IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received January 17, 2022, accepted January 30, 2022, date of publication February 1, 2022, date of current version February 11, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3148708

Interface Development for Digital Twin of an Electric Motor Based on Empirical Performance Model

ANTON RASSÕLKIN[1], (Senior Member, IEEE), VIKTOR RJABTŠIKOV[1], (Student Member, IEEE), VLADIMIR KUTS[2,3], (Member, IEEE), TOOMAS VAIMANN[1], (Senior Member, IEEE), ANTS KALLASTE[1], (Senior Member, IEEE), BILAL ASAD[1], (Member, IEEE), AND ANDRIY PARTYSHEV[2]

[1]Department of Electrical Power Engineering and Mechatronics, Tallinn University of Technology, 19086 Tallinn, Estonia
[2]Department of Mechanical and Industrial Engineering, Tallinn University of Technology, 19086 Tallinn, Estonia
[3]Electronic and Computer Engineering Department, University of Limerick, Limerick, V94 T9PX Ireland

Corresponding author: Anton Rassõlkin (anton.drassolkin@taltech.dee)

This work was supported by the Estonian Research Council through the Digital Twin for Propulsion Drive of an Autonomous Electric Vehicle under Grant PSG453.

ABSTRACT The concept of Digital Twin is creating and maintaining a digital representation of the real physical entity and supporting its performance utilizing simulation and optimization tools, which are fed with the real data obtained from the physical equipment. Development and implementation of the Digital Twin technology are one of the main challenges for today's industry, more detailed studies are needed on design methods for Digital Twins. Besides using Digital Twin as a high-quality simulation, one of the commitments is monitoring and maintaining control of the whole system via a constant live link between virtual and physical entities. The related research study presents a detailed structural description of the developed Digital Twin virtual entity and the development of a framework that allows Robot Operating System (ROS) to securely communicate with remote Digital Twins via the Internet and harness ROS's adaptability across vast distances and multiple systems. This paper is an extended version of the authors' International Conference on Electrical Power Drive Systems (ICEPDS20) paper, in which we propose a development case study of Digital Twin for an electric motor based on an empirical performance model.

INDEX TERMS Motor drives, model-driven development, telemetry, digital twin.

NOMENCLATURE

5D -	Five-dimensional
AR -	augmented reality
DS -	Digital Siblings
DT -	Digital Twin
IoT -	Internet of Things
JSON -	JavaScript Object Notation
LAN -	Local Area Networks
MQTT -	Message Queuing Telemetry Transport
PID -	Proportional–integral–derivative
ROS -	Robot Operating System
UDP -	User Datagram Protocol
VR –	Virtual reality

The associate editor coordinating the review of this manuscript and approving it for publication was Claudio Zunino.

I. INTRODUCTION

This paper is an extended version of the authors' IEEE International Conference on Electrical Power Drive Systems paper [1] that took place in October 2020 at Saint-Petersburg, Russia.

Digital Twin (DT) concept is creating and maintaining a digital representation of the real physical entity and supporting its performance through simulation and optimization tools, which are fed with real and updated data. DT is a part of the digitalization and simulation pillar of Industry 4.0 paradigm, according to Wang *et al.*d [2], DT covers three aspects of the system: knowledge content, effect and functionality, and application domain. For utilization of a DT technology, the relations of the three components must be recognized, and the gaps remaining for exploration must be identified and categorized.

The basic DT system, architecture consists of at least three main components – the physical entity in the real world, its virtual entity (or virtual model(s)), and the data; furthermore, services and connections are often presented as separate entities, what makes reasonable a five-dimensional (5D) DT representation [3]. Generally, a real physical entity consists of assorted subsystems and sensory devices that work online; however, it is possible to use already collected data to establish a virtual entity. According to the DT application, the virtual entity subsists of one or several models; it can be the spatial, physical, numerical, behavior, rule, or another model. The DT data entity contains various sets of data from the real physical entity that can be obtained by implementing different sensors and data acquisition platforms. The service entity mainly includes regulation for both virtual and physical entities and may carry several sub-services, such as maintenance and diagnostic, energy optimization, path planning, etc. The connection entity usually defines the interaction between other entities. However, the 5D DT is not the technological boundary, and more studies are needed on design methods for DT to allow full synchronization and connectivity between virtual and real environments [4].

Nowadays, DTs are being made for almost all physical things, including humans, vehicles, organizations, industrial systems, and even ice cream machines [5]. DT is already being used in the field of electrical drives, where the concept is being used for the predictive maintenance [6] and simulation aspects of various drive parts [7]. In [8], several modeling tools have been identified as an attractive solution for the development of DT of vertical transportation systems, which is based on physics-based models and estimation algorithms. However, the field is not fully uncovered, and there are demands for the fully synchronized DT of electrical drives, and to develop it, a virtual copy(ies) of the physical object should be created. It is important to notice that when the status of the physical object changes, all models used in DT must be updated, this issue can be solved by introducing transaction management functions into the framework [9]. DT of the system accommodates specific requirements of applications: providing special functions, or permit access to the data with different rates, or segmenting and securing the computational space. Open-source platforms allow researchers to study, change, and use of source code of such frameworks. One of the most popular platforms for DTs' development is Robot Operating System (ROS) which might be combined with other tools, such as Tecnomatix [10], [11]. Moreover, documentation of the DT is not a guided process, and special attention must be given to the platforms for managing and distributing meta-level DT data, such as Twinbase [5].

In the current study, Unity3D is used for physics simulations and visualization of the DT because of research team previous experience and high-quality documentation provided by software producers and the community.

A recent study presents a part of the project that aimed to develop a specialized unsupervised prognosis and control platform for electric propulsion drive systems performance estimation of an autonomous self-driving electric vehicle. This goal requires the development of several subtasks and related objectives, one of them is to develop physical models of different energy system components (motors, gearboxes, power converters, etc.) and the related reduced models of these components (testbed), which will serve to construct the DT of the system [12].

The paper is structured as follows: Section III presents details about the Empirical Performance Model (EPM) and structure description of the DT virtual entity and its implementation framework based on the chosen software packages. Section IV presents the integration of the middle layer for remote communication and shows the evaluation of various connectivity methods. Sections V and VI – the discussion and conclusion section, present the further development and application of the presented DT.

II. DIGITAL TWIN

A. EMPIRICAL PERFORMANCE MODEL

In order to have a reasonable EPM, 7.5 kW induction motor (IM) was studied. The process of efficiency map obtaining is described in previous research by authors [13]. The graphical representation of an empirical performance model of IM is presented in Figure 1, and the rated data is given in Table 1. Numerical representation of an IM EPM, obtained previously, may be used as one of the inputs for building DT to evaluate the IM performance over a given speed-torque region.

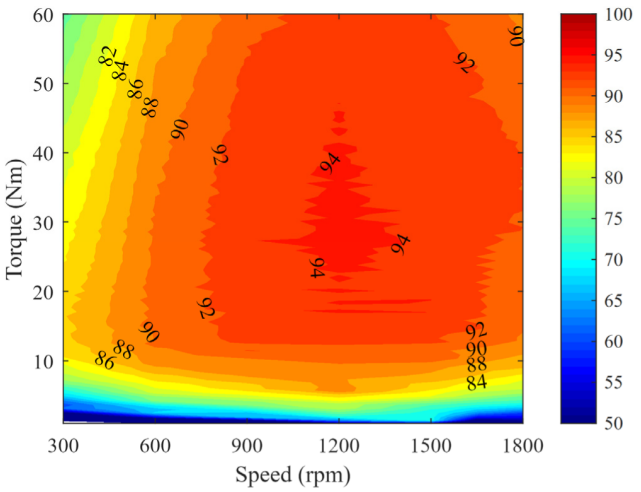

FIGURE 1. The empirical performance model of induction motor used for Digital Twin.

B. STRUCTURE OF THE DIGITAL TWIN

The DT for the IM is set up in the following way: Unity3D physics engine is used to simulate fundamental physics while connected via ROS Bridge. Specific Linux ROS nodes simulate more advanced electrical machine-specific behaviors, such as motor efficiency following the efficiency map and motor controller, as shown in Figure 2. Linux ROS Server sends a User Datagram Protocol (UDP) command packet via

TABLE 1. Rated data of induction motor.

Parameter	Unit	Value
Motor frame size		132 MA
Rated power	kW	7.5
Rated current	A	22
Rated speed	rpm	1460
cosφ		0.6
Moment of inertia	kgm2	0.048

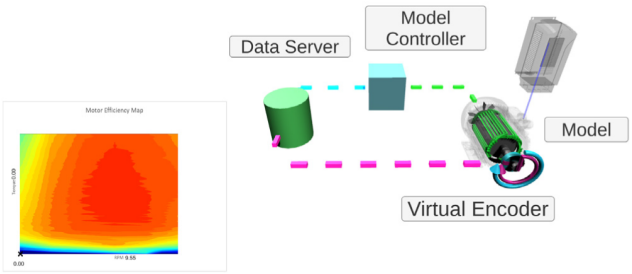

FIGURE 2. The spatial model used for Digital Twin visualization.

ROS Bridge standard to the Unity3D Visualizer. Its main task is to receive a control message from a specified IP address in a standardized ROS Bridge message format that is being published by the Linux ROS Server and control the 3D models to display the behavior of the motor, as it would be in real life. It is also responsible for sending feedback to the Linux ROS Server for processing, just as if a real encoder would send updates on the motor rotation rate to a motor controller. Linux ROS Server consists of multiple nodes that simulate various aspects (e.dg., mechanical, thermal, electrical, diagnostic [14], etc.) of the motor, a single motor controller simulator node, and a Simulation controller node that combines information coming from all other aspect simulation nodes and feedback from the Unity3D, process it and send the data to a visualization client in Unity3D.

More details of the DT physics engine are presented in Figures 3 and 4. Unity3D Visualizer message is received in the form of a UDP package in a custom ROS Bridge format by the ROS Bridge Client, reserialized into input variables (Empirically Estimated Velocity and Torque) of the motor and passed on to the 3D Object Controller as variables. Object Controller converts data from the EPM to DT model velocity at which the motor's shaft rotates. This data is applied to the motor's 3D visualization. Then, a simulated encoder (virtual sensor) takes the angular velocity of the motor shaft, converts it to feedback data, and sends it to the Linux ROS Server.

There are two active nodes in the simulation: a motor controller simulator node and a mechanical simulation node. The motor controller has a proportional–integral–derivative (PID) controller that provides a control signal to the motor. Mechanical simulation node works with an efficiency map, measured and recorded on a real motor, and applies it to a simulation to emulate proper torques and power output at

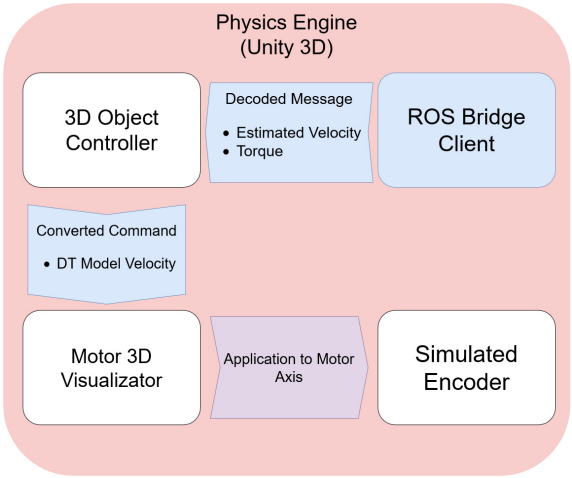

FIGURE 3. Detailed representation of digital twin physics engine.

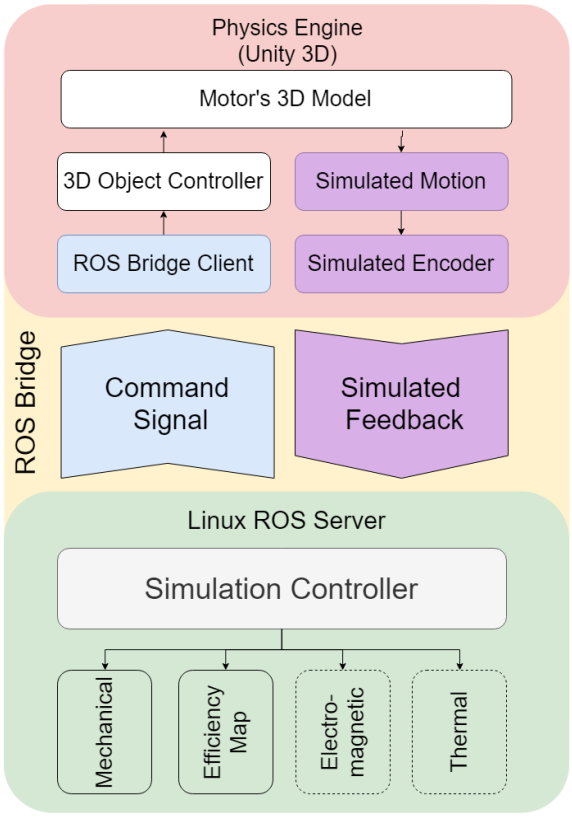

FIGURE 4. The operational architecture of digital twin.

certain angular velocity to make DT act just like a real entity motor.

C. SPATIAL MODEL SIMULATION IN UNITY3D

The DT model does not undoubtedly mean a spatial or graphical model, however, such representation is user-friendly and is broadly used by engineers and scholars. The main accent should be paid to process flow and relations behind such spatial model and the data entity. An application of augmented

reality (AR) and virtual reality (VR) tools for distance simulation via DT adds a safety layer for accelerated lifecycle tests, applications in hazardous environments, and remote work maintenance overall. The spatial model used in the current study is shown in Figure 2.

Physics Engine has a ROS Bridge client who receives a ROS Server data package. The data in that package is extracted by the ROS Bridge client and sent to the spatial model controller that recalculates new positions and orientations of the simulated motor components. After that, the spatial model is updated with a new state. This step ensures that a spatial model acts like a real motor from a visual standpoint. Then, the model considers any other physical forces applied to the motor, such as friction losses and moment of inertia of the motor, simulated via angular drag feature in Unity3D's Rigidbody. After that, the simulation results and updates of simulated encoder values are packed and sent to the ROS Server and act as feedback for the speed controller. The simulated encoder is modular and can be replaced by any necessary feedback simulator.

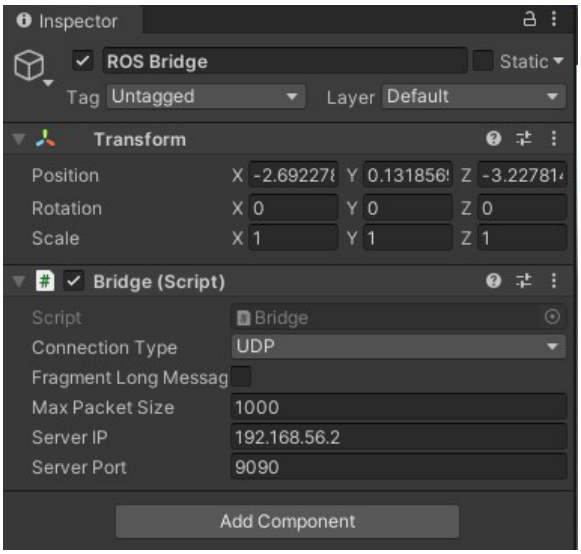

FIGURE 5. Bridge script featured as a component in Unity3D editor.

D. ROS BRIDGE STRUCTURE AND PURPOSE

The ROS Bridge is a client between Linux ROS server and Unity3D, which was designed in four main scripts:

- *Subscriber* – parent class that has to be inherited by any class that wants to receive messages from the ROS Bridge;
- *Publisher* – parent class that has to be inherited by any class that advertises topics to the ROS Bridge and publishes messages;
- *Bridge* – script that is responsible for establishing a connection with the ROS Bridge Server, sending and receiving messages and services; every Publisher and Subscriber reference it; responsible for serializing and deserializing outgoing and incoming messages, respectively, as JavaScript Object Notation JSON) strings;
- *Message* – a serializable parent class used to build a hierarchy of messages similar to that of the ROS framework; each message type has a unique class that corresponds to its type with all of the message's fields as parameters. Message class has a public abstract string *GetMessageType()* method that returns the message type name that should be used when ROS Bridge advertises a topic with any given message.

To note – a parent-child system in Unity3D shows the hierarchy and the dependency of project components to each other, which is crucial in the design of the related study architecture. The Bridge script is the core script that handles the connections. It is a child of *MonoBehaviour* – a superclass provided by the Unity3D engine – it has built-in methods that simplify the integration of the new code into the global loop of the Unity3D project. Such methods are universal and inherited by all *MonoBehaviour's* children. Inheriting *MonoBehaviour* also allows the class that inherited it to attach to the objects in the runtime scene and be considered a Unity3D Component. Unity3D Component is a developer user interface feature that gives a certain amount of control over the values of the variables from the Unity3D Editor, as shown in Figure 5. Bridge script has a couple of public variables such as *enum ConnectionType, int MaxPacketSize*, string *ServerIP*, *int ServerPort*, etc. All named values of these public variables can be edited from the Unity3D Editor while Bridge script is considered as Component due to inheriting from *MonoBehaviour*. This feature of Unity3D Engine is extremely useful for rapid prototyping and debugging.

Bridge script contains a private abstract class Connection with two children – *UDPConnection* and *TCPConnection*. An object of one of two classes is created at the beginning of the runtime and contains information about the network client used to communicate with ROS Bridge. The connection type that Bridge attempts to establish has to match the type of the ROS Bridge running on the ROS Server, which is decided via an *enum ConnectionType* that has two values: UDP and TCP.

After a connection has been established, children of the Publisher class advertise their topics by calling the Advertise() method of the Bridge through the object that references its only object in the runtime. Publisher (same as Subscriber for that matter) finds this object of the Bridge () through a built-in Unity3D-Engine method *FindObjectOfType<T>()*, where T has to be replaced with the class of the object being searched, Bridge is the variable T in this case. If the connection in the Bridge object is present, Subscriber children can request a subscription to the topic via the Subscribe() method. Once they are subscribed to a topic, the *MessageReceived* (T message) method will be called every time the message of the type T is received and used to further process the incoming data in the message. Type T can only be a child of the Message superclass that acts as a root of the message hierarchy implemented in this solution. Each new Message child requires a

corresponding Publisher and Subscriber children to be added to handle the new message type.

E. ROS MESSAGE HIERARCHY IN UNITY3D

ROS Messages are grouped by the type of data they carry, e.dg., *geometry_msgs* group carries information used to describe the objects' coordinates and geometry. For example, Pose that used for the representation of pose in free space, composed of position and orientation, PoseStamped that represent a Pose with reference coordinate frame and timestamp, Pose2D to expresses a position and orientation on a 2D manifold, PoseWithCovariance that represents a pose in free space with uncertainty, PoseWithCovarianceStamped that expresses an estimated pose with a reference coordinate frame and timestamp, Point contains the position of a point in free space, Quaternion that represents an orientation in free space in quaternion form, Vector3 used for the representation of a vector in free space, transforms message TFMessage, TransformStamped that expresses a transform from the coordinate frame, and Twist used to express velocity in free space broken into its linear and angular parts. *std_msgs* group contains messages used to communicate basic data types (e.dg., Int32Array - signed 32-bit integer array, Int32 - signed 32-bit integer, Float32 - 32-bit IEEE float, ASCII string - String, and time clock - Clock. Several other groups of messages that include but are not limited to *sensor_msgs* are used to commute sensor data (such as Joy, which reports the state of a joystick's axes and buttons), and *vision_msgs* that commute visual data and computer vision-related information, etc.

When a topic is advertised in ROS, it has to state its group and type. A similar process occurs when a node or a ROS Bridge client subscribes to the topic, e.dg. when it requests a subscription, it has to specify what type of messages it expects to be published in the topic. ROS Bridge serializes messages into a string format using JSON, a highly flexible and universal approach to recording data from objects. C# also has libraries that allow JSON serialization and deserialization.

Unity3D ROS Message system adaptation consists of a Message superclass inherited by all other classes that describe messages. All message classes inherit a string *GetMessageType()*used by the Subscriber and Publisher classes to specify a message type expected from ROS when subscribing or publishing. Other than that, message classes that are used as ROS messages in Unity3D have a few key guidelines to simplify JSON serialization and deserialization: they can only contain fields- no methods are allowed, their fields can only consist of simple data types and objects of another Message children, and they must override string *GetMessageType()* and clearly state the ROS message type corresponding to their specific class. Type Dependency Diagram of the message hierarchy, showing everything described above, can be seen in Figure 6. Solid blue lines represent inheritance, and green dotted lines show which classes use objects of other classes and the field name used by the object of that class.

III. INTEGRATION OF MQTT MIDDLE LAYER FOR REMOTE COMMUNICATION

ROS is a well-developed system, modular by design and with a thought-through messaging system, which was developed to work in relatively small systems. ROS communication was designed to work in Local Area Networks (LANs), and thus it has some downsides that make it almost impossible to use for remote communication via the Internet. The largest obstacle is the complete absence of security, ROS has no authentication or message encryption, which means anyone with access to the LAN with the ROS Server and an IP address of that server can read and write in all the topics of that ROS Server. Current work will use Message Queuing Telemetry Transport (MQTT) protocol as a transport layer for ROS messages to bypass the security issue. MQTT is considered a standard for Internet of Things (IoT) messaging and is used in various control systems.

A. INTEGRATION OF UNITY3D AND MQTT WITH THE ROS-UNITY3D INTERFACE

Because of yet agile structure, MQTT clients were implemented in almost every programming language, including C# used by Unity3D. MQTT client works with ROS Messages implemented in Unity3D, as described in Section IV. The main operational scripts used in the MQTT integration are:

- *MqttClient* – a script that acts as a networking interface – sends and receives messages using a.dNET framework;
- *MqttUnityClient* – a class that inherits from *MonoBehaviour* superclass and acts as a component in the Unity3D engine; it behaves as a wrapper for *MqttClient* that adapts its behavior to match the Bridge script created in the ROS-Unity3D interface; minor adjustments were made to work with *MqttSigleton*;
- *MqttSingleton* – a script that uses a variation of a common Singleton Design Pattern to allow the use of multiple MQTT Brokers in the same runtime contingency in case the main Broker fails;
- *MqttPublisher* and *MqttSubscriber* – superclasses that play the same role as Publisher and Subscriber in Unity3D.

MQTT protocol is used to transport an array of bytes, not a string, so the first order of business when creating *MqttUnityClient* was to add *SerializeJSON<T>* (T data) and *DeserializeJSON<T>*(byte[] data) methods:

- *SerializeJSON<T>* (T data) is used to serialize an object passed as a data parameter to JSON string using Unity3D's built-in *JSONSerializeModule* and then to a byte array.
- *DeserializeJSON<T>* (byte[] data) method does the exact opposite by first converting data from a byte array to a JSON-formatted string and then attempting to deserialize it into an object of a class specified as a T variable.

This seemingly small addition to the *MqttUnityClient* enables it to use ROS messages the same way they are used in the Bridge script that communicates with the ROS Bridge

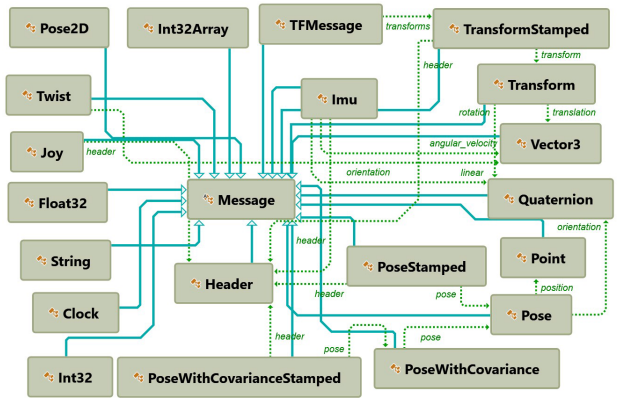

FIGURE 6. Type dependency diagram of the ROS message hierarchy implemented in Unity3D.

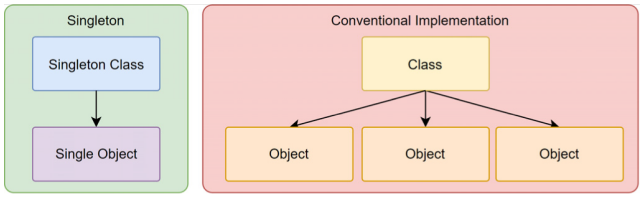

FIGURE 7. Comparison of singleton and conventional classes.

server, making the transition from ROS Bridge middle layer to MQTT effortless and with almost zero code footprint. This contributes to one of the main goals of developed DT - to make a flexible and modular solution that can be easily integrated into any further DT.

Like in the ROS Bridge interface created for Unity3D, each Message class child needs a MqttPublisher and MqttSubscriber children who will send and receive the data in the specified format. MQTT versions of the messages were created similarly to ROS versions of Publisher and Subscriber, with the same architecture.

MqttSingleton is an added feature to the Unity3D MQTT client. It is necessary since communication via the Internet can be unreliable, and having multiple MQTT Brokers acting as backups in case the main Broker connection fails can be irreplaceable for DT systems designed to sustain constant control of the real system and need live updates from the controlled system to update the simulation. It must be noted that *MqttSingleton* was not a part of the ROS-Unity3D Interface. *MqttSingleton* class uses a variety of common Singleton Design Pattern, a class design to ensure that there is only one instance of an object of the singleton class in the entire runtime by making its constructor private and replacing it with a public static *getInstance()* method. *getInstance()* method generates a private static object instance of the class internally on the first call and returns the same instance in the next call. A graphical comparison between a conventional class and singleton can be seen in Figure 7. This pattern is used to ensure that there will be only one *MqttClient* instance in the runtime; however, that solution requires multiple Brokers and multiple clients. To solve this complication, the *MqttSingleton* script has a private static *Dictionary<string,MqttClient>* Clients that contains only one *MqttClient* instance per MQTT Broker. To ensure that, public static *MqttClient getClient(string address, int port=1883, bool isEncrypted=true)* method is created. Much like the *getInstance()* method in a singleton, it creates an instance of a *MqttClient* that connects to an MQTT Broker at an IP address and a port specified by an address and port parameters, respectively. However, it is not limited to a single *MqttClient*, every time a new *MqttClient* instance is created, it is added to the Clients Dictionary, and the address parameter doubles as the key for that instance. If the *MqttUnityClient* instance calls the *getClient* method with an address parameter that already exists in the Clients Dictionary, instead of creating a second instance of the *MqttClient,* the *getClient* method gets the instance from the dictionary. Thus, making it impossible to have multiple clients connected to the same MQTT Broker but still allowing for multiple clients to be created if they are connected to different Brokers.

B. INTEGRATION OF ROS AND MQTT WITH THE ROS-UNITY3D INTERFACE

To completely transition all communication between ROS and Unity3D to MQTT, a ROS node that can run on the ROS side and act as a bidirectional ROS to MQTT converter is needed. *mqtt_bridge* is used for that purpose, this node transfers messages from ROS specified topics to their corresponding MQTT topics and sends them to the MQTT Broker, and in the opposite direction. A configuration file defines which outgoing and incoming topics is used whenever the node starts running. In this configuration file, MQTT Broker address, port, username, password, message direction of the conversion, ROS Message type, ROS message topic, and MQTT message topic are all specified in a specific format.

To maintain an existing functionality of the ROS-Unity3D interface and add a remote communication option, two modes of operation is used:

- LAN Mode – a mode in which no Internet connection is required; however, both the machine that runs the Unity3D application and the ROS system that unity communicates with have to be in the same Local network; no additional hardware besides the two present machines is needed;
- Remote Mode – a mode in which communication between Unity3D and ROS systems happens via a secure Internet connection; that mode requires a remote MQTT Broker that is accessible via a global IP address.

Both LAN and Remote Modes have an MQTT Broker that is used as a middle layer in the communication between ROS Node *mqtt_bridge* and Unity3D *MqttUnityClient*. However, in LAN mode system uses the same Linux machine that runs ROS to act as an MQTT Broker, as shown in Figure 8a. In contrast to the LAN Mode, the Remote Mode of operation uses an external machine as an MQTT Broker. This machine

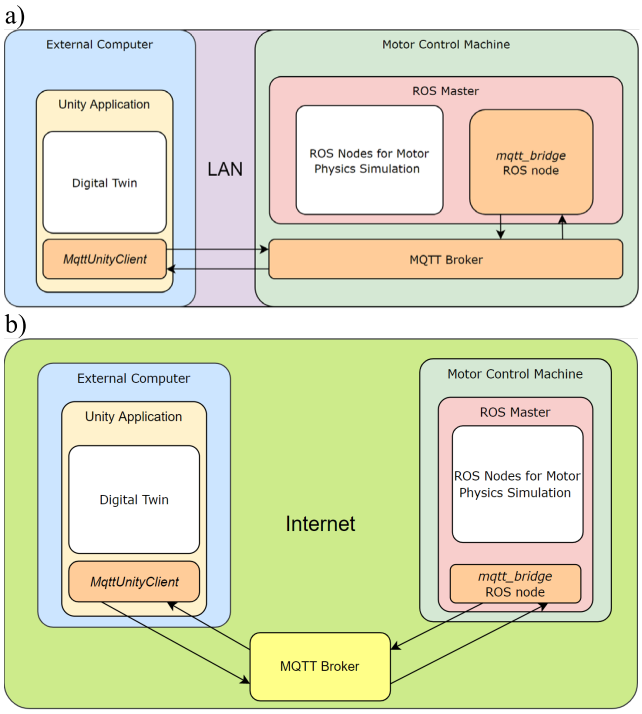

FIGURE 8. Operational architecture of the LAN Mode (a) and Remote mode (b) of MQTT-based ROS-Unity3D interface.

must have a global IP address and be accessible through a specified MQTT port. In addition, the administrator of that machine has to provide credentials that the MQTT clients can use to communicate with the Broker. The operational architecture of the Remote Mode can be found in Figure 8b. Due to the massive similarities of the operational architectures and the feature added through the *MqttSingleton* script that allows the Unity3D side of the interface to use multiple different MQTT Brokers, both modes can be used simultaneously for different machines or as a fallback option.

C. SYSTEM LATENCY ANALYSIS

Latency analysis, to verify the need for a private server, was comparing four communication approaches that could be used with developed DT: communication via the Unity3D-ROS Bridge interface, communication through MQTT via the LAN Mode and through the Remote MQTT Broker on the server, and a public available MQTT Broker [15].

One of the largest factors in DT controllers is the latency between the DT and the twinned system. High latency results in inaccurate and obsolete data, which in turn makes it impossible to consider DT an online representation of the real system, in production could lead to injury or damages to the equipment. Virtual entity of DT requires to mean latency under 100 ms to avoid causing seasickness to the user, thus making public MQTT brokers unusable for VR implementations [16]. There are two solutions to avoid latency problem - to host a private MQTT broker that is not overloaded with request, or to limit the scope of the system to a local network and run it directly through the ROS bridge. A latency analysis has been performed to avoid this and ensure that the developed solution is viable. Latency was analyzed by sending a message from the Unity3D application to the Broker, then forwarded it to the *mqtt_bridge* node in the ROS system. Then the message was converted to the ROS Message and then immediately back to the MQTT message and was sent back to the Unity3D application through the specific Broker. This latency measure represents the time it takes for a message to go on a round-trip; therefore, the latency is called bidirectional. Each approach was tested with two scenarios:

1. The 100 messages and a connection establishment message – 101 in total.
2. The 1000 messages and a connection establishment message – 1001 in total.

Sample sizes of 100 and 1000 were chosen arbitrarily but following the statistic rule that states that a sample size above 30 should be used for a statistically significant result. A range of 970 messages is used to compare significant differences in data amount.

Results of the latency analysis are presented in Figure 9, and a heavily loaded public MQTT Broker was the slowest by taking an average of ca 350 ms to deliver a message in both ways. It was an expected result that heavy network load and large amounts of simultaneous requests can easily be explained. However, LAN Mode and Remote Mode using a private server yielded almost identical results, taking slightly more than an average of 50 ms to send and receive the message, that the most probable cause of the latency is the middle layer between ROS and Unity3D in the form of MQTT Broker. As it can be seen from the experiments, both LAN Mode and Remote Mode can be used interchangeably without any noticeable changes in performance, in case the remote server is not overloaded. The fastest communication approach with the lowest latency was a direct ROS Bridge interface. This approach is the fastest for several reasons, most impactful of which are that there is no intermediary between ROS and Unity3D, and there is no need in *mqtt_bridge* that has to interpret ROS and MQTT messages. The resulting two-way latency was barely above 1 ms.

IV. DISCUSSION

Industry 4.0 revolves around interconnected systems consisting of independent devices that communicate with each other to achieve higher efficiency and productivity. Long-distance communication can be considered a higher-level tool that expands such systems' horizons, allowing them to interact in live regardless of their geographical location. Remote monitoring, tuning, reprogramming, and control of IoT systems shifted from being a useful additional option into a high-priority necessity. However, to effectively harness the power of such a feature, it must have the flexibility to support various types of systems and, more importantly, to join them into one network seamlessly.

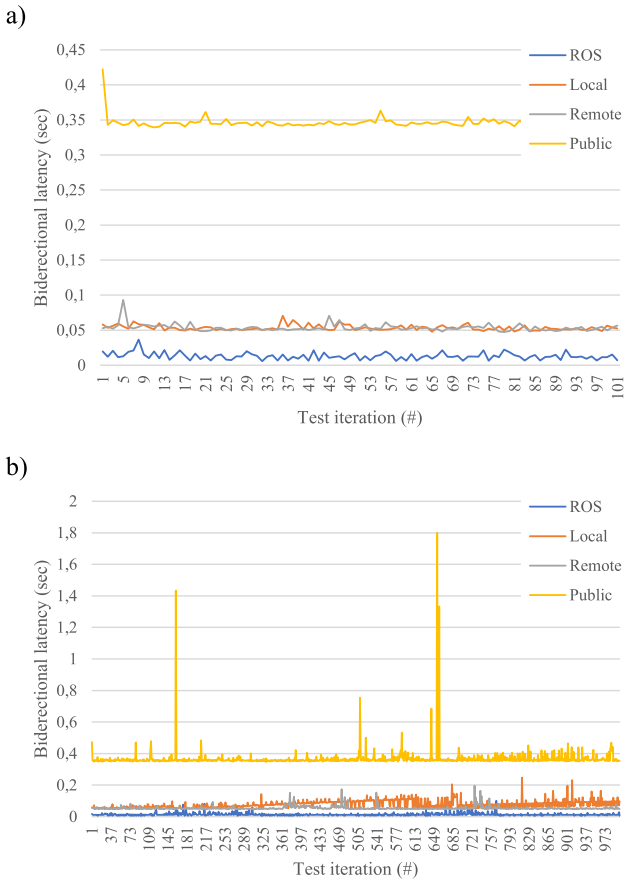

FIGURE 9. Latency analysis of the communication between Unity3D and ROS systems via various channels with 100 messages (a) and 1000 messages (b).

DT is a virtual copy of an entire manufacturing system. Besides using it as a high-quality simulation, one of the DT's main goals is monitoring and maintaining control of the whole system via a constant live link between a DT and a whole system. DT models already exist, but the vast majority of them are designed for a specific system or network of systems, and thus it is challenging to add new members to such DT models. Remote ROS interfacing could resolve this issue and make DT systems more versatile. ROS framework can be used to control a vast range of systems and be easily modified to evolve along with the systems it is supposed to regulate. Creating a framework that allows ROS to securely communicate with remote DT via the Internet and harness ROS's adaptability across vast distances and multiple systems is the topic researched in this paper.

The general concept of DT may also include the term Digital Siblings (DS) that was introduced in [7] and can be considered as a copy(ies) of the physical entity, which need not necessarily run in real-time but can be used to test out hypothetical scenarios for maintenance, diagnostics, ''what if?'' analysis and risk assessment. A variety of condition monitoring techniques are available nowadays and may be combined with DT and DS approaches, different optimization techniques may be integrated in an earlier stage [17]. Reduced models of the physical entities for DT and DS can be constructed using different model order reduction methods. Cross-platform software that combines multiphysics graphical applications with powerful pre-processing, solvers, and post-processing capabilities will be preferable for DS and DT creation.

The fact that developed DT ROS nodes are modular makes it easy to add nodes responsible for electromagnetic, thermal, diagnostic, or any other simulations. Studies show [18] that a model order reduction is a promising tool for controlling industrial processes, where some of the parameters cannot be measured directly. That means different reduced models of the devices running parallel and can be used to assemble a DT in real-time. Development [19], simulation [16], optimization [20], and even manufacturing processes [21] will definitely become easy using DT technology. Application of DT for optimization of the early-stage system may have the potential to change the optimization concept of it, changes suggested by optimization may be immediately applied in various simulations to evaluate how the final product will interact with a real environment.

V. CONCLUSION

The concept of DT is gaining popularity nowadays in many different industry-oriented fields. Creating and maintaining a digital representation of the real physical entity and supporting device performance using different simulation and optimization tools are key goals of many research works. There is no conventional methodology for DT development today, while the DT may include several physical or data-driven models.

This paper presents a development case of DT for an electric motor based on an EPM. A detailed structural description of the virtual entity based on Unity3D engine is presented. The paper considers obtained data required for the DT development.

Unity3D-ROS interface that was the one of the goals of this case study is presented in the Chapter VI. Data fed current DT via ROS as a connectivity layer. While standard ROS systems can only communicate locally, the use of MQTT as a middle layer opens new applications for the ROS systems. The developed solution consists of a software package that can be integrated into DT Unity3D projects and a configuration approach that has to be used on the ROS systems to allow secure communication. This DT also contains an additional local ROS-Unity interface that uses ROS Bridge for local ROS-Unity3D interfacing.

The main application of the recent DT is a loading motor-drive system for the test bench to estimate the performance of the electric propulsion drive system of an autonomous electric vehicle. The development and implementation of the concept of DT will help provide a brand-new approach for measuring and estimating the performance of motor-drive systems.

Future development includes improving the real-time collector from the physical entity (electrical motor) and the ability to control studied system from online DT environment and based on DT services entity. Moreover, the authors look to enlarge the use case with other visualization tools similar and different from Unity3D to prove modularity and flexibility of the middle layer.

APPENDIX

The visualization of real-time operation of the considered example one can see in the following URL: https://youtu.dbe/lralWRjRMkE

Source code of developed DT is available on: https://github.dcom/TalTech-PSG453/loading_motor_dt

REFERENCES

[1] A. Rassolkin, V. Rjabtsikov, T. Vaimann, A. Kallaste, V. Kuts, and A. Partyshev, "Digital twin of an electrical motor based on empirical performance model," in *Proc. 11th Int. Conf. Electr. Power Drive Syst. (ICEPDS)*, Oct. 2020, pp. 1–4, doi: 10.1109/ICEPDS47235.2020.9249366.

[2] K. J. Wang, T. L. Lee, and Y. Hsu, "Revolution on digital twin technology—A patent research approach," *Int. J. Adv. Manuf. Technol.*, vol. 107, pp. 4687–4704, 2020, doi: 10.1007/s00170-020-05314-w.

[3] F. Tao, M. Zhang, and A. Y. C. Nee, "Five-dimension digital twin modeling and its key technologies," in *Digital Twin Driven Smart Manufacturing*. New York, NY, USA: Academic, 2019, pp. 63–81, doi: 10.1016/B978-0-12-817630-6.00003-5.

[4] V. Kuts, G. E. Modoni, T. Otto, M. Sacco, T. Tähemaa, Y. Bondarenko, and R. Wang, "Synchronizing physical factory and its digital twin through an IIoT middleware: A case study," *Proc. Estonian Acad. Sci.*, vol. 68, pp. 364–370, 2019, doi: 10.3176/proc.2019.4.03.

[5] J. Autiosalo, J. Siegel, and K. Tammi, "Twinbase: Open-source server software for the digital twin web," *IEEE Access*, vol. 9, pp. 140779–140798, 2021, doi: 10.1109/ACCESS.2021.3119487.

[6] S. Venkatesan, K. Manickavasagam, N. Tengenkai, and N. Vijayalakshmi, "Health monitoring and prognosis of electric vehicle motor using intelligent-digital twin," *IET Electr. Power Appl.*, vol. 13, no. 9, pp. 1328–1335, Sep. 2019, doi: 10.1049/iet-epa.2018.5732.

[7] R. Goraj, "Digital twin of the rotor-shaft of a lightweight electric motor during aerobatics loads," *Aircr. Eng. Aerosp. Technol.*, vol. 92, no. 9, pp. 1319–1326, Apr. 2020, doi: 10.1108/AEAT-11-2019-0231.

[8] M. Gonzalez, O. Salgado, J. Croes, B. Pluymers, and W. Desmet, "A digital twin for operational evaluation of vertical transportation systems," *IEEE Access*, vol. 8, pp. 114389–114400, 2020, doi: 10.1109/ACCESS.2020.3001686.

[9] R. Minerva and N. Crespi, "Digital twins: Properties, software frameworks, and application scenarios," *IT Prof.*, vol. 23, no. 1, pp. 51–55, Jan. 2021, doi: 10.1109/MITP.2020.2982896.

[10] C. S. Sueldo, S. A. Villar, M. D. Paula, and G. G. Acosta, "Integration of ROS and tecnomatix for the development of digital twins based decision-making systems for smart factories," *IEEE Latin Amer. Trans.*, vol. 19, no. 9, pp. 1546–1555, Sep. 2021, doi: 10.1109/TLA.2021.9468608.

[11] D. Guerra-Zubiaga, V. Kuts, K. Mahmood, A. Bondar, N. Nasajpour-Esfahani, and T. Otto, "An approach to develop a digital twin for industry 4.0 systems: Manufacturing automation case studies," *Int. J. Comput. Integr. Manuf.*, vol. 34, no. 9, pp. 933–949, 2021, doi: 10.1080/0951192X.2021.1946857.

[12] A. Rassolkin, T. Vaimann, A. Kallaste, and V. Kuts, "Digital twin for propulsion drive of autonomous electric vehicle," in *Proc. IEEE 60th Int. Sci. Conf. Power Electr. Eng. Riga Tech. Univ. (RTUCON)*, Oct. 2019, pp. 1–4.

[13] A. Rassolkin, H. Heidari, A. Kallaste, T. Vaimann, J. P. Acedo, and E. Romero-Cadaval, "Efficiency map comparison of induction and synchronous reluctance motors," in *Proc. 26th Int. Workshop Electr. Drives, Improvement Efficiency Electr. Drives (IWED)*, Jan. 2019, pp. 4–7, doi: 10.1109/IWED.2019.8664334.

[14] V. Rjabtsikov, A. Rassolkin, B. Asad, T. Vaimann, A. Kallaste, V. Kuts, S. Jegorov, M. Stepien, and M. Krawczyk, "Digital twin service unit for AC motor stator inter-turn short circuit fault detection," in *Proc. 28th Int. Workshop Electr. Drives, Improving Rel. Electr. Drives (IWED)*, Jan. 2021, pp. 1–5.

[15] *Free Public MQTT 5 Broker Server | EMQ*. Accessed: Feb. 1, 2021. [Online]. Available: https://www.emqx.io/mqtt/public-mqtt5-broker

[16] V. Kuts, T. Otto, T. Tähemaa, and Y. Bondarenko, "Digital twin based synchronised control and simulation of the industrial robotic cell using virtual reality," *J. Mach. Eng.*, vol. 19, no. 1, pp. 128–144, 2019, doi: 10.5604/01.3001.0013.0464.

[17] B. Asad, T. Vaimann, A. Belahcen, A. Kallaste, A. Rassõlkin, and M. N. Iqbal, "Broken rotor bar fault detection of the grid and inverter-fed induction motor by effective attenuation of the fundamental component," *IET Electr. Power Appl.*, vol. 13, no. 12, pp. 2005–2014, Dec. 2019.

[18] D. Panek, T. Orosz, P. Kropik, P. Karban, and I. Dolezel, "Reduced-order model based temperature control of induction brazing process," in *Proc. Electric Power Quality Supply Rel. Conf. (PQ) Symp. Electr. Eng. Mechatronics (SEEM)*, Jun. 2019, pp. 1–4, doi: 10.1109/PQ.2019.8818256.

[19] T. Orosz, K. Gadó, M. Katona, and A. Rassõlkin, "Automatic tolerance analysis of permanent magnet machines with encapsuled FEM models using digital-twin-distiller," *Processes*, vol. 9, no. 11, p. 2077, Nov. 2021, doi: 10.3390/pr9112077.

[20] R. H. Guerra, R. Quiza, A. Villalonga, J. Arenas, and F. Castano, "Digital twin-based optimization for ultraprecision motion systems with backlash and friction," *IEEE Access*, vol. 7, pp. 93462–93472, 2019, doi: 10.1109/ACCESS.2019.2928141.

[21] A. Villalonga, E. Negri, G. Biscardo, F. Castano, R. E. Haber, L. Fumagalli, and M. Macchi, "A decision-making framework for dynamic scheduling of cyber-physical production systems based on digital twins," *Annu. Rev. Control*, vol. 51, pp. 357–373, Dec. 2021, doi: 10.1016/j.arcontrol.2021.04.008.