PYDISTSIM - DISTRIBUTED SYSTEM SIMULATION LIBRARY USING SIMPY

BY

KAI HUANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Roy H. Campbell

ABSTRACT

An easy to use and flexible distributed system simulation framework would be useful for beginners to learn about distributed systems or for researchers to prototype distributed system algorithms. This paper proposes a framework called PyDistSim for distributed system simulation implemented in Python. PDS focuses on providing a set of tools and libraries to make it easy and intuitive to set up simulations for distributed systems. Our framework is written in Python and is designed to be simple and user-friendly, while still being flexible and can be adapted to a wide variety of use cases. The simulation is deterministic and can be easily controlled. Moreover, the framework provides a variety of tools that can be used to conveniently collect and organize the data during simulation.

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Motivation

Researchers often want to quickly and easily set up simulations for distributed systems. For example, one may need to quickly set up a simulation for a prototype of a consensus algorithm and see if it has any obvious problems. Some of the desired characteristics of such a tool include ease of use, flexibility and determinism [1].

However, existing distributed system simulation tools usually do not meet the criteria above. Some tools such as SimGrid [2] could be fairly involved to setup, others, such as Simulink [3], requires a proprietary license. There are numerous existing distributed system simulators [4], but most of them use sockets and processes or threads to represent each node. Simulating distributed system using sockets and processes/threads introduces race conditions that make the simulation nondeterministic. There are existing distributed simulators that use discrete event simulation, but none of them are implemented in Python or have other limitations [5].

To address these issues, we introduce PyDistSim, a distributed system simulation framework that offers a light-weight and easy to use solution for implementing a distributed system simulation. Our framework is built on discrete event simulation[6] written in Python.

1.2 Advantage of simulation

Studying distributed systems using our framework has a few of advantages as opposed to using a full-fledged implementation that is mainly intended for real world application.

First, the architecture of the simulation software can be made much simpler. In simulation software, one does not have to deal with the many layers of plumbing that are necessary in real world application. This makes the logic of the algorithm much easier to understand, implement and debug and, therefore makes the implementation of the intrinsically complex algorithms less error prone.

Second, the simulation provides a more controlled execution environment. Using simulation makes it possible to model the system to rule out factors beyond our control, such as network hiccups, different processing power of different machine and race conditions between different processes. This makes the simulation execution deterministic - the execution takes the exact same path as long as the random number generator is seeded with the same seed. This not only facilitates tuning and debugging but also helps eliminate irrelevant factors from the experiments.

Third, simulation requires less computation resource usage. Simulation models can use very little resource without affecting the outcome of the executions. Therefore it is possible to simulate large-scale distributed systems with hundreds of individual hosts with limited resources.

When setting up the simulation, one often finds the need to write the code to model the nodes and network, collect data for analysis, as well as code to do other miscellaneous bookkeeping and plumbing, such as parsing network packets or writing various “adaptor code”. The motivation of this project is to eliminate such needs as much as possible, creating a simple but flexible environment where the user can focus more on the logics of the algorithm rather than spending a vast amount of time on plumbing code.

1.3 Contributions

This paper proposes a discrete-event simulation based distributed system simulation library that is simple and easy to use.

Our library is written entirely in Python, which is a modern and object-oriented language, with an elegant syntax. This makes working with our library highly efficient and less error prone.

Using Python also gives our library great portability, wherever Python environment is available, it can be used.

The users can easily write their distributed system implementation in Python based on our library. Our library is designed with Don’t Repeat Yourself principle in mind. It provides a lot of existing code that performs tasks commonly needed for distributed system simulation such as modeling network topology [9] so that the users do not need to do them over and over again. This makes our library particularly favorable for modeling distributed systems and prototyping distributed algorithms.

One of the most favorable features of our library is that the simulation can be easily controlled by the users. The simulation using our library is deterministic. The users can choose to run a fixed number of ticks (the smallest abstract time unit in our library) with the same random seed, and the result will always be the same. The user can also step the simulation and see every single event that occurs in the system. Therefore, compared to systems that are non-deterministic due to race conditions, simulations using our library are much easier to debug and therefore less prone to error in more complex simulations.

Our library also provides a stats module that can be used to integrate data collection into the simulation. By default, it provides a few basic plots and metrics to help the user gain a general idea of what is happening in the simulation. But the user can easily extend this and add more custom metrics to be collected and plotted.

To demonstrate the effectiveness of our library, we have included in this paper, a case study of successfully implementing and running the distributed system simulation of several variants of gossip protocols. The whole simulation only took less than a hundred lines to implement. The case study also showed that our library is very flexible, and can adapt to various different scenarios easily, to suit different needs.

The rest of the paper is structured as following: Chapter 2 provides some background information about SimPy, Python generators, discrete-event simulation and how our library models the simulation. Chapter 3 goes over the how our library can be used and the related design details. Chapter 4 presents a case study that uses our library to implement a few simulations of variants of gossip protocol, to demonstrate the usefulness of our library. Chapter 5 gives a conclusion.

2. BACKGROUND

2.1 Discrete-event simulation

PyDistSim is implemented using discrete-event simulation. In discrete event simulation, the simulation happens independent of real time, rather, the simulation can be carried out “as fast as possible”, always skipping to the next event that should occur. This allows the simulation to be run faster than real time. Also, the simulation is deterministic, as long as the random seed remains the same.

2.2 Simpy

Our implementation is based on SimPy [7], a process-based discrete-event simulation framework based on standard Python [8]. In SimPy, processes are represented by generator functions and can be easily used to model active components - and in our framework – the hosts.

SimPy provides various types of shared resources, which are suitable for modeling objects with limited capacity (like servers, or network connections).

2.3 Generators

The processes in SimPy are represented by generators. Python generators are iterators, which can be used in a for-loop. In PEP 255, generators are referred to as generator-iterators, which implies their nature as iterators. Generators are usually used in a for loop, but they can also be advanced by calling the next method on the object.

In Python, generator functions or just generators returns generator iterator objects. These generators are usually functions that contain the *yield* keyword. In Python, the yield keyword achieves the same effect as the __iter__ and next functions but is much more succinct and intuitive to write.

When the function is executed, the yield statement will cause the state of the function to be saved, all local state and variables are backed up, and the value of expression_list is returned to .next()‘s caller [10]. And when the .next() is called, the function can continue from where it left off last time, using the saved states. The yield statement effectively pauses the function which can be continued at a later time.

On the other hand, when a return statement is encountered in a function, the execution of the function is completed, the stack variables are popped and the execution returns to the caller.

Our library models every host as a collection of SimPy processes or generator functions. These processes yield timeout events every iteration. The timeout can be considered the timeout in our algorithms. And the execution of the rest of the process can be considered instantaneous in the simulation since they always complete in the same tick. Yielding timeout is important because it allows SimPy core to context switch to other nodes and allows the simulation to continue for other processes. In theory, the simulation is linear, because events are dequeued from an event queue, and processed in sequence. This has several advantages. First, the simulation is simplified, since, in reality, only one process will be running at the same time when the simulations are running, the code between two yield statements in a process is atomic in the simulated world, and therefore, no race condition is possible among the processes. Second, it allows the user to manually step through each event, running them one at a time, which provides a powerful tool for identifies bugs.

3. DESIGN

This section covers the API of the framework as well as some of the design details.

PyDistSim consists of three main modules, the Node Module, Network Module, and the Stats Module. The Node Module provides a basic representation of a node in a distributed system, allowing users to implement the node they want by extending it. The Network module represents the network topology, and is responsible for simulating the connections among nodes, it collects messages the nodes sent and delivers them to the desired destinations. It also simulates packet drops and latency, based on the networking model chosen by the user. The Stats module collects and generates a data report for the simulation. It generates a few plots by default but is easily extendable by the user.

Figure 1. shows a sequence diagram of an example simulation in action. As you can see in the diagram, simpy core calls the run() function of host1, which requests the network to send an RPC to host2. The network sets up an event with some delay for delivering that RPC, and returns. Moments later, when the timer on the event runs out, simpy.core invokes the callback on that event, which delivers the RPC to host2. Meanwhile, the stats module collects data for later use.

3.1 Node

The Node class represents a host. A single instance of Node represents a host in the distributed system simulation. A Node instance can have a number of periodic routines, in the form of a SimPy process.

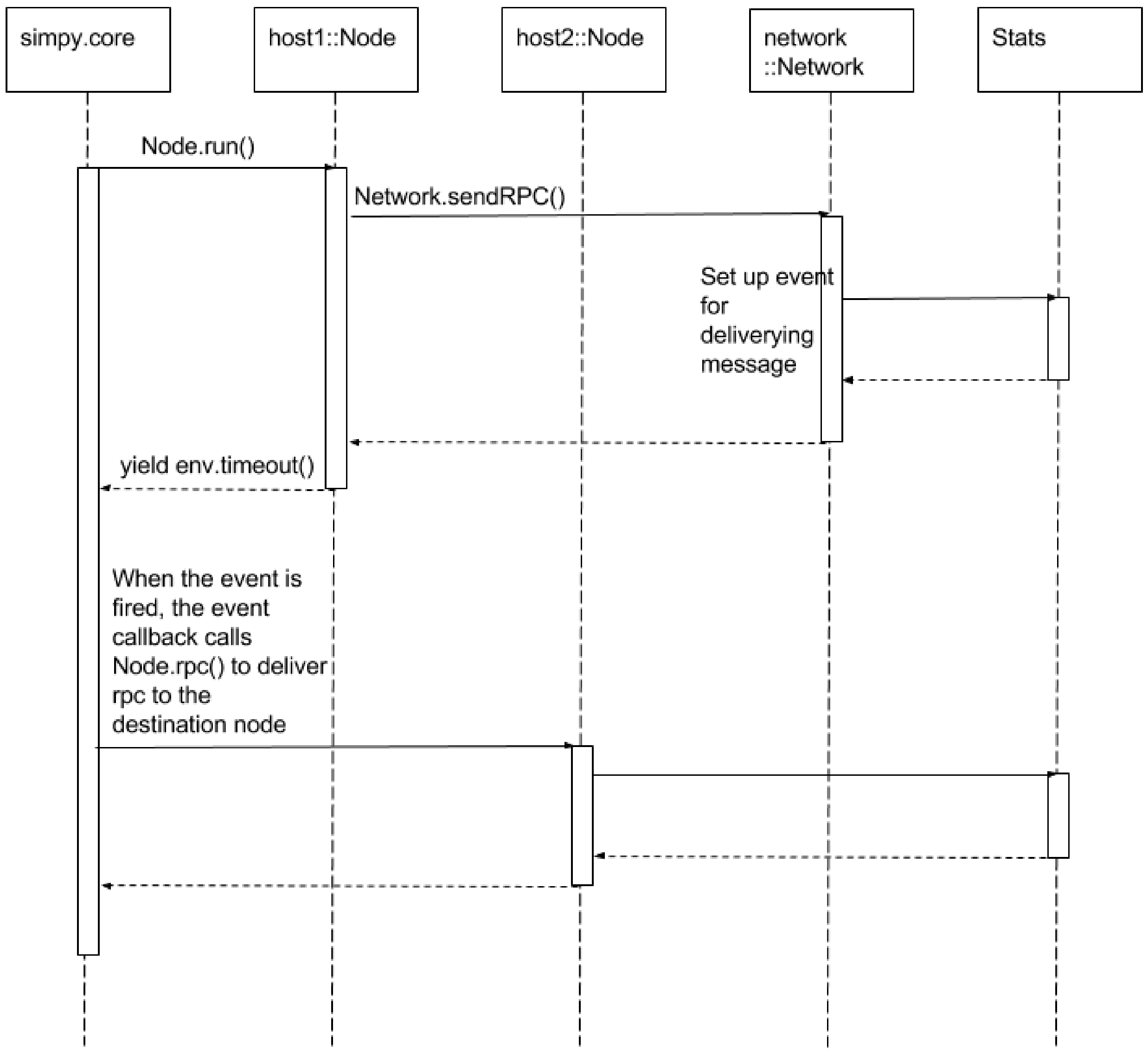
simpy.core
host1::Node
host2::Node
network ::Network
Stats
Node.run()
Network.sendRPC()
Set up event for deliverying message
yield env.timeout()
When the event is fired, the event callback calls Node.rpc() to deliver rpc to the destination node

Figure 1. Sequence diagram of an example simulation

As shown in the following code snippet, the user defines a class named MyNode that extends Node, and in the __init__ function of MyNode, it first calls the constructor of the parent class, does some initialization, then starts a process with the function myPeriodicRoutine. Inside the function myPeriodicRoutine, you can see a while loop, in which some actions are performed and then it yields a simpy.environment.timeout with a set interval INTERVAL. The yield statement is crucial for the operation of the simulation, yielding a timeout event object passes the control to the SimPy's simulation thread. Without it, the simulation will simply hang forever.

```
class MyNode(Node):
    def __init__(self):
        Node.__init__(self)
        # ...
        self.periodicRoutine = Node.env.process(self.myPeriodicRoutine())

    def myPeriodicRoutine(self):
        while self.running:
            # do periodic maintainance
            yield Node.env.timeout(INTERVAL)
```

Another important aspect of a Node is the ability to handle Remote Procedure Calls (RPC), which in this case are used as the only means of passing information between two Nodes(). The Node class has an instance method called rpc, it takes an RPC name and looks up the corresponding method in its RPC lookup table, and invokes that method if found. The user can add RPCs to the Node by calling the function registerRPC. For example, the following line of code will register the method self.foo as an RPC with the name “foo”.

```
self.registerRPC(‘foo’, self.foo)
```

The Node class also contains a static method fail() that allows the user to schedule failure of a group of nodes at a particular delay with or without randomness.

3.2 Network

The Network class represents the topology of the network. It also keeps track of the addresses of all the hosts. In PyDistSim, the addresses are simplified to be represented by integers, which should not affect the outcome of the simulation.

The Node class keeps a reference to an instance of Network as a class variable. The constructor of Node adds the new Node instance to the host list in Network class every time it is called.

The user can add links to the network by calling the addLink method of Network class.

All communication between nodes is represented as RPCs(Remote Procedure Calls). This simplification does not affect the completeness or correctness of the simulations. RPCs are trivially equivalent to any message passing mechanism since any message can be passed using an RPC.

The user can implement and add RPCs by calling the registerRPC function in the Node class. A Node can call an RPC by calling the rpcCall method of the Network class, it will generate an event with a delay that represents the latency of the link, with a callback that delivers the RPC to the intended recipient. The SimPy core calls the callback function at the desired delay and delivers the RPC.

```
def deliverMsg(evt):
    if dst in self.hosts and self.hosts[dst]:
        self.hosts[dst].rpc(evt.value[0], evt.value[1])

    #...
    event = simpy.events.Timeout(self.env, delay=latency, value=(rpcname, 
args))
    event.callbacks.append(deliverMsg)
```

The Link class represents a network connection between two nodes. The Link class determines the latency and chance of packet loss. The users should choose to use one of its subclasses to model the connection or implement their own. The SimpleLink class will give a fixed latency and packet loss rate, while the CongestionLink calculates latency and packet loss rate as a function of the level of congestion, which is updated when packets are sent through the link. The users can also choose to extend the Link class and implement their own logic.

The Network class also contains a function called visualize that automatically generate a plot showing the network graph.

To set up a simulation, the user would first extend the Node class and implement the RPCs. Then the user can call the constructor of Node to construct a number of nodes. After that, the user can add links needed to the Network. Finally, the user can run the simulation by calling Node.run(until_ticks).

3.3 Stats

The Stats class represents the data collected during a simulation. By default, the Network class will have an instance of Stats class as one of its member variables. Every time an RPC is sent or delivered, the Network class will call Stats.logRPC to append it to the log, along with a time stamp, source and destination address, name of the RPC and a boolean indicating whether it is sent or received.

```
self.stats.logRPC(self.env.now, src, dst, rpcname, True)
```

The Stats class has a few functions that can help users process or analyze the RPC log collected.

The *Stats.Filter* function can be used to filter out RPCs based on the criteria passed in. It returns a list of RPC log entries matching the criteria specified by the arguments passed in.

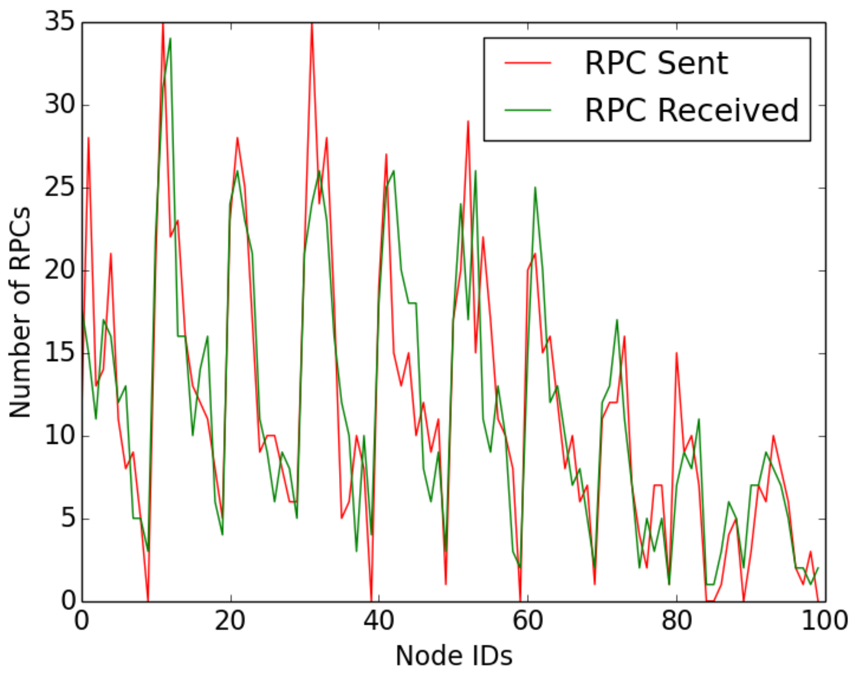

Figure 2. Plot generated by genSimplePlot
showing the number of RPCs sent and received for each node.

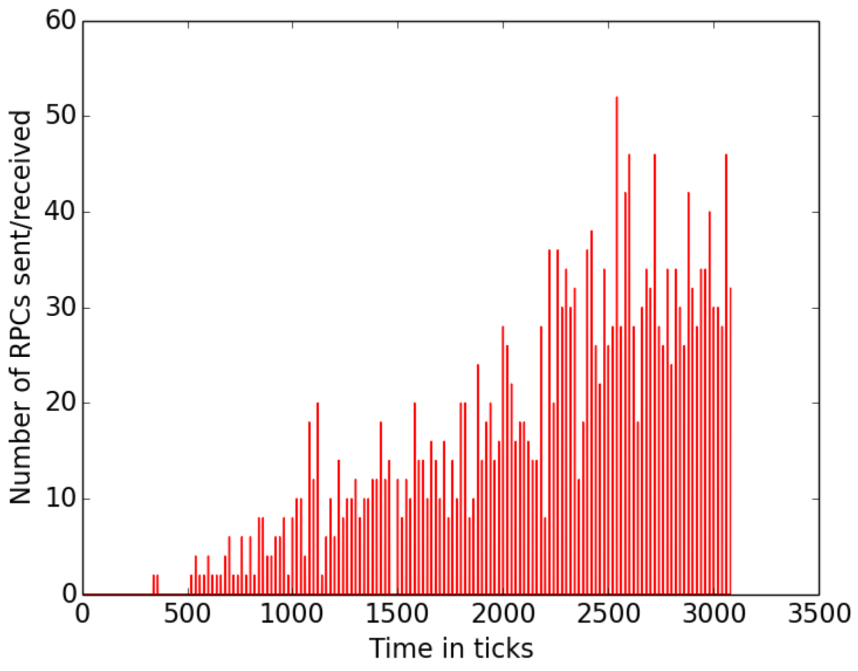

Figure 3. Plot generated by genSimplePlot
showing the number of RPCs per tick.

The *Stats.genSimplePlot* function can be used to generate three simple plots showing the RPC sent per node, RPC received per node and RPC against time. These plots can give the user a general idea of what is going on. Figure 2. and Figure 3. are examples of plots generated by *genSimplePlot.*

If the user wishes to generate other types of plots, they can conveniently use matplotlib to do so. matplotlib is a python library that provides an extensive collection of tools for plotting and visualizing data.

4. EVALUATION

In order to see how well our framework works, we present a case study of implementing a Gossip Protocol [11][12] simulation using our library. Gossip protocol is chosen mainly because it is one of the simplest distributed system communication protocol, and we can easily demonstrate the framework. There are multiple flavors of gossip protocol

We implemented a GossipNode class that is a sub-class of Node:

```
class GossipNode(Node):

  def __init__(self, chance):
      Node.__init__(self)
      self.registerRPC('gossip', self.gossipFunc)

      self.chance = chance
      self.value = self.address
      self.procGossip = Node.env.process(self.gossipProc())

   def gossipFunc(self, args):
      self.value = args



   def gossipProc(self):
      while self.running:
            if self.value == 0:
                  for n in Node.net.getNeighbors(self.address):
                  if random.uniform(0, 1) < self.chance:
                        Node.net.rpcCall(self.address, n, 'gossip',
self.value)

            yield Node.env.timeout(20)
```

We then conducted three experiments to demonstrate the effectiveness of our framework.

4.1 10 x 10 grid topology

A 10 by 10 grid topology was set up and with a node on one corner as the initiator, and with 0.05 chance of an “infected” sending a gossip to each of its neighbors. The simulation was run in steps of 20-time units until all nodes are “infected”, at each step, the number of “infected” nodes are collected. The simulation is repeated 50 times with different random seed, and the results are shown in Figure 4.

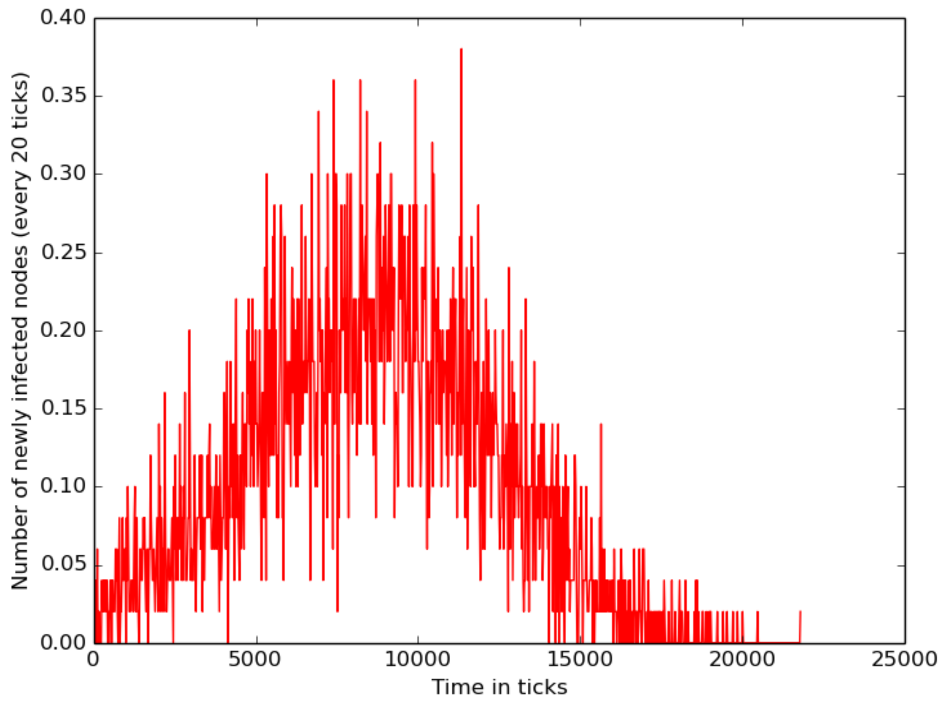

Figure 4. Grid topology of 100 nodes, with 0.05 chance of an “infected” node sending a gossip to each neighbor

4.2 Fully connected topology

A fully connected topology with 100 nodes was also tested in comparison, with a 0.01 chance of “infecting” neighboring nodes. And the results are shown in Figure 5.

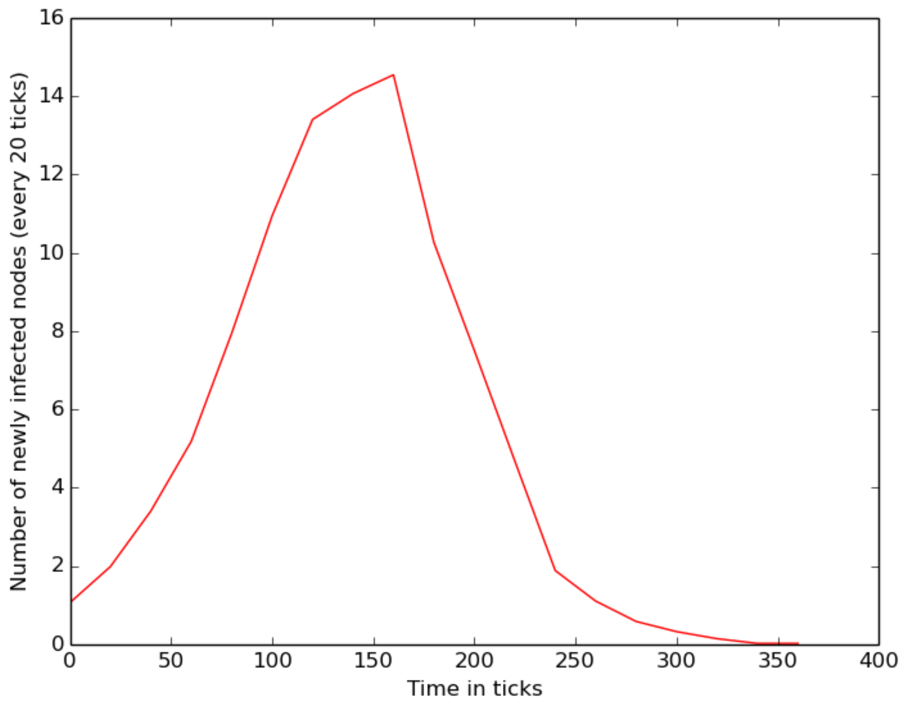

Figure 5. Fully connected topology of 100 nodes, with 0.05 chance of an “infected” node sending a gossip to each neighbor

4.3 Unbiased versus biased gossip

When a node uses an unbiased gossip it will more likely to choose gossip targets that are nearby, to increase efficiency.

To demonstrate the flexibility of our framework, we also experimented with biased versus unbiased gossip protocol, by setting up a network topology with two clusters of 50 nodes. Within each cluster, low latency links (20-tick) are added to make them fully connected. Then 50 high latency (200-tick) links are added to connect the two clusters.

The GossipNode is modified so that it only picks at most one target to gossip at each iteration. Two experiment setup was made, the biased setup make the nodes 10 times more likely to perform intra-cluster gossip than inter-cluster gossip, while the unbiased setup makes the nodes equally likely to pick any other node. The simulation for each setup was repeated 100 times and the result is presented in Figure 6. We can see in the plot, biased gossip converges much faster than unbiased gossip, as expected.

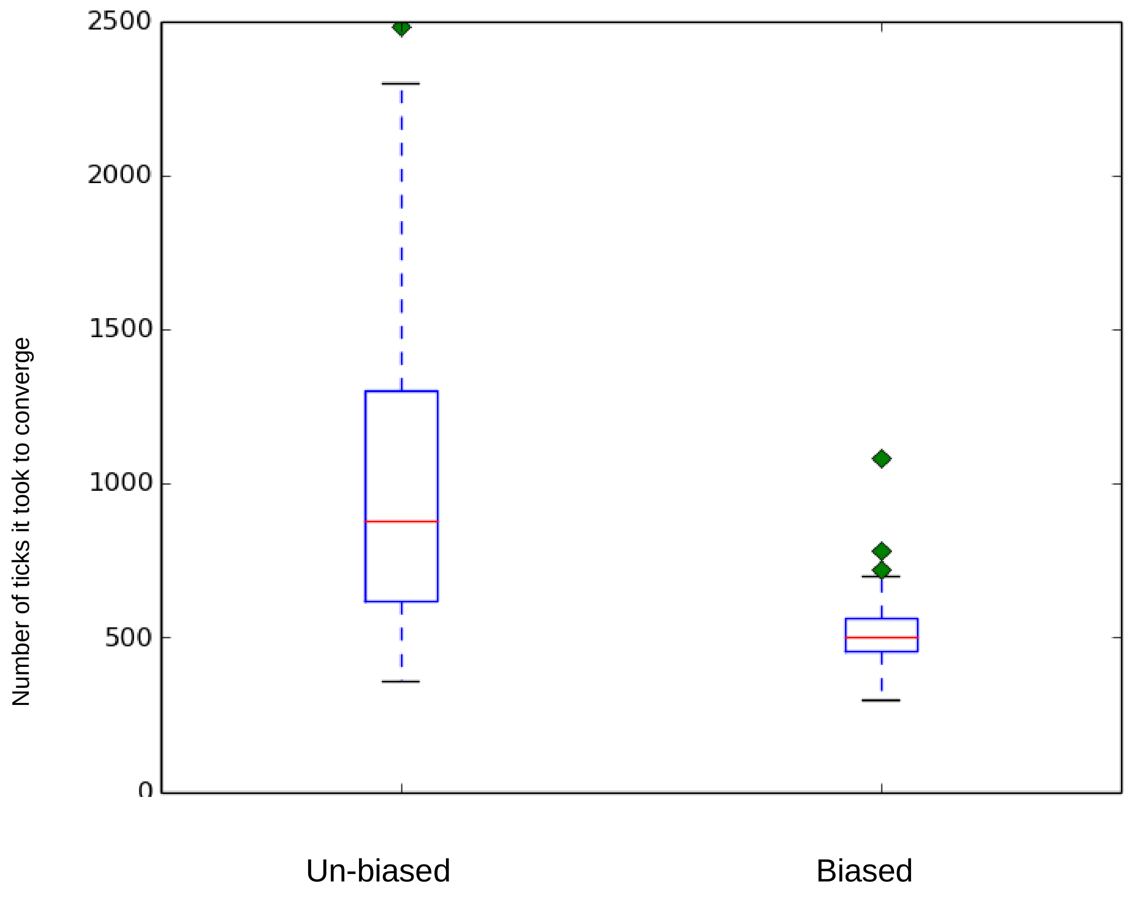

Figure 6. Unbiased versus biased Gossip

4.4 Gossip protocol for computing aggregates

We can modify our previous experiment slightly to simulate a gossip protocol that computes aggregate for a group of nodes. In this case, we would like to find the minimum value among all the nodes.

The gossip node is modified such that when it receives a gossip from another node, it will compare the value in the gossip message with its local value.

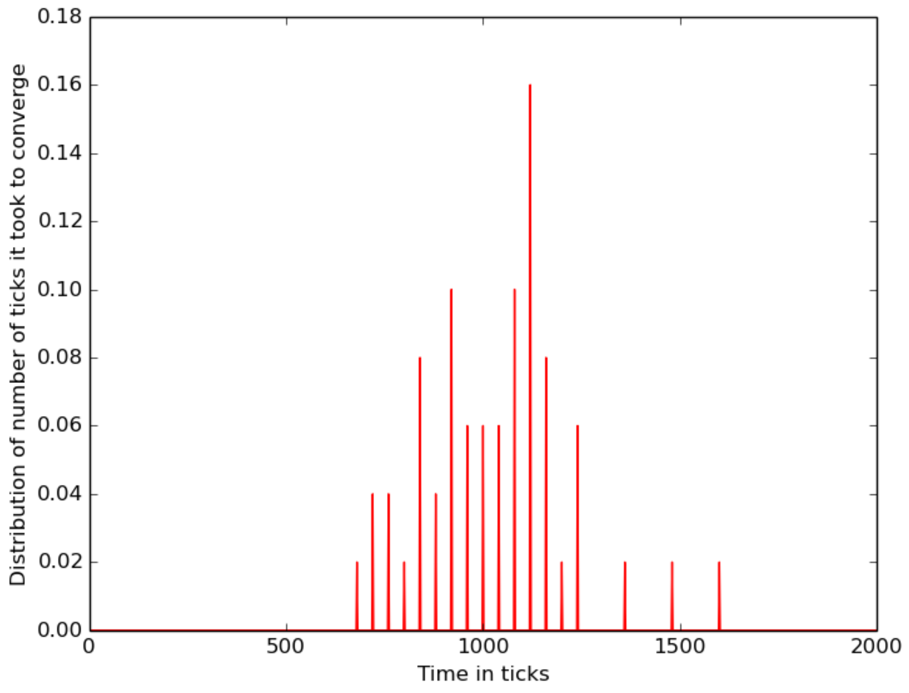

Figure 7. Gossip aggregation convergence distribution

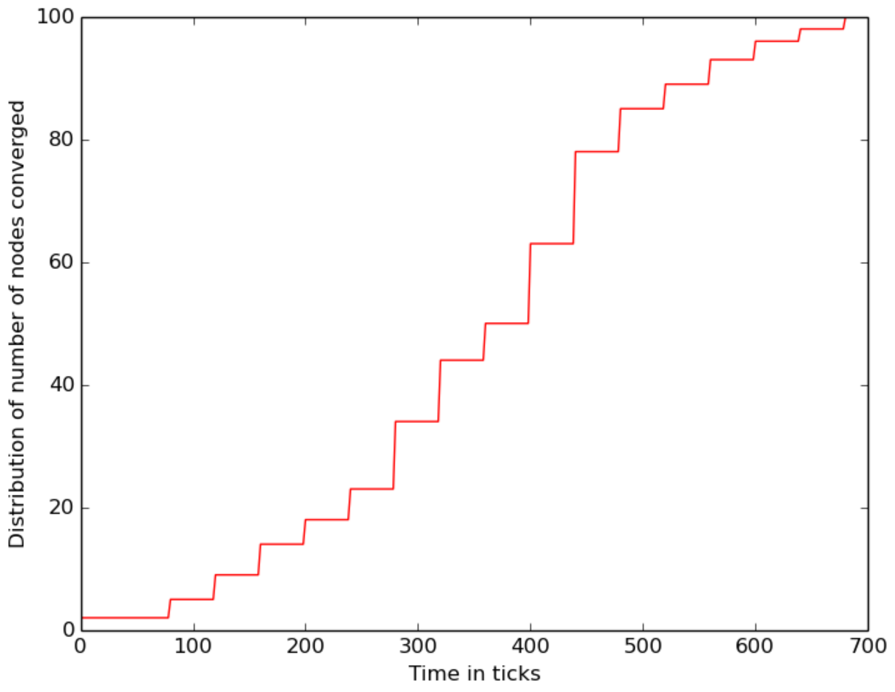

Figure 8. Gossip aggregation convergence per tick

If local value is smaller, it sends the local value to the sender, otherwise, assign the value in gossip message to local value. The system should converge, and all the nodes should have the minimum value in the end.

We set up 100 connected nodes, using 20-tick links. The simulation was run until it converged, and repeated for 50 times. We plotted the percentage of the number of ticks it took to converge, shown in Figure 7. We also plotted the number of nodes that have already reached the minimum value against number of ticks since the beginning of the simulation, shown in Figure 8.

Through our case study, we have successfully shown that our library is very easy to use and can be used to implement simulations for a variety of distributed.

5. RELATED WORK

There exist numerous simulation frameworks. This chapter will be discussing a few of the most popular simulation frameworks, which are listed in the table below.

Framework	Language	Other features/characteristics
NetSim	C	Specializes in simulating a variety of real world internet protocols [15]. It allows real data to be collected and used in the simulation. Provides a drag-and-drop UI but does not offer much flexibility for implementing custom protocols.
SIMUL8	Visual Basic	Simulation of work items that represent physical objects that are moved through the system [16], targeted for simulation of an assembly line.
Simulink	MATLAB	General purpose discrete-event simulator with a drag-and-drop GUI. It is implemented as a plugin for MATLAB and relies on MATLAB to work.
DESMO-J	Java	A discrete event simulation library implemented in Java, it supports hybrid event/process model.
PyDistSim	Python	A distributed system algorithm simulation library built on top of SimPy - a discrete event simulation library in Python. It is lightweight and easy to use for quickly modeling distributed systems.

Table 1: Comparison of simulation frameworks

The frameworks listed in the table above all serve some specific purpose, but they may not be as suitable or tailored to the purpose of modeling the simulation for algorithms used in distributed systems.

NetSim and SIMUL8 for example, are targeted to network simulations and industrial assembly line simulations respectively. These frameworks impose very rigid simulation models and may require a significant amount of rework in order to use them for simulating algorithms for distributed system. Simulink, a discrete-event simulation framework for MATLAB, is a powerful framework that has a wide variety of applications. However, it presents a huge and complex API, and requires a cumbersome installation of MATLAB. DESMO-J is a discrete event simulation library in Java, it is somewhat suitable for simulating distributed systems, however, many users may favor Python over Java for because of Python’s elegant and succinct syntax, which greatly improve the efficiency in programming. For example, in Python code blocks are denoted using white space, whereas in Java brackets are used, Python is dynamically typed and allows variable declaration without a type, whereas Java does not. PyDistSim is targeted specifically to simulating algorithms in distributed systems, and is flexible and easy to use. It also provides a number of tools for helping with data collection and plotting.

6. CONCLUSION

We have demonstrated that PyDistSim is a lightweight, easy to use and extendable distributed system simulation library. Using our case studies of gossip protocol simulation. We managed to set up the simulation with around one hundred lines of code and showed the potential to be expanded easily to cover more possibilities. We also demonstrated that our simulation can be easily controlled and debugged because it is deterministic.

Our library provides a simple and deterministic simulation environment that is easy to work with while still allowing the user a lot of freedom to extend and build upon. PyDistSim could potentially make learning and studying distributed system much easier especially for beginners.

REFERENCES

[1] Yabandeh, Maysam, Nedeljko Vasic, Dejan Kostic, and Viktor Kuncak. "Simplifying Distributed System Development." In HotOS. 2009.

[2] Casanova, Henri (May 2001). "A Toolkit for the Simulation of Application Scheduling". First IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01). Brisbane, Australia.

[3] Mathworks. "MatLab & Simulink: Simulink Reference R2015b" (http://www.mathworks.com/help/releases/R2015b/pdf_doc/simulink/slref.pdf)

[4] Casanova, Henri, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. "Versatile, scalable, and accurate simulation of distributed applications and platforms." Journal of Parallel and Distributed Computing 74, no. 10 (2014): 2899-2917.

[5] Franceschini, Romain, Paul-Antoine Bisgambiglia, Luc Touraille, Paul Bisgambiglia, and David Hill. "A survey of modeling and simulation software frameworks using Discrete Event System Specification." In OASIcs-OpenAccess Series in Informatics, vol. 43. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[6] Thomas J. Schriber, Daniel T. Brunner, INSIDE DISCRETE-EVENT SIMULATION SOFTWARE: HOW IT WORKS AND WHY IT MATTERS, Proceedings of the 1997 Winter Simulation Conference.

[7] Muller, K., T. Vignaux, and C. Chui. "SimPy: discrete-event simulation in Python." SimPy Development Team (2014).

[8] Sanner, Michel F. "Python: a programming language for software integration and development." J Mol Graph Model 17, no. 1 (1999): 57-61.

[9] Sterbenz, James PG, Egemen K. Çetinkaya, Mahmood A. Hameed, Abdul Jabbar, Shi Qian, and Justin P. Rohrer. "Evaluation of network resilience, survivability, and disruption tolerance: analysis, topology generation, simulation, and experimentation." Telecommunication systems 52, no. 2 (2013): 705-736.

[10] Python Software Foundation, 6.8 The yield statement (https://docs.python.org/2.4/ref/yield.html) Retrieved May 2017

[11] Efficient and Adaptive Epidemic-Style Protocols for Reliable and Scalable Multicast. Indranil Gupta, Ayalvadi J. Ganesh, Anne-Marie Kermarrec. IEEE Transactions on Parallel and Distributed Systems, vol. 17, no. 7, pp. 593–605, July, 2006.

[12] Epidemic Broadcast Trees. João Leitão, José Pereira, Luís Rodrigues. Proc. 26th IEEE International Symposium on Reliable Distributed Systems.

[13] Matloff, Norm. "Introduction to discrete-event simulation and the simpy language." *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August* 2 (2008): 2009.

[14] Banks, Jerry, JOHN S. CARSON II, and L. Barry. *Discrete-event system simulationfourth edition*. Pearson, 2005.

[15] Tetcos, NetSim Homepage, (http://www.tetcos.com/) Retrieved May 2017

[16] SIMUL8 Homepage, SIMUL8 Corporation (https://www.simul8.com/) Retrieved May 2017