the basic dependency for all parts of the system - the message schemas and data format specifications for all networked messages. They reflect almost any part of the system worth communicating like server configuration & services, client configuration, backend processes, device data etc. and can consequently be used as the common language to describe & understand any part of the system. Later chapters will go into detail - suffice to say for now that these messages are used to reflect the largest portion of internal state and configuration of system components as well as network data and therefore already serve as a big part of the specification API.

As for the technology used for the message format specifications, in the first iteration the choice fell on Google Protocol Buffers. While formats like JSON are universally used and have the benefit of being human-readable in all stages and self-describing without prior knowledge of the format / schema, performance considerations led to choosing a framework that reduces bandwidth while also providing performant (de-)serialization in a broad variety of languages so most client systems can be supported. As described in [15], protobuf still offers abstract human-readable schema files which can referred to for understanding what is being communicated and can be extended with new data formats.

5.2. Communication Patterns

Ubi-Interact uses two communication patterns between server and clients. A synchronous connection following the request-reply pattern handles service-like exchanges, e.g. basic initialization and registration of clients, starting/stopping sessions, etc. For the implementation of the service channel a Ubi-Interact client can rely on either HTTP(S) requests or a ZeroMQ's request-reply socket connection.

For continuous device I/O data with higher frequency a bi-directional asynchronous connection is better suited. Since clients should be able to join and leave

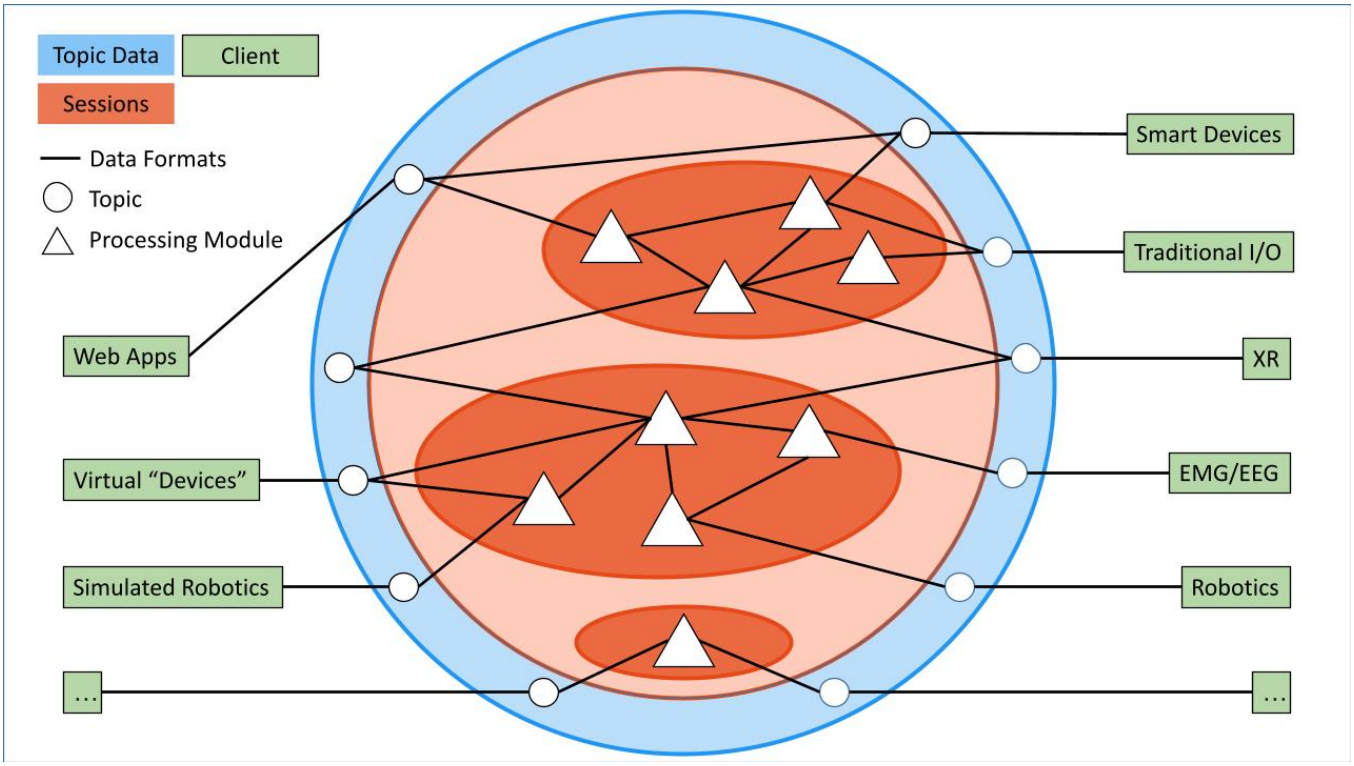

Figure 1. Ubi-Interact overview, topic data and sessions. An overview of the system with the outer layer (blue) representing network communication and the inner circle (red) representing modular processing. Boxes on the outer left and right indicate clients or their representative devices.

the system at any point in time and it is unknown what set of topic data the client might be interested in, the publish/subscribe pattern is the ideal solution for topic data communication. For this clients can connect to the server via a websocket or ZeroMQ's router-dealer sockets. ZeroMQ's publish-subscribe sockets are not used since we do not want to rely on all clients being able to integrate ZeroMQ as a library and instead offer at least one alternative for connecting to the server, i.e. web technologies. Consequently, we must manage topics and subscriptions ourselves to be able to connect constellations where one client is connected via ZeroMQ and another one uses web technologies.

Network data formats and network communication channels are deliberately kept separate and agnostic to each other in order to potentially support additional ways of connecting nodes in the future without having to change the rest of the system.

5.3. Topic data and Records

Any message sent over the asynchronous pub-sub connection is of the message format "TopicData". Such a message can contain one or more "TopicDataRecords" each containing a timestamp and one of the elemental data structures described in a message schema. These can be primitives like int, bool, string, bytes etc. up to higher-level descriptions of vectors, matrices, images, button events and so on.

The bundling of TopicDataRecords into one message helps saving bandwidth and overloading the connection with fragmented messages when there's a lot of individual components to be published and/or high traffic of subscriptions.

Apart from actual device data in records, the asynchronous channel can also be used to transfer error messages. This can be useful to notify clients of server and connection errors.

5.4. Clients, Devices and Components

When connecting to the Ubi-Interact server, the first step is to create a client node. It manages the network connections and is automatically assigned an ID by the server when registering. Optionally a human-readable name, description and tags can be provided. A client represents a physical machine or a process running on that machine.

From there a client can proceed to register *Devices*, which are in turn split into *Components*. A *Device* in Ubi-Interact terms does not necessarily mean a physical device like a smartphone with its individual components like display, IMUs, buttons, ... but could just as well be a virtual car in a game controlled

by a player or a simulated robot with components of simulated sensors and actuators. So the concept of a *Device* again groups, identifies and holds general status of a collection of *Components*. The *Components* in turn are the entities describing the individual capabilities of a *Device*, i.e. what data is to be produced or consumed on a specific topic with a specific message format. *Devices* and *Components* too can have tags and descriptions attached.

To clarify, the concepts of *Devices* and *Components* are not enforced on the user of Ubi-Interact. A client can just start publishing and subscribing to any topic. They may help with structuring and inferring status & relationships within the distribution of the system though.

So what about devices like IoT gadgets where it's not possible to run a client node and not necessary to do so because they already expose an interface for their data communication? The following chapter 5.5 will clarify how to integrate these cases into the system.

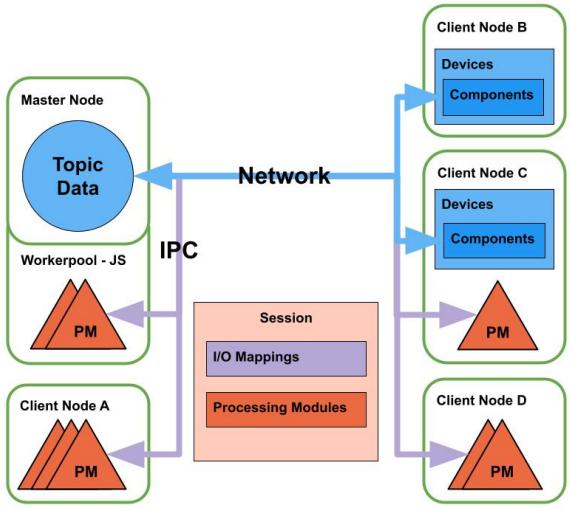

Figure 2. Node and data organization. Nodes can be connected via network or Inter-Process-Communication (IPC). Sessions provide runtime environments for Processing Modules, mapping between topics and a module's input/output specifications.

5.5. Sessions and Processing Modules

Apart from handling the network essentials like client connections and buffering topic data, one of the more distinguishing features of Ubi-Interact is the possibility to establish *Processing Modules* (PMs) within nodes that can manipulate topic data and provide system behaviour in a decoupled, I/O device agnostic, modular, reusable and shareable fashion.

In the typical scenario, processing modules are instantiated inside dedicated processing nodes that offer the correct environment (dependencies) for the module to run in. They can then provide the module's functionality to other nodes. If necessary, multiple instances of the same module can be run in parallel, each wired to its own subset of topic data.

That way, a web application can for example rely on functionality written in C++ to solve a high-performance task it would neither have the resources nor correct environment to perform. Or a readily available, tried and proven solution that is implemented in an environment different from the rest of our application can be encapsulated and run in its own node. Or two modules designed to perform the same task can be easily swapped and compared in their performance.

It may also be the case that we have a device, application or whole infrastructure that already comes with its own interfaces and API. If Ubi-Interact needs to integrate with it because a) other parts of the Ubi-Interact system can not talk to it directly or b) both systems need to have a tight integration for coordination of execution etc., a *Processing Module* can serve as a communication endpoint to translate between both worlds. Similarly, if a device is not able to run a Ubi-Interact node but has a ready-to-use network API, a *Processing Module* can serve as a communicator and status manager.

Of course any data analyzing & processing can also be done within one of the client nodes before publishing or after receiving data while Ubi-Interact merely serves as a networking and data communication framework. It is up to the user to decide at what stage computation is best performed and where resources are available. As stated in chapter 3, the idea is to support the concept of black-boxed behaviour modules that

- can handle clients and relationships/interactions between them dynamically with clients (dis-)connecting at runtime.
- work independently of specific devices used and how many are used.
- can be re-used and integrated with other modules based on well-defined I/O formats, allowing users to easily connect, share and exchange their implemented features.
- can interface with other libraries like OpenCV, TensorFlow, etc.
- reduce the need to re-implement parts of the application when replacing devices.

These modules are conceptually named *Processing Modules*. Figure 2 illustrates the structure of *Processing Modules* and *Sessions*. *Processing Modules* are grouped in *Sessions* which reflect the runtime environment

of the combined set of *Processing Modules* necessary for a certain distributed application. *Sessions* can be started and stopped to control the execution of all *Processing Modules* encapsulated. Communication between running *Sessions* is possible through the use of dedicated topics (Figure 1 shows this with the middle left topic).

A *Processing Module* defines its capabilities based on desired inputs and expected outputs following the same message formats and data structures used for topic data. It follows a typical set of life cycle methods like initialization and processing that are called at the appropriate point in time. To have better control over how a *Processing Module* does its work, three processing modes are defined. Note that a *Session* may potentially contain *Processing Modules* with differing processing modes.

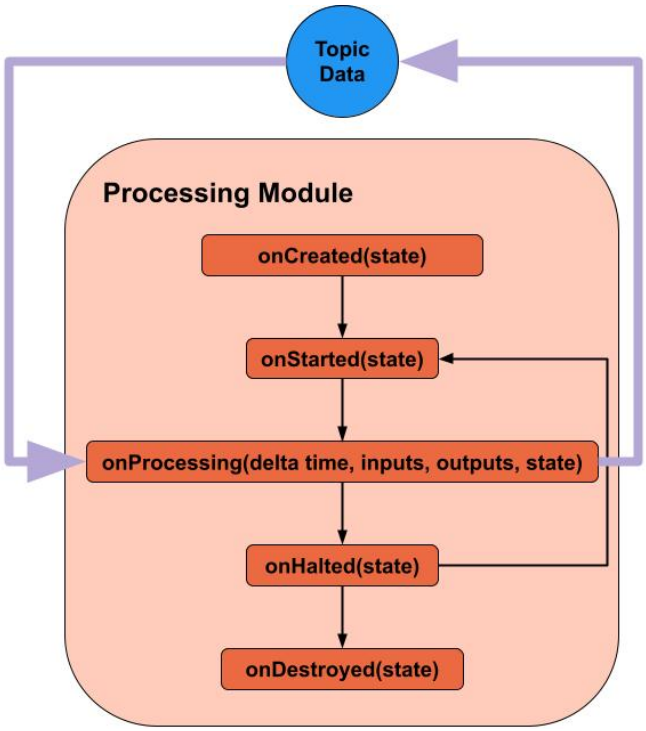

Figure 3. Processing Module life cycle overview. External dependencies are either accessed via the internal state (if defined globally for all PMs) or through the code of the life cycle callbacks directly.

Processing based on set frequency. In this mode, after starting a *Processing Module* it will execute its onProcessing() life cycle callback in set time intervals. Necessary input data will be pulled during execution and output data will be written after processing has finished.

As such is the most asynchronous and decoupled mode of execution. A *Processing Module* will execute on its own terms, only reacting to new data whenever it is running its cycle. The target execution frequency will be kept unless the server system hits the limit of its processing power. The delta time parameter for the call to onProcessing() will indicate time since the last processing iteration and therefore typically reflect this execution frequency.

For many cases, this mode is sufficient and is therefore chosen by default if no specifications are given.

Processing triggered by new input. If a more immediate execution is desired, this mode allows to run processing whenever one or more of the input topics are updated. It can be configured to start a processing iteration as soon as one of the input topics updates or make it necessary for all input topics to be updated before running again. To avoid an overwhelming influx of execution triggers from topics with high update frequency, a minimum delay between executions can be defined. Delta time will again reflect the time that passed since the last processing call.

If the goal is to chain multiple *Processing Modules* together in a pipeline where each consecutive module relies on the output of the previous one(s), this mode is probably a good choice to couple the modules more closely and avoid delays of updates.

Processing in lockstep. Certainly the most involved option is the lockstep mode. It covers the use cases where individual executions need to be kept in synchronization and are supposed to operate on the same snapshot in time of topic data.

The *Processing Module* will not determine its own execution. Instead a running *Session* on the master node will identify all *Processing Modules* inside it using lockstep mode and synchronize the execution of them. *Processing Modules* with different processing modes inside the same *Session* will follow their execution loop as described above.

During a lockstep processing cycle, first all topic data records necessary as input for the involved *Processing Modules* will be gathered at the same point in time. Then the *Session* will send out processing requests to each *Processing Module* containing the necessary input records as well as the time passed since the last execution or respectively a time-step to be advanced during processing. This time delta will of course be the same for all *Processing Modules*. The *Session* will wait for all processing requests to be responded to, indicating that all *Processing Modules* have finished their iteration. These responses will also contain produced output topic records which the *Session* can then write back into the general topic data buffer after the processing cycle has been completed.

To define relations to the outside world, a *Processing Module* relies on the same topic data message formats

used by device components which guarantees seamless integration with the rest of the system without additional steps of data conversion - except for what might be necessary while using external libraries. The inputs and outputs of a *Processing Module* are specified by an internal name serving as a getter / setter variable and the data format. That way a *Processing Module* can be specified without consideration for any topic conventions, i.e code can be written with internal naming conventions. The *Session* instantiating the *Processing Modules* can then define I/O mappings for each, linking topics to internal names verified to be legal based on fitting data formats.

Internally, a *Processing Module* consists of a state and life cycle callbacks to define its behaviour. Currently there is a callback after creation for initialization purposes and a callback for processing defining the runtime behaviour. All callback functions allow arbitrary code execution to guarantee maximum freedom. The life cycle specifications are continuously extended as necessary. Arguments for the creation callback are the internal state while the process method is called with references to inputs, outputs and state. External libraries can be loaded at start to be added to a library and are then made available to all *Processing Modules* by accessing the "modules" objects of the internal state. Alternatively external libraries can be referenced directly from code.

In its current state Ubi-Interact allows to run *Processing Modules* in Javascript (either inside a dedicated node or as a workerpool execution) and CSharp (dedicated node only). Work has already started on providing integration with nodes in other languages, which will be elaborated on in chapter 8.2. While not having optimal performance for *Processing Modules* in every case, Javascript so far can already cover a big portion of the initially targeted external libraries including ROS [9], TensorFlow [13] and OpenCV [14].

From a more theoretical point of view, *Processing Modules* can be seen as an effort to meet the demands of reactive systems [27] [28]. Another interpretation is an agent that can observe and react to its environment through inputs and outputs while relying on an internal state to keep continuity in its behaviour. Additional efforts definitely have to be made for time-critical use-cases though.

5.6. Topic Multiplexing and De-Multiplexing

As soon as applications involve several users and/or devices that interact with the system in a similar fashion - for example multiple users with their personal smartphones - it quickly becomes desirable for *Processing Modules* to be able to take all of these inputs with equal formats in, process them together and possibly in relation to each other and then react with output relating to the individual users/devices. That way we can write one *Processing Module* handling an arbitrary number of devices dynamically. This leads to the following requirements:

- Devices should be considered/disregarded while connecting/disconnecting at runtime.
- The inputs and outputs that should be bundled all have the same data format in order to guarantee what kind of data is being accessed or written inside the *Processing Module*.
- Within the code of the *Processing Module* we need to be able to identify which user/device the data originates from or respectively where to send output.
- While writing code for the *Processing Module* we want to handle references to users/devices in an abstract and flexible fashion without prior assumptions beyond input and output formats.

These requirements have been realized within topic muxers/demuxers.

Topic muxers are specified by a common message format for all input topics, a regular expression used to identify topics of interest as well as another regular expression used to extract identity patterns from the individual topics. To illustrate with an example, we might want to handle a number of smartphones all accessing the system through a common web interface. This web interface will create topics for each smartphone in the form of "/<client-ID>/smartphone/touch_position", ".../imu-accelerometer", etc. with the client or device ID included as a way to prevent conflicting topic URIs. A *Processing Module* interested in the proximity of touch positions could then utilize a topic muxer to identify all topics shaped like the first example while also extracting the client ID (commonly UUIDv4 is used in Ubi-Interact). During processing of the *Processing Module*, the muxer then provides a list of all relevant topic data with elements consisting of (topic, data, type, [optional] identity).

Analogously, a topic demuxer is specified by a data format and an output topic format shaped like a formatted printf() statement. To produce output, the demuxer must be provided again with a list of elements consisting of the data and a related list of parameters used to fill the placeholders in the output topic format. If the smartphone interface subscribes to a topic "/<client-ID>/smartphone/vibrate" then we could for example use the previously extracted client ID to make individual smartphones vibrate based on their touch proximity by providing sets of (vibration pattern, client ID) as output to the demuxer.

This is a first proof-of-concept. In the future it is possible to extend topic muxer/demuxer to automatically deduce client IDs from arbitrary topics or gather topics based on device component tags or message formats.

6. Web Frontend

Ubi-Interact provides a web frontend that comes with a lot of usage examples and test applications as well as administration and monitoring tools. In terms of connection and communication with the server, the frontend behaves just like any client. Run the web-server on the same machine as your Ubi-Interact server to get access to configuration and visualization tools as well as live examples and demos.

6.1. System Performance Evaluation Tools

The web frontend includes a short list of performance evaluation tools. As it's web-based, you can use these tools from any client with a browser. It includes a short round-trip-time measurement as well as session execution speed measurements.

6.2. System Overview and Configuration

An *Processing Module* editor allows creating, editing and deleting *Processing Modules* through a web interface. *Processing Module* files are saved in JSON format on the server and can also be edited directly.

As auto-discovery services are not implemented yet, a view of local IP adresses together with corresponding QR codes allows quick access to the web frontend from e.g. smartphones.

A topic inspector shows all services topics and current data topics with live updated values. A client inspector lists all client nodes, their devices and components in a similar way.

To quickly test system behaviour a web interface can be used to easily publish any topic data records.

7. Usage Examples

7.1. Base Example

One of the first applications implemented as part of the web frontend was a simple webpage that reflects the mouse cursor position. The current 2D position is sent to the server, run through a minimal *Processing Module* and then sent back to the client. At the position received back from the server, a small red square indicator is shown. Besides being a basic usage code example illustrating how to publish data and subscribe to it, this also allows the user to get a subjective and intuitive impression and "feeling" for the latency and update rate of the system by doing live mouse movements. For a detailed and commented code example please refer to the web frontend repository.

On one hand this serves as a minimal code example to learn how to connect to the back-end and use the publish & subscribe mechanisms for data. On the other hand it is meant as a personal latency evaluation tool. We're all very much used to how a 2D cursor is supposed to feel as we're using the standard mouse as input device daily. The frontend example then allows to hide either the original cursor or the red square to let the user make better comparisons. Especially hiding the original cursor gives a good indication and "feeling" of how well the system is reacting to live input as the red square becomes the only indicator for the user's mouse movement input.

As an additional exercise, the 2D vector doing the round-trip is run through an *Processing Module* on the server-side that can mirror the position in X and Y based on a boolean input flag. This interaction is specified and transmitted by the client-side on start to illustrate how this purely client-side specification of the whole runtime context can be achieved in principle.

7.2. 2D Object Detection for Image Streams

The web frontend provides an interface for devices with a camera that continuously sends camera images to the server and uses an *Processing Module* in combination with TensorFlow [13] and the CocoSSD model for 2D object detection [29]. The *Processing Module* takes images as input and produces a 2D object list output with image coordinates and classification results. The camera web interface can then use the model classification results to overlay the original image. In theory, any application producing images (even artificial ones) can then use this *Processing Module* to make use of the same functionality, only having to worry about the message formats for images and 2D objects.

7.3. Topic Multiplexing

A demo implementing the example of Chapter 5.6 is available as part of the web frontend. Under "applications - examples" you can open the "Smart Device Gatherer" page e.g. on your desktop and then connect additional smart devices using the web frontend's category "interfaces - smart device".

7.4. Ubi-Party Game

To put the system to the test in terms of usability, performance and scalability a multiplayer gaming application was developed where users could play using their personal smartphones. This application called "Ubi-Party" was designed to consist of multiple mini-games akin to popular video game series.

To test live performance, especially games that require quick reaction times were found to be ideal test cases as players in a competitive multiplayer environment would quickly and loudly criticise any delay or game-hindering latency recognized in the input.

The list of mini-games so far features a short racing game and a "king of the hill" style game where players would push each other off a slippery platform. Another game where players are tasked to quickly find real-life objects and take pictures of them to earn points is under development. It is designed to test communication of larger chunks of data (images) and using backend processes to analyze them (2D image recognition).

7.5. Serious Games, AR Escape Room, Superhuman Sports

Several applications developed by Plecher et al. also already integrated Ubi-Interact.

The first one is Oppidum - a Serious-AR-Game set in and teaching about Celtic culture [30]. It uses Ubi-Interact to implement the multiplayer mode.

Xanthippos [31] combined a tablet, a greek statue and a projector to interactively color the statue. Users would paint the 3D model of the statue on the tablet and the projector would then overlay the texture onto the real-life model.

A third application with the topic of designing an AR Escape Room [32] also used Ubi-Interact to define interactivity between teams and objects.

Another project [33] by Eichhorn et al. is exploring the possibilities of using a tracked and controllable drone in a spherical cage as a playing ball in the field of Superhuman Sports. Ubi-Interact has been successfully utilized to communicate game state and player actions here.

7.6. Physical Embodiment in VR

In Chapter 2 the idea of virtual re-embodiment in a physically simulated environment has been brought up. Such a system would try to mirror a user's real-life movements onto a virtual body that is part of a physics simulation, thus allowing the user physical interactions with the virtual environment.

In the most general terms such a system would always rely on some sort of user body tracking as a first input step and commanding a physics engine to apply forces to a virtual body as a last output step. Body tracking data could for example come from of a camera based pose/skeleton estimation or typical VR hardware's head and hand tracking positions possibly complemented by waist and feet tracking targets. On the other end might lie one of several physics engines embedded in e.g. a game engine or a robotics simulator - PhysX, (Py)Bullet, ODE, OpenSim or Simbody to name some.

In between several other steps need to happen. Depending on how detailed the data of the body tracking system is, a body pose estimation step using e.g. inverse kinematics (IK) needs to happen. Following this, from the current pose of the virtual body and the (estimated) target user pose we can determine the motion necessary for the virtual body to reach the target pose. From that pose error one can derive forces to be applied each time step of simulation.

One of the expressed goals of Ubi-Interact is to be able to abstract away solutions from the base input and output modalities, offering components that can be reused in different scenarios and keeping the time investment to switch platforms to a minimum. As an example, if one were to switch physics engines - or the whole simulation engine for that matter - one would like to only re-implement the base calls for the step of receiving target forces and applying these to a certain body in the new engine. The other steps could in theory stay the same. Depending on how much of the other steps are implemented inside the prior engine and what other libraries were available and used to accomplish this, one might be facing some additional work though.

Is it possible to create such a VR embodiment system using Ubi-Interact, taking only the base data (pose indicators, forces) of the components regarded as interchangeable (tracking, physics), communicating these directly with the Ubi-Interact system and the rest of the computation happening inside modules?

To test this, a scene was implemented using Unity3D. Fig.4 illustrates the three general stages of 1) getting out tracking data, 2) combining tracking data and the physical avatar's current pose to estimate forces and 3) a physical avatar reporting its current pose and being able to receive commands for appling forces.

To speed up development, human animations are played in a loop in stage 1 instead of relying on real body tracking data. Key transformations of head, body, hands and feet are extracted from that animation as IK target positions. Controlling animation speed also gives a good idea how fast the system can react to changes overall. The six IK targets are then published to Ubi-Interact.

Stage 2 consists of a pose estimator converting the IK targets of stage 1 to a full body pose. From there a physics estimator compares the pose estimator's result to the current pose of the physical avatar (stage 3) it is subscribed to, then publishes target linear and angular velocities for the individual body parts that intend to move the them towards their target pose and keep them stable while not moving.

In stage 3 we have the physical avatar that is continuously publishing current poses for each body part as well as being subscribed to the target velocities of stage 2.

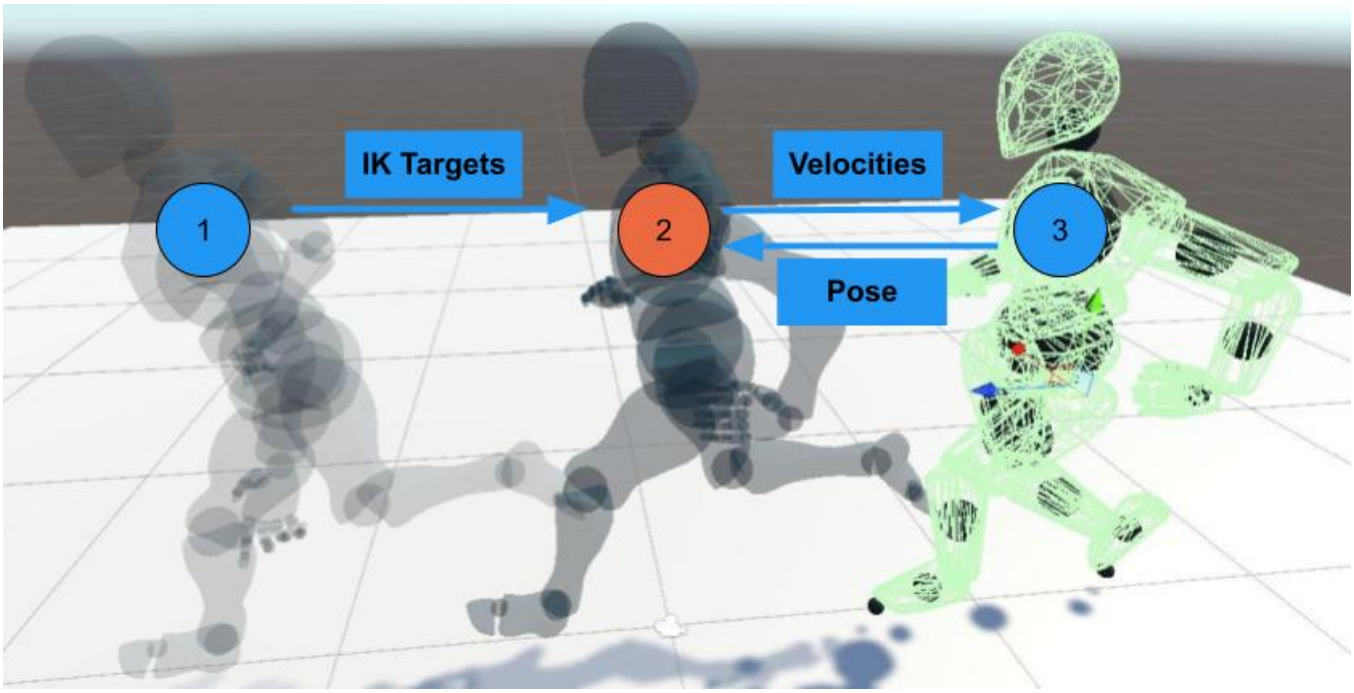

Figure 4. VR embodiment stages overview. The 3 stages for the physical embodiment system. 1: user tracking, 2: pose and velocity computation, 3: physical avatar being moved

Overall the communicated data consisted of the six IK targets, the avatar's current poses for each of the 46 bones in it's body structure and the target velocities for each bone - each represented by a 6-DoF transformation or two 3-D vectors respectively. All topics are published with a frequency of 30Hz.

For calculating the necessary velocities in stage 2, an idealistic approach using the velocity necessary to move to a target pose within the next time step of the physics simulation loop is chosen. When calculated and executed from within the physics loop, it adds the exact forces necessary to produce perfect stable results in every time step, taking into account the body part's current velocity. When introducing latency as will happen with network communication through Ubi-Interact, it is expected that the physical avatar will jitter or simple explode as forces overshooting the target quickly add up to infinity. This is indeed the result when naively applying the same approach in the physics estimator of stage 2. With some regulatory additions to the estimator however the error resulting from latency can be counteracted. In this case a simple linear scaling is enough to produce stable results. In more realistic robotics and teleoperation scenarios where infinitely fast acceleration is impossible and momentum is applied to joints, typical solutions like PD controllers can be applied in an analogous manner [34].

To allow direct comparisons with the integrated one-system solution, all parts in this example remained implemented in Unity3D. This should however exemplify that Ubi-Interact can successfully be used to implement solutions in a modular fashion and combine them in real-time systems. We could replace stage 1 with input from a WebXR application, adapt an existing solution producing more accurate results for humanoid inverse kinematics in stage 2 or use a robotics simulator in stage 3 next, only having to implement the callbacks for topic communication and basic API calls for retrieving poses and applying forces.

8. Next Steps

8.1. Introspection and Debugging

The development of a graph visualization & editing tool has started that will serve as a *Session* editor. The plan is to visualize (live) data flow and status of *Sessions* and *Processing Modules* and provide GUI tools to set up and edit sessions by e.g. dragging connections between topics and *Processing Module* input/output.

8.2. Extending node features and languages

Of course Ubi-Interact lives with the amount of nodes available to users. Extending this portfolio as well as bringing all the newest features to existing nodes will be a constant effort. Once a node or a *Processing Module* for that node has been established, everyone can make

use of it. Our hope is that a community of users will be able to benefit from each others' work here, and the first adoptions point towards succeeding here. In order to provide the functionality of *Processing Modules* in different languages, one of the first concerns has been performance. It is mandatory to utilize the full scope of a programming language, so *Processing Modules* need to run in separate processes that communicate with the current master node through sockets.

This however means constant (de-)serialization of protobuf messages when communicating with clients and again when input/output to/from *Processing Modules* needs to be transferred. This (un-)packing of messages could mean a significant performance overhead. Libraries like Flatbuffers or Cap'n Proto are very similar to Protocol Buffers in their use of schema files compiled into different languages. Their architecture as a flat message buffer with defined access offsets however gets rid of the (de-)serialization step. First tests within Javascript with multiple processing modules that each read and write a random subset of topic data have shown Flatbuffers to perform 15% ± 3% faster when transmitting smaller messages consisting of primitive types. Unfortunately it was also discovered that Flatbuffers takes a serious performance hit under Javascript when longer byte buffers, e.g. images, are involved. This issue seems to have been fixed already for other languages. It would be desirable to provide the same performance for all languages considered as continuous steps are taken in this direction.

8.3. Security

So far the application scenarios of Ubi-Interact are rather risk-free. This is why it is allowed for clients to provide full specifications of *Processing Modules* by stringifying callback functions and sending them over to the server to be executed. This arbitrary code execution obviously represents a big security risk in other scenarios and might need to be restricted for environments where security is a concern.

Another point is the amount of biometric data involved in user interactions with the system when we think about for example involving EMG/EEG. Close attention will be paid to keep user data sets anonymous when performing studies. It may be prudent to also provide secured communication channels.

8.4. Scalability and Utilities

Until now considerations like publishing frequency of topic data or execution frequency of *Processing Modules* has been an effort on the user's side. To improve usability and scalability, the aim is to support the user with sensible defaults, functionality and regulatory checks in order to prevent or warn about certain parts of the system taking over too many resources.

8.5. Time synchronization

The more time-critical the execution of *Processing Modules* becomes, the more necessary it becomes to provide time synchronization mechanisms between server and clients. Strategies are investigated and will probably be adapted from Ubitrack and/or VRPN. Furthermore, future development may provide channels to couple Ubitrack and Ubi-Interact more closely to benefit from a combined effort.

9. Conclusions

With Ubi-Interact we hope to provide an open community system that helps to prototype and develop scalable and performant applications in the realm of mixed reality and IoT. An emphasis lies on freedom and usability for the developer. First user feedback indicates good acceptance and satisfaction. The widespread public exploration of the full possibilities of *Processing Modules* remains an interesting topic for the future.

All efforts made here are already publicly available at https://github.com/SandroWeber/ubi-interact and base dependencies for Javascript can also be found in the form of NPM packages.

Acknowledgement. This project/research has received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreement No. 945539 (Human Brain Project SGA3).

References

[1] Pisarchik, A.N., Maksimenko, V.A. and Hramov, A.E. (2019) From novel technology to novel applications: Comment on "an integrated brain-machine interface platform with thousands of channels" by elon musk and neuralink. *Journal of medical Internet research* **21**(10): e16356.

[2] Boschert, S. and Rosen, R. (2016) Digital twin—the simulation aspect. In *Mechatronic futures* (Springer), 59–74.

[3] Tao, F., Cheng, J., Qi, Q., Zhang, M., Zhang, H. and Sui, F. (2018) Digital twin-driven product design, manufacturing and service with big data. *The International Journal of Advanced Manufacturing Technology* **94**(9-12): 3563–3576.

[4] Fraga-Lamas, P., Fernández-Caramés, T.M., Blanco-Novoa, Ó. and Vilar-Montesinos, M.A. (2018) A review on industrial augmented reality systems for the industry 4.0 shipyard. *Ieee Access* **6**: 13358–13375.

[5] Navab, N. (2004) Developing killer apps for industrial augmented reality. *IEEE Computer Graphics and applications* **24**(3): 16–20.

[6] Norouzi, N., Bruder, G., Belna, B., Mutter, S., Turgut, D. and Welch, G. (2019) A systematic review of the convergence of augmented reality,

intelligent virtual agents, and the internet of things. *Artificial intelligence in IoT* : 1–24.

[7] Weber, S. and Klinker, G. (2019) Vr re-embodiment in the neurorobotics platform. *Mensch und Computer 2019-Workshopband* .

[8] Falotico, E., Vannucci, L., Ambrosano, A., Albanese, U., Ulbrich, S., Vasquez Tieck, J.C., Hinkel, G. *et al.* (2017) Connecting artificial brains to robots in a comprehensive simulation framework: the neurorobotics platform. *Frontiers in neurorobotics* **11**: 2.

[9] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R. *et al.* (2009) Ros: an open-source robot operating system. In *ICRA workshop on open source software* (Kobe, Japan), **3**: 5.

[10] Koenig, N. and Howard, A. (2004) Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)* (IEEE), **3**: 2149–2154.

[11] Matthes, C., Weissker, T., Angelidis, E., Kulik, A., Beck, S., Kunert, A., Frolov, A. *et al.* (2019) The collaborative virtual reality neurorobotics lab. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)* (IEEE): 1671–1674.

[12] Tieck, J.C.V., Weber, S., Stewart, T.C., Kaiser, J., Roennau, A. and Dillmann, R. (2020) A spiking network classifies human semg signals and triggers finger reflexes on a robotic hand. *Robotics and Autonomous Systems* : 103566.

[13] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M. *et al.* (2016) Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*: 265–283.

[14] Bradski, G. and Kaehler, A. (2008) *Learning OpenCV: Computer vision with the OpenCV library* (" O'Reilly Media, Inc.").

[15] Google, Protocol buffers on github, https://github.com/protocolbuffers/protobuf. Accessed: 2021-04-19.

[16] Google, Flatbuffers on github.io, https://google.github.io/flatbuffers/. Accessed: 2021-04-19.

[17] Varda, K. (2015) Cap'n proto, 2015. *URL: https://capnproto.org* .

[18] Popić, S., Pezer, D., Mrazovac, B. and Teslić, N. (2016) Performance evaluation of using protocol buffers in the internet of things communication. In *2016 International Conference on Smart Systems and Technologies (SST)* (IEEE): 261–265.

[19] Biswal, A.K. and AL MALLAH, O. (2019) Analytical assessment of binary data serialization techniques in iot context (evaluating protocol buffers, flat buffers, message pack, and bson for sensor nodes) .

[20] Quigley, M., Gerkey, B. and Smart, W.D. (2015) *Programming Robots with ROS: a practical introduction to the Robot Operating System* (" O'Reilly Media, Inc.").

[21] Pustka, D., Huber, M., Waechter, C., Echtler, F., Keitler, P., Newman, J., Schmalstieg, D. *et al.* (2010) Automatic configuration of pervasive sensor networks for augmented reality. *IEEE Pervasive Computing* **10**(3): 68–79.

[22] Huber, M., Pustka, D., Keitler, P., Echtler, F. and Klinker, G. (2007) A system architecture for ubiquitous tracking environments. In *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality* (IEEE): 211–214.

[23] Taylor, R.M., Hudson, T.C., Seeger, A., Weber, H., Juliano, J. and Helser, A.T. (2001) Vrpn: a device-independent, network-transparent vr peripheral system. In *Proceedings of the ACM symposium on Virtual reality software and technology*: 55–61.

[24] Blackstock, M. and Lea, R. (2014) Toward a distributed data flow platform for the web of things (distributed node-red). In *Proceedings of the 5th International Workshop on Web of Things*: 34–39.

[25] Lekić, M. and Gardašević, G. (2018) Iot sensor integration to node-red platform. In *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)* (IEEE): 1–5.

[26] Rajalakshmi, A. and Shahnasser, H. (2017) Internet of things using node-red and alexa. In *2017 17th International Symposium on Communications and Information Technologies (ISCIT)* (IEEE): 1–4.

[27] Harel, D. and Pnueli, A. (1985) On the development of reactive systems. In *Logics and models of concurrent systems* (Springer), 477–498.

[28] Bonér, J., Farley, D., Kuhn, R., and Thompson, M., The reactive manifesto, https://www.reactivemanifesto.org/. Accessed: 2021-04-19.

[29] Tensorflow coco ssd model, https://github.com/tensorflow/tfjs-models. Accessed: 2021-04-19.

[30] Plecher, D.A., Eichhorn, C., Köhler, A. and Klinker, G. (2019) Oppidum-a serious-ar-game about celtic life and history. In *International Conference on Games and Learning Alliance* (Springer): 550–559.

[31] Plecher, D., Bloch, A., Kaiser, T. and Klinker, G. (2020) Projective Augmented Reality in a Museum: Development and Evaluation of an Interactive Application. In *ICAT-EGVE*. Accepted.

[32] Plecher, D., Ludl, M. and Klinker, G. (2020) Designing an AR-Escape-Room with Competitive and Cooperative Mode. In Weyers, B., Lürig, C. and Zielasko, D. [eds.] *GI VR / AR Workshop* (Gesellschaft für Informatik e.V.).

[33] Eichhorn, C., Jadid, A., Plecher, D.A., Weber, S., Klinker, G. and Itoh, Y. (2020) Catching the drone – a tangible augmented reality game in superhuman sports. In *2020 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)* (IEEE): 24–29.

[34] Nuno, E., Ortega, R., Barabanov, N. and Basañez, L. (2008) A globally stable pd controller for bilateral teleoperators. *IEEE Transactions on Robotics* **24**(3): 753–758.