CS 537 Handout Nov 19

- Google, Facebook, Amazon -- all instances of distributed systems
- A large number of machines working together to achieve a goal
- Problem: things fail all the time
    - Machines go down
    - Network links go down
    - Disk become corrupt or unusable
- Auxiliary problems:
    - Good performance
    - Security
- Communication Basics
    - Sockets: communication endpoint
    - UDP: unreliable communication
        - uses checksums
    - TCP: reliable communication
- Techniques:
    - acknowledgement
    - Timeout
        - How to set the value?
    - Retry
- Semantics
    - Exactly once
    - Sequence counter
- Communication Abstractions
    - Distributed Shared Memory
        - Failure means part of your address space is gone!
        - Performance is low due to page faults and remote fetches
    - Remote Procedure Call (RPC)
        - Just like a local procedure call
        - Stub generator
            - Turn function calls into messages and back
            - RPC Call steps (on client)
                - Create a message buffer, pack contents into it (Marshalling)
                - Send message
                - Wait for reply, resend on timeout
                - Unpack return code (Unmarshalling)
                - Return result to caller
            - Similar steps on the server
        - Run-time Library
            - Naming: where to find the server?
            - TCP or UDP?

- RCP provides atmost-once semantics
- How long should client wait for server?
    - Handled with periodic Yes
- How to handle large arguments?
    - Split up at sender, re-assemble at receiver
- How to handle different architectures?
    - Little endian/big endian?
    - external Data Representation (xDR)
        - convert data to this format
        - convert back at sender/receiver
- Synchronous/asynchronous?

Questions

1. When building a distributed system, what is the key thing to keep in mind?
2. Why are atmost-once semantics useful? Give a few examples where at-most once semantics are required for correctness.
3. Why do companies like Google use thousands of machines instead of a single, highly reliable supercomputer?
4. Given the way TCP works, is it better to send a lot of tiny messages or pack it all up into one large message and send?
5. What happens if the TCP timeout value is set too small? What happens if it is set too large?
6. Suppose you have a library L. You are curious whether function calls in L are executed locally or remotely. How might you figure this out? What tools would you use?
7. Why is TCP a bad fit for applications that use request/reply?
8. Suppose there is a magic machine that has a million threads. Would using this be better than using a distributed system with a million nodes? Why or why not?
9. What are the disadvantages of distributed shared memory?
10. Suppose a server X exports two functions: A and B. A takes 10 seconds, while B takes 10 minutes to complete. When client Y sends a message to X, what is a good timeout value to set for receiving a response? What other techniques are used to handle this?