ABSTRACT

Mutual exclusion is a fundamental paradigm in computing. Over the past two decades, several algorithms have been proposed to achieve mutual exclusion in asynchronous distributed message-passing systems. Designing such algorithms becomes difficult when the requirement for "fair" synchronisation needs to be satisfied. The commonly accepted definition of fairness is that requests for access to the critical section (CS) are satisfied in the order of their logical timestamps. If two requests have the same timestamp, the pid is used as a tie-breaker. Lamport's clock is used to assign timestamps to messages to order the requests. The algorithm to update the clocks and to timestamp requests keeps all logical clocks closely synchronised. A fair mutual exclusion algorithm needs to guarantee that requests are accessed in increasing order of the timestamps. Of the many distributed mutual exclusion algorithms, the only algorithms that are fair in the above context are Lamport, Ricart-Agrawala and Lodha-Kshemkalyani.[1] In this paper, we will discuss only the former two. The rest of the paper is organised as follows: Section 1 presents the technologies used. Section 2 presents the work done and how it is implemented. Section 3 is the code of the entire implementation. Section 4 comprises of the results and snapshots of the output. Finally, Section 5 concludes the work of this paper.

TABLE OF CONTENTS

TECHNOLOGIES USED

The technologies used to implement the code are as follows:

Python 2.7

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasises readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Since there is no compilation step, the edit-test-debug cycle is incredibly fast. Debugging Python programs is easy: a bug or bad input will never cause a segmentation fault. Instead, when the interpreter discovers an error, it raises an exception. When the program doesn't catch the exception, the interpreter prints a stack trace. A source level debugger allows inspection of local and global variables, evaluation of arbitrary expressions, setting breakpoints, stepping through the code a line at a time, and so on. The debugger is written in Python itself, testifying to Python's introspective power. On the other hand, often the quickest way to debug a program is to add a few print statements to the source: the fast edit-test-debug cycle makes this simple approach very effective.

Collections and Logging in Python

This module implements specialized container datatypes providing alternatives to Python’s general purpose built-in containers, dict, list, set, and tuple. Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

collections.namedtuple(typename,field_names,*,verbose=False,rename=False,module=None)

Returns a new tuple subclass named typename.

This module defines functions and classes which implement a flexible error logging system for applications.

Logging is performed by calling methods on instances of the Logger class (hereafter called loggers). Each instance has a name, and they are conceptually arranged in a namespace hierarchy using dots (periods) as separators. For example, a logger named "scan" is the parent of loggers "scan.text", "scan.html" and "scan.pdf". Logger names can be anything you want, and indicate the area of an application in which a logged message originates.

WORK DONE

The paper[1] "Performance of fair distributed mutual exclusion algorithms" has been implemented as a part of the DCS project. The paper proposes the advantages of a fair distributed mutual exclusion algorithm called Lodha-Kshemkalyani over Ricart-Agrawala. This algorithm has been implemented using Python with collections and logging. The performance metrics for mutual exclusion algorithms are the number of messages, the synchronisation delay, the response time, and the waiting time.

The Ricart-Agrawala algorithm is used to ensure that mutex blocks are allocated fairly to processes. The algorithm works as follows.

When a process wants to acquire the mutex, its sends a time-stamped mutex-request message to all other processes and waits until it has received mutex-acknowledgement messages from all processes. Only then does the process enter the mutex block.

Whenever a process *p1* receives a mutex request message from a process *p2*, it immediately sends a mutex acknowledgement message to *p2* unless one of two conditions is satisfied:

- The process *p1* is interested in acquiring the mutex itself - in this case, *p1* sends a mutex acknowledgement if the time-stamp of *p2*'s mutex request is earlier than the time in process *p1*.

- The process *p1* is currently inside of a mutex - in this case, *p1* queues the mutex request and sends it when it exits the mutex.

Lamport's logical clock is used to synchronise the time between the processes. Whenever a process performs an action (print, send, receive), it increments its time. Whenever a process *p1* sends a message to a process *p2*, *p1* attaches its time to the message. When *p2* receives the message, it will update its own time to the time at *p1* if that time is larger. The time at any process is thus the number of events that happened so far in the system that the process is aware of.

Implementation details

Processes are represented by a sequence of actions parsed from the system configuration file.

A scheduler polls all processes in a round-robin manner. Whenever a process is polled, the process can attempt to execute exactly one action. If the action is successful (e.g. a print inside of a mutex block), the process de-queues the action. If the process was unable to execute the action (e.g. a receive before the message has been sent), it will try to perform it next time it is polled.

A process can send a message to a recipient by putting the message into a shared message queue (every process has a pointer to this queue). When the addressee attempts to receive the message, it checks the queue and retrieves the message from there if possible.

Similarly every process has access (through a pointer) to a shared queue for mutex requests and acknowledgements. Every process sends any applicable mutex acknowledgements whenever it is polled, before attempting to execute its next action.

No process can perform an action unless it has acquired the mutex or no other process has the mutex. The implementation thus assumes that every process is aware of the other processes.

CODE

This module implements a simulation of two distributed algorithms:

- Lamport's logical clock for synchronisation

- The Ricart-Agrawala algorithm for fair mutual exclusion

The module can be run from the command line as follows: ./simulator.py input

Where *input* is the path to a file that specifies the processes that take part in the distributed system and the actions they perform.

The module takes as input a configuration file that specifies the distributed system to simulate. The configuration file contains processes and the operations they perform.

The format of a process with identifier *pid* is:

```
```

```
  begin process pid
    operation
    ...
  end process
```

The operations that a process can perform are the following:

- `send pid msg` to send the message *msg* to the process *pid*

- `recv pid msg` to receive the message *msg* from the process *pid*

- `print msg`   to print the message *msg* to the terminal

In addition to the basic operations, a mutex block can be specified as follows:

```
  begin mutex
    operation
    ...
  end mutex
```

All operations inside of a mutex block will be executed without any other

process interfering.

The code is put into one file:

simulator.py - This code takes the input file of the specified format and runs the processes. It generates an output executable that shows us which process was run first and in which order.

simulator.py

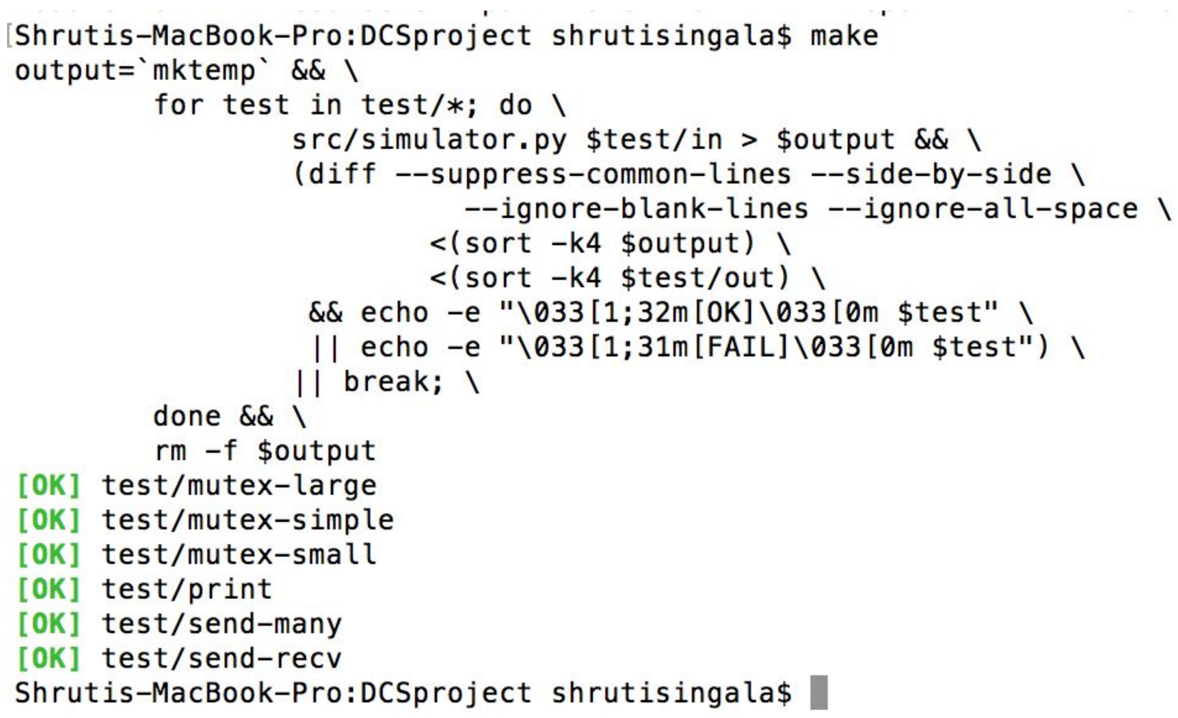
```
from collections import deque, namedtuple
import logging


SendAction = namedtuple('SendAction', 'destination payload')
ReceiveAction = namedtuple('ReceiveAction', 'sender payload')
PrintAction = namedtuple('PrintAction', 'payload')
MutexStartAction = namedtuple('MutexStartAction', '')
MutexEndAction = namedtuple('MutexEndAction', '')

Packet = namedtuple('Packet', 'sender destination payload time')
MutexReq = namedtuple('MutexReq', 'sender time')
MutexAck = namedtuple('MutexAck', 'sender destination')


class Process(object):
    def __init__(self, pid, actions, network, message_channel, mutex_channel):
        self.pid = pid
        self.actions = actions
        self.network = network
        self.message_channel = message_channel
        self.mutex_channel = mutex_channel
        self.clock = 0
        self.deferred_mutex_requests = set()
        self.mutex_req = None
        self.has_mutex = False

    def __repr__(self):
        return 'Process(pid={pid}, time={time})'.format(
            pid=self.pid, time=self.clock)

    def is_done(self):
        return len(self.actions) == 0

    def can_execute(self):
        # a process can execute an action if it has the mutex
        # or if no other process has the mutex
        other_mutex = any(process.has_mutex for process in self.network)
        return self.has_mutex or not other_mutex

    def execute_next(self):
        # respond to mutex requests
        self.handle_mutex_requests()
        # skip a turn if a different process has mutex or if the process
        # doesn't have any actions left to perform
        if self.is_done() or not self.can_execute():
            return
        # try to execute the next action
        # if the action was successful remove it from the action queue
        # otherwise try again next time
        if self.handle_action(self.actions[0]):
            self.actions.popleft()

    def lower_priority(self, mutex_req):
        # processes that asked for the mutex lock earlier or have a lower
        # process id have priority
        if self.clock > mutex_req.time:
            return True
        if self.clock < mutex_req.time:
            return False
        return self.pid > mutex_req.sender

    def handle_mutex_requests(self):
        # if the process doesn't need mutex or the process has lower priority
        # then approve all mutex requests
        # otherwise remember the mutex requests so that they can be approved
        # when the process releases the mutex
        reqs = (msg for msg in self.mutex_channel
                if isinstance(msg, MutexReq) and msg.sender != self.pid)
        acks = set()
        for req in reqs:
            ack = MutexAck(sender=self.pid, destination=req.sender)
            if self.mutex_req is None or self.lower_priority(req):
                acks.add(ack)
            else:
                self.deferred_mutex_requests.add(ack)
        self.mutex_channel.update(acks)

    def handle_action(self, action):
        if isinstance(action, SendAction):
            # send a message by putting it into the message pool
            self.clock += 1
            packet = Packet(
                sender=self.pid,
                destination=action.destination,
                payload=action.payload,
                time=self.clock)
            self.message_channel.append(packet)
            print 'sent {pid} {payload} {destination} {time}'.format(
                pid=self.pid, payload=packet.payload,
                destination=packet.destination, time=self.clock)
            return True
```

```
elif isinstance(action, ReceiveAction):
    # try to receive a message
    for i in reversed(xrange(len(self.message_channel))):
        packet = self.message_channel[i]
        if packet.sender != action.sender:
            continue
        if packet.destination != self.pid:
            continue
        if packet.payload != action.payload:
            continue
        self.clock = max(self.clock, packet.time) + 1
        print 'received {pid} {payload} {sender} {time}'.format(
            pid=self.pid, payload=packet.payload,
            sender=packet.sender, time=self.clock)
        del self.message_channel[i]
        return True
    # honor the happens-before relationship: wait for the message to
    # get sent before trying to receive it
    else:
        return False

elif isinstance(action, PrintAction):
    # use the shared printer
    self.clock += 1
    print 'printed {pid} {payload} {time}'.format(
        pid=self.pid, payload=action.payload, time=self.clock)
    return True

elif isinstance(action, MutexStartAction):
    # send a new mutex request
    if self.mutex_req is None:
        self.mutex_req = MutexReq(sender=self.pid, time=self.clock)
        self.mutex_channel.add(self.mutex_req)
    acks = set(msg for msg in self.mutex_channel
               if isinstance(msg, MutexAck)
               and msg.destination == self.pid)
    # wait until all processes have responded to the request
    if len(acks) != len(self.network) - 1:
        return False
    # acquire the mutex if all processes have responded to the request
    self.has_mutex = True
    self.mutex_channel.remove(self.mutex_req)
    return True

elif isinstance(action, MutexEndAction):
    # release mutex lock and respond to all deferred mutex requests
    self.mutex_channel.update(self.deferred_mutex_requests)
```

```
            self.mutex_req = None
            self.has_mutex = False
            self.deferred_mutex_requests = set()
            return True


def parse_input(infile, **kwargs):
    network = kwargs.get('network', set())
    message_channel = kwargs.get('message_channel', deque())
    mutex_channel = kwargs.get('mutex_channel', set())
    process_name = None
    process_actions = deque()
    lines = (line.lower().strip() for line in infile if line.strip())
    for lineno, line in enumerate(lines, start=1):
        tokens = line.split()
        try:
            if line.startswith('begin process'):
                process_name = tokens[2]
            elif line.startswith('end process'):
                network.add(Process(
                    pid=process_name,
                    actions=process_actions,
                    network=network,
                    message_channel=message_channel,
                    mutex_channel=mutex_channel))
                process_name = None
                process_actions = deque()
            elif line.startswith('begin mutex'):
                process_actions.append(MutexStartAction())
            elif line.startswith('end mutex'):
                process_actions.append(MutexEndAction())
            elif line.startswith('send'):
                process_actions.append(SendAction(tokens[1], tokens[2]))
            elif line.startswith('recv'):
                process_actions.append(ReceiveAction(tokens[1], tokens[2]))
            elif line.startswith('print'):
                process_actions.append(PrintAction(tokens[1]))
            else:
                logging.warning(
                    'ignoring unknown command: "{command}" '
                    'on line {lineno} in file {filepath}'.format(
                        command=line, lineno=lineno, filepath=infile.name))
        except IndexError:
            logging.warning(
                'ignoring misformed command: "{command}" '
                'on line {lineno} in file {filepath}'.format(
                    command=line, lineno=lineno, filepath=infile.name))
    return network
```

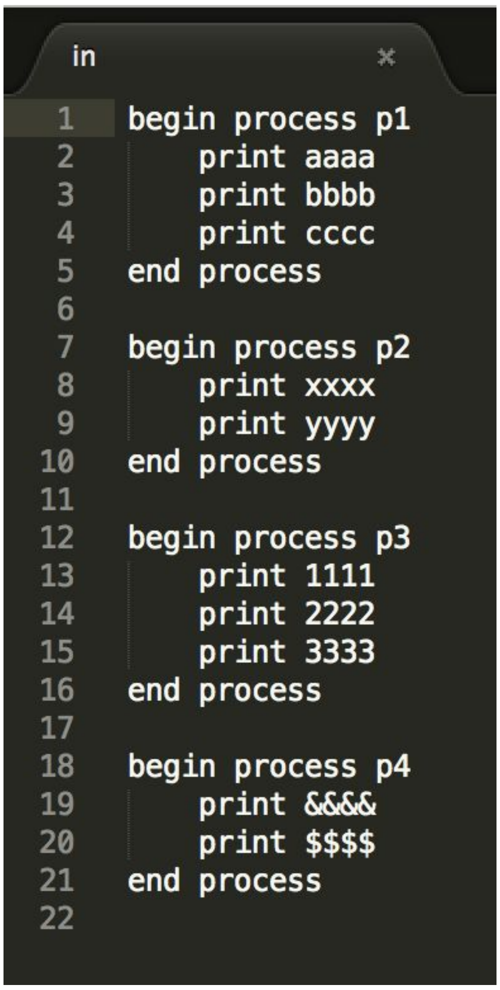
```
def run_processes(network):
    # use a process manager to run the processes
    # processes are polleod in a round-robin manner until all are teminated
    while not all(process.is_done() for process in network):
        for process in network:
            process.execute_next()


def main(infile):
    network = parse_input(infile)
    run_processes(network)


if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(
        description='Simulator for the Ricart-Agrawala mutex algorithm')
    parser.add_argument('infile', type=argparse.FileType())
    args = parser.parse_args()
    infile = args.infile

    try:
        main(infile)
    finally:
        infile.close()
```

RESULTS AND SNAPSHOTS



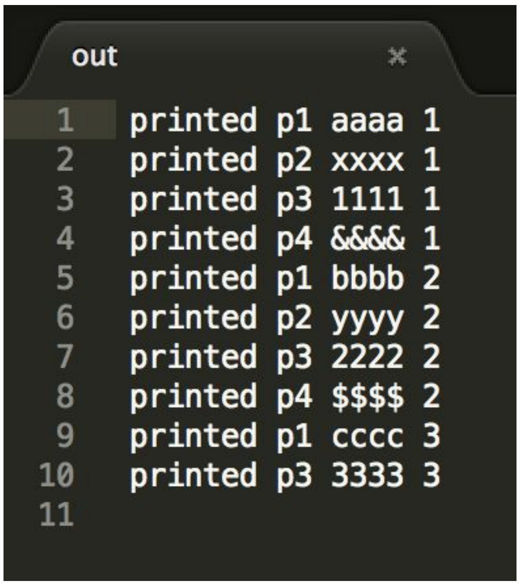
```
[Shrutis-MacBook-Pro:DCSproject shrutisingala$ make
output=`mktemp` && \
        for test in test/*; do \
                src/simulator.py $test/in > $output && \
                (diff --suppress-common-lines --side-by-side \
                          --ignore-blank-lines --ignore-all-space \
                        <(sort -k4 $output) \
                        <(sort -k4 $test/out) \
                 && echo -e "\033[1;32m[OK]\033[0m $test" \
                 || echo -e "\033[1;31m[FAIL]\033[0m $test") \
                || break; \
        done && \
        rm -f $output
[OK] test/mutex-large
[OK] test/mutex-simple
[OK] test/mutex-small
[OK] test/print
[OK] test/send-many
[OK] test/send-recv
Shrutis-MacBook-Pro:DCSproject shrutisingala$
```

Figure 1: Terminal Output



```
begin process p1
    print aaaa
    print bbbb
    print cccc
end process

begin process p2
    print xxxx
    print yyyy
end process

begin process p3
    print 1111
    print 2222
    print 3333
end process

begin process p4
    print &&&&
    print $$$$
end process
```

Figure 2: Sample input - print



```
printed p1 aaaa 1
printed p2 xxxx 1
printed p3 1111 1
printed p4 &&&& 1
printed p1 bbbb 2
printed p2 yyyy 2
printed p3 2222 2
printed p4 $$$$ 2
printed p1 cccc 3
printed p3 3333 3
```

Figure 3: Tribal man

CONCLUSIONS

The Ricart-Agrawala fair distributed mutual exclusion algorithm was implemented with the help of Lamport's logical clock. The performance was tested using the described parameters and it was found that the algorithm works efficiently.

REFERENCES

[1] Kandarp Jani and Ajay D. Kshemkalyani, "Performance of Fair Distributed Mutal Exclusion Algorithms" , IEEE Transactions on Parallel and Distributed Systems, 2004

[2] Y. I. Chang, "A Simulation Study on Distributed Mutual Exclusion", J Parallel and Distributed computing, Vol. 33(2):107-121, 1996

[3] S. Floyd, V. Paxson, "Difficulties in Simulating the Internet", IEEE/ACM Transactions on Networking, Vol. 9(4): 392-403, August 2001

[4] S. Lodha, A. Kshemkalyani, "A Fair Distributed Mutual Exclusion Algorithm", IEEE Transactions on Parallel and Distributed Systems, 11(6):537-549, June 2000

[5] G. Ricart, A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks", Comm. ACM, 24(1):9-17, January 1981