Problem Set 1 Solution

15-440/15-640 Distributed Systems Spring 2017

Assigned: Thursday February 2, 2017

Due: Thursday February 9, 2017 (by the start of class)

Question 1 (16 points)

In each of the following situations, identify the **weakest** RPC semantics that can be used. As explained in class, *exactly-once* semantics is stronger than *at-most-once*, which is stronger than *at-least-once*. Explain your answers.

A. Setting up a VoIP (Voice over IP) call to a friend

B. Uploading a blog post

C. Buying a mobile phone from an e-commerce website

D. Checking the stock price of a company

Solution:

A. At-most-once. We do not want to generate multiple call requests. Also, if the call fails the user can retry.

B. Either At-least-once or At-most-once with the right reasoning:

(1) At-least once: It is not harmful, in this case, to keep sending call requests to the server machine until we get an ACK. In other words, it wouldn't hurt to post two identical blogs in a row.
(2) At-most-once: User can simply re-post if the previous attempt is not successful. This semantic helps avoid indefinite blocking, or any other reasonable concerns.

C. Exactly-once: We need to make sure that the transaction happens, and happens once only.

D. At-least-once: Since the operation is idempotent, meaning the global state is preserved even if multiple calls may have been issued, we can keep sending requests; and this semantic guarantees that the `checkPrice()` call executes at least once.

Question 2 (20 points)

John is designing a location-aware printing service for the CMU campus. Because there are so many printers on the network, scattered in many buildings, it is often difficult to know where the nearest

printer is located. John's printing system will route your document to the printer nearest to you, and tell you where that printer is located. In this design, users send documents from their laptops, tablets or smartphones to a central machine in Cyert Hall. That central machine is aware of your current location as well as the locations of all printers on campus. It picks the optimal printer, sends your document to it, and returns to you the name of that printer and its location.

A. Identify the servers and clients in this system.

B. John creates an RPC interface on the central machine that implements a single `print` function. It requires the caller to provide the content and length of the document to print, and a structure that indicates the user's location. The RPC will return the location information for the printer that was actually used. It also sets a status to indicate whether the printer finished successfully or if something went wrong (e.g., the printer is jammed or out of paper). The `print` function has the following C prototype:

```
location_t* print (char *buf, int len, location_t *user_loc, int *status);
```

You may assume that `location_t` is a simple structure that is easy to marshall/unmarshall. What are the `in`, `out`, `in-out` parameters of this RPC?

C. Identify and explain what complexities might arise in generating the stub code for this RPC, even assuming that `location_t` is a simple structure that is easy to marshall/unmarshall.

D. On the CMU campus, end-to-end bandwidth between a wireless mobile device and the Cyert Hall server is typically 50 Mbps. End-to-end latency is typically 20 ms. What timeout value would you recommend for this RPC call? Explain your answer. Based on your answer, suggest an improved RPC interface for John's printing system.

Solution:

A. Laptops are clients. Printers are servers. Central machine is server to laptops, clients of printers.

B. `buf`, `len`, `user_loc` are in parameters; `status` is an out parameter.

C. Parameters are passed as pointers, so the client stub needs to dereference and serialize these. Return value is a pointer -- this will need allocation in the client stub, and deallocation in the server stub in addition to serialization.

D. Reasonable timeout should be greater than the worst case time for RPC. Most documents (pdfs) are typically around 1 MB or smaller. Assume worst case 10 MB. Time for RPC includes serialization, transmission time, network latency, deserialization, printing, and transmission of result (including serialization steps, network latency). Serialization, deserialization, sending reply, and network latency are negligible relative to transmission of the document and printing.

- Transmission = 10*8/50 = 1.6 s
- Printing = few seconds to several minutes on a long document / busy printer; say 15 minutes worst case.
- Time out will need to be >15 minutes!

Better approach: design RPC with separate call for enqueuing document, and another call for checking on its status.

Question 3 (24 points)

Terry's software for citizen science runs on (relatively slow) Android smartphones. This software involves statistical analysis on many `int32` arrays, each containing a million entries. The operation `VeryExpensive(...)` on these arrays is frequently invoked by his algorithms. To speed up execution, Terry is considering offloading execution of this operation via RPC to a server that uses a high-end GPU for ultra-fast computation of `VeryExpensive(…)`. The C prototype of the RPC is:

```
int32 VeryExpensive (int32 *array)
```

This RPC sends the specified array as input, and returns the single `int32` value of `VeryExpensive(…)` on that array.

You may assume the following performance costs:

- marshalling or unmarshalling one integer: 100 ns on smartphone, 10 ns on server
- OS cost of sending a request or receiving a reply: 30 µs on smartphone, 10 µs on server
- one-way client-server network latency (symmetric): 2 ms
- client-server bandwidth (symmetric): 50 Mbps
- compute time for `VeryExpensive(…)` on server for a million-integer array: 5 ms

All other performance costs can be ignored.

A. Suppose Terry wants to compute `VeryExpensive(…)` on one of his arrays. How long will it take to perform a single RPC call, assuming no failures occur?

B. Suppose Terry's smartphone is a single-core machine (without hyperthreading). The algorithm being run requires two other computations, each taking roughly 6 ms on the smartphone. These two computations have no dependencies on each other or on `VeryExpensive(...)`. In other words, all three computations can execute in parallel with no synchronization. The final result, however, depends on the result of all three. If Terry created multi-threaded code to perform both the local operations in parallel with the RPC for `VeryExpensive(…)`, how long would the entire algorithm take?

C. If Terry replaced his old single-core smartphone with a brand new quad-core smartphone, how would your answer to part B change?

D. Explain what would happen at the client and` server if Terry loses wireless connectivity immediately after his RPC request is sent. Clearly state and justify any assumptions you make.

Solution:

A. One array size = = 4 * 10^6 bytes = 3.2 * 10^7 bits

Transmission delay1 = array size/ bandwidth = (3.2 * 10^7) / (5 * 10^7) s = 0.64 s

Transmission delay2 = integer size/bandwidth = 32 / (5 * 10^7) s = 0.64 us

T = (client marshal time + OS send time + trans delay + network delay)
+ (OS receive + server unmarshal time + server process time + marshal result + OS send + trans delay + network delay)
+ OS receive + unmarshal result
= 100ns * 10^6 + 10ns * 10^6 + 100ns + 10ns + 2*30us + 2*10us + 0.64s + 640ns + 2*2ms + 5ms
= 100ms + 10ms + 0.08ms + 640ms + 9ms
≈ 759.08ms

B. Single Core, multi-threading: While Terry is waiting for the reply, he can also do the other computations. Since they are running on a single processor, although conceptually the threads are said to run at the *same time*, they are actually running consecutively in time slices allocated and controlled by the operating system.

Therefore:
T1 = client marshal time + OS send time
T2 = 2 * 6ms = 12ms (this starts at the same time while waiting for reply), which is a lot smaller than the time waiting for reply.

In total, T = 759ms

C. Quad-core means that these operations can run at the same time, but since the transmitting time is the dominate , so the time still doesn't change. (Answers that parallelize marshalling are also accepted)

D. VeryExpensive(...) is an idempotent operation (i.e. it is safe to execute it repeatedly). This indicates that at-least-once RPC semantic can be employed. As such, no history needs to be maintained and no duplicate filtering needs to be pursued at the server.
Two scenarios could happen:

1) The server receives Terry's request. VeryExpensive(...) will be executed at the server and a reply will be sent back to Terry's smartphone. However, the reply will not be received due to wireless disconnectivity. Also, the execution of VeryExpensive(...) that specifically pertains to Terry's request will NOT be an orphaned one due to the at-least-once semantic. The

server will not re-execute VeryExpensive(...) for Terry since Terry's smartphone is disconnected and will not, accordingly, perform retransmissions.

2) The server does not receive Terry's request. Nothing happens afterwards since Terry's smartphone is disconnected, thus re-transmissions are not performed.

Question 4 (12 points)

After wreaking havoc on Checkpoint 2 of Project 1 in 15-440, Murphy needs a break. However, he still needs to cause trouble for Alice. She is working for a Pittsburgh startup that is introducing a new service for last-minute vacations. Her client software first contacts a weather server to identify the best locations in the United States for a short vacation in the next 24 hours, taking into account the customer's vacation preferences. For example, if the customer likes skiing, optimal weather would be defined as fresh and deep snow. If the customer prefers a beach vacation, the optimal weather would be sunshine and blue skies with moderate wind. After the top few vacation destinations have been identified, Alice's client software contacts a travel website (like Orbitz) to check on air fares to those destinations. Based on the intersection of weather and fares, the system recommends the optimal vacation destination and its price to the customer. Murphy seeks your help to cause trouble for Alice.

A. Identify two problems you can help Murphy create if Alice is not careful in designing the system. Clearly state and justify any assumptions that you make.

B. Is there a scenario in which Alice buys a plane ticket for a suboptimal destination? If yes, give an example of such a scenario, and explain how Alice could fix her implementation to avoid such scenarios. If no, explain why such a problem cannot occur.

Solution:

A. If Alice is "not careful" a few issues arise:
   - The weather server may not respond to queries fast enough, so the destination forecasts will be wrong after the results come back
   - The travel service may have at least once semantics, so if Alice sends multiple buy requests, then her startup will lose money
   - If the customer were to wait long enough for the travel fares to climb above the quoted price, then Alice's startup will lose money, as she would charge the client x but would need to purchase at $> x$ (if not checked)

B. *Yes!* It is possible that the weather data is stale, in that Alice could suggest plane tickets to Hawaii, thinking it is sunny, when in fact it is rainy. This would happen in the following scenarios:
   - The response from the weather server takes *so long* that the information it returns is valid for the time of the original request, but not for the present, when the ticket is purchased
   - Similarly, the request to the travel agency could take *so long* that the weather changed in between the (accurate) weather report and the purchase of the airfare

- The customer waits long enough that the weather changes, and purchases the tickets then

A few *fixes* that could work:

- When prompting customers with options, provide a confirmation dialog (re-checking) the weather and travel fares
- Assuming that checking the weather is fast (and that delay is coming from inter-service communication) and assuming that booking / checking airfare is fast, you could consolidate both services to avoid stale information

Question 5 (12 points)

WE_NEVER_LOSE_IT is a highly reliable data archiving service located in suburban Seattle. To reduce the chances of a catastrophic site failure (such as a fire or flood) wiping out data, the company decides to use off-site mirroring. When archival data is added to the primary site in Seattle, copies are also sent via a dedicated network to its two backup servers in Tacoma, WA (so close to Seattle that it is almost a suburb) and Zurich, Switzerland. The network bandwidth on all links is guaranteed to be 800 Mibps (i.e., 800×10^6 bps).

To test the stability of the network, WE_NEVER_LOSE_IT first sends out a test packet of size 10 bytes from the Seattle server to each of the backup servers. Each backup server sends a 10-byte ACK in response. The time required for the Seattle-Tacoma-Seattle route is much smaller than for the Seattle-Zurich-Seattle route. Specifically, after multiple trials under carefully controlled conditions, the total time (from start to ACK) on the Seattle-Zurich-Seattle is observed to be about 54 ms longer than the value for Seattle-Tacoma-Seattle. You can assume that processing time is negligible, no packets are lost, and no data corruption happens. For your answers below, please explain your reasoning. State and justify any assumptions you make.

A. Why does the Seattle-Zurich-Seattle interaction take longer than the Seattle-Tacoma-Seattle interaction? From the data provided can you estimate the distance between Seattle and Zurich?

B. Now consider the following protocol for data transfer. Assume all data packets are of size 500 bytes. If necessary, data objects are padded with null bytes to make their length a multiple of 500 bytes. The source sends out exactly 2000 packets to the destination, and then stops to wait for an ACK. Once the ACK is received, the source immediately sends the next 2000 packets, and so on. What is the throughput of this protocol between Seattle and Zurich for a multi-gigabyte archived data object? Express your answer as a percentage of network bandwidth.

Solution:

A. The longer geographic distance between Seattle and Zurich accounts for the propagation delay, which is the eventual reason for the longer wait. Assuming the network is connected by optical fibre, then, using the speed of light *c* we can estimate the distance between these two cities:

$$d_{SZ} = \frac{1}{2} \cdot (0.054 \text{ sec} \cdot 3 \cdot 10^5 \text{ km/sec}) \approx 8100 \text{ km}$$

Note that we need to divide by 2 because the original product yields the RTT (round-trip time), not the one-way distance.

The answer is acceptable if the student answers in the unit of miles (about 5033 miles).

B. 2000 packets of size 500 bytes will be in total 1000 KB. According to the values provided above, the propagation delay RTT is about 54ms. Meanwhile, the transmission delay for 2000 packets should now be

$$t'_{TD} = \frac{1 \cdot 10^6 \text{ bytes}}{800 \cdot 10^6 \cdot \frac{1}{8} \text{ bytes/sec}} = \frac{1 \cdot 10^6}{100 \cdot 10^6} \text{ sec} = 0.01 \text{ sec}$$

This implies the total delay is 54ms + 10ms = 64ms. In other words, the main server A sends out 1000KB of data for every 64ms. Therefore, the long-term throughput is about (on average):

$$\frac{1000 \cdot 10^3 \text{ bytes}}{0.064 \text{ sec}} = 1.5625 \cdot 10^7 \text{ bytes/sec}$$

Since the bandwidth is 800Mbps (i.e. 100 Mbytes per second), this average throughput is exactly 15.625% of the bandwidth.

Question 6 (16 points)

Harry's startup provides a web site for posting pictures of vehicles and tagging them. The persistent storage for the images and tags is provided by a key-value store. Here is the design of his system:

User: HTTP⟶ *Web Server*: RPC⟶ *Tag Storage System*

In other words, the user interacts with a web page using standard HTTP requests. When the user clicks on a widget (e.g., a button to delete an image), the web server issues an RPC to the tag storage system to perform the operation. The RPC interface supports three operations on the tag storage system: `INSERT`, `DELETE` and `FIND`.

You may assume that the tag storage system does not crash. However, the network between the web server and tag storage system is unreliable and may drop packets. Assume that the Web server is single-threaded, and that no other services access the tag storage system. So the tag storage system only has to process one RPC at a time. Clearly and concisely explain your reasoning in your answers to the questions below:

A. Assume at-least-once RPC semantics. Can the following situation occur?

A `FIND` RPC performed on the web server is successful, but returns no matching tags even though matching tags exist in the tag store.

B. Again, assume at-least-once RPC semantics. Can the following situation occur?

A `DELETE` RPC on the web server returns a "NoSuchTag" error, even though it successfully deleted a tag.

C. Now assume at-most-once RPC semantics. If no reply is received after some time, the RPC terminates with a timeout error code that is then returned by the web server to the user as an error web page. Can the following situation occur?

A `FIND` RPC does not timeout, but fails to return a matching tag even though it exists in the tag storage.

D. Again, assume at-most-once RPC semantics as in Part C. Can the following situation occur?

A `DELETE` RPC returns "NoSuchTag" error, even though it successfully deleted a tag.

Solution:

A. No. At-least-once will keep trying until it succeeds. When it succeeds, it will always return tags if they exist in the system. Since the web server is single threaded, the requests will be serialized.

B. Yes. Suppose the first delete succeeds, tries to return 'Success', but the return value is dropped. When the delete is retried the tag will already be deleted and the delete will now return 'NoSuchTag'.

C. No. At-most-once tries exactly once. If the RPC does not time out, it must have succeeded on the first try. Thus, if matching tags are present, they must be reported.

D. No. If it does not timeout, the RPC must succeed. The response from the delete will therefore not be lost, and will return 'NoSuchTag' if and only if the tag was not present.