|  |  |
| --- | --- |
| **GDPARCM Lecture – Fault Tolerance and Recovery** | Instructor: Neil Patrick Del Gallego |

**The eight fallacies of distributed computing**

1. The [network](https://en.wikipedia.org/wiki/Computer_network) is reliable.
   * There will always be packet drops, connection interruptions, and possible data corruption.
2. [Latency](https://en.wikipedia.org/wiki/Latency_(engineering)) is zero.
   * Latency is not zero. While internet speed have vastly improved, there will always be delays in sending/receiving messages and this cannot be predicted.
3. [Bandwidth](https://en.wikipedia.org/wiki/Throughput) is infinite.
   * A design paradox 🡪 to reduce latency, we transfer more data to reduce network communication between two PCs. E.g. one-time, big-time send of all the necessary data for processing. But bandwidth is finite! Transferring more data, consumes more bandwidth.
   * Must address and find the balance between #2 and #3 fallacies.
4. The network is [secure](https://en.wikipedia.org/wiki/Computer_security);
   * Ensure your data is encrypted. Any user could possibly hijack your system. E.g. SQL injection.



1. [Topology](https://en.wikipedia.org/wiki/Network_topology) doesn't change.
   * What if your host/domain changes? Moving from a new server location. If IPs are hardcoded, then all clients must have updated IP directories.
   * What if one of the services fails? How do we route incoming connections? E.g. MMO -> Login system working, but item store services are not 🡪 should still be able to play the game, minus access to the online shop.
2. There is one [administrator](https://en.wikipedia.org/wiki/Network_administrator);
   * There can be many teams/developers for each service.
3. Transport cost is zero.
   * Related to #2. There will always be a cost in transporting data. E.g. Read/write operations in server disks, usage of virtual machines, GPU rentals, etc. Goal is to maximize usage, minimize cost.
4. The network is homogeneous.
   * The network is heterogenous 🡪 Many services have different software/hardware configurations. One must standardize the messages being passed across network 🡪 XML, JSON, protocol buffers.

**EXAMPLES OF HOW TO ADDRESS SOME OF THE FALLACIES**

**Addressing #1: Promoting network reliability**

Context: Without considering timeout/error handling, a client requesting a service from a server can be assumed to wait forever.

Refer to GDPARCM Sockets Demo. This example doesn’t support any timeout mechanism and assumes an indefinite connection between server-client.

In gRPC, any RPC call has a default timeout. The call returns a failure if no response from server was received. The default timeout, per client request is 1 second. To modify/extend the timeout, call **set\_deadline()** before making any RPC call.

|  |
| --- |
| context.**set\_deadline(std::chrono::system\_clock::now() + std::chrono::seconds(5));**  result = stub\_->CreateFolder(&context, request); |

A screen shot of a computer program

Description automatically generated

Documentation: <https://grpc.io/blog/deadlines/>

* Servers could also set deadlines.
* A server might receive requests from a client with an unrealistically short deadline 🡪 <1 second deadline, but requires a 100MB file. To address this, the server application has the last say whether or not to proceed with the request.
  + A gRPC server automatically cancels a call once a deadline set by the client has passed (CANCELLED status).
  + The server application is responsible for stopping any activity it has spawned to service the client request 🡪 periodically check if the request initiated has passed the deadline (CANCELLED status).

**Deadline Propagation**

Your server might need to call another server to produce a response. In these cases where your server also acts as a client you would want to honor the deadline set by the original client. Automatically propagating the deadline from an incoming request to an outgoing one is supported by some gRPC implementations. In some languages this behavior needs to be explicitly enabled (e.g. C++) and in others it is enabled by default (e.g. Java and Go). Using this capability lets you avoid the error-prone approach of manually including the deadline for each outgoing RPC.

Since a deadline is set point in time, propagating it as-is to a server can be problematic as the clocks on the two servers might not be synchronized. To address this gRPC converts the deadline to a timeout from which the already elapsed time is already deducted. This shields your system from any clock skew issues.

A diagram of a diagram

Description automatically generated

**Retry Policy**

A common solution is to implement a **retry policy 🡪,** periodically request for service to the server, then provide a graceful failure handling strategy if numerous retries are made.

A screenshot of a computer code

Description automatically generated

**Addressing #3: Assuming bandwidth is finite, make message buffers and service requests as small as possible.**

An example of a violation of #3 – a mini-Spotify web player. Use case: Get a list of user songs and their MP4 files all at once.

A screenshot of a computer code

Description automatically generated

* The Song message includes an **audio**\_**data** field of type bytes.
* This field represents potentially large binary audio data associated with each song.
* The **<large\_binary\_audio\_data>** represents a substantial amount of binary audio information.
* If this field is transmitted frequently between the client and server without proper optimization, it could lead to increased bandwidth usage and potential performance issues, violating the fallacy that "Bandwidth Is Infinite."

Possible solutions:

1. **Streaming:** Stream the audio data separately from other fields if it's not always needed.
2. **Compression:** Compressing the audio data before transmission to reduce its size.
3. **Caching:** Using caching mechanisms to avoid unnecessary repeated transfers of the same audio data.
4. **External Storage:** Storing large binary data externally (e.g., in a cloud storage service) and transmitting only references or URLs in the protobuf message.

Sample solution using gRPC bidirectional streaming approach:

A computer code with numbers and symbols

Description automatically generated

Server implementation:

A screenshot of a computer program

Description automatically generated

Client implementation:

A screenshot of a computer code

Description automatically generated

Explanation:

* The Song message now includes an **audio\_reference** field, which holds a reference or URL to the audio file.
* The **MusicService** service introduces the **StreamSongs** bidirectional streaming RPC, allowing the client to request and receive songs in chunks.
* The server's **StreamSongs** implementation reads the **SongRequest** from the client, fetches song details (including audio references), and streams the songs back to the client.
* The client's **StreamSongs** method sends a list of song IDs to the server and receives songs in chunks, processing each song as it arrives.

This approach allows the client and server to efficiently exchange references or URLs to audio files, reducing the impact on bandwidth and improving the efficiency of data transfer.

**RECOVERY**

* Attempts to addresses #1 and #3. Constant occurrences of #1 and #3 will cause numerous connection faults which could lead to cascading failures 🡪 a client requesting for service #1, but service #1 needs service #2, #2 needs #3,…. #3 needs #N. If as early as service #1, the request isn’t successful, failures will cascade across the network.
* Context: If **K** clients requests for service #1, but service #1 hasn’t successfully processed **K/50** (busy/unavailable), then it is not wise to repeatedly ping service #1 🡪 K clients must reroute/request service on other available servers.

**Additional background:**

* Consider a P3 scenario.
  + Your AI web viewer hung up and needed to restart, but your server keeps sending back images to the AI web viewer 🡪 the web viewer gets bogged down with several requests that cannot be processed.
  + Your 3D model viewer keeps requesting for “scene A” that isn’t available yet 🡪 processing “scene A” gets further delayed because numerous duplicate requests get sent.

**The circuit breaker design pattern**

* Inspired by an electrical circuit breaker. A circuit breaker design pattern prevents cascading failures, and promotes rerouting of requests.
* A circuit breaker instance monitors the status of the remote service. When a certain threshold of failures is recorded in the circuit breaker, it “trips/opens,” preventing any further requests from being sent to that service.

Circuit breaker class template:

A screenshot of a computer program

Description automatically generated

Sample usage. Server using circuit breaker instance:

A screenshot of a computer program

Description automatically generated

* The **CircuitBreaker** class tracks failures and decides whether the circuit should be open or closed.
* The **RemoteOperationsServiceImpl** class simulates a gRPC service that may fail under certain conditions.
* When the circuit breaker is open, the service returns an **UNAVAILABLE** status, simulating a failure.
* The circuit breaker records failures and successes based on the service's responses.

More sophisticated and dedicated libraries are available: Netflix Hystrix, Resilience4j, Sentinel.

C++ libraries

* Cpp-httplib: <https://github.com/yhirose/cpp-httplib>
* Envoy: <https://www.envoyproxy.io/>

**Advantages of circuit breaker**

* Efficient way to halt future client requests that are guaranteed to fail if the service is unavailable.
* Opportunity to reroute requests 🡪 circuit breaker #1 trips 🡪 reroute K clients to another service (with an associated circuit breaker #2).
* Allows unavailable services to focus their computing resources for restarting. E.g. rebooting.

**Disadvantages of circuit breaker**

* Complexity 🡪 N services = N circuit breakers?
* Configuration-dependent and educated guessing of thresholds 🡪 timeouts, recovery times, number of failuress require careful tuning to maintain the quality of the underlying services.
  + Overly aggressive thresholds lead to false positives, causing the system to unnecessarily go into a degraded state.
* Overhead 🡪 a circuit breaker is typically a running thread of its own.