ThingTalk: A Distributed Language for a Social Internet of Things

Giovanni Campagna Jiwon Seo Michael Fischer Monica S. Lam
Computer Science Department
Stanford University
Stanford, CA 94305
{gcampagn, jiwon, mfischer, lam}@cs.stanford.edu

Abstract

The Internet of Things has increased the number of connected devices by an order of magnitude. This increase comes with a corresponding increase in complexity, which has led to centralized architectures that do not respect the privacy of the users and incur high development and deployment costs. We propose the Open Thing Platform as a new distributed architecture which is built around ThingTalk, a new high level declarative programming language that can express the interaction between multiple physical devices and online services. The architecture abstracts away the details of the communication without sacrificing privacy. We experiment with ThingTalk in multiple domains and we show that several applications can be written in a few lines of code and with no need for any server infrastructure.

Categories and Subject Descriptors D.3.2 [*Concurrent, Distributed, and Parallel Languages*]

1. Introduction

We are at the advent of a new era where billions of IoT devices will be able to collect every bit of our daily life. Examples include phones tracking our GPS locations, fitness devices tracking our every step, remotely query our cars to find out where we are, thermostats that indicate whether we are home or not, smoke alarms that detect our motion in our bedrooms, and security cameras that watch the outside and inside of our rooms. Also becoming widely available are medical devices, from scales, electrocardiogram machines, apnea monitors, etc. Medical devices not only enable individuals to monitor their health continuously, they can enable lower cost and better care as doctors can tend to the patients in need of medical attention with improved monitoring of a large number of patients.

Today's trend is for all data to be stored in the cloud. For example, Facebook and Snapchat claim ownership right to our data. Besides losing privacy, we do not have easy access to our data which are now silo'ed in different cloud services. Having all our data at our fingertips, and having tools that can analyze, combine, and react to all the information becomes even more important as more data of ourselves become available. For example, Mint is a web service that allows users to manage all their finances involving different institutions in one place; however, users must entrust their credentials and financial information the Mint.

IFTTT (IF This Then That) [If-This-Then-That] is a centralized service that accepts users credentials and makes simple connections between web services and a person's IoT devices. One of the most popular functions on IFTTT is to update one's Twitter profile if the Facebook profile is changed. This app requires that we give our Facebook and Twitter password to IFTTT. At this rate, will we eventually give the credentials of all our resources to some central service?

This paper presents Open Thing Platform (OTP), a distributed system that lets users control their IoT and web resources without giving their credentials to a third party. Not only is OTP distributed and privacy preserving, it supersedes IFTTT by allowing computation on data produced by different devices, and enabling sharing between friends, and between friends' resources and their IoT devices. Many useful functions can be written in just a few lines: from updating and monitoring one's web accounts, home automation, personalizing social media, to multi-party interactions such as personal information exchange, collaboration, and gamification.

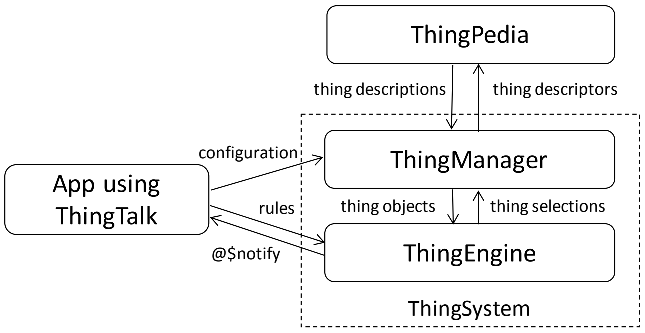

Figure 1: The Open Thing Platform architecture

1.1 An Open Thing Platform

Our design builds upon group messaging, a concept that has become ubiquitous on mobile devices. For example, the WhatsApp chat app has over 900 million users, and WeChat has over 600 million users. Common to these apps is the concept of a *feed*, a group communication channel: any messages sent to the feed are distributed to all the users in the same feed. Users can drop in any information to share with their groups of friends. Musubi [Dodson et al. 2012] is a research chat app that provides end-to-end encryption, Omlet [Omlet Chat] is a commercial open messaging platform based on Musubi that lets users save data in a repository of their choice. In addition, it has open APIs for the creation of social apps: users can connect to their friends in an open social graph, form a feed, and send/receive data from a feed.

Leveraging a privacy-preserving messaging platform like Omlet, our proposed *Open Thing Platform* allows collaborative programs be written in just a few lines, all without losing any private data to a third party. We use the term *thing* to refer to any web resource, mobile devices, or the internet of things. Shown in Figure 1 is the architecture of our Open Thing Platform. The interface of each thing is stored in a crowd-sourced repository called the *ThingPedia*. Just like a file system that manages our data, we propose a *Thing System* that manages our *things*. We expect that each user runs a Thing System, as a proxy on his behalf, possibly on a home server, in a cloud of his choice, or on his own phone. The Thing System has two services: a *Thing Manager* that manages our things and our credentials, and a *Thing Engine* that runs apps on the users' behalf.

Apps are written in a high-level distributed language called *ThingTalk*, which combines information from our things and our friends' things to generate meaningful results. These ThingTalk applications can also be deposited in ThingPedia, thus making them accessible publicly.

1.2 Example

Let us illustrate how the Open Thing Platform works by way of a simple ThingTalk app. Shown in Figure 2 is GlympseApp, an app that allows users sharing the same feed F to continuously update each others' locations on the map program on their personal devices, all without losing personal data to a third party.

```
GlympseApp$_F$() {
  @gps($l$, $t$)
      ⇒Location$_F$[SELF]($l$);
  Location$_F$[$m$]($l$), $m \in F$
      ⇒@(type="Map").plot($m$, $l$);
}
```

Figure 2: The GlympseApp location sharing app

GlympseApp, like any ThingTalk app, consists of a set of rules to be run on the user's Thing Engine. We will paraphrase each of these rules informally, the details on the language will be discussed in Section 4. The first rule specifies that it uses a GPS in the user's collection of things to periodically returns a user's location, which is stored as a value to the keyword Location visible to everybody in the feed. The second rule states that any new value assigned to the Location keyword by any members of the field, results in calling the map interface that plots a person's location.

When a user runs GlympseApp on a feed with a group of friends, the app interacts with the user's designated Thing Manager to retrieve, and configure if necessary, the user's thing. In this case, GlympseApp asks the Thing Manager for an ID for a GPS device available to the user. The Thing Manager consults with the open-source ThingPedia for interfaces if it encounters new things. The rules are then installed the Thing Engine. Thing Engine retrieves the GPS ID from the ThingManager.

This app shows how the GPS from different users' phones and their maps are connected in just a couple of rules. Programming concepts in ThingTalk greatly simplify writing social IoT apps like this one and many others. The simplicity opens up the possibility for ordinary people to write social IoT apps rather than experienced engineers; this can facilitate many innovative apps, which can be deposited in ThingPedia to be shared with other users.

1.3 Contributions

An Open Thing Platform (OTP) The Open Thing Platform enables the creation of distributed apps that leverage users' and friends' data, derived from a wide collection of things, in its computation. This platform consists of ThingPedia, an open repository of interfaces and apps of things. Apps are written in ThingTalk, which are run in the users' Thing System, consisting of the Thing Manager and Thing Engine services.

ThingTalk ThingTalk is a language that enables a class of useful apps be written in simple, declarative-style rules. Key concepts responsible for the expressivity of the language include the following: On the left hand side of each rule are *triggers* and *conditions* which, when true, cause *actions* on the right hand side to be taken. APIs for external services can be used as triggers in a rule. ThingTalk introduces a computation state represented as key-value pairs: conditions can be placed on the key-value pairs, and assignments to these pairs serve as additional actions. Multi-party communication capability is provided via feed-visible keywords; assignments to such keywords automatically result in sending data to the feed.

Experimentation of OTP We show a collection of ThingTalk programs that demonstrates many common day applications that can be enjoyed by many can be created in a few lines. These apps are 1 to 6 lines long.

1.4 Paper Organization

The rest of the paper is organized as follows:

- Section 2 gives an introduction on ThingTalk and shows a few examples of ThingTalk apps running on our system
- Section 3 introduces ThingPedia and the ThingManager part of the ThingSystem
- Section 4 details the syntax and semantics of ThingTalk
- Section 5 discusses the design and implementation of ThingEngine
- Section 6 contains our results from experimenting with ThingTalk, and addresses the limitations of the OTP

2. ThingTalk Examples

This section presents some examples of ThingTalk functions.

2.1 Single-Party Examples

We start by describing how ThingTalk can be used for connecting one's resources. Our first example is to set the profile picture of one's Twitter account if the Facebook profile picture has been changed. This profile synchronization can be expressed in a single ThingTalk rule as follows:

```
@facebook.profile (_, _, _, pictureurl, _, _, _, _)
    ⇒@twitter.setprofile (pictureurl);
```

The second example monitors the balances of a user's multiple bank accounts, and when the total of funds drop below $500, alerts the user. This example, shown below, illustrates how ThingTalk can bring information from different data sources together.

In each rule, the left hand side of ⇒ consists of one or more *conditions*, and the right hand has an *action*. A condition or an action can be referring to external interfaces, notated with an "@" sign, or keyword-value pairs, such as TotalBalance(b) in the second example, representing the state of the computation. A condition that refers to an external interface is called a *trigger*. Variables for the rule are

@Bank$_1$.getbalance(b_1)
  ⇒Balance$_1(b_1)$
@Bank$_2$.getbalance(b_2),
  ⇒Balance$_2(b_2)$
Balance$_1(b_1)$, Balance$_2(b_2)$
  ⇒TotalBalance$(b_1 + b_2)$;
TotalBalance(b), $b < 500$
  ⇒@$notify("Low balance");

bound by matching the formal variables with the actual values in the conditions, and are subsequently used in the actions.

Running a rule on a designated ThingEngine entails first installing the rules in the ThingEngine, then asking for the user's credentials if necessary. As the ThingEngine is executed on a trusted device, the user's privacy is honored. The ThingEngine will execute the rules continuously or until the result is reached according to the rule specification.

2.2 Multi-Party Example: LinkedIn at a Conference

ThingTalk enables the creation of distributed multi-party apps that accesses resources belonging to different individuals.

2.2.1 Motivation

Consider the scenario of a professional conference. Despite the presence of hundreds or even thousands of attendees, we typically ended up talking just a handful of them, and all by chance. In addition, of the few people we met, we would exchange business cards, which are often lost within hours if not minutes. What if we could exchange business cards automatically, and perhaps even detect common interests so we can find the right people to talk to?

Today many of us have joined the LinkedIn network and have already created a biography on the site. We can find out if the attendees come from, say, the same company, by reading our LinkedIn profiles and find matches. We can rely on LinkedIn to provide such function one day, but on the other hand, what if we do not want to have LinkedIn track each meeting that we attend. It is difficult for a third party to provide such functions because:

1. the overhead of authorizing each app is onerous
2. we may not want to leak information to a third party, and
3. more significantly, LinkedIn can deny access of a 3rd party if the latter is threatening LinkedIn's service.

Our philosophy is that our LinkedIn record belongs to us. Since our ThingEngine works on behalf of the user, there is little ground that LinkedIn can stand on to deny access. Thus, the ThingEngine of a group of participants can each fetch the LinkedIn profiles and share among the ThingEngines; the ThingEngines can save the LinkedIn "business cards" or compute with it. While it may appear that a centralized service can perform this function more efficiently, but our per-

```
LinkedInApp$_F$() {
  @linkedin.profile ($name$, $co$, _, _, _, _, _, _)
      ⇒Company$_F$[SELF]($name$, $co$);                                (R1)
  Company$_F$[SELF](_, $co$), Company$_F$[$m$]($name$, $co$), $m \in F$
      ⇒NewColleague($name$, $co$);                                     (R2)
  NewColleague($name$, $co$)
      ⇒Colleagues(Append(Colleagues, ($name$, $co$)));                 (R3)
  NewColleague($name$, $co$)
      ⇒@$notify($name$);                                               (R4)
}
```

Figure 3: The LinkedIn business card sharing app

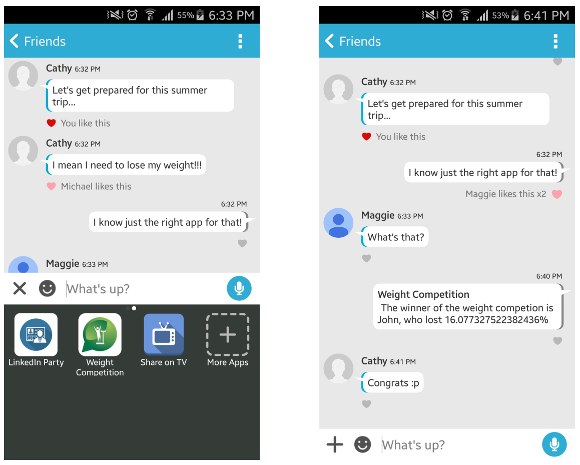

Figure 4: A possible scenario through the installation and use of the Weight Competition app. On the left, we see how the app is installed through the Omlet drawer. On the right, new results are produced as a message on the feed

sonal devices can dedicate much more resource and bandwidth to our own computations than what a centralized service can provide to the individuals.

2.2.2 ThingTalk Code

Shown in Figure 3 is the LinkedIn app for a given feed F written in ThingTalk. A multi-party ThingTalk app is parameterized by a feed. A feed consists of a unique identifier and a set of members. By joining a feed, a user is opting into sharing the capabilities of their resources with their friends. Shared keyword-value pairs are denoted with the subscript F. Every value stored into such keywords are shared across all the members in the feed. Individual's entry can be accessed by the *member* operator, denoted as []. A special keyword SELF allows us to refer to our own entry.

In this example, each user collects information about where he works. This information is shared to everybody in the feed. Once a colleague is identified, this information is inserted into a globally visible state and triggers a notification.

Putting it together In Figure 4 we see a screenshot of a group chat in Omlet. The user can choose any of the available ThingTalk apps from the drawer, and start them inside the current feed.

Other users then can opt in to the app by clicking on the invitation, and once everybody joins in the app starts in everybody's ThingEngine.

3. Representing Things

This section discusses how thing interfaces are specified in ThingPedia, and how a ThingManager manages a user's things in consultation with ThingPedia.

3.1 ThingPedia

ThingPedia is a crowd-sourced repository of thing definitions as well as ThingTalk apps. Thing definitions are mostly likely to be contributed by device manufacturers and web service providers. We imagine that technically savvy individuals will create the ThingTalk apps. Consumers will browse the ThingTalk apps in ThingPedia, like how they browse Apple App Store or Google Play Store, and download apps to their ThingEngine. Apps can be suggested based on the things the user owns.

3.2 Thing Classes

In ThingPedia is a repository of open-sourced thing class definitions. A *thing class* is a specification for things with the same interfaces. For example, LinkedIn is a thing class, each LinkedIn account is an instance. A class can also be a product model, such as the "lg-webos2-tv", as things from the same model must share the same interfaces.

A rule can refer to an object of interest by (1) a global name, such as LinkedIn.com, (2) a class name such as "lg-webos2-tv", or (3) by attribute-pair values. For example, there may be many models of GPS devices, but they all return a location. Thus, "type=GPS" refers to any nearby device with GPS capabilities. All the components of a thing class are described below:

Class name. The name of the class.

Name. A optional global name that can be used in rules to refer to this thing.

Attributes. An optional list of keyword-value pairs. A class definition may include attributes that apply to all instances, such as the device type. Users can also provide additional attribute-value pairs, such as condition="broken", location="home" or purpose="school".

Class Fields. These fields specify the properties of the specific instance of a thing. Every instance has an ThingID field. For online accounts, the ThingID is typically the account holder's user name or account number. For local home automation and home entertainment, the ThingID is like to be the BSSID of the home WiFi plus the local IP address or the Ethernet address of the device. Typical additional fields include credentials associated with the account.

Constructor method, to be invoked when an instance is configured to initialize the instance.

APIs and their implementations, each of which can be either a trigger or an action. Each API is invoked on an instance and thus have access to the ID fields.

For example, a LinkedIn is a thing class, each LinkedIn account is an instance. The ThingPedia entry for LinkedIn has the following information:

Class name: LinkedIn

Name: linkedin.

Class definition: ThingID corresponds to the LinkedIn account. It has an addition field, OAuth, to hold the OAuth token for the account.

Constructor: Calls the LinkedIn login API which presents the users with a login screen to obtain the user's ID and OAuth token.

APIs: The following are two example implementations.

- @linkedin.profile (id, name, headline, co, ...) This is a trigger that returns the user's profile whenever it is changed. It is implemented by polling the user's LinkedIn account once a day. Information returned includes the id, full name, headline, company, specialty and profile picture fields.
- @linkedin.share (update) This is an action entry point to post a new update. It takes a JSON object and calls the appropriate LinkedIn API.

3.3 ThingManager

A ThingManager manages the service to discover, create, and manage a user's thing on his behalf. We refer to an instance of a thing as a *ThingObject*, and every instance has a unique *ThingID* assigned by the Thing Manager. The ThingManager keeps a persistent directory of all the registered things.

As discussed above, a thing in a rule can be described by its ID directly, a global name like @LinkedIn, or by attribute-value pairs, such as "type=GPS", or "location=home". The ThingManager returns the ThingObject associated to the description if one exists, otherwise it performs the following steps.

1. Resolve the ThingObject for a given thing descriptor. If the thing descriptor uses a global name, such as @linkedIn, then its class is fetched from ThingPedia by name. Otherwise, ThingManager performs a local discovery for all nearby devices using a discovery protocol such as UPnP [UPnP Device Architecture – Part 1: UPnP Device Architecture Version 1.0] or AllJoyn [AllJoyn Framework] or manually from an interaction with the user. For each new thing discovered, it matches the descriptor from the class definition retrieved from ThingPedia.
2. ThingManager invokes the Constructor method from the ThingClass. For example, @facebook requires the user's Facebook credentials to be instantiated. The created ThingObject instance is stored in the directory for later lookup.

```
<function> := <name> '(' <params> ')'
              '' <body> ''
<params> := | <param>  | <params> ',' <param>
<param> := <name> ':' <type>
<body> := <rules>
<var-decls> := <var-decl> | <var-decls> <var-decl>
<var-decl> := <name> ':' <type>
<rules> := | <rule> | <rules> <rule>
<rule> :=  <thing-invocation>
           | <thing-invocation> ',' <conds>
           '=>' <action> ';'
<conds> := <cond> | <conds> ',' <cond>
<varlist> := '(' names ')'
<names> := | <name> | <names> <name>
<cond> := <keyword> <ownership>? <varlist> |
          <function-invocation> |
          <name> in F
<function-invocation> := '$' <varlist>
<thing-invocation> := '@' <thing> '.' <name> <varlist>
<thing> := <name> | '(' <attr-values> ')'
<attr-values> := <attr-value> |
                 <attr-values> ',' <attr-value>
<attr-value> := <name> '=' <string>
<action> := <thing-invocation> |
            <keyword> <ownership>? <varlist>
<keyword> := <private-keyword> | <shared-keyword>
<private-keyword> := <name>
<shared-keyword> := <name>_F
<ownership> := '[' ( <name> | SELF ) ']'
```

Figure 5: The BNF (Backus-Naur Form) of ThingTalk function declaration.

4. The ThingTalk Language

A ThingTalk function is parameterized by a *feed*, consisting of a unique identifier and a set of member ThingEngines on which the code is to be run. If no feed is specified, it runs on the user's own ThingEngine in an unnamed feed.

Figure 5 shows the syntax for ThingTalk functions in BNF (Backus-Naur Form). Each function consists of a set of rules of the form, $c_1, c_2, \ldots \Rightarrow a$: the left hand side consists of a set of conditions and the right hand side is the action to be taken. When executed, these rules are inserted into the ThingEngine; whenever the conditions on the left hand side are satisfied, the action on the right hand side is taken. This repeats until the function terminates.

All "things" in the system are prefixed by the "@" symbol, all other predicates refer to the computation state. We discuss the computation state first, then invocations of the thing interfaces.

4.1 Computation State

The state of the computation of each ThingEngine is represented by a set of keyword value pairs. The scoping of keyword value pairs is defined as follows:

Feed accessible. A keyword subscripted by a feed F, key$_F$ indicates that this value is visible by the ThingTalk function running on all the member ThingEngines in feed F. A feed keyword has a member operator []; for example, location$_F[m]$, refers to the value of keyword location in m's ThingEngine. A special keyword SELF allows us to refer to our own entry.

Local. All other keywords are simple local keywords accessible by only the function on the same ThingEngine.

Computation State Rules The left-hand-side is a set of conditions on the state of the computation, as defined by the keyword value pairs. Formal variables in the rule are bound to the actual values corresponding to the key-value pairs. A condition key(v) says that variable v is bound to the value of the keyword key. Unification is applied if the same variable is used in multiple conditions. Additional predicates such as arithmetic operations and function calls are allowed. Condition evaluation should have no side effects. An action, key(v), says that the keyword key is assigned the value v.

The left hand side is executed whenever any of the inputs are changed; if the condition is satisfied, then the right hand side action is taken.

Pure pre-defined functions are all denoted by a "$" character preceding the function name. The pre-defined functions have input parameters and output parameters; the input parameters must be bound to values by other keywords or functions. The functions are considered to be true if there exist output values for given inputs.

4.2 Things and Their Interfaces

Each interface in a Thing Class must be declared to be either a trigger or an action; triggers can appear only on the left hand side of a rule, and an action can appear only on the right hand side. The left hand side of a rule can have at most one thing invocation. While additional conditions may be placed on the result of a thing trigger, changes to any of the parameters in the condition will not initiate the execution of the rule. If the condition is true, then the action on the right hand side is taken.

As discussed in Section 3.2, things can be indirectly selected by their attributes. For example @(type = "TV", location = "livingroom") refers to the TV(s) located in living room. If the left hand side refers to a trigger by attributes, as long as there is one match, the rule will fire. If the right hand side refers to an action by attributes, the action is taken on *all* the actions that match.

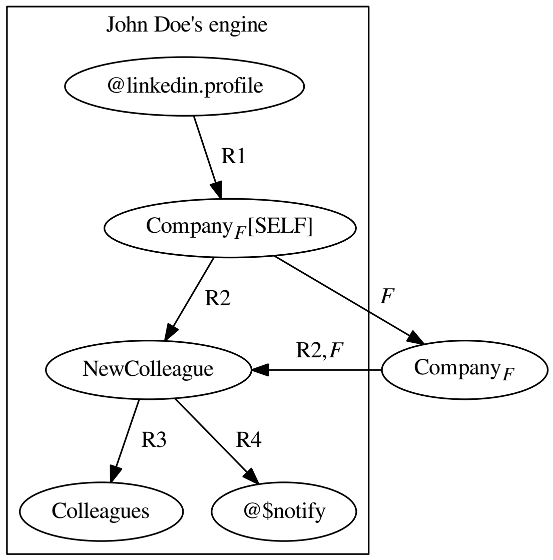

Figure 6: The data dependency graph generated for the LinkedIn app in Figure 3, when run on John's ThingEngine

5. Design and Implementation of ThingEngine

When a ThingTalk app is run, it first consults with the ThingManager to bind the thing descriptions with a user's things as discussed in Section 3.3. It then invokes the ThingEngine, passing to it the rules ad the parameters. The ThingEngine returns a unique identifier, which the app can use to kill the rules.

5.1 Executing a ThingTalk App

The ThingEngine takes the following steps in running a ThingTalk app.

1. Compile the parameterized ThingTalk function, insert the code in the persistent App directory, and reserve the keyword space.
2. Create the dynamic mapping from the thing description to the thing instance by consulting with the ThingManager.
3. Compute a dependence graph of all triggers, actions and keyword updates.
4. The event loop starts processing events from the triggers, evaluates each edge in the dependence graph until convergence.

5.2 Data dependence graph

The data dependence graph has a node for every trigger, every action and every keyword that is named in a rule. Each rule corresponds to an edge in the graph that links the rule's left hand side to its right hand side, and as such there are no incoming edges on nodes that represent triggers, or outgoing edges from nodes that represent actions (because these cannot be nominated on the opposite side of a rule). Figure 6 shows the dependence graph of the LinkedIn app in Figure 3.

A new value in any keyword node propagates down to its children, by evaluating each outgoing edge, and changing the value of the keyword node pointed to by the edge. Edges are evaluated in arbitrary order until there are no more changes to be applied to values of the keywords.

Calls to the triggers from the thing interface code will cause a trigger node to be have a new value, and will cause evaluation of the edges outgoing from it. Conversely, a new value on an action node will cause a call to the corresponding thing interface.

5.3 Feed accessible keywords

In the case of feed-accessible keywords, the data dependence graph constructed by the engine is augmented by splitting each feed accessible keywords into a SELF and a portion that is a proxy for the data on the remote engines. The edge that points to the proxy node is the edge tagged with the feed F.

Evaluating an edge tagged with F going from a node to a node in a different engine causes the originating engine to send a message with the new data over the feed. Similarly, an incoming edge tagged with F signals to the engine that it should expect incoming messages with the new value of the node source of the incoming edge.

5.4 State persistence and recovery

At any time, the current state of all enabled ThingTalk functions and their parameters, and the state of named keywords, including the current state of the feed accessible keyword portions that the engine does not own, is stored by the engine on disk, which allows it to be resilient in face of crashes or unreliable networking.

Additionally, the engine can spontaneously request a refresh of the cached data from all other members of a feed, which updates the engine view of the feed accessible keyword state. This refresh is always automatically requested when the engine first joins a feed, when a new app is installed, when the engine starts and when network connectivity is reestablished.

5.5 Implementation

We implemented the ThingSystem using a long running nodejs application, with a platform adaptation layer that allows it to run on Android as well as a generic server or a shared cloud service.

The data dependence graph is realized by registering a callback for each rule that is invoked on every change to a keyword and on any invocation from triggers.

Feed-accessible keywords are implemented by maintaining a local persistent list of subscriptions, that record whether a message should be sent to a given engine in response to a local change to a keyword.

Messaging is implemented using the Omlet messaging platform [Omlet Chat], which provides the primitives of sending and receiving data from a feed.

6. Experimentation

In this section, we present a selection of sample apps to illustrate the various functionalities provided by ThingTalk. Figures 7 and 8 show a summary of the single-party and multi-party apps, respectively. As we see from this figure, a small number of rules are needed to automate a large variety of tasks. Single-party apps range from simple utilities to financial monitoring, home automation, and personalized entertainment. Multi-party examples include compelling data sharing, gamification, and collaboration apps. These tasks may be event driven, timed, one-shot or continuous. These apps web accounts that contain financial data, location, weight, and intellectual property such as our slides. With the help of the thing class definitions in ThingPedia, all these apps can be written no more than 6 rules in ThingTalk.

6.1 Single-Party Functions

We now discuss the details of our single-party ThingTalk functions. These apps are closest to recipes in IFTTT, we will highlight the differences between ThingTalk and IFTTT in the discussion. Note that IFTTT does not have a language, and users input all the information by filling out forms on the web page.

Keep your profile pic updated One of the most popular recipes on IFTTT is synchronizing user profiles, such as changing a user's Twitter account profile whenever the Facebook account changes. The following ThingTalk rule does the profile photo synchronization:

@facebook.profile (_, _, _, *profileUrl*, _, _, _, _)
⇒@twitter.setprofile (*profileUrl*);

The Facebook service is represented by @facebook thing; it provides a profile interface that is triggered when the user's profile information changes. The above rule specifies that the rule is only interested in *profileUrl* changes; other fields are denoted with underscores indicating that they are not needed. On the action part of the rule, @twitter represents the Twitter service; it has setprofile interface that updates the users Twitter profile picture with the given URL.

To monitor profile picture changes and to update it, IFTTT requires its users to give their Facebook and Twitter login credentials to IFTTT. Applications in ThingTalk do not require giving up one's login credentials; the application runs on one's own ThingEngine, that securely stores one's credential information whether the engine runs on one's home server or in the cloud.

Video recommendation from social feeds One's social feeds such as Twitter have plenty of online video streams.

App	Purpose	Category	When	Things	Private data	# of rules
Facebook to Twitter	Update Twitter profile when Facebook profile changes	Common digital task	Event triggered	Facebook, Twitter	Account credentials	1
Bank	Monitor user's multiple bank accounts	Financial monitoring	Continuous	Bank	Financial	3
Twitter to TV	Pipe recommended videos from Twitter to the TV	Personalized entertainment	One-shot	TV, Twitter	Account credentials	1
Heatpad	Turn on/off heatpad at specified time	Home automation	Timed	Heatpad		2

Figure 7: Sample Single-Party Applications in ThingTalk

App	Purpose	Category	When	Things	Private data	# of rules
GlympseApp	Share locations, draw map	Data sharing	Continuous	GPS	Location	2
LinkedIn	Find colleagues in a group	Collaboration	Event triggered	LinkedIn	Account credentials	4
RaceApp	Determine winner in a race	Gamification	Continuous	GPS	Location	2
Weight Competition	Track weight in a weight-loss competition	Gamification	Continuous	Scale	Weight	6
Slides	Share slides remotely on TVs	Collaboration	Event triggered	TVs	Slides	2

Figure 8: Sample Multi-party Applications in ThingTalk

Since friends (and followers) often share common interests, these videos appearing on social feeds capture the user's interest very well. Suggesting videos based on one's social feeds can be a simple yet very useful application.

Here we implement the app that suggests YouTube videos based on one's Twitter feeds. There is one rule in the app that looks for YouTube URLs from your Twitter feeds and add those URLs to the user's TV.

```
@twitter.gettweets(_, url, _),
$Contains(url, “http://www.youtube.com”),
    ⇒@(type=“TV”, location=“livingroom”).enqueue(url);
```

@twitter.gettweets() is triggered when there is a new tweet in one's feed; the rule says that it is only interested in tweets with urls. When triggered, the condition in the rule tests if the URL points to a YouTube video using the pre-defined function ($Contains); if so, the URL is enqueued into your living room TV's watch list. This example illustrates how we can connect our web resources to our IoT devices.

Turn on Heat Pad at Night A simple but useful class of IoT applications is home automation. As an example, the following code shows how we can automatically turn a heat pad on and off at a given time in two ThingTalk rules:

```
@$at(“3:00”) ⇒@(type=“heatpad”).on();
@$at(“6:30”) ⇒@(type=“heatpad”).off();
```

The first rule is triggered at 3:00am in the morning; it selects all your things that have “heatpad” type and invokes their on() interface. Similarly, the second rule turns off your heatpad(s) at 6:30am.

For this application, we used heat pads [Parklon Iris Warm Water Mat] that support remote control via the XMPP protocol [Extensible Messaging and Presence Protocol XMPP]. The “type=heatpad” attributes were given to the heatpads nearby at configuration time. This app is used by one of the authors of this paper in his daily life.

While this app does relatively simple tasks, it is still surprising how succinct it is to write this in ThingTalk. To understand its power of abstraction, let us look at similar apps in IFTTT. These IFTTT apps [IFTTT Recipe: at sunset, turn on your lights; IFTTT Recipe: Every day at given time, turn the lights on] turn your lights on at a specific time or at sunset.

In IFTTT, you cannot describe actions for a group of things that have common attributes, such as types; so you must write separate apps to control the lights of different brands (e.g. one for Philips Hue lights and another for WeMo lights). Moreover, IFTTT does not have the notion of functions that group related rules together. so you cannot organize the rules that are conceptually related. This is a nuisance not only for programmers, but also the users. Most importantly, ThingTalk enables computation using results from different triggers, as illustrated in the banking balance example in Section 2.1.

6.2 Multi-Party Programs

Racing with Friends The RaceApp, shown in Figure 9, illustrates how we can easily gamify our daily routine, such as meeting up with our friends, in ThingTalk. This app shares the GPS locations among friends in a feed and declares a winner if a user's location is close enough to the destination. Note that if a group of friends use both GlympseApp, shown in Figure 2, and the RaceApp together, the ThingEngine will automatically detect the commonality and only one such measurement needs to be taken.

```
RaceApp$_F$($dest$: Location) {
  ¬ Arrived$_F$[SELF](_),
  @gps($l$,$t$),
  $distance($l$, $dest$) <= 50m
      ⇒Arrived$_F$[SELF]($t$)
  Arrived$_F[m](t), m \in F$
      ⇒@$return(ArgMin(Arrived$_F$))
}
```

Figure 9: Racing with Friends

Weight Competition Among Friends Competition among friends can make solo activities, such as losing weight, into social activities. Gamification has been shown to be effective in helping one achieve his goal as well as enjoy the process of it. RunKeeper [RunKeeper] or Endomondo [Endomondo] are examples of commercial apps that keep track of one's running and share (or compare) it with friends.

ThingTalk makes it possible to write a weight competition among friends in just a few rules. Shown in Figure 10 is the WeightCompetition function in ThingTalk, parameterized by the feed F and *stopTime*, which indicates the end time of the competition. This function continuously monitors the weight of the people on feed F, using the measurements received directly from their scales. The winner at any one time is the person who has lost the largest percentage of his weight.

The function has six rules. Rule R1 reads the weight measurement from a user's scale and assign it to the Weight keyword. Rule R2 is triggered with the first weight measurement in the competition, the InitialWeight keyword is update and its value is shared with everybody in the feed. Rule R3 is triggered with every weight measurement to compute the ratio of weight lost. The Loss keyword values are also shared in the feed. Rule R4 finds the one with the maximum weight loss, and rule R5 notifies who the winners are. The last rule R6 is triggered at the given stopTime and returns the winners. This example illustrates how the use of the feed abstraction makes sharing trivial.

```
WeightCompetition$_F$($stopTime$: Time) {
  @(type="scale").measure(w)
      ⇒Weight($w$);                              (R1)
  Weight($w$), ¬ InitialWeight$_F$[SELF](_)
      ⇒InitialWeight$_F$[SELF]($w$);             (R2)
  InitialWeight$_F$[SELF]($w_1$), Weight($w_2$)
      ⇒Loss$_F$[SELF]($(w_1 - w_2)/w_2$);        (R3)
  Loss$_F[m](l), m \in F$
      ⇒Winners(ArgMax(Loss$_F$));                (R4)
  Winners(w)
      ⇒@$notify(Winners);                        (R5)
  @time(stopTime)
      ⇒@$return(Winners);                        (R6)
}
```

Figure 10: The WeightCompetition App.

Displaying and sharing content between devices In this last example, we show how a ThingTalk can be used to enable collaboration using a multitude of IoT devices. Let's consider WebEx, one of the most popular applications that lets us share our desktop with remote audiences. The typical use of WebEx is to show and advance slides as one gives a presentation. The setup involves sending the session identity and passwords to all the participants. More importantly, the information shared is all uploaded to a third party, who also charges a nontrivial amount for the service.

The Slides function, shown in Figure 11, allows a group of friends in a feed to share slides in a collaborative fashion and have their pictures be displayed on their own big-screen TVs. Here, any of the participants can submit pictures to the app, as shown in rule R1. Rule R2 says that if any of the participants has a TV configured among his collection of things, then the pictures are automatically shown there as well.

This example illustrates that many centralized services can be provided in a distributed manner using our Open Thing Platform. First and foremost, the data shared are kept private. Second, if we wish to work with the same group of people all the time, and possibly using different apps, we only have to set up a feed once. This avoids the overhead of setting up sessions for each instance of collaboration. Finally, our examples show how easy it is to create new social apps. Our Open Platform can promote the creation of a wide collection of small and large collaborative software.

7. Related Work

7.1 Internet of Things

Commercial Internet of Thing Platforms SmartThings [Samsung SmartTthings], Vera [Vera Smarter Home Control] and Parse [par] are commercial services that allow developers to write Internet of Things enabled apps using a

$\text{Slides}_F() \{$
$\quad @\$\text{input}(url)$
$\qquad \Rightarrow \text{SharedSlides}_F[\text{SELF}](url); \qquad \text{(R1)}$
$\quad \text{SharedSlides}_F[m](url),\ m \in F$
$\qquad \Rightarrow @(\text{type="tv"}).\text{show}(url); \qquad \text{(R2)}$
$\}$

Figure 11: Sharing Slides collaboratively across TVs.

Figure 12: Images are sent from mobile phones to a big-screen TV.

traditional programming language and a set of APIs. SmartThings and Vera operate primarily in the context of home automation, while Parse is mostly used for social networking and analytics. Apps developed using SmartThings and Parse run on their cloud and interact all the devices that the user has configured.

Amazon IoT [Amazon Web Services IoT] extends on the SmartThings and Parse architecture with a declarative programming language, based on a dialect of SQL.

Declarative Internet of Thing Platforms DNS (Declarative Sensor Network) [Chu et al. 2007] is a Datalog adaptation to describe the operation of multiple Internet of Things enabled devices with a declarative language. Their systems models sensors and actuators, but does not model shared state, and as such does not extend directly in the social context.

7.2 Social sharing and automation systems

Commercial chat services Multiple commercial chat services exist, such as Facebook Messenger, Snapchat, Line and WeChat.

WeChat in particular has a system and a set of APIs that allow trusted developers to build social applications that are executed by all users of a chat. This is used as a gaming platform, by delivery services and for banking.

The main difference between the Open Thing Platform and these existing systems is that they rely on a central service that observes and controls all the user traffic, while the OTP operates atop a distributed and privacy sensitive messaging system that is agnostic to the data being delivered.

Web automation services IFTTT [If-This-Then-That] is a web automation service that allows users to create rules which can be triggered by events and in turn cause actions to occur.

Fundamentally, like IFTTT, the ThingEngine is a rule execution engine. However the ThingEngine evolves from IFTTT in its ability to maintain state across multiple rule executions, and in the ability to specify complex conditions that span multiple triggers.

Additionally, IFTTT is a centralized services, while the Open Thing Platform is built in a distributed fashion, and there is no data leak unless the user explicitly shares it.

Cloudwork [CloudWork Cloud Business App Integrations] and Zapier [zap] are also web automation services, but they focus on social web and online business application and cloud storages, rather than IoT devices.

Publish/subscribe systems Publish/subscribe [Eugster et al. 2003] is a well established paradigm used to synchronize a copy of a dynamic resource between multiple interested parties. Common publish/subscribe systems use content filters that are evaluated on a central server, and do not use the filters to address routing of the messages.

Contrail [Stuedi 2011] is a publish/subscribe system for social media that provides user privacy by evaluating the subscription filter only on the edge nodes and using end to end encryption. We extend on their work by decoupling the messaging infrastructure from the data sharing part, and by providing a declarative language to express the data interests of the multiple parties.

Poster [Tryfonopoulos et al. 2015] is another distributed publish/subscribe system that allows for multiple users to share dynamic data. They only focus on the synchronization aspect of the accessing the data, and as such they require a complex mechanism of distributed filtering, while we rely on multiple executable high level apps running on each node only transmitting relevant data. We also sidestep the issue of distributing the messages by building on an existing messaging system.

7.3 Rule based systems

Active Database Systems Active database systems support active rules that are automatically executed if given conditions are satisfied [Chakravarthy et al. 1994; Widom 1996; Gehani and Jagadish 1991]. The rules consists of three parts: *event*, *condition*, and *action*. *Event* describes what causes the rule to be triggered; typically events are relational data operations (insertion/deletion/update), database operations (such as transaction begin/end), or temporal events. Once a rule is triggered, *conditions* are checked to determine the execution of the action. Conditions are predicates on the data in the tables. Actions in active rules are either data manipulation operations or database operations (such as drop a table or abort the transaction). The ThingTalk rules are different from the active rules; they are designed to help writing social IoT applications and support necessary primitives, such as member operator or feeds as first class object.

Datalog Datalog is a declarative programming language for deductive databases. Initially introduced for logic programming, Datalog has gained popularity recently in many other areas including programming analysis [Whaley and Lam 2004], network systems [Alvaro et al. 2010; Loo et al. 2009], modular robotics [Ashley-Rollman et al. 2009], as well as large-scale graph analytics [Seo et al. 2013a,b]. Datalog programs consist of declarative rules. The high-level semantics of Datalog rules simplify writing programs in the aforementioned areas. ThingTalk apps have declarative rules that have some resemblance to Datalog rules; the rules have dependencies and updates are propagated through the rules. However, ThingTalk rules are more expressive than Datalog rules because ThingTalk rules are powered by ThingSystem services such as ThingManager, Messaging Layer, and ThingEngine; they can interact with many real-world things and enable a wide range of social IoT apps.

Tuple Spaces A tuple space is a programming language and paradigm for parallel processing [Gelernter and Carriero 1992; Carriero et al. 1994]. In this language is supported *tuple spaces*, which is a virtual, associative, logically-shared memory. A tuple space contains tuples, an ordered sequence of data. Multiple processes can concurrently access the tuples in the space that match a certain pattern. This is also referred as the blackboard model of computation. Keywords in ThingTalk are conceptually similar to tuple spaces. However, the context where the model is applied is very different. In ThingTalk we use the model to coordinate the communications among one's (and others) digital resources and implement social IoT apps.

8. Conclusions

This paper presents the Open Thing Platform (OTP) which shifts the computational paradigm from centralized services to distribute execution. Just like how PCs replace mainframes for personal computing, we believe the era of billions of mobile and IoT devices will disrupt the current trend of centralized proprietary services that own our personal data in different silos.

Our proposed OTP allows users to bring all their data together, either stored in our myriad of web accounts or generated by our personal internet of things. We can connect these resources together and automate many of the previously manual tasks. More importantly, our Open Thing Platform enables peers to share information and devices, without an intermediate who sees all the transactions. (Note that it is trivial to add encryption to data shared in a feed if the underlying messaging layer's protection of privacy is questioned.)

The distributed computing model of OTP requires individuals to have a ThingSystem service running on their behalf. Just like everybody has a FileSystem that manages our files on a device, we believe that it is natural that we have a ThingSystem that manages all our resources. All the examples we showed in this paper can be served by just running a ThingSystem on our mobile devices.

The ThingTalk language is succinct and expressive, making it possible to write a large variety of apps in just a few lines. The key ideas in ThingTalk include:

- The use of large building blocks, which are interfaces to web services and the internet of things, enables complex tasks to be described succinctly.
- An open-source crowd-sourced ThingPedia enables the collaboration of device makers and developers to create a common repository of interface codes and ThingTalk apps. Such a web service can also encourage the standardization of interfaces, because new devices are incentivized to use the interface used in popular devices with similar functions.
- The use of the feed abstraction makes it easy to write distributed code. Developers only have to specify that certain keywords are common to the feed, the system automatically generates the necessary messaging code.

Just like how MapReduce has a great impact on parallel computing by making it accessible to a larger audience, ThingTalk enables many more developers to create social and cross-device apps.

The Open Thing Platform also makes it convenient for users:

- The ThingManager lets users configure their devices once and use them in a wide range of applications. Also the ability to refer to devices with attributes makes it easy for users to use different brands.
- A group of friends needs to set up a feed only once, and they can reuse the feeds across many different apps without any further setups.

In conclusion, the major advantages of OTP include protection of privacy, scalability through distributed computing, and the ability to compute with data gathered from a wide disparate resources owned by different individuals. In addition, the succinctness and expressiveness of ThingTalk, as well as the convenience offered to users, can lead to a proliferation of social, collaborative applications.

References

URL https://parse.com.

URL https://zapier.com.

AllJoyn Framework. URL https://allseenalliance.org/framework.

P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears. Boom analytics: Exploring data-centric, declarative programming for the cloud. In *EuroSys*, pages 223–236, 2010.

Amazon Web Services IoT. URL https://aws.amazon.com/iot/.

M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell. A language for large ensembles of independently executing nodes. In *ICLP*, pages 265–280, 2009.

N. Carriero, D. Gelernter, T. G. Mattson, and A. H. Sherman. The linda® alternative to message-passing systems. *Parallel Computing*, 20(4):633–655, 1994.

S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of VLDB '94*, pages 606–617, 1994.

D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 175–188, 2007. ISBN 978-1-59593-763-6. doi: 10.1145/1322263.1322281.

CloudWork Cloud Business App Integrations. URL https://cloudwork.com.

B. Dodson, I. Vo, T. Purtell, A. Cannon, and M. S. Lam. Musubi: disintermediated interactive social feeds for mobile devices. In *Proceedings of the 21st international conference on World Wide Web*, pages 211–220. ACM, 2012.

Endomondo. https://www.endomondo.com/.

P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120, RFC Editor.

N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of VLDB '91*, pages 327–336, 1991.

D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96–107, 1992.

If-This-Then-That. URL http://ifttt.com.

IFTTT Recipe: at sunset, turn on your lights. https://ifttt.com/recipes/94447.

IFTTT Recipe: Every day at given time, turn the lights on. https://ifttt.com/recipes/105700.

B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, 2009.

Omlet Chat. URL http://omlet.me.

Parklon Iris Warm Water Mat. URL http://www.amazon.com/Parklon-Iris-Warm-Water-Mat/dp/8995691026.

RunKeeper. https://runkeeper.com/.

Samsung SmartTthings. URL http://www.smartthings.com.

J. Seo, S. Guo, and M. S. Lam. SociaLite: Datalog extensions for efficient social network analysis. In *Proceedings of ICDE '13*, pages 278–289, 2013a.

J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed sociaLite: A datalog-based language for large-scale graph analysis. *PVLDB*, 6(14):1906–1917, 2013b.

P. Stuedi. Contrail: Enabling decentralized social networks on smartphones. *Computer Science Journal*, 7049:41–60, 2011.

C. Tryfonopoulos, P. Raftopoulou, V. Setty, and A. Xiros. Towards content-based publish/subscribe for distributed social networks. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 340–343, 2015. ISBN 978-1-4503-3286-6. doi: 10.1145/2675743.2776770.

UPnP Device Architecture – Part 1: UPnP Device Architecture Version 1.0. ISO/IEC 29341-1:2011.

Vera Smarter Home Control. URL http://getvera.com.

J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

J. Widom. The starburst active database rule system. *IEEE Trans. Knowl. Data Eng.*, 8(4):583–595, 1996.