Research Article GU J Sci 35 (2): 464-483 (2022) DOI: 10.35378/gujs.803726

Gazi University

Journal of Science

http://dergipark.gov.tr/gujs

A New Multi-Target Compiler Architecture for Edge-Devices and Cloud Management

Erhan GOKCAY *

Atılım University, Software Engineering Department, Kızılcasar Mahallesi, 06830 Incek Golbasi, Ankara, Turkey

Highlights

- Distribution of a computation from a single source to multiple edge-devices.
- Combining multi-target sources in a single project.
- Lightweight and precise control of edge-devices.

Article Info

Received: 02 Oct 2020
Accepted: 18 Apr 2021

Keywords

Edge-device programming
Internet of things
Multi-target compiler
Cloud management

Abstract

Edge computing is the concept where the computation is handled at edge-devices. The transfer of the computation from servers to edge-devices will decrease the massive amount of data transfer generated by edge-devices. There are several efficient management tools for setup and connection purposes, but these management tools cannot provide a unified programming system from a single source code/project. Even though it is possible to control each device efficiently, a global view of the computation is missing in a programming project that includes several edge-devices for computation and data analysis purposes, and the devices need to be programmed individually. A generic workflow engine might automate part of the problem using standard interfaces and predefined objects running on edge-devices. Nevertheless, the approach fails in fine-tuning each edge-device since the computation cannot be moved easily among devices. This paper introduces a new compiler architecture to control and program edge-devices from a single source code. The source code can be distributed to multiple edge-devices using simple compiler directives, and the transfer and communication of the source code with multiple devices are handled transparently. Fine-tuning the source code and code movement between devices becomes very efficient in editing and time. The proposed architecture is a lightweight system with fine-tuned computation and distribution among devices.

1. INTRODUCTION

Cloud systems are improving themselves, and the community becomes aware of several advantages of these systems. One of the advantages is the reduced cost for the same computation and system. The performance benefit of the cloud system is also essential with the increased d,iğistributed computational capability. Edge computing is another research area where there are fundamental differences compared to cloud computing. Cloud computing is trying to remove the computation from the end-user, whereas edge computing focuses on moving the calculation to the edges, i.e., devices that collect the data. Edge-computing is promoting a local computation. This approach has several advantages, one of them being a reduced transmission load and latency. Edge computing applications can use local calculations and improve the rate of moving data from/to devices [1-5]. On the other hand, software developers do not have a unified tool to fine-tune edge applications [6-8].

Several papers analyze research opportunities in edge-computing. At the same time, edge development is considered one of the difficult challenges and problems in computing. A direct communication method is used in many models like P2P, which is based on exchanging messages. The development is per component in these models, and functionalities and communication are coded for any distributed component [6, 8-9]. To succeed in this model, each execution platform may need resources that are fixed.

* e-mail: erhan.gokcay@atilim.edu.tr

The research question is how we can remove the code's dependency and user from edge-device and make the process transparent. This paper presents a novel and lightweight architecture for developing reliable and efficient programs to be deployed in mixed-edge device environments.

Chapter 2 discusses previous work. Chapter 3 describes the need for edge-management, and chapter 4 describes the multi-target compiler architecture. In chapter 5, the compilation and execution stages are explained. Chapter 6 describes the state diagrams and the complete flow of the process. Chapter 7 describes several implementation details of the prototype, and sample input and output codes are also given. Chapter 8 gives a comparison table and discussion, including future research directions. Finally, chapter 9 summarizes the advantages of the multi-target compilation.

2. RELATED WORK

There are several models proposed in recent years, like microservices [10]. Microservices are designed to coordinate different computing devices. There are several problems with the proposed architecture. A pre-deployment of all executable resources is necessary for the model. The setup of each edge environment is different. This fact makes the usage of any standard very difficult, if not impossible. Therefore, for an edge application to be robust, it should be able to utilize different runtime environments simultaneously. This is also needed to be efficient as well. In another model [11], application replicability is identified, where recurring patterns are identified, and different components' reusability is utilized. Still, a global view is missing in the model. Although a global view is missing, this model can be utilized in our approach. A lightweight named object (LNO) solution is described where the solution depends on an ICN-based abstraction. It can provide IoT management and programmability with a flexible interface without any persistent communication links or bindings [12]. Still, the programming concept is per device without any global view. A formal approach is defined to create an entity service working at the edge-device [13]. Although the approach can parse user requirements and take the necessary action to detect or modify the physical entities' status information, the approach still depends on single devices. A combined approach is still missing. Another system defines a distributed and flexible protocol to create a location-independent identifier [14]. Another defined framework is the closest study for our proposed solution [15]. The model depends on Intents that are distributed to the cloud. It needs a predefined control and monitor tasks, and these need to be provided in the domain library. Individual control of each edge-device is not provided in the solution. General abstraction layers are also given without any details [16]. The work uses microservices described above, but it provides Technical-Units and Development-Units, and it keeps track of these units to ease the development lifecycle [17]. A new architectural model is defined, which is derived from current standards [18]. On top of that, there are several additions like "IoT Data Flow Dynamic Routing Entity," "IoT Topology Management Entity," and "IoT Visualization Entity." An object representation is proposed, which creates an abstract view of the edge devices and simplifies the development [19]. In another study, prototyping tools and virtualization techniques are summarized and reviewed [20]. All these tools help the development but controlling the whole project from a single project or source code is not available.

Several review papers explain development tools [7, 21-24]. Several tools are compared, and the problems are summarized [22]. The problems exist in several categories: standard interfaces, heterogeneous environments, awareness of the context, middleware, node identity, fault tolerance, and energy management. Most of the solutions depend on isolating the edge-device from the picture and create objects or interfaces so that the programmer can ignore the details of the edge-device. In many cases, this approach increases productivity, but it also removes the opportunity to fine-tune the computation among devices. The edge-devices are reviewed in terms of technology rather than development tools [24]. The secure update protocols and tools are also reviewed [25]. This is another problem, but it is not related to our research question. The operating systems running on edge-devices are reviewed [26]. Although an OS simplifies the operation using threads and other interfaces, it puts an extra burden on the device, and for simple edge-devices, programming may be impossible. The MOLE compiler using micro-services architecture is defined and given [27]. Fine-tuning may not be available because the code distribution depends on microservices' functionality, and individual edge-device access may not be available. The taxonomy, standardization issues, and networking are reviewed [28]. In contrast, another paper reviews the issue in terms of domains [29]. Another paper reviews edge-devices in terms of architecture layers [30].

The approach may simplify the programming problems but putting extra layers on low-capacity devices creates other performance problems. A different approach reviews sensing as a paradigm, but it fails to provide a solution to our research question [31]. An Attribute-driven design is also proposed [32]. A different approach is also presented, where the programming model distributes individual instructions to edge-devices instead of distributing the functionality [33].

3. EDGE-DEVICE MANAGEMENT AND PROGRAMMING

Because of the increased deployment of IoT devices, there is a massive amount of data obtained from devices. The data is usually transferred to servers to be processed. The edge computing concept is trying to minimize the data movement and the computational load on servers to reduce the data transfer problems. The processing is moved to edge-devices as much as possible to accomplish these goals, where the data is collected.

From the management perspective, there are two main issues here. One of them is deployment and setup, and the other is to program the edge device. There are many tools to install, setup, and manage edge devices like Microsoft Azure and others to fulfill the first requirement.

In terms of programming, each edge-device program should be designed and programmed one-by-one in most cases. The system is given in Figure 1. Individual compilation for different architectures poses no problem since there are compilers for any typical architecture and processor/controller. The data transfer to each device needs a different design as well. The computation should be divided between edge devices and servers manually. The development and deployment of these modules are independent of each other. In this model, the computation's continuous design between devices is difficult to manage because of several independent phases.

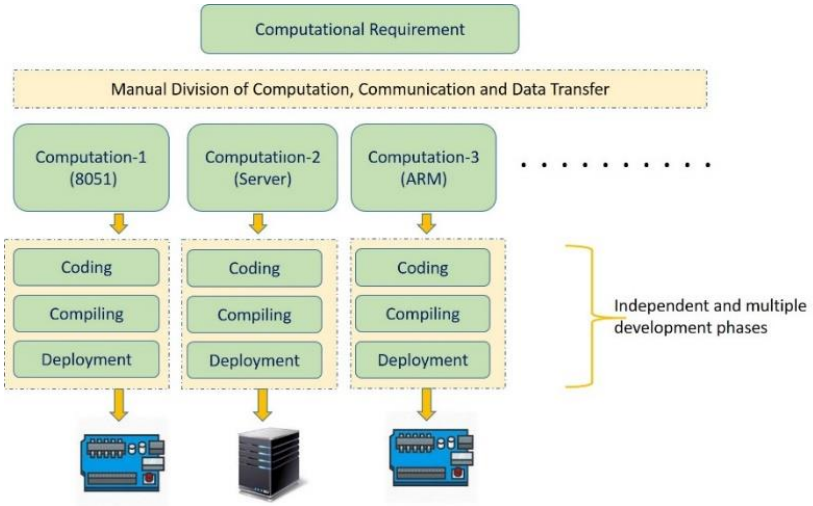

Figure 1. *A common computational structure*

The missing link is a global view and management of the computation among the edge devices and servers. To streamline a computation among different architectures and devices, we need an automatic separation, compilation, and execution paradigm from a single source. A block view of the proposed paradigm is given in Figure 2. The proposed model gives a flexible and powerful approach to program heterogeneous edge-devices from a single project/source code. Individual programming and fine-tuning are implemented, and at the same time, edge-device listeners provide device-specific information during coding, compilation, and execution.

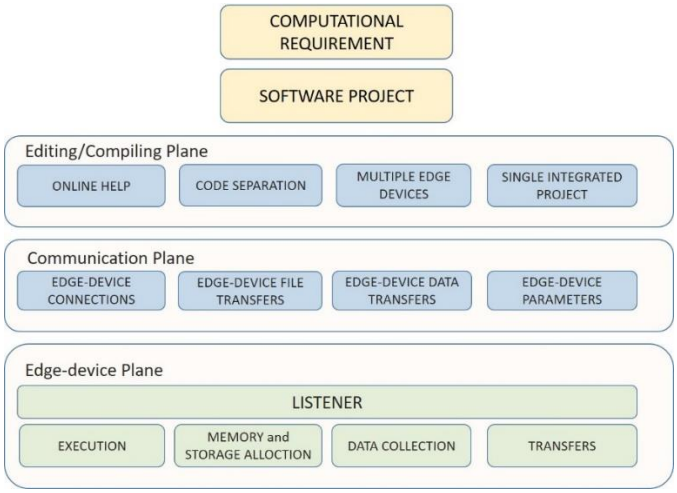

Figure 2. *Proposed architecture*

An example view of the single-source multiple-target programming is given in Figure 3. In this paradigm, the code line(s) can be moved from one edge device code section to another by modifying a single source code. The remaining process is transparent. In this approach, the user can concentrate on the source code's functionality instead of communicating with the edge device, designing individual programs for each device, and transferring the data.

Single source code is a symbolic representation, and it means that there is a single project defined for all edge-devices and servers. Obviously, the project may consist of several source codes, which is typical in Object-Oriented Development.

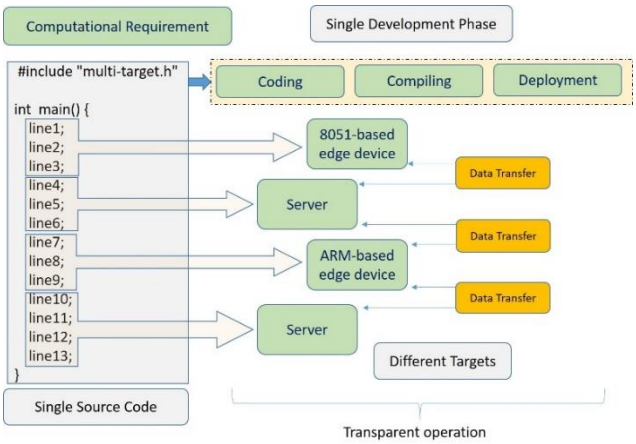

Figure 3. *Sample source code*

The data transfer that may be needed between devices is handled in the background during the execution. The Listener programs are written and transferred to each edge device only once. Compiled pieces of code are transferred to each device again only once unless they are modified. By handling all data transfer, code separation, compilation, code transfer, and execution transparently, the user can focus on the computation functionality.

4. MULTI-TARGET EDGE DEVICE COMPILER ARCHITECTURE

A Multi-Target Edge Device compiler (*MTED*) is proposed and implemented to provide the proposed functionality. The *MTED* compiler is a framework to create/compile, and execute a program where the computation is distributed between several edge devices and servers transparently. The *MTED* compiler architecture consists of several components, which are given in Figure 4.

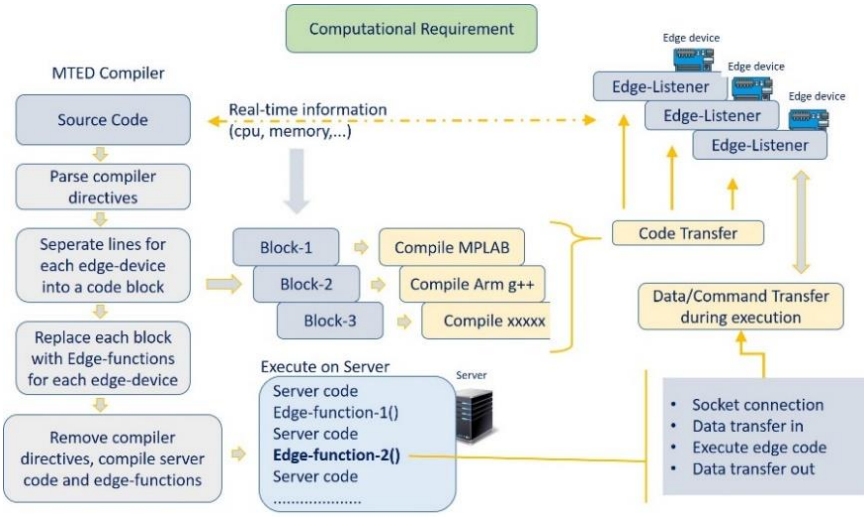

Figure 4. *MTED Compiler architecture*

There are two main components in the architecture. One of them is an edge-listener, which provides all the communication between the edge device and the *MTED* compiler. Also, during the execution, it connects the edge-device and the executable on the server. The second part of the architecture is the *MTED* compiler. The compiler's job is to parse the source code and separate the edge device code from the rest. It also needs to find out the required compilers to compile edge-specific code on top of the server code. In detail, *MTED* is a pre-compiler and management program. It is using the available compilers already installed.

4.1. Edge-Listener

The edge-device needs to run a simple edge-listener to communicate with the compiler and executable during compilation and execution. The purpose of the edge-listener is to communicate with the server where the compilation takes place.

Before edge-device specific compilation, there are several parameters to be determined. For example, the available memory, the target architecture (CPU type, clock speed, register sets, etc.), transfer speed to/from the device, and sensor types connected to edge-device, can be given. An edge-code working on the edge-device will provide these parameters to the compiler in real-time as to upload the code; the device needs to be alive.

Another functionality of the edge-code is installing the compiled code to the device, transfer data between edge-device and server, and execute the installed code when a request arrives from the server. The required functions are listed in Table 1. There is a specific protocol between the compiler and edge-listener to send/retrieve data, start execution, and connect/disconnect.

Table 1. *Edge-Listener functionality*

Function	Purpose
Parameters	The pre-compiler will communicate with the edge-listener to learn edge-specific parameters. This information can be provided to the compiler in real-time to display live-help during coding using a plugin.
Data Transfer	During execution, the edge-function at the server will transfer the data to the edge-device. The data will be considered as a byte stream.
Execution	During execution, the edge-function at the server calls the installed code at the edge-device with the listener's help. The listener will pass the request to the installed code and execute it.

4.2. Edge-Listener Commands

The communication with the edge-listener is accomplished using a predefined set of commands. A sample command list is given in Table 2. These commands will be used after initiating the connection to the device. The command set can be extended as necessary.

Table 2. *Edge-listener commands*

Parameter	Purpose
Transfer funcname binfile	The executable code at the server will transfer the binary file to be executed at the edge-device.
Send data len	The data will be considered as a byte stream. "len" parameter indicates # of bytes to send. For example, when we need to send a floating-point number, "len" becomes 4.
Execute funcname	This command will execute the previously sent "funcname".
Stop funcname	Stop the execution
Pause funcname	Pause the execution
Resume funcname	Restart the previously paused execution
Receive data len	The command will be used to receive the data generated at the edge-device as a byte stream.
Get DeviceParam	The command will be used to receive device-specific parameters like CPU, memory, etc.

4.3. Edge-Listener Parameters

The edge-listener is installed on the edge-device once. After the installation, it provides a communication bridge between the server and the edge-device. The *MTED* compiler needs several parameters. These are listed in Table 3.

Table 3. *Edge-device parameters*

Parameter	Purpose
Available memory	The compiler will use the available memory to decide if the edge-specific code can fit edge-device or not. Depending on the compiler, this parameter can be used during compilation.
CPU info (# of cores, speed, clock, availability)	The compiler needs this information to determine the computation performance of the edge-device.
Transfer speed	This information may not be available directly by the edge-listener, but it can be calculated using a test transfer between server and edge-device.
Sensor information (connections, types, data types)	The available sensor types and connections on the edge-device will help the compiler to minimize conflicts on the edge-device during execution.

The parameters are transferred from the edge device using JSON format to save space compared to XML. A sample output from an Arduino device is given below in Table 4.

Table 4. *Edge-device parameters*

```
{
   "edge-device-1": [
      {
         "Model": "Arduino UNO R3 - USB ChipCH340",
         "Microcontroller": "ATmega328",
```

```
            "Flash Memory": "32 KB",
            "SRAM": "2 KB"
            "EEPROM": "1 KB"
        }
    ]
}
```

4.4. Edge-Listener Implementation

The implementation of the Edge-Listener depends on the model and OS of the edge device. If the edge device runs an OS that is multitasking, like Raspberry Pi, a background process can be written easily. On the other hand, if the device is not running a multitasking OS, i.e., a single-threaded device like Arduino, the implementation is more difficult. In that case, there are a few possibilities.

Case A: The device communication port generates an interrupt upon receiving data: In this case, the edge-listener routine can be installed as a communication interrupt vector, and it will wake up whenever there is a request that arrives at the device. Since the listener is an interrupt routine, the main thread can be reserved for the program transferred to the device. Each time the listener wakes up because of a communication request, it will process the request, and the main thread will resume.

Case B: When the device cannot generate an interrupt because of the communication port, the edge-listener can be installed as a timer interrupt routine, which wakes up periodically and processes incoming requests, if any. This solution may be more time-consuming compared to A) because of the difficulty of finding an optimal timer expiration value. A timer expiration value, which is too long, will slow down the response of the device. On the other hand, a value that is too short will slow down the main thread. But this solution does not depend on a communication port interrupt, which may be available or not. Almost all microcontrollers have a timer interrupt. Therefore, solution B) is more generic and easily implementable compared to A). In the initial test implementation, solution B) is used.

4.5. Compiler

The *MTED* compiler is required to separate the edge-device code from the rest. Except for redesigning the compiler from scratch, one solution is to introduce a pre-compiler phase where the code separation is accomplished a priori to the actual compilation.

The code that needs to be separated and sent to an edge-device is replaced with an edge-function during pre-compilation. The function assumes that the edge-device code is compiled and installed on the edge-device by a suitable compiler. The purpose of the edge-function is to create a connection with the edge-device, transfer the data, and send the required commands to the edge-device listener to execute the installed code and collect/transfer any data generated.

The pre-compiler phase is controlled using compiler directives specific to the operation. These specific compiler directives are removed from source code after processing so that the source code can be compiled as usual.

4.6. Directives

In order to pre-process the code, several compiler directives are defined. These directives are listed in Table 5.

Table 5. *Pre-compiler directives*

Directive	Purpose
#device begin	It indicates the beginning of the description phase for device-specific compilation.

#device end	It indicates the end of a device-specific compilation.
#device label	It assigns a label or identification number to the device.
#device offline	It means that the edge device is offline. The compiler should continue with the available information about the device.
#device type xxxxx	It defines the type like PIC18, ARM, etc.
#device ip xxx.xxx.xxx.xxx	It sets the IP number of the device.
#device dns abc.def	It sets the DNS name of the device.
#device complexity	The user may prefer to indicate the complexity of the calculation. Otherwise, the complexity will be derived from the code if possible.
#device input inputlist	This directive indicates the input variables to the device-specific code.
#device output outputlist	This directive indicates the output variables to the device-specific code
#device codetype {C,C++,ASM}	The code language for the edge-specific target. With this directive, an assembler code inside the source code can be inserted as well.
#device constraintT time	This directive is used to instruct the *MTED* compiler to use edge-devices if the timing constraint can be satisfied by the device. For example, a long computation should not be sent to a prolonged device.
#device constraintM size	This directive is used to instruct the *MTED* compiler to use edge-devices if the data transfer constraint can be met by the device. For example, a long array should be processed in place, and only the result should be transferred. This can be accomplished by limiting the data transfer size.

4.7. Restrictions

The pre-compiler needs to compile the device-specific code and install it at the edge-device. Therefore, the limitations of the edge-device become crucial. To use these limitations, there are some restrictions in the device-specific code section. The first restriction is that all array allocations need to be static. No dynamic memory allocation is allowed here. This requirement is needed to compute the memory requirements and compare them with the memory of the edge-device.

The second restriction is that all loop variables need to be static or finally-static (a value is assigned only once). With this information, the pre-compiler can calculate the computational requirements and compare them with the edge-device speed.

5. EDGE –DEVICE COMPILATION AND EXECUTION STAGES

The *MTED* compiler separates edge-specific code from the source code, and it replaces the code with a function call. The separated code needs to be compiled with respect to the edge-specific target. The system will use an existing compiler executable for this purpose. During the setup of the *MTED* compiler, available compilers will be detected. If an existing compiler is not detected, the user can change the setup to register the compiler executable manually. For example, Microchip has a compiler for PIC18 microcontrollers called MPLAB® C Compiler for PIC18 MCUs (C18). The compiler can be added to the system manually if not detected automatically.

Rewriting a new compiler for a specific target is not considered at this stage since several compilers exist for the most popular target types.

5.1. Stage-1: Source Code

The first stage is to write the source code using pre-compiler directives. An example source code is given in Figure 5. The single source code includes all the functionality, including all edge-devices. The compiler directives isolate the code that needs to be sent to a specific edge-device. The IP-number/DNS-name of the

edge-device and a label need to be set as well. The IP number setting is optional as by setting the DNS name, the mapping can be implemented outside the source code. The *MTED* compiler will communicate with the device to learn operational parameters and transfer data/code. When the device is not available, the *MTED* compiler will give an error to indicate that this device cannot be included during the compilation. The user is responsible for finding a solution or for replacing the device with another one.

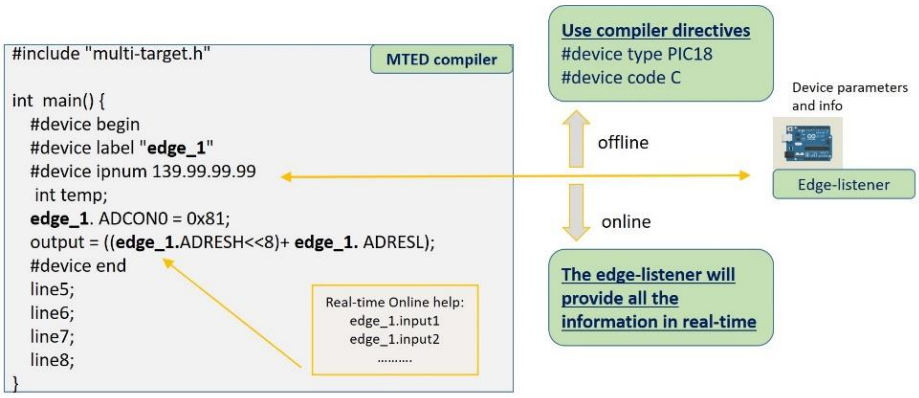

Figure 5. *Sample source code*

When the edge device is offline, it is still possible to compile the code using pre-existing information. Still, this option may introduce incorrect or missing information about the edge device. In order to compile with an offline device, the status should be indicated with another compiler directive.

5.2. Stage-2: Edge-Code

The edge code (the code that needs to be executed at the edge-device) is separated from the main source code using compiler directives as given in Figure 6. The edge code can be coded in the original language as the server, but the edge code can be in assembly as well. The type of code should be given using another compiler directive. The *MTED* compiler will choose the correct compiler depending on the type of the source code. To compile a code that supports multiple targets and languages, the user needs to obtain all the required compilers.

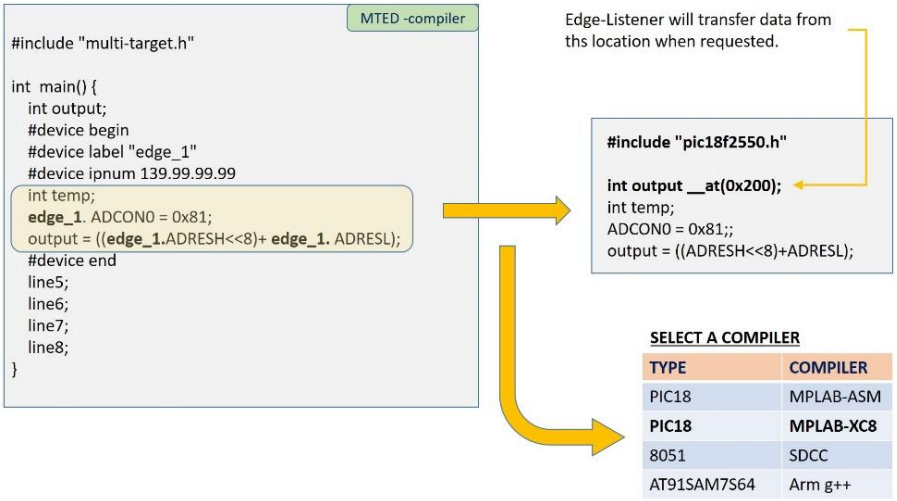

Figure 6. *Sample edge code*

The architecture does not define a new compiler but a new approach to combine several different targets and languages in a single framework.

There are several additions to the edge code. All variables inside the edge code need to be redefined if necessary. Since these variables' definition is left in the server code, the *MTED* compiler will add the

definitions to the separated code piece. Another addition is the code that is needed to transfer the data in/out of the device. The edge device code is compiled after these additions. The build binary file is transferred by the edge-compiler to the edge-device using the edge-listener. The edge-listener will allocate memory and install the binary to the memory.

5.3. Stage-3: Input/Output Parameters

The input/output variables and parameters to the edge-code should be determined. The user can define these parameters using compile directives. If not defined, the *MTED* compiler will scan the code to find out the parameters. Therefore, the definition of the user will speed-up the process. All arrays used in the edge code should be statically defined to evaluate memory restrictions and complex calculations. Dynamic variable allocations are not allowed here. All external input variables should also be defined as final (static) or effectively final to make complex calculations. The limitation will help the pre-compiler to determine loop boundaries and calculate the complexity of the calculation.

Case A: The input/output variables in the edge code are known. Depending on the compiler, it is possible to give absolute memory locations to these variables at the edge-device. Since the location is known, the Edge-listener can transfer any data in/out from/to these locations. For example, using MPLAB XC8 compiler, the *__at* specifier can be used to fix the location: *int myVariable __at(0x400)*; The data transfer is handled by edge-listener, and afterward, the edge code is started.

Case B: Another option is to add the extra code to transfer the data in front of the edge-device code. The extra code will initialize the network card, open the connection, and transfer data before proceeding with the original computation. Code portions are given in Table 6. The extra code will copy the data from the network directly into the input variable and continue. The output variable is also initialized similarly. The data transfer is handled by the edge code directly. The edge listener is not involved during the transfer in this case.

Table 6. *Sample communication*

```
Ethernet.begin(macAddr,ipAddr);
Udp.begin(localPortNumber);
Serial.begin(9600);
IPAddress remoteIP = Udp.remoteIP();
Udp.read(buffer,MAX_SIZE);
```

5.4. Stage-4: Edge-Function Generation

After processing the directives and locating the devices, the *MTED* compiler will replace the edge-code with a function call generated by the compiler. An example is given in Figure 7. The generated function will make the necessary connection to the edge-device, transfer data, and send the necessary command to execute the function installed at the edge-device.

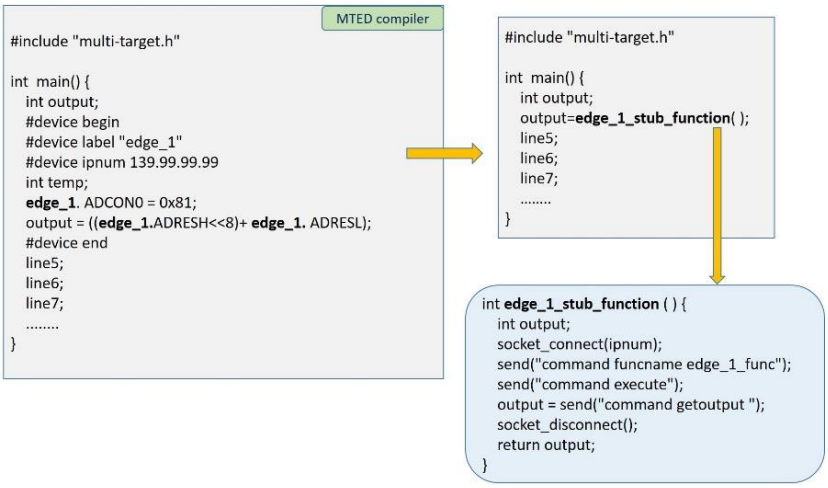

Figure 7. *Sample function*

5.5. Stage-5: Source-Code Compilation

The remaining source code is compiled at the server. All edge-code segments are replaced with edge-function calls. All pre-compiler directives are also removed. The compile errors are displayed separately for each device since each device has a different compiler. A plugin running in Eclipse or NetBeans IDE can display all different messages using the same error output window.

5.6. Stage-6: Execution

The binary file created at the server can now be executed. The edge-functions created during compilation will create all connections to edge-devices and execute the functions installed at each edge-device. During the execution, all communication code is embedded in the binary file. *MTED* compiler is not involved in the execution.

6. THE COMPLETE PROCESS

6.1. State Diagrams

The state transitions are given below as a summary. The diagrams are summarizing the essential primary states where there are other side-states as well. The state diagram of the edge-listener is given in Figure 8. The edge-listener needs to be active all the time. Therefore after the initialization, there is no ending state in the flow.

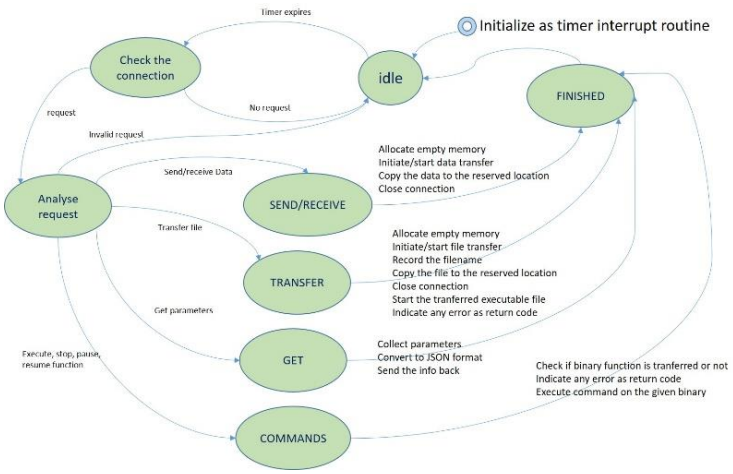

Figure 8. *State diagram of the edge-listener*

The state diagram of the *MTED* compiler is given in Figure 9. The *MTED* compiler is active during code development and code generation.

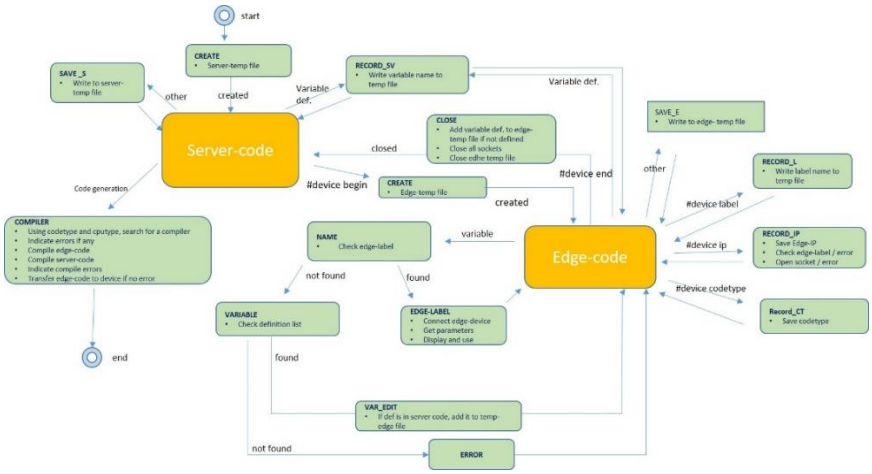

Figure 9. *State diagram of the compiler*

6.2. Process Flow

The flow of the compilation and execution is given in Figure 10. Until the execution stage, the process is controlled by the *MTED* compiler, and after the compilation, the compiled program starts talking to edge-devices and execute.

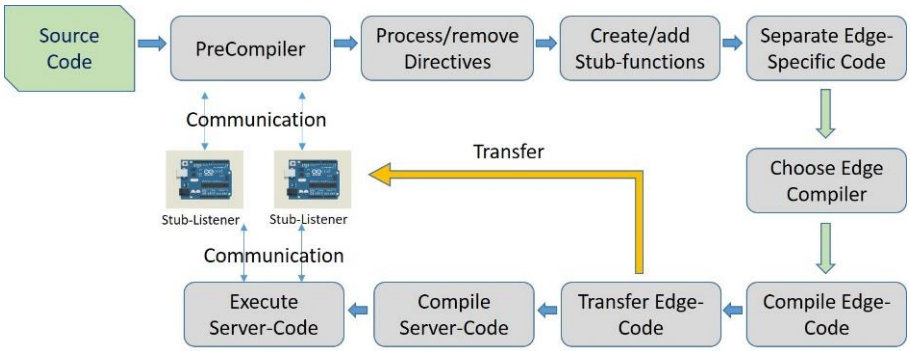

Figure 10. *Process flow*

7. IMPLEMENTATION HIGHLIGHTS

The *MTED* compiler is implemented as a prototype using Java language, and it is tested in a small network that includes Arduino and Raspberry-Pi boards as edge devices. The language used to program the edge-devices and the server is C/C++. All devices (PC and edge devices) are connected to the internet. The listener running on edge device is written using C/C++ only once for every device type.

7.1. MTED Compiler

The *MTED* compiler is written in Java to process the source code that is written in *C/C++*. The source code includes server and device-specific code.

7.1.1. Parser

The *MTED* compiler reads the source code written in *C/C++* to program the server and edge devices, and it parses the input accordingly depending on compiler directives. A simple parser is implemented using Java libraries to process tokens and patterns. The *StringTokenizer* class is used to extract individual tokens and, *Matcher* and *Pattern* objects are using regular expressions to differentiate different tokens. The parser

reads the source code line by line, processes compiler directives, and separates input and output variables from device-specific code. For example, the regular expression to recognize a variable is given as "*([a-zA-Z_][a-zA-Z0-9_]*)".* All edge device information and code are given between *#device begin* and *#device end* directives. Once a *#device begin* compiler directive is processed, the remaining information obtained from other compiler directives is stored in an object. The object is stored in a *HashMap* using the device IP number as an index.

The variables and their types need to be parsed because the *MTED* compiler needs to make the following modifications to the device-specific source code. Simple variables are considered in the prototype, and the type of the variables is limited to byte.

A) All byte variables are parsed and stored in another *HashMap* indexed by the variable name. This list is needed to insert extra variable definitions to device-specific code if the variables in the device-specific code are not defined between compiler directives as described in Figure 7. This process is required since the device-specific code will be separated from the original source code and compiled separately. We need to keep two different *HashMap's* to keep the variables outside and inside the device-specific code differently. When a variable is parsed inside the device-specific code, first, the *HashMap* of the inner variables is searched. If the variable is found here, the definition is added to the original source code. If not found, in that case, the variable definition is obtained from the outer *HashMap* table and inserted to the top of the device-specific code.

B) The device-specific code is parsed to find out input and output variables to the device-specific code section. An input variable to the code section is the variable that is used only on the right side of an assignment. Similarly, an output variable is used only on the left side of an assignment. The information to an input variable is coming from the remaining part of the code. Therefore, the value of an input variable needs to be transferred to the edge device, and the necessary code to transfer the data needs to be added to the device-specific code. The transfer will take place during the execution of the device-specific code. Similarly, an output variable needs to be copied to the original source code after executing the device-specific code. The code to transfer the value of an output variable is also added to the device-specific source code by the *MTED* compiler.

   As an option, the input and output variables can also be given using compiler directives. In that case, the code is not scanned for input and output variables.

   The byte restriction simplifies the data transfer since byte transfers are handled by a socket connection directly. Longer size variables can be transferred one byte at a time or as a stream as well if needed.

   The input and output variables are also used to define the device-specific stub function since the *MTED* compiler needs to pass all input variables to the function and return all output variables.

All code sections (all lines without a compiler directive) between *#device begin*, and *#device end* directives are dumped to a text file for further processing. This file will be compiled using a cross-compiler depending on the edge device. For example, the command-line version of the Arduino IDE compiler is used to compile programs for Arduino.

7.1.2. Network access

The *MTED* compiler will use *java.net.InetAddress* object to implement DNS lookups (to learn the IP number of edge devices if not given) and *java.net.** library to implement all socket connections (to connect to edge devices to send commands and transfer code).

7.2. Edge Device Listener

The listener program needs to be developed only once for each device type. Once the listener is running, the programming of the edge device can be controlled by the *MTED* compiler for different computational purposes.

The listener has 3 basic functionalities. The first one is to provide the edge device type, and other edge device features to the *MTED* compiler when requested. This information is used by the *MTED* compiler to decide the cross-compiler and to get online help for device port names and other hardware features. The second job is to transfer the binary code compiled at the server using a cross-compiler to the edge device. This transfer requires binary data communication. Once the binary executable file is transferred to the local file system, the edge device program is ready to be executed. As a third functionality, the listener will create a background job from the transferred program when requested by the server program.

7.2.1. Raspberry Pi

The device uses an operating system based on Linux (Debian), which is modified for Raspberry hardware. Therefore it is relatively straightforward to create a listener program and to execute background jobs. Although it is possible to override the OS completely by burning your own bootloader, this option is not exercised in the prototype.

The listener program is implemented using C/C++, and it is compiled using g++ running on Raspberry Pi OS. The socket communication is using <*sys/socket.h*> and <*netinet/in.h*> libraries with standard methods like *socket(…), gethostbyname(…), connect(…), accept(…), read(…) and write(…).* The listener will read the commands by the *MTED* compiler and respond accordingly, as given in Table 2.

7.2.2. Arduino

The Arduino device is using an Ethernet Shield board to implement network connections. The Arduino device does not contain an operating system but a bootloader that transfers the desired program to the device and executes it. The original bootloader is listening to USB connections, and it is used to upload a modified bootloader program to the device. On the other hand, the modified bootloader is listening to socket connections to communicate with the *MTED* compiler. On the Arduino side, the <*Ethernet.h*> library is used with C/C++ to create socket connections.

A background job is created by installing the modified bootloader (edge-listener) as an ISR (Interrupt Service Routine) triggered by the Timer. This way, the edge-listener will wake up periodically and execute commands coming from the *MTED* compiler.

The edge-listener is writing the device-code transferred by the *MTED* compiler to Arduino's flash memory using spm instruction.

7.3. Server Code

The server code is written in C/C++. The reason for the choice is that the C/C++ language is ported to many platforms and devices and widely available. The server code also contains device-specific code, which is separated by using compiler directives. There are no limitations in the server code except that all variable types used inside the device code are chosen to be bytes for simplicity.

The device code will be converted to a function call, and for the server code, the execution of the device code is transparent. The function call is created using the input and output variables extracted by the parser, as explained above.

The function call has the following features. 1) Start the device code, which is copied to the device by the *MTED* compiler. This is accomplished by sending a start command to the edge listener. 2) Create a socket

connection to transfer the input and output variables. This socket connection is between the server code and device code, and it is initiated by the device code. After obtaining the output variable, the function call returns to the calling server code.

7.4. Device Code

The device code embedded in the server code is extracted from the source code using compiler directives. The language is chosen to be C/C++ for the device code for the unified flow of the computation. Since the device code will be extracted and compiled separately, a different language like assembler language can be chosen as well.

The device code will be dumped to a file to be compiled by a cross-compiler for the target device. Several modifications are added to the device code. The device code needs to make a socket connection to the server code to get its input variables and another socket connection to the server to transfer the output variables after the execution. The socket connection code is added to the start and the end of the device code. During this addition, the device-specific libraries are considered, which are different for the Arduino and for the Raspberry Pi.

7.5. Sample Codes Generated for Arduino Edge-Device

A server code transformation generated by the *MTED* compiler is given in Tables 7, 8 and 9. The sample code is working on an Arduino device. The Arduino edge-code is compiled using the command-line version of the Arduino IDE compiler, and the binary file is transferred to the edge device by the *MTED* compiler. Several error checks, structures, and other extra information in the code are not given for simplification purposes. The stub function uses the WinSock library to make socket connections because the server code is running on a Windows machine. If the server code needs to run on a Linux machine, the socket code should be modified accordingly.

Table 7. *Sample server code*

Server-Code

```
int main () {
   char input;
   char output;

   scanf("%c",&input);
#device begin
#device label edge_1
#device ip 192.168.2.17
    char temp;
    temp = input * 2;
    PORTD = temp;
    output = PORTC;
#device end
    printf("%c",output);
}
```

Table 8. *Modified server code and stub function*

Modified Server Code

```
int main () {
   char input;
   char output;

   scanf("%c",&input);
   output = edge_function_1(input);
   printf("%c",output);
}
```

Stub-function (WinSock socket connections)

```
char edge_function_1(char input){
......
edgeSocket=socket(AF_INET , SOCK_STREAM , 0 )) ;
edge_device.sin_addr.s_addr = inet_addr("192.168.2.17");
// Edge-Device listener
edge_device.sin_port = htons( 8880 );
.......
connect(edgeSocket,(struct sockaddr *)&edge_device,
sizeof(edge_device));
// The edge function is transferred to edge device after
compilation.
message = "start edge_function_1";
send(edgeSocket , message , strlen(message);
......
// Start listening to the edge-device code
```

```
edge_device.sin_addr.s_addr = INADDR_ANY;
// Server-code listener
edge_device.sin_port = htons( 8885 );
bind(s ,(struct sockaddr *)&edge_device ,
sizeof(edge_device));
listen(edgeSocket , 3);
edge_socket = accept(edgeSocket , (struct sockaddr
*)&edge_code, &c));
// Send the input value of the function to the edge code
send(edge_socket , &input , 1 , 0);
......
// Get the output value (1 byte) from the edge code and
return it
recv(edgeSocket , outputBuffer , 1 , 0));
......
return outputBuffer[0];}
```

Table 9. *Device code*

Device-Code extracted/modified by *MTED* compiler and compiled for Arduino

```
// -------- Added by MTED compiler ---------
#include <Ethernet.h>

byte mac[] = { 0xDD, 0xAA, 0xEE, 0xFF, 0xAB, 0xCD };
byte ip[] = { 192, 168, 2, 17 };     // Edge-Device IP Number
byte server[] = { 192, 168, 2, 1 }; // Server-Code IP Number

EthernetClient client;
void setup() {
  char out;
  char inp1;
  // Connect to the server-code
  Ethernet.begin(mac, ip);
  client.connect(server, 8885);
  // Transfer the input variable from the stub-function
  if (client.available()) {
    inp1 = client.read();
  }
  // -------- END -------
  // --- EDGE-CODE: Code extracted from server-code ----------
  char temp;
  temp = inp1 * 2;
  PORTD = temp;
  out = PORTC;
  // --------- END ----------
  // -------- Added by MTED compiler ---------
  // Transfer the output variable to the stub-function
  if (client.available()) {
    client.write(out);
  }
  // --------- END ----------
}
```

8. DISCUSSION and COMPARISON

Having the fact that the first version is not a commercial product, the system may not be compared one-to-one with commercial tools. The system is developed to answer the research paradigm, and the tool can be commercialized in the future.

It could be argued that there are cross-platform systems like Java where the Java run-time will take care of running the programs on different systems and devices. Two cases need to be considered.

The first concern is that using Java will not provide a unified approach to the distributed computation. The Java code running on the server and the edge devices needs to be designed individually, and the communication needs to be taken care of by the user. There is no code separation in Java, as provided by the *MTED* compiler. Technically the *MTED* compiler can be modified to use Java as a destination language instead of C/C++. The *MTED* compiler is a framework independent of the compiler language.

The second problem of Java is that it needs to run (even if possible) on small device architectures with extreme memory limitations. Accessing the device hardware is more difficult by using a simulated language. The *MTED* compiler will generate native code that will run directly on the device hardware, unlike Java byte code. The performance difference could be ignored on a computer system, but the resources are minimal on an edge device.

In [22], the tools are compared in terms of management domains. The proposed *MTED* compiler is evaluated and compared in terms of the given domains. A comparison in Table 10 is provided using the information in [22] to evaluate the proposed system.

Table 10. *Comparison chart*

	Application development	**Device management**	**System management**	**Heterogeneity management**	**Data management**	**Analytics**	**Deployment Management**	**Monitoring management**	**Visualization**	**Research**
Aer cloud			*			*		+		
Arkessa		*			+					
Arrayant connect	*	*		+	*					
Axeda		*			+			*		
Ayla's cloud fabric		*	+						*	
Carriots	+	*						*		
Echelon	*	*					+			
Etherios		+						*		
Exosite		*	+					*		
GroveStreams	*							*	+	
IBM IoT		*								+
InfoBright					*	+				
Jasper Control Centre	*					+		*		
KAA	+				*					
Microsoft research lab of things	*									+
Nimbits					+	*				
Oracle IoT cloud			*	*	+		*		*	
OpenRemote	*			+						
Plotly						*		*	+	
SeeControl IoT		+				*			*	
SensorCloud		+						*	*	
Temboo	+									
Thethings.io	*		+					*		
ThingSpeak	*							+	*	
ThingWorx	*				+			*		
Xively	*	+						*		
MTED COMPILER	Development is fully supported using available compilers	The system is not designed to setup the system. Therefore this feature is partially supported	System management is not a feature of the proposed system	Different architectures are supported as long as the platform has a corresponding compiler	The data transfers related to execution are handled transparently, but the system is not designed to handle unrelated transfers	There is no extra system developed to perform analytics in the system but any available library can be included in the program. Therefore, any external analytics is supported by the system	Deployment management is not a feature of the proposed system as this can be added externally by other tools	Monitoring is partially supported as the system is communicating with each edge-device during compilation and execution. This is partially supported as full monitoring is not a requested feature.	This feature is not implemented.	This domain is implicitly supported as the compiler can connect all edge-devices and execute programs.

*: full, +: partial

A significant contribution can be stated as follows. Many systems are trying to isolate the edge-device using standard interfaces, objects, modules, and these abstractions help the programmer create a global model. Nevertheless, at the same time, the user is losing precise control of the edge-device. The proposed model is using edge-device listeners to help the programmer in terms of creating abstract models. Still, at the same time, it also helps the programmer to program each edge-device precisely. The proposed architecture does not override previous models, and these models can be integrated into the *MTED* compiler.

There are several places where the system can be improved. In the Arduino prototype, the edge-listener is installed as a timer interrupt routine to generalize the solution. On the other hand, the timer value seems to be difficult to set correctly for different programs. There is a need to find a relationship between the timer value and the edge-function duration (complexity).

Socket connections can be maintained by the *MTED* compiler globally and kept open to speed up the process during compilation and editing. During the execution, a connection manager can open and maintain all socket connections so that the execution of the compiled binary will not slow down because of socket operations.

The prototype can be improved, and the edge-device types should be increased by including different devices like PIC series devices and others. During the development, the required pre-compiler directives will also be improved as necessary. Another improvement is to merge the *MTED* compiler with other development methodologies. There are several approaches that are using objects, interfaces, and other abstract access methods. *MTED* compiler can also use these approaches in an integrated manner. In the future, a plugin version to NetBeans or Eclipse development environment will also improve the usage with an integrated GUI.

8. CONCLUSIONS

The proposed *MTED* compiler will streamline the whole edge-device executions and compilations using a single source code. The term “single source code” is used to represent a single project development phase. There is only one source code that needs to be modified. The edge-devices can be defined easily using compiler directives. The source code can be fine-tuned and moved between edge-devices easily. IoT management utilities concentrate on device control. For example, the Nokia IoT platform provides the security of the edge devices. It also has an authorization feature for edge device management. The Authentication is also provided by the platform. The analysis of the device is another feature listed. The platform simplifies deployment and management functions, but it fails to provide a unified code development. On the other hand, the *MTED* compiler is providing a transparent distribution of the code between the edge devices. Another solution for IoT management is the Microsoft Azure system. The platform can define user roles to control the devices. It can also reconfigure the devices remotely. Security is one of the main features of the Azure platform. Monitoring the devices and managing them remotely is also given as a basic feature. The problems can also be detected. The platform still fails to provide a single source code where the computation is divided automatically between devices.

In terms of management, the *MTED* compiler is not competing with management tools in general. Still, it provides a unified view of the project in addition to the management of IoT devices. Using the proposed approach, the user will start thinking about the programming project as a single process. The data transfers and program logic are generated automatically in the background, and there is no need for these operations to be designed individually for each edge-device.

CONFLICTS OF INTEREST

No conflict of interest was declared by the author.

REFERENCES

[1] Bhardwaj, K., Sreepathy, S., Gavrilovska, A. A., Schwan, K., "ECC: Edge Cloud Composites", in 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, 38-47, (2014).

[2] Chen, X., "Decentralized computation offloading game for mobile cloud", IEEE Transactions on Parallel and Distributed Systems, 26(4): 974-983, (2015).

[3] Chen, X., Jiao, L., Li, W., Fu, X., "Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing", IEEE/ACM Transactions on Networking, 24(5): 2795-2808, (2016).

[4] Loomba, R., Frein, R. D., Jennings, B., "Selecting Energy Efficient Cluster-Head Trajectories for Collaborative Mobile Sensing", in 2015 IEEE Global Communications Conference, San Diego, 1-7, (2015).

[5] Sani, A. A., Boos, K., Hong, M. Y., Zhong, L., "Rio: a system solution for sharing i/o between mobile systems", in Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, 343, (2014).

[6] Varghese, B., Buyya, R., "Next generation cloud computing: New trends and research directions", Future Generation Computer Systems, 79(3): 849-861, (2018).

[7] Yi, S., Li, C., Li, Q., "A Survey of Fog Computing: Concepts, Applications and Issues", in Proceedings of the 2015 Workshop on Mobile Big Data, 37-42, (2015).

[8] Varghese, B., Wang, N., Barbhuiya, S., Kilpatrick, P. Nikolopoulos, D. S., "Challenges and opportunities in edge computing", in 2016 IEEE International Conference on Smart Cloud, New York, 20-26, (2016).

[9] Hong, K., Lillethun, D., Ramachandran, U., Ottenwälder, B., Koldehofe, B., "Mobile Fog: A Programming Model for Large–Scale Applications on the Internet of Things", in Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing, 15-20, (2013).

[10] http://microservices.io., Available: 2020.

[11] Arumugam, S. S., Badrinath, R., Herranz, A. H., Holler, J., Azevedo, C. R. B., Xiao, B., Tudor, V., "Accelerating Industrial IoT Application Deployment through Reusable AI Components", in Global IoT Summit, 1-4, (2019).

[12] Bracciale, L., Loreti, P., Detti, A., Paolillo, R., Melazzi, N. B., "Lightweight Named Object: An ICN-Based Abstraction for IoT Device Programming and Management", IEEE Internet of Things Journal, 6(3): 5029-5039, (2019).

[13] Chen, H., Xie, K., Cui, L., Pescapè, A., "A Formal Methodology for Easing Development and Maintenance of Entity Services in Service Oriented Software-Defined Internet of Things", IEEE Internet of Things Journal, 6(6): 9516-9530, (2019).

[14] Fersi, G., "A distributed and flexible architecture for Internet of Things", in The International Conference on Advanced Wireless, Information, and Communication Technologies, 130-137, (2015).

[15] Nastic, S., Sehic, S., V¨ogler, M., Truong, H.-L., Dustdar, S., "A Novel Programming Model for IoT Applications on Cloud Platforms", in IEEE 6th International Conference on Service-Oriented Computing and Applications, 53-60, (2013).

[16] Zambonelli, F., "Key Abstractions for IoT-Oriented Software Engineering", IEEE Software, 34(1): 38-45, (2017).

[17] Vogler, M., Schleicher, J. M., Inzinger, C., Dustdar, S., "DIANE – Dynamic IoT Application Deployment", in IEEE International Conference on Mobile Services, 298-305, (2015).

[18] Peña, M. A. L., Fernández, I. M., "SAT-IoT: An Architectural Model for a High- Performance Fog/Edge/Cloud IoT Platform", in IEEE 5th World Forum on Internet of Things, 633-638, (2019).

[19] Pramudianto, F., Kamienski, C. A., Souto, E., Borelli, F., Gomes, L. L., Sadok, D., Jarke, M., "IoTLink: An Internet of Things Prototyping Toolkit", in IEEE 11th Intl Conf on Autonomic and Trusted Computing and 2014 IEEE 14th Intl Conf on Scalable Computing and Communications and Its Associated Workshops, 1-9, (2014).

[20] Vlasov, Y., Illiashenko, O., Uzun, D., Haimanov, O., "Prototyping tools for IoT systems based on virtualization techniques", in IEEE International Conference on Dependable Systems, Services and Technologies, 87-92, (2018).

[21] Liu, F., Tang, G., Li, Y., Cai, Z., Zhang, X, Zhou, T., "A Survey on Edge Computing Systems and Tools", in Proceedings of the IEEE, 107(8): 1537-1562, (2019).

[22] Ray, P. P., "A survey of IoT cloud platforms", Future Computing and Informatics Journal, 181: 35-46, (2016).

[23] Siow, E., Tiropanis, T., Hall, W., "Analytics for the Internet of Things: A Survey", ACM Computing Surveys, 51(4): 74, (2018).

[24] Sobin, C. C., "A Survey on Architecture, Protocols and Challenges in IoT", Wireless Personal Communications, 112: 1383–1429, (2020).

[25] Hernández-Ramos, J. L., Baldini, G., Matheu, S. N. N., Skarmeta, A., "Updating IoT devices: challenges and potential approaches", in Global Internet of Things Summit, Dublin, 1-5, (2020).

[26] Zikria, Y. B., Kim, S. W., Hahm, O., Afzal, M. K., "Internet of Things (IoT) Operating Systems Management: Opportunities, Challenges, and Solution", Sensors, 19(8): 1793, (2019).

[27] Song, Z. J., Tilevich, E., "A Programming Model for Reliable and Efficient Edge-Based Execution Under Resource Variability", in IEEE International Conference on Edge Computing, 64-71, (2019).

[28] Rafique, W., Yaqoob, I., Qi, L., Imran, M., Rasool, R. U., Dou, W., "Complementing IoT Services Through Software Defined Networking and Edge Computing: A Comprehensive Survey", IEEE Communications Surveys & Tutorials, 22(3): 1761-1804, (2020).

[29] Ray, P., "A survey on Internet of Things architectures", Journal of King Saud University – Computer and Information Sciences, 30(3): 291–319, (2016).

[30] Kumar, N. M., Mallick, P. K., "The Internet of Things: Insights into the building blocks, component interactions, and architecture layers", in International Conference on Computational Intelligence and Data Science, 132: 109-117, (2018).

[31] Kumar, S. Y. R. and Champa, H. N., "An Extensive Review on Sensing as a Service Paradigm in IoT: Architecture, Research Challenges, Lessons Learned and Future Directions", International Journal of Applied Engineering Research, 14(6): 1220-1243, (2019).

[32] Rafique, W., Zhao X., Yu, S., Yaqoob, I., Imra, M., Dou, W., "An Application Development Framework for Internet-of-Things Service Orchestration", IEEE Internet of Things Journal, 7(5): 4543-4556, (2020).

[33] Gökçay, E., "An on Demand Virtual CPU Arhitecture based on Cloud Infrastructure", in Proceedings of the 7th International Conference on Cloud Computing and Services Science, Setubal, 323-329, (2017).