Towards an Open Testbed for Tactile Cyber Physical Systems

Kurian Polachan*, Prabhakar T. V.†, Chandramani Singh‡, and Fernando A. Kuipers§
* † ‡ Indian Institute of Science, Bangalore, India
§ Delft University of Technology, Delft, the Netherlands
Email: {*kurian, †tvprabs, ‡chandra}@iisc.ac.in, §F.A.Kuipers@tudelft.nl

***Abstract*—In this paper, we consider Tactile Cyber Physical Systems (TCPS), which differ from typical CPS in that haptic sensory feedback is included. In particular, we design and implement a TCPS testbed, called *TCPSbed*, using well-defined components and interfaces glued together using APIs. In addition to real connections, our testbed supports the interconnection of components over an NS3-emulated network. The testbed also supports the integration of applications that mimic the behaviour of real-world embedded objects. Since controlling latency and ensuring stability is crucial for TCPS applications, the testbed includes tools for fine-grained characterization of latency and control performance. Finally, through proof-of-concept experiments with our testbed, we demonstrate *TCPSbed*'s capabilities to facilitate TCPS research and development.**

***Index Terms*—Tactile Internet, Haptic communications, Step response, Network latency.**

I. Introduction

The Internet was designed primarily for exchange of data, but over the years it has evolved into a medium for transferring real-time audio and video as well. Recently, Fettweis and Alamouti [1] have provided a new vision for the Internet, referred to as the *Tactile Internet*, in which the Internet is also used as a medium to transport haptic information. Exchanging haptic information enables real-time steering and control of remote objects. This brings forth a new type of Cyber Physical Systems (CPS), which we refer to as Tactile Cyber Physical Systems (TCPS). In a TCPS, an operator, called *tactile master*, manoeuvres a robot, called *tactile slave*, remotely through the *Tactile Internet*. An example of a TCPS application is telesurgery. Here a surgeon steers a robot over the Internet to perform surgery. The haptic feedback helps the operator to get a feel for the physical environment and objects with which the robot is interacting. For instance, it helps the surgeon to assess and communicate the right amount of force to the robot for handling surgical objects of different build and texture.

Humans noticing disputes between various sensory modes may experience discomfort. In TCPS, this may happen due to the delayed and asynchronous appearance of sensory feedback of different modalities like haptic, video, audio, etc., at the operator side, which is referred to as cybersickness. To avoid cybersickness in TCPS, the round-trip latency should not exceed a few *milliseconds* [2]. This requirement places an upper limit on the network round-trip latency, and this value is often reported to be 1ms, although it could somewhat be relaxed through machine learning techniques that anticipate certain movements. Current ambitions for 5G communications stipulate similar network latency objectives. However, even if we assume the availability of an ultra-low latency network in the foreseeable future, for realizing TCPS applications, significant research is required in designing and building the hardware, algorithms, protocols, and communication components that accompany it [3]. Being able to measure and isolate the latency of different components, instead of solely end-to-end as is mostly done, is imperative. This calls for building a testbed, as presented in this paper, where TCPS applications and associated components can be realized and characterized.

Our main contributions are as follows: In Section II, we present the design of our testbed, which, to the best of our knowledge, is the first TCPS testbed. In Section III, we propose methods to characterize component-level latencies and step-response behaviour using our testbed. In Section IV, we describe the implementation of the testbed and, through experiments, demonstrate its capabilities. We list related work in Section V, discuss potential avenues for future work in Section VI, and conclude in Section VII. Some of our Tactile Internet related software will be made open source at [4].

II. Testbed Design

A TCPS consists of three sub-systems: the operator end including the tactile master, the remote end including the tactile slave, and the *Tactile Internet* connecting the two end-systems. The design of our TCPS testbed, as presented in this section, is guided by the following feature objectives:

- Ability to measure and isolate the latency of individual components from a live TCPS application.
- Allowing for the use of both real and emulated networks.
- Ease of use through a user-centric design, including a graphical front-end to configure the testbed parameters and scripts to activate the testbed components.

A. Component-Level Design

The components of our testbed are shown in Figure 3 and are described below.

The *tactile master* represents the human operator while the *tactile slave* represents the robot being controlled.

We use embedded computing boards at both the tactile master and the tactile slave sides. The board on the master side, termed *ms embsys*, houses sensors and algorithms to capture the kinematic motions of the master. It also contains

978-1-5386-7902-9/19/$31.00 ©2019 IEEE

audio/video display devices and haptic actuators to convey haptic feedback information to the master. The board on the slave side, termed *ss embsys*, houses actuators and algorithms to drive the robot and the audio, video, and tactile sensors that capture its manoeuvres. *ms embsys* and and *ss embsys* could also be emulated in software; the emulated versions are called *ms embsys-app* and *ss embsys-app*, respectively.

ms com is the master-side communication component that connects *ms embsys* to the Internet. Similarly, *ss com* connects *ss embsys* to the Internet on the slave side.

We also use a computer, referred to as *srv*, to cater to different TCPS applications with varying degrees of complexity in sensing and actuation algorithms. *srv* provides insight into the actual computing power required by the embedded computers of TCPS applications.

The testbed supports two modes of network operation: *deployment mode* and *emulation mode*. In deployment mode, *ms com, ss com,* and *srv* are connected through a real network. In emulation mode, these components are connected through an emulated network realized in a desktop PC, called *emu*, running a network emulator.

B. Design of Data-paths

In our work, we support four data-flow types, namely kinematic, haptic, audio, and video flows. Figure 1 details the data-path in emulation mode. The design of the deployment mode is identical except that it does not have the emulator component *emu*. Here, *ms com*, *ss com*, and *srv* have one IP interface each. Each may have its own IP address, or they may share a common IP address if realized within a single PC. All data-flows pass through the IP interface.

The emulator (*emu*) has two IP interfaces which are used to join *ms com* and *srv* components through the emulated network. In the current version of the testbed, the interfaces between the *ms com*, *emu*, *srv*, and *ss com* are UDP based. Other transport protocols can also be supported by modifying the *receive()* and *send()* APIs discussed in Section II-C. The interfaces between *ms embsys* and *ms com* and between *ss com* and *ss embsys* depend on the type of used embedded device and could be USB, serial, PCI, or UDP based.

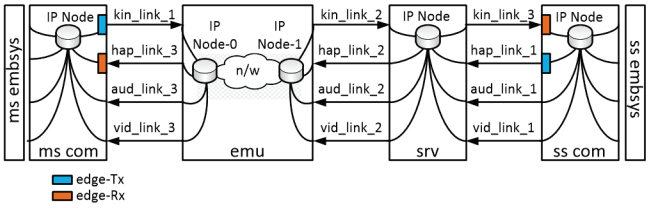

Figure 1. Data-path in emulation mode. Here, *emu* is placed between *ms com* and *srv*. Alternatively, one can also place the *emu* between *srv* and *ss com*.

The data-path also consists of *edge-Tx* and *edge-Rx* modules. They are located at the TCPS edge components *ms com* and *ss com*. The *edge-Tx* is used to duplicate the transmitted data packets for reliability. The duplicate packet count and the interval are parameterized. The *edge-Rx* is used at the receiving side to drop the redundant packets (identified via the packet sequence number).

C. API Framework Design

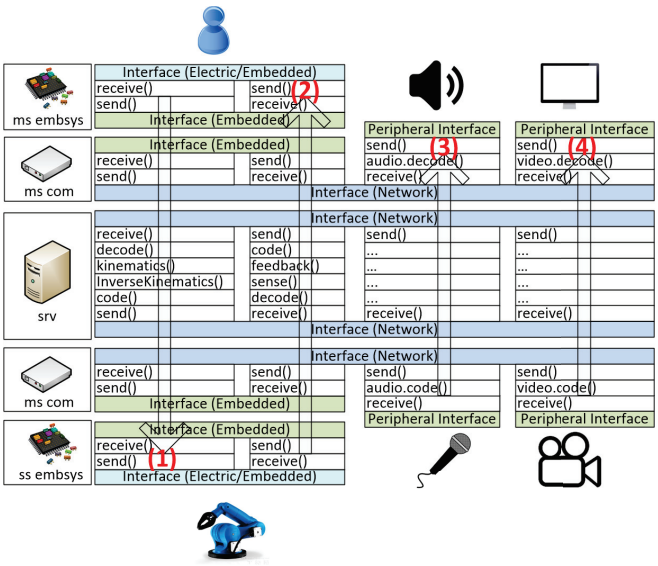

Figure 2. Set of generic APIs to realize the TCPS forward and backward data-flow paths. (1) represents the forward path for the kinematic data. (2), (3), and (4) represent the backward paths for the haptic, audio, and video data, respectively.

To enable the quick realization of TCPS applications, we have designed a set of generic application programming interfaces (APIs), following a layered framework. Figure 2 details these APIs and their framework as explained below.

1) receive() and send(): The *receive()* reads data from connected physical sensors or adjacent TCPS components. At the *ms embsys* end, the API reads data from the kinematic sensors attached to the operator. And at the *ss embsys* end, the API reads data from position sensors and haptic sensors mounted on the robotic arm. In both cases, depending on the type of sensors in use, the API should have the necessary analog and digital capabilities to read data. At the *ms com* and *ss com*, the API reads data from the embedded boards *ms embsys* and *ss embsys*, respectively, or from *embsys-apps*. The data in this interface are coded using embedded protocols, e.g., I2C, SPI, UART, USB, etc. in case of *ms embsys and ss embsys*, or using network transport protocols, e.g., UDP, TCP, RDP, RUDP, etc. in case of *embsys-apps*. At the *srv*, the *receive()* reads data from both *ms com* and *ss com*, running network transport protocols. The *send()* transmits data to connected physical actuators or adjacent TCPS components. Like the *receive()*, its design depends on the TCPS component on which it resides, the embedded/network protocol in use, and the type of sensor/actuator in use.

2) code() and decode(): Message formats and coding methods vary for different testbed components. The *code()* codes the data in the prescribed message format and coding method. It is then sent to the interface through the *send()*. The *decode()*

does the opposite, it extracts the data from the output of the *receive()*. Specific to audio and video flows are the *audio.code()*, *audio.decode()*, *video.code()*, and *video.decode()* APIs.

3) kinematics() and inverse_kinematics(): These APIs run the kinematics and inverse-kinematics algorithms used to capture the operator's pose and then to recreate this pose with the robot, respectively. For computing the operator's pose, the *kinematic()* uses data from the kinematic sensors. The *inverse-kinematic()* processes the computed pose to generate commands to drive the robotic actuators.

4) sense() and feedback(): *The sense()* does calibration, scaling, and filtering of the haptic data received from the haptic sensors mounted on the robotic arm. The *feedback()* uses the processed haptic data to drive haptic actuators at the operator end.

D. Network Emulation

We use NS3 [5] to emulate realistic network topologies in *emu*. Across this emulated network, we create emulation channels to communicate with the real network devices *ms com* and *srv* (see Figure 1). The emulator allows setting link type, latency, bandwidth, packet error rate, error model, etc., to emulate diverse realistic scenarios. We set these using a GUI as explained in Section II-F.

E. Tactile Slave Simulation

Due to cost and resource constraints, researchers may be interested to simulate the *tactile slave* component. This motivated us to support the integration of the Virtual-Robot Experimentation Platform, V-REP, with our testbed [6]. For this, we have developed a *ss embsys-app* to interface the V-REP scenes with *ss com*.

F. User Interface Design

To use the testbed, a user has to first set the configuration through a GUI. This includes (a) selecting the mode of testbed operation (emulation or deployment), (b) assigning resources (e.g., PCs) and IP addresses to *ms com*, *emu*, *srv*, and *ss com*, (c) defining the interface type, addresses and port numbers to realize the data-path, (d) defining the embedded device type (real or emulated) in use, and (e) configuring the network emulator settings such as delay, bandwidth, error model, and error rate. All the configuration details are stored in a spreadsheet for ease of use. The spreadsheet is located inside the testbed run-directory, which is copied to different components, see Section IV-A.

III. Latency and Step Response Characterization

In this section, we explain how we can use the testbed to perform latency and step response characterization.

A. Latency Characterization

Since a TCPS may consist of multiple distributed components, it is important to get an insight into the performance of each component. For example, the reaction time of a remote robotic arm to a user command has to be measured to budget for end-to-end latencies over a production network. This requires a tool similar to ping with enhanced support for TCPS. Such a tool should measure processing delay and actuation delay in addition to the network latencies.

Our testbed can be used as an in-situ latency test ecosystem with the help of an external test PC and ping points. Ping points are locations in the forward/backward data paths where we need to examine the latency values. Each TCPS component can have multiple ping points. A test PC, see Figure 3, issues ping commands targeting a particular ping point and then measures the time it takes for the echo to arrive. This process is repeated multiple times and for multiple ping points. Ping commands are UDP-based wrappers around data commands: they contain data which the *ms embsys* and *ss embsys* components expect. This means that the system is fully functional while the test is in progress and, hence, the latency numbers we obtain are representative of the TCPS under test.

In Figure 3, the test PC inserts the ping commands in the kinematic data path through *ms com*, bypassing *ms embsys* which does not have an IP address. Thus the resultant ping trajectory of the kinematic data path does not cover the latency between *ms embsys* and *ms com*. As a solution, we propose using the ping trajectory of the haptic data path to measure and isolate the forward kinematic latencies involved between *ms embsys* and *ms com*.

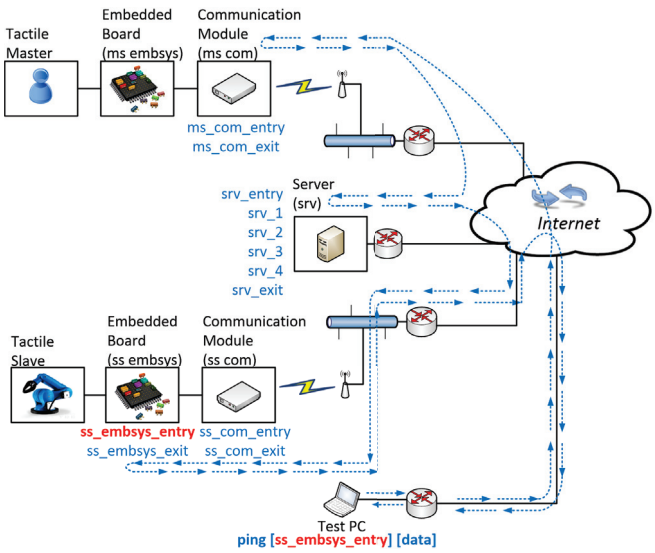

Figure 3. In-situ latency test. Direction of ping commands targeting ping point *ss_embsys_entry* is marked. Only ping points in the forward kinematic data path are shown.

B. Step Response Characterization

We can model a TCPS application using control system elements. In this model, the tactile master acts as the controller and the tactile slave as the device being controlled. The haptic data comprises the feedback to this controller and the kinematic data forms the controller output. With this control model in place, it is possible to evaluate the control performance of the TCPS application using classical control-theoretic meth-

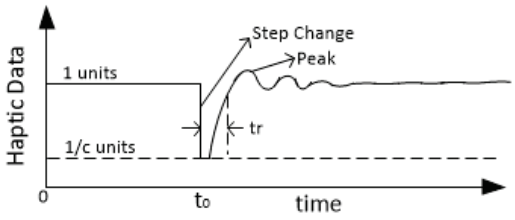

Figure 4. Sample step response profile with normalized haptic data. Step change is simulated at t_0.

ods. In our work, we use the step response method to profile the control performance of TCPS applications [7], [8].

To perform the analysis, we have created two *embsys-app*s; one is connected to the *ms com* and the other to *ss com*. The *ss embsys-app* at the *ss com* replaces the tactile slave. It takes kinematic data from the master side, kin, as input and simulates haptic data (pressure) as a linear function of the kinematic data; $hap = s(t) * kin$, where the scaling factor $s(t)$ is set to 1 for $0 \leq t < t_0$ and to $1/c$ for $t \geq t_0$. This is to simulate a step change in the pressure. The *ms embsys-app* at the *ms com* end senses this step change through the haptic data path. It then sends commands over the kinematic data path to nullify this change. Predefined Proportional Integral (PI) controller equations are used for this purpose [9]. Using a logger tool in *ss embsys-app*, we record the step response profile of the haptic data, see Figure 4. We define t_r as the time taken by the step response to rise from $(1/c)$ $units$ at t_0 to $(1/c + 0.9(1 - 1/c))$ $units$. Overshoot is defined as the peak percentage fluctuation in the step response relative to $(1 - 1/c)$ $units$. Both t_r and overshoot are affected by the characteristics of the TCPS components and the network.

Both rise-time (t_r) and overshoot depend on the TCPS component and network characteristics (latency, jitter, packet drops, etc.), e.g., both may increase with network latency. Further, each TCPS application can withstand rise-times and overshoots up to certain values depending on operator hand speed and accuracy requirements. For instance, a certain rise-time may suit an application with slow operator hand movement, but may not be suitable for another application with faster operator hand movement. Similarly, an overshoot value may be tolerable for an application requiring less accuracy, but may not for another application requiring more accuracy. We thus see that our step response metrics can be used to judge suitability of a TCPS for a certain application. Alternatively, given an application, our testbed can be used to prescribe TCPS components and network characteristics to meet the rise-time and overshoot requirements.

IV. Testbed Implementation and Demonstration

In this section, we describe the implementation aspects of the testbed and further demonstrate its capabilities.

A. Implementation

To implement the end systems, two desktop PCs, PC1 and PC2, running Ubuntu OS and two Programmable System on Chip (PSoC) boards were used [10]. We realized *ms com* in PC1 and *srv* and *ss com* in PC2. We realized *ms embsys* and *ss embsys* in the PSoC boards. The embedded boards and their communication components were connected using a full-duplex UART link operated at a baud rate of 1 Mbps.

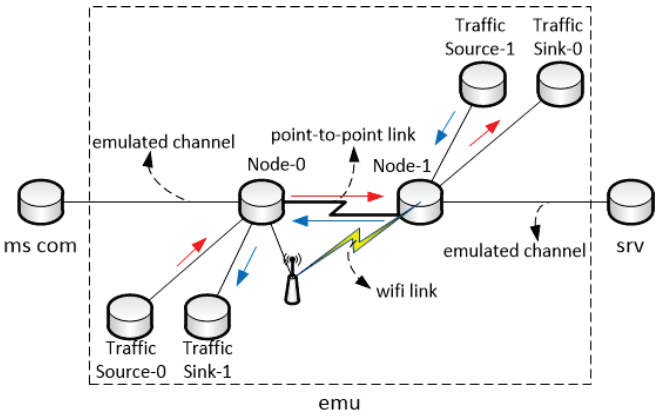

Figure 5. Emulated network topology in *emu*.

a) Network Emulation: For the network portion, we used deployment and emulation modes. In the deployment mode, PC1 and PC2 were connected over a physical network infrastructure. For the emulation mode, we considered an emulated network between PC1 and PC2, which was realized through *emu* hosted in a third desktop PC, PC3. We connected PC1 and PC2 through PC3. The network topology we emulated in *emu* is shown in Figure 5. We set various parameters (e.g., latency, error rate, link selection, etc.) through the GUI as described in Section II-F.

b) API Framework: We used Python scripts to code the APIs in *ms com, ss com* and *srv*. For implementing the API framework, we used looping functions to construct the forward and backward data flows. We used the multiprocessing package of Python to run these looping functions concurrently. For the *ms embsys* and *ss embsys* components, we used C code specific to PSoC to realize the APIs. For implementing the video flow, we used the Python Open-CV library [11]. We used this library to capture video frames from the USB camera connected to *ss com*. We encoded these frames in JPEG format to reduce size. We then used the Python pickle library to serialize these JPEG frames before sending them to *ms com*. At the *ms com*, we deserialized and decompressed the frames and displayed them on a connected LCD screen. For high-definition video transmission, we split the frames into multiple sub-frames and repeated the above process. We used the Python PyAudio library [12] to capture audio frames from the microphone connected to *ss com*. After serializing these frames, we sent them to *ms com*. At *ms com*, we deserialized the frames and played them on a connected speaker.

c) Embsys-Apps: The following embsys-apps are currently supported.

- End-users may not have an electronic glove to wear and control the remote robot [3]. To cater to their needs,

we developed *ms-embsys-app-mouseController*. The state machine in this *embsys-app* maps the movements of a normal computer mouse in the X-Y plane to a virtual hand movement in X-Y-Z space. For this, we mapped the X-Y movement of the mouse to the hand wrist-X-Y movement. Y movement of the mouse, while the left button is in the pressed state, is mapped to the wrist-Z movement. Additionally, the scroll wheel of the mouse is used to mimic the hand palm grip posture. Scroll up and down movements are mapped to mimic the hand palm closure and open actions. In the *embsys-app*, to read the mouse movements, Python pynput library was used [21].

- *ss-embsys-app-microphone* and *ms-embsys-app-speaker* are for recording and playing the audio at the *ss com* and *ms com* sides, respectively.
- *ss-embsys-app-Camera* and *ms-embsys-app-Display* are for recording and playing the video at the *ss com* and *ms com* sides, respectively. When using V-REP for simulating the remote-side robot, *ss-embsys-app-desktopCamera* is used. This is for recording V-REP scenes.
- For running step response experiments, *ms-embsys-app-PIController* and *ms-embsys-app-stepInput* are used. They simulate the PI controller and haptic step change at the master and slave side respectively.
- When using V-REP to simulate the remote-side robot, *ss-embsys-app-vrep* is used. This is for interfacing *ss com* with the V-REP scene.

d) Tools: At present, we support the following two tools.

1) *Latency Characterizer* The tool sends and receives ping commands to carry out TCPS latency experiments. At the end of the experiments, it summarizes the results with statistical data and graphs relating to latency, inter-packet delay, and packet drops.
2) *Step Response Analyzer* This tool reads the raw data generated by the step response experiments and measures the t_r and overshoot. Also, the tool classifies the step response profiles as good or bad depending on whether t_r and overshoot are within given specifications. This is useful for assessing the percentage of good step curves and thus the reliability of the TCPS.

e) Run-Directory: We use scripts *ms com.py*, *emu.py*, *server.py*, and *ss com.py*, corresponding to different testbed components to read the configuration (see Section II-F) and realize the data-paths (see Section II-B) using the APIs (see Section II-C). The scripts, the API libraries, and the testbed tools are stored in a *run-directory*, which is copied in various components (see Figure 6).

B. Demonstrations

1) Deploying an End-to-End TCPS Application: A prototype of a simple TCPS application is described in [13]. There, a remote robotic arm (PhantomX [14]) is controlled to pick and place lego blocks by a human operator wearing a tactile glove. We have deployed this TCPS application on our testbed. This was done by updating the APIs in Figure 2 to the corresponding algorithms described in [13]. Additionally, we substituted

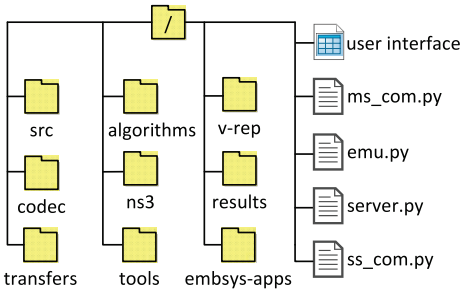

Figure 6. Testbed run-directory.

the tactile glove with a haptic mouse, to demonstrate the testbed's ability to substitute components. The haptic mouse is an ordinary USB mouse with an eccentric rotating mass motor for the purpose of applying haptic feedback [15]. We used the *ms-embsys-app-mouseController* to track the operator hand movements. At the robotic end, we mounted pressure sensors on the PhantomX gripper wings. This is to sense the pressure when the robot holds an object. Since we are dealing with a real network, through the testbed user interface, we configured the testbed in deployment mode. We further enabled the four supported data-paths, kinematic, haptic, audio, and video. The operator hand movements were sent over the kinematic data-path. Pressure, audio, and video data were streamed from the robotic end to the operator, over the haptic, audio, and video data-paths, respectively. For streaming and displaying audiovisual data, we used *ss-embsys-app-microphone* and *ms-embsys-app-speaker* for audio, and *ss-embsys-app-camera* and *ms-embsys-app-display* for video. We illustrate this application in Figure 7.

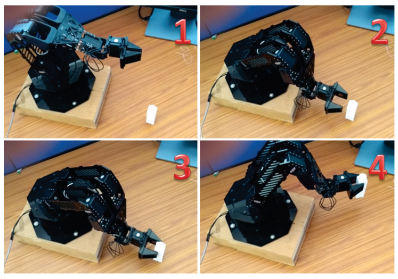

Figure 7. Demonstration of a TCPS application: a human operator guides a robotic arm to pick and place Lego blocks.

2) Isolating TCPS Component and Link Latencies: The Latency characterization method described in Section III-B is designed for isolating TCPS component latencies from a single central location. In the following, we demonstrate how the tool *latency characterizer* can be used for this purpose. For the demonstration, we configured the testbed in deployment mode as described in Section IV-A. Using the *latency characterizer* at the tactile master side, we sent N_{ping} ping commands to various ping points in the forward kinematic data-path. We selected N_{ping} so as to restrict the error in the computed average latency to be within +/-5% at 95% confidence. We

sent the ping commands at an interval exceeding the maximum expected Round Trip Time (RTT) of the TCPS. This is to prevent packet drops affecting the latency calculation. At ping intervals smaller than RTT packets get dropped at the interface between *ss com* and *ss embsys*. This is due to little buffering space in the memory-constrained *ss embsys*.

Figure 8 presents the average latency for different ping points in the forward kinematic data path, corresponding to the entry/exit locations in the testbed components. We had not enabled audiovisual streams. The latency depends on the location of the ping point in the data path. The farther the ping point is from the data source, the higher its latency will be. This is evident from Figure 8. From this figure, we can infer the interface/component latencies and actuation delay of the robot. For instance, δ_1 is the latency of *srv* and δ_2 is the latency of the UART interface connecting *ss com* and *ss embsys*.

a) Estimating Robot Actuation Delay: In Figure 8, the latency that corresponds to the ping point *ss_embsys_exit(1)* does not include the robot actuation delay. Instead, the ping packets targeting *ss_embsys_exit(1)* were sent back as soon as the actuation commands embedded in the ping message were extracted and sent to the joint motors of the robot. However, if we hold the ping packets until the robot moves to the commanded position, as is the case with *ss_embsys_exit(2)*, then the difference in latency numbers between the ping points *ss_embsys_exit(1)* and *ss_embsys_exit(2)* corresponds to the robot actuation delay. In this case, $\delta_3 = 110ms$ corresponds to the actuation delay of the robot in moving its end-effector position by $1cm$ back and forth. This corresponds to an estimated actuation speed of $0.09m/s$. For this experiment, we used a V-REP model of the PhantomX robot instead of a physical one. We interfaced the V-REP scene containing the robot model with the *ss com* module of our testbed using *ss-embsys-app-vrep*.

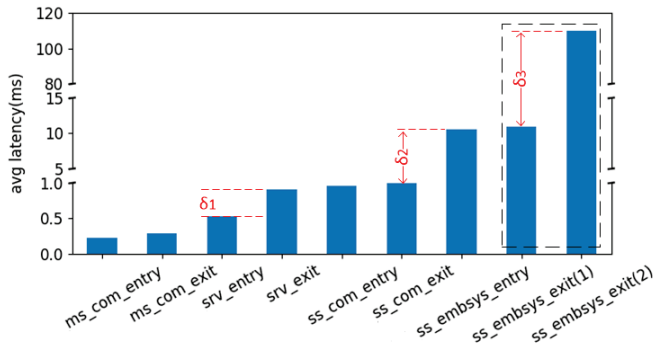

Figure 8. Average latency for different ping points.

3) Selective Enabling of Data-Paths: To study the effect of one modality on another the testbed supports selective enabling and disabling of data-paths. To demonstrate a use case, we conducted the following experiment. We studied the effect of video on latencies in the kinematic data path. For this, we isolated the TCPS component latencies in the kinematic data-path with and without the video data-path enabled. In the experiment, the video frame rate was 30fps with frame size of 60KB. We have plotted the results in Figure 9. We find that, with video, the average latency reduces. This decrease in average latency is non-intuitive, but can be attributed to application-layer overhead and interrupt coalescence at the network interface card (NIC) [16]. Interrupts to CPU are not generated by the NIC for every short packet it receives, rather it accumulates the received packets until they cross a threshold (or a time-out happens), after which an interrupt is generated. In our case, when the video is enabled, the video packets (being large in size) cause the threshold to cross more frequently. This results in the NIC generating interrupts more often, thereby reducing latency. Thus in TCPS applications, to improve latency of kinematic and haptic data paths, we need to disable the interrupt coalescence feature in NIC cards along the transmit path from tactile master to tactile slave.

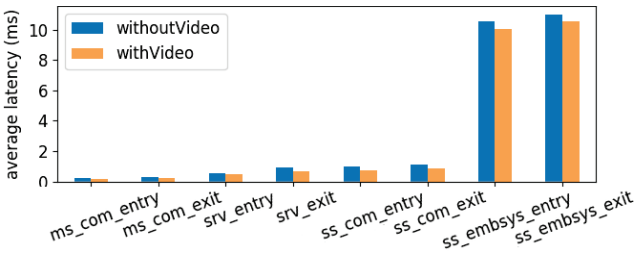

Figure 9. Effect of video on average latencies.

4) Step Response Experiments: In this section, we demonstrate how to extract and analyze the control performance of TCPS applications. For this demonstration, we operated the testbed in emulation mode as described in Section IV-A. We ran the step response experiments as described in III-B. For this, we connected *ms-embsys-app-PIController* and *ss-embsys-app-stepInput* at the *ms com* and the *ss com* side, respectively. We captured the step response profile of our TCPS application against different RTT and packet drops simulated in NS3. We have plotted the results in Figure 10. Our results show that RTT and packet drops affect the step response profile. Any application prescribes limits on overshoot and t_r. It is now possible to determine the maximum RTT and packet drop rate to keep the overshoot and t_r below the prescribed values. For instance, for the implemented TCPS and the sample PI controller coefficients in use, packet drops must not exceed 30% if the maximum allowed overshoot is 20% (see *graph 3* in Figure 10).

5) Testbed Component Placement: The testbed allows for flexible placement of its components through the user interface. A sample of these placement possibilities are demonstrated in Figure 11. In scenario (A) all components are placed in a single PC. This helps the TCPS application developers. In our work, we used this placement during the development of APIs and different *embsys-app*s. Scenario (B) is used for emulating network scenarios, for testing and characterizating TCPS applications. In this paper, (B) was used to generate the results of Figure 10. Scenarios (C) and (D) are used

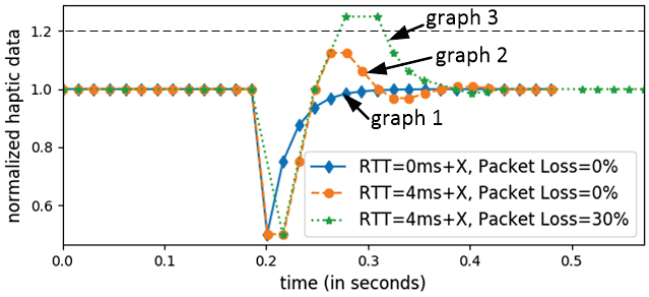

Figure 10. Step response for different RTT and packet drop percentages. X represents the component of RTT that is not part of emulation. The change in pressure ($1 \rightarrow 0.5$) occurs at around 0.2 seconds.

for deploying and testing the TCPS applications across a real network. Towards the demonstration of (C), we placed PC1 and PC2 in two universities: Indian Institute of Science (Bangalore, India) and Delft University of Technology (Delft, the Netherlands). We captured the jitter in the intercontinental TCPS by plotting the latency associated with ping point *srv_entry* (see Figure 12).

V. Related Work

To the best of our knowledge, we are the first to have developed a TCPS testbed. We, however, did find testbeds for specific CPS applications [17], [18], [19], [20], [21]. They are built for a specific use case and as such cannot be used for realizing TCPS applications. None of these aforementioned testbeds employ methods to automate component-level latency characterization nor extract control performance of the realized application, which are key contributions of our paper. Specific to latency characterization, open/commercial tools do exist that characterize the latency between networking nodes [22], [23], [24]. These tools, however, do not characterize the non-networking latencies like sensing, actuation, execution of code, etc. In TCPS applications, both networking and non-networking latencies are equally critical. This motivated us to design the latency and step response characterization tools described in this paper.

VI. Discussion

In this section, we discuss our initial experience with the testbed and present possible future research directions in the area of TCPS.

- In a TCPS, transmitting video can result in jitter in the kinematic data (see Section IV-B3) depending on its quality and frame rate. This can lead to control loop instabilities and consequently cybersickness. The jitter further increases if the video frame rate is modulated to optimize the transmitted data based on the tactile master's hand dynamics, i.e., if the video frame rate is increased when the operator's hand speed increases and is decreased when the hand speed subdues. A possible way to mitigate the increase in jitter is to isolate the video and kinematic data traffic in the network layer, application layer, and TCP/IP stacks in terms of network links used, buffers used, processor times being granted, etc.
- Buffering of network packets at the NIC card, kernel layer, and application layer can increase latency. Since a TCPS has a tight round-trip latency requirement, to support TCPS flows, priority should be given to serving high-frequency TCPS interrupts at the NIC, kernel, and application layers rather than avoiding these by buffering. Eliminating buffering will be helpful to TCPS flows, but will be detrimental for other use cases. An open challenge is to design the buffering and interrupt schemes in such a way that they support both TCPS and non-TCPS flows simultaneously.
- An RTT less than $1ms$ is necessary only for TCPS applications in which the tactile master is expected to make rapid hand movements up to $1m/s$, the natural limit on human hand speed. In applications like telesurgery, where the hand movement speed is an order less than $1m/s$, a round trip latency of even $10ms$ could be acceptable.
- Cybersickness is not the only concern in TCPS applications. Stability also plays a critical role. Apart from the TCPS components, stability also depends on network QoS metrics like latency, jitter, and packet drops (see Section IV-B4). It is thus desired to develop a model to determine for what combinations of the QoS parameter values a TCPS with a prescribed specification is stable. The model will be useful in making architectural and design choices for the implementation of a TCPS. Also desired is an evaluation method and a metric that can grade and compare different TCPS implementations by simultaneously accounting for stability, cybersickness, and TCPS use-case specifications.
- The overhead introduced by network emulators, like NS3, are in the order of a few milliseconds. Though this value is acceptable for some TCPS scenarios, the overhead can be detrimental in emulating TCPS applications that require sub-*millisecond* RTT. Thus the developers of network emulators need to ensure that the emulator overhead is within the sub-millisecond range to ensure compatibility with TCPS. The same applies to the tactile slave simulators.

VII. Conclusion

We have proposed and implemented a testbed design for Tactile Cyber Physical Systems (TCPS) applications. The main objective of our testbed, called TCPSbed, is to enable the quick prototyping and evaluation of TCPS applications. We have equipped our testbed with tools to assess latency and control-loop performance, which are critical TCPS characteristics. Our latency characterization tool can extract and isolate the TCPS latency, not only end-to-end, but also per component and interface, alongside with other QoS parameters like jitter and packet drops. Our step response analyzer can return a measure of the TCPS control-loop performance. Through a proof-of-

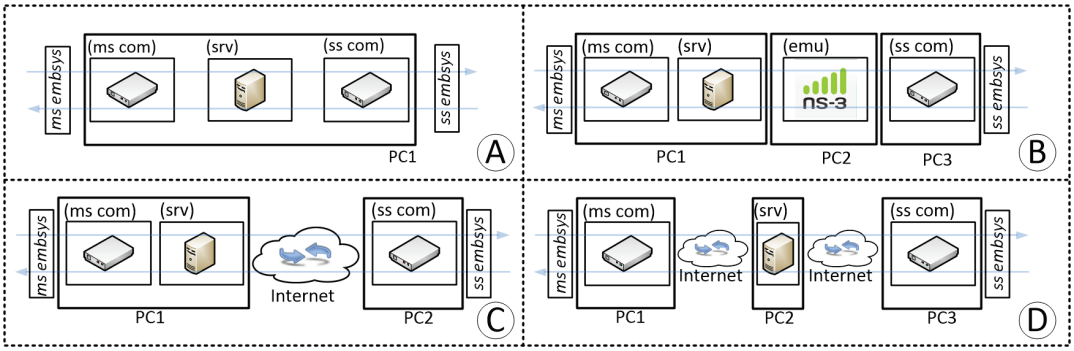

Figure 11. Different placement possibilities for the testbed components.

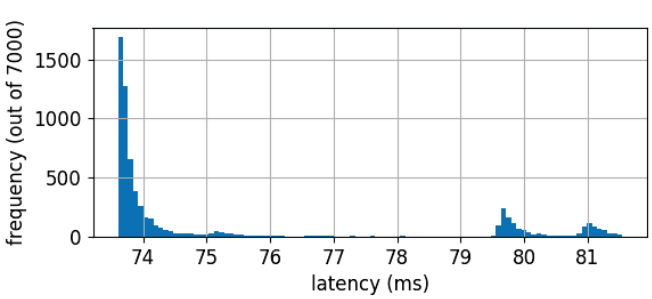

Figure 12. Latency histogram of ping point *srv_entry* for an intercontinental TCPS setup.

concept implementation and corresponding experiments, we have been able to analyze several parts of a TCPS application.

In the same vein as many open wireless networking testbeds, we have built our TCPS testbed modularly, such that it can be easily used and modified others in the future. For example, our intercontinental experiment is evidence that our testbed can be used remotely and that it can be ported to different locations. We have created a GitHub page https://github.com/tactileinternet/ to, in due course, provide open source access to several of our Tactile Internet software repositories.

Acknowledgement

We thank Belma Turkovic for having installed several *TCPSbed* components in Delft, which enabled our international experiment.

References

[1] G. Fettweis and S. Alamouti, "5G: Personal mobile internet beyond what cellular did to telephony," *IEEE Communications Magazine*, vol. 52, no. 2, pp. 140 – 145, February 2014.
[2] G. P. Fettweis, "The tactile internet: Applications and challenges," *IEEE Vehicular Technology Magazine*, vol. 9, no. 1, pp. 64–70, March 2014.
[3] D. van den Berg, R. Glans, D. de Koning, F. A. Kuipers, J. Lugtenburg, K. Polachan, P. T. Venkata, C. Singh, B. Turkovic, and B. van Wijk, "Challenges in haptic communications over the tactile internet," *IEEE Access*, vol. 5, pp. 23 502–23 518, 2017.
[4] [Online]. Available: https://github.com/tactileinternet/
[5] NS-3. (2018) NS-3 Consortium. [Online]. Available: https://www.nsnam.org
[6] Coppelia Robotics. (2018) V-REP Virtual Robot Experimental Platform. [Online]. Available: http://www.coppeliarobotics.com/
[7] MathWorks. (2018) Step Info. [Online]. Available: https://in.mathworks.com/help/control/ref/stepinfo.html
[8] B. Messner and D. Tilbury. (2018) Control System Analysis. [Online]. Available: http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=SystemAnalysis
[9] ——. (2018) Introduction: PID Controller Design . [Online]. Available: http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID
[10] Cypress Semiconductors. (2017) PSoC. [Online]. Available: http://www.cypress.com/products/32-bit-arm-cortex-m3-psoc-5lp
[11] OpenCV. (2017) OpenCV. [Online]. Available: http://opencv.org/
[12] PyAudio. (2017) PyAudio. [Online]. Available: https://pypi.python.org/pypi/PyAudio
[13] N. Arjun, S. M. Ashwin, K. Polachan, P. T. Venkata, and C. Singh, "An End to End Tactile Cyber Physical System Design," in *4th International Workshop on Emerging Ideas and Trends in the Engineering of Cyber-Physical Systems (EITEC)*, April 2018, pp. 9–16.
[14] Trossen Robotics. (2017) PhantomX Reactor. [Online]. Available: http://learn.trossenrobotics.com/interbotix/robot-arms/reactor-arm.html
[15] Adafruit. (2017) Vibrating mini motor disc. [Online]. Available: https://www.adafruit.com/product/1201
[16] K. M. Salehin, V. Sahasrabudhe, and R. Rojas-Cessa, "Remote measurement of interrupt-coalescence latency of internet hosts," in *IFIP Networking*, June 2017, pp. 1–9.
[17] S. Pandi, F. H. P. Fitzek, C. Lehmann, D. Nophut, D. Kiss, V. Kovács, Á. Nagy, G. Csovási, M. Tóth, T. Rajacsics, H. Charaf, and R. Liebhart, "Joint Design of Communication and Control for Connected Cars in 5G Communication Systems," in *IEEE Globecom*, 2016.
[18] H. Cao, S. Gangakhedkar, A. R. Ali, M. Gharba, and J. Eichinger, "A 5G V2X Testbed for Cooperative Automated Driving," *IEEE Vehicular Networking Conference (VNC)*, pp. 1–4, 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7835939/
[19] M. Szczodrak, Y. Yang, D. Cavalcanti, and L. P. Carloni, "An Open Framework to Deploy Heterogeneous Wireless Testbed for Cyber-Physical Systems," *Proc. of the IEEE SIES Symp.*, no. Sies, pp. 215–224, 2013.
[20] W. Wu, Y. Dong, and A. Hoover, "Measuring Digital System Latency from Sensing to Actuation at Continuous 1-ms Resolution," *Presence*, vol. 22, no. 1, pp. 20–35, 2013.
[21] M. Iglesias-Urkia, A. Orive, M. Barcelo, A. Moran, J. Bilbao, and A. Urbieta, "Towards a Lightweight Protocol for Industry 4.0: An Implementation Based Benchmark," in *IEEE International Workshop of Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM)*, May 2017, pp. 1–6.
[22] Linux Man Page. (2017) Ping. [Online]. Available: https://linux.die.net/man/8/ping
[23] SolarWinds. (2017) Network Performance Monitor. [Online]. Available: http://www.solarwinds.com/network-performance-monitor\#features
[24] Visualware. (2017) VisualRoute. [Online]. Available: http://www.visualroute.com/