Faculdade de Engenharia
Universidade do Porto

Reliable Publish-Subscribe Service

Large Scale Distributed Systems, Assignment 1

Class 3 Group 15
Carlos Gomes up201906622@edu.fe.up.pt
José Costa up201907216@edu.fe.up.pt
Pedro Silva up201907523@edu.fe.up.pt
Sérgio Estevão up201905680@edu.fe.up.pt

Sunday 23rd October, 2022

Contents

Chapter 1

Introduction

The publish-subscribe message pattern allows for asynchronous/loosely coupled communication between message senders and interested message receivers. It's normally used as part of a larger MOM (Message oriented middleware) system.

Messages are aggregated in topics of interest which receivers subscribe. In this context they are named Subscribers of a given topic and will receive any message published on that topic. Message producers post messages on a given topic, making effectively sending them to subscribers. Here the producers are refered as publishers.

A pub/sub system designed in this way cannot guarantee delivery of messages to any applications that might require such assured delivery. However, reliability in pub/sub is still desirable.

Chapter 2

System Architecture

The system follows a client-server architecture (Fig. 2.1). The clients are the topic subscribers and publishers. The server is the message broker that receives client requests to subscribe/unsubscribe a topic and get/put a topic update, sending the proper replies to the clients.

A client-server model is simpler to implement than a P2P model but introduces a single point of failure and makes it harder to scale the system.

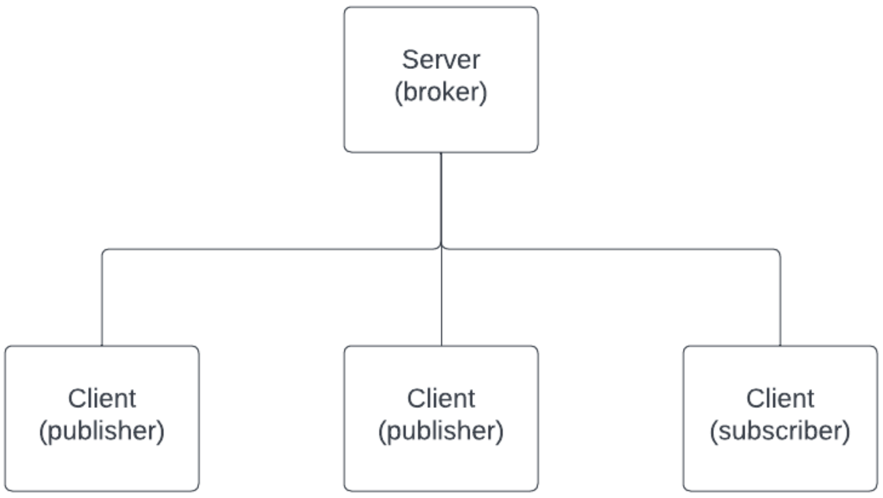

Figure 2.1: Client-Server system architecture

2.1 Model

2.2 Distributed System Model

Since the service specification was left open in its model restrictions, we present the distributed system model taken into account when designing the fault tolerance.

In terms of synchronism, a system can have a synchronous model, which means that all network, process and clock time intervals are bounded, or asynchronous where the times aren't bounded. Naturally, this system is easier to build on networks like LANs than on networks like the Internet, where packet delivery times can vary greatly. We try to build an asynchronous system.

When it comes to failures, we tolerate that the components of the system can suffer from crashes, message omission and timing failures. For simplicity's sake, we do not deal with Byzantine Faults, which matter less since this service doesn't have a consensus requirement between components.

2.3 System Assumptions

We present the assumptions and restrictions the system must comply with:

1. Clients and servers can crash and recover.
2. Network problems (i.e partitions) may make communication between the client and server impossible during a finite time interval.
3. Subscriptions are durable, which means that a topic's subscriber should get all messages put on a topic, as long as it calls get() enough times, until it explicitly unsubscribes the topic. This should be guaranteed even if the subscriber runs intermittently (I.e. a subscription is independent of the lifetime of the process that makes that subscription).
4. Both the client and server have non-volatile storage that does not get corrupted.
5. The service should guarantee "exactly-once" delivery in the presence of all faults the system is meant to tolerate. This means that on successful return from put() on a topic, the service guarantees that the message will eventually be delivered "to all subscribers of that topic", as long as the subscribers keep calling get() and that on successful return from get() on a topic, the service guarantees that the same message will not be returned again on a later call to get() by that subscriber.

Chapter 3

Implementation details

3.1 Software

The system was written in the Rust programming language, offering (code) safety guarantees and good performance.

The communication between components uses the ZeroMQ Rust bindings.

For JSON serialization and deserialization, we use the Serde crate.

3.2 Serialization and Disk Access

When writing the state to disk and sending messages between the system's components we opted to serialize everything as a JSON string, promoting good message readability in logs and tools like Wireshark. Using Serde is also convenient because we don't have to manually parse the message's contents and create a Rust object with them which, which helps if a message has a lot of optional fields.

For simplicity's sake, we write the server's state in a single JSON file, an inefficient process because every time the state changes (i.e a new subscriber is added to a topic), he serializes (a relatively expensive operation) and saves all unchanged topic queues and subscriptions without needing to do so. The efficient way is dividing the state through different topic directories and client sub-directories.

3.3 Messages

As was already said, all messages are serialized as a JSON string before being sent to the channel and then deserialized as a Rust object on the receiving side.

All client request messages include the client's IP which is used as a client unique identifier.

Message	Parameters	Use	Transport
Get Update	• Client IP • Topic • Sequence Number	Client requests a topic update	TCP
Put Update	• Client IP • Topic • Payload • Sequence Number	Client requests to put an update ona topic	TCP
Subscribe Topic	• Client IP • Topic	Client requests to subscribe a topic	TCP
Unsubscribe Topic	• Client IP • Topic	Client requests to unsubscribe a topic	TCP
Client Up	• Client IP • Sequence Numbers	Client informs the Server that he was crashed	TCP
Reply	• Update (on Get) • Service error (on error)	Server's reply to a client request	TCP

Table 3.1: Message types

3.4 Data Structures

- Update {content, pending_updates} - represents a topic update; the pending_updates is a counter of how many clients the service isn't that the update reached.
- SubscriptionInfo { last_recv_sequence_num, topic_update_idx } - contains the topic subscription information for a given client. The last_recv_sequence_num is the last sequence number the server received from a get request and the topic_update_idx is a pointer to the next topic update the client will receive.
- TopicInfo { subscriptions, update_queue } - contains a topic's subscriptions and update queue.
- Message (GET | PUT | SUB | UNSUB | UP | NOMSG) - represents a message.

3.5 Client

3.5.1 On start up

When the client starts he reads from the disk his previous state, if applicable. If he had state previously, he sends an UP message.

3.5.2 Sending requests

The Client reads the user input. After parsing the input it builds the message. Get, Subscribe and Unsubscribe messages take as arguments the topic name. Put message have two additional arguments: 1. a boolean that if set to true means the user is sending a repeated message, otherwise it means the user is sending the Put message for the first time; 2. and the message to be stored in the Server.

Either when sending a Put or a Get, it verifies if the sequence number of each operation for the specific topic is initialized. If they are, then they are incremented, otherwise initialized, and then the message is sent to the Server with this information. When sending Subscribes and Unsubscribes, the Client doesn't require sequence numbers.

3.5.3 Responses to the requests

Subscribe, Unsubscribe and Put messages are replied to by the server with a confirmation or error. Get messages are replied to with the sequence number to ensure the state between the server and client is synchronized, if not, the Client set its state equal to the Server.

3.6 Server

3.6.1 On start up

When the server starts he reads from the disk his previous state, if applicable.

3.6.2 Receiving a SUB/UNSUB

The server checks if the topic exists. If it doesn't exist, he creates it. If it exists, then he checks the client's subscription state to know if he can subscribe or not. A reply is sent with the appropriate service error.

3.6.3 Receiving a GET

The server checks if the topic exists. If it doesn't exist, he sends an error. If it exists, then he checks if the client is subscribed, sending an error if not.

The request sequence numbers are checked to see if a new or old update is being requested. When a new sequence number is received, the server can mark the previous update sent as received by the client and shift the queue pointer to point to the new update to send.

The server garbage collects an update when there 0 pending clients for that update.

3.6.4 Receiving a PUT

The behaviour is similar to receiving a GET, with the exception that the exactly-once is in not putting the same update on the queue more than once.

3.6.5 Receiving an UP

The server updates all sequence numbers on his side.

3.7 Fault tolerance

3.7.1 Loss of messages

A loss of a SUB and UNSUB request and the server's reply can happen. However, the server's state won't become inconsistent with request retries: he always knows a client's current subscription state, which means he can only subscribe if he isn't unsubscribed (or vice-versa). The dilemma is that the client can only be sure of his subscription when a future GET or a SUB/UNSUB request returns a server reply indicating that he is unsubscribed/isn't subscribed. However, we don't want to make the call block until he receives a reply.

A loss of a GET request is not problematic (exactly-once is assured). Sequence numbers on the client side allow the service to keep track which update he is requesting. If the server receives a new sequence number, it means the client wants a new update. If the server receives a request with the same sequence number, it indicates that the client didn't receive the previous update (i.e the client sent a request but didn't receive a reply because the server was crashed).

A loss of a PUT request is catastrophic. We don't make sure if a PUT was received by the server. This means that if the user tries again than the update will be replicated on the update queue.

A loss of an UP request is not dealt with. We never make sure if the server receiving this. However, this is not a problem because this message is just an optimization to make the server garbage collect old updates. He can still do it on subsequent GETs.

The loss of a REP reply doesn't generate request retries on the client-side. The operation blocks waiting for a response until a given timeout is reached.

3.7.2 Crashes

Crash fault tolerance is the same as message loss fault tolerance.

When the server crashes before sending a response, no retries are sent (besides the ones ZeroMQ sends).

3.7.3 Topic Availability in Server Crashes

Without topic sharding between servers and topic replication, in the presence of server crashes all topics become unavailable until the single server recovers. The sharding, however, doesn't increase availability in the same way replication does: a topic owned by a crashed server can become unavailable with the sharding but not if replicated (provided atleast one topic replica is available).

The topic sharding can be implemented with a proxy server that demultiplexes requests to servers and multiplexes their responses back to the clients. This proxy should guarantee that only one server owns a given topic unless replication comes into play. The server, however, infers topic ownership if he receives a subscribe/put request of a new topic, provided that the proxy isn't tricking him.

3.7.4 Keeping state and Garbage Collection

The implementation requires both the client and the server to keep the state in stable storage. The amount of state the client has to save on disk is minimal: sequence numbers and topic names. Now, the server keeps subscription information for each client (with sequence numbers associated) and an update queue for each topic. The updates in pub-sub are generally low in size (up to MBs of information). Having the client to keep the state in disk is a limitation because the service might not have permission to write files.

The client can garbage collect its topic sequence numbers whenever he unsubscribes. The server can garbage collect the subscriptions whenever a client unsubscribes. An update can only be garbage collected when the server has an assurance that all subscribers received it.

A limitation of using a new get sequence number to acknowledge the previous get request has the limitation that if a client doesn't request an update from a topic regularly, then the server won't garbage collect it because he is waiting for a new sequence number on the next request. This way, the updates start piling up on the server queue and can cause a memory overflow. The server can apply mechanisms to detect these "dead" updates, only keep them saved on disk and retrieve them when necessary.

3.7.5 Loss of state

Should a system component crash before or while writing its state to disk, the service does not guarantee to save all data before crashing. However, the underlying operating system might have mechanisms to attenuate this. Furthermore, it's a rare event by nature. With relatively small processing times per request, a component quickly starts to write to disk.

If we drop the fourth assumption, disk corruption would be catastrophic because both the client and the server would lose all required states to achieve exactly-once update delivery. The server would also lose all topic information (subscriptions and updates). Hardware and software redundancy mechanisms like disk backup help minimize this risk.

3.8 Concurrency Model

The server is single-threaded. The processing times for each request are low, so the bottleneck is disk access.

We can use multithreading to process multiple requests concurrently. However, a single topic's requests require processing by only one thread at a time to prevent inconsistencies with its update queue and subscriptions and resulting race conditions. We can use a personalized thread pool that dequeues a request of a topic not being worked on yet. It prevents a thread from doing any work because another thread has already locked that same topic. Profiling this mechanism will check if it is introducing, on average, an overhead more costly than just having threads locked in the thread pool.

3.9 Scalability

With no concurrency, the server can only answer a limited number of requests per second (one each time). As the number of clients grows, the requests start piling up on the service buffers, and the server needs to keep more state (subscriptions and updates). Thus, the system is not scalable size-wise.

The system isn't scalable geographically. The server couldn't be transparent about the increased (and varied) message travel times that increasing the client area would create.

3.10 Security

On the Internet, the number of untrustable machines makes the system vulnerable to data being stolen, modified, or spoofed. TLS adds a security layer to prevent this and attacks like TCP Reset and TCP Session Hijacking.

If the system runs on a trustable and secure LAN, then TLS is not required.

Chapter 4

Conclusion

We achieved fault tolerance on the get() but at the cost of saving state on the client. put() doesn't have idempotence but so get() guaranteeing "exactly-once" delivery is meaningless.

The implementation is not scalable to more users, resources and geographically dispersed clients. A multi-server architecture with server concurrency will offer more scalability.

Topic replication will give more system scalability and availability. An update never changes its contents, so issues like replica consistency don't exist. The challenge here is to garbage collect all replicas when they aren't needed anymore.

We found the ZeroMQ library to be completely over-engineered, user unfriendly and at times unorthodox.