ECE454 Practice Midterm Solutions

Q1) Define transient inter-process communication and persistent inter-process communication briefly. Give an example of each.

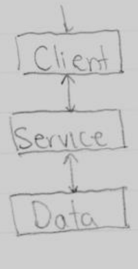
A: The definitions of these concepts are related to what happens to a message if the sender or receiver stops running.

Transient: the message disappears if the sender/receiver ceases to exist or function. An example of this is a remote-procedure call. An RPC will error out if it cannot reach the receiver.

Persistent: the message is stored by some middleware, so even if the receiver dies, the message is held until the receiver comes back up and receives it. An example is a message queue.

Q2) The architecture below is implemented in Java, using Thrift. Each time the client layer receives an RPC, the handler from the service layer creates a new AsyncClient and AsyncClientManager object, and invokes an asynchronous RPC on the data layer.

In an environment that limits the maximum number of processes per user, why might this be a bad design choice?

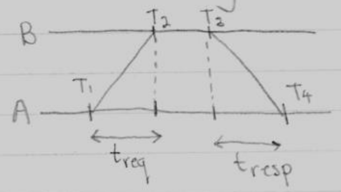


A: So first off, it's good practice to keep the number of instantiated clients to the bare minimum.

With respect to this environment, each Async Client Manager creates an internal thread to execute callbacks. If these cannot be garbage-collected quickly enough, the limit will be hit, and the service layer will crash.

To avoid this, we could create one Async Client and Async Manager, using some sort of lock system to allow for the re-use of the client object.

Q3) The following is an example of the NTP protocol.



$T_1 = 4$ ms $T_3 = 6$ ms
$T_2 = 4$ms $T_4 = 10$ ms

a) Compute Θ, the estimated offset.

A: The goal of finding the estimated offset is to average the differences in local clocks. It is always (receiver - sender).

$$\Theta = \frac{([T_2 - T_1] + [T_3 - T_4])}{2}$$
$$= \frac{[(4-4) + (6-10)]}{2}$$
$$= \frac{-4}{2}$$
$$= -2 \text{ ms}$$

b) Compute δ, the one-way offset.

A: δ is always (receiver - receiver), and this time it takes the difference instead of the sum.

$$\delta = \frac{[(T_4 - T_1) - (T_3 - T_2)]}{2}$$
$$= \frac{[(10-4) - (6-2)]}{2}$$
$$= \frac{4}{2}$$
$$= 2 \text{ ms}$$

c) Given the following (θ, δ) pairs, what is the most reliable estimate of offset and one-way delay?

(-2 ms, 2ms) (3 ms, 0 ms) (1 ms, 10 ms)
(2 ms, 1 ms) (13 ms, 3 ms)

A: The most reliable δ is the smallest δ. Its corresponding θ is then the best estimate for offset.

As such, $\delta = 0$ms and $\theta = 3$ms.

Q4)

a) Events e1 and e2 occur at t_1 and t_2, defined by vector clocks. $t_1 = [A:3, B:4, C:1]$ and $t_2 = [A:2, B:1, C:8]$. Does one of these happen before the other? Are they concurrent?

A: Remember that "happens before" for vector clocks is defined as the following:

An event $[A:n_1, B:n_2, C:n_3, \ldots]$ happens before another event $[A:m_1, B:m_2, C:m_3, \ldots]$ if $n_1 \leq m_1$, $n_2 \leq m_2, \ldots$ and at least one $n_x < m_x$.

So let's compare.

$$\begin{array}{ll} e1: & [A;3,\ B:4,\ C:1] \\ e2: & [A:2,\ B:1,\ C:8] \\ & \quad\downarrow \qquad \downarrow \qquad \downarrow \\ \text{Who's first:} & \quad e2 \qquad e2 \qquad e1 \end{array}$$

Since this isn't consistent throughout, these events must be concurrent.

b) Events e1 and e2 occur at t_1 and t_2, defined by Lamport clocks. $t_1 = 21$ and $t_2 = 7$. Can we say one happens before the other? Are they concurrent?

A: Lamport clocks ensure that if $a \rightarrow b$ (that is, a happens before b), the timestamp of a < the timestamp of b. However, that's the only promise it is capable of making.

As such, we can say for sure that $e1 \not\rightarrow e2$, but whether $e2 \rightarrow e1$ or they're concurrent, it's impossible to say.

Q5) Consider the Bully algorithm for leader selection. Assume that the processes are numbered consecutively from 0 to N-1, that k processes participate in the algorithm, and N-k are dead on arrival.

Ignoring coordinator messages, how many messages are sent in one execution in the worst case?

A: Okay, let's review how this algorithm works.

1) A process n sends ELECTION to all processes with higher ids
2) If a process with a higher id responds, n is eliminated. All processes must respond to an ELECTION message.
3) Any process that responded to the initial ELECTION now sends out ELECTION to higher ids.
4) If a process receives no response, it sends COORDINATOR to state it is now the leader to all other processes.

In the worst case, process 0 starts the election. It sends ELECTION to all processes with higher ids: 1, 2, ..., n-1.

Current messages sent: n-1

N-k processes are dead so they don't send any response. The other k-1 (as we're excluding 0) must send a response back to process 0.

Current messages sent: (n-1) + (k-1)

In the next round, those same k-1 processes all send out ELECTION to processes with higher ids.

Process 1 sends to 2, 3, ... n-1, a total of n-2 messages. Process 2 sends n-3, 3 sends n-4, etc, until the last process sends n-k messages. It may not be the case that process 1 sends n-1 messages as it might be a dead process. However, how we've enumerated this, it doesn't matter who sends how many as long as we know that in this stage, our total messages sent becomes:

$$(n-1) + (k-1) + (n-2) + (n-3) + \ldots + (n-k)$$

Then again, each process must respond to EVERY message it got from a lower process. There are k-1 of these non-dead processes that need to respond.

Going by the scheme we went with before, process 2 sends 1 OK to process 1. Process 3 sends two, to 2 and 1. This continues until the highest non-dead process responds k-2 times.

As such, the final message count is:

$$(n-1) + (k-1) + [(n-2) + (n-3) + \ldots + (n-k)] + [1 + 2 + \ldots + (k-3) + (k-2)]$$

Q6) Consider a $\overset{\text{Google}}{\wedge}$ file system with one master server and three chunk servers. Each chunk has a primary replica and two secondary replicas. Each host has a single 1 Gbps network interface, and is 1 Gbps between any pairs of hosts. Assume processing and network propagation delays are negligible.

a) Estimate how long it takes a client to write a 64 GB file to the cluster in the best case scenario.

A: As a reminder, let's go over how GFS does updates and reads. We'll start with updates.

1) Client contacts master to ask where the
   closest chunk server is.
2) Client pushes update to chunk server.
3) Updates from the written replica are pipelined to
   the next closest replica.
4) Once all replicas have the data, the client
   contacts the primary replica to assign the
   update a sequence number
5) Primary chunk passes update to other chunks

Now, this is a fairly involved process, but we have to look at the question closely to see that it's almost a trick question. It asks how long it takes to write the file, NOT how long does it take to write and for the replicas to completely replicate the data.

As such, it's simply $\frac{64\text{ GB}}{1\text{ Gbps}} = \frac{512\text{ Gb}}{1\text{ Gbps}} = 512\text{ s}$

It would actually be very difficult to figure out time to complete replication due to the pipelined nature of passing updates.

b) Estimate how long it takes a client to read a 64 GB file from the cluster in the best case.

A: To perform a read:
1) Client sends file name and chunk index
   to the master
2) Master responds with contact address of
   the chunk server
3) Client pulls data directly from the chunk

File striping might allow the client to read from multiple chunks at once, which reduces the disk I/O bottleneck, but since we need to send this data over the network, it's impossible to subvert that restriction.

As such, the read time will also be approximately 512 seconds as well.

Q7) Input data: Student1, ECE103, ECE254
Student 2, ECE103
Student 3, ECE356, ECE205

Write a MapReduce solution to yield a list of students who've taken ECE 103.

A: It is important to understand the roles of Mappers and Reducers to write an efficient solution to this problem. Mappers split the input, and for each split, can produce zero or one key-value pair as a result. Reducers take combine all values associated with a specific key, aggregates them, and produces zero or one key-value pairs.

A reduce operation requires a shuffle, a very costly operation, so we should only use them if we REALLY need it. Another neat trick is that setting the value to null in the key-value pair produces just the key.

So what will we map? We can split the input line, iterate through the non-student-id portions, and if we find ECE103, we can just emit the student id. So do we need a reducer at all? No!

```
public void map (Object key, Text value, Context context) {
        String [] symbols = value.toString().split(",");
        for (int i = 1; i < symbols.length; i++) {
                if (symbols[i].equals("ECE103")) {
                        context.write(symbols[0], null);
                        break;
                }
        }
}
```

Q8) Using the input provided in Q7, generate a list of distinct course names, in a file whose path is held in the variable 'outpath'. The input is located in "textFile", which is already an RDD. Use Spark.

A: By default, the input split of spark is on newlines. Let's go over some useful Spark operations.

Flatmap is the first one: it makes a list by performing operations on another list. Flatmap in this course is used almost exclusively for one thing: splitting a line into its comma-delimited parts.

```
textFile.flatMap(line => line.split(",").drop(1))
```

So what exactly does this line do?

"Student 1, ECE103, ECE254" ⇒ ["ECE103", "ECE254"]

Just for a single line, we've split it into its comma-delimited parts, and dropped the first entry (the student name) as it's unimportant to this question.

Overall, it produces the following list.

["ECE103", "ECE254", "ECE103", "ECE356", "ECE205"]

though not necessarily in that order. We've now converted the RDD to a list, but that's not what we want at the end. So here comes the next function.

Map: map takes each element in a collection, and passes it through some function, and returns a new RDD based on its results. Since all we need to do is convert this to an RDD, no complex functionality is needed, but we're going to output the results as key-value pairs in preparation for the next step.

```
textFile.flatMap(line => line.split(",").drop(1))
        .map(courseName => (courseName, null))
```

ReduceByKey: reduceByKey is a bit of an odd one. It doesn't work in the same way as MapReduce's Reducer, as we might expect. Let's break down the reduceByKey syntax.

```
reduceByKey((oldAggregation, newVal) => blah)
```

Okay, this seems kind of weird still, so let's clear up a few things. First, the function on the inside is executed once PER instance of the key encountered. The aggregation is the result of the PREVIOUS iteration of this function. newVal is the next value encountered associated with a given key.

So for example, with the RDD {(1,2), (3,4), (3,1)}

reduceByKey((x,y) => x+y)

Let's walk through what happens.

1) Encounters key = 1. Old aggregation does not exist. New value is 2.

(0,2) => 0+2

2 is returned as the aggregate value.

2) Encounters key = 3. No old aggregate. New value is 4.

(0,4) => 0+4

4 is returned.

3) Encounters key = 3. Old aggregate is 4. New value is 1.

(4,1) => 4+1

5 is returned.

The final RDD is {(1,2), (3,5)}. Okay. Back on topic. In our case, we don't actually care about the aggregate value. So all we need to do is return null.

```
textFile.flatMap(line => line.split(",").drop(1))
        .map(courseName => (courseName, null))
        .reduceByKey((oldAggregation, newVal) => null)
```

From here, we just want the names, not the values, so we can convert this just to a regular RDD of strings by mapping (courseName, null) tuple to just the course.

```
textFile.flatMap(line => line.split(",").drop(1))
        .map(courseName => (courseName, null))
        .reduceByKey((old, new) => null)
        .map(tuple => tuple._1)
        .saveAsTextFile(outPath)
```

That's it! Hope you enjoyed.