TinyLink 2.0: Integrating Device, Cloud, and Client Development for IoT Applications

Gaoyang Guan, Borui Li, Yi Gao, Yuxuan Zhang, Jiajun Bu, and Wei Dong*
College of Computer Science, Zhejiang University, and
Alibaba-Zhejiang University Joint Institute of Frontier Technologies, China
{guangy,libr,gaoy,zhangyx}@emnets.org,{bjj,dongw}@zju.edu.cn

ABSTRACT

The recent years have witnessed the rapid growth of IoT (Internet of Things) applications. A typical IoT application usually consists of three essential parts: the device side, the cloud side, and the client side. The development of a complete IoT application is very difficult for non-expert developers because it involves drastically different technologies and complex interactions between different sides. Unlike traditional IoT development platforms which use *separate* approaches for these three sides, we present TinyLink 2.0, an *integrated* IoT development approach with a single coherent language. It achieves high expressiveness for diverse IoT applications by an enhanced IFTTT rule design and a virtual sensor mechanism which helps developers express application logic with machine learning. Moreover, TinyLink 2.0 optimizes the IoT application performance by using both static and dynamic optimizers, especially for resource-constrained IoT devices. We implement TinyLink 2.0 and evaluate it with eight case studies, a user study, and a detailed evaluation of the proposed programming language as well as the performance optimizers. Results show that TinyLink 2.0 can speed up IoT development significantly compared with existing approaches from both industry and academia, while still achieving high expressiveness.

CCS CONCEPTS

• **Computer systems organization → Embedded and cyber-physical systems**; • **Networks → Cyber-physical networks**.

KEYWORDS

Internet of Things, Integrated development

ACM Reference Format:
Gaoyang Guan, Borui Li, Yi Gao, Yuxuan Zhang, Jiajun Bu, and Wei Dong*. 2020. TinyLink 2.0: Integrating Device, Cloud, and Client Development for IoT Applications. In *The 26th Annual International Conference on Mobile Computing and Networking (MobiCom '20), September 21–25, 2020, London, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3372224.3380890

* Corresponding author.

MobiCom '20, September 21–25, 2020, London, United Kingdom

ACM ISBN 978-1-4503-7085-1/20/09…$15.00
https://doi.org/10.1145/3372224.3380890

1 INTRODUCTION

The recent years have witnessed the rapid growth of IoT applications including environmental monitoring [1, 11, 16], shared bicycles [10, 35], and smart homes [8, 29]. A typical IoT application usually consists of three essential parts: **(1) The device side**: the IoT device senses the physical environment and transmits data to the gateway which can be a WiFi access point, cellular base stations, etc. **(2) The cloud side**: the cloud which stores the data, manages the IoT devices and provides other key functionalities like machine learning. **(3) The client side**: the end device like a mobile phone which interacts with end-users and provides functionalities like displaying data and controlling devices.

Today, developing such applications is still very difficult because developers need to deal with drastically different technologies (e.g., embedded systems technology, cloud technology, and mobile technology) and complex interactions between different sides. The above difficulties have attracted great attention from both academia and industry as it is widely accepted that "The IoT must be easy. What is needed is IoT solutions for everyone, not just experts" [17].

Traditional IoT development uses a *separate* approach for device, cloud, and client (e.g., mobile) development. For example, IoT studio [6] provides a one-site IoT development platform consisting of the device platform, service platform, and mobile platform. Although it brings us great convenience as there is a unified framework and many reusable components, the implementation details of different sides are left to different developers.

Separate development faces several issues. First, it requires the co-operation of different developers, e.g., embedded systems developers, cloud developers, and mobile developers. The interaction and dependency among different developers slow down the development speed which is critical to obtain first-mover advantages and reduce development cost. Second, it is often tedious and error-prone as extra models and interfaces must be explicitly and clearly specified.

To address the above issues, we propose TinyLink 2.0, which uses an *integrated* IoT development approach with a single coherent language. TinyLink 2.0 mainly targets non-experts in the consumer-oriented IoT domains such as education, maker and startup. Its integrated programming model and other easy-to-use features are user-friendly to students, makers, and small startups. Based on TinyLink 2.0, they can rapidly build the prototype, test the performance, validate their ideas, and get early feedbacks. Expert developers can also benefit from the system because it accelerates building the prototypes of IoT projects. Afterwards, they can continue their advanced development with the source code provided by TinyLink 2.0.

TinyLink 2.0 builds on top of many existing works. For example, it uses IFTTT-like rules [15, 33] to specify IoT application logic. It builds on top of TinyLink [12] to generate the hardware configuration as well the actual binary program for the corresponding hardware. The existing works shed light on many important aspects towards the goal of simplifying the IoT development. However, we are facing unique challenges. We highlight three challenges and our solutions in the following:

Challenge 1: How to design a single coherent language for developing a complete IoT application?

Solution 1: Based on our simple observation, IoT application developers usually reason about data collection, device control, rule configuration, and interaction (e.g., data flow and service call). Some existing declarative languages [5, 21] are proposed for their specific domains (e.g., data query) and can cover some aforementioned aspects. Until recently, however, there lacks an integrated approach for developing IoT applications on the three sides and it is extremely difficult to merge these languages into one. TinyLink 2.0 provides a domain-specific language (DSL) in a structured programming style which can cover all the above aspects. Its application usually consists of rule capsules called policies and code capsules called TinyApps. Policies consist of IFTTT (IF-This-Then-That) rules and interactions that are used to express the main IoT application logic across TinyApps, while one TinyApp describes the behaviors of an IoT device or a mobile client. TinyLink 2.0 compilers can automatically compile the application code to software programs at different sides, simplifying the process of separate programming and separate compilation.

Challenge 2: How to express diverse IoT application logic (e.g., data collection, device control, multiple device interaction) in an easy way?

Solution 2: There are mainly two drawbacks of using traditional IFTTT rules to express application logic. First, limited keywords (e.g., IF, THEN and ELSE) in traditional IFTTT rules make them only be able to express simple application logic. Second, it is difficult and time consuming to express application logic containing machine learning using traditional IFTTT rules. In TinyLink 2.0, we first use an enhanced IFTTT rule design to improve the expressiveness. More keywords are added into TinyLink 2.0. It further includes a virtual sensor mechanism to help developers rapidly express application logic with machine learning.

Challenge 3: How to optimize the performance (e.g., energy consumption) on resource-constrained IoT devices?

Solution 3: Traditional approaches usually focus on optimizing the performance on each side. For example, a recent work RT-IFTTT [15] uses a sensing data prediction technology to reduce the energy consumption at the device side. TinyLink 2.0 further uses a *cloud-centric* approach where the cloud effectively has overall control. It optimizes the performance by a static optimizer and a dynamic optimizer. At compile time, the former intelligently determines where to put the application logic, either the device side or the cloud side. At runtime, the latter analyzes the conditions in IFTTT rules and dynamically requests sensor data with variable intervals to reduce the energy consumption.

We implement TinyLink 2.0 and evaluate its performance extensively. Results show that: (1) Its programming language can express diverse IoT application logic. Concretely, within a set of about 100 real-world IoT projects, it can implement about 85% of them; (2) It can reduce more than 85.28% of the lines of code compared with the best existing approaches; (3) It can use virtual sensors to automatically draw inferences from sensing data on various deployment circumstances. (4) Its dynamic optimizer searches a larger optimization space and can generate more energy-efficient solutions than the state-of-art approach RT-IFTTT [15]; (5) It incurs acceptable overhead in terms of program space and memory space.

```
1  TinyApp SmartLED{
2  Interface:
3    TL_Data LIGHT,PIR; TL_Service bool TurnOnLED();
4  Program:
5    void setup(){
6      TL_Connector.bind(TL_WiFi);
7      TL_WiFi.join("SSID","PASSWD");
8      LIGHT.bind(TL_Light); PIR.bind(TL_PIR);
9    }
10   bool TurnOnLED() {return TL_LED.turnOn();}
11 }SL1,SL2,SL3;
12
13 TinyApp SmartDoor{
14 Interface: TL_Data GYRO;
15 Program:
16   ... //Configurations for GYRO sensor and the network
17 }SD;
18
19 Policy HomeOccupancy{
20 Interface: TL_Event EVT({"Home", "Out"});
21 Rule:
22   If(Any({SL1,SL2,SL3}.PIR.last("5min").avg()>0)){
23     EVT.trigger("Home");
24   } Else { EVT.trigger("Out");
25   } Within(30, 0);
26 }HO;
27
28 Policy DoorEvent{
29 Interface: TL_Event EVT({"Opening", "Closing"});
30 Rule: ... // Infer events from the Gyro data of SD
31 }DE;
32
33 Policy LightControl{
34 Rule:
35   If(DE.EVT.isTriggered("Opening") && Any({SL1,SL2,SL3
         }.LIGHT.last()<100) && TL_Time.hour>=17){
36     For(L In {SL1,SL2,SL3}) {L.TurnOnLED();}
37   }Within(10, 0.05);
38 }LC;
39
40 TinyApp ControlPanel@Client{
41 Program:
42   UI_Button B1; UI_Text T1;
43   void setup(){
44     B1.setText("Turn on LEDs");
45     T1.bind(HO.EVT.last()); TL_UI.append({B1,T1});
46   }
47   void B1.isPressed(){
48     For(L In {SL1,SL2,SL3}) {L.TurnOnLED();}}
49 }CP;
```

Figure 1: Code snippets of the smart home application.

We summarize the contributions as follows:

- We present TinyLink 2.0, a novel system which integrates device, cloud, and client side development of IoT applications. Its programming language enables developers to express diverse IoT application logic in an easy-to-use way.
- We propose an optimization approach combining compile time code partitioning and runtime task scheduling to minimize the energy consumption of IoT devices.
- We implement TinyLink 2.0 and extensively evaluate its performance. Results show that it can significantly speed up the IoT development while still achieving high expressiveness.

The rest of the paper is organized as follows. Section 2 introduces TinyLink 2.0 through a use case study. Section 3 presents its overview. Section 4 and Section 5 describe the design

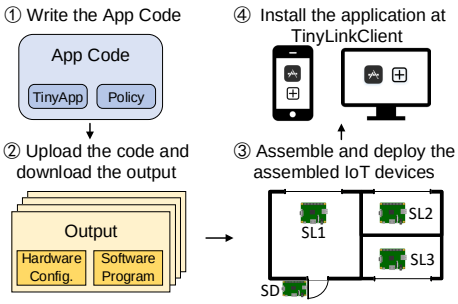


Figure 2: Usage workflow of TinyLink 2.0: 1) write the application code; 2) upload the code; 3) assemble and deploy devices; 4) install the application.

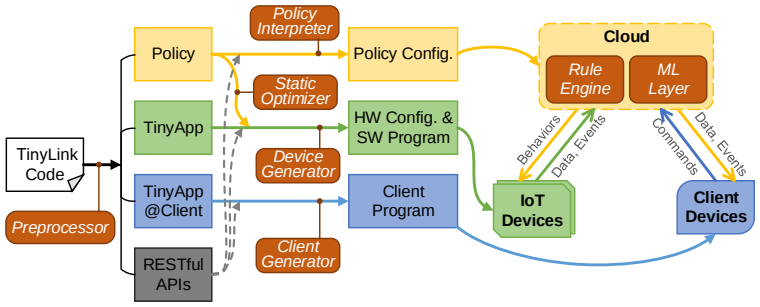


Figure 3: System overview of TinyLink 2.0. It generates energy efficient programs for the three sides at compile time. At runtime, it further optimizes the application performance.

and implementation details. Section 6 evaluates the performance extensively. Section 7 discusses some important issues and Section 8 presents the related work. Finally, Section 9 concludes the paper.

2 TINYLINK 2.0 USAGE

In this section, we develop a smart home application to show the usage of TinyLink 2.0. With this application, users can observe home occupancy of the house, remotely turn on/off LEDs via smartphones and even allow TinyLink 2.0 to automatically turn on all LEDs when users open the home door at dusk. Figure 1 shows the code snippets of this application, including the device side, the cloud side, and the client (e.g., mobile) side. Figure 2 shows the development process which contains the following four steps:

①**Write the application code.** With TinyLink 2.0 programming language, developers only need to describe key functionalities of IoT devices and clients in *TinyApps*, and the high-level application logic in *policies* without dealing with complex hardware drivers and data flows. Lines 1-11 show a TinyApp named `SmartLED` that has three instances, `SL1`, `SL2`, and `SL3`. Each one is used to generate an IoT device. Lines 2-3 describe the output data, events and services provided in the *Interface*. Lines 6-7 configure the WiFi network. Line 8 binds the data in the interface to IoT device functionalities (e.g., Light). Line 10 implements the service for turning on its LED. Lines 13-17 show a similar TinyApp, `SmartDoor`, which provides gyro sensor data. Lines 19-26 form a policy which detects home occupancy by analyzing the average PIR (Passive InfraRed) sensor value of the last five minutes from `SmartDoor`. Lines 33-38 show the policy that can automatically turn on all LEDs by using others' inferences and data. Lines 40-49 describe the TinyApp running at the client (i.e., mobile) side.

②**Upload the code and obtain the output software programs.** For each TinyApp instance on the device side, e.g., `SmartLED`, TinyLink 2.0 generates a hardware configuration, including a hardware component list and a connection figure, as well as the compiled software program for the hardware device. Similarly, TinyLink 2.0 also generates software programs for client sides. The cloud side programs include the main application logic, which can be generated from *policies*.

③**Assemble and deploy the assembled IoT devices.** Developers can assemble IoT devices by using the hardware configuration and burn the software program to the devices. Then the IoT devices can be deployed in desired places.

④**Install the application at TinyLinkClient.** Finally, developers install the application for the client side at TinyLinkClient, which is a container APP to host multiple IoT applications. Developers can observe visualizations of their application data and share them with others.

Unlike traditional developing approaches, developers need not explicitly specify the data flows and service calls. Moreover, they need not implement underlying details about when and how to retrieve data so that they can focus on the main application logic.

3 TINYLINK 2.0 OVERVIEW

We describe the design goals of TinyLink 2.0 and the methods we have adopted.

- **Rapid development**. TinyLink 2.0 intends to provide a simple and easy-to-use approach for non-experts to rapidly develop IoT applications.
- **Energy efficiency**. As the main application logic is expressed in policies running on the cloud, TinyLink 2.0 should control IoT devices' behaviors (e.g., sensing and uploading) in an efficient manner.
- **Reconfigurability**. Based on the abstractions like TinyApp models, developers can replace deployed IoT devices with new ones whose abstractions are the same at runtime, even those that are developed by other IoT development systems. It is the same case for policies.

Figure 3 shows the system overview of TinyLink 2.0. At compile time, it analyzes the application code and generates the programs for the three sides. The static optimizer will optimize the application performance, which will be described in the implementation section in detail. At runtime, TinyLink 2.0 will exploit the cross-side optimization potential to further improve the energy efficiency performance of the IoT application.

4 SYSTEM DESIGN

In this section, we will first present the design of the programming language and highlight the features which enable integrated development. Then we will describe how TinyLink 2.0 generates IoT devices' behaviors from policies in an energy-efficient manner, including details about the problem formulation and its solution.

4.1 Programming Language

TinyLink 2.0 uses a structured DSL language for IoT application programming, including the device side, the cloud side, and the client side. It provides developers with a TinyApp model which is the key to enable implicitly specified interactions (e.g., data flows

and service calls) across different sides and allows the cloud to define device behaviors. We present the important features in the following.

TinyApp model. It defines an IoT device's capabilities and behaviors. To be more specific, it abstracts an IoT device's sensing data, detected events and available service calls which are exposed to other TinyApps and policies. For example, `LIGHT`, `PIR`, and `TurnOnLED` in the TinyApp `SmartLED` of Figure 1 are exposed. On the other hand, how to sample sensor data and implement service calls in the `Program` are hidden from developers. They can quickly realize what a TinyApp provides and write interactions between it and others (i.e., TinyApps and policies).

Implicit interaction. In order to achieve rapid development, we analyzed 101 commonly-used IoT projects published in popular IoT communities such as [3, 4, 7]. We find that large amounts of engineering efforts (e.g., about 45% of lines of code) are spent on the interactions (or their preparations) such as formatting data, transmitting over HTTPS or MQTT [30], and calling services from mobile phones to IoT devices, etc. To alleviate this, TinyLink 2.0 adopts a structured way to implicitly specify the interactions with the help of TinyApp models. It allows developers to retrieve other devices' data defined in `Interface` like retrieving member data of a C++ class on a local device, as shown in line 22 of Figure 1. It is a similar case for invoking service calls. TinyLink 2.0 implements the detailed code for interactions during compilation. Moreover, by keeping track of each IoT device status, it can maintain the consistency of data flows and service calls. For example, if a service call is invoked when an IoT device is offline, TinyLink 2.0 will retransmit it once the device is back online.

Cloud-centric device behavior. One important feature of TinyLink 2.0 is that it shifts the main application logic from IoT devices to *policies* running on the cloud and allows the cloud to take full control of device behaviors at runtime. Our major challenge is how to abstract the logic among multiple IoT devices and express them in terms of policies in an easy way. Existing approaches use IFTTT programming [33, 34] to express the application logic, which is simple and effective. Up to 2015, [34] had attracted 106,452 authors who created 224,590 IFTTT programs that had been added by 11,718,336 end-users and the numbers were increasing dramatically.

However, two issues arise when we directly adopt IFTTT programming in TinyLink 2.0: (1) its limited keywords make writing the logic across multiple TinyApps tedious and error-prone; (2) it only contains high-abstracted application logic without the management of IoT devices' behaviors. For the first issue, TinyLink 2.0 enhances IFTTT capabilities by adding new keywords, e.g., *All* and *Any* to simplify writing rules among a set of TinyApp instances with the same interface, *For* and *In* to traverse the set of TinyApp instances. For example, line 22 of Figure 1 should be written as follows.

```
If(SL1.PIR.last("5min").avg()>0 || SL2.PIR.last("5min")
     .avg()>0 || SL3.PIR.last("5min").avg()>0){
```

For the second issue, TinyLink 2.0 can automatically generate behaviors, in terms of task schedules that instruct IoT devices to do specific tasks at the proper time, for each IoT device from the implications in policies. Tasks schedules can be generated from IFTTT-like rules with the keyword *Within* which specifies the deadline and the miss ratio [15]. The details are described in Section 4.2.

Table 1: Keywords of TinyLink 2.0.

Keywords	Description
TinyApp	A complete piece of code run on IoT devices and client devices.
Policy	It describes the application logic among TinyApps.
Interface	It describes the public interfaces for inner data, events and services.
Program	It contains the main program, including services and configurations.
Rule	It describes detailed rules of the policy via extended IFTTT syntax.
Model	It describes the configuration of virtual sensors.
If, Else	IFTTT keywords for conditional executions in rules.
All, Any	IFTTT keywords for expressing logic on a set of elements.
For, In	IFTTT keywords which executes a for loop over a set.
Within	IFTTT keywords for specifying deadlines and miss ratios of rules.
Require	It specifies user requirements such as enabling debugging.
Import	It imports the source code of other TinyApps and policies.

Virtual sensor. Another important issue to impact the development speed is the inference model design for sensing data, e.g., detecting the door events from gyro sensors. This is because: (1) Non-experts usually do not know which sensors are critical to making specific inferences and what the relationships are between the inferences and the sensing data (e.g., gyro sensor detects 3-axis data); (2) The deployment circumstances may be critical to the inferences, which means it is difficult for non-experts to write robust code against various deployment circumstances. Take the policy `DoorEvent` in Figure 1 as an example. It uses the sensing data of a gyro sensor which detects rotations around 3-axis. On a swing door, turning upside down the gyro sensor in deployment will lead to the opposite results since it reverses the sign of the sensing data. If it is deployed on a sliding door, the inference could not work at all due to no rotations.

To solve this problem, TinyLink 2.0 provides developers with *virtual sensors* written in the `Model`. The code for the policy `DoorEvent` in Figure 1 can be written as follows.

```
Policy DoorEvent {
Interface: TL_Event EVT({"Opening", "Closing"});
Model: TL_VirtualSensor vs;
  vs.setOutput(EVT);vs.setInput({SD.GRYO,SD.ACC,SD.MAG});
}DE;
```

In the TinyApp `SmartDoor`, we write additional code to bind functionalities `TL_Accelerometer` and `TL_Magnetometer` to data `SD.ACC` and `SD.MAG`. Taking these code, TinyLink 2.0 enables developers to draw door event inferences from the combination of sensing data by automatically generating an inference model. Virtual sensors need initialization to record samples and train models by clicking the UI buttons on the client side. TinyLink 2.0 uses several widely-used classifiers, e.g., Support Vector Machine (SVM), random forest, and K-Nearest Neighbors (KNN), to train models and chooses the most accurate one to use. Developers can also add their own features or models via TinyLink 2.0 APIs.

The benefits of using virtual sensors are: (1) Developers do not need to write sophisticated code for inferring from sensing data; (2) The influence of deployment circumstances has been trained into the model. TinyLink 2.0 will give guidance on which sensors are positive or negative to the inferences.

Summary. Table 1 summarizes the keywords of TinyLink 2.0. With them, it can accomplish multiple task operations in order

Table 2: Comparison of task operations among TinyLink 2.0 and three state-of-the-art systems, RT-IFTTT, Beam, and TinyLink.

Functionality	TinyLink 2.0	RT-IFTTT [15]	Beam [29]	TinyLink [12]
Poll sensor data	✓	✓	✓	
Push sensor data	✓		✓	✓
Poll stream sensor data	✓		✓	
Push stream sensor data	✓		✓	✓
Actuator control	✓	✓	✓	✓
Data storage	✓	✓	✓	✓
Trigger events from sensor data	✓		✓	✓
Inference from sensor data	✓		✓	✓
Dynamically alter polling/pushing intervals	✓	✓		
Multi-device interaction	✓	✓		
Data visualization	✓			
Virtual sensor	✓			

to express diverse IoT application logic. The currently supported task operations are shown in Table 2. We compare TinyLink 2.0's supported task operations with other three state-of-the-art IoT frameworks, RT-IFTTT [15], Beam [29] and TinyLink [12]. Compared with RT-IFTTT, TinyLink 2.0 enables more task operations, e.g., multiple sensor data retrieving methods (e.g., pushing sensor data), triggering events and drawing inferences from sensor data, as well as visualizing the data. Compared with Beam and TinyLink, TinyLink 2.0 supports multi-device interactions, virtual sensors, dynamically altering polling/pushing intervals and visualizing sensor data.

4.2 Performance Optimization

We describe how the dynamic optimizer generates tasks for instructing IoT devices at the proper time and how it optimizes the task schedules as mentioned in Section 4.1. When the policies are written in IFTTT-like rules (i.e., condition-action pairs), TinyLink 2.0 only needs to find the proper time for acquiring conditions because the actions are invoked only if the conditions are true. A naive method is to poll the sensor data with fixed intervals, which could be a waste of energy since the conditions may be false for a long time. A recent state-of-the-art approach, RT-IFTTT [15], proposes a real-time solution to dynamically calculate efficient condition-aware polling intervals for each sensor. However, RT-IFTTT has two key drawbacks. First, it assumes that a device can only upload the sensing result from a single sensor at a time. This assumption makes the optimization problem easier to solve, but fails to achieve better solutions where multiple sensing results are uploaded in a single uploading message. Thus, it couples the sensing and uploading, i.e., one sensor reading requires one data polling. Moreover, TinyLink 2.0 provides more available task operations for transmissions and have full control of them, e.g., pushing/polling and streamed pushing/polling while RT-IFTTT only supports polling as shown in Table 2.

For example, a Mosaic [11] IoT device monitors urban air quality by sampling and uploading temperature, humidity, and PM2.5 data periodically. Suppose at one moment the maximal polling intervals are 15s, 21s and 32s, respectively. Figure 4 shows the timeline when the device uses maximal polling intervals. The power consumption

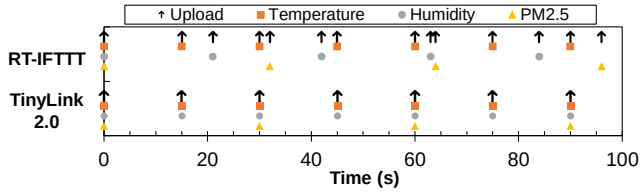


Figure 4: Timeline of uploading sensor data by using RT-IFTTT and TinyLink 2.0.

is about 17.61mW. Intuitively, we can reduce transmission times by decreasing humidity and PM2.5 intervals to 15s and 30s, respectively. We observe the power consumption drops to 9.07mW, reducing about 48.5%. The actual problem of choosing these intervals is much more complex due to the heterogeneity of IoT sensors and radios, and the dynamics of maximal intervals.

Problem Formulation. We present the notations for the sensing/transmitting schedule generation problem formulation.

- $\boldsymbol{D} = \{d_1, d_2, ..., d_M\}$ is the set of M IoT devices. IoT device d_i includes N_i sensors. $\boldsymbol{S_i} = \{s_{i1}, s_{i2}, ..., s_{iN_i}\}$ is the set of sensors on IoT device d_i.
- $\boldsymbol{T_i^s} = \{t_{i1}^s, t_{i2}^s, ..., t_{iN_i}^s\}$ is the set of current sampling intervals for the set of sensors $\{s_{i1}, s_{i2}, ..., s_{iN_i}\}$. Since we can combine the data transmissions of different sensors when they sample at the same moment, we use t_i^d to represent the average transmission interval of d_i.
- $\boldsymbol{E_i^s} = \{e_{i1}^s, e_{i2}^s, ..., e_{iN_i}^s\}$ is the set of energy consumption of each sensor data sampling on IoT device d_i.
- e_i^d denotes the energy consumption of transmitting a message from d_i. In IoT scenario, a data transmission usually involves radio powering on/off and connecting to the gateway/server, the number of transmissions has much more significant impact on the energy consumption, compared with the message length. Therefore, for simplicity, we assume that the energy consumption of transmitting a message only depends on d_i.

With the above notations, we can formulate the schedule generation problem as the following optimization problem with the criterion being the power consumption of the sensor data sampling and uploading operations.

$$\begin{aligned} &\text{Find the values of all } \{\boldsymbol{T_1^s}, \boldsymbol{T_2^s}, ..., \boldsymbol{T_M^s}\} \\ &\min \sum_{i=1}^{M} (\sum_{j=1}^{N_i} e_{ij}^s / t_{ij}^s + e_i^d / t_i^d) \\ &\text{s.t.} \begin{cases} 1 \le t_{ij}^s \le t_{ij}^{\max}, & \forall s_{ij} \in \boldsymbol{S_i} \\ t_i^d = DTI(\boldsymbol{T_i^s}), & \forall d_i \in \boldsymbol{D} \end{cases} \end{aligned} \tag{1}$$

Note that the resolution (i.e., minimal time unit) of all intervals is set to 1 second in our formulation for simplicity. The first constraint means that the used sampling interval t_{ij}^s should be smaller than the maximal sampling interval $t_{ij}^{\max}$ which mainly depends on the sensor data updating requirement of the application. The second constraint calculates the average transmission interval t_i^d for IoT device d_i, given its sensor sampling intervals $\boldsymbol{T_i^s}$, i.e., $DTI()$ (device transmission interval). Considering that multiple sensor samples may be combined into one message and be transmitted at the same time, the DTI calculation is a non-trivial task. We solve this problem

Algorithm 1: Heuristic algorithm for calculating sampling intervals.

Input: All of the maximal sensor sampling intervals $\{T_1^{\max}, T_2^{\max}, \ldots, T_M^{\max}\}$, all of sensor energy consumptions $\{E_1^s, E_2^s, \ldots, E_M^s\}$ and all of the device transmission energy consumptions $\{e_1^d, e_2^d, \ldots, e_M^d\}$

Output: $\{T_1^s, T_2^s, \ldots, T_M^s\}$

Function DEC(T_i^s, E_i^s, e_i^d) *// Calculate the device power consumptions of data transmitting and sensor sampling*
  return $\sum_{j=1}^{N_i} e_{ij}^s / t_{ij}^s + e_i^d$ / DTI(T_i^s);
$T^S \leftarrow \emptyset$;
for $i := 1$ ***to*** M ***step*** 1 **do**
  $T_i^{\min} \leftarrow T_i^{\max}$;
  for $k := 1$ ***to*** $\min(T_i^{\max})$ ***step*** 1 **do**
    $T_i' \leftarrow \emptyset$;
    forall $t_{ij}^s \in T_i^{\max}$ **do**
      $T_i' \leftarrow \lfloor t_{ij}^s / k \rfloor \cdot k \cup T_i'$;
    if DEC($T_i^{\min}, E_i^s, e_i^d$) >DEC(T_i', E_i^s, e_i^d) **then**
      $T_i^{\min} \leftarrow T_i'$;
  $T^S \leftarrow T^S \cup T_i^{\min}$;
return T^S;

by using the Venn diagram formula [27], and omit the details due to the page limit.

Then we focus on the optimization problem. Since the calculation of $DTI()$ is non-linear, the optimization problem is a non-linear integer programming (NIP) problem, which is NP-hard [14]. To find the near-optimal solutions, we propose a heuristic algorithm as shown in Algorithm 1. For each device, the algorithm first finds the smallest sampling interval of all maximum sensor sampling intervals, i.e., $\min(T_i^s)$. For example, in Figure 4, the smallest interval is 15 for the interval set {15, 21, 32}. Then it traverses all sampling intervals ranging from 1 to $\min(T_i^s)$ (e.g., [1, 15]) as the candidate of the transmitting interval of that device. For each candidate, the algorithm calculates the largest sampling interval for each sensor where the sampling interval is divisible by the candidate interval. In this example, when the candidate interval is 15, the largest sampling interval for the third sensor is 30. Given the largest sampling intervals for all sensors, the algorithm search for the set of intervals with the minimal sampling/transmitting power consumption for each device. The time complexity of this heuristic algorithm is $O(\sum_{i=1}^{M} \min(T_i^s) \cdot N_i)$. When the smallest sampling interval and the number of sensors are bounded values, the time consumption of this algorithm is almost linear to M, i.e., the number of devices, making it scalable to applications with a large number of devices. In the evaluation section, we will show that this relatively simple heuristic algorithm can achieve satisfactory performance under various settings.

5 SYSTEM IMPLEMENTATION

In this section, we present details about how a piece of application code is compiled into programs of the three sides. Due to the page limit, we omit the detailed description of several components, e.g., preprocessor, and focus on the key components of TinyLink 2.0.

Device side. As discussed in Section 4.1, TinyLink 2.0 generates task schedules for heterogeneous IoT devices. A possible approach is to use virtual machines (VMs) to run arbitrary program code [26]. Clearly, the VM approach is flexible, but it suffers from drastic slowdown compared to native code, especially on resource-constrained IoT devices. Also, to the best of our knowledge, there is no unified VMs for heterogeneous IoT devices, which needs great engineering efforts. Therefore, TinyLink 2.0 uses another approach in which IoT devices execute pre-defined functions in native code. On the device side, a task handler containing specific state machines is implemented to execute instructions contained in messages from the cloud. For each data and service calls in the interface, TinyLink 2.0 binds an execution state and a corresponding pre-defined function to it.

Moreover, TinyLink 2.0 builds on top of a device generation system, TinyLink [12], which can generate IoT applications from hardware-independent C-like code. The language for the TinyApp part is an embedded domain-specific language (eDSL) [9] of TinyLink language that: (1) it adds DSL elements such as macros and data types (e.g., `Service`); (2) it implements numerous task handlers to facilitate interactions and behavior managements of IoT devices; (3) its high-level API wraps TinyLink's APIs by hiding unimportant procedures, e.g., uploading data via WiFi in Figure 1; (4) it enriches TinyLink's functionalities, e.g., support of LoRa and concurrent execution.

TinyLink 2.0's *device generator* builds abstract syntax trees of the application code by using parser generator ANTLR [24]. Then it translates TinyLink 2.0 code into TinyLink code via its simple finite-state machine written in Python. The translated code will be processed by TinyLink to generate hardware configurations and binary programs. Moreover, to enable concurrent execution on low-end IoT devices [13] (e.g., Arduino Mega) which use single-threaded programming, TinyLink 2.0 incorporates protothreads [28] to create stackless lightweight threads for concurrent executions.

Client side. TinyLink 2.0 generates client programs from client side code (e.g., TinyApp with the postfix `@Client`). For simplicity, TinyLink 2.0 uses web pages as the client side program, e.g., WebViews on Android phones. It first builds the abstract syntax tree of the client code and then translates the interactions to JavaScript (JS) code, e.g., sensor data retrieval into RESTful API request by using `$.post`. Besides, it translates the UI widgets to mixed elements by using HTML and JS with TinyLink 2.0's default template. Finally, the combined WebView code is stored in the cloud and can be accessed through the container app TinyLinkClient.

Besides, TinyLink 2.0 supports successive advanced development by allowing hybrid development, i.e., using its approach in combination with other development tools. For example, developers can: (1) build device side software based on systems other than TinyLink, e.g., using Microsoft Azure IoT to develop IoT devices; (2) customize the client side program to allow more sophisticated UI interfaces, e.g., uploading their own WebView templates which include customized style sheets and JS code. The preprocessor automatically transforms data flows and service calls in `Interface` into RESTful APIs (e.g., MQTT topics and HTTPS links). Developers can customize their own device and client programs by using other tools as long as they use the allocated RESTful APIs.

Static optimizer by code partitioning. By analyzing policies, TinyLink 2.0 aims to further improve the efficiency of the application by using domain knowledge. This is based on the insight that some policies may hamper the performances of TinyApps if they are processed in the cloud. Instead, they should be shifted to the device and compiled as part of the device side program. For example, policy `DoorEvent` in Figure 1 requires the gyro sensor to continuously transmit its data with an interval of 50*ms*. The interval is smaller than the estimated transmission time via WiFi (e.g., 74*ms* on average [18]), which may cause missing deadlines. Therefore, static optimizer shifts the rule to the TinyApp `SmartDoor` at compile time as shown in the following code snippets.

```
Policy HomeOccupancy{          | TinyApp SmartDoor{
Rule:                          | Program:
If(SD.GYRO.last("z")<-20){     | void loop() {
  EVT.trigger("Opening");      |   TL_Gyro.read();
}Within(0.05, 0);              |   if(TL_Gyro.data("z")<-20)
If(SD.GYRO.last("z")>20){      |     EVT.trigger("Opening");
  EVT.trigger("Closing");      |   if(TL_Gyro.data("z")>20)
}Within(0.05, 0);              |     EVT.trigger("Closing");
                               |   TL_Time.delayMillis(50);}
```

To achieve this, the static optimizer of TinyLink 2.0 first extracts a tuple <TinyApp instance, conditions, deadline> for each rule in policies. Then it searches the TinyLink 2.0 database for average transmission time of the network which is set to values reported in articles like [18]. If the deadline is smaller than the average transmission time, it will: (1) shift the rule in policies to IoT device program code if the condition and the action contain interactions from only one TinyApp instance; otherwise (2) throw warnings about the potential of missing deadlines. For the former, it uses a translator similar to device generator and adds the translated device code snippets to the main loop.

Cloud side: ①*policy interpreter.* Policies run in the cloud and they are the key to the application logic. There are roughly two methods to execute policies, compiled as binaries and executed, or translated as scripts and interpreted at runtime. Considering the requirements of scalability and user modifiability at runtime, TinyLink 2.0 adopts the latter. Policies are translated into rule configurations in JSON format. The rule conditions are analyzed and interpreted to postfix expressions together with RESTful APIs, so do the service calls in rule actions.

Cloud side: ②*rule engine.* The rule engine runs at the cloud, which is same as the dynamic optimizer. It can process a large number of rule configurations and registered triggers for each rule. There are two kinds of triggers, event-based triggers (e.g., the arrival of sensor data and events) and time-based triggers (e.g., `If(TL_Time.hour >= 17)`). Rule engine uses three threads, each with one simple FIFO message queue implemented by Redis, including: (1) the message queue which stores uploaded sensing data and events to the MySQL database and searches for new triggers, (2) the rule queue which judges conditions of the triggers, (3) and the command queue which sends commands in the rule action. We modify the source files of the MQTT server, mosquitto [20] to push the involved rules into the rule queue when a message arrives. For judging each condition, the thread with the rule queue uses the latest sensor data which are still valid in the time window (the default is 60 seconds).

Cloud side: ③*Machine learning layer.* Machine learning layer (MLL) facilitates the main functionalities of virtual sensors.

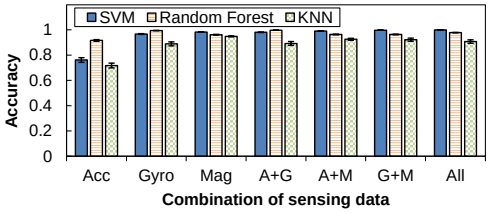


Figure 5: Accuracy of door event inferences using SVM, random forest and KNN.

Table 3: List of hardware components, servers, PC and mobile phones that are used in the evaluation.

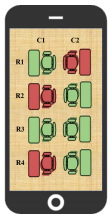
Mainboard	Arduino UNO, Arduino Mega2560, LinkIt One, Raspberry Pi 3 Model B+, and BeagleBone Black
Shield	Base Shield V2, Grove Mega Shield V1.2, WiFi Shield V2, SD Card Shield V4, and Grove Pi+
Peripheral	ESP8266 WiFi module, Grove UART WiFi, Grove IMU 9DoF sensor, Grove ultrasonic ranger sensor, Grove PIR sensor, Grove light sensor, Grove digital light sensor, Grove temperature and humidity sensor (DHT11), Soil temperature sensor (DS18B20), Grove soil moisture analog sensor, PPD42NS dust sensor, Grove Chainable RGB LED, LED bulb, motor, relay, etc.
Cloud Server	Intel(R) Xeon(R) Platinum 8163 CPU@2.50GHz, 2GB memory, and a 40GB HDD disk
Desktop PC	Intel Core i7-7700 CPU@3.60GHz, 8GB memory, and 1TB HDD disk
Smartphone	iPhone 7 and Huawei Honor 8

There are four phases for drawing inferences by using virtual sensors: (1) MLL loads the inputs and outputs of virtual sensors from rule configurations. MLL generates task schedules that sample data for initialization, and registers them on the rule engine. (2) After deploying the IoT devices that provide the sensing data, developers can use the client side program to record the training data, including the raw sensing data and the virtual sensor outputs manually labeled by the developers. (3) Then MLL extracts significant statistical features from the data. MLL trains these data frames using different classifiers including SVM, random forest, and KNN. (4) Finally, MML chooses the trained model with the best accuracy during cross-validation and deploys it on the cloud. MLL also registers triggers of the input sensing data in the rule engine, and waits for classification requests from it. We implement the MLL using Python with scikit-learn library [25]. In addition, developers can add their own features or alter the machine learning models by downloading TinyLink 2.0's template Python script, modifying it with scikit-learn APIs, and uploading it to the system.

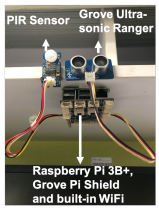
We revisit the example of the TinyApp `SmartDoor` and the policy `DoorEvent`, and deploy the IoT device on a swing door where the device's x-axis is vertical to the door and z-axis is vertical to the ground. Figure 5 shows the accuracy of the three classifiers among different combinations of the accelerometer, the gyro, and the magnetometer. We observe that SVM using all the sensors can achieve the best accuracy. Note that ideally the virtual sensor mechanism can be applied to infer any user-defined events from arbitrary sensors, but the inference accuracy varies depending on the correlation between the desired events and the raw sensor data.

6 EVALUATION

In this section, we evaluate TinyLink 2.0 from different perspectives. We first briefly introduce the experiment setup. Then we describe case studies and a user study in detail. Afterward, we evaluate



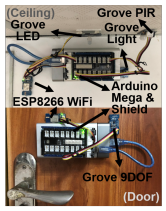
(a) Client Program



(b) IoT Device

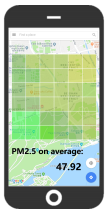
Figure 6: Smart seat application.

(a) Client Program

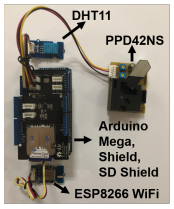


(b) IoT Device

Figure 7: Smart door application.



(a) Client Program



(b) IoT Device

Figure 8: Mosaic application.

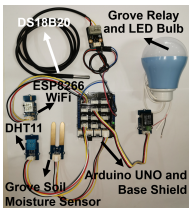


Figure 9: IoT device of FarmBot App.

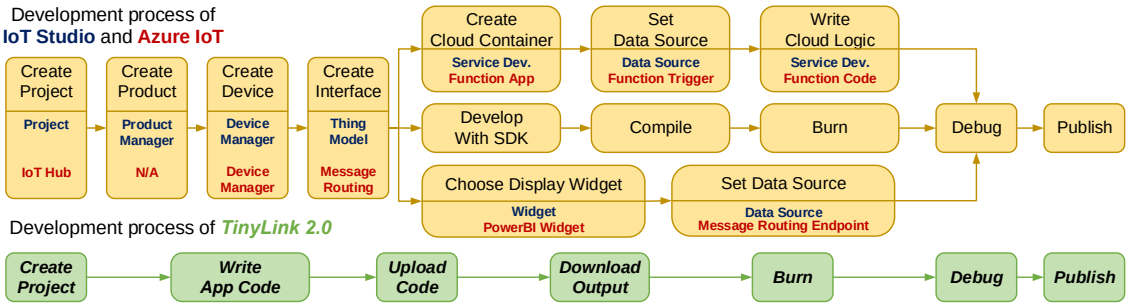


Figure 10: Development processes of IoT Studio, Azure IoT and TinyLink 2.0.

the programming language and the performance of TinyLink 2.0. Finally, we show the overhead of TinyLink 2.0.

6.1 Experiment Setup

In our experiments, we use four kinds of different mainboards, five shields and several sensors and actuators to build IoT platforms. Table 3 shows the detailed information of these hardware components, as well as the configurations of cloud server, desktop PC and smartphones used in the evaluation. We deploy the TinyLink 2.0 system, which is encapsulated in multiple Dockers, on the cloud server. In addition, we use a local desktop PC for running trace-based simulations and random tests. By using two Monsoon power monitors [31], we can measure fine-grained energy consumptions of IoT devices.

6.2 Case Studies

We design and implement eight representative IoT applications with TinyLink 2.0. We list their application features and descriptions in Table 4. Figures 6, 7, and 8 show three representative applications with their assembled IoT devices and client programs. Others are omitted due to the space limit. The IoT devices and the policies of all eight cases are developed purely by using TinyLink 2.0. As mentioned in Section 5, it provides basic WebView templates for customization which will then be compiled with TinyLink 2.0's RESTful APIs. The client programs in Figure 7 and 8 are customized by modifying styles and using external maps.

In order to evaluate the cost of TinyLink 2.0's integrated development, we find an "expert implementation" from Arduino Project Hub [22] and implement the application `FarmBot`. Figure 9 shows the IoT device implemented by using TinyLink 2.0, and Table 4 shows the implementation information. We omit the figure for the client program because it is very similar to Figure 6(b). We can observe that TinyLink 2.0 achieves ease-of-use and rapid development because it uses only 92 lines of code to implement the application, while the expert one requires 328 lines of code for the device side, 10 fields of settings for the cloud side, and 260 drag-and-drop logic blocks for the client side. TinyLink 2.0's application achieves almost the same functionalities of the original version, including sampling environment data, controlling actuators, and sending alerts to users. However, there is one slight difference between the two implementations, i.e., the original IoT implementation may perform better than TinyLink 2.0's implementation in terms of network performance. This is because the original one can tune the TCP/IP layer parameters by using low-level APIs (i.e., AT commands [36]), while TinyLink 2.0 uses the default parameters.

6.3 User Study

Methodology. To evaluate how TinyLink 2.0 achieves rapid development, we conduct a user study using three IoT development platforms, TinyLink 2.0 and two representative platforms, Microsoft Azure IoT and IoT Studio [6]. The user study contains two parts. In the first part, for each platform, we provide participants with a step-by-step user manual to guide them to implement an IoT application. It is a simplified version of the example in Figure 1 which includes a `SmartLED` TinyApp, a `HomeOccupancy` policy, and a `ControlPanel@Client` client. They just need to follow the steps and use the application code that we have provided. In the second part, we guide the participants to complete the example in Figure 1 by themselves. We provide them with abundant resources like API references and hardware components. Participants need to record the timestamps of important steps according to instructions in the user guide and leave feedbacks in the end. In addition, we capture screen videos during the experiments for further analysis.

Figure 10 shows the overall development processes of the three platforms. In Azure IoT, we use the IoT Hub and the Azure Python SDK to implement IoT device code, use the Message Routing, the Function App and the Function Trigger to implement its rule engine and set up the dataflow, and use the PowerBI to visualize data. On the other hand, we use the IoT Studio and the AliOS Things SDK

Table 4: Real-world applications implemented by TinyLink 2.0.

Application	TinyApp	Policy	Lines of Code	IoT Device	Description
Mosaic [11] - Air Quality Monitoring	2	2	41	8	It monitors the air quality of a selected area by deploying IoT devices on mobile vehicles. Its client displays the air quality data on the map and alerts people if the pollution exceeds a threshold.
Baby Care	2	4	103	2	It monitors the sleeping status of a baby by detecting sound, movements and ambient light. It can report to parents if any abnormal events happen by sending text messages and making phone calls.
Intelligent Parking	2	5	90	4	It helps users to find a vacant lot in two example parking lots. Its policy infers lot occupancy status by using an ultrasonic ranger sensor and a magneto-meter sensor. It displays current status of all lots.
Smart Hanger	2	4	124	1	It can intelligently notify users whether it is a good time to hang their clothes according to its weather inferences inferred from fetched data of local weather reports and its sampled humidity data.
Smart Home	3	3	78	4	This case has been described in Section 2.
Smart Plant	2	12	197	1	It can detect ambient environment (e.g., soil humidity), infer accidents (e.g., falling and turning over) of a flowerpot from an IMU 9DoF sensor and report the status to the client side program.
Smart Seat	2	6	156	8	It checks whether a seat is available in a room by using the PIR sensor and the ultrasonic ranger sensor, visualizes the room seat status and notifies users if a new seat is available.
FarmBot	2	4	92	1	It monitors the environment (i.e., air/soil temperature/humidity, and light) of a farm, uploads the data, and sends alerts to users. It also allows users to control the LED bulb from the smartphone.

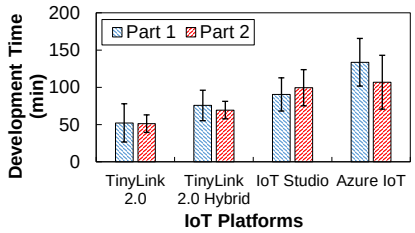


Figure 11: Development time of applications under different IoT platforms.

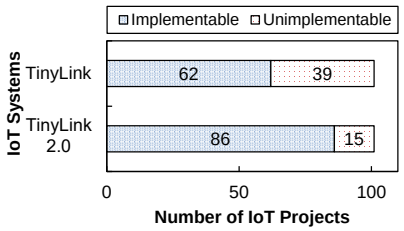


Figure 12: Implementation status of 101 IoT projects via different systems.

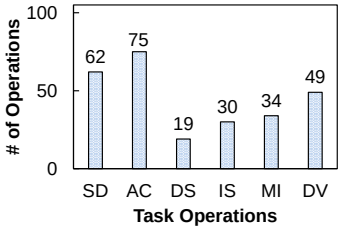


Figure 13: Breakdown of IoT project implementations.

to develop IoT device code, use the Service Development to specify service workflows which acts as a rule engine, and use the Web Visualization Development to visualize data. Unlike TinyLink 2.0, they both adopt separate development approaches and require clear specifications of the underlying services and interactions.

Furthermore, we build another platform where participants can customize TinyLink 2.0's device side development. Participants first write application code and leave `Program` part of the device side application empty. Then they generate the three side applications and retrieve the RESTful APIs of the device side. Afterwards, they can customize device side application by using their own hardware components, and development tools like Azure IoT instead of the default ones provided by TinyLink 2.0.

Developers in the user study. We recruit twenty volunteers, including six Ph.D. students, eight master students, and six undergraduate students. Six of them are well-experienced in IoT development, eleven of them have a little experience and three of them has almost no experience. We randomly select five volunteers for each platform. We collect the timestamps of important steps and analyze the data.

Quantitative results. Figure 11 shows the results. We can observe that the development time on average by using TinyLink 2.0 is about 52.2 minutes and 51.2 minutes for the two parts, respectively. When using IoT Studio and Azure IoT, participants need about 1.84x and 2.32x of TinyLink 2.0's development time on average, respectively. This is because TinyLink 2.0's integrated development, including implicit interactions and automatic generation of device behaviors, greatly reduces the development overhead. On the other hand, when using TinyLink 2.0's hybrid approach, participants need about 1.40x of TinyLink 2.0's development time on average. However, compared to IoT Studio and Azure IoT, the hybrid approach still increases development speed.

Feedbacks of Azure IoT. (1) Every participant complains about the complex development process and too many configurations, which are not user-friendly for non-experts; (2) Three of them encounter difficulties when analyzing the event streams in the Function App since they lack background knowledge of debugging on .NET framework and spend much time on it; (3) Two of them mention that the IDE contains too many undesired functionalities which make the desired ones difficult to find.

Feedbacks of IoT Studio. (1) Four of them mention that the drag-and-drop approach which specifies workflows in the Service Development is easy-to-use and user-friendly, and reduces the development overhead; (2) One of them thinks the drag-and-drop approach is a little bit tedious in the second part of experiments when she completes the first part and gets familiar with the system; (3) One of them says that the debugging logs are clear and helpful.

Feedbacks of TinyLink 2.0. (1) All of them say that the integrated development is very easy-to-use and user-friendly because the underlying details are hidden and less code needs to be implemented; (2) Two of them think that the application logic is clear and of good readability; (3) One of them suggests that providing more logs would be helpful.

Feedbacks of TinyLink 2.0 hybrid approach. (1) Three of them mention that this approach does accelerate the development process and it is great to give them more development choices; (2) Two of them think that it may be faster if they directly use TinyLink 2.0 without customization.

6.4 Programming Language

Expressiveness. We use the aforementioned 101 commonly-used IoT projects published on popular IoT community for evaluation. For each project, we try to implement it by using TinyLink 2.0 and mark it as implementable if the code can achieve the same functionalities, otherwise as unimplementable. We also execute the

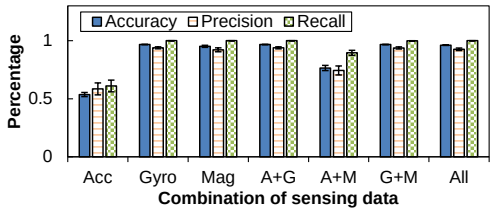


Figure 14: Inferences of swing door events when the IoT device is rotated 45 degree on the z-axis and 45 degree on the x-axis.

same procedure by using TinyLink [12]. Figure 12 shows the results. We can observe that 62 IoT projects are implementable by both TinyLink and TinyLink 2.0, while another 24 IoT projects are only implementable by TinyLink 2.0 because TinyLink lacks interactions with the cloud or the client. However, 15 IoT projects are still unimplementable by TinyLink 2.0 because they require additional hardware development. Nine of them require supplementing the circuit (e.g., connecting resistors or capacitors to sensors), seven of them require hand-built sensors or actuators (e.g., building specific LED matrix from light-emitting diodes), and three of them require both tasks. The last two require manufacturing PCB boards. These IoT projects cannot be implemented since both TinyLink 2.0 and TinyLink only use COTS hardware components.

In order to take an in-depth look, we break down the implementation of each project into TinyLink 2.0 operations in Table 2 and summarize the result in Figure 13. We can observe that Actuator Control (AC), Sensor Data acquisition (SD), and Data Visualization (DV) are the three most required operations, while Data Storage (DS), Inference from Sensor data (IS) and Multi-device Interaction (MI) are less used. Other operations are not used in these IoT projects. Therefore, TinyLink 2.0's easy-to-use APIs and UI designs can also satisfy developers' demands easily.

Virtual Sensor. We intend to evaluate whether the virtual sensor can draw correct inferences when the IoT device is deployed by developers at liberty. We use the smart door application and deploy IoT devices of TinyApp `SmartDoor` in three cases, 0-degree rotation around z-axis (i.e., x-axis is vertical to the door and z-axis is vertical to the ground), 45-degree rotation around the z-axis, as well as 45-degree rotation around z-axis and 45-degree rotation around x-axis, on a swing door. For each case, we gather thirty samples of opening the door and thirty samples of closing the door. Then for each classifier, we randomly select twenty samples from the sixty samples for training and different twenty samples for testing. In the first two cases, TinyLink 2.0 performs well with an accuracy of 98.44% on average, while in the last case, the overall accuracy degrades to 96.71%. Figure 14 shows the result of the last case. We observe that the accuracy of the accelerometer drops drastically to 53.53%. TinyLink 2.0 chooses the most accurate classifier automatically, thus it uses KNN. In addition, it will notify developers that the accelerometer may cause a negative impact on the inference and should not be used.

Furthermore, we repeat the same experiment on a *sliding* door by using the same setup. The gyro sensor causes a very negative impact on the accuracy, while the magnetometer achieves an accuracy of 100%. Since there is no rotation when sliding, the gyro can only detect noise. Thus TinyLink 2.0 gives guidance of not using the gyro. The two experiments show that the virtual sensor can achieve good applicability under different deployment circumstances.

Table 5: Lines of code for implementing applications.

Solution	Case	Device	Cloud	Client
TinyLink 2.0	Mosaic	12	18	11
	Smart Home	27	31	20
TinyLink and RT-IFTTT	Mosaic	39 TinyLink	32 RT-IFTTT	137 Java 218 H5/JS
	Smart Home	95 TinyLink	36 RT-IFTTT	137 Java 262 H5/JS
Native Code	Mosaic	103 C/C++	57 Python 261 C++	137 Java 218 H5/JS
	Smart Home	247 C/C++	74 Python 305 C++	137 Java 262 H5/JS

Lines of Code. We compare the lines of code needed to implement the Mosaic application and the smart home application via three methods, TinyLink 2.0, TinyLink with RT-IFTTT, and directly using native APIs, respectively. Besides device functionalities, these applications also include management of communication and database on the cloud, as well as data visualization and user interactions on the client. Table 5 shows the detailed result. TinyLink 2.0 can reduce the lines of code by 85.28% to 94.72% for the two examples, because it can automatically complete implicit data flows and service calls, and generate device behaviors from policies, which both need to be explicitly expressed by others.

6.5 Performance Optimization.

We evaluate the performance (i.e., energy consumption) of generated IoT device behaviors. First, we evaluate the one-time scheduling performance of TinyLink 2.0's heuristic algorithm by using random data. Then we evaluate its long-term performance by using both traces and real IoT devices.

We implement five baseline algorithms for comparison, the original RT-IFTTT algorithm [15], an optimal algorithm, a random walk algorithm, a random algorithm and a memetic algorithm [23]. The optimal algorithm uses a brute-force method that enumerates all possible solutions to find the solution with the minimal energy consumption. The random walk algorithm takes the maximal sensor sampling intervals as its initial position and stops when the step difference is smaller than $0.1s$. The random algorithm just randomly picks up a solution and repeats this for constant times (i.e., 10,000). The memetic algorithm initializes a population of 10 chromosomes and iterates selections 100 times where a local search is adopted.

Random data test. Considering an IoT device may possess multiple sensors, we set the number of sensors to range from two to ten, because ten sensors are enough to meet most IoT application requirements. For each number of sensors, we generate 10,000 random test units. In each test unit, the maximal sensor sampling intervals range from $1s$ to $3{,}600s$, the energy consumptions of sensors range from $1mJ$ to $1{,}000mJ$ and the energy consumptions of data transmission range from $1mJ$ to $1{,}000mJ$ as well.

Figure 15 shows the percentage of reduced energy consumptions on average of different algorithms compared with the RT-IFTTT algorithm, and Figure 16 shows their execution time on average. When there are two sensors, TinyLink 2.0's heuristic algorithm achieves the optimal solutions. As the sensor number increases to seven, it can still achieve near-optimal solutions, i.e., about 99.2% on average of the optimal energy consumptions. Other three

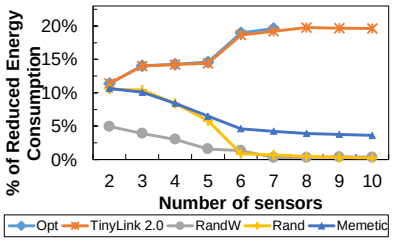


Figure 15: Percentage of reduced energy consumptions on average of different algorithms compared with the RT-IFTTT.

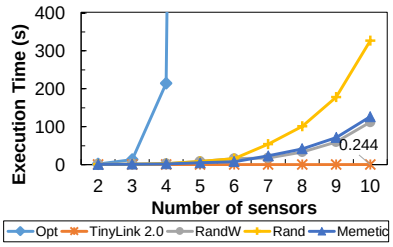


Figure 16: Execution time on average of different algorithms under different number of sensors.

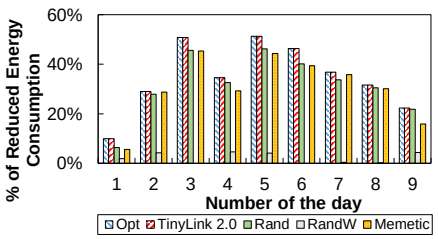


Figure 17: Reduced energy consumptions cumulated in a day of different algorithms compared with the RT-IFTTT.

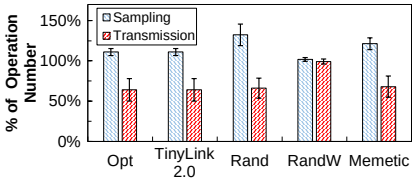


Figure 18: Percentage of operation numbers on average performed by different algorithms in a day.

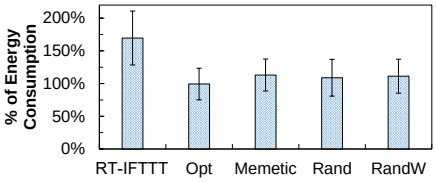


Figure 19: Percentage of energy consumptions of different algorithms compared with TinyLink 2.0.

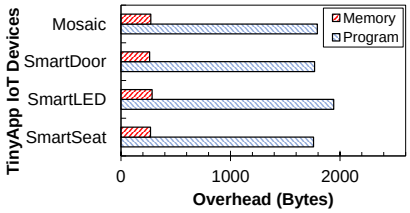


Figure 20: Program space overhead and memory space overhead.

Table 6: Energy consumption of operations.

Operation	Temp. Read	Humid. Read	Light Read	Sound Read	PM25 Read	Gyro Read	MQTT Pub&Sub
Energy (mJ)	1.13	1.12	0.02	0.03	6.25	0.03	139.23

baseline algorithm degrades quickly since the search space grows exponentially and becomes too large for them. The random walk algorithm does not perform well because energy consumptions between consecutive sensor intervals are usually discrete and change drastically, which hampers the algorithm's walking procedure.

Moreover, TinyLink 2.0's algorithm uses much less execution time than the other baselines algorithms. When the sensor number is five, it uses 8*ms* on average to execute TinyLink 2.0's algorithm, while the memetic algorithm, the random algorithm, and the random walk algorithm use 4,657*ms*, 8,198*ms*, and 8,516*ms* on average, respectively. When the sensor number is ten, it uses 244*ms* on average, which is still very small, while the others use more than hundreds of seconds on average. The time consumption of the optimal approach grows drastically as the sensor number increases. It takes more than one day when the number exceeds six.

Trace-based simulation. We intend to evaluate whether the one-time scheduling of the algorithms can perform well in the long term. We choose the `Mosaic` application as our test case. First, we measure the energy consumptions of its operations and Table 6 shows the results. The MQTT operation contains powering on and off the radio, connecting and disconnecting the MQTT broker, publishing a message of 100 Bytes, subscribing a topic and yielding a message of 100 Bytes. Then we assemble three IoT devices instanced from the `Mosaic` TinyApp. Each IoT device samples temperature, humidity, and PM2.5 data per second and stores the data along with the timestamps in an SD card. We deploy them in different places around our lab and retrieve three pieces of nine-day-long sensor data traces. Afterward, we write a simulator in Python that takes the traces and the measured energy consumptions as input. It outputs the number of operations, including sampling operations and data transmission operations by using different algorithms.

Figure 17 shows the reduced energy consumptions cumulated in a day of different algorithms compared with the RT-IFTTT algorithm. We can observe that TinyLink 2.0's algorithm can achieve the optimal cumulated energy consumptions in each day. This matches the observation from the random test. We can also observe that under various situations of different days in the long term, TinyLink 2.0's algorithm can also perform well. Figure 18 shows the percentage of operation numbers on average performed by different algorithms in a day. We can find that TinyLink 2.0's algorithm introduces more sensor sampling operations while reducing data transmission operations.

Real device test. We conduct the experiments on real IoT devices of the Mosaic application. We measure the energy consumptions of two IoT devices that use task schedules from TinyLink 2.0's algorithm and one of the baseline algorithms, respectively, by using two power monitors. We do five rounds of measurements and each lasts for two days. Figure 19 shows the cumulated energy consumptions of different algorithms compared with TinyLink 2.0's algorithm. We can observe that it achieves near-optimal performance on real IoT devices in the long term. It can reduce about 41.06% on average of the cumulated energy consumptions of the RT-IFTTT algorithm.

6.6 Overhead

We evaluate the overhead of TinyLink 2.0 from two aspects, the program space overhead and the memory space overhead on TinyApps. Since the Arduino Mega2560 uses an AVR MCU, we can calculate the program space by the sum of `.text` and `.data` segments. The measured memory space overhead is composed of the overhead from static memory overhead (i.e., `.text`, `.data`, etc.) and run-time memory overhead (i.e., stack and heap). Figure 20 shows the results. We can observe that the program space overhead on average is 1.816 KB, which is negligible compared with 256 KB of Flash. On the other hand, the memory space overhead on average is around 0.272 KB, which is acceptable compared with 8 KB of RAM.

7 DISCUSSION

Trade-off between simplicity and flexibility. To support rapid development, TinyLink 2.0 tradeoffs flexibility for simplicity to some extent. For example, the current implementation of TinyLink 2.0 does not support fine-grained concurrency control over the IoT device, e.g., the virtual sensor abstractions. In some critical applications where the developers want explicit control on program behavior, they can upload their own inference models instead of using TinyLink 2.0's predefined ones.

System scalability. TinyLink 2.0 targets for rapid development without introducing much overhead to the cloud side which is the key for system scalability. It indeed has overheads in cooperating between the cloud and device sides. In Section 4, we have mentioned that the time complexity of our heuristic algorithm is almost linear to the number of IoT devices, indicating that our system does not sacrifice much performance in terms of scalability.

Debugging problem. Developers can use detailed built-in logs for all TinyLink 2.0 APIs, and also can specify their own debugging information. Further, when developers find problems which are hard to debug at the API level, they can obtain the source code for detailed debugging (e.g., interactive debugging with GDB). Nevertheless, debugging an IoT application with interactions of three sides remains a very challenge problem.

Edge computing. The current implementation does not consider the computation capability of the edge server. It may cause problems when the connectivity to the cloud is intermittent or the delay is excessively long. In principle, TinyLink 2.0 can support edge computing by properly placing the rule execution on the edge server. We would like to investigate this direction in future work.

8 RELATED WORK

Rapid Development. TinyLink [12] is the state-of-the-art system for rapid development of IoT applications, which adopts a top-down approach that a developer can first write the application code and obtain the hardware configuration for device assembling and the software program. Both TinyLink 2.0 and TinyLink focus on accelerating the IoT development process. TinyLink 2.0 reuses TinyLink's solution to generate IoT hardware configurations. However, there are two important differences: (1) TinyLink is a device-side development system. But TinyLink 2.0 is an integrated development system for the three sides and focuses on its unique challenges. It provides an integrated programming language that includes features like implicit interactions, virtual sensors, and the automatic generation and the optimization of device's behaviors instructed from the cloud, while TinyLink mainly focuses on the hardware and software co-design of an IoT device. (2) TinyLink does not support the development of multi-device IoT applications, and does not consider complex interactions among three sides. The set of TinyLink's feasible hardware configurations is a subset of TinyLink 2.0's feasible hardware configurations.

TinyDB [21] is a distributed query processor that can collect data from a sensor network of motes which run on top of TinyOS operating systems. Its powerful runtime can automatically optimize the declarative SQL-like queries. DSN [5] is a declarative sensor network platform that aims to specify the sensor network system stack within a few lines of code. LibAS [32] is a cross-platform framework that enables rapid development of mobile acoustic sensing applications. Developers only need to implement the sensing signals and the callback function to handle each repetition of sensing signals. However, these declarative platforms focus on specific domains like network stack, data collection of sensor networks, and mobile acoustic sensing applications. TinyLink 2.0 focuses on the domain of rapid development of IoT applications. It faces unique challenges like integrating development of three sides and simplifying interactions.

Beam [29] is a framework that simplifies IoT applications by letting developers specify "what should be sensed or inferred". Beam introduces the key abstraction of an inference graph to decouple applications from the mechanics of sensing and drawing inferences. GIOTTO [2] is a safe, secure and easy-to-use open-source IoT infrastructure which can capture and store a large amount of user data, and provide data analytics services. However, TinyLink 2.0 can achieve the same effect by chaining multiple rules together and automatically drawing inferences from rules. It can give additional guidance on the IoT device generation. Furthermore, TinyLink 2.0 puts large amount of efforts in developing holistic IoT applications including programs of three sides and multiple task operations.

IFTTT Framework. RT-IFTTT [15] extends the existing IFTTT syntax for users to describe real-time constraints. It analyzes elements of all the applets, and dynamically calculates efficient polling intervals for each sensor. This interval reflects current sensor values, their related trigger conditions, and real-time constraints, with a sensor value prediction model. Liang *et al.* [19] provide a safety-centric programming platform for connected devices in IoT environments called SIFT. To simplify programming, users express high-level intents in declarative IoT apps. The system then decides which sensor data and operations should be combined to satisfy the user requirements.

Unlike the above work, TinyLink 2.0 not only focuses on how to write IFTTT rules, but also exploits the potential of rapid development of multiple devices, as well as the cooperative sensing and inference among all devices and the cloud. This can achieve more improvements in power efficiency, application logic expressiveness and fine-grained control of devices.

9 CONCLUSION

In this paper, we present TinyLink 2.0, a novel system which integrates device, cloud and client side IoT development. The programming language is expressive for diverse IoT applications, as well as considerations about energy efficiency and machine learning support. TinyLink 2.0 extends traditional IFTTT syntax by adding new keywords and syntax, and optimizes application performance with both static and dynamic optimizers. We carefully evaluate the performance by case studies, a user study, and experiments of all its important components. Results show that TinyLink 2.0 can significantly speed up IoT application development, while achieving high expressiveness and low overhead. Up to now, both TinyLink [12] and TinyLink 2.0 are supported by our IoT testbed, LinkLab (http://www.emnets.org/linklab) which allows remotely developing and experimenting various IoT applications.

ACKNOWLEDGMENTS

We thank our shepherd and reviewers for their constructive comments. This work is supported by the National Key R&D Program of China under Grant No. 2019YFB1600700, the National Science Foundation of China (No. 61772465 and No. 61872437), Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020001.

REFERENCES

[1] Joshua Adkins, Branden Ghena, Neal Jackson, Pat Pannuto, Samuel Rohrer, Bradford Campbell, and Prabal Dutta. 2018. The signpost platform for city-scale sensing. In *Proc. of ACM/IEEE IPSN*.

[2] Yuvraj Agarwal and Anind K Dey. 2016. Toward Building a Safe, Secure, and Easy-to-Use Internet of Things Infrastructure. *IEEE Computer* 49, 4 (2016), 88–91.

[3] Hackster.io an Avnet community. 2019. The community dedicated to learning hardware. https://www.hackster.io/about.

[4] AutoDesk. 2019. Instructables - Yours for the making. https://www.instructables.com/.

[5] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. 2007. The design and implementation of a declarative sensor network system. In *Proc. of ACM SenSys*.

[6] Alibaba Cloud. 2019. IoT Platform: Connect to Devices via Data Transmission. https://www.alibabacloud.com/product/iot.

[7] DFRobot. 2019. DFRobot - Quality Arduino Robot IoT DIY Electronic Kit. https://www.dfrobot.com/.

[8] Colin Dixon, Ratul Mahajan, Sharad Agarwal, AJ Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. 2012. An operating system for the home. In *Proc. of NSDI*.

[9] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A programmable programming language. *Commun. ACM* 61, 3 (2018), 62–71.

[10] Jon Froehlich, Joachim Neumann, Nuria Oliver, et al. 2009. Sensing and predicting the pulse of the city through shared bicycling. In *Proc. of IJCAI*.

[11] Yi Gao, Wei Dong, Kai Guo, Xue Liu, Yuan Chen, Xiaojin Liu, Jiajun Bu, and Chun Chen. 2016. Mosaic: A low-cost mobile sensing system for urban air quality monitoring. In *Proc. of IEEE INFOCOM*.

[12] Gaoyang Guan, Wei Dong, Yi Gao, Kaibo Fu, and Zhihao Cheng. 2017. TinyLink: A Holistic System for Rapid Development of IoT Applications. In *Proc. of ACM MobiCom*.

[13] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. 2016. Operating systems for low-end devices in the internet of things: a survey. *IEEE Internet of Things Journal* 3, 5 (2016), 720–734.

[14] Raymond Hemmecke, Matthias Köppe, Jon Lee, and Robert Weismantel. 2010. Nonlinear integer programming. In *50 Years of Integer Programming 1958-2008*. Springer, 561–618.

[15] Seonyeong Heo, Seungbin Song, Jong Kim, and Hanjun Kim. 2017. RT-IFTTT: Real-Time IoT Framework with Trigger Condition-aware Flexible Polling Intervals. In *Proc. of IEEE Real-Time Systems Symposium (RTSS)*.

[16] Yidan Hu, Guojun Dai, Jin Fan, Yifan Wu, and Hua Zhang. 2016. BlueAer: A fine-grained urban PM2.5 3D monitoring system using mobile sensing. In *Proc. of IEEE INFOCOM*.

[17] Texas Instruments Incorporated. 2013. TI: The IoT technology leader. http://www.ti.com/lit/ml/swpb013/swpb013.pdf.

[18] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. 2010. CloudCmp: comparing public cloud providers. In *Proc. of ACM SIGCOMM*.

[19] Chieh-Jan Mike Liang, Börje F Karlsson, Nicholas D Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. 2015. SIFT: building an internet of safe things. In *Proc. of ACM IPSN*.

[20] Roger A Light. 2017. Mosquitto: server and client implementation of the MQTT protocol. *Journal of Open Source Software* 2, 13 (2017), 265.

[21] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)* 30, 1 (2005), 122–173.

[22] MJRoBot. 2017. IoT Made Easy w/ UNO, ESP-01, ThingSpeak & MIT App Inventor. https://create.arduino.cc/projecthub/mjrobot/iot-made-easy-w-uno-esp-01-thingspeak-mit-app-inventor-da6a50.

[23] Pablo Moscato et al. 1989. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report* 826 (1989), 1989.

[24] Terence Parr and Kathleen Fisher. 2011. LL (*): the foundation of the ANTLR parser generator. In *Proc. of ACM Sigplan Notices*.

[25] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[26] Niels Reijers and Chi-Sheng Shih. 2018. CapeVM: A Safe and Fast Virtual Machine for Resource-Constrained Internet-of-Things Devices. In *Proc. of ACM SenSys*.

[27] Frank Ruskey and Mark Weston. 1997. A survey of Venn diagrams. *Electronic Journal of Combinatorics* 4 (1997), 3.

[28] Paul H Schimpf. 2012. Modified protothreads for embedded systems. *Journal of Computing Sciences in Colleges* 28, 1 (2012), 177–184.

[29] Chenguang Shen, Rayman Preet Singh, Amar Phanishayee, Aman Kansal, and Ratul Mahajan. 2016. Beam: Ending Monolithic Applications for Connected Devices. In *Proc. of USENIX Annual Technical Conference*.

[30] Meena Singh, MA Rajan, VL Shivraj, and P Balamuralidhar. 2015. Secure mqtt for internet of things (iot). In *Proc. of IEEE Communication systems and network technologies (CSNT)*.

[31] Monsoon Solutions. 2019. Monsoon Power Monitor. https://www.msoon.com/online-store.

[32] Yu-Chih Tung, Duc Bui, and Kang G Shin. 2018. Cross-Platform Support for Rapid Development of Mobile Acoustic Sensing Applications. (2018).

[33] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. 2014. Practical trigger-action programming in the smart home. In *Proc. of ACM SIGCHI Conference on Human Factors in Computing Systems*.

[34] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L Littman. 2016. Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes. In *Proc. of ACM CHI Conference on Human Factors in Computing Systems*.

[35] Zidong Yang, Ji Hu, Yuanchao Shu, Peng Cheng, Jiming Chen, and Thomas Moscibroda. 2016. Mobility modeling and prediction in bike-sharing systems. In *Proc. of ACM MobiSys*.

[36] Qian Zhicong, Luo Delin, and Wu Shunxiang. 2008. Analysis and design of a mobile forensic software system based on AT commands. In *Proc. of IEEE International Symposium on Knowledge Acquisition and Modeling Workshop*.