Function as a Service, an Ad Hoc Approach to Cloud Computing

Keith Downie[1]

[1]School of Computing, University of Utah

May 4, 2016

Abstract

Cloud computing has become an integral piece of many services and has had a major impact on how software is developed, both in the research and commercial fields. However, the traditional types of cloud computing are best used for applications that fit well into a client-server model. While this is an invaluable service, it is important to question this paradigm and to research uses for cloud computing that reach beyond traditional models.

In this paper, we define Function as a Service, a paradigm for cloud computing that more closely resembles a distributed systems model rather than the traditional client-server model. Under this paradigm, the cloud has a more intimate relationship with interpreted programming languages, allowing software systems to utilize computation from the cloud directly through an API. We define a model for cloud authentication that allows developers to either maintain control over how their software interacts with the cloud, or empower users to choose their cloud provider and themselves define cloud interactions without risk to the developer.

1 Introduction

Cloud Computing has become an integral part of many modern applications and is a major topic of research in both industry and academia. Of the various types of cloud computing, there are three main categories. The first of these, and likely the most well known, is Infrastructure as a Service. In this model, software distributors reserve remote machines from a cloud provider to run and maintain their software.

The second is Platform as a Service, a model that offers software creators an easy way to run their server-side code in the cloud. The developer simply supplies the cloud provider with the code that they wish to run. From there the provider handles everything from the running environment to scaling servers.

Finally, Software as a Service is a model where both the cloud servers and the software are provided by the same entity and the user then purchases access to the software that utilizes the cloud servers. Most often, this type of cloud used either as part of a thin-client web application or to synchronize user data, files, and other application data.

While each of these models differ in many aspects, they share some important traits. First, at their core these models all follow a client-server workflow, which revolves around creating long-standing proprietary servers for client applications to communicate with. This means that any software that utilizes the cloud must work within this client-server model. While this model works for many cases, it also limits the role that the cloud can play in relation to the user-end software. There are many clever ways to skirt around this limitation, but they are typically hacks and don't address the core issue.

The primary area where these models differ is what the developer is responsible for maintaining. For example, in PaaS the distributor is only responsible for maintaining the code that they give to the cloud

provider and the cloud provider charges them accordingly. In contrast, the cloud provider in the Software as a Service model is responsible for both the software and the cloud.

However, all of these models are similar in requiring the developer maintain this responsibility for the lifetime of their product. If a cloud-based application is utilizing a server in the cloud and the developer stops supporting that service, then the user-end application becomes useless if the cloud played any sort of meaningful role. This perpetual responsibility also discourages both smaller ventures and open source projects from using cloud computing.

Having the developer running the server that an application talks to also brings up some privacy concerns. Opening up the source code of a server is rarely done, so a user of that cloud service can't be sure what is being done with their private data. While most consumers of cloud-enabled applications understand this risk, it also means that users who need to work with secure data must avoid anything that utilizes the cloud.

We propose Function as a Service, which provides a cloud abstraction similar to a function call. This abstraction is used by the application itself, giving it the flexibility to choose whether to execute a function locally or in the cloud. By utilizing this API, an application is able to ship functions and state to the cloud to be executed remotely using a token-based authentication system. This model differentiates itself from the standard models in the following ways:

Function-like Workflow This model uses workflow that exists in between that of a platform and a service, as it is the software that decides what parts of its execution to offload. This means that the software itself that is in control over what gets processed in the cloud, and can make decisions based on its environment or hardware capabilities.

This level of control is interesting cases where preserving resources on the device is important, such as devices with low battery life or minimal processing power. This allows for the cloud to be utilized to alleviate these weaknesses without fundamentally changing the design characteristics of the software.

Choice in Provider Two different authentication models are defined, the first of which is most similar to existing cloud models. In this model, the developer sets the permissions for each user and is financially responsible for what resources are used. This allows for distributors to maintain control over how their software is being run in the cloud and what actions users can make.

The second model puts the control in the hands of the user, allowing them to define what cloud provider they use and what they use it for. This allows developers to make open source projects to make use of the cloud without being accountable for the actions of others. Instead, each user registers directly with a provider and is responsible for what they run.

Ad Hoc Communication Devices in this workflow can communicate though state stored in the cloud. This allows devices to collaborate while still running software specific to that device, promoting an "Internet of Things" approach to working with the cloud. This communication does not have to be previously established, which means this communication can be done on the fly.

2 Related Works

Of the three main categories of cloud computing, the model that is the most closely related on the surface is Platform as a Service (PaaS) [4]. This is because in the PaaS model, the cloud provider abstracts away nearly everything about how the servers run. To use a cloud like this, the developer gives the cloud provider the source code for their server and the provider handles running and maintaining it.

What makes this work different is that this interaction is done prior to the execution of the client-side code. PaaS, like the other models, establishes a server for clients to communicate with. This work focuses on defining the cloud interaction in band with the execution of the client application rather than communicating with a proprietary server.

A common way for machines to coordinate remote execution is through Remote Procedure Calls (RPC). In the RPC model, an application sends a request to another machine. This message is intended to invoke the second machine to perform some task. There have been many implementations of this protocol,

such as XML-RPC [5], which uses HTTP requests as the medium for invocation and returning data. What we propose here is different from RPC however.

In RPC, the remote machine runs a middleware that listens for RPC requests. Upon receiving a request from the first machine, this middleware launches a process on the remote machine to service the request. This is a key difference between RPC and the model we propose. This means that in order for RPC, and similar systems like Remote Method Invocation [2], to service a request, the services must already be implemented on the second machine. Like the differences with PaaS, this work differs because it focuses on executing arbitrary code that is not implemented on the other side.

There have been other efforts for developing and supporting the Internet of Things (IoT) concept. SmartThings [3] is a cloud-enabled framework for IoT devices. This framework uses the cloud and a SmartThings enabled router to abstract an event system that smart objects in the home subscribe to. This lets remote devices send event commands to objects inside the home. AllJoyn [1] is another framework that makes it easier for IoT devices to discover one another and communicate.

These frameworks differ than what is proposed here however, as they are primarily focused on abstracting the communication layer. The devices here must be subscribed to the framework and must be built to services requests that are given to it, which means that the execution of an event is still done on a given device. It also means that, in order to send command to another device, it must also know what methods the other device implements.

The communication in the model we propose is more indirect than this. When a device sends code to the cloud to be executed, it has the option to pick up sets of state that has been stored in the cloud. We consider this to be a benefit for IoT devices, as this allows an ad hoc communication without each device explicitly knowing about the implementation of others.

Research into sending the code that is to be processed remotely does exist in some form. In particular, a similar style of sending code to another entity is a part of research efforts such as active networking [6]. ANTS [7] is an architecture for achieving active networks by having network devices that are able execute remote code in order to realize new protocols.

While research into remote execution already exists in some form, to our knowledge the this is the first work to be done in expanding this to fit into a cloud environment where the application sends the arbitrary code to be executed. What we propose is a model for how Cloud Computing can be abstracted as a service directly to software, extending any application to be able to utilize cloud computing as a part of its own execution.

3 Function as a Service

In the traditional models of cloud computing, what the model abstracts is the details server side of an application. The level of this abstraction can vary considerably, even to the extent of the entire cloud being handled by the provider, such as in Software as a Service. However, all of these abstractions are still in the details of the server. A consequence of this is that nearly all cloud-enabled applications will fit into a client-server workflow.

What makes Function as a Service (FaaS) different from the traditional models of cloud computing is the reach of the abstraction. Instead of the cloud only abstracting the details of a server that a client makes calls to, in this model the abstraction is used directly by the client-side application. Through the use of an API, this enables the client to interact with the cloud in a way that is similar to a function call.

When calling a defined function in a typical programming language, at a high level there are two things to expect as the function caller. The first is that work will be done in another part of the program, perhaps accessing and changing global variables in the process. The second is that the execution of the program will return to the caller. Once returned, the caller gets a return value and can use this changed global.

What Function as a Service provides is a choice over where this called function is run. By extracting and packaging the code of the function that is being

called, the API can send the code to be executed in the cloud while still satisfying the expectations of the caller. When the cloud finishes executing the function, it sends the return value back to the API and the caller sees that it has a return value upon resuming.

In this model, the actions of the cloud are defined by what already exists in the client. This allows the application itself to decide when and how much to use the cloud, as the code that is being executed also exists locally on the client. This allows an application to be written in a model other than the standard client-server model while still taking advantage of cloud resources.

4 Authentication

This section details how the client can communicate with the cloud. In the FaaS model, the cloud does not consist of proprietary application servers as in other models. Instead, the cloud consists of general servers that take in arbitrary code and interprets it. This means that the communication protocol itself must be able to define who a user is and what they can do. As a part of achieving a system that strikes a balance between flexibility and customization for the user and control for the developer, we present two token-based authentication models as shown in Figure 1.

4.1 Public Tokens

This model is one that most closely resembles how responsibility in a cloud operates today. In this model, the software developer is the one who is billed for what executes in the cloud and typically the developer either takes payment from the user or monetizes in other ways. This model authentication is important for cases where the developer wishes to remain involved in how their software is run and allows for them to maintain more control.

Public tokens are a way for a developer to maintain control over what is run on their behalf. The distribution of these public tokens are done out of band with the cloud interaction. For example, when a user wishes to run their application, the developer can host a server that distributes these tokens to customers. Another option would be to make a permanent token that gets distributed with the application.

If the developer chooses to authorize a user, it then creates with a public token that contains encrypted identifying information that the cloud can then use to authenticate the user. After obtaining this public token, the client application can then make requests directly to the cloud server by including this public token for as long as the token is valid.

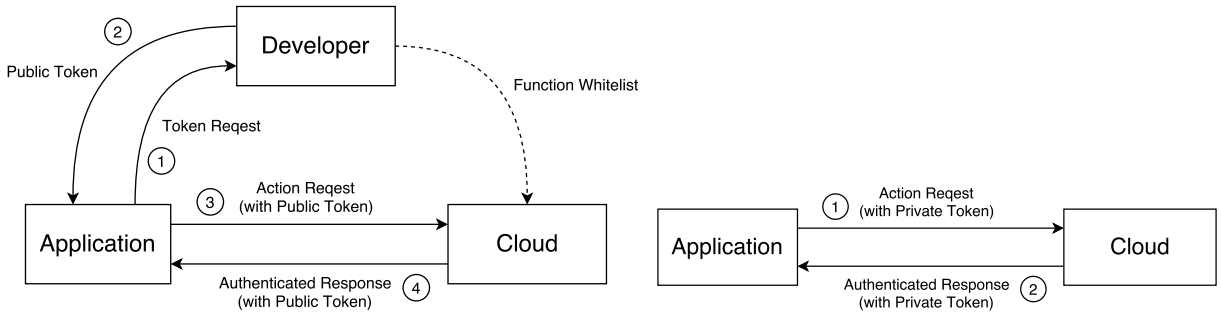
Code White Lists

When a developer gives a user a public token, they are authorizing that user to execute code on the cloud at their expense without any knowledge of what each request might contain. As this could potentially be abused by an attacker to execute arbitrary code at the developer's expense, it is important to allow the developer to limit what code is being executed on the cloud under their token.

Each developer is given their own private token in which to communicate with the cloud. With this token they can register a list of namespaces and a white list of code hashes for each ID. These white lists specify what code is allowed to run on the cloud for that particular namespace. When request comes in using a public token from that developer, the code that the user is attempting to run is then hashed and compared against the white list that is registered for that particular namespace. If the hash of the code does not match, then the code is not executed and the client is given an error message.

4.1.1 Permissions

As they are accountable for what resources a user under a public token consumes, a developer is able to further limit how users can interact with the cloud. Permissions are an extensible way to define what restrictions a user has. For example, these restrictions include how often a user can run functions, if they are able to schedule recurring functions, and what memory state they are allowed to access. More about these permissions are discussed in further sections.



(a) Public Token

(b) Private Token

Figure 1: Token Authentication

4.1.2 Token Security

As part of the goal of this model is to provide cloud access to low hardware devices, this must also be taken into consideration in the security model. As things like key pair encryption can be expensive for impoverish devices, this model instead uses a token checksum in order to protect interactions from abuse.

For each public token that is issued by a provider, a checksum for that token is created that must also be included in the message to the cloud. This checksum is simply a SHA1 hash of the entire token, including the identities and permissions for the token. This allows the cloud, upon receiving a request, to ensure that the permissions of the token have not been tampered with.

4.2 Private Tokens

In contrast to a public token, private tokens impose no artificial restrictions on the holder. When a user registers with the cloud provider that given their own private token. This private token can then be used with any application that they wish and are free to execute what they want in the cloud. In this case, the user is billed directly based on what cloud resources they use, which means that the developer is not responsible for what the user runs.

As there is no longer a third party involved, the requirements for a private token system is much more simplified. For this a unique private token is issued to the user with a unique identifier. This token can then be used by any application the user wishes across any software platform within the limits of the cloud provider. Anyone with a private token can create public tokens under it, as was described in the previous section.

This type of authentication in particular provides some interesting properties for the cloud. For example, it makes it more realistic for open source project to utilize the cloud, as each user can have their own cloud interaction. Since each person is paying for what they use, users can make any modifications they wish to the software while posing no risk to the developers. These benefits and more are explored via prototypes in a later section.

5 Anatomy of a Message

Communication with the cloud is done through the concept of message passing. Here, an API is invoked in the software which creates a message for the user to communicate with the cloud. Upon receiving a message, the cloud will then perform actions based on this request. For some of these messages, the cloud will then create a response message that is sent back to the user. Following is the structure of a request message. The actual fields that a user will include in a message will vary based on what the user is requesting.

Function Code

There are two options for providing what function to execute on the server. The first is to provide the byte code for the function in the message. In this case, the cloud can be instructed to either execute the function and return the result or simply store the function in the cloud for later use. When storing the function, the byte code is then hashed by the server and stored relative to the user making to the request.

A second option is available when a function has already been hashed and stored on the cloud. In this case, the device can instead provide the hash of the function instead of the byte code for the function itself. When the cloud receives this hash, it then matches and loads the correct function to execute.

Parameters and Global Variables

When a function's byte code is sent to the cloud, the function itself keeps the same anatomy that it had on the originating machine. This means that in order to call the function, the user is also responsible for ensuring that the correct number of input parameters of the correct type are also sent along. If these are not provided then the cloud send a return message back to the user containing an error and the function will not be executed. When the cloud prepares for executing the function, it matches the parameter names that the function is expecting to the pair with the same name in the parameters set and passes those into the function when started.

In a similar vein, during execution global variables that are referenced inside of the function will also be called. The user is also responsible for ensuring that these global variable are also passed to the cloud. When the cloud creates the isolated container to execute the function in, it can load these global variables into the container that any global references inside of the function will reference.

In both cases, the user provides a separate set of key-value pairs. Each of these key-value pairs represents a single variable or parameter, where the key is the name of the variable and the value is the intended value of that variable. These sets are provided each time the user wants to execute a function and only live for the current execution. An obvious limitation to this is that the function is only able to use information that already exists on a single device. To address this, we introduce the concept of namespaces.

Memory Namespace

A memory namespaces is a mechanism that offers a parity to storing global variables on the cloud. These namespaces allow users to store state on the cloud and access it from any function using that namespace, which allows devices to reduce the amount that is has to store and allows this state to be shared between multiple devices without additional communication. Similar to global variables, a namespace simply consists of a set of key-value pairs.

In the request message the user provides the name of the namespace, given as a string. To avoid conflicts, only one namespace can be defined per message. The user can also provide a set of key-value pairs as well as what action they want to cloud to perform on this set. In the default case, the pairs in this set are added to the namespace and are available to any future executions. If the request is also performing the execution of a function then there are two additional options.

The first of these options is to add the given join the given pairs and the requested namespace into a new structure that lives only for the given execution, which allows for references to global variables in the function to be defined without adding everything to the namespace. The second option is to take what the function returns and store it in the namespace.

Creating a namespace can be done by anyone with a private token and are unique within that token. When creating public tokens, access to a namespace must be explicitly given in the token before a call is able to access it. This allows applications to communicate within the same namespace, but provides a way for the holder of the private token to limit what applications can modify what namespaces.

Token

Included in the message is the public or private token discussed in the previous section. Providing a valid

token is required to be able to communicate with the cloud, as it ties the message to a user. No matter the type of token, it is added to the message in the same way. Once a message is received, the cloud parses the token to determine the type and check whether it is valid.

In the case of a public token, this will also (optionally) include the permissions of the user. This includes how often the public token can be used and what namespaces the token is allowed to access. The checksum for the token must also be provided or the cloud will not service the message. This is to ensure that the cloud is not servicing tokens that have been tampered with.

6 Cloud Prototype

Implementing the entire cloud infrastructure for this paradigm is beyond the scope of this project. As such, for the prototype the cloud provider is treated as a black box, where the user sends a message and receives a response from the cloud. The plumbing for the prototype is simply a server that takes in messages and returns what the protocol specifies. In this section we work through the details of how this server implements this protocol.

Function Introspection

Python 2.7 was chosen as the language for this prototype in order to take advantage of the introspection features that it provides. Specifically, Python makes it easy to get the byte code of a function. Byte code in Python is what is generated when the interpreter converts the text source code into something that it can run. What this enables us to do is send the byte code to another location and use it to create a new function. While there are multiple ways to accomplish this in Python, each with their own advantages, the simplest way is to use the __code__ structure that is automatically added to each function definition. Below is an example of retrieving this byte code and using it to create a new function.

```
import new

def foo(b, c):
    return b+c+z

byte code = foo.__code__
bar = new.function(byte code, {z: 3})

bar(1,2)
```

Using the "new" library to recreate the function also puts the function in a container. In the function bar, the only global variables that are available are those passed in as the second parameter. When bar is called, z=3 will resolve to what it sees as a global variable, even though z was never defined in the main program. This also means that any dependencies that the cloud wishes to support must pass these in as well. The original foo function remains unaffected and if run will result in an exception since it does not see z as defined.

In addition, Python also makes available a list of variables that the function makes reference to. This can be used alongside the globals() command to have the API automatically pull the required global variables into the message as apposed to the manual approach shown above. As the two are simply a Python dictionary, a mix of the two approaches can also be used and the data structures will be joined into a single dictionary.

API and Packaging messages

Messages in the prototype take the form of a Python dictionary. Below is an example of what a message sent with a public token looks like.

```
{'action': 'load',
 'execute': 1,
 'hash': # SHA1 function hash
 'namespace': 'app1',
 'token': {'private_id': 0x1234,
           'public_id': 0x5678,
           'permissions': {'rate_limit': 5,
                           'namespaces': ['app1']
                            }
          },
 'checksum': # Encrypted checksum
}
```

In order to send these messages, the JSON library is used to pack and unpack these messages. With this library, json.loads(msg) with turn the message

into a string that can be send through a socket, and json.reads(msg) will take that string and recreate the dictionary on the other side. This is used for both requests from the user and responses from the cloud. Below is an example of how this is done from the perspective of the application.

```
import cloudapi as api

def bar(a,b):
    return a+b+z;

if runLocal:  # Run bar locally
    z = 3
    ret = bar(1,2)
else:  # Run bar in the cloud
    ret = api.ex(bar, {z: 3}, token, ...)
```

This example shows how the application invokes the API of the cloud. It also shows that, since the function bar is defined locally, the application has a choice as to whether to execute the function locally or send it to the cloud. When the API is called, by default the following actions are taken by the API and the cloud.

1. API checks if the function was recently hashed or if the user provided the hash. If not, it makes a SHA1 hash of the function.

2. API builds a message from the hash and application-defined parameters and sends this message to the cloud.

3. Cloud checks permissions of the token and checks if it has the function that the hash matches to cached.

   If the function is cached, the cloud executes the function with the parameters in the message and sends back the return value. The API then sends this back to the application.

   If the function not cached, the cloud returns an error saying this to the API.

4. If the API gets a message that the function was not cached, it sends a new message using the byte code for the function.

5. When the cloud receives the byte code for a function, it caches the function and its hash and then executes the function, sending the return value back to the API who returns it to the application.

White lists and Namespaces

In this prototype, namespaces are stored in a dictionary of dictionaries, which can be up to three levels deep. At the first level, the keys in the dictionary are the *private_id* of the token. The second level keys are the namespaces that have been set up under that private token. The third level are the key-value pairs that reside in that namespace for functions to use.

In addition to the values of a namespace, there is another key labeled *whitelist*. This contains the information that the owner of the private key has set up and is not writable for any public keys. This contains a list of function hashes that users with access to that namespace are allowed to run. A user is only able to use one namespace per message, so these white lists can not be mixed together to perform unwanted actions in a namespace.

When a user executes a function with a namespace declared, the values of that namespace are joined together with the supported libraries as well as any global variables the user may have passed in. This joined set is then passed into contained function which accesses them as if they were global variables.

7 Application Prototypes

In this section we detail the prototype applications that were made to demonstrate how this cloud model can be used to achieve the properties discussed earlier in the paper.

7.1 Smart Home Thermostat

The motivation for this prototype is to show that the ad-hoc nature of this cloud can have important properties for the "Internet of Things" mindset. Specifically, this prototype shows that this cloud can be used to enable smart home devices such as a "smart" thermostat. This is a device that has very minimal hardware and a small computational throughput, which

makes it something that could benefit from the ability to utilize cloud computing.

While the hardware of a typical thermostat is sufficient to monitor the local temperature and maintain a static schedule, it likely wouldn't suffice for anything more advanced. For example, incorporating weather forecasts or using machine learning to adjust to the behavior and preferences of the user would likely be beyond the ability of the hardware. This is where cloud computing can help, by allowing the thermostat to offload computation and state aggregation to the cloud.

This thermostat prototype consists of three parts: the driver that controls the furnace and wireless communication, a temperature reader, and a set of control firmware. Inside of this firmware is the logic to decide whether to turn the furnace on or off. Once the firmware was completed, a "compiler" was ran that saved the hashes of those functions to a file and registered those functions in the cloud. Both the compiled files and the source code are included on the thermostat. Both online and offline modes are included.

Offline Mode

If there a lack of a connection to the cloud, the thermostat still needs to be able to operate. In this prototype, the thermostat has the entirety of the source code. This means that it is able to import the firmware and can execute all of the required functionality locally using its own state. Below is an snippet of the thermostat running in offline mode.

```
import firmware
import temperature as temp

ID = firmware.registerDevice()

while 1:
    currentTemp = temp.read()
    firmware.updateTemp(ID, currentTemp)
    decision = firmware.makeDecision()
    turnFurnace(decision)

    time.sleep()
```

While the workflow may look odd, this is because the firmware was written to work when run both locally and in the cloud. Here, the thermostat gives the firmware the latest reading it took with *updateTemp* and then calls *makeDecision*. The firmware will then evaluate the temperatures and determine if the furnace needs to be turned on, returning with a "yes" or a "no". If "yes" is returned, the driver will turn the furnace on until a "no" is given.

Online Mode

Once the thermostat has an internet connection and is able to connect to the cloud, it no longer needs to run things locally. Instead of importing the firmware directly, it can instead send the functions to be executed on the cloud. Since the thermostat has limited resources, the driver can load the saved hashes from file instead of hashing the functions itself. Below is a snippet of the driver using the cloud API instead of the firmware to make a decision.

```
import cloudapi as api
import temperature as temp

api.setToken = getToken('public.token')
api.addNamespace(ns)

registerDevice = getHash('rD.hash')
updateTemp = getHash('uT.hash')
makeDecision = getHash('mD.hash')

ID = api.ex(registerDevice)

while 1:
    currentTemp = temp.read()
    api.ex(updateTemp, currentTemp)
    decision = api.ex(makeDecision)
    turnFurnace(decision)

    time.sleep()
```

In online mode, the thermostat also has the option to utilize additional information other than just the one that it reads. In addition to the thermostat, this prototype also includes individual temperature readers. There readers represent small devices that are placed in other rooms in the house. The entire functionality of these readers is to read the current temperature and report it to the cloud using the same namespace. When the firmware running in the cloud is making a decision, it will see the current temperature for all of these devices and tell the thermostat to turn on or off based on the minimum of these values.

In this model, when the user buys a thermostat

or a reader it comes with a unique public token that allows that device to communicate with the server. The information that they upload is unified by using the same namespace, such as the SSID of the network they are connected to. To prevent users from running unauthorized code on their token, the developers set up a white list of function hashes that their public tokens are allowed to run. Each time the firmware is updated they add the new hashes to the white list, which enables the new firmware to run.

Now, say that the company that makes the thermostat goes out of business and they remove their services from the cloud. Traditionally this would mean that the thermostat would now only function in offline mode, as there would be no server to communicate with. In this paradigm however, the user can themselves register with a cloud provider. After getting their own private key, uploading it to the devices, and registering the functions, the thermostat would still be able to function online despite the official token no longer being serviced.

Since this gives the user financial responsibility for the resources they use, they are also in charge of what functions can be run in the cloud. This means that, were the firmware open source, the user can upload custom firmware to the devices. Say that a user wants to instead have the temperature decided by the average of the last five readings rather than the most recent reading. After they update the firmware, all they would need to do is run the compiling program to generate new hashes and upload them to the devices and the cloud. Now, when the devices load the hashes from file they will be telling the cloud to run the new functions.

7.2 Mobile Feed Reader

To help motivate this prototype, we list several realistic goals that an application might have during development. Imagine that there is a developer that wishes to make an application with the following properties:

- The user can set what feeds they wish to follow, and the results from all of these feeds are shown.
- They are only interested in looking for posts with specific words in the title. The user can provide a set of search terms. All results that match the search terms should be displayed since the last time the user viewed the feed.
- The application is meant to be run on a mobile phone, so it should spend a minimal amount of time processing feed changes in order to save on battery life.
- The developer isn't sure if they want to keep maintaining the application, so they want to make this application open source so users can make modifications.
- The developer wants to limit their financial burden, limiting a user's resources usage to what they think they can get through things like ad revenue.
- The developer understands that some users are concerned about their privacy, so an informed user should be able to know what is being done with their data and can take steps to protect it.

A user could run an application locally and be able to satisfy the functional requirements for this app. In order to store the feed results in between viewing the application would have to poll the feeds every so often, match the results against the search terms, and add them to the pending posts. However, having to frequently poll the feeds to maintain this application would demand a non-trivial battery life.

Another option is to have the feed gathering part of the application run in a traditional cloud, where the cloud keeps state on what pending posts match the search terms since the last sync. While this would work functionally, it does not satisfy several of the developer's needs.

The developer would have to maintain a server on the cloud to service any active applications. If they decide that they don't want to maintain the software and take the server down, the application will cease functioning. Having to maintain a server would also disincentivize making the application open source, as any changes in functionality that a user would make

could only be on the client side without the developer approving changes and updating the server. Making the code open source also increases the attack vector on the server should someone attempt to exploit it with unauthorized code.

Prototype

The focus of this prototype is to show how all of these properties can be achieved when using this cloud paradigm. The application here is a web page that the user visits to view the feeds that they have subscribed to. This page was created with HTML and jQuery in order to show that, even though no Python is being run on the page, communication is still possible.

When the web page first loads it does not have any information about the Reddit feeds that the user has visited. To find this out, it needs to send a message to the cloud to read the namespace. To support these interactions, a separate Python script is created. In this script are all of the functions that the web page will need to use. When run, this script will take the byte code of those functions, hash them, and spit them out to a file. Then, when the page wants to communicate with the cloud, it loads these hashes. The snippet below shows how the *getUpdates* function is compiled. This simple function grabs the current subscriptions and pending post updates that currently reside on the cloud.

```
def getUpdates():
    return pendingPosts

f = open('getUpdates.hash', 'wb')
m = hashlib.sha1()
m.update(marshal.dumps(getUpdates.__code__))
f.write(m.hexdigest())
```

Since the web page is not running Python, it does not have access to the API. It can, however, build a message itself since it has the hash for the function it wants to execute and the cloud sends and receives messages in JSON, the native data structure of Javascript. To avoid having to use web sockets in Javascript, the cloud was updated to supports HTTP headers when the web page sends the message through a GET request. This is all that is required of the web page to display the posts that have been curated by the cloud, and the user can resend this message any time they wish to get the latest content.

When the user first visits the page they need to register the post gathering function in the cloud. This function is hashed in the same way that the *getUpdates* function is. To set or change what feeds they are subscribed to or what search terms they follow, the web page first stores the user set values in the user's namespace. When the web page wants to clear the posts that the cloud has gathered, it sets the pending variable in the cloud to an empty set. The function to get the feed uses the requests library to make a HTTP call to reddit.com. A snippet of this function is below.

```
def getPosts():
  subs = {}
  for sub in subscriptions:
    url = 'reddit.com/r/'+sub+'/.json'
    page = requests.get(url)
    pageJSON = page.json()
    if len(filters) > 0:
      # Filter the data
      posts = filtered
    else:
      posts = page.json()
    subs[sub] = posts

    return json.dumps(subs)
```

This function will grab the current posts from each of the subscriptions and put the JSON for those posts into a dictionary. Since this function will only run once, it needs to utilize the scheduling functionality of the cloud. Below is the message that the web page sends to schedule this function to repeat.

```
{'action': 'load',
 'execute': 1,
 'repeat': 30,
 'write_to': 'pending',
 'namespace': 'ns',
 'hash': # SHA1 function hash
 'token':{'private_id': 0x1234,
          'public_id': 0x5678,
          'permissions':{'repeat': 30,
                         'repeat_limit': 1,
                         'namespaces': ['ns']
                         }
          },
 'checksum': # Encrypted checksum
}
```

This request tells the cloud that it wants the func-

tion to execute when the message is received and to repeat every 30 seconds after. Since the web page will not be listening for return messages after the first execution, the *write_to* parameter instructs the cloud that, when the function is repeated, the return of the function should be stored in the *pending* variable in the *ns* namespace.

In the permissions object, the developer has set that most frequent that a function can repeat is every 30 seconds and only one function can be scheduled, which allows them to limit the amount of resources that is used. If the web page were modified to lower the requested repeat time below what the permission state, then the cloud will send back an error and the function will not be scheduled. As state previously, the checksum over the token prevents users from changing the permissions, as the cloud recalculates this checksum and compares them before performing the request.

8 Discussion and Future Work

This prototype largely focuses on the communication between devices and the cloud. As such there are some issues that would arise in creating a full cloud environment that were not addressed in this prototype. In this section, we highlight a few of the more glaring issues and discuss how these issues could be mitigated.

Token Security

An integral piece to any network-based system is the issue of security. The primary security mechanism in the token exchange model is the signed hash of the message so that the cloud can confirm that the message was not tampered with. While this prevents users from modifying their public token or from a man in the middle attack, there are other security concerns that should be considered. One of the primary issues with the current prototype is the lack of end-to-end encryption.

This could be addressed with the addition of end-to-end encryption, such as using a public/private key pair. In this case, the user would register their public key with the cloud prior to sending any messages. When creating a message, the user would provide the API with a private key pair to encrypt the message with. Once the cloud receives a message it would then decrypt the message using the public key.

While this would work, it has the potential to run contrary to one of the primary goals of the system. This goal is to enable low powered devices to communicate with the cloud, which means that the performance of the encryption used will play a factor in what performance can be achieved on the device. In general, using key pair encryption is a heavy task that is non-trivial to perform on sufficiently weak hardware.

A middle ground for this issue would be to give users a choice as to what messages to encrypt. An example would be if a user is concerned about others snooping on the byte code or namespace updates that they are sending to the cloud, but do not want to encrypt every message they send. In this case, the user would register functions and update the namespace with encrypted messages. After the functions are already registered, they can switch to using only function hashes to call functions in the unencrypted messages that follow.

State Synchronization

In this prototype, the cloud was running on a single server. As such, it was easy to prevent two devices using the same namespace from causing irregularities. The server knows what calls a function will be making beforehand and in what order, so the server can simply check for conflicts to decide whether to run two functions concurrently.

However, in a full cloud environment this issue becomes much less clear. An integral part of the cloud is the concept of distributing work across many machines, which raises the question of how the state of a namespace can be shared in a way that resolves namespace conflicts. A conflict in the namespace happens when a function is beginning its execution just after the namespace has been updated. If the change in the namespace is not synchronized quickly enough, the function will execute with a stale value and could erroneously overwrite values in the names-

pace.

One way to address this issue is for each resource to keep a time line of what functions were executed since the namespace was last synced with the rest of the cloud. Specifically, this time line would need to keep track of only what functions modified the namespace. These time lines are compared when a synchronization occurs, and if actions occurred out of order then the local namespace state is discarded and the functions are run again with the updated namespace.

9 Conclusion

We believe that this model of cloud computing shows a contrast to the standard model of cloud computing, allowing for a new interaction between devices and the cloud. It enables ad-hoc approaches like cloud hopping, optional local execution, and creating a shared state for multiple devices on request. It also for the user to easily control their cloud interactions, enabling cloud-based open source projects and allows for software to operate beyond the attention of the developer.

While the work presented in this paper is not focused on evaluating the performance of a cloud of this type, it does show that the key concepts of FaaS can be done using an interpreted language like Python, and that it allows for interactions that would be difficult using standard cloud computing models. It also shows two distinct applications operating under this workflow as opposed to a traditional cloud and shows that there are some clear advantages in doing so.

References

[1] AllJoyn Framework. `https://allseenalliance.org/framework`.

[2] Introduction to Java Remote Method Invocation. `http://www-itec.uni-klu.ac.at/~harald/ds2001/rmi/rmi.html`.

[3] SmartThings. `https://www.smartthings.com/`.

[4] Understanding the Cloud Computing Stack: SaaS, PaaS, IaaS. `https://support.rackspace.com/white-paper/understanding-the-cloud-computing-stack-saas-paas-iaas/`.

[5] Merrick, P., Allen, S., and Lapp, J. Xml remote procedure call (xml-rpc), Apr. 11 2006. US Patent 7,028,312.

[6] Tennenhouse, D., and Wetherall, D. Towards an active network architecture. *Computer Communication Review 26*, 2 (Apr. 1996).

[7] Wetherall, D., Guttag, J., and Tennenhouse, D. Ants: A toolkit for building and dynamically deploying network protocols. *IEEE Open Arch '98* (Apr. 1998).