Uncovering Twilio: Insights into Cloud Communication Services

Ramnatthan Alagappan
ra@cs.wisc.edu

Sourav Das
souravd@cs.wisc.edu

1 Abstract

Cloud communication service (CCS) with its simplicity and lower investment cost is becoming increasingly popular. In contrast to its growing popularity, very little is known about the internals of CCS with respect to its architecture and protocols. To gain insights into CCS, we study a popular cloud communication service "*Twilio*" using gray box techniques. In our study, we provide insights into the Twilio ecosystem, its components, the interaction among components and the protocols. We also measure some guarantees provided by Twilio and show how the measurements fair against what is promised. Our analysis un-veils a number of interesting aspects about the Twilio ecosystem and have strong implications for developers who want to build applications on top of *Twilio* APIs.

2 Introduction

CCS is an upcoming service model which provides sophisticated APIs for enterprises to develop applications and offload communication related tasks from enterprise applications. All the communication services are offered through simple REST APIs for the applications to make use of them. Cloud communication services have a lot of advantages compared to *on-premise hosted* communication infrastructure. Firstly, it offloads the burden of communication from the applications and separates it as a separate service. The applications can seamlessly interact with the communication APIs to accomplish complex communication tasks like sending promotional messages to customers, providing critical real-time information to mobile phones, etc. Secondly, the development effort involved in integrating an application with CCS is very less compared to developing and maintaining a home-brewed communication infrastructure. Thirdly, enterprises can build communication applications in a cost effective way because of the pay-per-use model provided by most the CC services.

Our work focuses on study of such cloud communication services. There are a lot of players in the market which provides CC services like Avaya, Clickatell, Plivo, etc. and we chose a popular CCS *Twilio* for our study purpose. *Twilio* is one of the prominent players in the CCS space and has a huge customer base which includes popular enterprises like *Coca-Cola*, *WalmartLabs*, *Intuit*, *Box*. These enterprises use *Twilio* to accomplish a wide variety of tasks ranging from two-step authentication, powering lending machines with music, secure file sharing, delivering deals and ads to mobile phones, etc. *Twilio* enables lot of new scenarios for businesses that were rather cumbersome to implement in the past. The APIs are simple and intuitive to understand and develop. *Twilio* also provides rich documentation and code examples to build simple applications like VoIP communication.

For our study, we developed a simple VoIP service atop Twilio APIs which we call as VoT (VoIP on Twilio). Coupled with logs collected at several places in the system we present a detail study of Twilio which we believe is somewhat representative of CC service in general and is first of its kind. We give insights into the Twilio ecosystem, the high level as well as packet level protocol details and measurement of some guarantees that Twilio provides. We also discuss some interesting oddities that we found during the course of our study and point to some possible enhancements in the system.

The reminder of the paper is organized as follows. In section 3, we motivate our study. In section 4, we provide a detailed analysis of the *Twilio* ecosystem, architecture of the system, our experimental setup, some of the scenarios that Twilio supports, high level protocols and packet level analysis for some key scenarios. We give detailed measurements with respect to call and message dequeuing rates in section 5. Then, we present some oddities that we found in the ecosystem in section 6. In section 7, we discuss some of the possible enhancements to the system. Finally we conclude by presenting the implications of our study and our future work.

3 Motivation and Related Work

Businesses want to delegate communication from their applications and services because of the advantages provided by CCS. The number of enterprises using CCS has seen a steady increase since its advent. Though there are not enough evidences that this pattern is going to continue, but because cloud communication services provide an attractive cost-model and easy-to-use APIs, we strongly believe that this trend is going to continue in the future. As more and more enterprises start using CC services like *Twilio*, there is a good chance that this traffic may contribute to a good fraction of Internet traffic in the future.

There have been lot of studies on VoIP services like *Skype* in the past [4]. There have been recent studies

on cloud storage services like *Dropbox* [3]. Best to our knowledge, we are the first to study cloud communication services. [6] provides a thorough characterization of the Dropbox protocol and traffic patterns. They provide insights into how traffic to *Dropbox* varies across four different networks including home and campus networks. Our study on the other hand does not deal with traffic analysis since we did not have the sophistication of collecting the packet traces on the campus network. Our study aims at studying the architecture and protocols of the entire *Twilio* ecosystem. Driven by our study, we also aim at exploring some possible enhancements to the entire *Twilio* ecosystem. [4] provides a detailed packet level study of the *Skype* protocol. The authors also provide deep insights into the *Skype* architecture. Our study involves studying the architecture of the system and the protocols involved using a suite of gray box tools that we have developed.

4 Twilio Overview

In what follows, we describe the Twilio Ecosystem, some of the possible scenarios using the Twilio APIs, the experimental setup that we used for our study, high level protocol study and then packet level protocol study.

4.1 Twilio Ecosystem

The Twilio ecosystem, as depicted in the Figure 1 can be viewed as a layered architecture. In the bottom most layer lie the $Twilio\ servers$. These servers lay the foundation of this ecosystem by exposing a set of data and voice communication APIs for sending and receiving voice calls and messages.

In the middle layer, lie the $Application\ servers$. These servers are installed by Twilio customers with dedicated Twilio accounts. For e.g. These application servers might belong to some company X that wants to provide VoIP service to its customer. Each Twilio account is linked to 1) one or more Twilio numbers, 2) an Account SID and 3) an Auth Token. A Twilio number is a ten digit phone number. Account SID is an unique identifier for a Twilio account. Auth Token is a token used by Twilio Servers to authenticate an account. The Application server uses the Account SID and Auth Token to access the Twilio APIs. The cost model for this second layer Application servers (or Twilio customers) is typically *pay per message* or *pay per call*.

In the top most layer, lie the $Clients$ which could be browser based clients or phone based clients. An important thing to note here is that these Clients are *not* the direct customers of Twilio, instead they are the customers of the services that are built on top of Twilio. For e.g. These clients could be the customers of company X that is providing VoIP service. These clients, can be free customers or paid customers of company X. Also, depending on the type of service that the layer two based Twilio

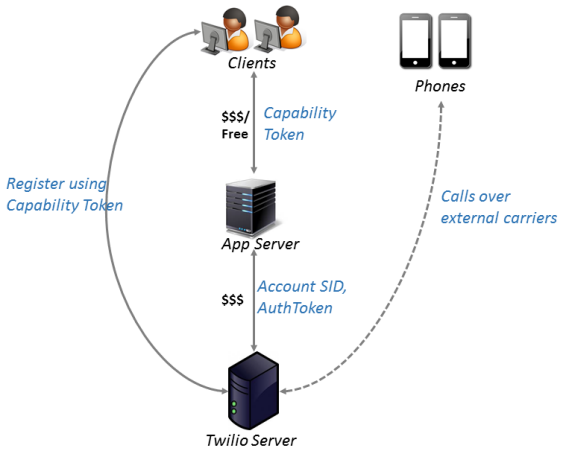

Figure 1: Twilio Ecosystem

```
<Response>
<Say voice = "alice">
Hello World
</Say>
</Response>
```

Figure 2: Sample TwiML Snippet

customers wants to provide, it is possible that these layer three is non-existent. The Clients when present, in order to use the services and communicate with each other need to register with the layer three Twilio servers. This is done using a *capability token* which is provided by the Application Server.

So far we have only discussed how the Twilio clients connect to each other in the Twilio ecosystem. The Twilio clients and the Application servers can also interact with the external phones belonging to different carriers via the Twilio servers. Hence, these external phones also form a part of the Twilio ecosystem.

Application Container: Each Twilio number is linked to an application container. The application container contains two URLs: $VoiceURL$ and $MessageURL$. These URLs are configurable and are usually configured by the Application server administrators (or Twilio customers). Whenever an incoming call or message for a Twilio number arrives, the Twilio server makes post request to these URLs. The content generated by these URLs direct Twilio servers to perform the needed actions on the incoming calls or messages. These contents are in form of a special markup language called $TwiML$ described in the subsequent section. Additionally, Twilio appends the *query parameters* that it obtains from the end-clients before making the requests to the application servers.

TwiML: It is a markup language developed by Twilio.

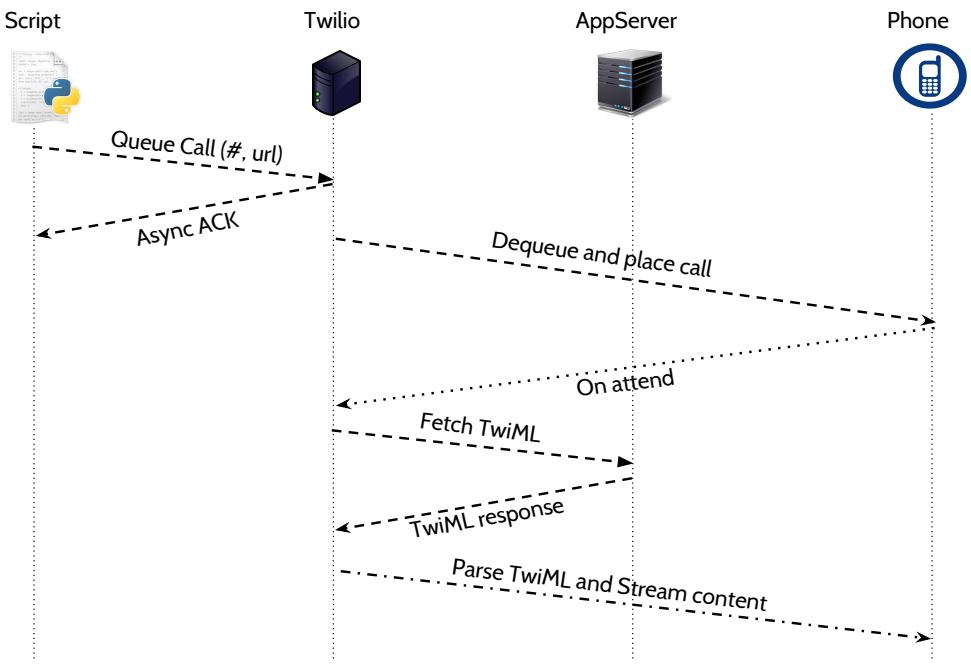

Figure 3: Automated call to a phone

It contains a set of simple verbs that can be used by the Application Servers to direct the Twilio servers about the action that needs to be taken whenever a call or message is received to its number. Various kinds of verbs are supported as shown below which can be used to create interactive applications atop Twilio.

- Say - Read text to the caller
- Play - Play an audio file for the caller
- Dial - Add another party to the call
- Record - Record the caller's voice
- Gather - Collect digits the caller types on their keypad
- Sms - Send an SMS message during a phone call
- Hangup - Hang up the call
- Queue - Add the caller to a queue of callers.
- Redirect - Redirect call flow to a different TwiML document
- Pause - Wait before executing more instructions
- Reject - Decline an incoming call without being billed

For e.g. The TwiML snippet shown in Figure 2 will say Hello World (dictated by the verb "say") to the caller in female voice (dictated by the attribute "voice"). Our experience with TwiML suggests that it is very simple to use and powerful with respect to the diversity of verbs that it supports.

4.2 Scenarios

There are multiple scenarios that one can enable with the help of Twilio Apis. Some of them are:

- $Automated\ calls$: Twilio APIs can be used to place automated calls to phone numbers to deliver a pre-recorded message.
- $Voice\ calls$: VoIP applications can be built atop Twilio voice APIs which can be used to place voice calls. There are three scenarios possible for VoIP applications:
  - Call between VoIP Clients
  - Call from Phone to VoIP Client
  - Call from VoIP Client to Phone
- $Messages$: Twilio message APIs can be used to send automated messages.

Voice calls between VoIP clients:

4.3 Experimental Setup

For our study, we developed a simple VoIP service (called VoT) atop Twilio and deployed it on OpenShift RedHat Cloud. The VoT server also hosts a simple web interface which can be used to place voice calls by specifying a phone number or the registered names of VoIP clients. For making automated calls and sending automated messages

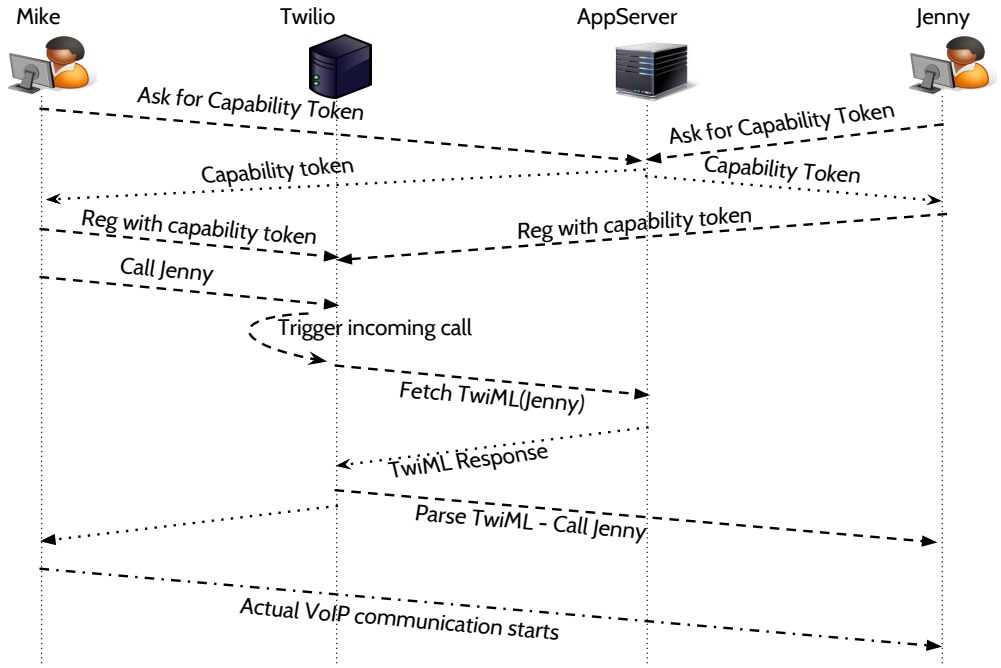

Figure 4: Call between two VoIP clients

we have developed python scripts using the Twilio client libraries which directly calls into the Twilio APIs. For our study, we extensively use traces collected at different places. This includes application level trace messages collected at our VoT server, logs in Twilio user portal, Twilio object store queries and *Wireshark* traces collected at client machines.

4.4 High Level Protocol Study

We now present a high level protocol study of some of the important scenarios that Twilio supports.

Automated calls to phones: The protocol diagram for this scenario is shown in Figure 3. At first, a script (our python script) queues a call request specifying the phone number to which the call needs to be placed and an url indicating the location of the TwiML response. The Twilio server acknowledges the request and places the call. Once the phone attends the call, the Twilio server does a *HTTP POST* or *HTTP GET* to the url in the call queue request to fetch the TwiML response. In our experiments, we hosted these pre-recorded messages in our application servicer. The application server processes the request and sends a TwiML response in return. The Twilio server parses the TwiML response and finally streams the content to the phone.

Voice calls from phone to VoIP clients: The protocol diagram for this scenario is shown in Figure 4. Let us say, Mike and Jenny are the two VoIP clients and Mike wishes to call Jenny. As shown in Section 4.1, for Mike and Jenny to communicate they need to first register with the Twilio Server. Hence, Mike and Jenny at first request the Application server for capability tokens. Once the Application server delivers the tokens, Mike and Jenny register with the Twilio server using the tokens. Mike then queues a call to Jenny. An interesting thing to note here is that the Twilio server does not have information about what to do with the call. It need to communicate with the Application server to get this information. For this, it triggers an incoming call to the Twilio number to which Mike is linked to. As stated in Section 4.1, every incoming call to a Twilio number generates a post request on the voice URL present in the application container linked to that number. This post request is directed to the Application server with query parameter as "Jenny". The Application server parses the request and generates the appropriate TwiML content indicating the action that needs to be taken. In this case, the application server would serve a response that contains a *Dial* verb and the client for the *Dial* verb would be "Jenny". The Twilio Server parses the TwiML and places a call to Jenny. Finally, Mike is notified about this and the voice communication starts. Note that the actual voice communication is not peer-to-peer and is routed via Twilio servers.

Voice calls from VoIP clients to phone: The protocol diagram for this scenario is not shown for space constraints and is similar to that shown in Figure 4 except that Jenny has a dedicated phone number and hence does not need to register with the Twilio server. Lets say, Mike is a VoIP client and Mike wishes to call Jenny with a dedicated phone number (may belong to any carrier). Mike will at

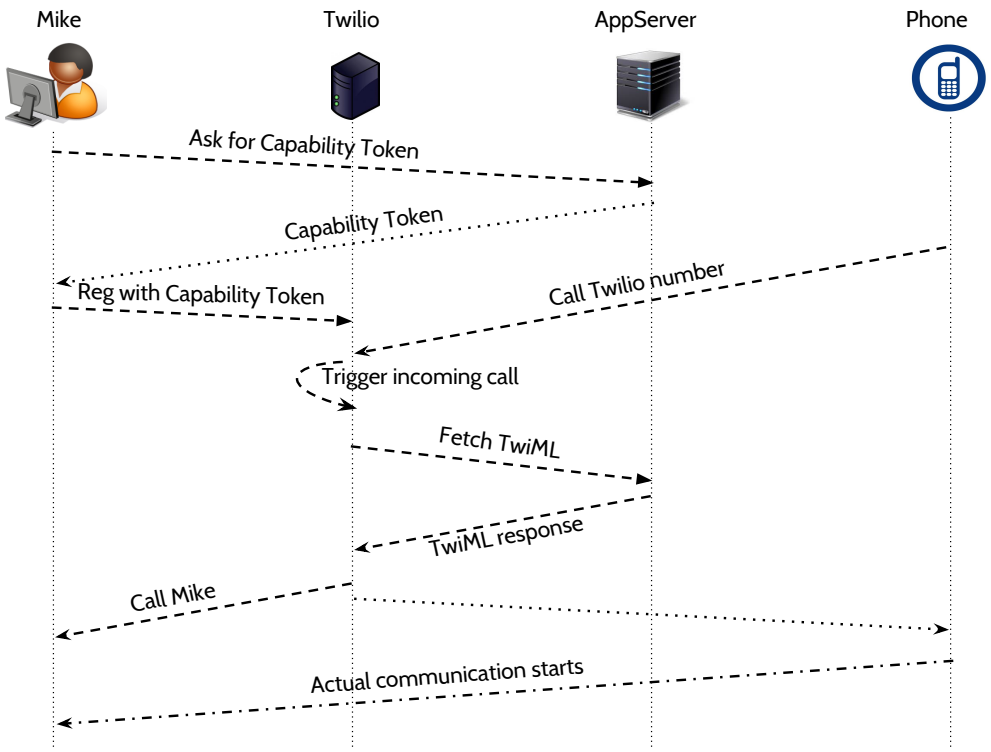

Figure 5: Call from phone to a VoIP client

first request the Application server for a capability token. Once the Application server delivers the token, Mike will register with the Twilio server. Mike then places a call to Jenny's number. As in the case of two VoIP clients in the previous section, the Twilio server communicates with the Application server to get the information about the action that needs to be taken for the queued call. For this, it triggers an incoming call to the Twilio number linked with Mike. This incoming call generates a post request on the voice URL present in the application container. This post request is directed to the Application server with query parameter as "Jenny's phone number". The Application server parses the request, generates the appropriate TwiML content for the action that needs to be taken and delivers it to the Twilio server. The Twilio Server parses the TwiML and places a call to Jenny's phone. Finally, Mike is notified about this and the actual communication starts.

The protocol diagram for this scenario is shown in Figure 5. Lets say, Mike is a VoIP client and a phone wish to call Mike. Mike will first request the Application server for capability token. Once the Application server delivers the token, Mike registers with the Twilio server. When the phone calls the Twilio number associated with Mike, the Twilio server triggers an incoming call to that Twilio number. The incoming call generates a post request on the voice URL present in the application container. This post request is directed to the Application server. The Application server parses the request, generates the appropriate TwiML content for the action that needs to be taken and delivers it to the Twilio server. The Twilio Server parses the TwiML and places a call to Mike. Finally, the phone is notified about this and the actual communication starts.

4.5 Packet Level Analysis

In this subsection, we dig deeper and present the packet level analysis for a scenario where a browser makes a voice call to a phone or another browser based client.

Figure 6 shows the sequence of events that happens when a browser based client wants to make a VoIP call to another browser based client or a phone. As shown, at first the client browser does a *HTTP GET* request to the application server that we have deployed. The application server generates a *capability token* and passes it onto the client. The client side javascript includes the *twilio js* library and does *Twilio.Device.Setup* to register its capability with the *Twilio* servers. The sequence of messages for the second block is also shown. The client passes the *token* as *HTTP* data after establishing a TLS session with the server. After this step, the client is free to make a call to either another browser based client or another phone number. The actual voice communication is tunneled through *Twilio* servers and it involves *RTMP* protocol. This sequence of packet exchanges are not shown due to space constraints.

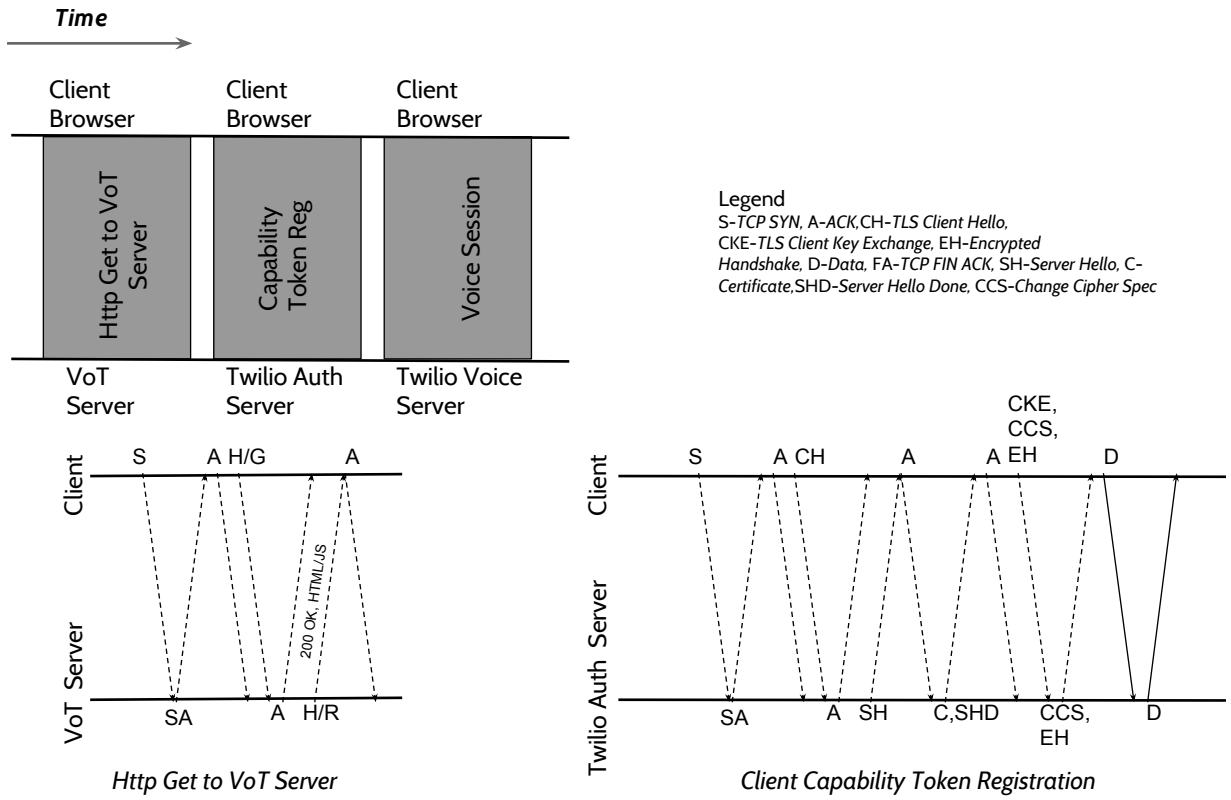

Figure 6: **Voice Call from Browser Client** *The figure shows sequence of packet exchanges that happen between the client and other components of the system. The top diagram shows the high level operations. The participants are shown above and below the top and bottom lines. The left bottom figure shows how the client interacts with the application server to obtain the capability token and it corresponds to the first block in the top diagram. The right bottom diagram shows the sequence of messages exchanged between the client and the Twilio server when the client executes the Twilio.Device.Setup method and corresponds to the second block in the top diagram.*

5 Measurements

In this section, we provide measurements with respect to call and message dequeuing rates. *Twilio* provides guarantees that the placed calls and messages will be dequeued at a rate of 1 per sec and placed onto the phones or the browser clients [1]. We measure the degree to which this guarantee is met. All the experiments were conducted on Lenovo W530 laptop with 8 GB RAM running on 4 cores connected to a 30 Mbps Internet link.

5.1 Calls

We developed scripts that can queue automated calls to US phones and online browser clients. We queue the first call to the browser client and queue a variable number of calls to a US phone and then finally place one more call to the browser client. *Twilio* dequeues the calls in the order it was placed into the queue. Using *Wireshark* in the browser client, we collect the packet traces and the timestamps of the incoming call connections. We then can calculate the time difference between the first call and the second call received by the browser. A careful reader would note that the call timestamps may not reflect the actual time when the call was dequeued from the queue and it will include the network latency involved in placing the call to the browser client. We noticed that this latency was lesser than 50 ms and so was discounted for the purpose of our calculations.

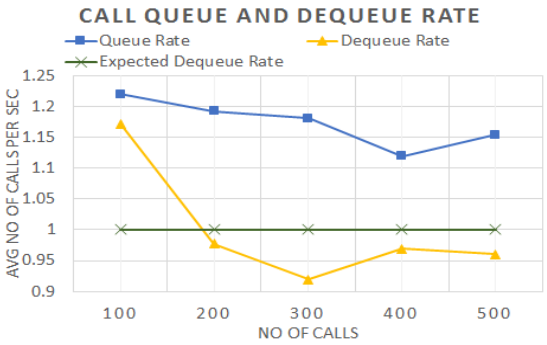

Figure 7: **Calls queuing and dequeuing.** *The figure shows the queuing and dequeuing rate for calls*

Figure 7 shows the expected dequeue rate, observed dequeue rate and queue rate for calls. The horizontal axis shows increasing number of calls that we place to the phone and the vertical axis shows the queue or dequeue rate. As mentioned before *Twilio* gives a guarantee to de-

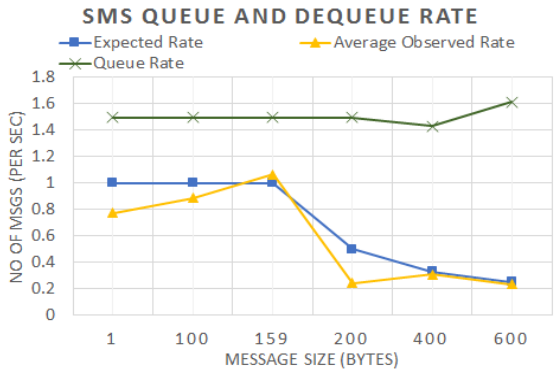

Figure 8: **SMS queuing and dequeuing.** *The figure shows the queuing and dequeuing rate for messages*

queue calls at the rate of 1 per second. This is shown by the flat line. We define the queue rate as the rate at which the client library can place calls into the *Twilio* call queue. It is quite possible to achieve higher values of queue rate with high speed links. This value does not necessarily imply that the *Twilio* server has a throttling mechanism for adding calls into the queue. The *Dequeue rate* line shows the observed dequeue rate in our experiments. We notice that for a small number of calls such as 100, *Twilio* dequeuing mechanism performs really well than expected. As the number of calls increases, the dequeue rate drops slightly below 1 but stays between 0.9 and 0.95. We believe that this behavior is completely acceptable if further increasing the number of calls to say few thousands still does not reduce the dequeue rate.

Summary.Twilio call dequeuing guarantees are met and exceeded for small number of calls like 100. The dequeue rate slightly falls below 1 as the number of calls increases from 100. If applications need very critical and real time data to be delivered to phones or clients, we believe it is advisable to use one Twilio number for say around 100 outgoing client/phone connections. For applications that do not need this level of delivery accuracy, we believe this behavior is completely acceptable.

5.2 Messages

We developed scripts that can queue automated messages to US phones. We queue 100 messages with varying message sizes to a sinlge US phone number. *Twilio* dequeues the messages in FIFO order similar to calls. We noticed that telephone networks take a very variable amount of time to deliver messages to phones. So, for calculating the dequeuing rate, we used the timestamps present in the *message resource* in the *Twilio* object store. We query these message resources using the REST APIs and arrive at the dequeue rate by looking at the *sent timestamp* in the message resource.

Figure 8 shows the expected dequeue rate, observed dequeue rate and queue rate for SMS. The horizontal axis shows increasing size of the messages that we place to the phone and the vertical axis shows the queue or dequeue rate. As mentioned before *Twilio* gives a guarantee to dequeue SMS at the rate of 1 per second. As mentioned earlier in 5.1, the queue rate shows the rate at which the client can push messages into the queue. It is interesting to note that the expected rate itself drops as the message size increases. *Twilio* treats a single message as just 160 bytes. If the message contains say 170 bytes, *Twilio* considers this as two messages and so it can queue at the rate of one message per two seconds and so the expected rate falls to 0.5 from 1. Similarly for a 400 byte message, there are three such chunks and so the expected rate drops to 0.33. It can be noted that the observed rate closely follows the expected rate.

Summary.Twilio message dequeuing guarantees are well maintained. An interesting observation is that naive developers who do not notice the size limits of the messages may wrongly believe that *Twilio* can dequeue at a rate of 1 message per second irrespective of the message size. We bring this out clearly in our study by showing that the expected rate itself falls down as the message size increases.

6 Oddities

We now present some of the oddities in the *Twilio* ecosystem that we discovered during our study. First, as discussed in section 4, *Twilio* charges twice for making a single outgoing call. We discovered this from the call logs available in the Twilio user portal. On first seeing this, we thought this was a problem in the VoIP service code that we developed. But it turns out that even after using the code examples from Twilio developer forums, we were still observing this behavior. It should be noted that we built our VoIP service on top of a single *Twilio number*. Second, we noticed that some applications may require *ordered* message delivery for the messages that they try to send through *Twilio* APIs. For example, a message based query system will require that the messages sent are delivered to the mobile phone in the same order. We understand that the telephone service provider can also reorder messages when delivering them. We understand that *Twilio* does not provide any guarantee as such for ordered message delivery. But we wanted to measure to what extent the messages can be reordered.

Figure 9 shows the message re-orderings that can happen. It is understandable that *Twilio* does not itself provide ordered message delivery. Application developers who develop on the *Twilio* platform should be aware of this and should use suitable application level techniques to solve this issue. For example, applications can use a label denoting the position of the message in the sequence.

Third, we injected few faults into some parts of the system and observed how *Twilio* reacts to these corner cases. For example we tested cases where in the *TwiML*

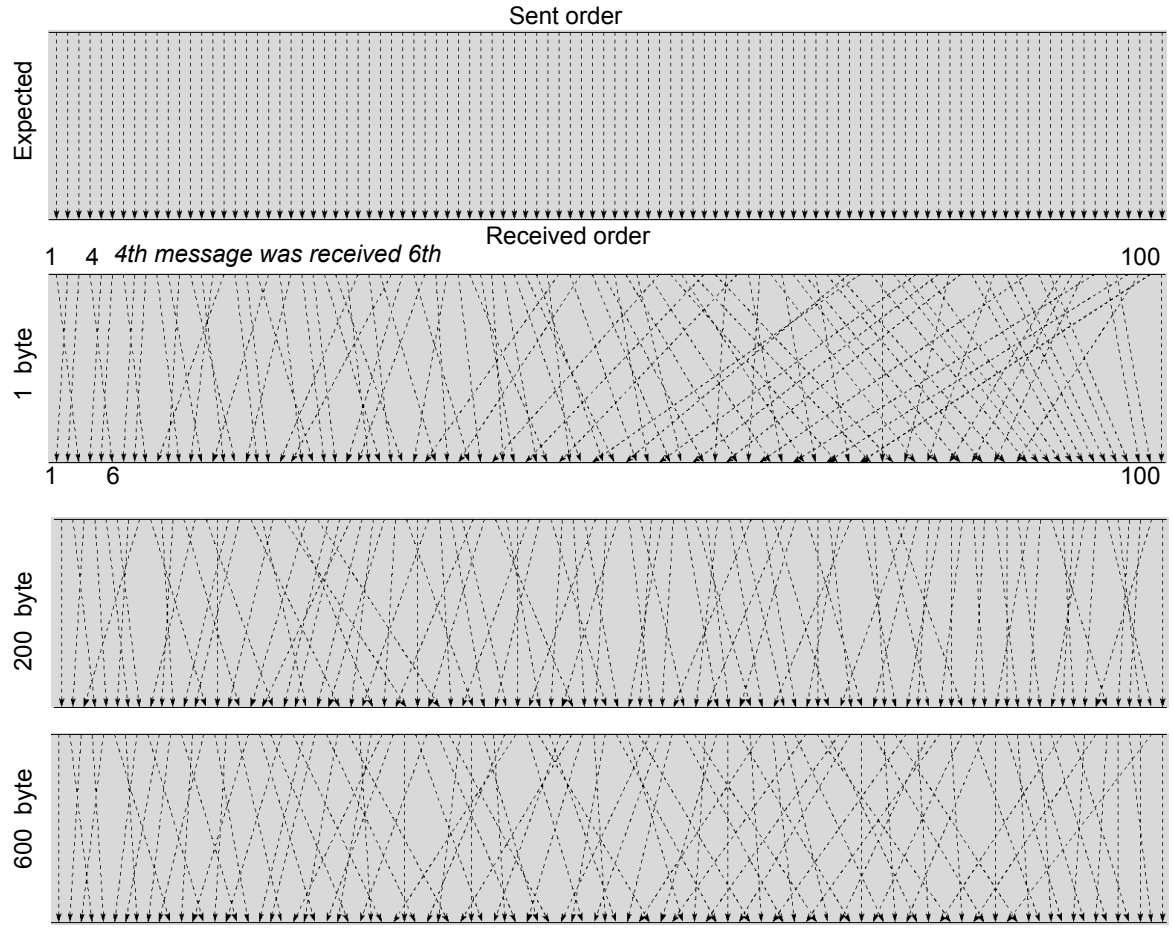

Figure 9: **SMS reordering** *The figure shows the extent to which messages can get reordered. The top line in each strip denotes the sent order and the bottom line denotes the received order. The top most strip shows the ideal message delivery behavior. The second strip shows the re-orderings that happened when 100 one byte messages were sent. The third strip shows the same for 200 byte messages and the last strip for 600 byte messages. Note that the sent time is identified from the sent timestamp from the resource store so it represents the time for nearest upstream telephone carrier network to acknowledge the message was received and will be sent to the phone.*

response returned by our VoIP server was malformed, unacceptably large, corrupted etc. We noted that the fault handling mechanisms were not uniform across all error scenarios. For example, if the *TwiML Say verb* contained content greater than 4096 bytes, the call was placed but an error message was delivered to the attendee. On a different scenario when there are nested *Say* verbs in the *TwiML* response, then a blank call was placed to the attendee. We are barely scratching the surface in terms of fault injection. We strongly believe that much better extensive techniques can be developed to analyze all the possible error conditions in the service.

Fourth, during our study we observed a weird behavior in the *Twilio Python client v3.66* library. We observed that when a client script tried to queue a call or a message using the *create()* API, there was a HTTP level auth failure that was happening before the actual sequence of packets that were happening between the client and the *Twilio* server. On further investigation and debugging, we found that the client was not passing the *Auth Token* and *Account SID* when the first REST API call is made to the server. Then the exception is handled and then the required credentials are passed onto the server for the subsequent REST API calls. We noted that this unwanted auth failure wastes 6 RTTs. We also believe that this can be easily optimized in the client library.

Figure 10 explains this client library behavior.

7 Discussion

We now discuss some possible enhancements to the entire *Twilio* ecosystem.

Client Library Improvements: We showed that the client library can be optimized to save few RTTs in the previous section. It is important to notice that we identified this just by manually studying the behavior of the client library. We believe that there should be more robust and extensive ways to study the client's interaction with the server. We also showed in the previous section

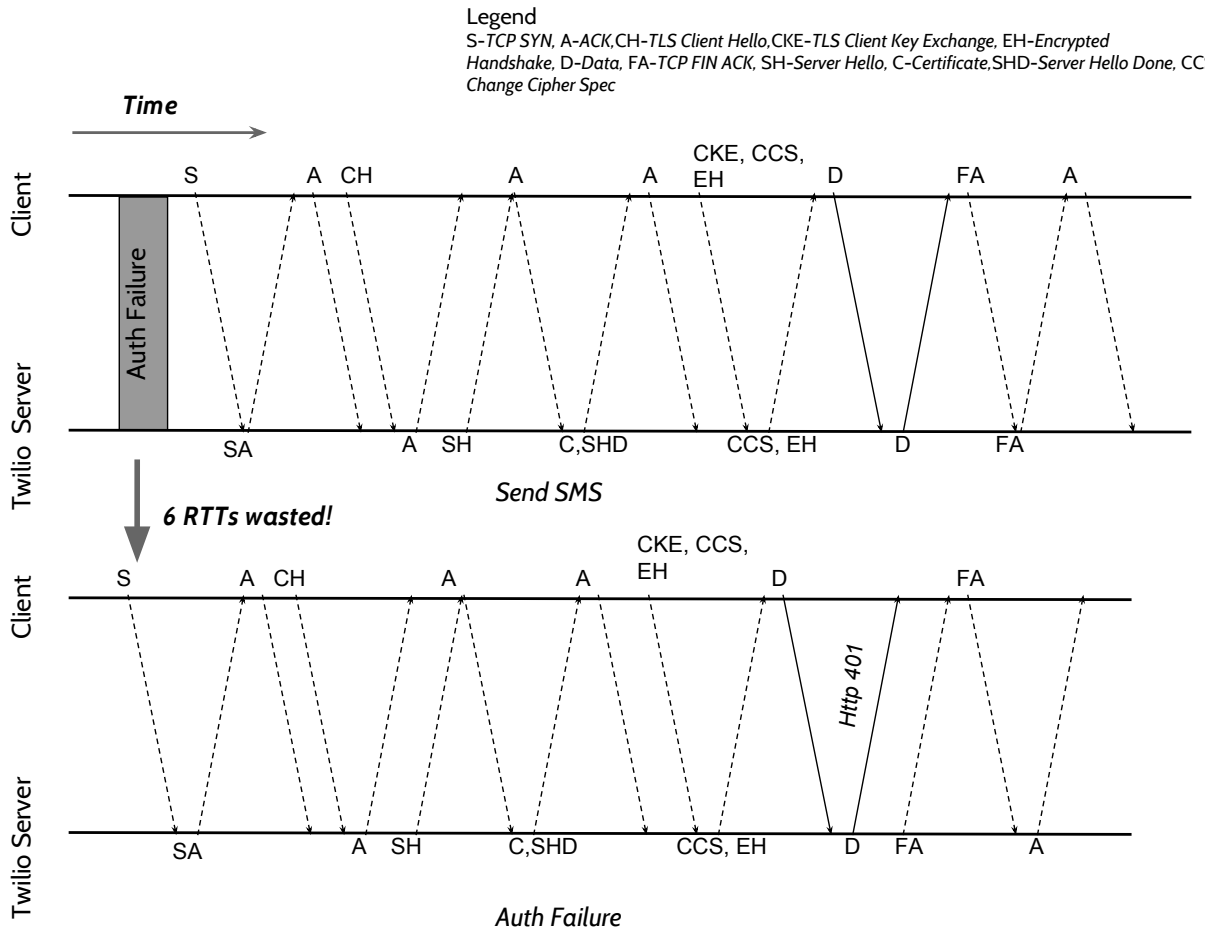

Figure 10: **Packet analysis for creating automated SMS** *The figure shows the sequence of packets exchanged between the client and the server when the client script tries to create a message.*

that fault handling mechanisms are not uniform across the system spanning the service and the client library. Also, in most of the cases that we tested by inducing faults from the caller side, we found that the callee is notified of the error conditions instead of caller. We believe that, in such scenarios it is best to inform caller with consistent error notifications since it is the caller who can take actions against those error and correct the errors.

Security: We found that presently there is no way for Application servers to authenticate the Twilio servers when the later makes the *HTTP POST* requests. In the interest of securing the application servers from attacks, we believe Twilio may provide IP whitelists which can be used by the Application servers to authenticate requests. Even IP whitelists might not be sufficient for scenarios in which Application servers are located behind proxies and the IP from which request originates gets masked (e.g. application server hosted behind load balancer in PaaS architecture). For such scenarios some more robust security measures needs to be provided.

Intelligent TwiML fetching: As we have shown in Section 4.4, in case of automated calls to phone, Twilio server fetches the TwiML only after placing the call. We understand that this technique saves extra RTTs in case the call is rejected by the phone. But from the perspective of a callee and assuming that most calls will be answered, we believe it is best to prefetch TwiML before placing the call. Also, we see that in the current automated call model the application server is contacted each time a call is placed. One of the most popular use of automated call is to send/broadcast a single message to multiple numbers (e.g. to send promotional messages or alerts). In such scenarios, it is wasteful to flood the application server to fetch the same content multiple times both from Application server as well as Twilio server point of view. To handle such scenario Twilio might provide some special options using which the application server can indicate that the same message has to be used for a specified number of calls or for some specified period of time and the Twilio server can then fetch the request once and cache it.

8 Future Work and Conclusions

In this section we discuss some of the possible avenues to extend this work, then present the implications of our study and finally conclude. As a future work, we think to study other CC services and compare different services in terms of traffic characters, usage patterns, user base diversity etc. We also intend to develop a much more generic and sophisticated gray-box framework to study CC services. Our current implementation of the toolbox is highly specific to *Twilio*. We also started looking at the security aspects of the *Twilio* servers by doing port scans using *nmap*. From our initial analysis, the servers have only ports 80 and 443 open. As a future work, we would like to analyse if some impersonation attacks are possible with *AuthTokens*. We also want to study how misbehaving clients can attack or cause some form of interference in the system.

Our study has implications for both application developers and *Twilio* developers. First, we showed different components of the *Twilio* ecosystem and how they interact. Our study gives more insights for developers to build robust applications atop *Twilio* platform. We also showed how *Twilio* meets its guarantees in terms of call and message dequeuing rates. This gives the application developers a good sense of what to expect from the *Twilio* service. Second, we should some possible improvements to *Twilio* in sections 6 and section 7.

To conclude, we developed a simple VoIP service atop *Twilio* platform and a graybox toolset to gain insights in to the protocols and the architecture of the system. We also empirically measured call and message queue/dequeue rates. We showed some interesting oddities in the service and the client library. We also pointed out to some possible enhancements to the *Twilio* ecosystem. We showed that our study has implications for both application developers and *Twilio* developers. Our study is a small step towards studying the rapidly growing Cloud Communication Services arena and we strongly believe further research is required in exploring this area.

References

[1] Twilio Call Rate Limit. https://www.twilio.com/help/faq/twilio-basics/what-are-the-limits-on-outbound-calls-and-sms-messages-per-second.

[2] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.

[3] Paul Barford and Mark Crovella. Critical path analysis of tcp transactions. *SIGCOMM Comput. Commun. Rev.*, 31(2 supplement):80–102, April 2001.

[4] Salman A. Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. 2004.

[5] Nathan C. Burnett, John Bent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 29–44, Monterey, CA, June 2002.

[6] Idilio Drago, Marco Mellia, Maurizio M. Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: Understanding personal cloud storage services. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 481–494, New York, NY, USA, 2012. ACM.

[7] James Nugent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Controlling your PLACE in the File System with Gray-box Techniques. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 311–324, San Antonio, Texas, June 2003.