Fog Computing

SoSe 2022

Prototyping Documentation Report

János Brodbeck
Maximilian Stendler

Submission date: 13.07.2022

We have implemented a client and server application prototype to fulfill fog computing specific challenges.
This prototype is based on the volcano scenario from our last semester's FogDataBench project[1]: The prototype's general idea and project structure is adapted from the FogDataBench sensor[2]. (Which was written in Go, compared to this Java implementation, and did not focus on reliability). We use gRPC[3] for message definition and exchange and SQLite[4] for logging events to disk.

Architecture

The client consists of several threads. A thread generates the sensor data events containing sensor UUID, volcano name, data point UUID, seismic data (x,y,z), a timestamp and an Adler32 checksum (similar to CRC). These events are written into an SQLite database. Another thread handles the client application main loop composed of the following three processes as presented in Figure 1: scheduling events to send, checking for and handling received events and state management.

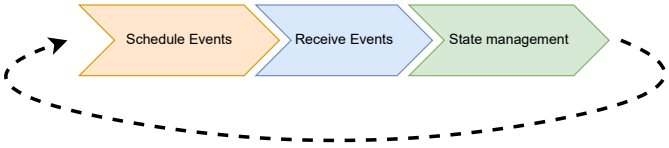

Figure 1: Flowchart showing the main loop of the scheduler of the client

The *schedule events* process, as portrayed in Figure 2, manages a thread pool of blocking gRPC clients. It scans the database for events, that do not have a received timestamp, and registers events with clients until either the pool of clients or the list of to be scheduled events is exhausted. Already scheduled events are remembered and cannot be rescheduled until their client receives a response or timeouts.
After that, the *receive events* process is entered. Figure 3 shows an overview of its steps. Events with successful responses are updated with a received timestamp in the database. Instead of a response, the gRPC call may throw an exception containing a status code. The codes `OK` and `ALREADY_EXISTS` are handled like a successful response. If any otherwise unhandled error occurs, an incident counter is incremented. In any case, the client thread is added to the list of free clients again.

For a real application, we would have a lot of sensors running and sending data to the same server. In case of failure or outage, all clients would try almost at the same time to reconnect and catch up with their data. This could lead to server overload, wherefore we implemented three states for our application - normal, failure and recovering - to adapt transmission to different situations. Figure 4 displays the failure behavior. When a threshold of consecutive incidents occurred, the client switches to failure state and is slowing down their traffic to the server. With each consecutive failed transmission, the client waits longer until the next transmission, this time is called slowdown time. This slowdown time is combined with a jitter. On successful response again, the server goes into recovering state in which it slowly reduces the slowdown time again until it is back to normal again. Then it switches back to normal state. Figure 5 depicts this recovery process.

The server receives the evens and first validates the checksum and checks, if it already received that event, based on its data point UUID. If both checks are successful, the event is inserted into the server's SQLite database and a response is sent. If either check fails, an exception is thrown, which is propagated by gRPC to the client.

[1] `https://git.tu-berlin.de/fogdatabench`
[2] `https://git.tu-berlin.de/fogdatabench/sensor`
[3] `https://grpc.io/`
[4] `https://sqlite.org/index.html`

Message Delivery Reliability

We tested the behavior of our application regarding reliable message delivery with traffic control (tc)[5] using tcgui[6] on the server's network interface.
The following network disruptions are handled gracefully by the application. Packet loss, duplication, and reordering are handled by TCP. Packet corruption is either handled by the underlying protocols, if their invariants are affected, or, in the case of data corruption, the detection is handled via the checksum. Corrupted events have to be resent. On disconnect or server crash, the clients still store their generated events locally and resend them on reconnection.
Not reliably handled are client crashes, since the client produces the events, which it will stop generating in case of a crash, and inability to write to the database. If the disk is full or any other reason prevents writing to the client's database, the client will crash and stop generating events, as there is no in-memory queue implemented as a fallback.

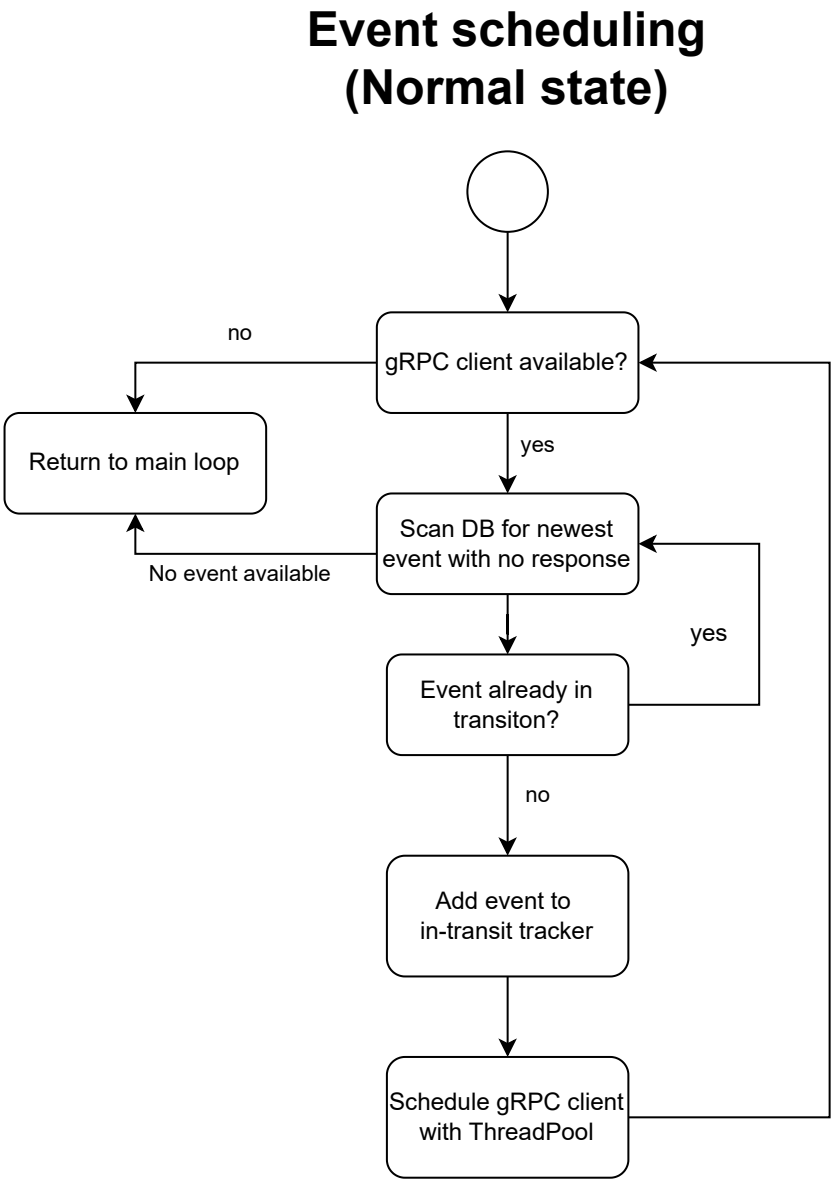

Figure 2: Flowchart showing the event scheduling of the client

[5] https://wiki.debian.org/TrafficControl
[6] https://github.com/tum-lkn/tcgui

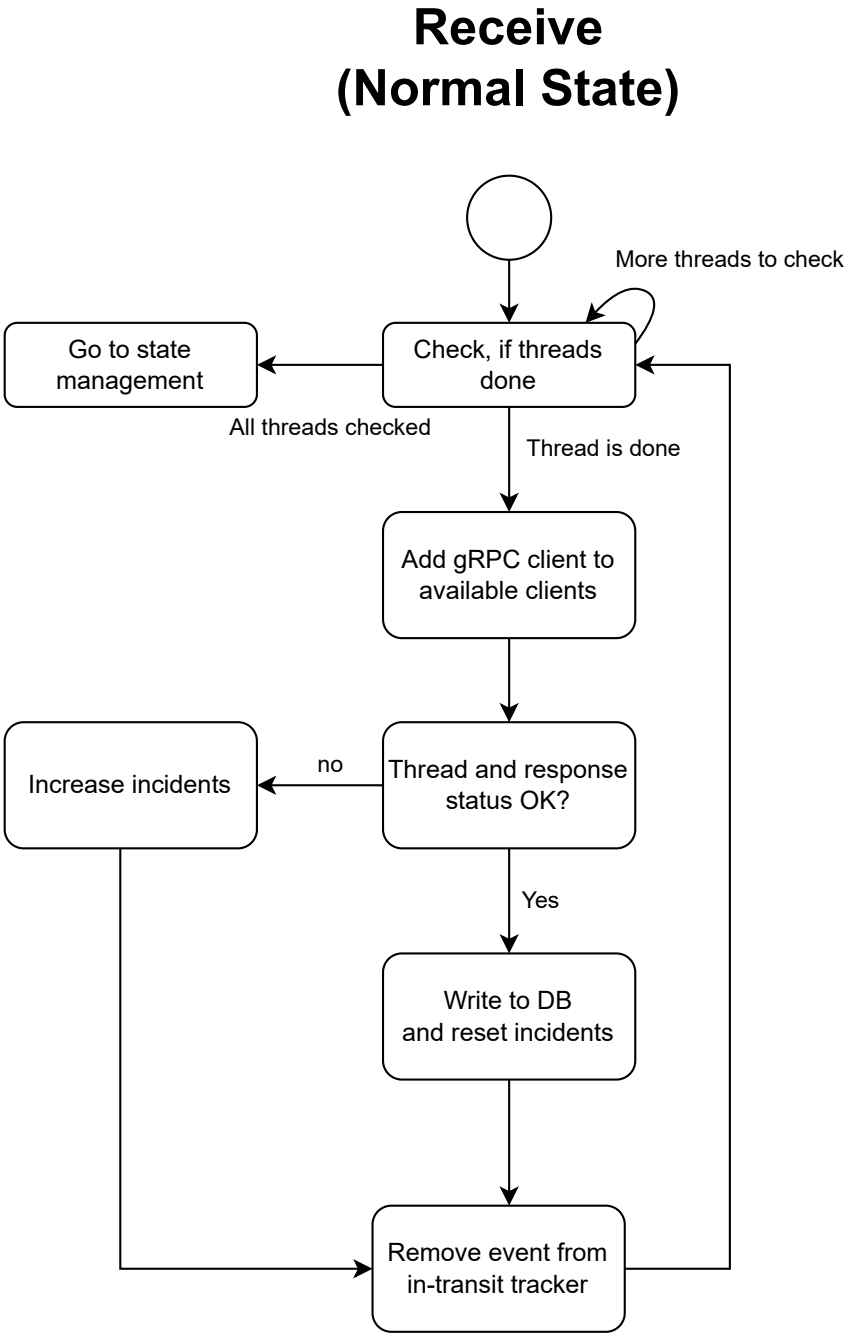

Figure 3: Flowchart showing the response receiving of the client

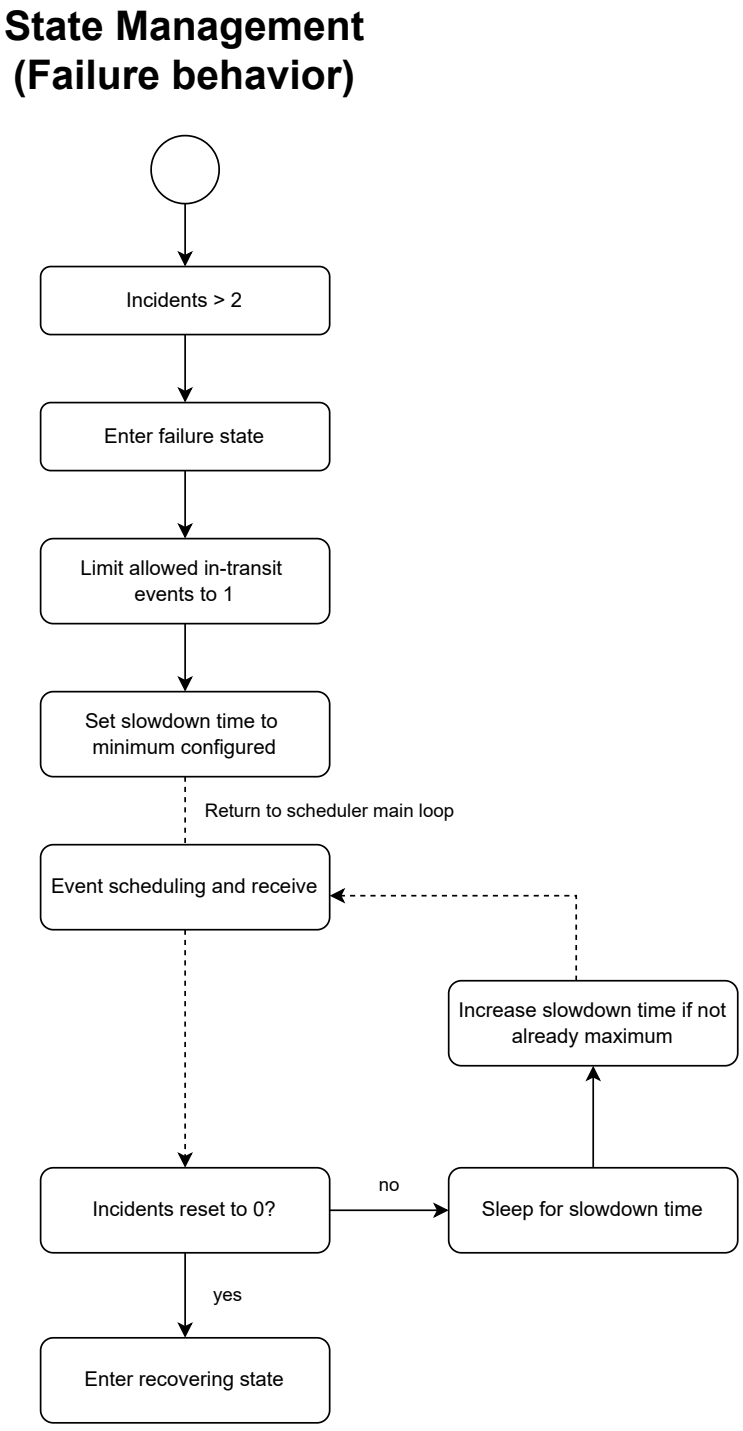

Figure 4: Flowchart showing the process of the failure behavior of the client

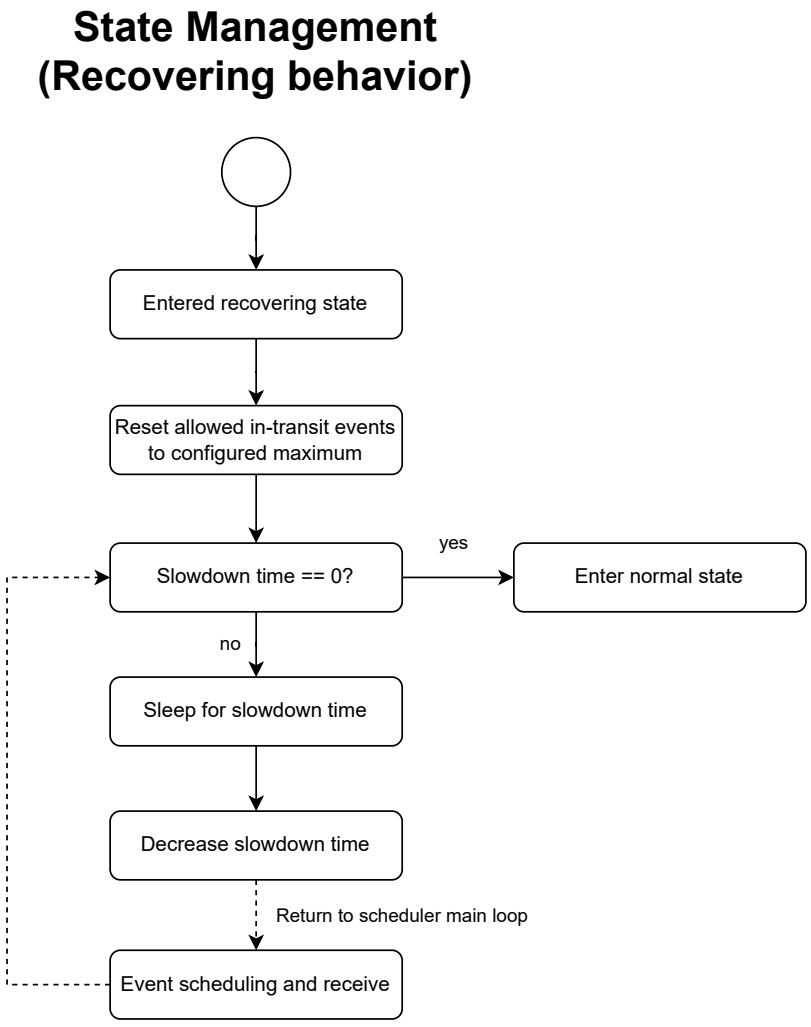

Figure 5: Flowchart showing the recovering process of the client