CMSC 327 Distributed Systems

Project 4: Group Communication
Due November 15, 2010

In this project you will implement causally ordered reliable multicast. The Python program provided implements both unordered and totally-ordered multicast. The totally ordered multicast procedure uses the algorithm outlined in the textbook relying on Lamport clocks and a hold-back queue. You should use the vector-clock based algorithm we discussed in class and described in the textbook to implement causally ordered multicast.

Learning Objectives

- Explore Group Communication
- Implement Causally-Ordered Reliable Multicast

Design

Describe how you have to modify the totally-ordered multicast implementation to provide causally-ordered multicast. Explain in prose how the vector clocks are updated and how message delivery is decided. Furthermore, design a small program that highlights the difference between different types of multicast.

Implementation

First, modify the current program to use vector clocks rather than Lamport clocks. Once the current functions use vector clocks correctly, implement causally ordered multicast. You can assume that the group is static, i.e. you do not have to handle failures or nodes leaving or joining the group. You should also implement a small program highlighting the differences between ordered and unordered multicast, e.g. a simple text-based chat program is provided.

Evaluation

You should compare unordered, totally-ordered, and causally ordered multicast. You should verify that the messages are being delivered in the correct order. You should evaluate your program with a group size greater than two and involving more than one machine. If you have time, investigate the cost of totally or causally ordering the messages in terms of overall message throughput.

Deliverables

Submit a zip file with the following directory structure:

```
cmsc327_proj4_LASTNAME_FIRSTNAME/
 Group.py      implements the causally ordered multicast
 README        simple text file explaining how to run your code
 results.pdf   your evaluation
```

Resources

The following program implements unordered and totally ordered multicast. The Python documentation for heapq[1], socket[2], and pickle[3] are useful for understanding the code.

```
# a list of the members of the group (IP, PORT)
group_members = [("127.0.0.1", 9000), ("127.0.0.1", 9001), ("127.0.0.1", 9003)]

class Group:

    def __init__(self, hosts, id):
        self.group = hosts

        # this processe id: [0, len(group) - 1]
        self.id = id

        self.host, self.port = self.group[id]

        # our lamport clock
        self.clock = 0
        self.clocklock = threading.Lock()

        # sockets to communicate to the group (outgoing)
        self.snd_sockets = []
        # a lock to coordinate use of the outgoing scokets
        self.sndlock = threading.Lock()

        # the hold back queue; used to deliver messages in order
        self.hold_back_q = []
        self.qlock = threading.Lock()

        # a dictionary to keep track of message acknowledgements
        self.acks = {}

        # process incoming messages in a separate thread
        self.server = IncomingConnections(self)
        self.server.start()

    def connect(self):
        ''' Connect to the other members of the group'''

        # make a list of the hosts we need to connect to
        left_to_connect = self.group[:]

        while len(left_to_connect) > 0:
            try:
                # grab the next group member to connect to
                host = left_to_connect[0]
                sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

[1] http://docs.python.org/library/heapq.html
[2] http://docs.python.org/library/socket.html
[3] http://docs.python.org/library/pickle.html

```
            sock.connect(host)

            # create a file from the socket; useful for reading/writing pickled objects
            self.snd_sockets.append(sock.makefile())

            # we have connected so we can delete it from our list
            del left_to_connect[0]
        except:
            print "could not connect to", host

        # give some time for others to connect
        time.sleep(.5)

def multicastMessage(self, msg, order = Message.TOTAL):
    ''' Send a multicast message to the group'''

    self.clocklock.acquire()
    self.clock += 1
    self.clocklock.release()

    msg.clock = self.clock
    msg.sender_id = self.id
    msg.order = order

    self.sndlock.acquire()
    for s in self.snd_sockets:
        # send the msg; write the pickled object to the socket file
        pickle.dump(msg, s)
        # flush file to make sure message is sent now
        s.flush()
    self.sndlock.release()

def receiveObj(self):
    ''' Receive a Message object '''

    message = None
    self.qlock.acquire()
    if len(self.hold_back_q) > 0:

        # grab the earliest message
        priority, m = heapq.heappop(self.hold_back_q)

        # make sure the message is ready to be delivered
        if m.ready:
            message = m
        # push it back on the heap, if it is not ready to be delivered
        else:
            heapq.heappush(self.hold_back_q, (priority, m))
    self.qlock.release()
    return message
```

```
def processMessageTotal(self, msg):

    '''
     Process this message according to totally ordered delivery.
     The messages and their acknowledgements are stored in the
     'ack' dictionary. The messages are hashed by the sender id
     and the sender's timestamp. Once a message has been acked by
     the entire group, it is ready for delivery. All ready
     messages are placed in a hold-back queue ordered by the
     sender's timestamp.
    '''

    assert(msg.order == Message.TOTAL)

    # lamport update
    self.clocklock.acquire()
    self.clock = max(msg.clock, self.clock) + 1
    self.clocklock.release()

    # a mesage is identified by the original sender's id, it's clock value
    hash = (msg.orig_id, msg.orig_clock)

    #if we have seen this message (or an ack from it) before
    if hash in self.acks:

        #if this is an ack, add to the end
        if msg.ack:
            self.acks[hash].append(msg)
        #if this is the original message, put it in the front
        else:
            self.acks[hash].insert(0, msg)

    #first time we have seen this message, add it to our list
    else:
        self.acks[hash] = [msg]

    self.qlock.acquire()

    # this is the original message, push it in our hold-back queue
    if not msg.ack:
        # the priority is the time-stamp of the message, and then the sender id
        heapq.heappush(self.hold_back_q, ((msg.orig_clock, msg.orig_id), msg))

    # got all the acks, mark this message to be delivered
    if len(self.acks[hash]) == len(self.group):
        # grab the orignal message stored at the first spot, set it to ready
        self.acks[hash][0].ready = True
        # delete this message from our dictionary of acks
        del self.acks[hash]

    # make a copy of the message
```

```
        m = copy.copy(msg)
        self.qlock.release()

        #if this message was a regular message (not an ack), ack it
        if not m.ack and m.sender_id != self.id:
            m.ack = True
            self.multicastMessage(m)

    def processMessageUnordered(self, msg):

        assert(msg.order == Message.UNORDERED)

        # lamport update
        self.clocklock.acquire()
        self.clock = max(msg.clock, self.clock) + 1
        self.clocklock.release()

        # deliver this message immediately
        self.qlock.acquire()
        msg.ready = True
        heapq.heappush(self.hold_back_q, (-1, msg))
        self.qlock.release()

class IncomingConnections(threading.Thread):
    ''' A server to listen to incoming connections and receive messages'''

    def __init__(self, group):
        threading.Thread.__init__(self)
        self.group = group
        # sockets for incoming  data
        self.rcv_sockets = []

    def run(self):
        print "Listening for connections on ", self.group.host, self.group.port
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((self.group.host, self.group.port))
        s.listen(50)

        while len(self.rcv_sockets) < len(self.group.group):
            conn, addr = s.accept()
            print "New connection:", addr
            # reads will timeout after 1 ms (useful when reading multiple sockets at once)
            conn.settimeout(.001)
            self.rcv_sockets.append(conn.makefile())

        while True:
            # go through the sockets and read an object from the socket
            for sf in self.rcv_sockets:
                try:
                    obj = pickle.load(sf)
```

```
                    if isinstance(obj, Message):
                        if obj.order == Message.UNORDERED:
                            self.group.processMessageUnordered(obj)
                        elif obj.order == Message.TOTAL:
                            self.group.processMessageTotal(obj)
                except socket.error:
                    # socket has no data, go on to the next
                    continue


if __name__ == "__main__":

    id = 0
    if len(sys.argv) > 1:
        id = int(sys.argv[1])

    try:
        g = Group(group_members, id)
        g.connect()

        while True:
            m = raw_input(">>> ")
            if len(m.strip()) > 0:
                g.multicastMessage(Message(id, m, g.clock), Message.TOTAL)

            v = g.receiveObj()
            while v:
                print v.sender_id, "> ", v.message
                v = g.receiveObj()

    except (KeyboardInterrupt, EOFError):
        sys.exit(0)
```