# Ryu Controller's Scalability Experiment on Software Defined Networks

Saleh Asadollahi
*Computer Science*
Saurashtra University
Rajkot, India
asadollahimcitp@gmail.com

Bhargavi Goswami,
*Computer Science*
Christ University
Bangalore, India
bhargavigoswami@gmail.com

Mohammed Sameer
*Computer Science*
Christ University
Bangalore, India
sameersam9519@gmail.com

***Abstract*—Software defined networks is the future of Computer networks which claims that traditional networks are getting replaced by SDN. Considering the number of nodes everyday connecting to the global village of internet, it becomes inevitable to adapt to any new technology before testing its scalability in presence of dynamic circumstances. While a lot of research is going on to provide solution as SDN to overcome the limitations of the traditional network, it gives a call to research community to test the applicability and caliber to withstand the fault tolerance of the provided solution in the form of SDN Controllers. Out of the existing multiple controllers providing the SDN functionalities to the network, one of the basic controllers is Ryu Controller. This paper is a contribution towards performance evaluation of scalability of the Ryu Controller by implementing multiple scenarios experimented on the simulation tool of Mininet, Ryu Controller and iPerf. Ryu Controller is tested in the simulation environment by observing throughput of the controller and checked its performance in dynamic networking conditions over Mesh topology by exponentially increasing the number of nodes until it supported tested on high end devices.**

***Keywords— Software Defined Netwoks (SDN), Mininet, OpenFlow, Ryu, iPerf, Gnuplot***

## I. INTRODUCTION

IoT, 4G, 5G, VANET [1], etc, are all thirst area of network communication that offer more comfort in our life in one side and make computer network complex in other side by adding more and more users to Internet. Huge Data Center infrastructure build before 10 years are still running on complex traditional network equipment such as routers and switches, where administrator are supposed to implement high level policy through low level and pre-defined commands that any company such as Cisco, HP, Juniper who have their own commands and configuration method [2]. Lack of programmability was resisting programmers, researcher and developers from writing the custom applications for networks. As a solution, implementing the idea of research with new approach in the area of network is the major focus of researchers to look deeper to this complex equipment architecture and that is why SDN was born [3].

According to Traditional Architecture, Forwarding plane, Control plane and Management plane all are tied up together in traditional routers. Combination of these three planes in a same chassis makes it complex and tough to manage. Software Defined Networks simplifies traditional routers with main idea of separation of control plane from data plane, and provide a centralized control by means of controllers for whole or a part of networks.

According to SDN Architecture, devices are composed of only forwarding plane that includes a) logical and physical ports, b) flow and group tables. Based on the configuration specifications of control plane (brain), flow tables get filled up by controller. Once device receives new packet, look into the flow tables and take proper action. In case of lack of information, forwarding device sends the packet to the controller or drops the packet base on policy. In accordance to the controller's configuration, new records get added to the flow tables and further forwarding of the packets happens independently in next cycles.

The key component in SDN is Controller where, POX [4], OpenDaylight [5], Floodlight [6], Beacon [7], Ryu [8], NOX [9], etc, are few of them with different features that were compared by the team of authors in [10].

There are two another major components: 1) northbound interface, who is responsible to provide an abstract for application layer. It also hides the detail of down layer and makes writing the network management and control application easier for developer. 2) Southbound interface, which provides communication between the controller and forwarding devices. OpenFlow [11] is the most well documented SDN protocol that is used majorly in research community with current version of 1.5. Opflex [12], NETCONF [13], ForCES [14], POF [15] are other options for southbound interface.

Separating the forwarding plane from control plan and taking it on a remote system will generate questions on its capabilities of scaling on diverse scenarios. To throw light upon the scalability of the controller and checking the behavior of controllers in multiple diversified networking situations, the authors of this paper presents here the experiments with criteria of scalability and performance of the controllers.

The paper is formed in the following manner. Section II is providing helicopter view on Ryu controller. Section III provides details about the simulation test bed set to perform the experiments on scalability with diversified networking conditions. Section IV provides the obtained experimental results and evaluation of performance statistics followed by conclusion and references.

## II. RYU SDN CONTROLLER

The Ryu Controller is open source and under the Apache 2.0 license, written completely based on Python, supported and deployed by NTT cloud data centers. Main source code can be found on GitHub, provided and supported by Open Ryu community. It supports NETCONF and OF-config network management protocols, as well as OpenFlow. Considering the compatibility, OpenFlow switches, Hewlett Packard, IBM, and NEC are tested and certified with Ryu controller. It supports the OpenFlow protocol up to the latest version 1.5.

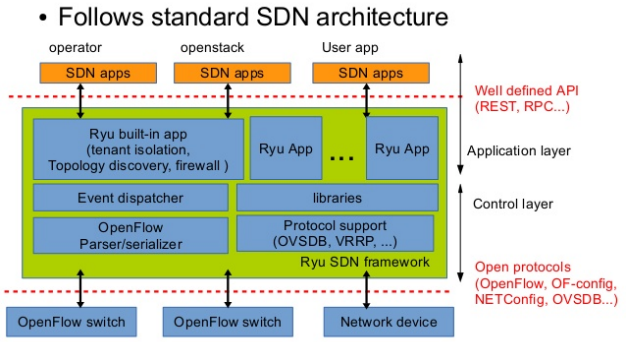


Fig. 1. Ryu SDN controller architecture

Same as other SDN Controllers, Ryu is also creating OpenFlow packets, managing events related to incoming and outgoing packets. It has abundant list of libraries which supports packet processing operations. In respect of support for southbound protocols, Ryu is working hand in hand with protocols such as XFlow (Netflow and Sflow), OF-Config, NETCONF, Open vSwitch Database Management Protocol (OVSDB), etc. VLAN, GRE and VLAN, etc is also supported by Ryu Packet Libraries.

Let us have a look on Ryu Managers and Core-Processes. The main executable is Ryu Manager. Ryu runs and listens to peculiar IP and Port, eg. 0.0.0.0:6633 to connect to Ryu manager which uses RyuApp class using inheritance where, the Ryu messaging service does support components developed in other languages.

Ryu is distributed with multiple applications such as a simple_switch, router, isolation, firewall, GRE tunnel, topology, VLAN, etc. Ryu applications are single-threaded entities, which implement various functionalities. Ryu applications send asynchronous events to each other. The functional architecture of a Ryu application is shown in Figure 2.

To preserve the order of events, each Ryu application has a receive queue (FIFO) for events FIFO. The thread's main loop pops out events from the receive queue and calls the appropriate event handler. Hence, the event handler is called within the context of the event-processing thread, which works in a blocking fashion, i.e., when an event handler is given control, no further events for the Ryu application will be processed until control is returned.

## III. SIMULATION ENVIRONMENT

As a simulator, Mininet [16] is used and as a controller Ryu. Mininet and Ryu controller, both are installed in a same virtual machine. The switch used in this experiment is OpenFlow kernel switch, also known as Open vSwitch or OVSK-Switch [17] by enabling OpenFlow protocol mode.

Python is used as scripting language to write the topology instead to accepting the automatic decision of number of host connecting to switch or default command provided by Mininet. The code of the python script is provided in Fig. 2. Python script of customized topology includes the specification of host to switch, switch to switch and switch to Ryu controller.

```
from mininet.topo import Topo
class MyTopo( Topo ):
   "Simple topology example."
   def __init__( self ):
      "Create custom topo."
   # Initialize topology
      Topo.__init__( self )
   #Add switches
   for s11Switch in range(1):
      s1Switch = self.addSwitch( 's1' )
   for s22witch in range(1):
      s2Switch = self.addSwitch( 's2' )
      self.addLink( s1Switch, s2Switch )
   for s33witch in range(1):
      s3Switch = self.addSwitch( 's3' )
      self.addLink( s2Switch, s3Switch )
      self.addLink( s1Switch, s3Switch )
   for s44witch in range(1):
      s4Switch = self.addSwitch( 's4' )
      self.addLink( s1Switch, s4Switch )
      self.addLink( s2Switch, s4Switch )
      self.addLink( s3Switch, s4Switch )
   for s55witch in range(1):
      s5Switch = self.addSwitch( 's5' )
      self.addLink( s1Switch, s5Switch )
      self.addLink( s2Switch, s5Switch )
      self.addLink( s3Switch, s5Switch )
      self.addLink( s4Switch, s5Switch )
   for s66witch in range(1):
      s6Switch = self.addSwitch( 's6' )
      self.addLink( s1Switch, s6Switch )
      self.addLink( s2Switch, s6Switch )
      self.addLink( s3Switch, s6Switch )
      self.addLink( s4Switch, s6Switch )
      self.addLink( s5Switch, s6Switch )
      # Add hosts and link to switches
   for h1_ in range(0,30):
      h1=self.addHost('h1_%s' % (h1_+1))
      self.addLink( h1, s1Switch )
      for h2_ in range(30,60):
      h2=self.addHost('h2_%s' % (h2_+1))
      self.addLink( s2Switch, h2 )
   for h3_ in range(60,90):
      h3=self.addHost('h3_%s' % (h3_+1))
      self.addLink( s3Switch, h3 )
   #for h4_ in range(30,40):
      #h4=self.addHost('h4_%s' % (h4_+1))
      #self.addLink( s4Switch, h4 )
   for h5_ in range(90,120):
      h5=self.addHost('h5_%s' % (h5_+1))
      self.addLink( s5Switch, h5 )
   for h6_ in range(120,150):
      h6=self.addHost('h6_%s' % (h6_+1))
      self.addLink( s6Switch, h6 )

topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Fig. 2. Python script for generating scenarios

To evaluate the statistics related to the performance of controller, mesh topology is implemented over 6 switches with five different scenario having difference in only number of nodes connected to each peripheral switch. As an effort to implement and test the controller's performance using scalability, we created a custom topology with five different scenarios having difference in the number of nodes as shown in Table 1.

Table1: Scenario Table for Experiment

| Scenario | Number of switch | Number of nodes |
|---|---|---|
| Scenario 1 | 6 | 50 |
| Scenario 2 | 6 | 100 |
| Scenario 3 | 6 | 150 |
| Scenario 4 | 6 | 200 |
| Scenario 5 | 6 | 250 |
| Scenario 6 | 6 | 300 |

Table2: Configuration Specification for Experiments

| | |
|---|---|
| Ubuntu | 16.04.3 LTS |
| Mininet | 2.2.1 0dl |
| OpenFlow | ( 0x1:0x4)1.3 |
| iPerf | 3.0.7 |
| CPU | Intel Core i5 520M |
| RAM | 6GB DDR3 |

Now the step by step procedure is followed to perform the experiment on Ryu using Mininet. For obtaining statistics tool used is iPerf.

Step 1: The first step is to run the Ryu controller using the script. Here, the name of application program is simple_switch_stp_13.py. Simple_switch_stp_13 is an application program to develop spanning tree scenario because, we are using mesh topology and mesh topology has loops. To avoid loops we need spanning tree and thus, stplib.py library is used which performs Bridge Protocol Data Unit BPDU packet exchange. Before executing this command, control must be in the folder of ryu. Now, we provide Command:

./bin/ryu-manager ryu/app/simple_switch_stp_13.py

Step 2: Next step is to run the mininet mesh topology script, by providing topology name and switch OVSK with the following command: sudo mn --custom ~/mininet/examples/mymesh.py --topo=mytopo --mac --controller remote --switch ovsk. Once the command is executed, check the connectivity between all the hosts using mininet Command: pingall.

Step 3: Now we define one client and one server which is done by any two host of the developed network. The command to perform this task is: xterm h1_1 h6_60. We have used first and last host with xterm command. This will open two terminal windows, one as client and another as server. Check configuration details on both the windows with command: ifconfig.

Step 4: Now we need to generate the traffic between client and server and log the events using iPerf tool. First we go to server window and enter the command: iPerf –s –p 6633 –i 1 > result. Here, 'result' is the filename provided to store the results. Once the server starts waits for the client. Now at the client side, to generate traffic, we need to provide IP address of the server with the port address by following command: iPerf –c 10.0.0.50 –p 6633 –t 100. Here 100 represent time in seconds.

Step 5: Next step is filtering the logged file for obtaining experiment specific results. We can check the content of generated file using command: more result. For filtering we have used grep and awk command: cat result | grep sec | head -100 | tr – " " | awk '{print $3,$5}' > myresults. Here, 'myresults' is the name of file where filtered results are stored which can be checked for content using 'more' command: more myresults.

Step 6: Next step is to plot the graphs of obtained results for which Gnuplot is used in this experiment. To start gnuplot tool, the command is: gnuplot. Next, plot the content of 'myresults' file using command: plot "myresults" title "Tcp_Flow" with linespoints.

In the same way, using the python script all other scenarios are developed as stated in Table 1 with configuration provided in Table 2.

One by one each scenario is tested using the step 1 to step 6 and results are obtained which is discussed in upcoming section of performance analysis. Kindly note that simulation execution of experiment needs RAM support not less than 6GB, especially for the simulations having nodes more than 200.

## IV. PERFORMANCE ANALISIS

This section provides the results obtained during the experimentation. With this paper, authors have made attempt to address the scalability features of the Ryu controller by implementing six scenarios in simulation experimental environment which will be discussed in this section in detail. This section has total 6 graphs which represents results for each scenario listed in Table 1. To evaluate performance of Ryu controller in incremental experiment with respect to scalability, throughput is the best matching parameter which will suffice our aim of experiment. Thus, in this section, we have limited the study to throughput only.

Graph of Fig. 3 shows the results obtained by performing transmission between client and server having the strength of nodes supported by network limited to 50. It is observed from the graph that average throughput stays at 1.65 Gbps. Graph of Fig. 3 also shows that the variations are very high within the duration of 100 sec of simulation.

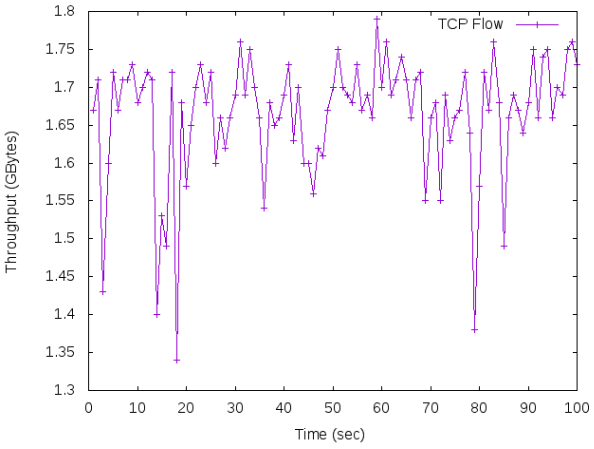


Fig. 3. Throughput for scenarios with 50 nodes.

Similarly, the stability is a big concern if we look at the Fig. 4 which demonstrates the throughput graph of scenario with 100 nodes. It is even worst as far as stability is concerned in comparison of Fig. 3. Again, if observed well, Fig. 5 is a bit stable even in presence of number of nodes equal to 150. However, there are few instances of highly volatile behavior of the network in Fig.5. Fig. 6 graph describes again excessive variations in the throughput when the number of nodes reaches to 200. Once again, if we observe the graph shown in Fig. 7

having 250 nodes, it seems to be stable in comparison of graph of Fig. 4 and Fig. 6 with 100 and 200 nodes. Still few instances does not prove it better in comparison of Fig.3 and Fig.5 with 50 and 150 nodes. Once, the final result of 300 nodes scenario was obtained, it was observed that throughput was increasing will tolerance of few instance of dropping. For few seconds, simulation was running well but, by the end of reaching to middle of the simulation time, slowly, dropping instances were observed frequently leading to degraded performance in the second half of the simulation run.

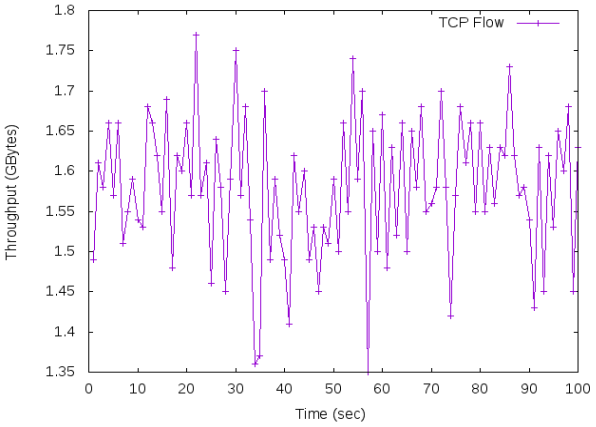


Fig. 4. Throughput for scenarios with 100 nodes.

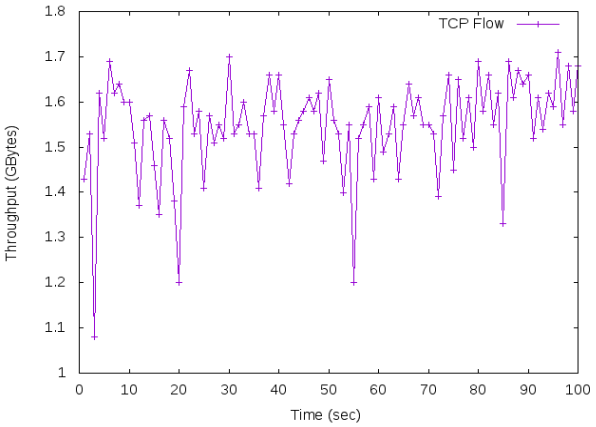


Fig. 5. Throughput for scenarios with 150 nodes.

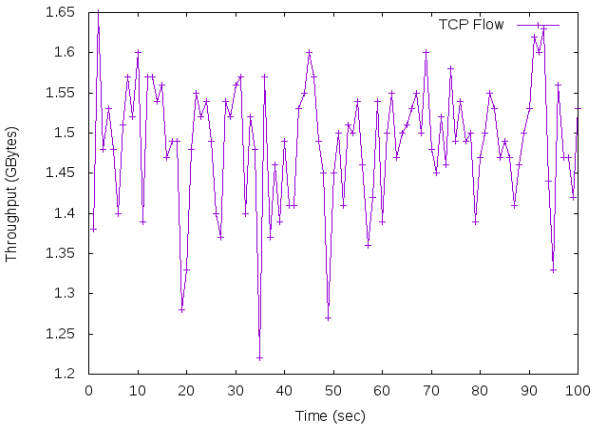


Fig. 6. Throughput for scenarios with 200 nodes.

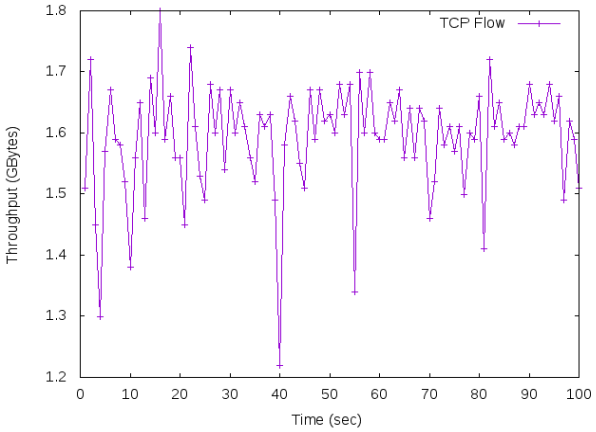


Fig. 7. Throughput for scenarios with 250 nodes.

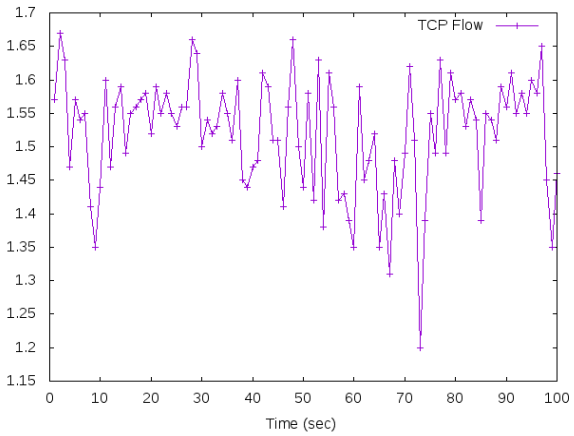


Fig. 8. Throughput for scenarios with 300 nodes.

With these graphs shown from Fig. 3 to Fig. 8, it was observed that Ryu is very much resource demanding controller which uses CPU and RAM utilization to optimum and thus results to degraded performance in presence of increasing number of nodes. Even after checking the performance with high end systems, the resource utilization was observed to be too high in comparison of scale of nodes support provided by the controller. We limited the study to 300 nodes because with even i7 processors with 8GB RAM resources were proved not enough during the execution of experimentation.

## V. CONCLUSION AND FUTURE SCOPE

With this paper, authors have made attempt to address the scalability features of the Ryu controller by implementing various diversified scenarios in simulation experimental environment. In this paper, authors have provided the clear idea how to create experimental test bed along with analysis of obtained statistical results keeping the throughput performance as the central focus. We would conclude this paper by providing negative sign to move forward to the researchers who are looking for implementation of their idea over Ryu Controller in the domain of Software Defined Networks without any doubt.

The controller not just provide the simulation experimental test bed support but, also provides clear explanation for analysis of obtained statistics after the experiments are simulated. The tools suggested, simulated, shown through figures and graphs will help the research community to further conduct such experiments in the future by implementing their desired parameters though these experiments to improvise Ryu Controllers which is required as far as current performance is concerned. This paper will also address the programmers, developers and new bees in the area of SDN, who are looking forward to touch the practical aspects of the SDN by following implementation details provided in the paper. Further, the research team will come up with few more papers on implementation of other SDN controllers in the coming future. The team also has planned to compare the controllers of SDN, once all the stellar controllers are implemented and experimented by them.

## REFERENCES


[1] Gowsami, B. Asadollahi, S., (2016). "Novel Approach to Improvise Congestion Control over Vehicular Ad Hoc Networks (VANET)" Proceedings of the 10th INDIACom; International Conference on "Computing for Sustainable Global Development", March 2016. Delhi, India. IEEE Xplore ISBN: 978-9-3805-4421-2

[2] Gowsami, B. Asadollahi, S., (2017). "Enhancement of LAN Infrastructure performance for data center in presence of Network Security" Proceedings of Computer Society of India, Springer, Next-Generation Networks pp 419-432 ISBN: 978-981-10-6005-2

[3] Asadollahi, S., Gowsami, B. (2017). Revolution in Existing Network under the Influence of Software Defined Network. Proceedings of the INDIACom 11th, Delhi, March 1-3.2017 IEEE Conference ID: 40353

[4] McCauley, M. (2012). POX, from http://www.noxrepo.org/

[5] Asadollahi, S., Gowsami, B. (2017). Implementation of SDN using OpenDaylight Controller. Proceeding of An International Conference on Recent Trends in IT Innovations - Tec'afe 2017. ISSN(Online) : 2320-9801

[6] Project Floodlight, Floodlight. (2012). from http://floodlight.openflowhub.org/

[7] Erickson, D. (2013). The Beacon OpenFlow controller. Proceedings of ACM SIGCOMM Workshop Hot Topocs Software Defined Network II, 13-18 p, 2013.

[8] Nippon Telegraph and Telephone Corporation, RYU network operating system, 2012, from http://osrg.github.com/ryu

[9] Gude al, N. (2008). NOX: Towards an operating system for networks. ACM SIGCOMM - Computer Communication Revie. vol. 38, no. 3, pp. 105–110.

[10] Asadollahi, S., Gowsami, B. (2017). Software Defined Network, Controller Comparison. Proceedings of Tec'afe 2017,Vol.5, Special Issue 2, April 2017. ISSN: 2320-9798.

[11] McKeown et al, N. (2008). OpenFlow: Enabling innovation in campus networks. ACM SIGCOMM - Computer Communication Revie, vol. 38, no. 2, p. 69–74.

[12] Smith et al, M. (2014). OpFlex control protocol, Internet Engineering Task Force, from : http://tools.ietf.org/html/draft-smith-opflex-00

[13] Enns, R., Bjorklund, M., Schoenwaelder, J., Bierman, A. (2011). Network configuration protocol (NETCONF). Internet Engineering Task Forc, form http://www.ietf.org/rfc/rfc6241.txt

[14] Doria et al, A. (2010). Forwarding and control element separation (ForCES) protocol specification. Internet Engineering Task Forc, from http://www.ietf.org/r/fc/rfc5810.txt.

[15] Song, H. (2013). Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. Proceedings of ACM SIGCOMM Workshop Hot Topics Softw Defined Netw II. p. 127–132.

[16] Lantz, B. Heller, and N. McKeown. (2010). A network in a laptop: Rapid prototyping for software-defined network. Proceedings of ACM SIGCOMM Workshop Hot Topics Netw, 19th. p. 19:1–19:6

[17] B. Pfaff and B. Davie. (2013). The Open vSwitch database management protocol. Internet Engineering Task Force, RFC 7047, from http://www.ietf.org/rfc/rfc7047.txt