Sabrina: A Decentralized, Trigger-Action Based System for the Internet of Things

Giovanni Campagna
School of Engineering
Stanford University
Stanford, California
gcampagn@cs.stanford.edu

Albert Chen
School of Engineering
Stanford University
Stanford, California
hselin@cs.stanford.edu

Ramon Tuason
School of Engineering
Stanford University
Stanford, California
rtuason@cs.stanford.edu

Abstract—In this paper, we will discuss the process taken in the creation of Sabrina, a friendly, virtual assistant that helps users make efficient use of efficient use of our trigger-action based system. This system is built on rules, which are logic expressions comprised of inputs (triggers) and outputs (actions). When the system detects a rule's triggers, it causes its actions to be executed in turn. Sabrina is grounded in Rulepedia, a central web server that hosts channel code and metadata, and also stores user-generated rules that users can install and edit. But to avoid using an impersonal interface, we built Sabrina to speak like a human to the user. One of our main reasons for building this project was to tackle problems surrounding data control and user privacy with regards to rule executions that depend on a centralized server, like IFTTT. We also sought to tackle the current limitations of these systems in terms of flexibility and user friendliness.

I. Introduction

Run by a cloud-based web service of the same name, the IFTTT model ("IF This, Then That") allows users to connect and have creative control over various physical and virtual products. A user can setup multiple "rules," where each rule contains a trigger and an action. When the system detects the trigger being executed, it executes the action of the rule in turn. As we can see, the power of the IFTTT model allows users to integrate the capabilities of disparate devices and services to bring greater convenience to their lives. For example, a homeowner could create a rule in which a security camera right outside his front door takes a picture when someone rings the doorbell. The rule could then automatically send that picture to the homeowner through SMS, allowing him to know if it is safe to open the door. To show how IFTTT can go as far as save a person's life, a user with a fitness band could set up a rule in which his phone automatically calls 911 if the band notices a dramatic decrease in his heart rate. [1]

Related to the IFTTT model is the ever-growing Internet of Things (IoT), the networks of physical objects embedded with the software and hardware needed to autonomously interact with each other. As the objects within an IoT network each receive a unique identifier, their individual capabilities and usefulness improve significantly since they do not require human-to-human or human-to-computer interaction to transfer data to others in the network. By making effective use of wireless technologies, micro-electromechanical systems (MEMS) and the Internet, IoT makes data collection and control much more convenient, enabling people to make better use of the interoperable devices that they own. For example, in the agriculture sector, sensor posts installed by farmers can be used to aggregate data about the current states of the soil and weather, allowing these farmers to make more informed decisions about their work going into the future. [2]

II. Motivations and Contributions

A. *User Privacy and Data Control*

Given how powerful and flexible the IFTTT model and the Internet of Things are, our group wanted to explore how they work and learn what we could do to take these technologies a step further. During our research, we considered how IFTTT performs the execution of rules. As their service stores all the rules that their users create, it requests that their users allow it to connect with the products that these rules make use of. For example, if a user wants to set up a rule in which he receives an Android notification every time a specific friend makes a Facebook status, he would have to provide IFTTT with access to his Facebook account. And since these rules are stored in IFTTT's servers, they essentially have control over the user's data. This is one of the problems that our group sought to address. Building upon the IFTTT model, we want to allow users to run their rules without having to reveal sensitive information to external services. By maintaining control over their own data, they uphold the integrity of their privacy.

B. *User Friendliness*

In addition, as technology becomes more and more prevalent, it is important for developers to remember to keep it reasonably usable to the general populace. Keeping this in mind, our group sought to build a system that abstracts complications away from the user. This is where Sabrina plays a role in our application. Built to act like a "magical, personal assistant," Sabrina helps users get accustomed to use our rule-based system. Rules are described as "spells" in which triggers magically cause actions to be executed without the involvement of the user. Channels, which rules work through to detect triggers and run actions (e.g. Facebook, Google Fit, physical devices, etc.), are described as "wands" that Sabrina uses to cast spells, keeping users from having to learn about the technical intricacies of how these channels operate. By creating a fun and friendly personality that eases users into working with our rule-based system, we hope to optimize their experience with our application.

C. *Flexibility*

Our system builds upon the flexibility that IFTTT offers. As of now, their service only allows for one trigger to execute one action. While users can set up multiple rules with the same trigger but different rules, they currently cannot create rules with multiple triggers that have to be detected before executing an action. With our more expressive rule model, users have the freedom to create more specific, elaborate rules. In addition, because we do not use a centralized server to run our users' rules, our users have true ownership over their data. They need not fear that any of their information is being sold off to third parties, since we the developers and administrators never see this information anyway. In line with the decentralized nature of our system, Sabrina has direct access to the features of our users' phones (e.g. calls, texts, geolocation, camera, etc.). This allows the application to run offline with rules that don't rely on retrieving data from the Internet, since a user's rules are stored in and run directly in the smart phone. Finally, we built Sabrina such that she would be aware of the user's local environment. By adding the power of IoT integration into our application, we allow Sabrina to detect nearby devices so that the user can smoothly integrate them into the rule-execution process. For example, imagine a user walks into a hotel room with an IoT thermostat that Sabrina can interact with. The user has a rule in which Sabrina provides his temperature/fan speed preferences to a detected thermostat. Once Sabrina notifies the user about the IoT thermostat and receives approval from both ends (the user and the thermostat) to send these preferences over, the room will automatically start to feel comfortable for the user.

III. Related Works

Our application combines the strengths of various technologies in industry. To promote user accessibility and make users feel more welcome when they are introduced to the system, a friendly human-like personality similar to those by Microsoft, Apple, and Google helps them learn about what they can and can't do with ease. Our backend makes use of an improved IFTTT rule execution model that allows for greater flexibility without feeling too overwhelming for the user. Finally, our device discovery mechanisms use existing technologies that maximize our application's potential for integrating into the Internet of Things.

A. *Front-End Personality*

XiaoIce is a virtual, social assistant that people can add as a friend on various Chinese social networking sites like Weibo, a microblogging services similar to Twitter that is used by about 700 million people. By adding XiaoIce to a chat room, users can talk with her for extended periods of time, but what differentiates her from ordinary bots that most people are familiar with is her distinct personality and her ability to have a sophisticated conversations. With complex natural language processing and artificial intelligence running in the background, XiaoIce can seem like teenage girl who chimes in with context-specific facts about a range of topics like sports and finance. On top of this, she can speak with sensitivity, empathy, and humor. She can adapt to a given situation by varying her speaking pattern and molding her responses based on positive or negative cues from the real people she speaks with. She can "tell jokes, recite poetry, share ghost stories, relay song lyrics, and pronounce winning lottery number." [3]

Given how successful XiaoIce was in China, our group wanted to make use of a similar engaging personality in our application. This way, our users can feel much more welcome when they start to learn about rules or "spells" and how they can make their lives easier. However, while Sabrina adds a nice twist, her natural language processing is not yet as developed.

Other similar products that would be more familiar to American audiences are Siri (Apple), Cortana (Microsoft), and Google Now (Google). While each of these virtual assistants can only work on their respective companies' smart phones, they generally work towards the same goal: allow users to perform commands more conveniently. Rather than traversing the many subpages on his smart phone to find the right one for executing a command (e.g. set an alarm, find nearby restaurants, send a text, etc.), the user can simply tell his virtual assistant to do it for him. [4] [5] [6]

B. Rule Execution

As mentioned previously, IFTTT is a powerful automation service in which users can set up a trigger condition that executes some action. IFTTT describes rules as "recipes" that can combine the functions of multiple products (e.g. "If I post a picture on Instagram, save the photo to Dropbox"). While these are called "If recipes," they also support "Do recipes." Here, users can execute some action by tapping a button. (One can image this button tapping as the trigger for the pre-set action). [7]

Similar to IFTTT, Zapier also provides its users with a trigger-action based system, but specifically for web applications. One difference between these two services is that IFTTT focuses more on individual consumers, while Zapier targets small and medium sized businesses and enterprises. Another difference is that, while Zapier is concerned with automating web application execution, IFTTT has started to move towards the integration of hardware and the Internet of Things. [8]

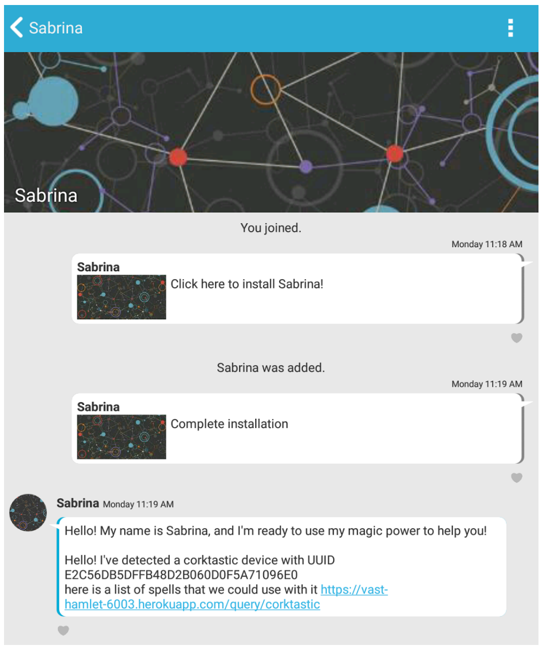
Like IFTTT and Zapier, Sabrina runs as a rule execution engine. However, we have also been working towards building Sabrina as a front-end personality that users can interact with through chat (i.e. natural language processing), which is a feature that systems like IFTTT and Zapier lack. In addition, because of her ability to initiate a conversation and communicate with the user, Sabrina can get to know the user over time and even suggest rules that reflect their preferences.

One crucial difference between Sabrina and the services mentioned above is that Sabrina supports discovering and managing nearby devices. Sabrina was designed with the Internet of Things in mind since device-to-device communication is becoming more and more important in our increasingly connected world.

C. Device Discovery

Tied with the Internet of Things, AllJoyn is a collaborative open-source software framework that allows physical devices and applications to discover and communicate with each other without having to use the cloud. It is currently able to support many language bindings (e.g. C, C++, Objective-C, and Java) and can be integrated into small and large platforms alike (e.g. RTOS, Arduino, Linux, Android, iOS, Windows, and Mac). Emphasizing the importance of flexibility, AllJoyn enables users to connect devices of differing brands, transports, categories, and operating systems. Abstracting away the intricacies of transports like Wi-Fi, Ethernet, Serial, and Power Line (PLC), it creates sessions between devices that can be managed with a user-friendly API. AllJoyn can also support multiple connection session topologies, like point-to-point and group sessions. Keeping in mind the importance of security, they also support different mechanisms and trust models like peer-to-peer encryption (AES128) and authentication (PSK, ECDSA). Embracing the open-source community, the AllSeen Alliance is currently working to define and implement more services and interfaces that deal with specific use cases, like onboarding a new device for the first time, controlling a device, and sending notifications. By doing this, developers can integrate these services into their own products and use them conveniently and smoothly with other devices and applications in the AllJoyn ecosystem. In addition to this, a device or application can also implement private interfaces. This means that, while an application can use common services and interfaces to take part in the larger AllJoyn ecosystem, it can also use the AllJoyn framework to communicate with specific devices privately. [9]

Seeking to promote the Internet of Things like the AllSeen Alliance, the Open Interconnect Consortium (OIC) is working to provide a secure and reliable approach for device discovery and connectivity across different platforms and operating systems. Building a broad industry consortium of companies that can help create this common, interoperable approach, OIC is defining the specification, certification, and branding of a connectivity framework that abstracts complexity. With its open-source implementation, the OIC is creating a "consistent implementation of identity, authentication, and security across the modes of User ID, Enterprise/ Industrial ID, and Credentials." It also seeks to enable various new modes of communication, like Peer-to-Peer, Mesh & Bridging, and Reporting & Control. Overall, the OIC is currently establishing a comprehensive communications framework that will



You joined.

Monday 11:18 AM

Sabrina

Click here to install Sabrina!

Sabrina was added.

Monday 11:19 AM

Sabrina

Complete installation

Sabrina Monday 11:19 AM

Hello! My name is Sabrina, and I'm ready to use my magic power to help you!

Hello! I've detected a corktastic device with UUID E2C56DB5DFFB48D2B060D0F5A71096E0 here is a list of spells that we could use with it https://vast-hamlet-6003.herokuapp.com/query/corktastic

Figure 1. Our application's user interface for device discovery. Once Sabrina discovers a local device that she can connect with, the user receives a prompt to use it with a set of recommended spells.

accommodate future applications in every key vertical market in the industry. While the OIC's open-source work is geared towards the developers of operating systems, applications, and platforms who want all of their products to run smoothly across different brands, the OIC firmly keeps in mind the individual consumers who want their IoT devices to conveniently communicate with their appliances and embedded devices. [10]

In addition, iBeacon is a similar technology that extends the capabilities of location services in iOS, allowing iOS devices to broadcast their presence to nearby agents, and to alert their applications when users approach or leave a location with an iBeacon. On top of monitoring a user's location, an application has the ability to estimate his proximity to the iBeacon using Bluetooth Low Energy signals instead of latitude and longitude coordinates. Developed for short-range control and monitoring applications, Bluetooth Low Energy (BLE) is a trade-off between energy consumption, latency, and throughput. [11] [12]

Currently, Sabrina only makes use of iBeacon in device discovery, but we plan on integrating AllJoyn and IOC in the future. By running at an abstraction level above the complexities of these technologies, Sabrina is built to be independent from the underlying device discovery/ management framework and transport.

IV. Design

As shown in figure [2] to the right, our design of Sabrina includes four main aspects. First, we have the user interaction component. This is grounded in our construction of the Sabrina persona as human-like assistant that speaks amicably to the user, making feel as comfortable as possible when navigating through the application. Second, Sabrina has a discovery component, which is responsible for collecting data about devices in the surrounding environment. It also takes care of running a recommendation system that suggests new rules for the user based on his past preferences and what these nearby devices are capable of. Third, we have our most important component pair: the rule execution system. This pair is tasked with running continuously in the background of the user's smart phone, collecting data from a set of input channels (i.e. triggers) and turning it into a set of output actions.

A. *User interaction*

As the name of our application implies, Sabrina was designed with the idea that the complications and intricacies of the system should be hidden away from the user. We follow the commonly-used principle in industry and in the field of human-computer interaction that users should not be required to spend time understanding the inner workings of a system, which they feel should "just work." [13]

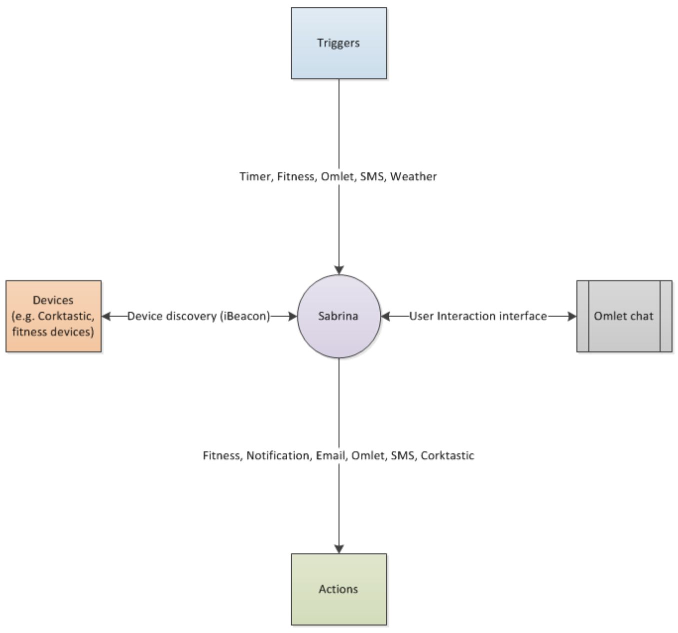

Triggers
Timer, Fitness, Omlet, SMS, Weather
Devices (e.g. Corktastic, fitness devices)
Device discovery (iBeacon)
Sabrina
User Interaction interface
Omlet chat
Fitness, Notification, Email, Omlet, SMS, Corktastic
Actions

Figure 2. Here we have the four components that make up our Sabrina application: user interaction, devices (and discovery), inputs/trigger, and outputs/actions.

However, any programmable system inherently faces a discoverability problem: some of the features of the system will probably not be known to the casual user, who will in turn not use the application as efficiently as its developers would hope. Our design tackles this problem by having the system periodically alert the user of potential new functionalities for the application. This takes the form of spontaneous, friendly offers of help from Sabrina, like when a nearby device is discovered and Sabrina tells the user what he can possibly do with it.

The result of building Sabrina with flexibility and user accessibility in mind is a two-part user interface. On one hand, we offer a powerful interface that allows users to leverage the full potential of the execution engine. This includes the construction of very detailed and specific rules that users who are more familiar with the system may want to create. On the other hand, we have simplified, automated interactions through the chat and the spontaneous prompts by the Sabrina persona. This promotes user retention by easing new users into the system and keeping them from feeling intimidated by the application's array of capabilities.

The user, when asked by Sabrina through chat, is able to decide among various programmed, suggested rules, which we describe as "spells" to keep with the theme of magic and technical abstraction. The user could also decide to build his own spell using the available "wands," also known as channels in the context of IFTTT.

This brings us to the crowd-sourced model for the deployment of the rules, as shown in figure [3] to the right. A relatively small set of contributing users generates rules that they feel would interest many other people. These rules are then shared to a common central repository, while each Sabrina instance is capable of discovering what rules in this repository would be useful to her owner in a given situation. The spells that Sabrina decides to suggest would be based on factors such as her awareness of the user's location, the time, the properties of the device that the application is running on, and nearby IoT device broadcasts.

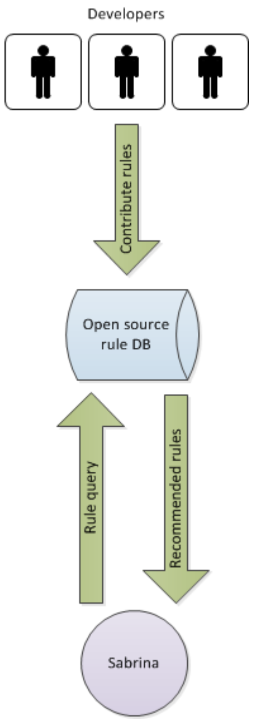
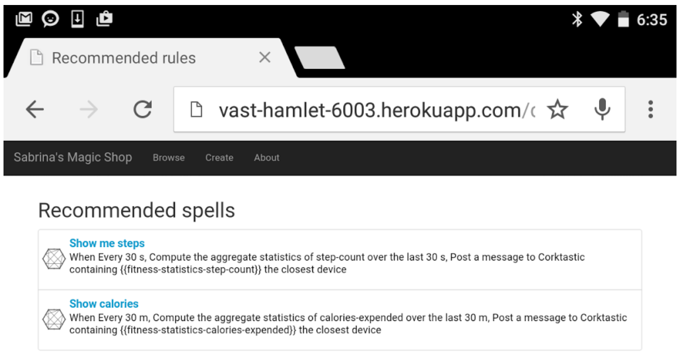


Figure 3. Users can provide rules to an open-source database, which each Sabrina instance can take from and recommend



Recommended spells

Show me steps
When Every 30 s, Compute the aggregate statistics of step-count over the last 30 s, Post a message to Corktastic containing {{fitness-statistics-step-count}} the closest device

Show calories
When Every 30 m, Compute the aggregate statistics of calories-expended over the last 30 m, Post a message to Corktastic containing {{fitness-statistics-calories-expended}} the closest device

Figure 4. Example rules that Sabrina may recommend for a user who has shown recent interests in fitness after she locates a nearby Android Wear watch.

B. *Execution engine*

In building our rule execution engine, we took inspiration from the existing model of "recipes" from IFTTT as described previously. However, we extended this model to allow for more flexibility while maintaining a limited number of channels and actions, given that we wanted to strike a balance between user friendliness and application capabilities. After all, the user will become exposed to the full potential of the engine only after getting accustomed to the system through spells that Sabrina makes readily available.

Our model thus formulates the following structure for a rule to be executed, in pseudo-BNF:

```
Rule := Trigger ActionList
Trigger := an arbitrary Boolean expression (and, or)
    of SimpleTrigger
SimpleTrigger := Channel Event ParamList
ActionList := Action | Action ActionList
Action := Channel Method ParamList
ParamList := Param | Param ParamList
Param := Name Value
Value := Literal | TriggerValue
```

In this structure, a channel (known as "wand" in the user interface) is a URI reference to a feature, application, or account in the system, while a method or event is a named reference to a specific trigger or action in the context of the channel. The channel abstraction provides namespacing to all features since different methods with the same name should be able to have different semantics. This also allows the application to have a shared point of state for all triggers and actions, across multiple invocations of a rule, which in particular can reduce memory pressure and the number of wakeups when multiple rules use the same channels.

Moreover, each event or method can be parameterized, such that the parameter value is either a literal value that the user had provided when creating the rule, or a meta-reference to another value generated while processing the rule. Parameters in the context of a trigger serve as query filters that limit the set of the events for which the trigger should be active. But in the context of a method, they serve as additional data passed to the method, thus reducing the number of different features that have to be implemented separately.

At any given time, we say a simple trigger is *firing* if the associated event code, using the current channel state, computes a Boolean value of True. As one can expect, a composite trigger is considered to be firing if the result of evaluating the Boolean expression on all simple triggers yield a value of True. When the trigger of a rule is firing, each action in its action list is executed in order, resulting in the following pseudo code:

```
1. Collect all sources of events for all triggers for all
   rules
2. While there is at least one incoming event e
   a. For each channel c
      i.  Update c given e
   b. For each installed and enabled rule r
      .   If r.trigger is firing
          1. For each action a in r.actions
             a. Execute a
   c. Clear e
```

In addition, actions and triggers are evaluated with respect to a context, which is a mutable set of name-value pairs created and destroyed around each iteration of loop (b). A trigger value parameter can be resolved by referencing a name in the context, and each event or method can update the context with new name-value pairs using the values produced by the previous event or method in order.

The advantage of this context system, in addition to allowing the application to use a value computed from the event into the action, is the ability to create "get" actions, that is, actions whose whole purpose is to compute a value (possibly using a shared channel state) and store it inside the context, so that a subsequent action in the same rule can output that value.

Unfortunately, using this design means that, within a composite trigger, we have the side effect that all component triggers have to be evaluated fully, even if the Boolean outcome had already been determined, in order to properly update the context. We deem this side effect to be acceptable because triggers should be short and idempotent, since they do not interact with the outside world and only use the state cached by the associated channel for the current event.

One may immediately notice that there is no conditional execution for actions: if a trigger is firing, then all associated actions will be executed unconditionally. When combined with "get" actions, this means that the produced value cannot be used to determine if subsequent actions are to be executed or not. Again, we consider this an acceptable tradeoff, similar to the lack of while loops or advanced control flow, because we want to keep the programmability aspect affordable to someone that is not used to algorithmic thinking, and, as detailed later in the evaluation section of this paper, we have not found a significant use case that is impaired by this limitation.

C. *Interaction between Sabrina and external devices/ services*

For our application, device discovery and the suggestion of available rules uses existing, well-known mechanisms from the IoT space, such as iBeacon: once a set of available objects is collected, they are classified into categories and a set of available rules from each category is retrieved from the shared repository.

Once discovered, Sabrina notifies the user of the availability of the device with an appropriate chat message. As a rule is installed as a result of a suggestion from a specific discovery operation, the communication details (e.g. IP address, protocol and port of the device, etc.) are implied and contextual, meaning that the user need not be aware of this information but may access it if he wants to.

In addition, Sabrina supports a set of so called placeholder object references that are mapped to different real object references at different times. They allow a rule to refer to the "closest Corktastic board" or the "currently paired Android Wear watch." Again, this mapping is transparent to the user as devices are discovered and remapped in the background. Rules can be installed even if the corresponding device is unavailable, and these placeholder object references become useful as the user moves around and the local environment changes.

V. IMPLEMENTATION

The system with its current design is naturally comprised of a user component, running continuously and providing the services of Sabrina, and a central server component (also known as Rulepedia), which provides hosting for the sharing of rules. In order to evaluate the feasibility of this system, we implemented the user aspect as a native mobile application using the Android 5.1 platform (API 21), and ran it on our phones and tablets, whereas we implemented the central server component as a node.js web server with a JSON file database.

It should be noted, however, that this is not a client-server design: the client does not need the server to function, nor it needs a stable internet connection, except for initially loading the rules and channels that the user can take advantage of. After all, the data for the application is stored client-side, and the execution of rules also happens locally. Nevertheless, due to implementation simplicity constraints, our client currently does not function unless the server is properly accessible when the application is booted up.

The client side is then further distinguished into two parts: a rule execution service, and a user interface/ device discovery service (interacting through the Omlet chat, installed separately). On top of this, to test our application sufficiently, we added a simple native Android UI that exposes most of the execution service features in their raw forms.

A. *Rule execution*

The rule execution service starts at boot and runs continuously in the background of the user's smart phone. It functions in a single-threaded fashion, essentially implementing the pseudo-code described above in the Design section of this paper in Android Java code. It exposes to the UI layer the ability to install, delete, and reload a rule.

We used JSON, a widely used and simple format with library code in both JavaScript and Java, to represent rules. We also have the ability to encode rules as URLs (and then further encode them as QR codes), which further simplifies the integration of our system with existing communication platforms. After all, these links can be copy-pasted and shared over various mechanisms as our application can intercept the opening of the link to decode it into a rule object.

In the execution service, we used a homegrown event system abstraction. This counteracts the fact that most event systems have the implied equality "one event = one handler," given that we want to be able to share a source of events across different instances of a trigger, or different triggers in the same channels. As mentioned previously, this was done to minimize the number of wakeups and the CPU overhead associated with processing an event, which is a necessity in a mobile device with limited battery life.

An event source in our system is an entity that is capable of listening to some outside interaction, waking up the execution thread, and producing a queue of some values, the type of which is specific to the event source. We include an event source for timeouts, for Android intents, for HTTP(S) polling, and for the Omlet messaging system. As we will describe further in the evaluation section of this paper, we observed that these basic primitives are powerful enough to integrate with most useful services.

Furthermore, each simple trigger (as previously defined) can be associated with a set of event sources, some of which are private to the trigger (allowing them to be parameterized) and some of which are shared in the context of a channel. When a channel is updated, each trigger looks at and caches the current event from the source, in order to compute the *firing* state and update the context when requested (unlike the abstract description above, updates are handled by the trigger, not by the channel, to avoid creating a caching state that will not be useful).

Event sources are started and collected by the execution engine altogether at boot. The service then goes to sleep until the event source notices some external event happening and causes a wakeup, at which point the service will update the channels and fire all rules appropriately.

B. *Extensibility and internals of channels*

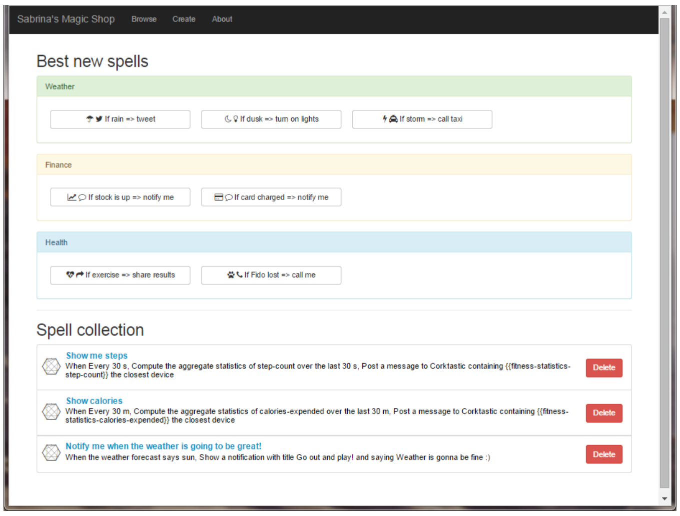
While we initially planned to offer a limited set of hardcoded channels, we quickly realized that this system would not scale. So, in addition to channels that are distributed as native code in the application package (which are useful to integrate with specific Android libraries and features like SMS), we also extended our design with the ability to download and install code for a channel on demand.

We use the same Rulepedia web server to host the channel metadata and code. This allows us to access that storage in the creation user interface, and to simplify an implementation in which we do not actually load the code on demand. Rather, we load everything upfront, but this is a limitation that can be easily lifted.

However, we did not want to download code arbitrarily from the Internet, even if it comes from a trusted central source. After all, this would clearly be a weakness from a security standpoint (even if this is not directly a vulnerability since the code is served over HTTPS and channels are vetted by the administrators). To cover for this, we designed a declarative description (also using JSON serialization) of a channel in terms of the available triggers and actions, so that the code associated would be able to run in a restricted sandbox with a limited attack surface. We chose JavaScript (using the Rhino JS library) as the language of choice, because of its limited standard library and because it is easy to run sandboxed. It thus stands as a better alternative to Java, whose sandbox attempts in applets are not as guaranteed to be secure. [14]

Inside the sandbox, code is essentially allowed no interaction with the outside world, except when it seeks to access arbitrary shared channel states. Triggers are also allowed to declaratively define a set of event sources they need to use (of the types described above) and to access their current values in the code. On the other hand, actions are allowed to return a declarative representation of the behavior they want to generate from a fixed set (HTTP calls, Omlet messages, emails, Android intents). This declarative mechanism is designed to keep external interaction to a minimum and to contrast obfuscated code, which would allow our application to obtain greater security through a system of semi-automatic review before it gives its approval to the main channel distribution system.

Future extensions could also include digital signing of channel code, as well as a certificate-based system for third parties, instead of relying on a central authority and TLS. However, the problem of safely distributing software is well-known and was not the focus of our work. One has to realize, though, that because a rule is limited in its power and its behavior cannot be obfuscated such that a malicious rule would be installed by an unsuspecting user, an extension channel has the same security concerns of a third party app.



Best new spells

Spell collection

Show me steps
When Every 30 s, Compute the aggregate statistics of step-count over the last 30 s, Post a message to Corktastic containing {{fitness-statistics-step-count}} the closest device

Show calories
When Every 30 m, Compute the aggregate statistics of calories-expended over the last 30 m, Post a message to Corktastic containing {{fitness-statistics-calories-expended}} the closest device

Notify me when the weather is going to be great!
When the weather forecast says sun, Show a notification with title Go out and play! and saying Weather is gonna be fine :)

Figure 5. Our current user interface for examining the spells stored in the central database. Users may submit admin-approved spells for the community to use. Each instance of Sabrina may in turn add any of these spells to the user's device and run them offline.

C. *User Interface Layer and Integration with a Chat System*

We chose Omlet as our chat application of choice because of our familiarity with its implementation, the wide range of integration points for third-party apps like Sabrina, and the ability to ask the developers directly for documentation. Additionally, we use Omlet as the messaging layer of choice for internal communication with certain devices, and we also expose the application to our rule engine.

Our implementation uses a specific chat room between the user and Sabrina, and it is configured the first time the application is opened (using the web hooks mechanism for posting to the chat, which unfortunately constrains us to require Internet connectivity).

The background service of our application would then periodically scan for new devices and announce their availability through the chat room, including in the message to the user the option to choose a rule for the device (refer back to figure [1]). Our implementation also keeps a stored map from placeholder identifiers to actual device addresses (encoded as URIs to get namespacing for free), and this map is used when resolving and executing a specific rule.

The only scanning protocol we implemented is iBeacon (with custom semantics for the major code in order to differentiate the various known device types), but we believe the system to be sufficiently generic that it will be possible to accommodate more discovery and communication protocols such as AllJoyn or IOC in the future, as previously mentioned. Indeed, iBeacon was chosen because it is widely supported by many platforms, and also because it was easy to turn into a testable prototype. However, one should note that it is fairly limited compared to other discovery mechanisms (there is no pairing, for example).

We did not, at this time, consider on the issue of onboarding (i.e. assigning a routable name to a device on a network). After all, we assume this would be handled by existing systems that will be integrated in the future (for example, AllJoyn has existing generic onboarding capabilities). The discovery protocol would also provide the application with all the information it would need to reach a local device over a supported communication channel.

D. *Server Side*

Our implementation of the server side of our application was kept to a minimum, essentially operating as simple storage for rules and extension channels. We also completed a powerful creation user interface in order to test the features of the system using JavaScript and Twitter Bootstrap. However, we did not focus on user interaction with this interface since we realized that it would only be used sporadically (the main aspects of Sabrina are the recommendation system and the chat UI). Therefore, we did not prioritize polishing this component, and creating a complex rule may seem somewhat clunky at first.

Nevertheless, we still kept in mind that the creation user interface is still an important part of the end user experience, albeit an advanced feature. Thus, in order to make the integration seamless between a UI in a desktop browser and a client running on a smart phone, we implemented a QR code system in which a newly-created rule will generate a QR code that another user can scan and have install on his own phone right away.

On the server side of our application, we also run the rule recommendation system, which receives a set of keywords from the client side to produce a set of suggested rules from the central repository. In our implementation, this is used directly as a mobile web page that Sabrina opens when she detects changes in the nearby environment, thus simplifying the source code and avoiding another layer of client-side processing.

VI. Evaluation and Findings

Because our team focused on completing the major features of our application, the quality of our code, and flexibility in terms of integrating other technologies in the future, we did not have sufficient time to conduct formal user studies. However, we casually pitched our idea and showed our demo video to several peers, and fortunately, most were intrigued by Sabrina's concept and utility. While we are confident that we have built Sabrina in a way that makes the application user-friendly and accessible, we are certain that having the opinions of outside testers would help us patch up any rough patches that we can improve. As stated previously, we are aware of the disorganized look of our rule creation user interface, but we hope to receive user feedback on how to best fix it. Aside from input from casual users, however, we also hope to gain feedback from the open-source community. By making our code accessible to the public, we hope that our integration of IFTTT and the Internet of Things can help further the state of the art.

As an integral part of our development process, we frequently conducted unit and system level testing, which helped us find and resolve a number of complicated implementation bugs. A rather difficult aspect of our project was testing interoperability with various devices and services such as Corktastic, Omlet, Android Wear (via LG smart watches), and Google Fit. In particular, we found it difficult to make our application flexible enough to accommodate other types of devices in the future.

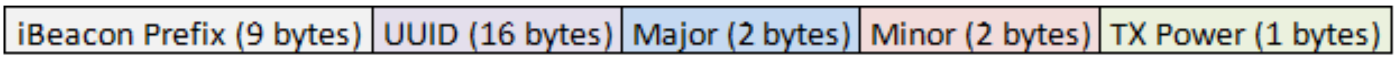
With regards to device discovery, we ran tests with iBeacon using device simulation. We used an iPhone app by Radius Networks to generate iBeacon broadcasts that would then be detected by a Sabrina instance running on a nearby Nexus tablet. [15]

At the bottom of the page, Figure [6] shows the format of the iBeacon advertisement protocol data unit (PDU). We can interpret its fields in the following way. We expect each device to advertise through the UUID field a unique identifier for the Omlet chat room which Sabrina can use to communicate with the device. We then use the major number to identify the class of device (e.g. 0 for Corktastic device, 1 for Android Wear). Given that support for a variety of IoT device frameworks is imperative to the long term success of Sabrina, we bring these technical details to a higher abstraction level to make our system agnostic. That said, we hope to support AllJoyn and IOC in the future.

Overall, our system successfully respects the privacy of our users. When a rule is installed by an instance of Sabrina, the user gives the rule access to the appropriate channels (e.g. Facebook, security camera, etc.) without having to upload any sensitive information to a centralized server. The rule execution system does not even need to connect to the Internet to run.

While we consistently kept in mind the important of user friendliness and accessibility, we found it difficult to implement a teenage persona that can make users feel welcome. For the sake of abstracting away technical complexities, it is easy to describe rules as magical spells that just work, and input/output channels as wands that Sabrina can use. However, adding a personalized touch for each individual user posed a problem. Right now, our rule recommendation system uses a set of keywords generated in the client side, but accurately learning complex preferences over time is an artificial intelligence problem that is not immediately solvable by our team. Though, given the success enjoyed by XiaoIce as described previously, we know that this feature is important in the long run.

Our team made a lot of progress in terms of making Sabrina flexible in many different aspects. First, the



iBeacon Prefix (9 bytes)	UUID (16 bytes)	Major (2 bytes)	Minor (2 bytes)	TX Power (1 bytes)

Figure 6. The format of the iBeacon advertisement protocol data unit (PDU)

ability to stack multiple triggers and actions within one spell enables users to have a great amount of freedom in creating a spell. In addition, the fact that Internet connection is required only when Sabrina wants to recommend rules for the user allows rule execution to be a pervasive offline service. Of course, one difficulty in promoting flexibility is keeping it from undermining user accessibility. For example, while stacking multiple triggers and actions is a powerful concept, our team must create an interface for this feature that feels unintimidating and organized. This is where user test data would help us significantly in striking this balance.

Finally, we integrated Sabrina with Deal, another project developed in the class, and provided them with an event-and-logic engine for their backend to determine bet outcomes. The Deal team was able to use our system to create rules for each bet and develop custom channels to support different objects of contention. Our experience with Deal reinforced our belief that Sabrina's rule model and API could be applicable to a variety of usage scenarios and intentions.

Overall, our team learned an incredibly great deal through this project. Some things that we have learned include but are not limited to Android development, the technical details behind the IFTTT model, the importance of user feedback and human-computer interaction, and the integration of devices within the IoT framework.

VII. Future Works

This paper described in detail our team's initial design and implementation of Sabrina. Based on our experience with her, we would like to improve Sabrina in several areas in the long run.

As previously mentioned, Sabrina only supports iBeacon for device discovery currently, but we would like to integrate and support other device management/ discovery frameworks such as AllJoyn and OIC. By keeping Sabrina accessible to the public as an open-source project, we hope to better facilitate future support for emerging Internet connectivity frameworks. By doing so, we promote the flexibility of our application as we give users a greater range of interoperable devices that they can connect with. In line with this goal, we would like to open Rulepedia (the central web server component of the Sabrina application) to the public so that we can leverage the power of crowdsourcing to include more channels and rules in the future. While this seems like a lofty goal, we hope Rulepedia can someday become the de facto standard for rule modeling and sharing.

We would also like to improve Sabrina's natural language processing abilities and her capacity to initiate communication with the user. We hope this will improve Sabrina's efficiency and user friendliness since this would allow her to better understand the user's personality and preferences over. Our team understands that this is a very difficult undertaking and that Sabrina may not be as sophisticated as XiaoIce or as powerful as Siri, but this is still an important factor for the user experience in the long term. Eventually, we would like to move beyond our current text-based chat interface and start using natural speech like other established products.

Another aspect of our application that we would like to continue building is our support for geolocation services. While we use iBeacon for detecting devices in the user's vicinity, Sabrina would be more powerful if she were not limited to local environments. Imagine the following scenario: a user wants to set up a rule in which the thermostat in his home turns on the heater only when he is at most 15 minutes away from his home (he may be coming home from work, or just taking a quick trip to the supermarket). Our current implementation cannot support long-range location services like this, but it is undoubtedly a powerful tool that can be proven useful for many users.

As discussed previously, we would like to implement a functional artificial intelligence component to Sabrina. Through this feature, she would be able to develop a constantly evolving understanding of the user's dynamic personality and preferences. Imagine another scenario: a user wants to set up a rule in which, if Sabrina detects that he has just turned on his car, she will change the radio station to one that she feels he will enjoy. What Sabrina believes may be linked with her analyses of other music-related spells that the user has recently installed. Because we seek to improve the user experience and make users' lives more convenient, it would be a worthwhile undertaking to implement an intelligent system like this.

Finally, to speak in more general terms, we would like to conduct more holistic evaluations of Sabrina to determine if our rule and channel models are sufficient for most user needs, and to better understand their limitations. While this includes personally thinking of ways to make the use of our application smoother, an even better way of learning how to improve the overall user experience is by gathering meaningful user data to get crucial feedback on our design and implementation.

VIII. Conclusion

Sabrina is a user-friendly and powerful application that combines the strengths of the IFTTT model and the Internet of Things to make users' lives more convenient while maintaining the integrity of their privacy. As users retain control of data that their rules use, the application can run these rules without having to rely on a centralized server. Designed with the user experience in mind, Sabrina seeks to smoothen the process of learning how to use our trigger-action based system and abstracts away any technical jargon. Using Rulepedia as central web server that stores sample rules that users can install and edit to fit their needs, Sabrina includes a recommendation system that we hope improve by strengthening her artificial intelligence component. In promotion of the Internet of Things, Sabrina also keeps a lookout for nearby interoperable devices that may be of interest to the user. With a rule execution system that emphasizes both flexibility and functionality, users can set up rules that can be as complex as they want. Overall, this project showed us many of the challenges of integrating different technologies in the industry, but we learned a great deal about mobile and social computing in the process.

IX. Acknowledgements

Our team would like to thank Michael Fischer for his help with integrating our system with Corktastic. We would also like to thank the Deal team for working with our system, testing it out, and helping us learn how to make it more flexible. We would also like to thank the Stanford Computer Science department for providing us with the Mobisocial Computing Lab (our main working space) and our budget for the Hackathons. We would also like to thank the Omlet team for providing us with relevant documentation, helping us better understand their API. Finally, we would like to thank Professor Monica Lam for giving us constant guidance throughout the quarter, making us think critically about our overall goals and the greater impact of our work.

X. References

[1] "About IFTTT," IFTTT, [Online]. Available: https://ifttt.com/wtf.

[2] "What is Internet of Things (IoT)?," WhatIs, [Online]. Available: http://whatis.techtarget.com/definition/Internet-of-Things.

[3] S. Weitz, "Meet XiaoIce, Cortana's Little Sister," Microsoft, [Online]. Available: https://blogs.bing.com/search/2014/09/05/meet-xiaoice-cortanas-little-sister/.

[4] "iOS - Siri," Apple, [Online]. Available: https://www.apple.com/ios/siri/.

[5] "Meet Cortana for Windows Phone," Microsoft, [Online]. Available: https://www.windowsphone.com/en-us/how-to/wp8/cortana/meet-cortana.

[6] "Google Now," Google, [Online]. Available: https://www.google.com/landing/now/.

[7] "Put the Internet to Work for You - IFTTT," IFTTT, [Online]. Available: https://ifttt.com/.

[8] "How It Works - Zapier," Zapier, [Online]. Available: https://zapier.com/how-it-works/.

[9] "Learn - AllSeen Alliance," The Linux Foundation, [Online]. Available: https://allseenalliance.org/developers/learn.

[10] "About Us | Open Interconnect Consortium," Open Interconnect Consortium, [Online]. Available: http://openinterconnect.org/about/.

[11] "iOS: Understanding iBeacon," Apple, [Online]. Available: https://support.apple.com/en-us/HT202880.

[12] "Overview and Evaluation of Bluetooth Low Energy," National Center for Biotechnology Information, [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3478807/.

[13] K. A. Butler, R. J. Jacob and B. E. John, "Introduction and Overview to Human-Computer Interaction," ACM SIGCHI, [Online]. Available: http://www.sigchi.org/chi95/proceedings/tutors/kb_bdy.htm.

[14] "Rhino Documentation | MDN," Mozilla Developer Network, [Online]. Available: https://developer.mozilla.org/en-US/docs/Rhino_documentation.

[15] "Locate Beacon Mobile App," Radius Networks, [Online]. Available: http://store.radiusnetworks.com/products/locate-ibeacon-app.