A Framework for Implementing Iterative Algorithms on Distributed Systems

Vikram Mudaliar

Senior Thesis in Computer Engineering

University of Illinois Urbana Champaign

mudalia2@illinois.edu

Spring 2017

Advisor: Nitin Vaidya

Abstract

In this thesis, I build a framework for implementing iterative algorithms by abstracting the code for node communication. This thesis explains the building of said framework using a distributed algorithm and introduces the tools and methods used. I first implemented an algorithm and tested it out on a testbed of 15 Raspberry pi's. After the desired functionality was met, I then went on to proceed with abstracting the code so that similar iterative algorithms could reuse the parts of the code that dealt with inter-node communication and communication link setup. This work is funded in part by the National Science Foundation

Subject Keywords: Optimization, multi-agent systems, fault-tolerance, communication failures

Acknowledgments

I would like to thank all those people who have provided their valuable time and generous help in helping me finish my senior thesis. My deepest gratitude is to my adviser, Prof. Nitin Vaidya. I have been fortunate to have an adviser who gave me constant guidance and resources during the entirety of my research experience. His elucidation of tough subject matter at different stages of my research helped me finish this thesis. I would specially like to thank my research partner, Jihui Yang, for his expertise and help. He was the ideal partner and helped me understand the convoluted aspects of socket programming with ease. I am deeply grateful to him for pushing me into trying out new aspects of programming; I am a significantly better programmer because of it. His company throughout the project was cherished and memorable.

Table of Contents

1.Introduction

In the research area of distributed systems, most of the time, algorithms are specifically developed to determine the behavior of a node - an independent unit that processes work - within a network of such nodes. These nodes are only aware of properties pertaining to itself and its neighbors. Each node has the ability to communicate with its neighbors through some form of message passing system (i.e. broadcast or unicast). Therefore, consider a set of interconnected nodes with some initial value such that after running the algorithm for several rounds, they all are left with the same value. Algorithms that execute certain blocks of code repeatedly are called iterative algorithms. Also consensus is defined as when each node possessing an initial value, follows a distributed strategy to agree on the same value by calculating some function of these initial values. In this thesis we use iterative algorithms to obtain average consensus among the nodes. The purpose of this project is to build and study the behavior of these algorithms, such as those described in [1-5].

The goal of the thesis was to quicken the setup of a network of nodes to study how an algorithm behaves. Often, valuable time is spent on setting up the testbed to run the algorithm on and also finding a means for communication between them. Cutting down on this time, we can help to get to the actual algorithm testing stage much quicker. This framework provides a way for the user to quicken development of such algorithms by abstracting the communication block of the code and testing the algorithm on a network of nodes by remotely uploading the data.

We implement the program in Python for its general ease and use and abstraction. In the interest of speed and minimizing communication overhead, a shared memory approach was chosen to pass messages between the different threads that represented individual nodes. For the nodes themselves we use Raspberry pi's with a 150 Mbps wireless USB network adapter TL-WN727N.

2.Previous Work

In order to further improve the setup time of iterative algorithms, there has been quite a lot of work in the research community. A system [6] to simulate a theoretical network of nodes to study how an algorithm behaves was developed. Real-world constraints like network delay and faulty nodes were not a concern for said project. Therefore, to be more real-world friendly we build upon this idea by configuring a testbed and testing different topological scenarios. As far as consensus (and average consensus) problems go, it has received extensive notice from the research community. The applicability to topics such as modeling of flocking behavior in biological, multi-agent systems, and physical systems [1], [2] makes it an extensively researched topic.

3. Implementation

In this thesis I implement the algorithm described in [5] to test out the framework. As described in there, the algorithm helps to address the problem of achieving average consensus over lossy links. By average consensus, we mean to say that each node will end up with a value which is the average of the all the initial node values. By lossy links, we mean that communication channels between the nodes might be prone to packet loss. We achieve this lossy communication by using broadcast. In Figure 1, we can see the general topology of the testbed. The arrows represent the direction of communication; i.e. a recipient arrow means that a node can only receive information along those channels. Thus, we can see that every node can only send/receive information to/from one other adjacent neighbor node. Each node also receives data from the host. Thus, we simulate topological constraints through this cyclic nature of communication, thereby implementing a ring based routing system between the Raspberry pi's using a neighbor list.

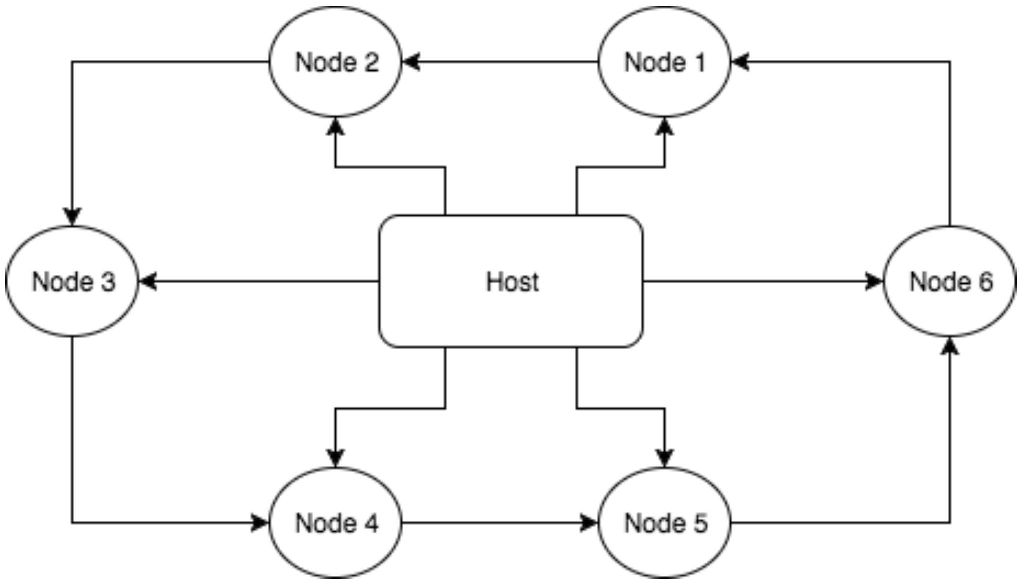


Figure 1. High Level Network Topology

Figure 2 shows the sub files present in each of the node and the host. We split up the implementation on the node side into 3 files. Neighbor_list.txt contains the neighbor list of each node. Comm_socket.py consists of the socket programming and node communication methods and finally Iterative_algo.py consists of the iterative algorithm we are implementing. On the host side, we have two files Upload.py uploads the files remotely to the nodes and compiles them inline. Run.py sends the start signal to begin the algorithm.

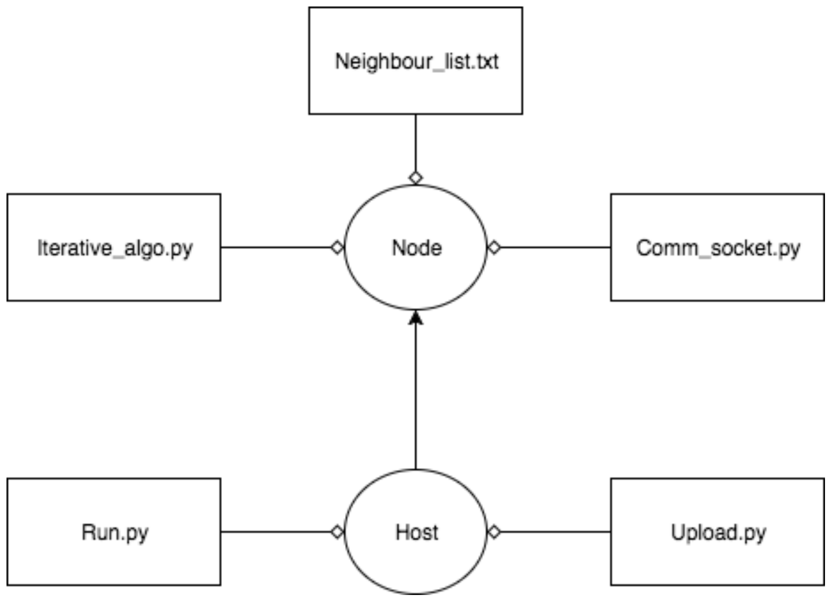
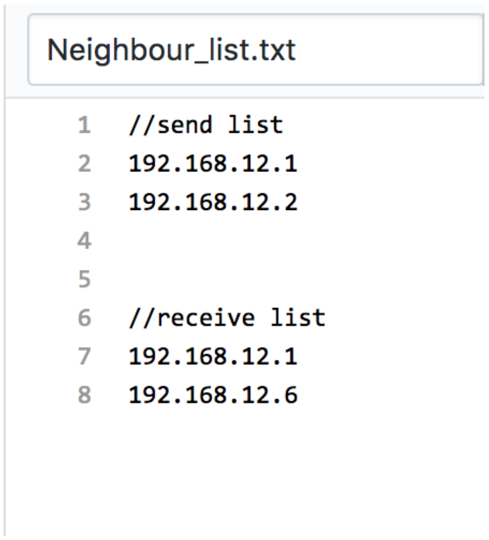


Figure 2. Code components of each entity

3.1 Neighbor_list.txt

This file consists basis for the ring based routing algorithm to work. Essentially, this file governs which neighbors to talk to and which neighbors to listen to. In Figure 3, we have the list for node with IP address 192.168.12.1. There is an oddity in the fact that the node itself appears in both categories, but this is due to the specificities of the algorithm [5] we are implementing.


Neighbour_list.txt

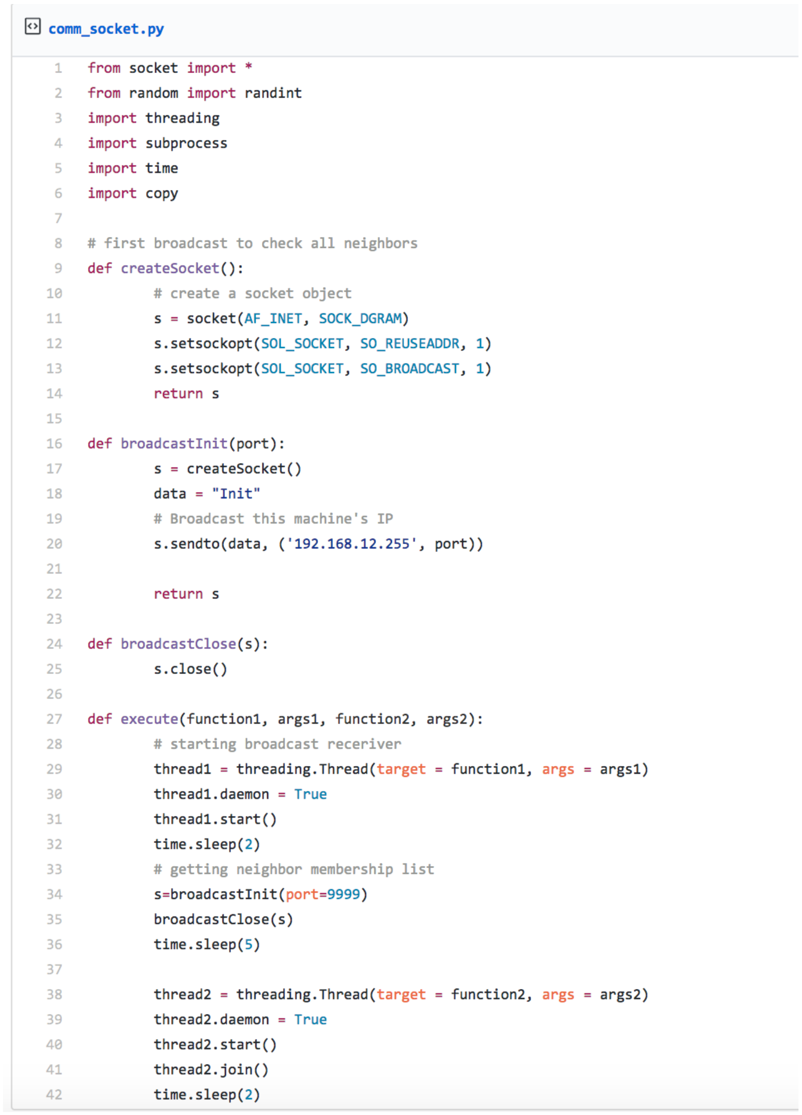
```
//send list
192.168.12.1
192.168.12.2


//receive list
192.168.12.1
192.168.12.6
```

Figure 3. Neighbor_list.txt

In our setup we utilize a ring based network topology. However, since we have a dedicated file to specify whom to receive/send from, in theory any network topology can be obtained by changing the corresponding IP addresses in this file. For example, if I want to recreate a bus topology we would use the host as the common line of communication between the nodes and each node can only send/receive from the host.

3.2 Comm_socket.py

This file contains most of the abstractions we have built in the framework. It can be imported in order to achieve broadcast communication between two nodes. All the user has to do is to call broadcastInit(port) and specify the IP address of the node he wants to send the information to. The file handles all the socket programming instructions such as opening, closing and specifying the type of transmission between sockets.



comm_socket.py

```python
from socket import *
from random import randint
import threading
import subprocess
import time
import copy

# first broadcast to check all neighbors
def createSocket():
        # create a socket object
        s = socket(AF_INET, SOCK_DGRAM)
        s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
        s.setsockopt(SOL_SOCKET, SO_BROADCAST, 1)
        return s

def broadcastInit(port):
        s = createSocket()
        data = "Init"
        # Broadcast this machine's IP
        s.sendto(data, ('192.168.12.255', port))

        return s

def broadcastClose(s):
        s.close()

def execute(function1, args1, function2, args2):
        # starting broadcast receriver
        thread1 = threading.Thread(target = function1, args = args1)
        thread1.daemon = True
        thread1.start()
        time.sleep(2)
        # getting neighbor membership list
        s=broadcastInit(port=9999)
        broadcastClose(s)
        time.sleep(5)

        thread2 = threading.Thread(target = function2, args = args2)
        thread2.daemon = True
        thread2.start()
        thread2.join()
        time.sleep(2)
```

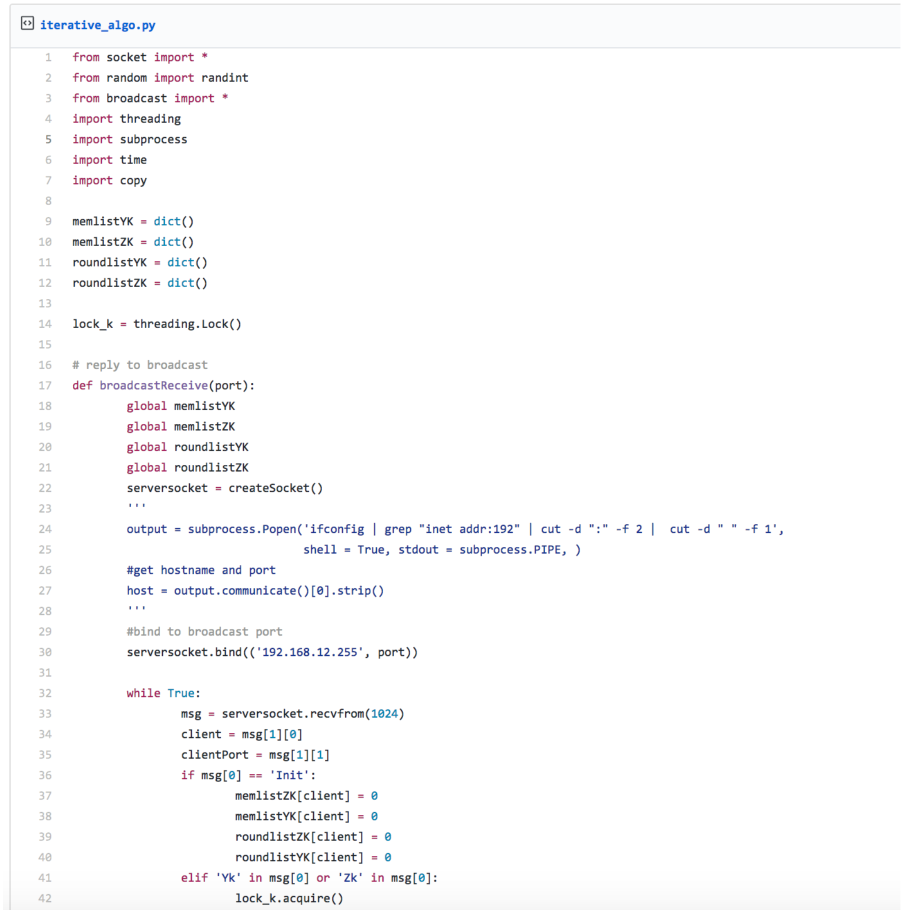
Figure 4.Comm_socket.pyt

3.3 Iterative_algo.py

This file consists of the bulk of the algorithm code to be implemented. This would be the file that the user modifies to implement the core functionality of the iterative steps in the algorithm. Figure 5 shows implementation of the algorithm stated in [5].

broadcastReceive(port) is the function through which data is received by a node

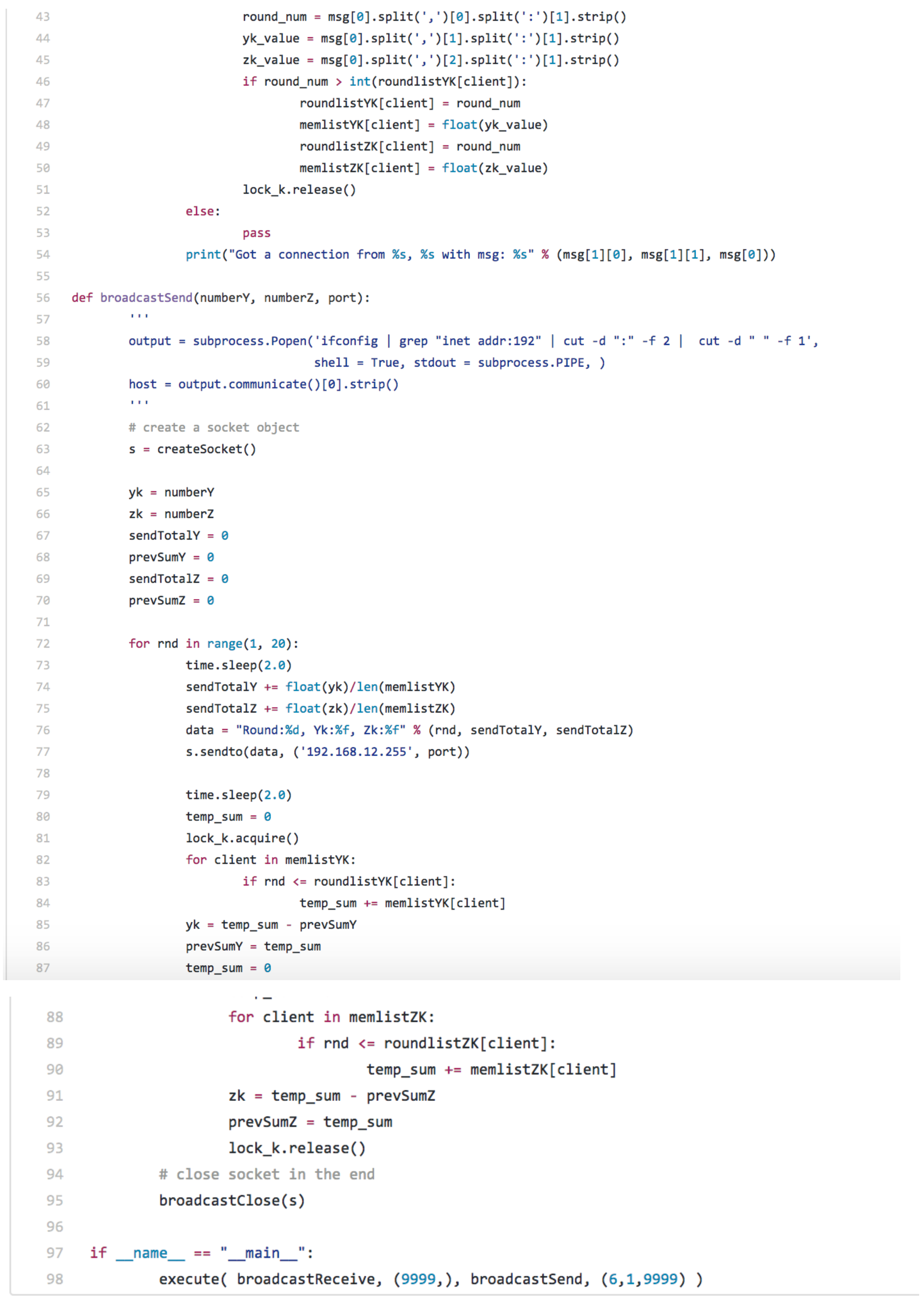
broadcastSend(port) is the function that calculates the new 'node value' by applying some function (i.e. average in our implementation 'line 74&75') on the initial values and transmits this value to other nodes. We also specify how many number of rounds the algorithm should run for in this function.



iterative_algo.py

```
1   from socket import *
2   from random import randint
3   from broadcast import *
4   import threading
5   import subprocess
6   import time
7   import copy
8
9   memlistYK = dict()
10  memlistZK = dict()
11  roundlistYK = dict()
12  roundlistZK = dict()
13
14  lock_k = threading.Lock()
15
16  # reply to broadcast
17  def broadcastReceive(port):
18          global memlistYK
19          global memlistZK
20          global roundlistYK
21          global roundlistZK
22          serversocket = createSocket()
23          '''
24          output = subprocess.Popen('ifconfig | grep "inet addr:192" | cut -d ":" -f 2 |  cut -d " " -f 1',
25                                    shell = True, stdout = subprocess.PIPE, )
26          #get hostname and port
27          host = output.communicate()[0].strip()
28          '''
29          #bind to broadcast port
30          serversocket.bind(('192.168.12.255', port))
31
32          while True:
33                  msg = serversocket.recvfrom(1024)
34                  client = msg[1][0]
35                  clientPort = msg[1][1]
36                  if msg[0] == 'Init':
37                          memlistZK[client] = 0
38                          memlistYK[client] = 0
39                          roundlistZK[client] = 0
40                          roundlistYK[client] = 0
41                  elif 'Yk' in msg[0] or 'Zk' in msg[0]:
42                          lock_k.acquire()
```

Figure 5. iterative_algo.pyt

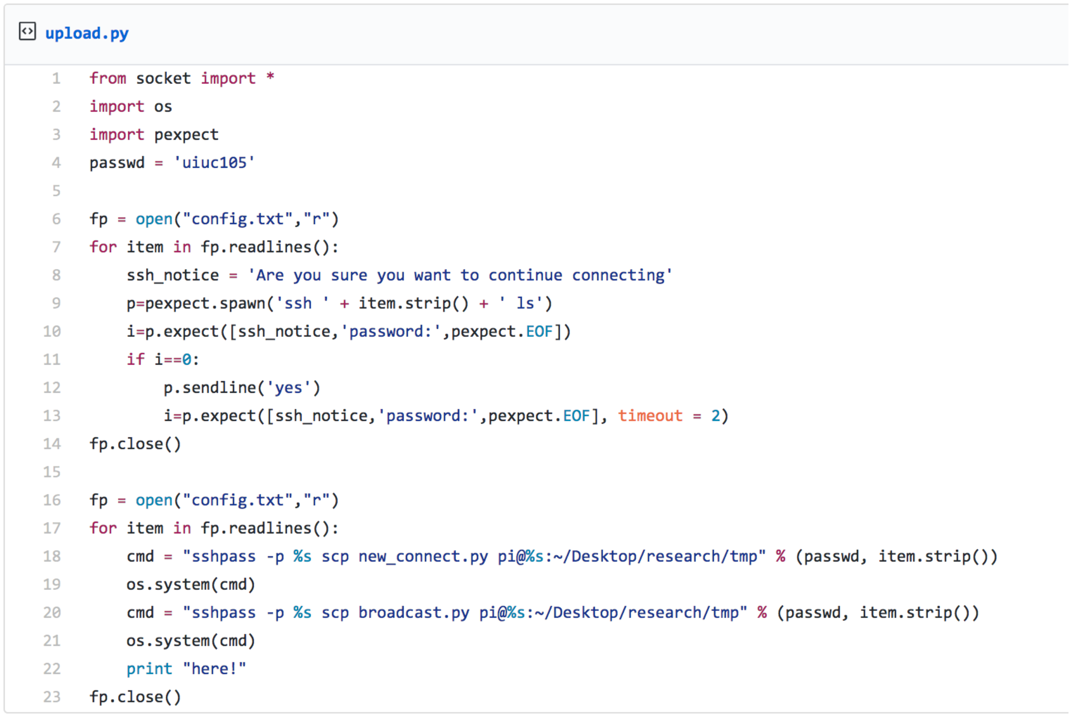


```
                        round_num = msg[0].split(',')[0].split(':')[1].strip()
                        yk_value = msg[0].split(',')[1].split(':')[1].strip()
                        zk_value = msg[0].split(',')[2].split(':')[1].strip()
                        if round_num > int(roundlistYK[client]):
                                roundlistYK[client] = round_num
                                memlistYK[client] = float(yk_value)
                                roundlistZK[client] = round_num
                                memlistZK[client] = float(zk_value)
                        lock_k.release()
                else:
                        pass
                print("Got a connection from %s, %s with msg: %s" % (msg[1][0], msg[1][1], msg[0]))

def broadcastSend(numberY, numberZ, port):
        '''
        output = subprocess.Popen('ifconfig | grep "inet addr:192" | cut -d ":" -f 2 |  cut -d " " -f 1',
                                  shell = True, stdout = subprocess.PIPE, )
        host = output.communicate()[0].strip()
        '''
        # create a socket object
        s = createSocket()

        yk = numberY
        zk = numberZ
        sendTotalY = 0
        prevSumY = 0
        sendTotalZ = 0
        prevSumZ = 0

        for rnd in range(1, 20):
                time.sleep(2.0)
                sendTotalY += float(yk)/len(memlistYK)
                sendTotalZ += float(zk)/len(memlistZK)
                data = "Round:%d, Yk:%f, Zk:%f" % (rnd, sendTotalY, sendTotalZ)
                s.sendto(data, ('192.168.12.255', port))

                time.sleep(2.0)
                temp_sum = 0
                lock_k.acquire()
                for client in memlistYK:
                        if rnd <= roundlistYK[client]:
                                temp_sum += memlistYK[client]
                yk = temp_sum - prevSumY
                prevSumY = temp_sum
                temp_sum = 0
                for client in memlistZK:
                        if rnd <= roundlistZK[client]:
                                temp_sum += memlistZK[client]
                zk = temp_sum - prevSumZ
                prevSumZ = temp_sum
                lock_k.release()
        # close socket in the end
        broadcastClose(s)

if __name__ == "__main__":
        execute( broadcastReceive, (9999,), broadcastSend, (6,1,9999) )
```

Figure 5. continued

3.4 Upload.py

While developing one of the major obstacles encountered was to update the recent version of the code on the different nodes in the network. This process can become very tedious indeed because manually removing the SD-card and updating the recent versions on the Raspberry pi's is not the most optimized solution of doing it. Therefore Upload.py helps us to do this remotely from the host computer. Because Python does not need to compiled before it runs, we send an in-command line statement to run the file. Thereby this greatly reduces the development time and helps achieve quick code revisions on all nodes.



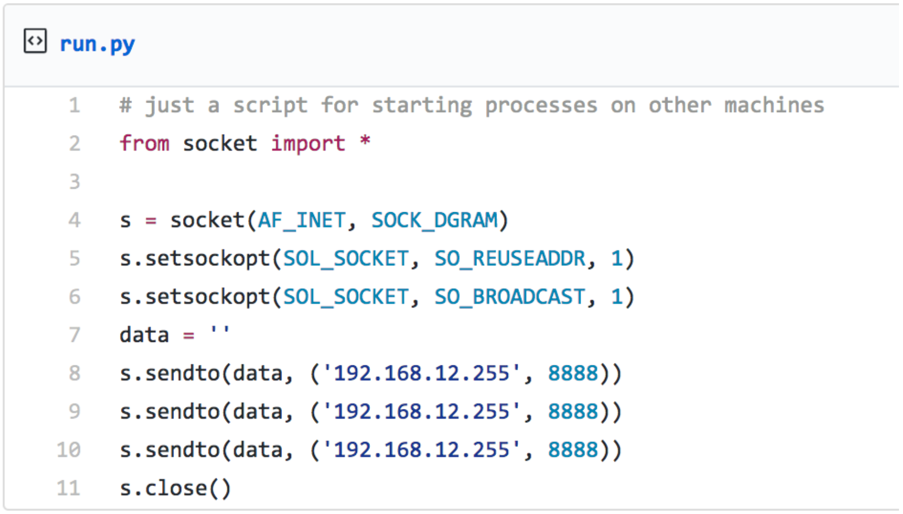
upload.py

```
from socket import *
import os
import pexpect
passwd = 'uiuc105'

fp = open("config.txt","r")
for item in fp.readlines():
    ssh_notice = 'Are you sure you want to continue connecting'
    p=pexpect.spawn('ssh ' + item.strip() + ' ls')
    i=p.expect([ssh_notice,'password:',pexpect.EOF])
    if i==0:
        p.sendline('yes')
        i=p.expect([ssh_notice,'password:',pexpect.EOF], timeout = 2)
fp.close()

fp = open("config.txt","r")
for item in fp.readlines():
    cmd = "sshpass -p %s scp new_connect.py pi@%s:~/Desktop/research/tmp" % (passwd, item.strip())
    os.system(cmd)
    cmd = "sshpass -p %s scp broadcast.py pi@%s:~/Desktop/research/tmp" % (passwd, item.strip())
    os.system(cmd)
    print "here!"
fp.close()
```

Figure 6. Upload.py

3.5 Run.py

The second of the host files, Figure 7. Run.py is used to remotely activate the algorithm on the testbed. Due to the iterative nature of the algorithms, it is a necessity that all of the nodes begin the execution of the code at roughly the same time so as to not miss a round of computation. During the development process we observed that it was very hard to get this timing right, it was practically impossible to do so when we have more than 3 nodes. Therefore, in order to get around this obstacle, the nodes now await a execution signal from the host before starting the algorithm. Run.py sends this execution signal across broadband to all the nodes on the port (8888 in our case).


run.py

```python
# just a script for starting processes on other machines
from socket import *

s = socket(AF_INET, SOCK_DGRAM)
s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
s.setsockopt(SOL_SOCKET, SO_BROADCAST, 1)
data = ''
s.sendto(data, ('192.168.12.255', 8888))
s.sendto(data, ('192.168.12.255', 8888))
s.sendto(data, ('192.168.12.255', 8888))
s.close()
```

Figure 7. Run.py

3.6 Testbed

In Figure 8 we see the setup of the testbed. The white boxes are housings for the Raspberry Pi’s. They have a TpLink TL-WN727N network adapter attached via USB. For the purpose of user interface we have also connected keyboards and monitors. The rightmost node is always marked 1 and the convention observed is incrementation in the anticlockwise direction.

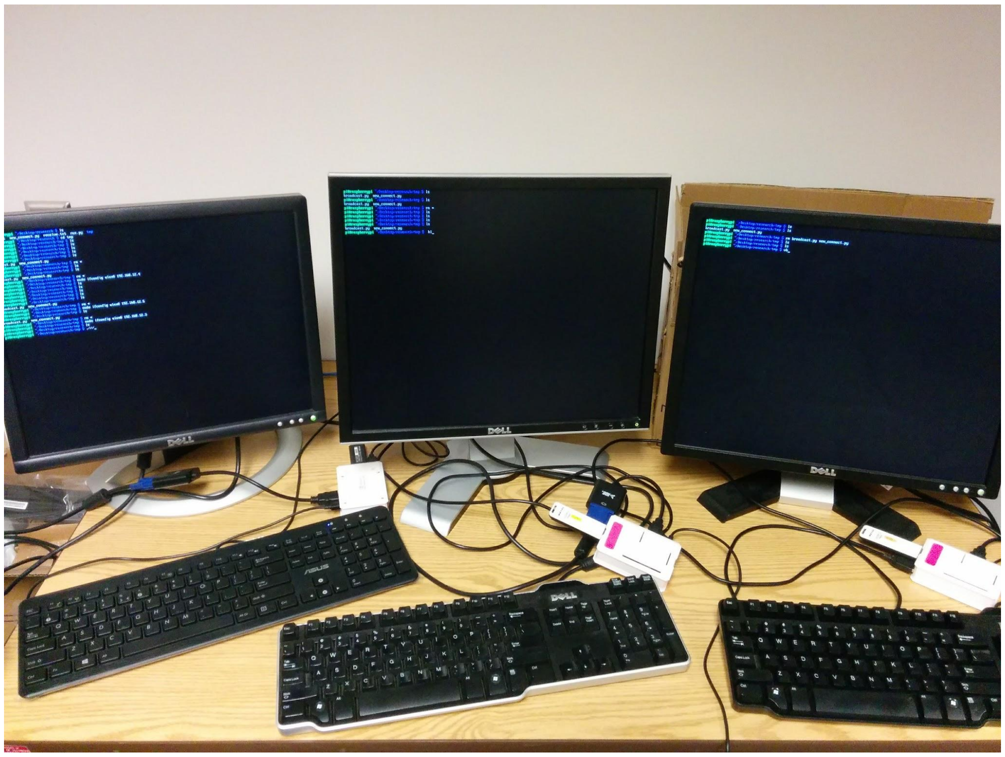


Figure 8. Testbed

4.Conclusion

During this thesis I learned about setting up an ad-hoc network. Then using this network configuration, I learned to send data packets between two computers using Unicast. I then extended the functionality to Broadcast. Using Broadcast, I implemented the average consensus algorithm to over lossy links. Once we got the algorithm to work between two computers, I expanded the testbed onto 15 Raspberry pi’s. Finally, in order to simulate topological constraints, I implemented a ring based routing algorithm in between the pi’s using a neighbor list. In order to make usability easier, a Python script was developed to remotely upload the files through broadcast onto the testbed. Then another script was run to simultaneously run the algorithm on the nodes. In the future, if given the opportunity, I would like to further extend the project by display the different statuses of the nodes on the main host computer. Also the nodes could return the data obtained through the iterations back to the host computer so that it would be easier for somebody to evaluate the test results.

References

[1] Bernadette Charron-Bost, Matthias Függer, Thomas Nowak. *Approximate Consensus in Highly Dynamic Networks: The Role of Averaging Algorithms*, arXiv:1408.0620.

[2] John Duchi, Alekh Agarwal, Martin Wainwright. *Dual Averaging for Distributed Optimization: Convergence Analysis and Network Scaling*, arXiv:1005.2012.

[3] Lili Su, Nitin Vaidya. *Fault-Tolerant Multi-Agent Optimization: Part III*, arXiv:1509.01864.

[4] Lili Su, Nitin H. Vaidya. *Fault-Tolerant Distributed Optimization (Part IV): Constrained Optimization with Arbitrary Directed Networks*, arXiv:1511.01821.

[5] C. N. Hadjicostis, A. D. Dominguez-Garcia and N. H. Vaidya, "Resilient Average Consensus in the Presence of Heterogeneous Packet Dropping Links", *Disc.ece.illinois.edu*, 2012. [Online]. Available: http://disc.ece.illinois.edu/publications.php. [Accessed: 20- Apr- 2017].

[6] S. Peter, "Undergrad Thesis - A system to simulate distributed algorithms (Java Application)", *GitHub*, 2016. [Online]. Available: https://github.com/speter52/GraphSim. [Accessed: 20- Apr- 2017].