A Case Study of API Design for Interoperability and Security of the Internet of Things

Dongha Kim[0000−0001−7660−9646], Chanhee Lee[0009−0009−0191−6874], and Hokeun Kim[0000−0003−1450−5248]

Arizona State University, Tempe, AZ 85281, USA
{dongha,chanheel,hokeun}@asu.edu

Abstract. Heterogeneous distributed systems, including the Internet of Things (IoT) or distributed cyber-physical systems (CPS), often suffer a lack of interoperability and security, which hinders the wider deployment of such systems. Specifically, the different levels of security requirements and the heterogeneity in terms of communication models, for instance, point-to-point vs. publish-subscribe, are the example challenges of IoT and distributed CPS consisting of heterogeneous devices and applications. In this paper, we propose a working application programming interface (API) and runtime to enhance interoperability and security while addressing the challenges that stem from the heterogeneity in the IoT and distributed CPS. In our case study, we design and implement our application programming interface (API) design approach using open-source software, and with our working implementation, we evaluate the effectiveness of our proposed approach. Our experimental results suggest that our approach can achieve both interoperability and security in the IoT and distributed CPS with a reasonably small overhead and better-managed software.

Keywords: Internet of Things · Interoperability · Security · API Design.

1 Introduction

Heterogeneous distributed systems, including the Internet of Things (IoT) or distributed cyber-physical systems (CPS), have been rising, especially with the benefits of edge computing, such as low latency, privacy protection, and scalability [35, 52]. As more IoT and distributed systems running on the edge are used in many different domains, it is increasingly challenging to support a diversity of communication models because of the heterogeneity in edge-computing environments where the heterogeneous devices and applications interact with one another [8]. One representative such application is the smart city applications [9,23] that work across the boundaries of different domains, including smart homes and buildings with various purposes under heterogeneous environments.

Thus, the problem of *interoperability* [3] between various components in the IoT and distributed CPS has become one of the major obstacles blocking further deployment of such systems. This interoperability problem includes dealing with the different security requirements depending on the specific applications and the characteristics of their data [51, 53]. For example, safety-critical applications such as transportation or medical applications will require stronger security guarantees [49], while environmental sensing systems may require minimal security, such as data integrity while prioritizing low power consumption over security [46].

There have been research efforts to address the interoperability and security problems in the IoT and distributed CPS, such as a semantic-based collaboration platform [47], networking strategies for interoperability in real-time systems [12], or a secure network for mobile edge computing [24]. As programming models and APIs play a critical role in the development and deployment of edge-based IoT and CPS [11, 26], programming models dedicated to edge-based systems [28] have been proposed. However, to the best of our knowledge, there has not been enough research work on programming models or APIs tailored to the IoT and distributed CPS with a working implementation trying to address both interoperability and security problems as a single integrated platform.

In this paper, we design an API and conduct a case study with the working implementation of the runtime based on the designed API to address the interoperability and security issues of the IoT and distributed CPS. Our contributions are threefold:

1. We design an API that supports multiple communication models, including point-to-point (e.g., client-server) and publish-subscribe paradigms, to facilitate seamless interaction between heterogeneous devices.
2. We incorporate a flexible security framework that can be adaptively applied based on varying security requirements.
3. We develop a working runtime system using open-source platforms to demonstrate the practicality of the API, and evaluate its performance, showing that it achieves interoperability and security with reasonably small overhead while simplifying software development and enhancing maintainability.

2 Related Work

Extensive surveys and reviews have been given by the literature [2,4,13,25,36,43] on the interoperability and security issues in the IoT and distributed CPS. Abounassar *et al.* [2] point out that the IoT, especially in the healthcare sector, still suffers from security and interoperability challenges even now in the 2020s. Lee *et al.* [25] provide a survey on current standards and efforts to bring interoperability and security to the IoT; however, they also present a number of remaining challenges, including the lack of developer support for enabling interoperability and limited security considerations in standards. A survey by Amjad *et al.* [4] reviews the interoperability and security challenges in industrial IoT (IIoT) with a focus on data transfer and application protocols, including the security vulnerabilities in the widely used publish-subscribe protocols [43]. Noura *et al.* [36] discuss the interoperability issues in the IoT from various perspectives, including, network interoperability, syntactical/semantic interoperability, and platform interoperability. Gürdür and Asplund [13] emphasize the importance of interoperability in CPS under distributed environments.

There have been several efforts and approaches to dealing with interoperability and security in the IoT and distributed CPS. Oh *et al.* [38] identify the security threats in the widely used interoperability protocol for heterogeneous IoT platforms, the OAuth 2.0 framework. Margarita *et al.* [33] consider interoperation among various entities, including IoT devices and production lines focused on Industry 4.0; however, this approach primarily targets the REST communication model and does not support various network protocols or security concerns. Pereira *et al.* [40] propose an ontology-based middleware to mitigate interoperability issues, also in industrial IoT (IIoT). The OPC unified architecture (OPC UA) [39] has been designed to support the interoperability between the smart sensors and the cloud by embracing heterogeneous communication protocols.

Some research work focuses on encryption or authentication to ensure security in the IoT and distributed CPS. Kannan *et al.* [16] apply encryption and digital signatures only targeting military applications to secure communication in an IoT environment. Sensor data, commonly used in military equipment, are collected and tested using encryption to show the effectiveness against data modification. This literature [16], however, does not consider the authentication and authorization process. Quadir *et al.* [41] propose an authentication framework targeting consumer electronics in medical, home, and personal applications. The framework proposed by Quadir *et al.* [41] mainly depends on a centralized server to communicate the distributed device via TLS. Secure Swarm Programming Platform (SSPP) [19] is a conceptual platform for providing security at the programming platform level, however, yet without a concrete implementation.

Also, some prior arts address the security of edge-based IoT and distributed CPS running on Robot Operating System (ROS). Sanchez *et al.* [48] integrate a smart personal protective system into ROS with edge computing. For communication between devices, only the MQTT protocol is provided with a Mosquitto message broker, and encryption based on SSL is considered only. Zhang *et al.* [54] utilize ROS2, where data distribution service (DDS) is used only to provide authentication and access control. This approach, however, depends on both cloud computing and edges. It causes user privacy problems immediately across the cloud and edge and can experience performance degradation due to communication overhead.

3 Proposed Approach

This section introduces our proposed API and runtime, designed based on our observation of different modes of communication and security requirements that are common to the IoT and distributed CPS.

For the discussion of our API design, we define two types of nodes, i.e., *Listener* and *Connector* (shown in Fig. 1), to represent typical nodes in the IoT and distributed CPS. We note that these terms (*Listener* and *Connector*) or

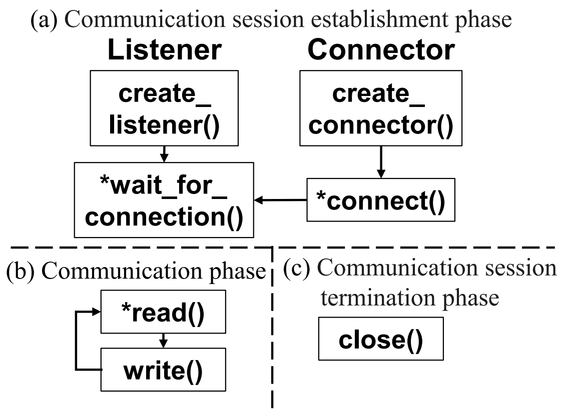


Fig. 1: Proposed API functions. Functions with * are blocking functions.

their concept are not novel, but we use these terms to capture the important and common characteristics of the nodes used in networked and distributed systems, mainly for our discussion in this paper. A *Connector* requests a connection to be waited and accepted by a *Listener*. These nodes are defined so that we can flexibly and seamlessly support different modes of communication and optional security guarantees.

3.1 Design of API Functions

Our proposed API consists of seven functions for *Listeners* and *Connectors*. These functions are used to establish connections and transfer data between different communication models. The API is roughly divided into three phases: communication session establishment phase, communication and data transfer phase, and communication session termination phase.

The first phase involves establishing a single session between two nodes as shown in Fig. 1a. A *Connector* and a *Listener* initiate the process by creating their respective objects. The *Listener* calls *wait_for_connection()*, while the *Connector* calls *connect()*, which are both blocking functions, to establish a communication session. The second phase is for data transfer between the *Listener* and the *Connector* as shown in Fig. 1b. For this purpose, we use *read()* and *write()*, which encapsulate the application layer protocol's data transfer. The final phase is disconnecting the session, using *close()*, as shown in Fig. 1c. This *close()* function sends a disconnection request to the other node and releases the resources associated with the session.

3.2 Interoperability Support

Our proposed API is designed to be applied to various communication models. We apply our API to two representative communication models that are most commonly used in the IoT and distributed CPS.

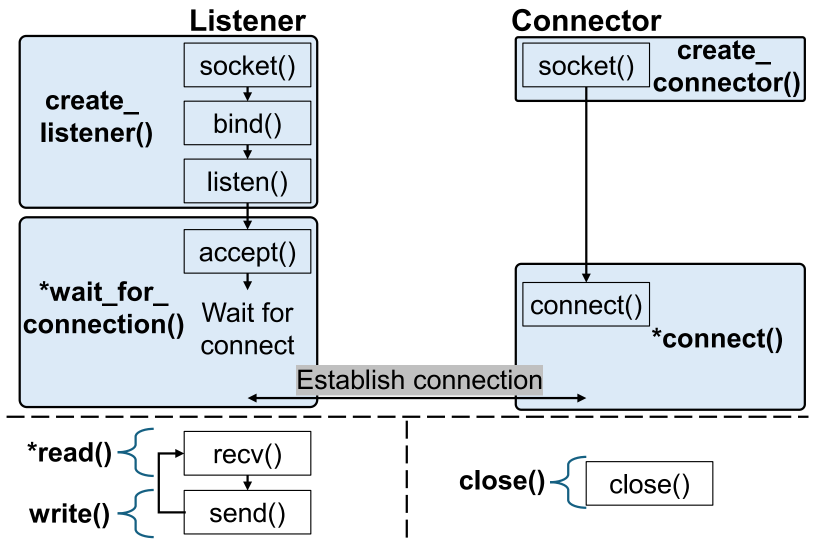


Fig. 2: Proposed API functions. Functions with * are blocking functions. These seven functions are used for the following three communication phases of networked nodes: (a) Communication session establishment (initialization) phase, (b) communication (data transfer) phase, and (c) communication session termination (closing) phase.

Point-to-Point Communication Fig. 2 shows how our API functions work for point-to-point communication, for example, a client-server model such as TCP or FTP.

Here, we take TCP as an example. A server and a client in TCP correspond to a *Listener* and a *Connector* in our proposed API, respectively. On the server side, our API function *create_listener()* handles the server socket creation, binding, and listening of the socket. The API function *wait_for_connection()* waits and accepts a client's request with blocking. On the client side, our API function *create_connector()* initializes the client socket, and another API function *connect()* calls the TCP socket function, connect(), which establishes a TCP connection with the server. After the session is set up, API functions *read()* and *write()* perform TCP socket functions recv() and send() to transfer data. Finally, *close()* is used to shut down and close the TCP sockets, disconnecting the server and the client.

Publish-Subscribe Communication Our proposed API design also supports pub-sub communication as illustrated in Fig. 3. The main challenge in adopting pub-sub into our API is the session establishment. For the discussion in this paper, we call an application running on nodes of an distributed system running on the edge a *federation*. We call a unique identifier of a federation *federationID*. In our API, we assume that all distributed nodes know *federationID* as well as

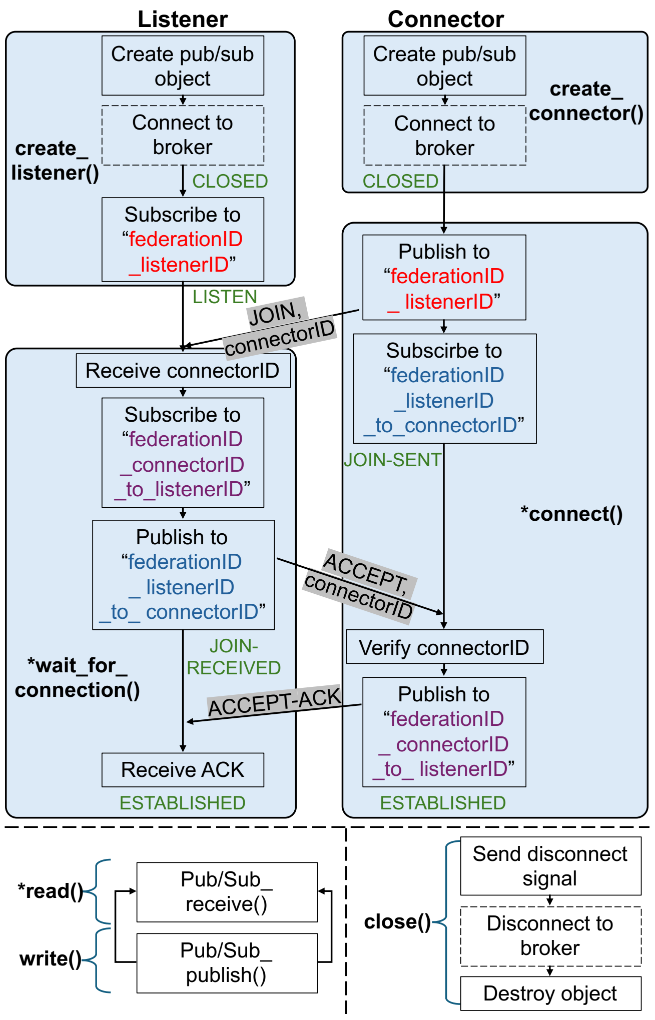


Fig. 3: Execution model of the proposed API for publish-subscribe communication. Blocking functions are marked *.

the IP address/port number of the centralized coordinator before the federation starts. Prior arts such as Robot Operating System 1 (ROS1) [42] are based on a similar assumption; for example, the ROS Master's IP address and port number are known to all nodes before the runtime starts.

As shown in Fig. 3, *create_listener()* and *create_connector()* create *Listener* and *Connector* nodes with an initialized pub/sub object. If the underlying pub-sub communication uses a centralized third party message broker such as the MQTT [1] broker, it will also connect to the broker. It is important to note that this broker is not considered a *Listener* or *Connector* node, as it is a feature of the pub-sub protocol itself rather than part of our design. Therefore, this connection step is bypassed in decentralized pub-sub protocols like Data Distribution Service (DDS) [37]. Both *Listener* and *Connector* start from a CLOSED state. The *Listener* node additionally subscribes to a topic named "*federationID_listenerID*" and enters a LISTEN state.

After that, the *Listener* and the *Connector* establish a session through a three-way handshake. First, the *Connector* node publishes to the topic "*federationID_listenerID*", sending a JOIN message with its *connectorID*. The *listenerID* should be known to the *Connector*, such as the topic name should be known for broadcasting communication. The *Connector* also subscribes to the topic "*federationID_listenerID*_to_*connectorID*". The *Connector* enters a JOIN-SENT state.

The *Listener* node receives the *connectorID*, and subscribes to the topic "*federationID_connectorID*_to_*listenerID*". Then it publishes an ACCEPT message and its *listenerID* to the topic "*federationID_listenerID*_to_*connectorID*". The *Listener* enters a JOIN-RECEIVED state.

The *Connector* verifies the received *connectorID*, publishes an ACCEPT-ACK message to "*federationID_connectorID*_to_*listenerID*" which is the specific topic for the *Listener*, and enters an ESTABLISHED state. The *Listener* finally receives the ACCEPT-ACK message and also enters an ESTABLISHED state. Finally, there are two topics for one *Connector-Listener* session, which are one-way for each.

The *read()* operation is a synchronous call, causing the node's execution to block until data is received from the corresponding *write()* operation on another node. The *close()* sends disconnect signals to the other node, disconnects to the broker if exists, and destroys the pub/sub object.

3.3 Security Support

For the authentication of each node, we assume that there is a key distribution center (KDC), which is a trusted third party nor Listener or Connector. KDC is only responsible for generating and distributing encryption keys to multiple *Listeners* and *Connectors* over a network, and do not directly participate in the system's communication. For example, a well-known network authentication protocol, Kerberos [45], uses a centralized authentication server (AS) as its KDC. The KDC issues tickets containing encrypted session keys, which are symmetric

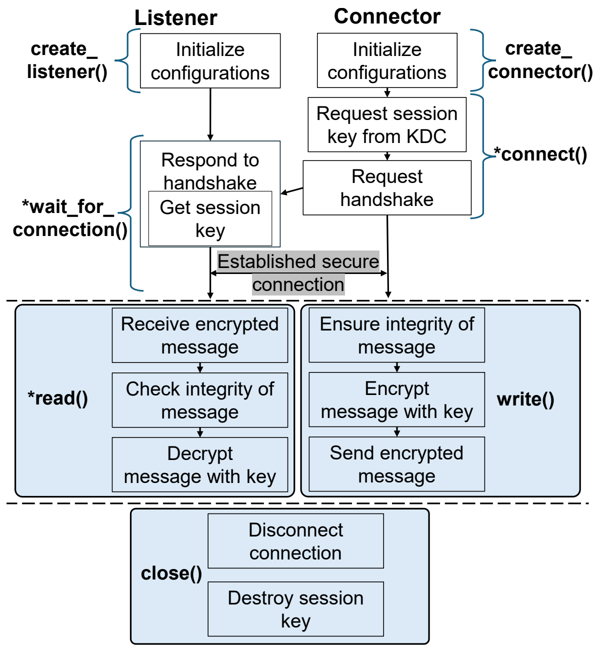


Fig. 4: Execution model of the proposed API with security enabled. Functions with * are blocking functions.

cryptographic keys as a common secret between two nodes, allowing nodes to authenticate and establish secure communication.

Fig. 4 illustrates an overview of how our API design seamlessly applies security. The *create_listener()* and *create_connector()* first initialize the *Listener* and *Connector* object with configurations, including any certificates necessary for authentication with the KDC. When *connect()* is invoked, the *Connector* requests a session key from the KDC and sends a connection request to the *Listener*. Next, the *Listener* will call *wait_for_connection()* and wait for a connection request from the *Connector*. When the request arrives, the *Listener* must obtain the same session key from the KDC to ensure secure communication with the *Connector*. The connection request includes a handshake to ensure that both ends have the same session key and session nonce to encrypt the traffic.

When security is enabled, the *write()* function internally encrypts the message to ensure confidentiality and adds authentication methods, such as Message Authentication Codes (MAC), to maintain message integrity. The *read()* function checks the integrity of received messages through the MAC and decrypts the message content using the session key. Finally, *close()* disconnects the connection and cleans up the resource related to the secure session, such as the session key.

4 A Case Study: Design and Implementation

This section provides a detailed overview of the design and implementation for a case study of the proposed API design, including example code. For efficient implementation, we leverage open-source software libraries and runtime.

4.1 Open-Source Software Used for Implementation

We leverage existing open-source software projects for our case study. For the communication and coordination among distributed nodes in the IoT or distributed CPS, we use an open-source coordination language and runtime, Lingua Franca (LF) [30], and for security support, Secure Swarm Toolkit (SST) [21].

Lingua Franca Lingua Franca (LF) is a coordination language designed to guarantee deterministic concurrency with *reactors* [32]. Reactors are lightweight concurrent entities that communicate with each other via timestamped messages. LF's C-runtime supports *federated execution* [6], allowing reactors to run across distributed systems and communicate over networks. The resulting networked system is called a *federation*, with each individual component known as a *federate*. The LF compiler generates separate code for each federate.

LF also provides a separate run-time infrastructure (RTI), which manages time synchronization among federates, ensuring a deterministic message flow and coordinating startup and shutdown. RTI also facilitates message exchange among federates, working as a message broker in *centralized coordination*.

Secure Swarm Toolkit The Secure Swarm Toolkit (SST) is an open-source toolkit framework designed to offer robust authorization and authentication mechanisms for distributed environments. The local entity *Auth* [22], provides authentication and authorization for its locally registered entities, and also supports resilience to migrate trust to another Auth [20]. SST supports a C API [17] designed to support resource-constrained devices. It has also been used for access control of decentralized and distributed file systems [15].

4.2 Software Design Considerations

We chose LF as the runtime environment of our target systems primarily due to LF's compatibility with a wide range of embedded platforms [14], including Arduino [5], RP2040 [44], Zephyr [29], and also bare metal devices. However, LF has some limitations on network communication, relying on TCP sockets for message exchanges between RTI and federates. This reliance on TCP restricts interoperability, especially in heterogeneous environments.

For authentication of the federates, LF has two options, basic *federationID* check, and key-hashed message authentication code (key-hashed MAC or HMAC) authentication. As default, LF currently supports the basic *federationID* check when a federate joins a federation although there is no encryption or message integrity involved. A federate sends its *federationID* in plaintext to the RTI, which verifies if the *federationID* is correct. However, this allows malicious entities to join the federation if they can eavesdrop the *federationID* sent over the network. To address this vulnerability, HMAC is used to secure the connection between devices [18], but LF still does not provide message encryption over the network.

To address the aforementioned security gaps, we integrated SST into LF. SST is well-suited for large-scale distributed environments, with the Auth providing decentralized authentication and authorization. SST's flexibility in supporting lightweight cryptography and hash algorithms allows for the customization of security levels to meet different requirements for heterogeneous devices. The C APIs in SST made integration with LF's C runtime straightforward, thus enhancing both interoperability and security within our distributed system.

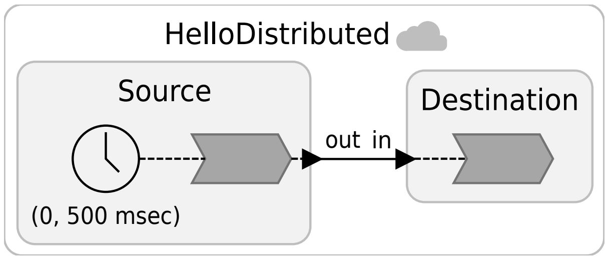
4.3 Code-Generation and Compilation

To illustrate how we enhance interoperability and security in LF, a simple LF program is organized as shown in Fig. 5. Fig. 5a represents a diagram of the example LF program where an integer value is sent from a `Source` node to a `Destination` node. Note that the diagram is automatically generated in either Visual Studio Code or Eclipse IDE after LF plug-in is installed. In Fig. 5b, the LF program named `HelloDistributed` has two reactors `Source` and `Destination`. With this LF code, LF compiler generates the executable binaries of two federates for two reactors automatically.

Line 2 indicates that the whole execution of the generated executable binaries is performed with *centralized coordination*, which means that the RTI mediates all messages between the generated federates. Line 5 sets the federate joining process to use the HMAC authentication. With *centralized coordination*, in line 12, the message sent from `Source` reactor is transferred to RTI first and then forwarded to the `out` port of the `Destination` reactor. During communication between federates, the RTI acts like a message broker, providing deterministic timing controls and traceability of all messages among all federates.

In addition to the existing target properties in LF, we newly add a property to select the underlying communication stack as part of our proposed approach. In line 3, the `comm-type` keyword enables users to specify a network protocol selectively to be used. Currently, this new feature supports three communication stacks: TCP for client-server, MQTT for pub-sub, and SST for a security model. Note that other communication protocols can be easily added to this feature based on our proposed API design.

The LF program is conditionally compiled to ensure federates install only the libraries they need, avoiding unnecessary dependencies. To enable this flexibility, LF includes a network abstraction layer called the *network driver* (netdriver), which connects low-level communication protocols to the high-level API. The netdriver's design allows for polymorphism, enabling it to switch between different communication methods without additional code changes. As shown in Fig. 6, the netdriver contains a void pointer which is cast to another struct

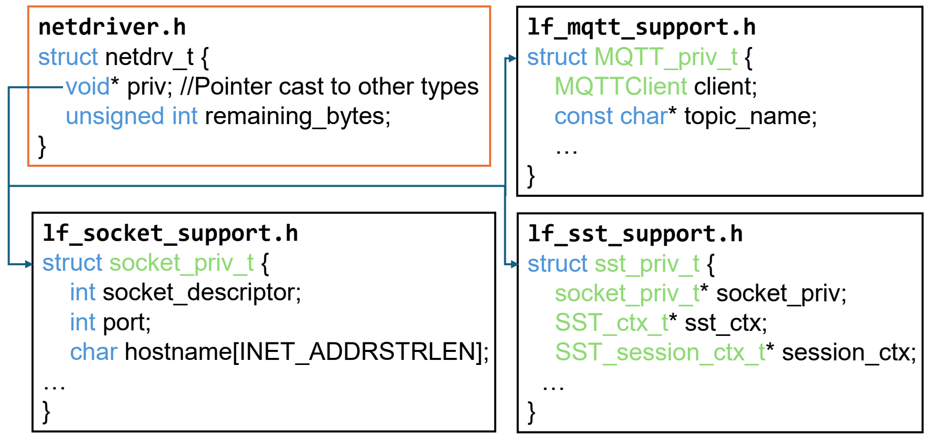


(a) Diagram of a simple federated Lingua Franca program with two federates, `Source` and `Destination`.

```
      target C {
        coordination: centralized,
        comm-type: MQTT,
        timeout: 500 sec,
        auth: true
      }

      reactor Source {
        output out: int
        timer t(0, 500 msec)
        reaction(t) -> out {=
          lf_set(out, 0);
          =}
      }

      reactor Destination {
        input in: int
        reaction(in) {=
          lf_print("Dest received: %s", in->value);
          =}
      }

      federated reactor HelloDistributed{
        s = new Source()
        d = new Destination()
        s.out -> d.in
    }
```

(b) Lingua Franca code for HelloDistributed.lf.

Fig. 5: Example of a simple Lingua Franca program.



```
netdriver.h
struct netdrv_t {
  void* priv; //Pointer cast to other types
  unsigned int remaining_bytes;
}
```

```
lf_mqtt_support.h
struct MQTT_priv_t {
  MQTTClient client;
  const char* topic_name;
  …
}
```

```
lf_socket_support.h
struct socket_priv_t {
  int socket_descriptor;
  int port;
  char hostname[INET_ADDRSTRLEN];
…
}
```

```
lf_sst_support.h
struct sst_priv_t {
  socket_priv_t* socket_priv;
  SST_ctx_t* sst_ctx;
  SST_session_ctx_t* session_ctx;
 …
}
```

Fig. 6: The structure of the *netdriver*. The void pointer in `netdrv_t` is cast to another struct depending on the network type for polymorphism.

pointer depending on the communication type. This enables providing a unified interface for multiple communication protocols. The details of *remaining_bytes* will be explained in section 4.5. To avoid confusion from the POSIX system calls, the defined APIs include a *netdrv_* prefix, such as *netdrv_read()*.

4.4 Runtime Implementation with Proposed API

We illustrate how our API deisgn supports network interoperability and security.

Client-Server Runtime We implement the client-server model using TCP sockets, which establish connections and enable data transfer between the RTI and federates. Since functions of TCP protocol can be easily mapped to the proposed API, the implementation is straightforward.

Publish-Subscribe Runtime We build the pub-sub model based on the MQTT protocol [1], utilizing the Eclipse Paho C library [10], a widely used MQTT client library for C.

The proposed three-way handshake in section 3.2, establishes a unique session for each connection. The RTI acts as the *Listener* with a designated *Listener* ID of ‘RTI’, while each federate has a unique *Connector* ID. Setting the MQTT Quality of Service (QoS) level to 2 ensures that each message is delivered exactly once. Moreover, synchronous send and receive operations are critical in LF, providing reliability to message exchanges.

Security To improve security, we integrated SST into LF. When a federate wants to establish a secure session with the RTI, it initiates the connection through *netdrv_connect()*. The federate requests for a session key from the *Auth*, which acts as a KDC. The Auth authenticates the federate through a three way handshake, and then sends out a session key. The federate now sends a connection request to the RTI sending the received session key's ID. The RTI receives the request from the federate, and requests the Auth the same session key with the matching key ID. Once the RTI and federate have the same key, they complete the handshake, establishing a secure session. Subsequent messages between the federate and the RTI is encrypted and decrypted using the session key.

4.5 Addressing Further Implementation Challenges

One of the significant challenges in implementation of our proposed API decouples network-related components that are tightly integrated with the LF code base. Many existing software platforms use a single communication model, thus, their implementation is deeply coupled with the underlying network protocol for various optimizations, for example, using the socket-level buffers for parsing network messages. Similarly, the current LF implementation is also integrated with TCP. We discuss how we tackle this problem in our implementation of the proposed API design.

To receive messages, LF uses the POSIX *read()* functions. However, in some cases, it calls *read()* multiple times to process a single message, first to retrieve the message type and then again to collect the rest of the payload. Similarly, LF sometimes sends messages in multiple chunks with separate *write()* calls. These inconsistent number of system calls for a single message increases code complexity, requiring additional logic to manage these operations. Also, it increases the chance to cause I/O errors multiple times and makes it difficult to trace errors.

The proposed *netdrv_read()* and *netdrv_write()* are designed to receive and send complete messages. It is critical to establish a clear mechanism to manage message boundaries and length. Protocols like MQTT and SST operate at the application layer and include information on the message length to ensure message boundaries are maintained. However, TCP is a stream-based transport layer protocol, meaning data is transmitted as a continuous stream of bytes without explicit message boundaries. This characteristic introduces challenges when trying to receive complete messages.

Managing message boundaries and lengths can be straightforward when each message explicitly indicates its total length. However, like other existing software platforms, not all LF messages have the payload length in their header. We classify LF messages into three types: *fixed-length*, *structured variable length*, and *variable length*, which can also be generalized for other distributed-system platforms.

Fixed-length messages have constant-sized payloads as defined by the message type. For example, MSG_TYPE_TIMESTAMP, which is one of the message types in LF for sending the start time of the RTI and federates, has a fixed payload of 8 bytes. Next, structured variable-length messages have a variable number of fixed-length structures. So, the message consists of an integer indicating the

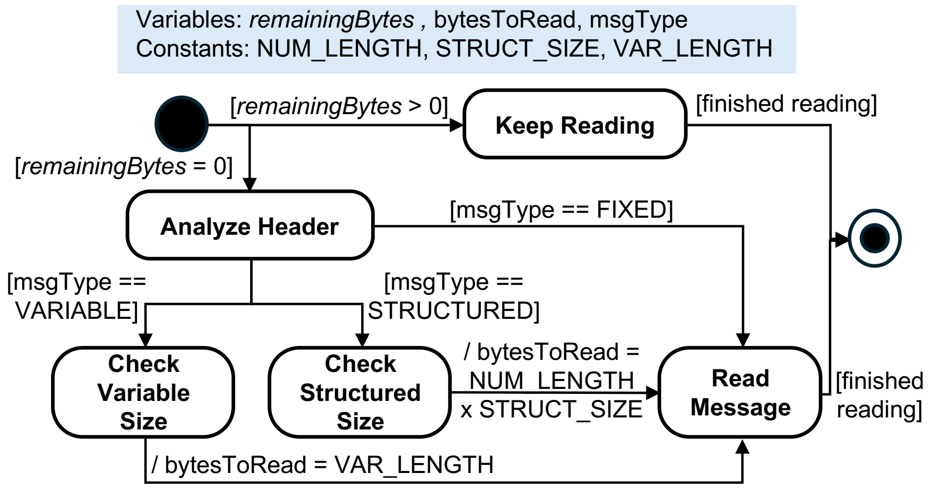


Fig. 7: The simplified state machine for the *netdrv_read()*. The boldface are the states and italic is the input.

number of structures of 'n,' followed by 'n' fixed-length structures. Finally, the variable length has an integer 'n' indicating the length of the payload, followed by the variable payload.

To address this challenge, the TCP model's *netdrv_read()* uses a state machine to handle each type of message. Fig. 7 shows a simplified version of the state machine. The *netdrv_read()* starts with an INITIAL state denoted by a filled black circle. Depending on the size indicated by *remainingBytes*, the state goes to either `Analyze Header` or `Keep Reading` state. When the *netdrv_read()* function is called and the *remainingBytes* is more than zero, the state machine enters a `Keep Reading` state. In this state, *netdrv_read()* reads as much data as the buffer allows, updating *remainingBytes*. This design provides flexibility for processing variable-length messages and reduces the risk of buffer overflows.

If the state machine enters an `Analyze Header` state, it calls the POSIX *read()* to receive only one byte to check a message type. Regarding the message type, *bytesToRead* is set to indicate the number of bytes of the payload. If the message type is a fixed-length message, it reads exactly the amount according to the message type header after entering into a `Read Message` state. For structured length messages, it calculates *bytesToRead* with `NUM_LENGTH` and `STRUCT_SIZE`, i.e., the number of structures and a struct size, respectively. For variable length messages, it first reads the integer 'n' indicating the variable length, and then reads 'n' bytes. The netdriver structure keeps track of the additional bytes to be read through an integer, *bytesToRead*, until it reaches to the end state from `Read Message` state.

5 Evaluation

In this section, we evaluate our APl's overhead in terms of communication time, binary size, and message lengths. We also clarify the strong points of the proposed API design qualitatively through a software design analysis.

5.1 Experimental Setup

An IoT or distributed CPS that runs the LF program in Fig. 5, where the federate `Source` sends a message to the federate `Destination` is used for the evaluation of our API. The RTI, Mosquitto message broker [27] for MQTT, and Auth for SST run on a workstation as the edge, equipped with an i9-13900 CPU and 128 GB memory. The federates are deployed on two Raspberry Pi 4 (Model B 4GB RAM) devices. The workstation and the devices communicate over Wi-Fi, and the average end-to-end round-trip latency from the Raspberry Pi to the workstation, measured by a simple ping, is 13.60 milliseconds. The federates joining the federation use HMAC authentication mode as a baseline.

5.2 Communication Time Overhead

To estimate the communication time overhead, we measure a *lag*, the time difference between the physical time and logical time. Specifically, we measure the time lapse between the system clock (physical time) and the semantic notion of intended global time agreed by all nodes (logical time) [31]. From the example above in Fig. 5b, the timer on line 10 triggers the `Source` reactor to send a message to the `Destination` reactor every 500 milliseconds. Due to the timeout in line 4, this will stop when the logical time reaches 500 seconds. We also set the timer with a period of 50 milliseconds, with a timeout of 50 seconds, to test sending messages in shorter periods. Consequently, in both cases, the message will be sent 1,000 times. We measure the average of each message's lag.

Fig. 8 shows the average lag when sending messages in a period of 500 milliseconds and 50 milliseconds. In the case when the period is 500 milliseconds and the TCP model is compared with the baseline code, there is a 0.53% increase in lag which is 0.08 milliseconds. The SST model also has a similar lag of 0.80% lag increase. When we compare the SST model to the TCP model, there is only a 0.26% increase in lag when is security enabled for authentication of federates and confidentiality/integrity of messages. The 50 millisecond period test also had a similar tendency, showing a very small overhead.

MQTT has a longer lag due to the synchronous behavior. To ensure deterministic timing, all messages in LF must be sent in order. MQTT's underlying TCP sends messages in order, but the protocol itself does not guarantee it without QoS level 2. So, the *netdrv_write()* must call MQTTClient_waitForCompletion() after publishing the message to ensure the message is delivered. For synchronous receive of incoming messages, the *netdrv_read()* calls MQTTClient_receive().

As the synchronous parts become the bottleneck, each netdrv_write() function call takes an average of 90 milliseconds to transfer a message. Due to the

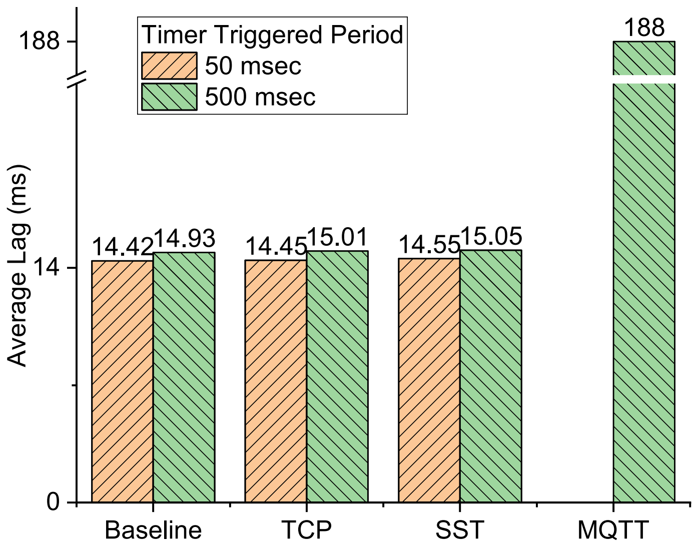


Fig. 8: Average lag compared with the baseline LF code. The federate `Source` sends messages when the timer is triggered as in line 10, with a period of 50 and 500 milliseconds.

centralized coordination, passing a message via the RTI, the netdrv_write() is called twice, sending signals from the federate `Source` to the RTI, and RTI to the federate `Destination`. This explains the average latency of 188 milliseconds. However, when using MQTT, centralized coordination becomes inefficient because the RTI acts as a message broker while MQTT itself has its own broker. To remove this inefficiency, LF supports a different mode of runtime execution called *decentralized coordination*, where federates directly communicate with each other without the RTI. We plan to support this for MQTT in the future.

5.3 Message Size Overhead

We measure communication overhead in terms of the message size of each communication type using Wireshark [7]. To be specific, we measure the size of MSG_TYPE_TAGGED_MESSAGE, which includes a header with the destination information and a variable payload. Fig. 9 illustrates the total message size when sending 4, 8, 16, 24, 32, and 48-byte long messages as payload. When this message is sent, a 21-byte header is attached, so when sending a 32-bit integer, a total of 25 bytes are sent.

Details are provided for when 4 bytes are sent. The baseline code and the client-server model is both based on TCP, so they have the same length of TCP/IP headers of 66 bytes, sending a total of 91 bytes. The message size of the baseline and client-server models both linearly increase in proportion to the

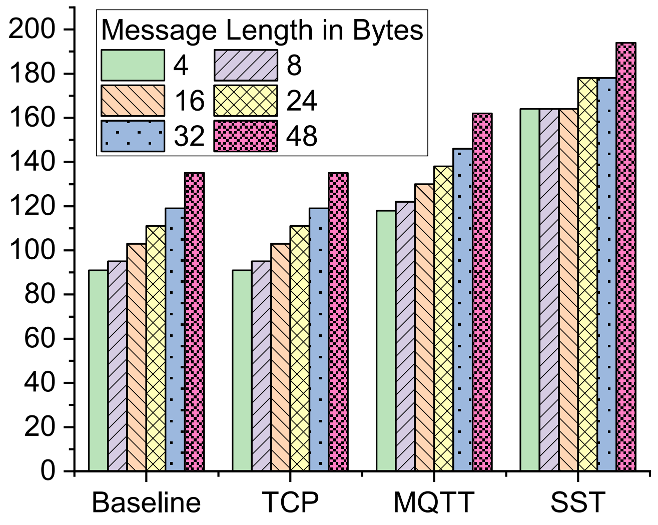


Fig. 9: Sizes of messages sent over the network in bytes for different communication types supported by the proposed API compared to the baseline implementation.

message size. The MQTT model sent 118 bytes, including 66 bytes of TCP/IP headers and 27 bytes of MQTT headers. The MQTT header size varies depending on the topic name, which is 'MQTTTest_fed0_to_RTI'.

The SST model sent a total of 164 bytes, including the TCP/IP header, SST header, and encrypted message. SST employs the AES-128-CBC mode, which is a block cipher with a fixed block size of 16 bytes. As a result, the size of the encrypted output increases in 16-byte increments, demonstrating a stepwise pattern of growth.

Note that the block cipher encryption provided by SST also prevents side-channel attacks. As described in section 4.5, LF has a message type with fixed lengths. Due to these fixed lengths, eavesdroppers can infer the message type from the message length. If the eavesdroppers can access to the trace of the messages, the message lengths can be a side channel, allowing them to determine which message types are being transmitted. By using a block cipher like AES-128-CBC, SST ensures that all encrypted messages are in block sizes, making it difficult to infer the message type.

5.4 Binary Size Overhead

To estimate the overhead in terms of binary sizes, we measure the binary size of the RTI, federate `Source`, and federate `Destination`. As illustrated in Fig. 10, the overhead introduced by the abstraction layer is minimal when compared with the baseline code. The increase in binary size for the RTI is 1.02%, 5.23%, and 1.44% for TCP, MQTT, and SST, respectively. For the same protocols, the

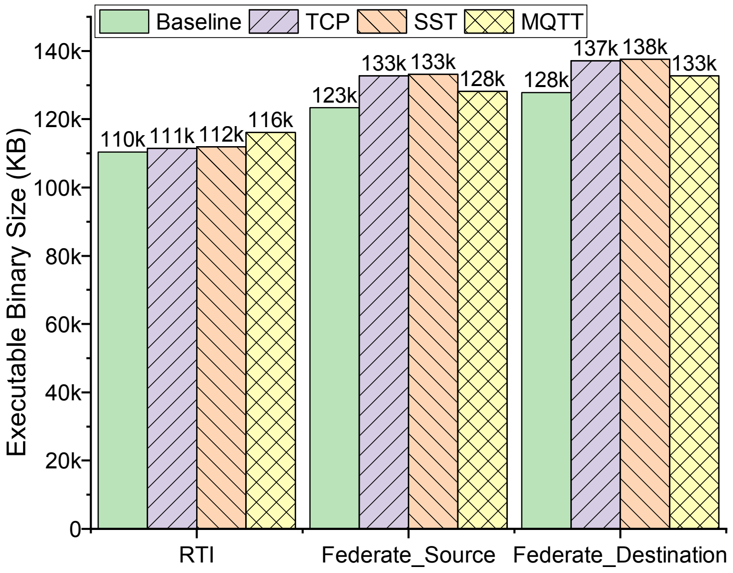


Fig. 10: Binary size in kilobytes compared with the baseline LF code.

Destination federate's binary size increases by 7.29%, 3.71%, and 7.67%, while the Source federate show a similar increase.

The increased binary size for both RTI and federates mostly comes from compilation of the network abstract layer as a separate library, making it more modular and flexible to extend. The RTI has a smaller overhead compared to the federates because it functions only as a *Listener*, whereas the federates have dual roles, serving as both *Connectors* and *Listeners* in *decentralized coordination* which has been briefly introduced in Section 5.2. In this case, the RTI does not relay messages to other federates; instead, the federates communicate directly with each other. However, the decentralized coordination of currently only supports TCP yet and supporting other protocols will be future work.

Among different modes of communication, MQTT shows the largest binary size for the RTI. However, for the federates, MQTT has smaller binary sizes than TCP and SST. Although we did not perform a detailed analysis of this difference, we speculate compiler optimizations and additional libraries caused this increase.

5.5 Software Design Analysis

Fig. 11 shows the improved software design from the previous existing LF network module. In the previous software design, as shown in the left side of Fig. 11, each federate owns a TCP socket instance to call network functions to communicate with RTI or other federates. The direct inclusion of the TCP socket instance requires a large number of code modifications in the federate to add other types of communication protocols (e.g., publish-subscribe or secure communication mode) in addition to the existing TCP protocol. The right side of Fig. 11 denotes the improved software design after applying the proposed API design, which decouples a TCP instance from a federate.

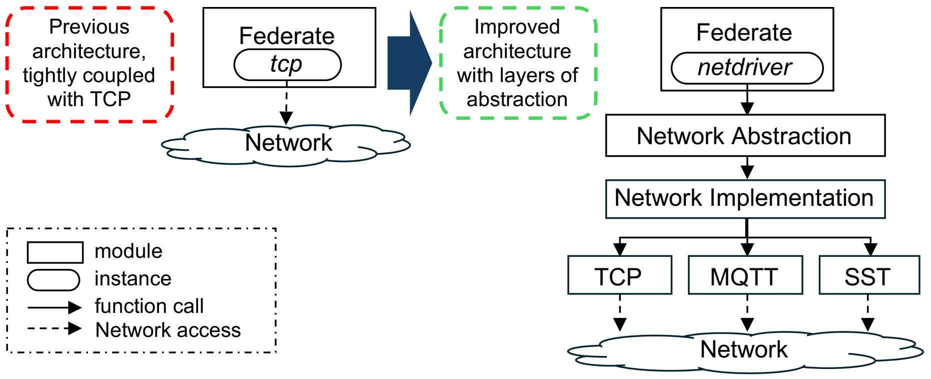


Fig. 11: Improved software architecture after applying the proposed API design to the previous LF network node.

To evaluate the effectiveness of the proposed API in terms of software design, we conduct a qualitative analysis focused on how the application of the proposed approach achieves the software design considerations in [50]. This paper applies only to qualitative parts of the design considerations, i.e., abstraction, modularity, and information hiding since quantitative evaluation using software engineering metrics is not the primary concern. Analyzing software design based on quantitative metrics such as cyclomatic complexity [34] is left to future work.

Abstraction The goal of abstraction is to concentrate only on the essential features. To achieve this, two types of abstraction, i.e., *procedural abstraction* and *data abstraction* should be achieved. *Procedural abstraction* aims to find a hierarchy in the software's control and can be achieved by decomposing the software into sub-modules step-wisely so that the abstraction draws a hierarchical structure. In this structure, the top node denotes the problem to be solved through the software. As shown in the right side of Fig. 11, our design shows the APIs to solve the network access from federate as a top node, separates implementation and protocol types as next-level nodes, and forms a hierarchical structure. Our design also separates protocol types as another layer and hides data types from a federate.

Modularity Coupling between modules should be considered to increase software modularity [50]. Six types of coupling, i.e., content coupling, common coupling, external coupling, control coupling, stamp coupling, and data coupling, need to be analyzed to evaluate the software modularity. Content coupling occurs when a module changes another module's data or control flows using jump indicators, and common coupling occurs when two modules share data. Since we separate *Network Abstraction* as the API layer and *Network Implementation* as its implementation, content and common coupling are avoided. External coupling occurs when modules communicate through an external medium, such as a file, and control coupling occurs when one module sets control flags that are reacted to by the dependent modules. Neither files nor control flags are used as arguments in our API design. Either stamp or data coupling occurs by passing complete data structures or simple data between modules. Our API design also does not pass any data between modules, avoiding all six types of coupling.

Information Hiding The key to implementing information hiding is to hide design decisions and details from other modules. By designing each selection of a communication protocol as an option *comm-type* in Fig. 5b and implementing each protocol as a separate C source file, we can hide all design decisions and implementation details of each communication protocol into each source file. This design also enables developers to easily add other communication protocols with a simpler, easier-to-maintain, and more robust interface.

6 Conclusion and Future Work

In this paper, we perform a case study of the design of an API and its runtime to achieve network interoperability and security in the IoT and distributed CPS. The proposed API encapsulates the underlying network implementation layer through seven key API functions. We implement our approach using open-source software, Lingua Franca, and SST as a case study. The evaluation assesses the proposed API design using our case-study implementation by measuring the communication time overhead, message size, and binary size, showing a minimal overhead. The proposed API and its implementation will be available on GitHub.

As future work, we plan to support more communication modes in our API. We also plan to allow distributed nodes with different communication modes to join a single federation. In this case, we envision that a centralized entity such as RTI will be able to handle multiple protocols in one process. Security options for communication among distributed nodes can be extended to enable fine-grained configurations in our proposed API.

Acknowledgment

This work was supported in part by the NSF I/UCRC for Intelligent, Distributed, Embedded Applications and Systems (IDEAS) and from NSF grant #2231620. This work was supported in part by ATTO Research.

References

1. MQTT Version 3.1.1. OASIS Standard (2014), https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html
2. Abounassar, E.M., El-Kafrawy, P., Abd El-Latif, A.A.: Security and interoperability issues with Internet of things (IoT) in healthcare industry: A survey. Security and privacy preserving for IoT and 5G networks: techniques, challenges, and new directions pp. 159–189 (2022)
3. Albouq, S.S., Sen, A.A.A., Almashf, N., Mohammad Yamin, A.A., Bahbouh, N.M.: A survey of interoperability challenges and solutions for dealing with them in IoT environment. IEEE Access **10**, 36416–36428 (2022)
4. Amjad, A., Azam, F., Anwar, M.W., Butt, W.H.: A systematic review on the data interoperability of application layer protocols in industrial IoT. Ieee Access **9**, 96528–96545 (2021)
5. Arduino: Arduino: Open-source electronics platform (2005), https://www.arduino.cc/
6. Bateni, S., Lohstroh, M., Wong, H.S., Kim, H., Lin, S., Menard, C., Lee, E.A.: Risk and mitigation of nondeterminism in distributed cyber-physical systems. In: Proceedings of the 21st ACM-IEEE International Conference on Formal Methods and Models for System Design. pp. 1–11 (2023)
7. Beale, J., Orebaugh, A., Ramirez, G.: Wireshark & Ethereal network protocol analyzer toolkit. Elsevier (2006)
8. Carvalho, G., Cabral, B., Pereira, V., Bernardino, J.: Edge computing: current trends, research challenges and future directions. Computing **103**(5), 993–1023 (2021)
9. Costin, A., Eastman, C.: Need for interoperability to enable seamless information exchanges in smart and sustainable urban systems. Journal of Computing in Civil Engineering **33**(3), 04019008 (2019)
10. Eclipse Foundation: Eclipse Paho C Client Library (2009), available at https://github.com/eclipse/paho.mqtt.c
11. Giang, N.K., Lea, R., Blackstock, M., Leung, V.C.: Fog at the edge: Experiences building an edge computing platform. In: International Conference on Edge Computing (EDGE). pp. 9–16. IEEE (2018)
12. Gomez, D.L., Montoya, G.A., Lozano-Garzon, C., Donoso, Y.: Strategies for assuring low latency, scalability and interoperability in edge computing and TSN networks for critical IIoT services. IEEE Access **11**, 42546–42577 (2023)
13. Gürdür, D., Asplund, F.: A systematic review to merge discourses: Interoperability, integration and cyber-physical systems. Journal of Industrial information integration **9**, 14–23 (2018)
14. Jellum, E.R., Lin, S., Donovan, P., Soyer, E., Shakir, F., Bryne, T., Orlandic, M., Lohstroh, M., Lee, E.A.: Beyond the threaded programming model on real-time operating systems. In: Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2023). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2023)
15. Jo, Y., Cho, Y., Kim, H.: Secure and lightweight access control for highly decentralized and distributed file systems. In: Proceedings of the 1st International Workshop on Middleware for the Computing Continuum. pp. 1–6 (2023)
16. Kannan, B.M., Solainayagi, P., Azath, H., Murugan, S., Srinivasan, C.: Secure communication in IoT-enabled embedded systems for military applications using encryption. In: 2nd International Conference on Edge Computing and Applications (ICECAA). pp. 1385–1389. IEEE (2023)

17. Kim, D., Jo, Y., Kim, T., Kim, H.: SST v1.0.0 with C API: Pluggable security solution for the Internet of Things. SoftwareX **22**, 101390 (May 2023). https://doi.org/10.1016/j.softx.2023.101390
18. Kim, D., Kim, H.: Poster abstract: Securing edge-based real-time IoT systems. In: Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems. p. 544–545. SenSys '23 (2024). https://doi.org/10.1145/3625687.3628408
19. Kim, H.: Secure programming platform for edge-based IoT: Wild-and-crazy-idea paper. In: 2023 Forum on Specification & Design Languages (FDL). pp. 1–4. IEEE (2023)
20. Kim, H., Kang, E., Broman, D., Lee, E.A.: Resilient authentication and authorization for the Internet of Things (IoT) using edge computing. ACM Trans. Internet Things **1**(1) (mar 2020). https://doi.org/10.1145/3375837
21. Kim, H., Kang, E., Lee, E.A., Broman, D.: A toolkit for construction of authorization service infrastructure for the Internet of Things. In: The 2nd ACM/IEEE International Conference on Internet-of-Things Design and Implementation. pp. 147–158. Pittsburgh, PA (Apr 2017)
22. Kim, H., Wasicek, A., Mehne, B., Lee, E.A.: A secure network architecture for the Internet of Things based on local authorization entities. In: The 4th IEEE International Conference on Future Internet of Things and Cloud (FiCloud). p. 114–122. Vienna, Austria (Aug 2016)
23. Koo, J., Kim, Y.G.: Interoperability requirements for a smart city. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing. pp. 690–698 (2021)
24. Lai, X., Fan, L., Lei, X., Deng, Y., Karagiannidis, G.K., Nallanathan, A.: Secure mobile edge computing networks in the presence of multiple eavesdroppers. IEEE Transactions on Communications **70**(1), 500–513 (2021)
25. Lee, E., Seo, Y.D., Oh, S.R., Kim, Y.G.: A survey on standards for interoperability and security in the internet of things. IEEE Communications Surveys & Tutorials **23**(2), 1020–1047 (2021)
26. Li, B., Dong, W.: Edge-centric programming for IoT applications with automatic code partitioning. IEEE Transactions on Computers **71**(10), 2408–2422 (2021)
27. Light, R.A.: Mosquitto: server and client implementation of the MQTT protocol. Journal of Open Source Software **2**(13), 265 (2017)
28. Lin, H., Zeadally, S., Chen, Z., Labiod, H., Wang, L.: A survey on computation offloading modeling for edge computing. Journal of Network and Computer Applications **169**, 102781 (2020)
29. The Linux Foundation: Zephyr RTOS (2016), https://www.zephyrproject.org/
30. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a lingua franca for deterministic concurrent systems. ACM Transactions on Embedded Computing Systems (TECS) **20**(4), 1–27 (2021)
31. Lohstroh, M., Menard, C., Schulz-Rosengarten, A., Weber, M., Castrillon, J., Lee, E.A.: A language for deterministic coordination across multiple timelines. In: 2020 Forum for Specification and Design Languages (FDL). pp. 1–8. IEEE (2020)
32. Lohstroh, M., Romeo, Í.Í., Goens, A., Derler, P., Castrillon, J., Lee, E.A., Sangiovanni-Vincentelli, A.: Reactors: A deterministic model for composable reactive systems. In: Cyber Physical Systems. Model-Based Design: 9th International Workshop, CyPhy 2019, and 15th International Workshop, WESE 2019, New York City, NY, USA, October 17-18, 2019, Revised Selected Papers 9. pp. 59–85. Springer (2020)
33. Margaria, T., Chaudhary, H.A.A., Guevara, I., Ryan, S., Schieweck, A.: The interoperability challenge: Building a model-driven digital thread platform for CPS. In:

Leveraging Applications of Formal Methods, Verification and Validation. ISoLA 2021. Lecture Notes in Computer Science. pp. 393–413. Springer (2021)
34. McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering (4), 308–320 (1976)
35. Ning, H., Li, Y., Shi, F., Yang, L.T.: Heterogeneous edge computing open platforms and tools for Internet of things. Future Generation Computer Systems **106**, 67–76 (2020)
36. Noura, M., Atiquzzaman, M., Gaedke, M.: Interoperability in Internet of Things: Taxonomies and open challenges. Mobile networks and applications **24**, 796–809 (2019)
37. Object Management Group (OMG): Data Distribution Service (DDS) Version 1.4. OMG Specification formal/2015-04-10, Object Management Group (OMG) (2015), https://www.omg.org/spec/DDS/1.4/PDF
38. Oh, S.R., Koo, J., Kim, Y.G.: Security interoperability in heterogeneous IoT platforms: threat model of the interoperable OAuth 2.0 framework. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. pp. 22–31 (2022)
39. OPC Foundation: The OPC Unified Architecture (UA) (2008), available at https://opcfoundation.org/about/opc-technologies/opc-ua/
40. Pereira, P.H.M., Cainelli, G., Pereira, C.E., Costa, J.P.J.D., Freitas, E.P.D.: An interoperability middleware for IIoT. In: International Symposium on Industrial Electronics (ISIE). pp. 1–6. IEEE (2023)
41. Quadir, M.S.E., Chandy, J.A.: Embedded systems authentication and encryption using strong PUF modeling. In: 2020 IEEE International Conference on Consumer Electronics (ICCE). pp. 1–6. IEEE (2020)
42. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y., et al.: ROS: an open-source robot operating system. In: ICRA workshop on open source software. vol. 3, p. 5. Kobe, Japan (2009)
43. Rana, B., Singh, Y., Singh, P.K.: A systematic survey on internet of things: Energy efficiency and interoperability perspective. Transactions on Emerging Telecommunications Technologies **32**(8), e4166 (2021)
44. Raspberry Pi Foundation: RP2040 Microcontroller Datasheet (2021), https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdft
45. Saltzer, J.H., Reed, D.P., Clark, D.D.: End-to-end arguments in system design. ACM Transactions on Computer Systems (TOCS) **2**(4), 277–288 (1984)
46. Shapsough, S., Aloul, F., Zualkernan, I.A.: Securing low-resource edge devices for IoT systems. In: International symposium in sensing and instrumentation in IoT Era (ISSI). pp. 1–4. IEEE (2018)
47. Sigwele, T., Hu, Y.F., Ali, M., Hou, J., Susanto, M., Fitriawan, H.: An intelligent edge computing based semantic gateway for healthcare systems interoperability and collaboration. In: 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud). pp. 370–376. IEEE (2018)
48. S´anchez, S.M., Lecumberri, F., Sati, V., Arora, A., Shoeibi, N., Rodr´ıguez, S., Rodr´ıguez, J.M.C.: Edge computing driven smart personal protective system deployed on NVIDIA Jetson and integrated with ROS. In: International Conference on Practical Applications of Agents and Multi-Agent Systems. pp. 385–393. Springer (2020)
49. Tedeschi, P., Sciancalepore, S.: Edge and fog computing in critical infrastructures: Analysis, security threats, and research challenges. In: 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 1–10. IEEE (2019)
50. van Vliet, H.: Software Engineering: Principles and Practice. Wiley (2008)

51. Xiao, Y., Jia, Y., Liu, C., Cheng, X., Yu, J., Lv, W.: Edge computing security: State of the art and challenges. Proceedings of the IEEE **107**(8), 1608–1631 (2019)
52. Yu, W., Liang, F., He, X., Hatcher, W.G., Lu, C., Lin, J., Yang, X.: A survey on the edge computing for the Internet of Things. IEEE access **6**, 6900–6919 (2017)
53. Zeyu, H., Geming, X., Zhaohang, W., Sen, Y.: Survey on edge computing security. In: 2020 International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE). pp. 96–105. IEEE (2020)
54. Zhang, J., Keramat, F., Yu, X., Hernández, D.M., Queralta, J.P., Westerlund, T.: Distributed robotic systems in the edge-cloud continuum with ROS 2: a review on novel architectures and technology readiness. In: 2022 Seventh International Conference on Fog and Mobile Edge Computing (FMEC). pp. 1–8. IEEE (2022)