Problem Set 2

15-440/15-640 Distributed Systems Spring 2015

Assigned: Thursday February 19, 2015

Due: 5pm, Tuesday February 24, 2015

Submission: Submit .pdf file via Autolab. ***No other file format will be accepted.*** The .pdf file can be a scan of handwritten answers, or .pdf created from a word processor such as Word, Latex, LibreOffice, Google Docs, etc.

Question 1 (20 points)

Which of the following workloads would MapReduce be good for? In each case, provide a couple of sentences justifying your answer.

A. Building a real-time news aggregation service that provides up-to-the-minute news updates. It should be designed to parse feeds from multiple news websites and refresh its index in real-time.

B. Building an interactive data exploration tool which lets users provide their own queries and rapidly see partial results from their queries without waiting for the query to run over an entire dataset.

C. Building a high-frequency trading application. You've been asked to architect a high-frequency stock trading application which needs to consume stock ticker updates and provide quick decisions for trades.

D. Building a database of features from images for face recognition. The input is hundreds of millions of user uploaded and tagged photos, each stored as a file in the local file system. For each image an algorithm must run that extracts features for training face recognition algorithms in a later stage.

E. Crawling through billions of log entries in log files, one per line of input, across thousands of websites and counting the number of unique visitors (unique by IP address) that access those websites. This will feed daily, weekly, and monthly reporting systems that provide website analytics.

Question 2 (30 points)

Suppose global variable, `A, on a server` is a very large one-dimensional array of two-dimensional points. The server uses RPC to communicate with clients, and implements the following operation on `A`:

```
        void storeAverageInIndex(int minIndex, int maxIndex, int targetIndex)
```

that stores in `A[targetIndex]` the average of all of the points in the range `{minIndex, minIndex+1, ..., maxIndex-1, maxIndex}`.

Currently, the server pseudo code looks like this:

```
01. struct Point {
02.    float x;
03.    float y;
04. }
05.
06. Point [] A;
07.
08. void storeAverageInIndex(int minIndex, int maxIndex, int targetIndex) {
09.    float sumX = 0.0;
10.    float sumY = 0.0;
11.    for(int i = minIndex; i <= maxIndex; i++) {
12.        sumX += A[i].x;
13.        sumY += A[i].y;
14.    }
15.    int count = maxIndex - minIndex + 1;
16.    A[targetIndex].x = sumX / count;
17.    A[targetIndex].y = sumY / count;
18. }
```

For example, if the array `A` is `[(0,0), (1,1), (2,2)]`, we expect the following output from this sequence of operations:

```
storeAverageInIndex(0,2,2); // A is now [(0,0), (1,1), (1,1)]
storeAverageInIndex(2,2,0); // A is now [(1,1), (1,1), (1,1)]
```

A. Suppose multiple concurrent clients access this server. What is the problem with this solution? Give an example of an erroneous execution that involves just two clients.

B. Is it ever possible to get correct results with this code when there are two clients? When there are three clients? When there are 100 clients?

C. Consider this modified code for the server:

```
01. Lock superLock;
    lines 01 to 06 from original code unchanged
08.
09. void storeAverageInIndex(int minIndex, int maxIndex, int targetIndex) {
10.      superLock.lock();
    lines 09 to 17 from original code unchanged
20.      superLock.unlock();
21. }
```

Will this code work correctly when there are many concurrent clients? Explain your answer.

D. Performance with multiple clients is observed to be very slow with solution C. What might be the cause of this performance problem?

E. Consider this new version of the server code. Notice the use of reentrant locks.

```
01.ReentrantLock [] miniLock; // assume same size as array A
```
lines 01 to 06 from original code unchanged
```
08.
09. void storeAverageInIndex(int minIndex, int maxIndex, int targetIndex) {
10.    for(int i = minIndex; i <= maxIndex; i++) {
11.        miniLock[i].lock();
12.    }
13.    float sumX = 0.0;
14.    float sumY = 0.0;
15.    for(int i = minIndex; i <= maxIndex; ++i) {
16.        sumX += A[i].x;
17.        sumY += A[i].y;
18.    }
19.    int count = maxIndex - minIndex + 1;
20.    miniLock[targetIndex].lock();
21.    A[targetIndex].x = sumX / count;
22.    A[targetIndex].y = sumY / count;
23.    miniLock[targetIndex].unlock();
24.    for(int i = minIndex; i <= maxIndex; ++i) {
25.        miniLock[i].unlock();
26.    }
27. }
```

Why is this solution using reentrant locks instead of regular locks? Give an example of an execution in which the use of reentrant locks is necessary.

F. Can the solution in E ever deadlock? If so, give an example execution sequence in which this occurs. If it cannot deadlock, given an informal explanation of why (we are not looking for a formal proof).

Question 3 (25 points)

In the Haber process, ammonia is created by the combination of one unit of nitrogen with three units of hydrogen under the influence of an appropriate catalyst. A chemical company that manufactures ammonia has just switched over to computer-based process control. Unfortunately, serious production and quality problems are being experienced. The trouble shooters claim that the process control software is to blame. The pseudocode of this software is shown on the next page.

```
01. #define MAX 100
02.
03. /* Buffers for chemicals; each array used as a ring */
04. buffer AmmoBuf[MAX], NitroBuf[MAX], HydroBuf[3*MAX];
05.
06. /* Counting semaphores */
07. semaphore AmmoEmpty = MAX, AmmoFull = 0;
08. semaphore NitroEmpty = MAX, NitroFull = 0;
09. semaphore HydroEmpty = 3*MAX, HydroFull = 0;
10.
11. /* Pointers into buffers; all zero initially */
```

```
int FirstFullAmmo = 0, FirstEmptyAmmo = 0;
int FirstFullNitro = 0, FirstEmptyNitro = 0;
int FirstFullHydro = 0, FirstEmptyHydro = 0;

/* Mutual exclusion lock, to protect access to above pointers */
Lock PointerMutex; /* initially unlocked */

/* Each of the procedures that follow is run as a separate thread; assume
  they started up by a main procedure that is not shown here */

void MakeAmmonia() {
   buffer myammo, mynitro, myhydro[3];
   while (TRUE) {
       /* Fill local buffers */
       P(NitroFull);
       PointerMutex.lock();
       mynitro = NitroBuf[FirstFullNitro];
       FirstFullNitro = (FirstFullNitro + 1) % MAX;
       PointerMutex.unlock();
       V(NitroEmpty)

       P(HydroFull);
       PointerMutex.lock();
       myhydro[0] = HydroBuf[FirstFullHydro];
       myhydro[1] = HydroBuf[FirstFullHydro+1];
       myhydro[2] = HydroBuf[FirstFullHydro+2];
       FirstFullHydro = (FirstFullHydro + 3) % (3*MAX);
       PointerMutex.unlock();
       V(HydroEmpty);

   /* Produce ammonia from mynitro and myhydro into myammo */

   /* Put ammonia into global buffer */
       P(AmmoEmpty);
       PointerMutex.lock();
       AmmoBuf[FirstEmptyAmmo] = myammo;
       FirstEmptyAmmo = (FirstEmptyAmmo + 1) % MAX;
       PointerMutex.unlock();
       V(AmmoFull);
   }
}

void FillTanker() {
   buffer myammo;

   while (TRUE) {
   P(AmmoFull);
   PointerMutex.lock();
   myammo = AmmoBuf[FirstFullAmmo];
   FirstFullAmmo = (FirstFullAmmo + 1) % MAX;
   PointerMutex.unlock();
   V(AmmoEmpty);

   /* Fill tanker from myammo; assume tanker always available */
   }
}

```

```

void MakeNitrogen() {
   buffer mynitro;

   while (TRUE) {
   /* Produce nitrogen from raw materials and fill mynitro */

   P(NitroEmpty);
   PointerMutex.lock();
   NitroBuf[FirstEmptyNitro] = mynitro;
   FirstEmptyNitro = (FirstEmptyNitro + 1) % MAX;
   PointerMutex.unlock();
   V(NitroFull);
   }
}

void MakeHydrogen() {
   buffer myhydro;

   while (TRUE) {
   /* Produce hydrogen from raw materials and fill myhydro */

   P(HydroEmpty);
   PointerMutex.lock();
   HydroBuf[FirstEmptyHydro] = myhydro;
   FirstEmptyHydro = (FirstEmptyHydro + 1) % (3*MAX);
   PointerMutex.unlock();
   V(HydroFull);
   }
}
```

A. Do you agree with the trouble shooters? In other words, are there one or more bugs in this code? If so, identify the bug(s) and suggest an appropriate fix. If not, offer a crisp explanation to confirm that this code works correctly.

B. For optimal efficiency of the chemical plant how many FillTanker, MakeNitrogen, and MakeHydrogen threads should there be relative to MakeAmmonia threads? State any assumptions you are making.

Question 4 (10 points)

In the days of yore, dinosaurs roamed the earth, the Gopher protocol was still a viable alternative to HTTP, and animated gifs were exciting web objects. In those days, people used Netscape Navigator to browse the World Wide Web. This browser pioneered innovations such as SSL, cookies, and Javascript. However it ran on a single thread. What negative implications would this have had for web page design and user experience? Why were those negatives acceptable?

(Hint: consider the technology of the computers and networks of that era.)

Question 5 (15 points)

Consider this Java pseudo code to simulate human interactions. The intent is to have multiple concurrent threads, each representing a person. The Java `synchronized` feature is used to avoid a possible race condition in the access to variables `sent` and `received`.

```
class Greeting {
   String message;

   public Greeting(String _message) {
       message = _message;
   }

   public Greeting sendTo(Person p) {
       p.listen(message);
  }
}

class Person extends Thread {
   int sent = 0;
   int received = 0;

   public void run() {
       while (true) {
           talk();
       }
   }

   public synchronized talk() {
       Person p = getRandomFriend();
       Greeting g = new Greeting("Hello!");
       g.sendTo(p);
       sent += 1;
   }

   public synchronized listen(String message) {
       received += 1;
   }

   public Person getRandomFriend(){
           some code here to randomly pick a person
   }
}
```

Unfortunately, this code has a serious concurrency bug. Find it and suggest a fix (just give a verbal description of the fix, you don't need to show actual code).