Solutions: Assignment 1

1. Client issues the RPC to the server via the persistent data-space. If the server is available, then the data-space immediately delivers the message to the server. Otherwise, it is stored in the data-space, with the RPC request keyed on the server's ID. When the server is available again, it checks the data-space for RPCs sent to it while it was unavailable. It then fetches and removes these requests and continues the RPC operations. The reverse direction is identical except the message is delivered to the client. The client will check if there are any responses for it if it leaves and comes back (a common scenario for wireless clients).

   To support "at-least-once" semantics, the client will re-send the request after a timeout. This is to ensure that, even if requests get dropped at the server, due to server crashes, or in the persistent data-space, the asynchronous RPC will eventually complete.

2. This ended up being a tricky question to grade. We decided to be "flexible" with what constitutes a correct answer.

a. A number of you answered the following: avoid packet drops by using the >90% full message as a signal to perform multiplicative decrease. This is an okay answer, and in fact, is the basis for ECN, a real implementation of this design. You should point out how this is helpful; it reduces latency and jitter by mostly eliminating packet drops. However, it does not increase throughput or link utilization. I was hoping that your answers would more fully explore the problem space. For example, is it possible to change the slope of the additive increase once a >90% full message is received? By reducing the slope, the system would spend more time with queues mostly full, improving link utilization. What about marking the point where the first >90% full message was received? Perhaps instead of performing a multiplicative decrease immediately, the sender can reduce its send rate to the marked point before using multiplicative decrease to address additional packet drops. I gave a few bonus marks for those that explored more of the design space. I received a number of answers that involve re-routing the packets. This is incorrect, as TCP does not have control over the underlying routing protocol.

b. With additional bits, the sender can be much more aggressive with its send rate when it knows the queues are far from full. Multiplicative increase or just more aggressive additive increase would work. Also, the size of each fullness level does not have to be the same. It would make more sense to have fine-grain levels closer to 100% full, and coarse-grain levels when it is almost empty.

c. At the very least, your algorithm should fall back to standard TCP behavior on packet loss in a partial deployment. Additionally, one can potentially infer the fullness of a legacy router if its adjacent routers support explicit congestion signals. For example, if a legacy router is known to drop packets when its adjacent routers are at 70% and 80% full, your algorithm could renormalize its fullness scale where 70% full is equivalent to being 100% full. Also, what about non-supporting clients? They would use normal TCP, which ignores the congestion signals and use standard AIMD for congestion control. Generally, the solution is to just fall back to standard TCP in order to preserve fairness with the existing TCP streams. However, if fairness with standard TCP is not an issue, then your sender can increase its utilization by, for example, ignoring the first *k* packets as a congestion signal in order to wait for the standard TCP to perform a multiplicative decrease first. Again, I gave full marks for answers that simply fall back to standard TCP behavior on packet loss, and gave bonus marks for those that further explored the problem.

3a. The modified RPC and RMI implementation would need to support marshaling the exception and sending that back as the return value in place of the standard return value. This requires changes in the client and server stub, or the object proxy and skeleton. The stubs will also need to catch the exception on the server and re-throw the exception on the client.

3b. The answer is workload specific. For those workloads where the exception is large (uncommon, but could happen), is re-thrown multiple times, and/or the contents of the exception is not actually accessed, then implementing exceptions as remote objects makes more sense. Otherwise, it might be better to just marshal the exception, especially if the exception is used to indicate the server is about to become unavailable.

4a. The answer is straight from the slides and most of you got it right. There was some confusion regarding whether to count the last acknowledgement from the client to the server for asynchronous RPC. I accepted both answers, although I omitted the final acknowledgement as we are measuring time with respect to the client. Sync RPC: 248 ms, Async RPC: 226 ms

4b. For synchronous RPC, the time is the same. For asynchronous RPC, given that you know the response for the first request is either True or False, then it is possible to issue one of the second requests speculatively if the second requests only involve read operations that do not change any state on the server. Assuming you speculate correctly, you should be able to have the same performance as asynchronous RPC in 4a: 226 ms. However, if you speculate incorrectly and assuming you can cancel an existing request (through extermination), then it would be equivalent to synchronous RPC: 248 ms. It is also fine to not assume that an existing request can be cancelled, in which case, the answer would be much larger for incorrect speculations.

5a. No, standard TCP does not provide strict at-most-once semantics as timeouts due to omission failures are not correctly handled. One must also save the return values that were not sent due to connection timeouts, introduce unique request IDs, and have the sender re-send requests with the same ID after a TCP connection is broken. On receiving a repeated request, the server just returns the saved return value.

5b. Again, this is a fairly open-ended question, and the grading was very flexible. To provide "exactly once" semantics, the system must handle client and server failures. This means that, in addition to the mechanisms needed in 5a, the server must also log all requests to disk before executing the request. After a server failure, the server must reload from the last saved snapshot and re-execute the requests in its log – assuming that the requests only affect the state on the server. The RPC IDs will also need to be persistent across client and server crashes, which means they too will need to be logged. As long as you identified most of these problems and tried to address them, you should get full marks for this question.

6a. Routing on a structured overlay is very rigid. The overlay largely dictates which intermediate nodes must be traversed. Unfortunately, these intermediate nodes may be geographically distant, causing very long, circuitous paths that incur very high latency.

6b. Many of you gave min(B_{out}, B_{in}) which does not account for seeders behaving differently than normal peers. It also does not account for different nodes having different upload and download bandwidths. The answer I was looking for involve each node getting an equal fraction of the seeder bandwidth (from one seeder), and a bandwidth proportional fraction of the upload bandwidth from its peers.

6c. How does the RPC implementation know how to marshal a union of two different types? Marshaling an integer is very different than marshaling a floating point number or a string. For this reason, unions are generally not supported by RPC implementations.