The Network Project

How to solve the consensus problem with crash failures

Leader:Haimin Luo Member:Jinhui Zhu, Chao Dong

Date: Jan. 5th 2018

Part 1 Introduction

1.1Purposes

(1)Learn some common failures in distributed system.
(2)Find some algorithms to get consensus in distributed system.
(3)Implement the algorithm to solve the consensus problem while crash failures.

1.2What's consensus problem

The consensus problem requires agreement among a number of processes (or agents) for a single data value. Some of the processes (agents) may fail or be unreliable in other ways, so consensus protocols must be fault tolerant or resilient. The processes must somehow put forth their candidate values, communicate with one another, and agree on a single consensus value.

The consensus problem is a fundamental problem in control of multi-agent systems. One approach to generating consensus is for all processes (agents) to agree on a majority value. In this context, a majority requires at least one more than half of available votes (where each process is given a vote). However one or more faulty processes may skew the resultant outcome such that consensus may not be reached or reached incorrectly.

Protocols that solve consensus problems are designed to deal with limited numbers of faulty processes. These protocols must satisfy a number of requirements to be useful. For instance a trivial protocol could have all processes output binary value 1. This is not useful and thus the requirement is modified such that the output must somehow depend on the input. That is, the output value of a consensus protocol must be the input value of some process. Another requirement is that a process may decide upon and output a value only once and this decision is irrevocable. A process is called correct in an execution if it does not experience a failure. A consensus protocol tolerating halting failures must satisfy the following properties.[1]

Termination:Every correct process decides some value.

Validity:If all processes propose the same value v, then all correct processes decide v.

Integrity:Every correct process decides at most one value, and if it decides some value v, then v must have been proposed by some process.

Agreement:Every correct process must agree on the same value.

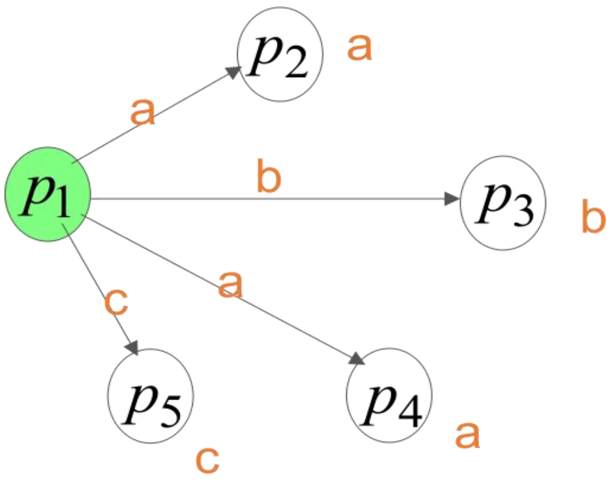
A protocol that can correctly guarantee consensus amongst n processes of which at most t fail is said to be t-resilient.

In evaluating the performance of consensus protocols two factors of interest are running time and message complexity. Running time is given in Big O notation in the number of rounds of message exchange as a function of some input parameters (typically the number of processes and/or the size of the input domain). Message complexity refers to the amount of message traffic that is generated by the protocol. Other factors may include memory usage and the size of messages.[2]

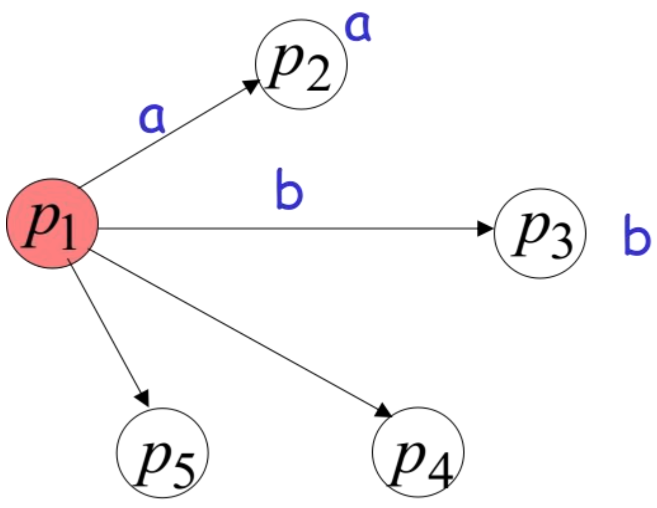
1.3What's crash Failure

In distributied system, some of the processors (agents) may stops taking steps at some pointer. These processors (agents) can't work any more and other processors can't receive the imfomation to get the consensus. This is the crash failure.

The picture 1-1-a and 1-1-b is the results of information transfer of non-faulty processor and faulty processor.



P1-1-a Non-faulty Processor



P1-1-a Faulty Processor

Part 2 Design

2.1Algorithm

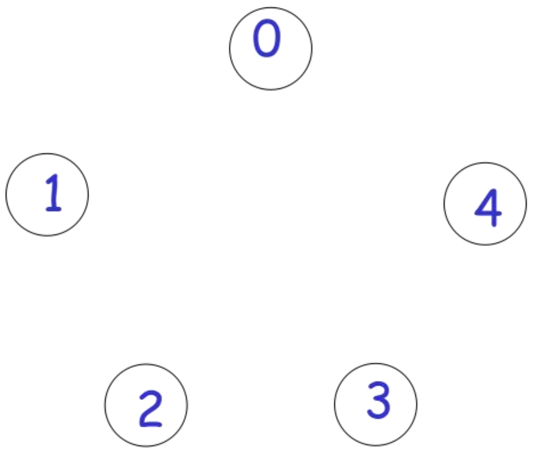
In this project, we want to get consensus with Crash Failures.

We find an f-resilient algorithm to get consensus by save the minimun value transferred by each other.

For each processor, this algorithm includes three steps:

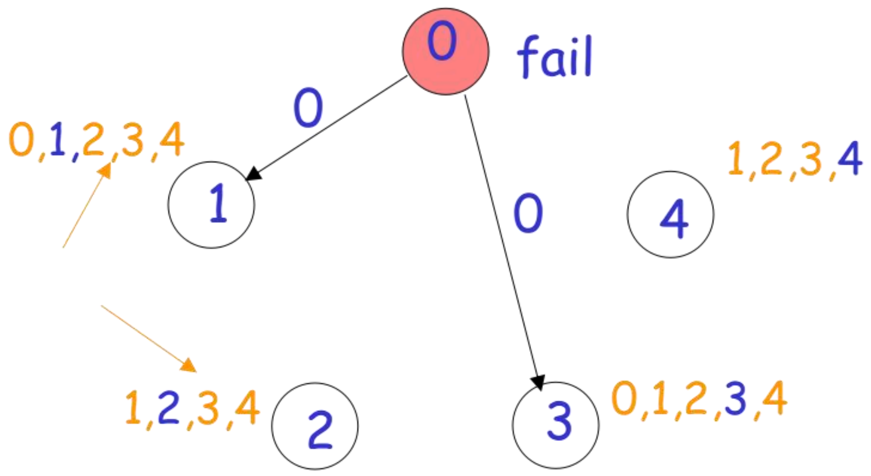
```
Round 1:
    Broadcast my value
Round 2 to round f+1:
        Broadcast any new received values
End of round f+1:
        Decide on the minimum value received
```

There are 5 Node in picture 2-1. It means there are 5 processors in the distributed system. Initially, each processor hold a random value.



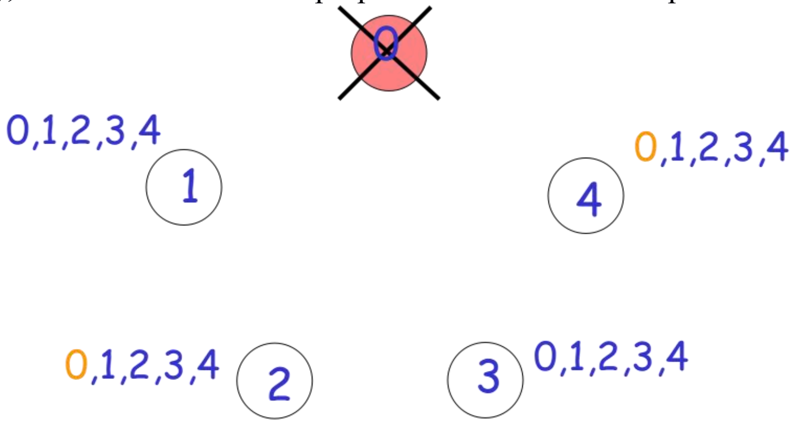
P2-1 Init of the distributed system

Then each processor transfer their value to the others.We assume at round 1, processor 0 crashes and can't work. And before the processor crashed, it sends message for processer 1, processer 3. Just like picture 2-2 shows.The processors save the value they received.



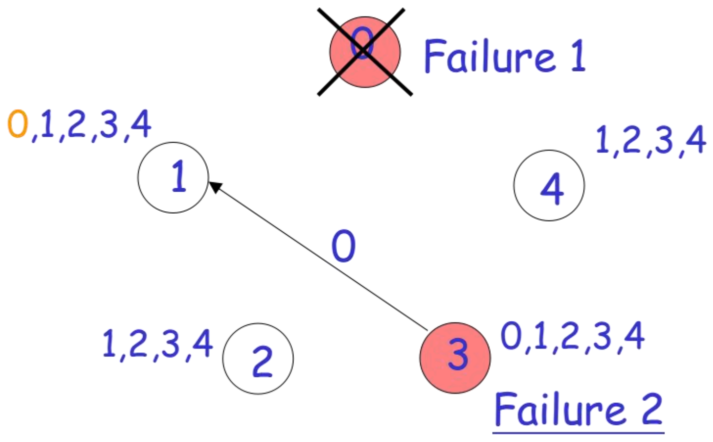
P2-2 Round 1

Then at round 2, transfer value. We can find each processor hold the same sequence.We choose the min-value of the sequence to be the result. And obviously, this result satisfies the properties about consensus problem.



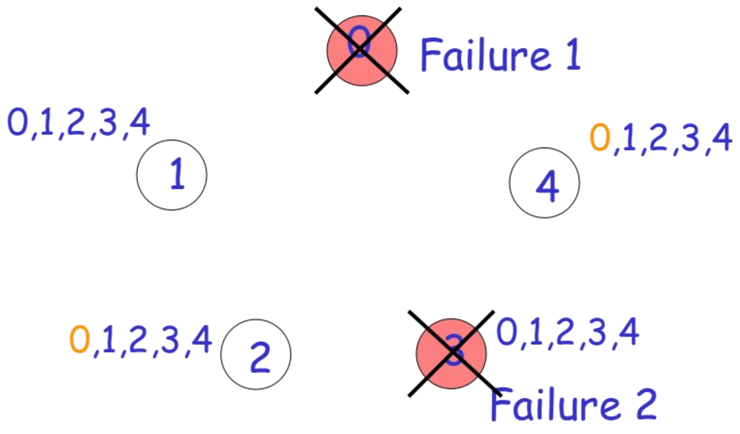
P2-3 Round 2

What happens when some node crashes at round 2? Picture 2-4 shows, the result is no consensus.



P2-4 crash Failure in Round 2

So we need one other round more. At round 3, the result is consensus.



P2-5 Round 3

We get a vital lemma:

In the algorithm, at the end of the round with no failure, all the processors know the same set of values.

Proof: For the sake of contradiction, assume the claim is false. Let x be a value which is known only to a subset of (non-faulty) processors. But when a processor knew x for the first time, in the next round it broadcasted it to all. So, the only possibility is that it received it right in this round, otherwise all the others should know x as well. But in this round there are no failures, and so x must be received by all.

Then, at the end of the round with no failure, Every (non-faulty) processor knows about all the values of all other participating processors.

This knowledge doesn't change until the end of the algorithm

Therefore, at the end of the round with no failure, everybody would decide the same value. However, we don't know the exact position of this round, so we have to let the algorithm execute for n+1 rounds.

2.2Performance

Assume there are n processors in which f processors are faulty.($n > f$).

In this algorithm, it runs n + 1 rounds totally. At each round, a processors send its value to n - 1 processors.

We define the complexity is T.

$T = O(n * n * n) = O(n^3)$

This algorithm will finish in polynomial time.

Part 3 Implementation

3.1Document introduction

crash_tolerant.py: the source code of the project which should run in the enviroment of Python 3.6 and above.

runc.sh: a shell script contains runing command and some parameters. This script should run in Linux system or Mac.

3.2How to run

Run the shell script "runc.sh" in terminal, and input 2 parameter 's' and 'e'which is the range of the random value. The value is a random intager between s and e.

For example, we input "./runc.sh 200 3000". It means each processers' value is a random value between 200 to 3000.Then the result and some information will print in the screen.

3.3Implementation

In this project, we defined some threads to represent the processors.

```
class ThreadNode(object) :

    def __init__(self, func , addr, val, targets, index, fail, name = '') :
        self.skt = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
        self.addr = addr
        self.func = func
        self.args = (self.skt,val,targets, index, fail)
```

Code 3-1 The definition of thread

Then we define a function belongs each thread to send and receive message.For convenience, we assuming at each round only one processer may crash.

Code 3-2 Distribute message in a Round

```
    while True:
        if round >= len(targets):
        #when the number of round greater then the number of node, break the loop.
            break

        #send my value to others
        for i in range(len(targets)):
            if i == idx : continue
            skt.sendto(pickle.dumps(fst),targets[i])

            #simulate the crashing
            if fail[round] != None:
                if fail[round] == idx and i == fail[round] + 5:
                    print("Node %d crash during round %d!" % (idx,round))
                    return

        #keep listening until receive enough message
        if len(rcvd) < len(targets):
            while True:
                t, addr = skt.recvfrom(4096)
                rcvd.append(pickle.loads(t))
                fst = min(rcvd)
                if len(rcvd) == len(targets): break
        round = round + 1
    print("index: %d, val: %d, round: %d" % (idx, fst,round))
```

Part 4 Result

Input:

```
./runc.sh 200 3000
```

Output:

```
crash list :
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Node's original values :
[512, 2410, 1816, 1983, 732, 1324, 1973, 248, 311, 1615, 2601, 1864, 1686, 531, 1792,
2836, 863, 2666, 1351, 492, 1748, 255, 555, 2303, 473, 1184, 1252, 1608, 722, 2935, 619,
2655, 2205, 2510, 1201]

The smallest is: 248

Start...
Threadode 0 start with ip 127.0.0.1:5000 with value 512
Threadode 1 start with ip 127.0.0.1:5001 with value 2410
Threadode 2 start with ip 127.0.0.1:5002 with value 1816
Threadode 3 start with ip 127.0.0.1:5003 with value 1983
Threadode 4 start with ip 127.0.0.1:5004 with value 732
Threadode 5 start with ip 127.0.0.1:5005 with value 1324
Threadode 6 start with ip 127.0.0.1:5006 with value 1973
Threadode 7 start with ip 127.0.0.1:5007 with value 248
Threadode 8 start with ip 127.0.0.1:5008 with value 311
Threadode 9 start with ip 127.0.0.1:5009 with value 1615
Threadode 10 start with ip 127.0.0.1:5010 with value 2601
Threadode 11 start with ip 127.0.0.1:5011 with value 1864
Threadode 12 start with ip 127.0.0.1:5012 with value 1686
Threadode 13 start with ip 127.0.0.1:5013 with value 531
Threadode 14 start with ip 127.0.0.1:5014 with value 1792
Threadode 15 start with ip 127.0.0.1:5015 with value 2836
Threadode 16 start with ip 127.0.0.1:5016 with value 863
Threadode 17 start with ip 127.0.0.1:5017 with value 2666
Threadode 18 start with ip 127.0.0.1:5018 with value 1351
Threadode 19 start with ip 127.0.0.1:5019 with value 492
Threadode 20 start with ip 127.0.0.1:5020 with value 1748
Threadode 21 start with ip 127.0.0.1:5021 with value 255
Threadode 22 start with ip 127.0.0.1:5022 with value 555
Threadode 23 start with ip 127.0.0.1:5023 with value 2303
Threadode 24 start with ip 127.0.0.1:5024 with value 473
Threadode 25 start with ip 127.0.0.1:5025 with value 1184
Threadode 26 start with ip 127.0.0.1:5026 with value 1252
Threadode 27 start with ip 127.0.0.1:5027 with value 1608
Threadode 28 start with ip 127.0.0.1:5028 with value 722
Threadode 29 start with ip 127.0.0.1:5029 with value 2935
Threadode 30 start with ip 127.0.0.1:5030 with value 619
Threadode 31 start with ip 127.0.0.1:5031 with value 2655
Threadode 32 start with ip 127.0.0.1:5032 with value 2205
Threadode 33 start with ip 127.0.0.1:5033 with value 2510
Threadode 34 start with ip 127.0.0.1:5034 with value 1201

Communicating...

Result...
Node 0 crash during round 0!
Node 1 crash during round 1!
Node 2 crash during round 2!
Node 3 crash during round 3!
Node 4 crash during round 4!
Node 5 crash during round 5!
Node 6 crash during round 6!
Node 7 crash during round 7!
Node 8 crash during round 8!
Node 9 crash during round 9!
Node 10 crash during round 10!
index: 20, val: 248, round: 35
index: 27, val: 248, round: 35
index: 19, val: 248, round: 35
index: 22, val: 248, round: 35
index: 23, val: 248, round: 35
index: 14, val: 248, round: 35
index: 24, val: 248, round: 35
index: 15, val: 248, round: 35
index: 33, val: 248, round: 35
index: 26, val: 248, round: 35
index: 16, val: 248, round: 35
index: 32, val: 248, round: 35
index: 18, val: 248, round: 35
index: 12, val: 248, round: 35
index: 31, val: 248, round: 35
index: 13, val: 248, round: 35
index: 17, val: 248, round: 35
index: 34, val: 248, round: 35
index: 28, val: 248, round: 35
```

```
index: 21, val: 248, round: 35
index: 29, val: 248, round: 35
index: 25, val: 248, round: 35
index: 11, val: 248, round: 35
index: 30, val: 248, round: 35

End...
```

Part 5 Discussion

While makeingthe the project, we have some problems.

First, because of the few experience of code multi-threading programme, we make some mistakes to result in deadlock.

Second, for convience, we simplify the problem. We assume there are at most 1 node crashing at each round. The crashing node has already defined before the programme runnig.

Third, the time is so limited for us to finish the project perfectly on time. At first, we want to make some projects about the Byzantium Problem but when the deadline is coming, we change our mind and choose a easy title.

Part 6 Division of Labour

Haimin Luo(hainesluo@gmail.com): Design and implement the core code.
Jinhui Zhu(huieric2015@gmail.com): Design and implement the GUI,
Chao Dong(948914729@qq.com): Make the report.

Part 7 Reference

[1]:Coulouris, George; Jean Dollimore; Tim Kindberg (2001), Distributed Systems: Concepts and Design (3rd Edition), Addison-Wesley, p. 452, ISBN 0201-61918-0;
[2]:Wikipedia:Consensus (computer science)
[3]:An unnamed ppt shared by monitor.

Part 8 Appendix

Core code

```
# This Python file uses the following encoding: utf-8

###########################################
#This programme need three parameter, the first and second are the range of the
value.
#The third is the IP addresses.
###########################################

import threading,sys,socket,pickle,time,random
from Queue import Queue

#define the thread
class ThreadNode(object) :

    def __init__(self, func , addr, val, targets, index, fail, name = '') :
        self.skt = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
        self.addr = addr
        self.func = func
        self.args = (self.skt,val,targets, index, fail)
        self.skt.bind(addr)
        print("Threadode %d start with ip %s:%s with value %d" %
(index,addr[0],addr[1],val))

    def __call__(self):
        apply(self.func,self.args)

#define the replying function
def reply(skt, val, targets, index, fail):
    fst = val      #Initial value
    idx = index #The index of the thread
    rcvd = [fst]#The sequence of all received value. Initialize by its value.
    round = 0
    while True:
        if round >= len(targets):
        #when the number of round greater then the number of node, break the
loop.
            break

        #send my value to others
        for i in range(len(targets)):
            if i == idx : continue
```

```
                        skt.sendto(pickle.dumps(fst),targets[i])

                        #simulate the crushing
                        if fail[round] != None:
                                if fail[round] == idx and i == fail[round] + 5:
                                        print("Node %d crush during round %d!" % (idx,round))
                                        return

                #keep listening until receive enough message
                if len(rcvd) < len(targets):
                        while True:
                                t, addr = skt.recvfrom(4096)
                                rcvd.append(pickle.loads(t))
                                fst = min(rcvd)
                                if len(rcvd) == len(targets): break
                round = round + 1
        print("index: %d, val: %d, round: %d" % (idx, fst,round))

#IP partition
def toAddr(s):
        addr = s.split(":")
        return addr[0], int(addr[1])

#create the random sequence which contains the initial value of each node.
def random_int_list(start, stop, length):
    start, stop = (int(start), int(stop)) if start <= stop else (int(stop), int(start))
    length = int(abs(length)) if length else 0
    random_list = []
    for i in range(length):
        random_list.append(random.randint(start, stop))
    return random_list

def main():
        fail = [0,1,2,3,4,5,6,7,8,9,10]                 #The crushed node
        threads = []                                            #sequence of thread.Initialized by
none.

        #input the parameter
        start = int(sys.argv[1])
        stop = int(sys.argv[2])
        nodes = map(toAddr,sys.argv[3:])               #IP sequence
        values = random_int_list(start,stop,len(nodes))#Initial value
        print("Crush list : ")
        print(fail)
        for i in range(len(fail),len(nodes)):
                fail.append(None)
        print("\nNode's original values : ")
        print(values)
        print("\nThe smallest is: %d\n" % min(values))

        print("Start...")
        #create the threads
        for i in range(len(nodes)):
                t = threading.Thread(target =
ThreadNode(reply,nodes[i],values[i],nodes,i,fail,reply.__name__))
                threads.append(t)

        print("\nCommunicating...")

        print("\nResult...")
        for t in threads:
                t.start()

        #wait for the end of those thread
        for t in threads:
                t.join()

        print("\nEnd...")

if __name__ == '__main__':
        main()
```