# WARBLE: Programming Abstractions for Personalizing Interactions in the Internet of Things

Yosef Saputra, Jie Hua, Nathaniel Wendt, Christine Julien
*The University of Texas at Austin*
{yosef.saputra,mich94hj,nathanielwendt,c.julien}@utexas.edu

Gruia-Catalin Roman
*University of New Mexico*
gcroman@unm.edu

***Abstract*—Advances in sensing and networking along with ubiquitous Internet connectivity have paved the way for today's massive Internet of Thing (IoT) market. Despite the vast potential of connecting to myriad devices across homes, office buildings, and public spaces, there is still a large need to unify the scattered protocols, hubs, and cloud services while personalizing end-user experiences. Enabling personalized IoT experiences requires an expressive and flexible middleware that enables simplified development of applications that address diverse individual needs and seamlessly cross multiple vendors and administrative domains. We introduce Warble: a middleware for such personalized IoT applications; Warble encapsulates device and protocol complexities, represents interaction with IoT devices as flexible programming abstractions, and enables applications to learn from their prior interactions in the IoT on behalf of their users. In this paper, we present Warble's architectural abstractions, API, and implementation. We then evaluate the middleware through a case study application using our Android implementation; this evaluation showcases the novelty of the Warble architecture and its programming abstractions.**

## I. INTRODUCTION

In the Internet of Things (IoT), devices in our spaces interact with us to respond to our (explicit or implicit) needs. Each day sees new devices, bringing the long vision of the IoT closer to reality. Yet, many challenges related to enabling robust, reliable, and efficient development of IoT applications remain. This paper presents the abstractions and architecture defining the WARBLE middleware, which enables the development of novel applications in the emerging IoT. We ask a simple yet fundamental question: *what does a user expect from an IoT-enabled environment*? In basic terms, the answer is simple: *one expects natural interactions and responsive behavior*. Unfortunately, the gap between users' expectations and developers' abilities to meet those expectations is significant.

An essential need of middleware for the IoT is to enable *interoperability* among device types and manufacturers [34]. In most IoT deployments, rigid barriers prevent a user from carrying out identical activities in different settings. Even trivial endeavors like turning on a light or using entertainment equipment fail to transfer from one setting to another due to different manufacturers, administrative domains, or software versions. Our goal is not to remove these barriers, which are necessary to enable diversity in the IoT space, but rather to allow application developers to navigate successfully around the barriers, making functionality seamlessly available.

Existing middleware for the IoT almost exclusively supports a *centralized* view [34]; this view, in turn, promotes a *macro-programming* view of constructing IoT applications. To make this more concrete, consider the way existing IoT middleware approaches the construction of a smart building application. Existing middleware systems generally consider centralized applications that control a set of smart devices in concert. In this fashion, existing systems cannot provide personalized user experiences and preserve privacy at the same time. In contrast, we promote a more *personalized* and *decentralized* view of the IoT. Rather than having to create a program that is mediated by a single authority to leverage a set of available devices, our goal is to allow user-facing applications to opportunistically discover and immediately leverage individual capabilities in the surroundings. While this might ultimately lead to applications that dynamically assemble collaborations of devices, the approach is more naturally *egocentric* than the common centralized view taken by existing approaches.

To make personalization and interoperability concrete, consider a simple smart lighting application. Today's smart lighting systems provide slick demonstrations, albeit confined to a single home or building, with limitations related to high setup and device discovery costs. Ideally, a smart light switch app would consist of, literally, a light switch that a user presses to illuminate the surroundings. The app might initially discover and switch on the light closest to the user, which may or may not illuminate the user's space. If the user somehow rejects the choice of light, the application may try a different light (perhaps the next closest) until the application determines how best to illuminate the user's space. A more sophisticated app may implement "follow-me" lighting, continuously illuminating the user's position as he moves. This application can also be taken a step further: if the user crosses multiple enterprises, e.g., moving from the user's apartment, into the building's hallway, then into a friend's apartment, the application should seamlessly interoperate across these spaces, even if each one uses a different (potentially proprietary) IoT technology.

IoT middleware can be leveraged to ease the development burden. However, existing approaches force developers into an *enterprise* mindset rather than a personal one. Using existing middleware, developers *script* the behavior of an entire space. For instance, a developer might create an application that detects the presence of a user and adapts the lighting based on the user's movement or activity. This program, however,

belongs to the space rather than to the user who interacts with the space. As a result, interactions are less flexible, less opportunistic, and potentially leak users' private preferences and actions to the owner of the space. However, personalizing IoT systems requires non-trivial programming from experts to translate user intents into device actions. In contrast, we believe that the means for developers (and users) to specify interactions must be intuitive, leverage users' cognitive processes, and naturally map to user behaviors; therefore we also emphasize simplicity of programming as a key requirement.

Our WARBLE middleware addresses the challenge of simplicity of programming directly, while being attentive to the needs of interoperability and personalization. We define the WARBLE Thing Registry, which allows applications to seamlessly discover available devices even across differing vendors or administrative domains. To ensure personalization, an app running on a user's smartphone can seamlessly discover IoT devices that fulfill that user's instantaneous needs. With respect to interoperability, apps using WARBLE need not distinguish between IoT devices from different vendors or administrative domains; the WARBLE architecture makes these challenges transparent to users and their applications. WARBLE also maintains an Interaction History from which applications can use the results of prior interactions with IoT devices to influence future interactions, further personalizing the IoT. Finally, WARBLE defines Bindings, which raise the level of abstraction of programming from controlling a particular device with a particular communication mechanism to specifying the nature of a (potentially long-lived) user-level interaction with the IoT.

We next examine related work, including middleware for the IoT. We then frame an evolved view of the structure of the IoT and state assumptions that underlie WARBLE. Section III and IV describe WARBLE's conceptual model and key elements. Section V describes our initial implementation of WARBLE on Android for multiple IoT devices and Section VI evaluates this prototype. Section VII concludes the paper.

## II. MOTIVATION AND RELATED WORK

To scaffold our discussion of related work, we note that, in the IoT, there are effectively two classes of devices: (1) devices embedded in the environment, providing services as sensors and/or actuators and (2) users' personal devices (e.g., smartphones, tablets) that host personalized applications that interact with the surroundings. While there is potential overlap (e.g., a nearby user's smartphone might provide a camera), we conceptually distinguish the two; for the remainder of this paper, we refer to the former as THINGs, and the latter as user's devices that host IoT CONTROLLERs. Throughout, we focus on a single CONTROLLER, though a single device may host multiple IoT applications, each with their own CONTROLLER.

### A. Related Work

With the rise of IoT devices from a variety of manufacturers, several middleware systems that target the IoT have emerged [34]. With a similar goal of simplifying programming, web-based IoT systems use protocols like REST and UPnP to bridge IoT devices and existing web services [13], [40], [47]. These approaches, termed the Web of Things [7], [15], [55] bring resources under a single managing umbrella, enabling interoperability. As these web-based systems rely on cloud-based architectures, they naturally inherit privacy issues related to cloud systems [38], [56]. Some cloud-based solutions like OpenIoT [48] and GSN [1] require high-performance hardware and thus are too heavyweight for the highly fluid opportunistic mobile scenario we target.

A primary goal of our work is to simplify development, making applications easier to program without impacting flexibility or performance. Existing work on programming for wireless sensor networks and early efforts for the IoT have goals similar to ours [41], [45]. We aim for a higher level of abstraction that allows one to reference relationships of applications to THINGs in the IoT. We use history of a CONTROLLER's interactions with THINGs to improve subsequent interactions. Several existing approaches promote *macroprogramming*, the idea of programming a set of things in concert [5], [18], [27], [33]. Ravel [43] uses the model-view-controller paradigm to provide programming abstractions that span embedded devices, mobile devices, and backends. In contrast, our focus is not control of a networked system but on allowing applications to act on behalf of individual users to tailor the immediate surroundings to personal needs.

SmartThings [46] and IFTTT [20] offer reactive APIs to control IoT devices based on predefined rules. Hydra [10], [19] and Calvin [39] offer programming tools to help consumers create their own applications. However, these are enterprise-level approaches in which the developer scripts the overall behavior of the space *a priori* based on knowledge or assumptions about the available devices and their potential interactions with users. Other approaches expose the social aspect of device interactions [12]. WARBLE tackles complementary goals, though the social aspect is an area of future work.

An important gap WARBLE addresses is making expressive discovery of and connection to THINGs easily accessible to application developers. Recent research addresses mechanics (i.e., protocols) of discovery. Assuming a description of a resource, multiple ways to distribute that information [21], [23], [24], [53] exist, and expressive naming schemes have emerged [2], [4], [25], [52]. Named Data Networking leverages semantic matching to achieve discovery flexibility [42]. However, exposing naming schemes to users is unintuitive and distant from natural interaction with an ambient environment. Other approaches embed semantics and context in discovery [14], [16], [29]. Many of these approaches focus on proximal discovery [8], [9], [32], [53] or on selecting the appropriate device in a large-scale IoT [26]. WARBLE layers on these approaches to include (1) using a history of prior interactions to guide future interactions and (2) further abstracting interactions into Binding schemes that mediate and simplify how CONTROLLERs interact with THINGs. This is similar to work on dynamic service bindings [6], [17], [22]; WARBLE brings these ideas into the IoT.

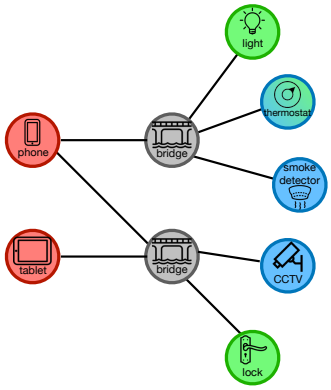


Fig. 1. State of the practice IoT. The structure is rigid and prevents interoperability.

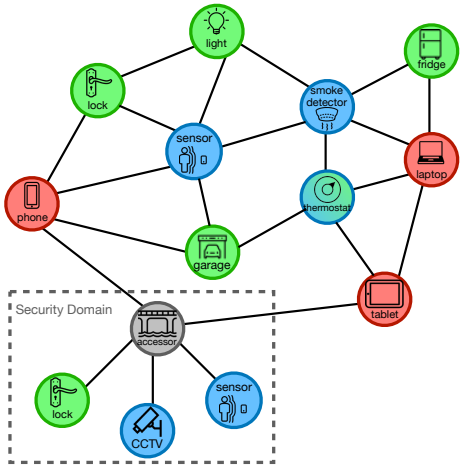


Fig. 2. WARBLE-envisioned IoT structure.

### B. The Structure of the IoT

THINGs assume two roles: SENSORs and ACTUATORs; the former collect information about the environment (e.g., a camera), while the latter change the environment (e.g., a light). A single THING can assume both of these roles, for example, a thermostat is a SENSOR, providing temperature data, and also an ACTUATOR, providing a service to regulate the A/C system.

Fig. 1 shows an IoT structure. Red circles (phone, tablet, laptop) host CONTROLLERs. Gray circles are bridges. Blue circles (smoke detector, proximity sensor, camera) are SENSORs. Green circles (light, lock, garage, fridge) are ACTUATORs. The thermostat is both SENSOR and ACTUATOR. Users interact with devices through CONTROLLERs, which connect to bridges on behalf of the applications; these bridges mediate access to THINGs in vendor-specific ways.

WARBLE takes a different structural view of the IoT, one enabled by increasing device-to-device communication capabilities and an open vision of the IoT. Fig. 2 shows this vision, in which THINGs (i.e., SENSORs and ACTUATORs) are directly accessible from one another and from CONTROLLER's host devices without applications being concerned about connections through proprietary bridges. This shift demands increased *interoperability* among CONTROLLERs and THINGs from varying vendors; in this paper, we describe how WARBLE accomplishes this interoperability. We then show how this vision impacts WARBLE's ability to *simplify* the development of applications that *personalize* the IoT for users.

While this vision enables direct interaction between CONTROLLERs and THINGs without requiring applications to directly connect to a bridge, WARBLE maintains the notion of a more abstract ACCESSOR. Conceptually, WARBLE ACCESSORs facilitate indirect connections among CONTROLLERs and THINGs. More general than a proprietary bridge that is *required* to gain access to THINGs, an ACCESSOR can act as a conduit to other THINGs to extend network reachability, provide isolation, implement security or authentication, etc. For instance, the light at the top of Fig. 2 may be accessed via the network interfaces of the neighboring light, proximity sensor, etc., rather than through a manufacturer-specific bridge.

WARBLE assumes every THING is discoverable in a device-to-device fashion. For THINGs connected to a proprietary hub (e.g., a Wink or Philips Hue hub), WARBLE assumes that the hub is discoverable in a device-to-device fashion and then supports access to devices via the hub. This is a realistic assumption given state-of-the-art IoT devices; while device-to-device discovery is not commonly consistently enabled on today's IoT devices, it is a quite common mechanism to bootstrap devices into a proprietary app, meaning that the devices have capabilities necessary for device-to-device discovery. We assume well-defined protocols for each communication technology used to connect CONTROLLERs and THINGs. That is, we assume THINGs are *open* and accessible, both to user-facing applications and to other THINGs. This does not mean that every application or user can invoke every possible function on every IoT device; instead a device's accessor may limit access, e.g., through a bridge's proprietary authentication mechanism. In our proof of concept, we use a combination of technologies to connect to THINGs, including direct BLE connections, a Wink Hub, and a Philips Hue Bridge. WARBLE realizes the vision of Fig. 2 by making proprietary bridges *transparent* to applications, making it appear that the CONTROLLER accesses THINGs directly, even though it employs a proprietary bridge under-the-hood.

Some WARBLE constructs benefit from the knowledge of THINGs' and CONTROLLERs' locations. We assume this information is provided by an external component. More generally, WARBLE relies on the ability of CONTROLLER devices to assess the state of the ambient *context*, so we assume that CONTROLLER devices have access to, for instance, on-device sensors to acquire such context information (most importantly the device's location, but also heading or degree of movement); this context could also be provided by nearby THINGs or other out-of-band sensing mechanisms [3], [24].

## III. WARBLE'S CONCEPTUAL ARCHITECTURE

We next overview WARBLE in the context of our goals of interoperability, personalization, and simplicity of programming. Fig. 3 shows WARBLE's conceptual architecture. As we step through WARBLE's components, we consider a simple smart lighting application in which a user can: (1) make a one-time request to turn on the surrounding lights or (2) make a

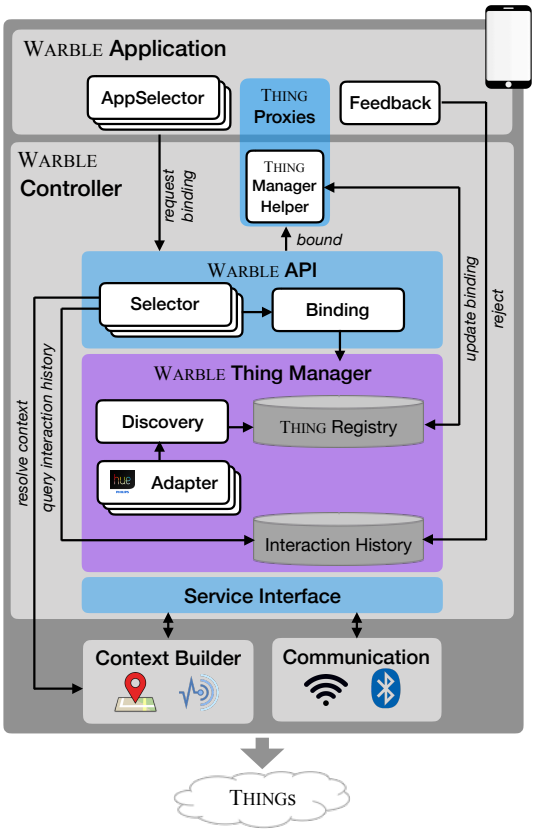


Fig. 3. WARBLE Conceptual Architecture. The application sits atop the CONTROLLER. The WARBLE API abstracts away WARBLE's key activities of discovery, selection, binding, and context acquisition. These activities rely on the Thing Manager, which encapsulates the Thing Registry and Interaction History.

persistent request to continuously illuminate his surroundings, even as he moves to new spaces. We refer to the former as a *one-time* binding and the latter as a *dynamic* binding.

The Thing Manager is the heart of WARBLE. It performs continuous Discovery of nearby THINGs, populating the Thing Registry as new THINGs are encountered. This Discovery occurs independently of application requests; the WARBLE Discovery process ensures that a CONTROLLER's Thing Registry maintains a current view of nearby available THINGs.

An application request to interact with THINGs starts with a Binding request. When the binding is ready, the middleware returns Proxies for bound THINGs. The application interacts with the Proxies, which provide canonical interfaces for SENSORs and ACTUATORs. A Proxy relies on a THING-specific Adapter, which explicitly enables interoperability by making all THINGs of a certain type appear the same (e.g., an application can be programmed to a `LightAdapter` instead of directly to, for example, some Philips Hue light API). WARBLE logs information about all user interactions with bound THINGs into WARBLE's Interaction History. If an application determines that a bound THING is not appropriate, it can provide feedback through a *reject* notice. WARBLE will log the application's feedback in the Interaction History and attempt to replace the Proxy. WARBLE uses this information to inform the process by which it satisfies subsequent application binding requests.

Applications can request two types of Bindings. In a one-time binding, selection and Proxy creation are done once at the time of the request and not reevaluated. The Proxy encapsulates a Thing Manager Helper, which holds a subgraph of the IoT network graph (Fig. 2). When an application interacts with a Proxy, the Thing Manager Helper resolves the path to the THINGs, which potentially traverses multiple ACCESSORs. In a dynamic binding, WARBLE continuously reevaluates the THINGs bound to the Proxies as the context changes (e.g., because the user moves or because the set of available THINGs changes). In a dynamic binding, the Proxy's Thing Manager Helper coordinates with WARBLE's Thing Registry to invoke *update binding* as needed. This architecture allows applications to personalize the use of the underlying THINGs to a user's requirements, in contrast to the current state of the practice in which THINGs are bound only to a proprietary vendor-specific application or API. A key element of an application's Binding request are Selectors, which constrain both the *types* of THINGs and their non-functional attributes (e.g., location).

WARBLE relies on two external components: the Context Builder, which interfaces to system elements such as location services and sensor data and Communication services such as Bluetooth and Wi-Fi. WARBLE's Selectors rely on the Context Builder to assess the state of the environment to select the "best" THINGs for a request and to capture the context of interactions in the Interaction History. WARBLE's interactions are limited only by the CONTROLLER's host's communication limits and the set of defined Adapters. This architecture allows various CONTROLLER implementations; our prototype uses Android to host the CONTROLLER, and the Android OS to provide the needed underlying services.

## IV. REALIZING WARBLE

We now describe the details of Fig. 3, starting from components most closely associated with THINGs, and working up the levels of abstraction to the WARBLE API.

### A. Adapters

Myriad technologies are used to connect to THINGs, from low-level network interfaces (like BLE and WiFi-Direct) to high-level programming interfaces (as exposed by the Wink and Phillips Hue hubs). WARBLE introduces Adapters to enable interoperability. WARBLE's Adapters transform a THING's stock interface into a canonical one. As a result, each THING exposes an interface that (a) describes the THING's capabilities and (b) provides the methods to interact with the THING. For instance, a light Adapter allows access to read and change the state of a light (e.g., on/off status, brightness, location, etc.). All light THINGs, regardless of their manufacturer, must implement the light Adapter for the lights to be used in WARBLE. All THINGs in WARBLE have a unique identifier (i.e., a UUID) and a location; THINGs can also have varying states, support multiple communication protocols, require different authentication mechanisms, etc. We assume each THING comes with its needed adaptor(s), i.e., that adaptors are written

Listing 1: Thing Registry for the IoT structure in Fig. 2

```
Thing Table
UUID  Type           Class                  ...
0001  Light          VendorB-Light
0002  SmokeDetector  VendorA-SmokeDetector
0003  Sensor         VendorC-Sensor
0004  Thermostat     VendorA-Thermostat
0005  Fridge         VendorB-Fridge
0006  Lock           VendorB-Lock
0007  GarageDoor     VendorC-GarageDoor
0008  Accessor       VendorB-Bridge
0009  Lock           VendorB-Lock
0010  Camera         VendorB-CCTV
0011  Sensor         VendorB-Sensor
...
Connection Table
ID Source  Destination Type        ...
1  0001    0002        Bluetooth
2  0003    0002        BLE
3  0004    0002        WiFi
4  0005    0002        Bluetooth
...
Authentication Table
ID Thing Type          Details              ...
1  0001  UserPassword  myusername, ******
2  0002  Token         **********
```

by manufacturers. In the future, one could envision auto-generating adaptors based on semantic descriptions or the use of some existing multi-purposes IoT service language.

### B. The Warble Thing Registry

WARBLE employs *proactive discovery*, continuously (in the background) probing the surroundings for available THINGs. WARBLE supports standardized discovery protocols such as Wi-Fi SDP, Bluetooth SDP, and SSDP, but WARBLE applications and vendors can extend the middleware with custom discovery protocols. Discovered THINGs are placed in the Thing Registry using the unifying adapter interfaces to mix THINGs accessible via different mechanisms (i.e., THINGs accessible via BLE appear in the registry alongside THINGs accessible via a Wink hub). A WARBLE instance has a discovery interval that dictates the frequency at which to rescan the surroundings; when a scan completes, the registry is considered consistent with the state of the IoT and remains static until the next scan. Application requests for available matching THINGs are immediately resolved using the contents of the Thing Registry.

Abstractly, the Thing Registry specifies a graph, with THINGs as nodes and THING-to-THING connections as edges. These edges are important because THINGs may act as ACCESSORs that extend the reach of a CONTROLLER or to provide authentication or isolation. Internally, WARBLE's Thing Manager stores the full graph in a database of THINGs, connections, and authentication credentials. This allows WARBLE to compute a partial graph at runtime that contains only THINGs relevant to a given request. The Thing Registry is a set of database tables: one to map the UUIDs of discovered THINGs to their types and implementing classes, one to specify the physical connections among THINGs, and one to specify any needed authentication mechanism(s) for the THINGs in the registry. Listing 1 shows an example built directly from the IoT network in Fig. 2.

Consider an application that needs to use a "Smoke Detector". A "Smoke Detector" THING is in the registry in Listing 1 (with a UUID of 0002 and four connections to other THINGs as depicted in Fig. 2). The application requests a THING that provides the smoke detector Adapter; WARBLE returns a Proxy of type `SmokeDetector` to the application. At runtime, that type is provided by the concrete `VendorA-SmokeDetector`.

Within the Proxy's Thing Manager Helper, WARBLE returns the entire path from the CONTROLLER to the THING, retrieved by recursively traversing the graph stored in the database tables. The application's subsequent interactions with the THING are managed through the Proxy. In the case that the application uses the THING Proxy for an extended period of time, the path may change (e.g., because the physical connectivity in the underlying THING network changes). This is handled transparently for the application in the Thing Manager Helper.

### C. The Selector

An application's interaction with WARBLE to retrieve THINGs has two steps. First, the application supplies one or more Selectors. For instance, an application requests a "nearby" light or a camera with pan/tilt/zoom capabilities. Second, WARBLE resolves Selectors using the Thing Registry and returns a Proxy to the application.

WARBLE has two categories of Selectors: *type* Selectors use the Adapter interfaces to specify functional requirements, while *context* Selectors provide non-functional requirements. A *type* Selector indicates which THING types to include (e.g., lock, camera, light). If the application does not provide a *type* Selector, all types are returned; if more than one type Selector is provided, WARBLE treats the request as an OR, i.e., any of the types are equivalently satisfying. In contrast, a *context* Selector could invoke the Context Builder to, for example, access location, orientation, or movement sensors. This allows selectors that return, for example, "the closest" THING, "the closest THING in the direction the user is facing" [28], or even, "a THING in the direction of the user's movement." A *context* Selector can also use the Interaction History to rely on the success or failure of previous similar requests. For instance, a Selector may stipulate that a selected THING be one for which the application has not sent a *reject* notice from the CONTROLLER's current location. We describe the use of the Interaction History in more detail in Section IV-E.

### D. The Binding Abstraction

The next step of an application's interaction with THINGs is to create and maintain connections to the THINGs, via a process we call Binding. Conceptually, WARBLE presents an abstract operation called `bind`. The application provides Selectors in a call to a `bind` method, and such a call returns Proxies to one or more bound THINGs, assuming a satisfying THING can be identified. This Binding abstraction shields the developer from explicitly dealing with the nuts and bolts of connections to THINGs, thereby simplifying programming IoT applications. Conceptually, the abstract `bind` is defined as:

$$\texttt{bind}(\texttt{template}, \texttt{options}, [\texttt{k}])$$

where `template` is a set of zero or more Selectors, and `options` dictates the binding logic (e.g., one-time vs. dynamic, additional actions to take on binding, etc.). Because a

binding request can select more than one THING, the optional `k` determines the maximum number of THINGs to return. In some cases, there may not exist `k` THINGs that match, so at most `k` items are returned. The default value is one. The result of a binding is an abstract data structure containing Proxies to bound THINGs. For simplicity, we treat the data structure as a list, though future work could layer more complex data structures (e.g., placing THINGs on a map of physical space).

**One-time Binding.** Requesting a one-time binding requires no binding options in `bind`. We refer to the concrete action as `fetch`, since it simply fetches Proxies for THINGs that match the `template`. Upon binding, the Thing Manager builds a subgraph containing the currently available ACCESSOR paths from the CONTROLLER making the request to each selected THING. This subgraph is given to the Proxy's Thing Manager Helper, which, when the application interacts with the THING via the Proxy, finds a path to the THING in the graph. The Thing Manager Helper also manages any necessary authentication via the ACCESSORs in the graph. If the paths to the THING bound to a one-time binding are no longer traversable when an application attempts to interact with the Proxy, WARBLE returns an exception. It is entirely in the hands of the application (and user) to ensure that the THING is placed in a consistent end state if necessary (e.g., turning a light off when the application is finished with it) before the THING becomes unreachable.

Some applications may desire to immediately execute a fixed sequence of operations on a set of selected THINGs. In WARBLE, `batch` allows the application to provide such a sequence of operations, removing the need for the application programmer to manage THING Proxies. When invoking the `batch` one-time binding, the application provides an `onBind` parameter, which is effectively a script of the actions to take upon binding to each selected THING. Instead of returning a Proxy to the application, a `batch` operation binds the selected THINGs then executes the actions directly, internally handling exceptions (such as disconnections). When the `onBind` actions have completed, the bindings effectively go out of scope.

**Dynamic Binding.** In a dynamic binding, as WARBLE's discovery mechanisms update the Thing Registry, the specific THINGs behind a Proxy may be updated. This is handled in each dynamic binding's Thing Manager Helper, which listens for and responds to updates in the Thing Registry. For instance, if the application is connected to the two *closest* light THINGs, when new lights are discovered, WARBLE notifies the Thing Manager Helper, which checks whether the new lights are closer and updates the Proxy bindings as needed. If the connectivity paths to a bound THING change, the graph embedded in the Proxy's Thing Manager Helper is similarly updated.

A dynamic binding Proxy is *read-only*. To *change* the state of a dynamic binding Proxy, applications specify a Plan that defines the desired continuous state of bound ACTUATOR THINGs. When new THINGs are bound to the proxies, the Thing Manager Helper automatically adjusts their state using the instructions encoded in the Plan without explicit application interaction. Conceptually, this Plan is provided within the binding `options` of the dynamic binding's `bind` operation; practically, the WARBLE API allows specifying the Plan separately from the binding, which also allows the Plan to be updated without tearing down a dynamic binding and building a new one. When WARBLE updates a dynamic binding, prior bound THINGs are *unbound*; upon unbinding the Thing Manager Helper returns unbound THINGs to their state before they were bound to the dynamic binding.

### *E. The Interaction History*

WARBLE's Interaction History stores information about interactions with particular THINGs in particular contexts. Each user device maintains its own Interaction History. This is an intentional design decision to personalize the use of prior interactions per user; placing this information in a shared repository may violate privacy. Further, what constitutes a successful interaction for two users may be different. Associating the Interaction History with a user is therefore motivated by our goal of personalizing the IoT. On the other hand, there are good reasons for sharing Interaction History among users who interact with the same THINGs. Sharing may lead to faster learning, especially in spaces that are new to a user or with which the user interacts only rarely. Therefore, an alternative design could push (some of) the Interaction History to THINGs themselves, leading to the emergence of an IoT infrastructure that learns about itself and its ability to support applications.

The Interaction History maintains entries for all actions performed by the application on a binding, including an entry each the time a dynamic binding's Plan changes the state of some bound thing. Each interaction history entry represents a successful or unsuccessful action taken on a THING that was bound based on some `template`. We associate a timestamp to each piece of Interaction History, and applications can tailor Selector behavior based on these timestamps.

Entries in the Interaction History are initialized during selection, but because entries require information collected as the application interacts with a bound THING, they must be updated over time. The complete process is:

1) An application initiates a `bind`, including the `template`. This creates a pending Interaction History entry (noted with the $p$ subscript) for each THING that matches the selection:

   ($id$, `template`, THING)$_p$

   $id$ is a unique identifier for the Interaction History entry.
2) The application interacts with the Proxy, either directly, in the case of a one-time binding or indirectly through the Plan, in the case of the dynamic binding. The Thing Manager Helper captures each interaction (e.g., each method called on the Proxy), copies the related pending entry, marks the copy as complete (noted by the $c$ subscript), and notes the specific `action` taken and its timestamp. The `action` includes the context of the interaction; in our current implementation, we simply log the CONTROLLER device's location. Multiple interactions create multiple completed entries in the history. WARBLE assumes that the interaction is successful until notified otherwise:

   ($id$, time, `template`, THING, `action`, *true*)$_c$

3) An application may give explicit feedback by invoking *reject* on a THING for its most recent `action`, marking the entry unsuccessful (i.e., changing *true* to *false*):

$$(id, \text{time}, \texttt{template}, \text{THING}, \texttt{action}, false)_c$$

A *reject* notice also initiates a rollback, which examines the rejected action (e.g., `camera.turn(45)`) and performs the logical undo-action (e.g., `camera.turn(-45)`). These undo actions are specified in the WARBLE THING adapter.

It is now possible to see how WARBLE can use the Interaction History to influence selection. For example, when invoked given a reference location and time range ($[t_{start}, t_{end}]$), a WARBLE Selector can examine each THING in the registry that satisfies the applications' `template`. For each matching THING, the selection retrieves relevant Interaction History tuples (i.e., those about the selected THING whose `timestamp`s are within the range) and examines whether interacting with the THING is expected to be successful at the given location.

## V. IMPLEMENTATION: PROGRAMMING WITH WARBLE

We have implemented WARBLE on Android (in Java); its API is in Table I.[1]. We next walk through several use cases relating to smart lights with the specific aim of demonstrating how WARBLE achieves the design goals of interoperability and personalization. These examples also demonstrate how developers use WARBLE to simplify the programming task; the next section quantifies these details. We use lighting examples because they are straightforward and accessible; additional examples with other types of SENSORs and ACTUATORs are available. All told, WARBLE currently supports 11 types of THINGs. We have integrated four concrete THINGs from three vendors with four different discovery mechanisms.

**Personalization in WARBLE.** Unlike a centralized view of the IoT in which one program controlled by the space determines THINGs' behavior, WARBLE provides abstractions to allow individual applications to use THINGs directly. A WARBLE application interacts with the Thing Registry to determine *at runtime*, what the best THINGs are for a given need. In contrast, existing IoT middleware create program scripts *at compile time* that choreograph the behavior of the space's THINGs. This is a subtle shift; WARBLE is appropriate for application interactions that are highly adaptive and context dependent. In contrast, existing approaches like Node-RED [35] and Calvin [39] are more suitable for these predictable and scriptable interactions. WARBLE is more similar to middleware that take a service-oriented approach [48]. These approaches provide web-programming based interfaces to THINGs in which applications construct and invoke web requests. In contrast, WARBLE's Binding and feedback concepts provide a higher-level of programming abstraction more tailored to interacting with THINGs in an object-oriented way. Further, the service-oriented IoT middleware are designed to run on heavyweight cloud infrastructure and are not compatible with an approach that executes entirely on lightweight edge devices [34].

[1]WARBLE and examples available at https://github.com/UT-MPC/Warble3

Listing 2: Application-defined selector based on line of sight

```
1  public class LOSSelector extends AbstractSelector {
2    public LOSSelector(Location location, double heading,
3                       double angle, double range) {
4      // ... save instance variables
5    }
6    @Override
7    public List<Thing> select() {
8      CircleSector sector = // ... compute sector
9      List<Thing> reg = Warble.getInstance().getThings();
10     List<Thing> selectedThings = new ArrayList<>();
11     for(Thing thing : reg)
12       if (sector.contains(thing.getLocation())
13         selectedThings.add(thing);
14     return selectedThings;
15   }
16 }
```

**Interoperability in WARBLE.** Once THINGs are available within the WARBLE Thing Registry, all WARBLE applications interact with all things using the same set of interfaces (as shown in Listing 1). This approach is very similar to the many other existing efforts at semantic mapping of THINGs to software interfaces [10], [49], [51], [54]. Our current implementation of the Service Interface at the bottom of Fig. 3 is, therefore, a straightforward mapping of vendor classes to WARBLE types. However, it is a straightforward extension of the Service Interface to also enable mapping from other semantic descriptions (e.g., SensorML [44] or OWL [37]).

**Concrete Things and Selectors.** Listing 1 shows several examples of adapters, which are provided to WARBLE's `TypeSelector` when creating a `template`. WARBLE also currently has three context selectors. The `NearestThingSelector` takes a `location` and selects the THINGs closest to the `location`. The `RangeSelector` requires returned THINGs to be within the specified `range` of the provided `location`. The `InteractionHistorySelector` returns THINGs for which the Interaction History has not logged a negative interaction at the given `location`. Below, we also develop a line of sight selector as an example of how applications can add new Selectors to WARBLE.

**Adding New Things to Warble.** The `Thing` base class assumes every THING has a UUID, location, a set of discovery mechanisms, and a set of (direct) connections to other THINGs. WARBLE employs the Command design pattern [11], so every THING must also implement a `callCommand` method that handles requests to change the state of the THING (for ACTUATORs) or to retrieve the state of the THING (for SENSORs). Integrating a new THING simply requires selecting the appropriate Adapter (e.g., `Light` or `SmokeDetector`) and overriding the `callCommand` method. This override is simplified by `State` definitions for each of Adapter type (e.g., `LightState`, etc.) and a `setState` method in the `Thing` base class.

**Adding new Selectors.** WARBLE's Selectors can be extended to implement application-specific selection. An example is a line of sight Selector, whose code is in Listing 2. The author of the `LOSSelector` (1) extends WARBLE's `AbstractSelector`; (2) defines a constructor, which takes parameters required to scope the selector; and (3) overrides the `AbstractSelector`'s `select` method, which encodes the selector logic, in this case selecting THINGs within the

TABLE I: Warble API methods summary

| Method | Description |
|---|---|
| *Instantiation* | |
| Warble() | creates a WARBLE instance to access its API; initiates continuous discovery in the background |
| *Discovery* | |
| warble.discover() | performs a full discovery process manually |
| *One-Time Binding* | |
| warble.fetch(template, [k]) | fetches k THINGs matching requirements in template; provides the abstract `bind` for a one-time binding |
| warble.batch(template, onbind, [k]) | selects k THINGs matching requirements in template and executes the actions listed in onBind immediately |
| warble.reject(THINGs) | indicates a mismatch in binding for each of the provided THINGs |
| *Dynamic Binding* | |
| warble.dynamicBind(template) | creates a dynamic binding instance for THINGs matching the requirement in template |
| dBind.fetch([k]) | fetches k THINGs matching requirement in template for dynamic binding |
| dBind.bind(plan) | starts the dynamic binding to serve the configuration in plan |
| dBind.unbind() | stops the dynamic binding |
| dBind.reject([THINGs]) | indicates a mismatch in dynamic binding and triggers another binding process |
| *Plan* | |
| Plan() | creates a Plan instance |
| plan.set(preferenceKey, value) | sets value of key idenfitied by preferenceKey (e.g. `Plan.Key.LIGHTING_ON`) |
| plan.unset(preferenceKey) | unsets a preferenceKey |
| plan.unsetAll() | clears all preferenceKey settings |

Listing 3: Application code for a WARBLE one-time binding

```
1  Warble warble = new Warble(); //initiates discovery

2  List<Selector> template = new ArrayList<Selector>();
3  template.add(new TypeSelector(THING_CONCRETE_TYPE.LIGHT,
4                       THING_CONCRETE_TYPE.THERMOSTAT));
5  template.add(new NearestThingSelector(myLoc));
6  template.add(new InteractionHistorySelector(myLoc));

7  List<Thing> things = warble.fetch(template, 3);
       //things contains the Proxies

8  for (Thing thing : things) {
9    if (thing instance of Light)
10     ((Light) thing).on(); //simplified, Command Pattern
11   else if (thing instance of Thermostat)
12     ((Thermostat) thing).setTemperature(298); //in Kelvin
13 }
```

sector of a circle centered at `location`, with a radius of `range`, given the `heading` and `angle`. An application can use this Selector when creating a `template`; the selection process combines it with the rest of the Selectors in the `template` to determine which THINGs to engage in the binding.

**Programming with Warble.** Binding is driven by application-supplied `templates` that state the requirements of selection. Listing 3 shows an application initiating a one-time binding to select the three closest `Light` or `Thermostat` THINGs for which the Interaction History has no record of *reject* actions at the location `myLoc`. After constructing the `template`, the application invokes `fetch`, which returns a `List` of Proxies. The application interacts with these Proxies using the `Light` and `Thermostat` Adapter interfaces. Listing 4 shows a (partially elided) native implementation of the same functionality, albeit without the Interaction History. The application itself must directly call discovery and directly implement the detailed selector logic. This approach is unwieldy for the developer, error-prone, not future-proof, and does not provide generic forms for interacting with THINGs.

Listing 5 shows a WARBLE dynamic binding using the same `template`. The application's `plan` sets a `Light` to on. This `plan` is executed any time the Proxy is bound to a `Light`. The `plan` also sets the ambient temperature; this is executed any time the Proxy is bound to a `Thermostat`. The use of

Listing 4: (Partial) Native implementation of one-time binding

```
1  public void useThings() {
2    int[] types = {THING_CONCRETE_TYPE.LIGHT,
3                   THING_CONCRETE_TYPE.THERMOSTAT};
4    // ... other variables, e.g., location, threshold, etc.

5    // rely on underlying discovery mechanism
6    List<Thing> discoveredThings = Thing.discover();
7    List<Thing> selectedThings =
         selectType(discoveredThings, types);
8    selectedThings = selectLocation(selectedThings);

9    for (Thing thing : selectedThings)
10     // ... same as lines 9-13 in Listing 3

15 }

16 List<Thing> selectType(List<Thing> things, int[] types){
17   List<Thing> selectedThings = new ArrayList<>();
18   for (Thing thing : things)
19     if (types.contains(thing.getThingConcreteType()))
20       selectedThings.add(thing);
21   return selectedThings;
22 }

23 // return things within a threshold distance
24 List<Thing> selectLocation(List<Thing> things){/*...*/}
```

the `plan` for specific THING types is implicit; a `Thermostat` simply does not understand the command `LIGHTING_ON`, so it is ignored. Listing 5 also shows the application updating the `plan`, which updates the state of bound THINGs and changes the binding behavior for future bound THINGs.

WARBLE's programming simplification is even starker for the dynamic binding. Listing 6 shows natively implemented code that is similar in functionality to the WARBLE dynamic binding in Listing 5. As shown the native application must implement a form of runnable thread that periodically invokes discovery of THINGs and manually enacts the application's desired behavior (i.e, the WARBLE Plan) on the discovered THINGs. Omitted from this listing is all the necessary exception handling code and how the application handles THINGs it was using that have gone out of scope. All of these behaviors are handled automatically in WARBLE; none of the dynamic binding code for the WARBLE interaction is elided in Listing 5.

**Interaction History.** Conceptually, we view the Interaction History as a single unit as shown in Fig. 3. The implementation,

Listing 5: Application code for a WARBLE dynamic binding

```
Warble warble = new Warble(); //initiates discovery

List<Selector> template = new ArrayList<>();
template.add(new ThingConcreteTypeSelector(
     THING_CONCRETE_TYPE.LIGHT,
     THING_CONCRETE_TYPE.THERMOSTAT));
template.add(new NearestThingSelector(myLoc));
template.add(new InteractionHistorySelector(myLoc));

Plan plan = new Plan();
plan.set(Plan.Key.LIGHTING_ON, true);
plan.set(Plan.Key.AMBIENT_TEMPERATURE, 298);

DBinding dBind = warble.dynamicBind(template, 3);
dBind.bind(plan); // start binding based on plan
// ...
plan.set(Plan.LIGHTING_COLOR, "RGB#EEEEEE");
dBind.bind(plan); // bind again with changed light color
```

Listing 6: (Partial) native implementation of dynamic binding

```
public class ContinuousThread implements Runnable {
  public ContinuousThread(/* ... */){/*initialization*/}
  public void run() {
    List<Thing> discoveredThings = Thing.discover();
    List<Thing> selectedThings = //...Listing 4 Lines 8-9

    // manually enact the plan
    for (Thing thing : selectedThings)
      // .. same as lines 9-13 in Listing 3

    // continuously recheck the surrounding things
    while (!stopFlag) {
      discoveredThings = Thing.discover();
      newSelectedThings = // select again by type/location
      if (!newSelectedThings.equals(selectedThings)){
        // manually enact the plan on any new things
        // put old things in their original states
      }
      selectedThings = newSelectedThings;
      // exception handling omitted
      Thread.sleep(rediscoveryPeriod);
    }
  }
}
```

however, contains two distinct components: the *local* and *global* histories. Each CONTROLLER maintains its own local history in main memory. The local history functions like a cache and 1) provides fast access to entries that will be duplicated (e.g., pending entries copied when an application interacts with a THING) or updated (e.g., completed entries updated upon a ***reject***) and 2) maintains `action` objects that contain code for rolling back an `action` in case of a ***reject***.

In contrast, the global history resides on-disk and is accessible to and maintains histories for all WARBLE CONTROLLERs on the user device. Entries that are unlikely to be updated or duplicated are flushed from the local history to the global history. Code (such as a rollback operation) is not stored in the global history; rollback should not be required on entries in the global history. When to flush entries is an open research question; for now, WARBLE flushes entries prior to new selection operations because selection may rely on the availability of the information in the global Interaction History.

## VI. EVALUATION: BENCHMARKING WARBLE

The previous section gave exemplars of programming with WARBLE to demonstrate how WARBLE provides both interoperability and personalization. Through these examples, we can also start to see how WARBLE simplifies the programming task. In this section, we move from these qualitative judgments toward quantitative ones. We also benchmark the trade-offs associated with these programming simplifications.

### A. Measuring Ease of Programming

To measure ease of programming, we created an application and implemented diverse Bindings. We use MetricsReloaded [31] to benchmark the implementation. Our experiments use a combination of devices, including a Philips Hue Bridge, Philips Hue Lights, a Wink hub, and GE lights. To control the experiment, we hardcoded locations. All implementations use the same Android communication services.

To characterize the overhead of employing WARBLE's abstractions, we also benchmark the latency and energy costs of interacting with THINGs. We use Android logging to measure latency and the Trepn Profiler [50] to measure energy consumption using a sampling rate of 10Hz. All of our measurements are done using a Moto X (2nd Gen.) with Android 5.1 Lollipop, which covers 85% of Android devices.

We first compare a native implementation of a one-time binding with WARBLE's implementation. In the native implementation, the application finds the closest light by manually querying the Philips Hue Bridge for the available lights, computing their distances to the user's CONTROLLER location, selecting the closest ones, and performing an HTTP request to the Philips Hue Bridge. In the case of the thermostat, we assume each thermostat exposes its own interface that the CONTROLLER can interact with. This interface is not limited to HTTP requests but extensible to any communication protocol supported by both systems. Each implementation executes a query to the thermostats and selects the one discovered that is closest to the CONTROLLER's location. Subsequently, the application sends commands to the lights and thermostats according to the desired ambient conditions using the appropriate interface given the particular `type` of each THING. In contrast, the application implemented with WARBLE one-time Binding uses exactly the code in Listing 3 to achieve the same goal.

Table II gives the quantitative comparisons between the two versions. We use three programmability metrics: (1) the *maximum number of indentations*, since a high level of nesting reduces code readability [36]; (2) *cyclomatic complexity* [30], which counts paths through the code; (3) and *lines of code*, which is a crude measure of programmer effort. The WARBLE implementation of the one-time binding is substantially better on these metrics than the native implementation. For instance, the native approach requires more than 2x the number of lines of code than that by WARBLE; this difference is even greater if we include THING discovery, especially in the common case that multiple Adapters are used. Additionally, WARBLE's abstractions are future-proof towards new types of THINGs, context, and selection strategies, whereas the native implementation only considers hard-coded lights and thermostats.

**WARBLE's Dynamic Binding.** We use a similar scenario to evaluate the degree to which the dynamic binding simplifies the implementation of continuous interactions with THINGs. In supporting this continuous behavior, the application must make

TABLE II: WARBLE's code metrics; WARBLE's improvements are dramatic for all three metrics

| Metric | One-time | | Dynamic | |
|---|---|---|---|---|
| | Native | WARBLE | Native | WARBLE |
| Cyclomatic Complexity | 3 | 1 | 25 | 1 |
| Max # Indentations | 4 | 2 | 5 | 0 |
| Line of Codes | 34 | 13 | 101 | 7 |

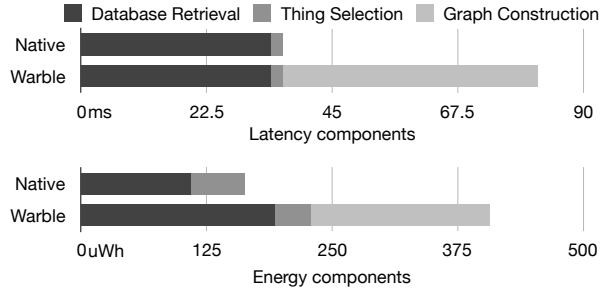


Fig. 4. Latency and energy performance components of WARBLE vs. a native implementation for retrieving THINGs.

ongoing adjustments in response to a changing set of available THINGs. For instance, as the user's location changes, the user may continuously require a different set of closest lights and thermostats to maintain the desired ambient conditions. The native application must manually adjust bound THINGs and control different sets of THINGs based on the changing context. In contrast, WARBLE's dynamic binding handles this seamlessly on behalf of the application. The application implemented with WARBLE uses the code in Listing 5; Listing 6 shows the analogous native implementation.

Table II includes programming metrics for continuous interaction. WARBLE's dynamic binding significantly reduces the programming effort. The level of complexity decreases dramatically because WARBLE's abstractions allow developers to easily and directly capture the user's intent in the binding's template and Plan. WARBLE then evaluates and adjusts its fulfillment of the user's needs in the background. This is a typical use case of personalized IoT applications.

### *B. The Overhead of* WARBLE

Simplifying programming using a middleware like WARBLE comes at a cost in terms of energy and latency. Fig. 4 shows the components of the latency and energy usage for both approaches. Each reported energy and latency measurement is an average of 30 repetitions.[2] The magnitudes of the energy costs are dependent on the particular device settings; what is relevant, however, is the relative difference between the two implementations. There are three major contributors to each measurement: retrieving the available THINGs from a database (i.e., registry), constructing the IoT graph (only in the WARBLE case), and THING selection. WARBLE's added latency is entirely in constructing the IoT graph. This graph is used by the Thing Manager Helper to carry out the applications' interactions, so the cost is amortized over the Proxy's lifetime. In the native case, in contrast, an application would incur more exceptions in interacting with THINGs behind Proxies, having to handle exceptions manually in the application code. The energy difference is also sizable; in addition to the cost of graph construction, WARBLE also maintains more detail in the Thing Registry that must be navigated to resolve requests. However, the magnitudes of these latency and energy values are well with reason for today's mobile devices.

[2]The errors for the total energy are $\pm 4.5 \mu Wh$ for the native case and $\pm 19 \mu Wh$ for the Warble case, both of which are within $\pm 5\%$.

## VII. CONCLUSIONS AND FUTURE WORK

WARBLE provides a middleware architecture to unify the plethora of IoT technologies and to allow application developers to create personalized instantiations of user's interactions with the IoT. In this paper, we have demonstrated WARBLE's success in addressing three fundamental challenges in the long vision of the IoT: interoperability, personalization, and simplicity of programming. Several open challenges remain.

Clearly, security and privacy are essential. In the current form, WARBLE tackles security and privacy issues only through the ACCESSOR concept, which allows an IoT device or vendor to achieve isolation, authentication, or authorization. The vision of WARBLE, however, raises new privacy issues, in particular, the fact that a user's inherent interactions with things has the potential to leave a digital footprint. WARBLE prevents this, for now, by maintaining all Interaction History locally; the reach of WARBLE could be extended if novel IoT privacy protections can be developed. As introduced in the paper, there exist very good reasons for users to share their histories of interaction, most notably to provide more robust interactions with novel THINGs. With proper privacy controls in place, we can expect that CONTROLLERs and THINGs can work together with their own knowledge and voice to create a highly-functional collective expression of the environment.

This paper's presentation of WARBLE assumes a single application's interactions with the THINGs in an IoT space. Many spaces are often occupied by multiple users and those users may have conflicting views of interacting with the available THINGs. While enterprise-based solutions mediate these conflicts at the enterprise level; a personalized solution like WARBLE requires a distributed mediation of these conflicts. This is an important area of future work.

At a finer scale, enhancements to WARBLE could make it more energy and latency efficient. More streamlined models of Interaction History and more sophisticated models for representing and navigating the Thing Registry are just two such opportunities. These enhancements could even affect the user experience, for instance resulting in a Proxy representation that displays bound THINGs on a map of the user's physical space.

The creation of these opportunities speaks to the utility of the existing WARBLE middleware. WARBLE is a first in the IoT space: a middleware that truly embraces and enables the vision of open, interoperable, and personal IoT experiences.

## REFERENCES


[1] K. Aberer, M. Hauswirth, and A. Salehi. A middleware for fast and flexible sensor network deployment. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1199–1202. VLDB Endowment, 2006.

[2] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. of the $17^{th}$ ACM Symp. on Operating Systems Principles*, pages 186–201, 1999.

[3] J. Adkins et al. The signpost platform for city-scale sensing. In *Proc. of ACM/IEEE IPSN*, 2018.

[4] M. Amadeo, C. Campolo, A. Iera, and A. Molinaro. Named data networking for IoT: An architectural perspective. In *Proc. of the European Conf. on Networks and Communication*, pages 1–5, 2014.

[5] A. Azzar, D. Alessandrelli, S. Bocchino, M. Petracca, and P. Paganp. PyoT: A macroprogramming framework for the Internet of Things. In *Proc. of the $9^{th}$ Int'l. Symp. on Industrial Embedded Systems*, pages 96–103, June 2014.

[6] C. Borcea, O. Riva, T. Nadeem, and L. Iftode. Context-aware migratory services in ad hoc networks. *IEEE Transactions on Mobile Computing*, 6:1313–1328, 06 2007.

[7] A. Botta, W. de Donato, V. Persico, and A. Pescape. Integration of cloud computing and the Internet of Things: A survey. *Future Generation Computing Systems*, 56:684–700, March 2016.

[8] K. Choi and Z. Han. Device-to-device discovery for proximity-based service in LTE-advanced systems. *IEEE Journal on Selected Areas in Communications*, 33(1):55–66, 2015.

[9] M. Corson, R. Laroia, J. Li, et al. FlashLinQ: Enabling a mobile proximal internet. *IEEE Wireless Communications*, 20(5):110–117, 2013.

[10] M. Eisenhauer, P. Rosengren, and P. Antolin. A development platform for integrating wireless devices and sensors into ambient intelligence systems. In *Sensor, Mesh and Ad Hoc Communications and Networks Workshops, 2009. SECON Workshops' 09. 6th Annual IEEE Communications Society Conference on*, pages 1–3. IEEE, 2009.

[11] J. V. R. H. Erich Gamma, Ralph Johnson. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] I. Farris, R. Girau, M. Nitti, L. Atzori, R. Bruschi, A. Iera, and G. Morabito. Taking the SIoT down from the cloud: Integrating the social Internet of Things in the INPUT architecture. In *Proc. of the $2^{nd}$ World Forum on the Internet of Things*, pages 35–39, 2015.

[13] H. G. C. Ferreira, E. D. Canedo, and R. T. de Sousa. Iot architecture to enable intercommunication through rest api and upnp using ip, zigbee and arduino. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*, pages 53–60. IEEE, 2013.

[14] T. Gu, H. Pung, and D. Zhang. A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28(1):1–18, 2005.

[15] D. Guinard and V. Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain*, volume 15, 2009.

[16] B. Guo, D. Zhang, Z. Wang, Z. Yu, and X. Zhou. Opportunistic IoT: Exploring the harmonious interaction between human and the Internet of things. *Journal of Network and Computer Applications*, 36(6):1531–1539, 2013.

[17] R. Handorean, R. Sen, G. Hackmann, and G.-C. Roman. Context aware session management for services in ad hoc networks. In *Proc. of the IEEE Int'l. COnf. on Services Computing*, 2005.

[18] M. Hossain, A. Alim Al Islam, M. Kulkarni, and V. Raghunathan. $\mu$SETL: A set based programming abstraction for wireless sensor networks. In *Proc. of IPSN*, pages 354–365, April 2011.

[19] Hydra. http://hydramiddleware.eu.

[20] https://ifttt.com/.

[21] S. Jenson. The physical web. In *Proc. of CHI'14*, pages 15–16, 2014.

[22] C. Julien and D. Stovall. Enabling ubiquitous coordination using application sessions. In *Proc. of COORDINATION*, 2006.

[23] X. Lin, J. Andrews, A. Ghosh, and R. Ratasuk. An overview of 3GPP device-to-device proximity services. *IEEE Communications Magazine*, 52(4):40–48, 2014.

[24] C. Liu, C. Julien, and A. Murphy. PINCH: Self-organized context neighborhoods for smart environments. In *Proc. of the $12^{th}$ Int'l. COnf. on Self-Adaptive and Self-Organizing Systems*, 2018.

[25] C. Liu, B. Yang, and T. Liu. Efficient naming, addressing and profile services in Internet-of-Things sensory environments. *Ad Hoc Networks*, 18:85–101, 2014.

[26] W. Lunardi, E. de Matos, R. Tiburski, L. Amaral, S. Marczak, and F. Hessel. Context-based search engine for industrial iot: Discovery, search, selection, and usage of devices. In *Proc. of Emerging Technologies & Factory Automation (ETFA)*, pages 1–8, 2015.

[27] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. In *Proc. of the $13^{th}$ ACM SIGPLAN Int'l. Conf. on Functional Programming*, pages 335–346, September 2008.

[28] C. Martella, A. Miraglia, M. Cattani, and M. van Steen. Leveraging proximity sensing to mine the behavior of museum visitors. In *Proc. of PerCom*, 2016.

[29] S. Mayer, N. Inhelder, R. Verborgh, R. Van de Walle, and F. Mattern. Configuration of smart environments made simple: Combining visual modeling with semantic metadata and reasoning. In *Proc. of the Int'l. Conf. on the Internet of Things*, 2014.

[30] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[31] https://plugins.jetbrains.com/plugin/93.

[32] J. Michel, C. Julien, and J. Payton. Gander: Mobile, pervasive search of the here and now in the here and now. *IEEE Internet of Things Journal*, 1(5):483–496, October 2014.

[33] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proc. of the $1^{st}$ Int'l. Workshop on Data Management of Sensor Networks*, pages 78–87, 2004.

[34] A. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and Q. Z. Sheng. Iot middleware: A survey on issues and enabling technologies. *IEEE Internet of Things Journal*, 4(1):1–20, 2017.

[35] Node-RED: Flow-based programming for the Internet of Things. https://nodered.org/.

[36] P. Oman and C. Cook. A paradigm for programming style research. *ACM Sigplan Notices*, 23(12):69–78, 1988.

[37] Web Ontology Language (OWL). https://www.w3.org/OWL/.

[38] S. Pearson and A. Benameur. Privacy, security and trust issues arising from cloud computing. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 693–702. IEEE, 2010.

[39] P. Persson and O. Angelsmark. Calvin–merging cloud and iot. *Procedia Computer Science*, 52:210–217, 2015.

[40] D. Pfisterer, K. Romer, D. Bimschas, O. Kleine, R. Mietz, C. Truong, H. Hasemann, A. Kröller, M. Pagel, M. Hauswirth, et al. Spitfire: toward a semantic web of things. *IEEE Communications Magazine*, 49(11):40–48, 2011.

[41] J.-F. Qiu, D. Li, H.-L. Shi, C.-D. Hou, and L. Cui. EasiSMP: A resource-oriented programming framework supporting runtime propagation of RESTful resources. *Journal of Computer Science and Technology*, 29(2):194–204, March 2014.

[42] J. Quevedo, M. Antunes, D. Corujo, D. Gomes, and R. Aguiar. On the application of contextual iot service discovery in information centric networks. *Computer Communications*, 2016.

[43] L. Riliskis, J. Hong, and P. Levis. Ravel: Programming iot applications as distributed models, views, and controllers. In *Proc. of the Int'l. Workshop on Internet of Things towards Applications*, pages 1–6, 2015.

[44] sensor model language (sensorml).

[45] A. Sivieri, L. Mottola, and G. Cugola. Drop the phone and talk to the physical world: Programming the internet of things with erlang. In *Proc. of the $3^{rd}$ Int'l. Workshop on Software Engineering for Sensor Network Applications*, pages 8–14, June 2012.

[46] http://developer.smartthings.com.

[47] J. Soldatos, N. Kefalakis, M. Hauswirth, M. Serrano, J.-P. Calbimonte, M. Riahi, K. Aberer, P. P. Jayaraman, A. Zaslavsky, I. P. Žarko, et al. Openiot: Open source internet-of-things in the cloud. In *Interoperability and open-source solutions for the internet of things*, pages 13–25. Springer, 2015.

[48] J. Soldatos, N. Kefalakis, M. Hauswirth, M. Serrano, J.-P. Calbimonte, M. Riahi, K. Aberer, P. P. Jayaraman, A. Zaslavsky, I. P. Žarko, et al. Openiot: Open source internet-of-things in the cloud. In *Interoperability and open-source solutions for the internet of things*, pages 13–25. Springer, 2015.

[49] C. Tan, B. Sheng, H. Wang, and Q. Li. Microsearch: A search engine for embedded devices used in pervasive computing. *ACM Transactions on Embedded Computing Systems*, 9(4), April 2010.

[50] https://developer.qualcomm.com/software/trepn-power-profiler.

[51] H. Wang, C. C. Tan, and Q. Li. Snoogle: A search engine for pervasive environments. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1188–1202, August 2010.

[52] P. Wehner, C. Piberger, and D. Göhringer. Using JSON to manage communication between service in the Internet of Things. In *Proc. of the* $9^{th}$ *Int'l. Symp. on Reconfigurable and Communication-Centric Systems-on-Chip*, pages 1–4, 2014.

[53] X. Wu, S. Tavildar, S. Shakkottai, T. Richardson, J. Li, R. Laroia, and A. Jovicic. FlashLinQ: A synchronous distributed scheduler for peer-to-peer ad hoc networks. In *Proc. of Allerton*, 2010.

[54] K.-K. Yap, V. Srinivasan, and M. Motani. Max: Wide area human-centric search of the physical world. *ACM Transactions on Sensor Networks*, 4(4), September 2008.

[55] D. Zeng, S. Guo, and Z. Cheng. The web of things: A survey. *JCM*, 6(6):424–438, 2011.

[56] J. Zhou, Z. Cao, X. Dong, and A. V. Vasilakos. Security and privacy for cloud-based iot: challenges. *IEEE Communications Magazine*, 55(1):26–33, 2017.