Managing Latency and Excess Data Dissemination in Fog-Based Publish/Subscribe Systems

Jonathan Hasenburg*, Florian Stanek*, Florian Tschorsch†, David Bermbach*
TU Berlin & Einstein Center Digital Future
*Mobile Cloud Computing: {jh, fst, db}@mcc.tu-berlin.de
†Distributed Security Infrastructures: florian.tschorsch@tu-berlin.de

***Abstract*—Today, communication between IoT devices heavily relies on fog-based publish/subscribe (pub/sub) systems. Communicating via the cloud, however, results in a latency that is too high for many IoT applications. In this paper, we describe the design of a fog-based pub/sub system that integrates edge resources to improve communication latency between end devices in proximity. To this end, geo-distributed broker instances organize themselves in dynamically sized broadcast groups. Each broadcast group comprises a set of well connected edge brokers that communicate directly using flooding. This minimizes communication latency and copes well with frequently updated subscriptions and mobile end devices, which is required by many IoT applications. Messages between broadcast groups are routed via a massively scalable fog broker that pre-filters messages to reduce excess data dissemination. Our approach, therefore, manages the tradeoff between latency and excess data.**

***Index Terms*—Publish/Subscribe, Fog Computing, Latency, Excess Data, Tradeoff**

I. Introduction

To realize asynchronous, loosely coupled communication between a large number of IoT devices, widely used systems such as AWS IoT[1] or Google Cloud IoT[2] build upon topic-based publish/subscribe (pub/sub). Pub/sub systems are generally beneficial as they are payload agnostic and facilitate communication between end devices that do not even have to know each other [1]. These systems, however, usually run in cloud environments, which leads to higher latency than many IoT applications can tolerate.

The emerging fog computing paradigm promises low latency while also keeping the scalability aspects of the cloud [2]: For low latency, application components are deployed close to or at the edge; for high scalability, they are deployed in or near the cloud. To make topic-based pub/sub systems fog-ready, we propose to geo-distribute broker instances across edge resources in a way that individual messages of local end devices can be delivered with low latency. Especially in global deployments, the number of edge brokers can quickly increase to thousands of individual instances. All these brokers have to be interconnected—this is needed to deliver messages to subscribed end devices independent of the broker at which it has created a subscription.

There are two general strategies for message distribution: (i) *global flooding*, i.e., broadcasting messages directly to all brokers, and (ii) *cloud relay*, i.e., a cloud instance forwarding messages from and to edge brokers. With global flooding, communication latency is optimal and mobile end devices can create their subscription at any broker to immediately receive matching messages. With cloud relay, edge brokers forward their messages and subscription to a central cloud broker, which uses the subscription information to decide where to forward the messages. While minimizing excess data dissemination, this increases latency.

From the two strategies, it becomes clear that there is a tradeoff between latency and excess data dissemination when deploying pub/sub systems in the fog. Existing solutions, e.g., [3]–[5], do either not address this tradeoff or require a holistic view on all messages and subscriptions. Furthermore, these solutions are incompatible with existing pub/sub systems, which render scenarios such as running AWS IoT alongside an adapted broker at the edge infeasible.

In this paper, we propose a decentralized solution that relies on dynamic broadcast groups. Within a broadcast group, brokers exchange messages based on flooding, whereas a cloud broker relays inter-group messages. With the broadcast group size, we have a tunable parameter for the management of the tradeoff between excess data and latency. Consequently, we make the following main contributions:

- We present the design of a fog-ready topic-based pub/sub system that relies on *broadcast groups*, a novel communication strategy (Section II).
- We demonstrate that our approach integrates out-of-the box with any MQTT broker available today and present our proof-of-concept implementation (Section III).
- We extensively evaluate our approach with simulation and experiments executed on an emulated fog computing testbed (Section IV).

The results confirm that our approach achieves low communication latency *and* reduces the excess data, effectively managing the tradeoff.

II. Broadcast Groups

The main idea behind our broadcast groups approach is to split the set of edge brokers into well connected groups which use flooding for intra-group communication and a cloud relay for inter-group communication. This minimizes communication latency at the edge, i.e., where a low communication latency is often required by end devices in close proximity [6].

[1]https://aws.amazon.com/iot/
[2]https://cloud.google.com/solutions/iot/

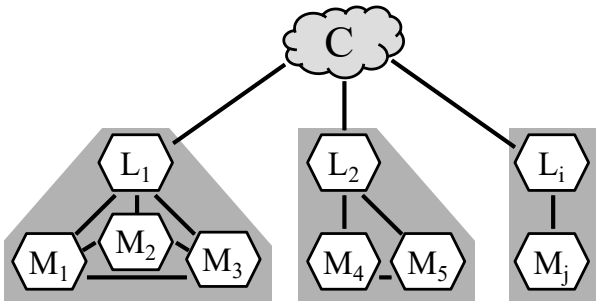

Figure 1. Broadcast group overlay network topology.

Examples for this can be found in Internet-of-Vehicles [7], [8], Smart City [9], or Mobile Health [10] scenarios. As a side effect, flooding also handles frequently updated subscriptions of mobile end devices particularly well, as messages are sent preemptively to all brokers at which an end device could create its subscriptions. For global communication, each broadcast group elects a leader that communicates on behalf of the group with the cloud broker (and thus also other group leaders).

A. Topology

Our approach builds upon a broker overlay network arranged in three tiers as shown in Figure 1. Each topology comprises a cloud broker C, a set of leaders $L_1, \ldots, L_i$, and a set of members $M_1, \ldots, M_j$. This topology is the result of the group formation process (Section II-C) that assigns individual edge brokers[3] to a single leader. A leader and its members form a broadcast group (gray area in the figure); each broadcast group must, at least, consist of a single leader.

Member nodes act as normal pub/sub broker instances, i.e., they allow end devices to connect, subscribe, unsubscribe, and publish messages. Besides matching messages locally, they forward all subscribe and unsubscribe messages to the group leader. While the group leader can also act as a normal pub/sub broker, it additionally creates subscriptions on behalf of its end devices at the cloud broker. This has the advantage of relieving (computationally) weaker members of the group from managing a connection to the cloud broker, while also preventing multiple subscriptions to the same topic by individual group members as these are collapsed into a single one by the leader.

B. Message Dissemination

The brokers in a broadcast group form a broadcast domain for publish and subscribe messages. When a broker receives a subscribe message from an end device, it forwards the message to its leader. The leader then creates a subscription on behalf of the entire group at the cloud broker.

When a broker receives a publish message from an end device, it broadcasts the message to the group. In addition, the leaders forwards every received message (either from end devices or its members) to the cloud broker. The cloud broker distributes the message to all other group leaders that have created a matching subscription. If any leader receives such a message, it broadcasts the message to the group.

[3]Note, that any broker may in fact be a clustered pub/sub system.

Algorithm 1 Group merge, notification, and join

```
function doGroupMerge(otherLeader)
    newLeader ← negotiateLeader(otherLeader)
    if newLeader = otherLeader then
        notifyMembersAboutMerge(newLeader)
        joinLeader(newLeader)
    end if
end function

function joinLeader(newLeader)
    connectToLeader(newLeader)
    brokers = getBrokersInBroadcastGroup(newLeader)
end function

function onMergeNotification(newLeader)
    if latencyBelowThreshold(newLeader) then
        joinLeader(newLeader)
    else
        createNewBroadcastGroup()
    end if
end function
```

C. Group Formation Process and Leader Election

Initially, each broker joining the system takes the role of a leader and forms its own broadcast group. Leaders subscribe to a dedicated topic at the cloud broker. They regularly publish their IP address to this topic to announce their presence to other leaders. In case of very large deployments, leader announcements can be partitioned by using diverse topics. For example, brokers in Europe could subscribe and publish to the topic *leaders/europe* while brokers in North America could use the topic *leaders/northamerica*.

Leaders continuously monitor the latency to other known leaders. When a leader observes a latency below a threshold, it initiates a group merge (cf. Algorithm 1). Part of the group merge process is the leader election, which can build on properties such as the compute power or the current bandwidth to the cloud broker. In many cases, however, it is sufficient to just assign a value to each broker on startup that indicates available resources; we call this value *Leadership Capability Measure (LCM)*. During the negotiation, the leaders would then exchange their LCMs and the leader with a higher LCM becomes the leader of the joint group.

Members continuously measure the latency to their leader. If a member observes a latency above a threshold, it leaves the group. By leaving the group, the broker automatically becomes the leader of a new broadcast group that only comprises a single broker. Members also start their own broadcast group when they receive a merge notification but the latency to the new leader is above the latency threshold. One solution to avoid oscillating membership is to use two latency thresholds:

a lower one for joining and a higher one for leaving. In addition, whether latency exceeds or falls below a threshold should also be based on a moving average of observed latency values.

The group formation process terminates, when the following two conditions are true: (i) For every leader, the latency to all other leaders is above the latency threshold. (ii) For every member, the latency to its leader is below the latency threshold. If either condition is violated, e.g., because of infrastructure changes or broker failure, group formation continues until both conditions are met again.

D. Summary

In essence, the group formation process transitions from a pure cloud relay solution (every broker is its own leader), to an intra-group flooding solution. The size of resulting broadcast groups (and thus the tradeoff between excess data and latency) can be controlled through the latency threshold. In the presence of failures or changing network conditions at the edge, brokers can always fall back to cloud relay. This ensures continuous, global message delivery as long as brokers can connect to the cloud broker, i.e., overall availability is at least as good as cloud relay, but possibly higher through group internal message distribution near the edge. Even if a leader node fails, its member nodes will start their own, individual broadcast groups before running through the group formation process again. As this process does not require central orchestration, network partitions also do not prevent group formation in general; only the broadcast groups that cannot communicate with the Cloud anymore are affected. Furthermore, local traffic between end devices that are connected to the same broker is always delivered, even if there is temporarily no connection to any other broker available.

III. Proof-of-Concept Prototype

As a proof-of-concept, we extended the implementation of the popular MQTT broker Moquette[4] with the features necessary for our proposed broadcast group approach. Our implementation is available on GitHub[5]. While it is necessary to add some functionality for the formation of broadcast groups (see below), these *broadcast group brokers* can interoperate with unmodified vanilla brokers for two reasons: First, the cloud broker that ensures global communication can be any kind of MQTT broker (e.g., in the subsequent evaluation, we use a vanilla Mosquitto broker[6] for that purpose). Second, leaders act on behalf of their broadcast group's members. Thus, to other brokers, broadcast groups appear to comprise a single broker only and the modified brokers simply assume that vanilla brokers are unwilling to join a broadcast group for latency reasons.

For our proof-of-concept prototype, we added the following functionality to Moquette (about 1,700 lines of Java code):

- Leader announcements and continuous latency measurements to other leaders.
- Group merges in compliance with our described group formation process.
- Latency aware members, i.e., they leave a broadcast group when they measure a higher latency to their leader than allowed by the latency threshold.
- Broadcast group message dissemination.
- Communication with a cloud broker (can be any MQTT compliant broker); for this, we use Eclipse Paho[7].

IV. Evaluation

Our evaluation comprises a simulation analysis of the group formation (Section IV-A) and experiments using our proof-of-concept prototype in an Internet-of-Vehicles (IoV) scenario (Section IV-B). The simulation analysis builds upon a global deployment so that we can study effects in our target environment. The experiments, on the other hand, only comprise a limited number of brokers as this is sufficient to validate that our prototype can manage the tradeoff between excess data and latency.

A. Simulation: Overhead of the Group Formation Process

Many parameters influence the group formation process, e.g., the latency between broker machines, the latency threshold, heterogeneity of broker resources, or the number of broker instances. To better understand these effects, we implemented event-discrete simulation of the group formation process to evaluate it in large geo-distributed deployments. In the following, we will use simulation results to discuss:

1) how the latency threshold influences the total number of broadcast groups,
2) and the group formation overhead, i.e., the number of messages needed to complete the process.

For this discussion we executed 160 simulation runs of the group formation process with different broker numbers and latency thresholds. Broker locations and resulting network latency are based on the worldcities data set from simplemaps[8]. The intuition behind this is that each city has access to its own pub/sub broker, and that all these edge brokers are interconnected for global communication. More details can be found in the GitHub repository[9] of our simulation tool.

As expected, Figure 2 shows that increasing the latency threshold decreases the total number of broadcast groups quadratically (note the logarithmic scales in the figure). In addition, the total number of brokers does not influence this result for higher broker numbers. This confirms the effectiveness of our threshold-based approach as fewer broadcast groups mean that, on average, each group comprises more members.

Figure 3 shows that the number of leader and member join operations needed to reach a stable state scales linearly with the number of brokers—also for different latency thresholds.

[4]https://moquette-io.github.io/moquette/
[5]https://github.com/MoeweX/moquette
[6]https://mosquitto.org/
[7]https://github.com/eclipse/paho.mqtt.java
[8]https://simplemaps.com/data/world-cities
[9]https://github.com/MoeweX/broadcast-group-simulation

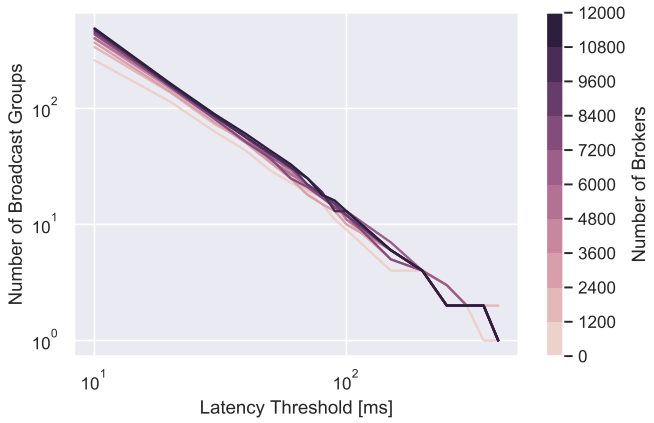

Figure 2. Latency thresholds control the amount of broadcast groups (logarithmic scale).

Figure 3. The number of operations scales linearly with the number of brokers.

The minimum number of messages[10] required to negotiate a leader join is three (join request with LCM, join reply with proposal who becomes new leader, actual leader join message). The minimum number of messages for a member join is two (member notification message, actual member join message). In our simulation, on average 1.04 member notification messages have been sent for every member join message, as members only join if the connection to the new leader has a latency below the threshold. As the number of messages increases linearly with the number of operations, the required message overhead of the group formation is $\mathcal{O}(N)$.

Infrastructure changes such as the addition or removal of a broker also trigger the group formation process. However, it is considerably shorter than the initial one and depends on the extent of change. In the best case, a new broker simply joins another leader while all other brokers are not affected.

Note, that for the group formation process, brokers rely on latency measurements to other brokers. The number of required measurements depends on the number of brokers and leaders: If N is the total number of brokers, the process involves ($\mathcal{O}(N-L+L^2)$) measurements; $N-L$ measurements from members to their leaders, and L^2 measurements between leaders. If the latency threshold is set to 0, every broker would be a leader (similar to cloud relay) so $N = L$. This leads to $\mathcal{O}(N - N + N^2) = \mathcal{O}(N^2)$ measurements in the worst case. If the latency threshold is set to a very large value, all brokers end up in the same broadcast group (similar to global flooding) so $L = 1$. This leads to $\mathcal{O}(N-1+1^2) = \mathcal{O}(N)$ measurements in the best case. When using a latency threshold that results in a middleground solution, e.g, $L = \sqrt{N}$, the number of measurements is still $\mathcal{O}(N - L + (\sqrt{N})^2) = \mathcal{O}(N)$. The number of needed latency measurements can be additionally reduced by partitioning leader announcements as explained in Section II-C.

B. Experiment: Effectiveness of Broadcast Groups

To compare latency and excess data dissemination between communication strategies, we ran experiments based on an IoV scenario. Unfortunately, existing solutions are not applicable to our envisaged environment. Here, pub/sub systems must have an acceptable overhead that allows them to scale to deployments comprising thousands of edge brokers. In addition, they must handle end devices that frequently update subscriptions, and interoperate with other broker implementations. We therefore refrain from evaluating these related solutions experimentally (cf. Related Work for details) and compare the different communication strategies, instead. To this end, we used an emulated infrastructure with multiple broker instances as fog computing testbed [11].

Scenario: Our scenario is a simplified IoV use case with three types of clients (end devices): cars, monitoring equipment, and traffic authorities. For improved driving safety, cars exchange telemetry data with other cars so that they know when cars brake or change lanes. The monitoring equipment, e.g., a camera, collects traffic information that it sends to the traffic authority for processing. The traffic authority could use the collected data to inform cars about events such as traffic jams or accidents; we refrained from doing so to keep the use case simple.

Evaluation Setup: Our evaluation setup can be seen in Figure 4. We deployed three of our proof-of-concept brokers at the edge, to which a total of four end devices (three cars and one piece of monitoring equipment) connected. Because our brokers still support the MQTT protocol, we can use the standard Mosquitto command line clients for communication. As cloud broker, we deployed a vanilla Mosquitto MQTT broker; every other system that supports the MQTT protocol could be used as well.

Infrastructure and Deployment: As we do not have access to a fog computing infrastructure with the characteristics shown in Figure 4 available, we used MockFog [11] to emulate such an infrastructure. Based on an infrastructure definition, MockFog deploys one virtual machine for each component, configures networking delays, and deploys the brokers and

[10]More messages might be required if messages are acknowledged or a different negotiation protocol is used.

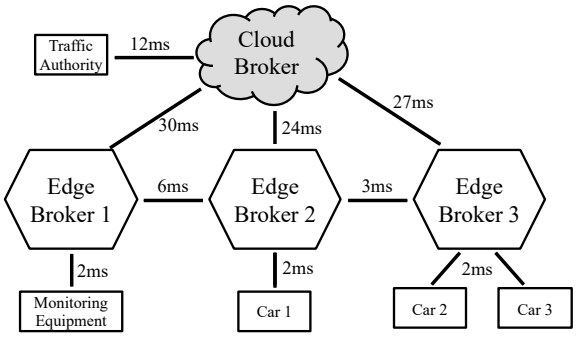

Figure 4. Evaluation setup.

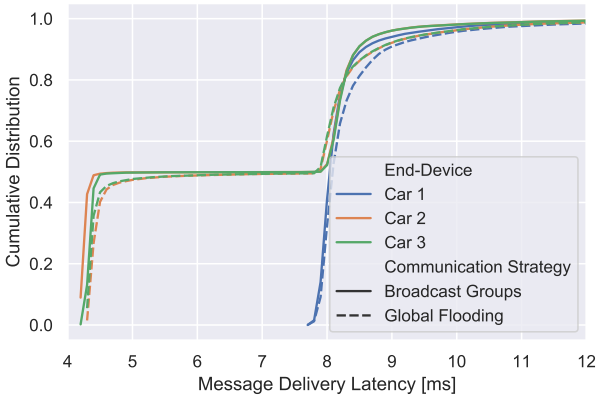

Figure 5. Car Telemetry MDL for global flooding and for broadcast groups.

clients. For our evaluation, MockFog used Amazon EC2[11] with `t3.small` instances for brokers and `t3.nano` instances for end devices.

Experiment Execution: For each of the three communication strategies, global flooding, broadcast groups, and cloud relay, we ran the same 15-minute workload: For the exchange of telemetry data, each car publishes 20 bytes of data to a unique topic 20 times per second, e.g., Car 1 publishes to the topic */car-telemetry/realtime/1*. At the same time, all cars subscribe to the wildcard topic */car-telemetry/realtime/+* which matches the individual topics. The traffic authority collects data from monitoring equipment by creating a subscription to the topic */traffic-control/monitoring/#*. The monitoring equipment publishes its data (1000 bytes, once per second) to the (unique) topic */traffic-control/monitoring/1*.

All experiments were run with our proof-of-concept prototype. For the global flooding strategy, we set up MQTT bridging [12] between all brokers, which is supported by Moquette out of the box. As a consequence, each message received by one broker is forwarded to every other broker. For the group formation, we used a latency threshold of 5 ms. This leads to two broadcast groups, one that comprises only Edge Broker 1, and one that comprises Edge Broker 2 and Edge Broker 3. For cloud relay, we set the latency threshold to 0 which leads, in our setup, to three broadcast groups that each comprise a single edge broker.

In the following, we first discuss the message delivery latency measured for each strategy (Section IV-C), before discussing the amount of excess data that is produced by each strategy (Section IV-D). Note that due to space constraints, we did not include experiment results on all aspects of the group formation process, e.g., members leaving their broadcast group when the measured latency exceeds the latency threshold. However, from the experiments, it already becomes clear that the group formation works as designed.

C. Message Delivery Latency

The message delivery latency (MDL) for each individual message is defined as $\text{MDL} = t_{\text{received}} - t_{\text{send}}$, with t denoting the send and receive timestamp measured at each client. Since Amazon EC2 machines have highly synchronized clocks[12], clock drift is negligible and can, thus, be disregarded for our experiments.

Considering the topology, the global flooding and broadcast group MDL should be similar. For both strategies, we expect 50% of the messages received by Car 2 and Car 3 to have an MDL of about 4 ms, as the corresponding messages can be matched at Edge Broker 3 directly and the one-way latency between Edge Broker 3 and each car is 2 ms. The remaining 50% of the messages received by Car 2 and Car 3, as well as all messages received by Car 1, should have an MDL of about 7 ms, as the messages additionally need to be sent via the link between Edge Broker 2 and Edge Broker 3. Figure 5 shows the experiment results which confirm our expectation. In particular, there is a clearly visible step when the cumulative distribution reaches 50% for Car 1 and Car 2, which confirms the general performance of broadcast groups[13].

For the cloud relay strategy, we expect 50% of the messages received by Car 2 and Car 3 to have a similar MDL as in the global flooding or broadcast group experiment. The remaining 50% of their received messages, as well as all messages received by Car 1, should be routed via the cloud, which leads to an MDL of 55 ms or higher. Figure 6 shows that the used implementation achieves this latency for some messages. However, this experiment also reveals that our setup with the Eclipse Paho MQTT client library, which is used by the Moquette edge brokers to create subscriptions at the Mosquitto cloud broker, negatively influences MDL (visible by the long-tail)[14].

All messages received by the traffic authority, i.e., all messages published by the monitoring equipment, should have

[11]https://aws.amazon.com/ec2/

[12]https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-the-amazon-time-sync-service/

[13]The latency values are not exactly 4 ms (and 7 ms), as processing the messages on each edge broker also requires some time.

[14]This is not visible in any other experiment, as only here a subscription created from one broker (Edge Broker 2 and 3) at another broker (Cloud Broker) matches incoming messages. Intra-group communication is done via broadcasting which does not depend on subscriptions.

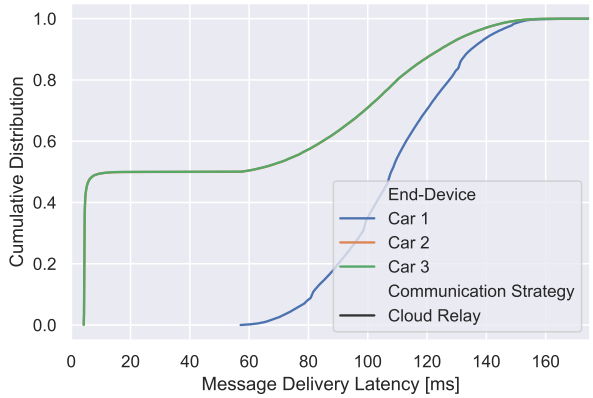

Figure 6. Car Telemetry MDL for cloud relay (note, lines for Car 1 and Car 2 overlap).

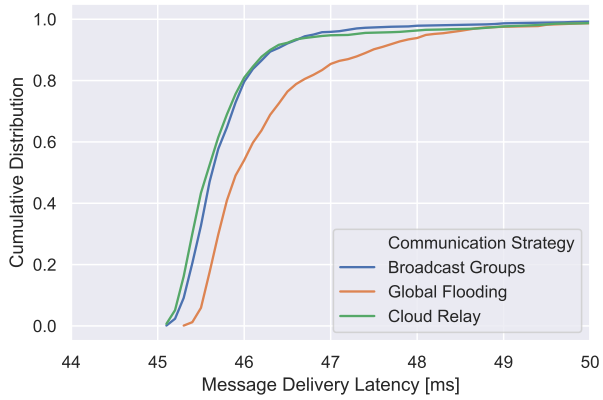

Figure 7. Monitoring data MDL.

a latency of 44 ms. Figure 7 shows that, considering the mentioned overheads, the results match our expectation. Note that our forwarding implementation seems to have a better performance than Moquette's bridging which is used by the global flooding strategy.

In conclusion, broadcast groups can achieve a communication latency close to the one of global flooding. Depending on the workload, this can be significantly better than the communication latency achieved by cloud relay.

D. Excess Data

In the following, we evaluate the impact of each communication strategy on excess data dissemination. For that, we logged each message processed by every broker and determined the amount of correct and redundant messages. Correct messages are messages processed by brokers that either have a matching subscriber or to which the publishing client is connected. Redundant messages are all other messages processed by brokers; these messages have to be discarded and therefore count as excess data.

Table I
EXCESS DATA FOR EACH COMMUNICATION STRATEGY: CAR TELEMETRY

	Global Flooding	Cloud Relay	Broadcast Groups
Correct Msgs.	107948	107944	107944
Redundant Msgs.	107948	53972	53972
Excess Data	50%	33%	33%

Table II
EXCESS DATA FOR EACH COMMUNICATION STRATEGY: MONITORING DATA

	Global Flooding	Cloud Relay	Broadcast Groups
Correct Msgs.	1798	1798	1798
Redundant Msgs.	1798	0	0
Excess Data	50%	0%	0%

Table I and Table II show the amount of correct and redundant messages, as well as the share of excess data, for each communication strategy and both message flows. The car telemetry messages have to be processed by Edge Broker 2 and Edge Broker 3 only; thus, all messages processed by Edge Broker 1 and Cloud Broker are excess data. The monitoring messages have to be processed by Edge Broker 1 and Cloud Broker only; thus, all messages processed by Edge Broker 2 and Edge Broker 3 are excess data. While we used the same workload for all three communication strategies, the respective message numbers are not exactly identical due to minimal runtime variations of our publishing clients.

In conclusion, global flooding produces the largest amount of excess data, while broadcast groups achieves a similar efficiency as cloud relay. Overall we can see that broadcast groups achieves results in terms of communication latency and excess data that can be considered the best of both worlds, effectively balancing this tradeoff.

V. RELATED WORK

In a fog environment, many approaches for distributed pub/sub system are a poor fit. For example, [13], [14] are tailored for the cloud and thus assume LAN connectivity between machines. Therefore, they cannot cope with geo-distribution. Furthermore, with approaches building on distributed hash tables (e.g., [15]–[17]) or rendezvous nodes (e.g., [18]), messages might be routed across any node even though the node might not have access to sufficient compute or networking resources. Particularly in the fog, this is not acceptable due to the heterogeneity of machines, as well as partly unstable and relatively slow network connections. There is, however, a number of approaches that also aim to make pub/sub systems ready for an environment such as the fog.

Rausch et al. [3] propose a fog-enabled geo-distributed broker. In contrast to our work, they aim to provide optimal latency for all communication. To this end, they use a centralized cloud service that continuously orchestrates brokers and migrates MQTT clients. The cloud service, however, needs a comprehensive global view on inter-node latency, edge

brokers, clients, and subscriptions. Keeping this view up to date can be challenging in volatile deployments. Moreover, migration might lead to message loss which is especially problematic in scenarios with mobile end devices.

An et al. [4] propose PubSubCoord which, at first glance, looks very similar to our solution as they also group local brokers. There are, however, two key differences. First, their local groups are based on network segments rather than inter-machine latency; because we can change the latency threshold that controls group formation, we can manage the latency and excess data dissemination tradeoff. Second, coordination and message exchange between local groups relies on Zookeeper and a custom broker implementation, while our approach integrates with arbitrary vanilla MQTT brokers which makes it possible to use the best solution for any situation.

Cao and Singh [19] propose MEDYM and H-MEDYM. In MEDYM, brokers need to know about all other brokers in the network and the sum of their subscriptions, which is infeasible in large deployments. To circumvent this limitation, H-MEDYM also proposes to create broker groups. In each group a so-called matcher broker handles the communication with the matchers of other groups. Still, in H-MEDYM, each matcher needs be aware of subscriptions from all other matchers which is infeasible in scenarios with a high amount of frequently updated subscriptions, especially in global broker deployments.

Kawaguchi and Bandai [5] propose a distributed broker system that supports heterogeneous broker resources. For this, they rely on a custom topic structure that embeds geographic information, so each message is associated with a certain area. As every broker is responsible for exactly one area, it only has to process the messages associated with its area. Changing area sizes can be used to control the broker load, but the approach only works for location-dependent data.

Shun et al. [8] propose a topic-based fog computing architecture which they use for the exchange of semantically enhanced IoV data. In contrast to our approach, their fog nodes have to create subscriptions at all other fog nodes which limits scalability and potentially overloads brokers with few resources.

Banno et al. [20] propose to interconnect heterogeneous and distributed MQTT brokers through an additional middleware layer between brokers and end devices. This layer also takes care of distributing messages between brokers based on customizable routing strategies. For their paper, they only implemented flooding, but we assume that adding our broadcast group strategy is possible as well. Unfortunately, their source code is not publicly available so that we could not verify this. In contrast to all other related work, this approach is also the only one that is interoperable with existing setups.

VI. Discussion

In this section, we discuss limitations and design choices. First, the group formation process (Section II-C) optimizes locally, but therefore may not lead to a globally optimized topology. For example, two very well-connected brokers that exchange large amounts of data might end up in separate broadcast groups, and therefore have to communicate via the cloud. As future work, we plan to make brokers more aware of their environment and message flows. For example, brokers could detect that a majority of relevant messages originate at a broker from another broadcast group, and update their membership accordingly. Another exciting topic for future work is not to use a global, and fixed latency threshold, but rather to let each broadcast group define their own threshold based on local conditions. Then, broadcast groups that can tolerate a higher amount of excess data could increase their threshold to improve latency, while overloaded broadcast groups can decrease their threshold to reduce excess data.

Second, we chose flooding for intra-group communication because it offers the lowest communication latency and copes well with mobile subscribers; excess data remains manageable as long as broadcast groups do not become too large. However, depending on the infrastructure environment and workload, it is certainly possible to use other strategies that, while not being an option when distributing messages across "the entire fog", are feasible for machines in the same broadcast group. Note, however, that all strategies without flooding involve a *warmup phase*, i.e., a subscription to a topic formerly unknown to the broker cannot be served immediately as, for example, the broker first has to tell other brokers about the subscription. Depending on the use case, this might not be acceptable.

Third, if needed, our approach can also be used to mimic the two other strategies used in the evaluation. To flood messages to all brokers, the latency threshold can be set to a very large value so that all brokers end up in the same broadcast group. To implement the cloud relay strategy, the latency threshold can be set to zero so that every broker creates its own broadcast group. This observation also emphasizes that our approach indeed provides a way to manage the tradeoff between excess data and latency.

VII. Conclusion

In this paper, we described the latency and excess data dissemination tradeoff that needs to be taken into account by geo-distributed pub/sub systems running in fog environments. We proposed a solution that relies on dynamically sized broadcast groups to manage this tradeoff. Broadcast groups split the set of edge brokers into well connected groups which use flooding for intra-group communication and a cloud relay for inter-group communication. This minimizes communication latency and copes well with frequently updated subscriptions and mobile end devices. We evaluated our approach through simulation and experiments. For the latter, we used an emulated fog infrastructure testbed and compared results to cloud relay and global flooding. The results confirm the effectiveness of our approach and that involved overheads remain manageable—this even applies to global deployments with thousands of individual broker instances.

References

[1] K. Paridel, E. Bainomugisha, Y. Vanrompay, Y. Berbers, and W. De, "Middleware for the internet of things, design goals and challenges,"

EASST Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services, vol. 28, 2010.

[2] D. Bermbach, F. Pallas, D. G. Pérez, P. Plebani, M. Anderson, R. Kat, and S. Tai, "A research perspective on fog computing," in *2nd Workshop on IoT Systems Provisioning & Management for Context-Aware Smart Cities*, vol. 10797. Springer, 2018, pp. 198–210.

[3] T. Rausch, S. Nastic, and S. Dustdar, "EMMA: Distributed QoS-aware MQTT middleware for edge computing applications," in *2018 IEEE Int. Conf. on Cloud Engineering*. IEEE, 2018, pp. 191–197.

[4] K. An, A. Gokhale, S. Tambe, and T. Kuroda, "Wide area network-scale discovery and data dissemination in data-centric publish/subscribe systems," in *Proc. of the Posters and Demos Session of the 16th Int. Middleware Conf.* ACM Press, 2015, pp. 1–2.

[5] R. Kawaguchi and M. Bandai, "A distributed MQTT broker system for location-based IoT applications," in *2019 IEEE Int. Conf. on Consumer Electronics (ICCE)*. IEEE, 2019, pp. 1–4.

[6] P. Bellavista, A. Corradi, and A. Reale, "Quality of service in wide scale publish-subscribe systems," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1591–1616, 2014.

[7] D. Frey and G.-C. Roman, "Context-aware publish subscribe in mobile ad hoc networks," in *Coordination Models and Languages*, A. L. Murphy and J. Vitek, Eds. Springer, 2007, vol. 4467, pp. 37–55.

[8] Sejin Shun, Sangjin Shin, Seungmin Seo, Sungkwang Eom, Jooik Jung, and Kyong-Ho Lee, "A pub/sub-based fog computing architecture for internet-of-vehicles," *2016 IEEE Int. Conf. on Cloud Computing Technology and Science*, pp. 90–93, 2016.

[9] L. Sanchez, L. Muñoz, J. A. Galache, P. Sotres, J. R. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco, E. Theodoridis, and D. Pfisterer, "SmartSantander: IoT experimentation over a smart city testbed," *Computer Networks*, vol. 61, pp. 217–238, 2014.

[10] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.

[11] J. Hasenburg, M. Grambow, E. Grunewald, S. Huk, and D. Bermbach, "MockFog: Emulating fog computing infrastructure in the cloud," in *First IEEE Int. Conf. on Fog Computing*. IEEE, 2019, p. 9.

[12] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, "MQTT version 5.0." OASIS Standard. https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html, accessed 06/09/2019, 2019.

[13] R. Barazzutti, P. Felber, C. Fetzer, E. Onica, J.-F. Pineau, M. Pasin, E. Rivière, and S. Weigert, "StreamHub: A massively parallel architecture for high-performance content-based publish/subscribe," in *Proc. of the 7th ACM international Conf. on Distributed event-based systems*. ACM Press, 2013, pp. 63–74.

[14] J. Gascon-Samson, J. Kienzle, and B. Kemme, "MultiPub: Latency and cost-aware global-scale cloud publish/subscribe," in *2017 IEEE 37th Int. Conf. on Distributed Computing Systems*. IEEE, 2017, pp. 2075–2082.

[15] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure," in *Networked Group Communication*, J. Crowcroft and M. Hofmann, Eds. Springer Berlin Heidelberg, 2001, vol. 2233, pp. 30–43.

[16] Y. Zhao, K. Kim, and N. Venkatasubramanian, "DYNATOPS: A dynamic topic-based publish/subscribe architecture," in *Proc. of the 7th ACM Int. Conf. on Distributed Event-based Systems*. ACM, 2013, pp. 75 – 86.

[17] R. Banno, S. Takeuchi, M. Takemoto, T. Kawano, T. Kambayashi, and M. Matsuo, "Designing overlay networks for handling exhaust data in a distributed topic-based pub/sub architecture," *Journal of Information Processing*, vol. 23, no. 2, pp. 105–116, 2015.

[18] P. Pietzuch and J. Bacon, "Hermes: a distributed event-based middleware architecture," in *Proc. 22nd Int. Conf. on Distributed Computing Systems Workshops*. IEEE, 2002, pp. 611–618.

[19] F. Cao and J. P. Singh, "MEDYM: Match-early with dynamic multicast for content-based publish-subscribe networks," in *Middleware 2005*, G. Alonso, Ed. Springer Berlin Heidelberg, 2005, vol. 3790, pp. 292–313.

[20] R. Banno, J. Sun, M. Fujita, S. Takeuchi, and K. Shudo, "Dissemination of edge-heavy data on heterogeneous MQTT brokers," in *2017 IEEE 6th Int. Conf. on Cloud Networking*. IEEE, 2017, pp. 1–7.