

# 基于 gRPC 的实时消息广播网关

## 一、功能概述：

系统包括两部分：

1. 服务器 (Server)：接收数据发布 (Publish) 请求并将从指定序列号的数据广播到所有订阅者。收到的 Frame 按照序列号大小和时间戳大小 (序列号相同时) 排序广播给订阅者。提供 Publish 和 Subscribe 接口。
2. 客户端 (Client)：支持同时订阅服务器推送数据并发送新数据帧，客户端通过并发模拟大规模连接压力以测试服务器性能。

## 二、技术难点：

### 1. 并发的管理可能造成的读写冲突：

使用 `std::unordered_map` 结构存储当前连接的客户端，采用递增的 `next_client_id_` 作为连接标识。多个线程同时读写时，例如一个线程广播消息进行读操作，一个线程新建订阅者并接收订阅者发送的 Frame 进行写操作，可能会导致读写冲突；

收到的 Frame 使用 `std::vector` 存储，同样面临资源竞争的问题；

在所有对 `frame_` 和 `subscribers_` 数据操作的时候加锁，考虑 RAII 原则，使用 `std::lock_guard` 加锁，保证在作用域结束后自动释放。

### 2. 服务端稳定性：

客户端随时可能断联，断联是如果服务端正在发送 Frame，如果没有异常处理机制，会导致服务端写入数据失败而导致崩溃；

因此在 Publish 广播时，对 Write 操作进行检测，Write 返回 false，则记录错误日志，不中断整体广播流程，并且在监听 `context->IsCancelled()`，检测客户端断开，并及时移除订阅者。防止造成资源泄露。

### 3. 历史消息发送：

要求升序的发送所有 `seq_no` 域不小于订阅者的 `replay_seq_no_start` 的历史 Frame，对于 `seq_no` 相同的情况，在 Frame 结构里增加了一个 `int64 timestamp = 3` 字段，排序时按照时间戳升序排序。

在这一部分，需要对历史数据根据 `seq_no` 和 `timestamp` 字段进行排序，考虑到如果加锁，排序可能时间较长阻塞其他进程从而不能高效的进行数据发送，因此在这里进行了一次数据的复制，对复制的数据进程排序和广播。

### 4. 客户端封装 gRPC 通信的实现细节：

考虑到客户端需用让用户无感知，因此需要对 `subscribe()` 和 `publish()` 方法以及订阅和发布线程进行封装，使用 `run()` 方法负责启动订阅和发布线程，并自动等待它们的完成。这样用户不需要手动管理多线程，只需调用 `run()`，提供服务端地址，想要发送的数据帧信息，和想要接收的数据帧信息（即 `replay_seq_no_start`）即可。

## 5. 不足和优化的方向：

由于时间问题，这个项目只是实现了基本功能，还存在一些潜在的问题我认为可以进行优化：

（1）服务端没有一个优雅的关闭连接的方式，可以提供接口去捕获系统关闭信号，在关闭前通知客户端，防止出现资源浪费或者资源泄露的情况。

（2）在对历史数据进行排序时选择复制一份数据，如果数据量大的话反复复制和排序操作会非常的耗费资源，而造成性能瓶颈，考虑可以选择更优的数据结构，在数据到达时进行一次排序，比如说平衡二叉搜索树，插入时间复杂度为  $O(\log N)$ ，不需要进行接下来的复制和排序，但是由于时间原因，没有进行实现。

## 三、扩展：

1、服务端对历史 **Frame** 做持久化存储，首先最简单的是将收到的数据写入一个文件中，发送的时候从文件中读取，但是这种反复进行磁盘读写的效率很低，因此考虑数据库，考虑到场景中会有多个客户端并发的发送 **Frame** 数据，并且要求实时的广播 **Frame** 数据，可以考虑 **redis** 内存数据库做 **Frame** 数据持久化，**redis** 支持高并发写入和低延迟读取，提供了 **AOF** 和 **RDB** 两种持久化策略，并且相较于其它内存数据库，提供了更多的数据结构适合存储，适合实时消息广播的场景。

2、考虑到广播的数据是按照 `seq_no` 和时间戳进行排序的，可以使用 **redis** 提供有序集合 **Zset** 进行存储，每条 **Frame** 使用 `seq_no` 作为 `score`，**Frame** 作为 `value`，并且 **Zset** 底层使用跳表或者哈希表结构实现，查找和插入删除的复杂度都是  $O(\log N)$ ，可以解决我上面提到的复制和排序带来的性能问题。

3、**redis** 提供两种持久化策略，**AOF** 日志和 **RDB** 快照，考虑安全性和性能兼顾，可以使用两种策略混合使用的持久化策略，即先把当前内存快照（**RDB** 格式）写到新的 **AOF** 文件的开头，然后在此基础上继续把后续增量写操作继续追加到 **AOF** 文件后面。

4、对 **Frame** 做持久化的流程为每次 **Frame** 到达时，将 **Frame** 写入一个 **Zset** 格式的 `frame_` 结构中，写入命令被写入 **AOF** 日志中，当 **AOF** 文件增长到一定大小后，触发 **AOF** 重写，把当前内存数据使用 **RDB** 格式写到新的临时 **AOF** 文件。同时异步的捕获重写期间新的写命令，记录在内存增量缓冲区。等 **RDB** 写完后，

把增量命令也追加到临时 AOF 文件后面。最后用新 AOF 文件替换老的 AOF 文件。

5、考虑到 Frame 数据安全性比较重要，AOF 文件可以使用 Always 写会策略，每执行一条写操作命令就同步将 AOF 内容写回硬盘，虽然性能比较差，但是数据安全性高。

6、当 Redis 重启时，直接用这个混合格式 AOF 文件恢复，先用 RDB 内容快速恢复内存状态，再应用 AOF 里后续的增量命令。

## 四、AI 助手使用：

由于平时接触这方面比较少，题目里有很多描述不能很好的理解具体要做什么，询问 AI 帮理解。

### 1、问题：

客户端(生产者)可以(单次调用,无状态地)向服务端发送一个带有 sequence number 的 buffer，当前在服务端已经订阅的客户端（消费者），如果其订阅时指定的 replay\_seq\_no\_start 小于 等于生产者提交的 seq\_no，便会收到该 Frame；帮我解释这句话 单次调用 无状态 展示要求的流程

### 回复：

整体意思总结一下：

生产者每次都直接发 (seq\_no, buffer)，不管历史；服务端收到后，判断哪些订阅者需要这个 buffer（按 replay\_seq\_no\_start 条件）；符合条件的消费者收到这个 Frame。

---

你要的【流程图式】展示，简洁版如下：

生产者 (Producer) ——发送(seq\_no, buffer)——> 服务端 (Server)

服务端内部流程：

1. 接收 buffer + seq\_no
2. 遍历已订阅的消费者列表
3. 对于每个消费者：
  - 如果 consumer.replay\_seq\_no\_start <= seq\_no:
  - 推送该 Frame 给该消费者

---

要不要我再画一版时序图（sequence diagram），让它更可视化一点？

要的话告诉我！

### 2、问题：

Python 客户端(1-2 小时)包装出比较合理的让用户对 grpc 无感知(除了 server endpoint 配置等)的接口。比如一个吐出 bytes 的 generator 解释这段话 让用户对 grpc 无感知

**回答:**

这段话的意思是, 设计一个 Python 客户端接口, 目的是让用户在使用时不需要关心 gRPC 细节, 只需要调用一个高层次的接口, 像常规的 Python 函数那样方便, 而不需要知道它背后如何实现与 gRPC 的通信。

**3、问题:**

如果出现了多个相同的 seq\_no 值, 可以随意 tie-break 解释

**回答:**

在这句话中, 提到的“多个相同的 seq\_no 值”指的是在某个操作或系统中, 可能会出现两个或更多的数据包或请求具有相同的 seq\_no (序列号)。而“随意 tie-break”则意味着对于这些相同的 seq\_no, 你可以根据具体需求或业务逻辑来决定它们的处理顺序或优先级。