Problem Set 3

15-440/15-640 Distributed Systems Spring 2015

Assigned: Thursday April 2, 2015

Due: 5pm, Thursday April 9, 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 (10 points)

A. What is the difference between “scale up” and “scale out”?

B. Give two reasons why scale up may not benefit a particular application.

C. Give two reasons why scale out may not benefit a particular application.

Question 2 (30 points)

A system administrator costs $100 per hour and can administer 50 machines at a time. Currently, there are 25 machines in a cluster. Each machine can handle 10 requests per second. Therefore, the current capacity of the cluster is 25 * 10 = 250 requests per second.

It costs $50 to scale up (i.e., upgrade) an existing machine to handle 20 requests per second. Alternatively, you can scale out by adding a new machine that is just like the existing ones at a fixed cost of $15; like the others, this machine can serve 10 requests per second.

Suppose you want to increase the capacity of your cluster to handle 600 requests per second, and run it at this increased capacity for a full year (24x7). Relative to system administrator costs, you can assume that cooling, power, hardware maintenance, and other costs are negligible. You are free to use any combination of scale up and scale out, but remember that you will have to hire another system administrator for every 50 machines.

What is the most cost-effective path to reaching the desired capacity?

Question 3 (15 points)

A simple transaction transfers one hundred dollars from one account to another. The pseudo-code looks like this:

```
01. START_TRANSACTION
02. X = read value from account A // currently 1000
```

```
03. Y = read value from account B // currently 2000
04. X = X + 100
05. Y = Y - 100
06. set value of account A to X
07. set value of account B to Y
08. END_TRANSACTION
```

Assume you are using intentions lists to implement transactions.

A. When recovering from a crash, your transaction system finds the following intentions list:

```
01. <begin transaction T1>
02. <T1: X = 1100>
03. <T1: Y = 1900>
04. <commit T1>
```

How should the system handle this intentions list?

B. During recovery, how should the system handle the following intentions list

```
01. <begin transaction T1>
02. <T1: X = 1100>
03. <T1: Y = 1900>
```

C. During recovery, your friend claims he saw the following intentions list

```
01. <begin transaction T1>
02. <T1: X = X + 100>
03. <T1: Y = Y - 100>
04. <commit T1>
```

Is this possible, or is your friend hallucinating?

Question 4 (15 points)

Consider a distributed system consisting of 3 fail-fast servers and one fail-fast client. To perform an operation, the client contacts all three servers in parallel, using separate threads. For the operation to succeed, the client has to receive success return codes from at least two of the servers. Unfortunately, there is a software bug on the servers that strikes at random during an operation. This causes a fail-fast crash of the affected server. Crashes are NOT correlated across servers.

Suppose the probability of this bug striking a server during an operation is 0.05. Further suppose that only a single client is active and that failed servers are restarted with negligible delay. What is the fraction of operations by the client that fail?

Question 5 (30 points)

When giving a raise to an employee, John uses a distributed transaction to ensure that all four copies of the employee database (located at different servers) are atomically updated. The client is acting as the coordinator of the two-phase commit. At each server, the pseudocode that is executed to give the raise looks like this:

```
01. begin transaction
02. obtain persistent lock on this employee
03. x = current salary of this employee (obtained from the database)
04. x += amount of raise
05. set this employee's salary in database to x
06. send new salary as success reply to coordinator
07. release persistent lock on employee
08. end transaction
```

After some weeks of intense use and many crashes due to buggy software elsewhere in the system, John is horrified to learn that some employees have different salary values on different servers. It was precisely to avoid this situation that he used a distributed transaction in the first place.

Identify the bug in the above pseudo code that could lead to this situation. How would you fix it?