CHALMERS | 

GÖTEBORGS UNIVERSITET

Miniproject: Distributed systems

Performance and Fault Tolerance Documentation

Group 5

Emanuel Dellsén - emanuel.dellsen@gmail.com

Hannes Ringblom - gusringbha@student.gu.se

Adam Klevfors - gusklevad@student.gu.se

Niklas Möller - gusmolnia@student.gu.se

Karl Westgårdh - karl.westgardh@gmail.com

Application Resilience Feature	**MQTT Protocol Feature**	**Application handling needed**	**Implemented /used in system**	**Comment**
Quality of Service	✓	✗	✓	Implemented. Specified on connection to MQTT Broker per each subscriber / publisher.
Keep-Alive / Heartbeat	✓	✗	✗	Not implemented/unused.
Client Takeover	✓	✗	✗	Not implemented/unused.
Message Queuing	✓	✗	✓	Taken care of by MQTT Broker.
Automatic Reconnect	✗	✓	✓	Implemented, specified in onDisconnect/onOffline per each subscriber/publisher. onDisconnect will be called once a subscriber or publisher loses connection to the broker and reconnection will be called for a period of time.
Offline Buffering	✗	✓	✓	Implemented. Messages that are received to the client side will be forwarded to a queue that aims both to handle spikes of incoming data (while still executing client functions at the same rate regardless of incoming pace) and also
Throttling	✗	✓	✗	Not implemented/unused.
Input validation	✗	✓	✓	Implemented through filter component.
Handling errors / exceptions	✗	✓	✓	Implemented to some degree. Not specific to respective errors but errors are handled in the same way regardless of the type of error.

Regarding QoS Levels for publishers and subscribers

By default the MQTT has QoS 0 which we decided to maintain between our request generator and the pipe. The reason for this being that there will be a lot of requests from our generator, so if some packages are lost, we do not consider this being a problem (as the generated data is the representation of an actual request for transportation). Within the rest of the system we decided to change our QoS to 1 which we argue is a good choice due to the fact that we have spent processing power and time on validating the data, generating a topic and making sure its data we want to show on our map. If we would not have QoS 1 we would not be sure that the time and effort we spent on that package is not a waste. We were thinking about using the QoS-level 2 but we felt that this would be excessive since the data we are working with is not so important that we need to ensure it arrives once and only once. A duplicated request would still be representative of actual data and would probably not skew the summary of data especially much. For the future, we would have logic in place that would ensure that no package gets duplicated or sent twice upon receiving. We have however not focused on implementing this yet.

Regarding fault-tolerance mechanism(s)

While researching circuit-breaker options we only found libraries that require NODE.js which would not work with our setup for the visualiser due to us using plain JavaScript and not having a server side running Node. Because of the aforementioned we decided to write our own implementation for a circuit breaker since the second alternative would be rethinking our entire frontend implementation and we estimated that this would take longer time than implementing a simple but working version of a circuit-breaker.

We realised however that a circuit-breaker would not be sufficient for fault-tolerance for our system since the circuit-breaker would not help mitigate request peaks that might occur. This forced us to come up with a safety measure that would take care of these problems and we ended up creating our queue buffer which basically saves all requests in a queue and then feeds the map with data dequeued from the queue and in this way peak overload is mitigated and the client/visualiser is protected by being fed new data at the same pace regardless of the pace of messages being received (since the map cannot render new data at high speed).

Potential impact on architecture

Latency will be increased by our architecture due to having several components through which each message will travel and this will also cause the performance to be reduced. Using QoS-level 1 throughout almost all components will also increase the latency but we feel that the trade off is worth it, as written about in more detail under the section regarding QoS.

When developing our architecture we had scalability in mind since we are developing a distributed system where scalability is an important feature. We believe that we have separated our components in a good way so that they are clearly different but not overdoing it by being as

strict as possible. By having these separations we can say that our system handles scalability well but of course the pressure on the broker will increase as the number of replicas of each component increase and if we were to deploy this system in the real world we would have to mitigate this fact in a proper way, controlling for how the broker is used.

Throughput will obviously be higher with several components using QoS-level 1 in relation to QoS-level 0. However, throughput is controlled for by the validator component which will not forward packets/messages different to the valid format, thereby controlling for packets/messages that might differ in format and bit-size.

By our implementation with a queue buffer we have managed to increase the availability of data since we are storing upon receiving packages/messages and rendering data on the map at a specific pace that the map/visualiser are able to handle. Thus the pace of receiving packages does not affect the map rendering.