An Implementation Experience with SDN-enabled IoT Data Exchange Middleware

Luca Scalzotto, Kyle E Benson, Georgios Bouloukakis, Paolo Bellavista, Valérie Issarny, Sharad Mehrotra, Nalini Venkatasubramanian

An Implementation Experience with SDN-enabled IoT Data Exchange Middleware

Poster Abstract

Luca Scalzotto[1,2], Kyle E. Benson[2], Georgios Bouloukakis[2,3], Paolo Bellavista[1], Valérie Issarny[3], Sharad Mehrotra[2], Nalini Venkatasubramanian[2],
luca.scalzotto@studio.unibo.it,{kebenson,gboulouk}@ics.uci.edu,paolo.bellavista@unibo.it,valerie.issarny@inria.fr
sharad@ics.uci.edu,nalini@uci.edu
[1]DISI, University of Bologna, Italy
[2]Donald Bren School of Information and Computer Sciences, University of California, Irvine, USA
[3]MiMove Team, Inria Paris, France

ABSTRACT

This poster presents our prototype implementation of FireDeX [1], a cross-layer middleware that supports timely delivery of mission-critical messages (i.e. events) over an IoT data exchange service. Emergency scenarios may challenge/congest the network infrastructure. FireDeX addresses these situations by prioritizing event delivery and by dropping some low priority events.

1 FIREDEX PROTOTYPE IMPLEMENTATION

Based on the promising results from our experimental evaluation of the FireDeX middleware in [1], we implemented a prototype (available at https://github.com/boulouk/firedex) to study its real-world performance. Consider the implementation according to the three layers in Fig. 1. We adopt a publish/subscribe interaction paradigm [3] in which publishers represent IoT sources and subscribers represent interested devices, services, or persons (e.g. fire fighters or building occupants). The FireDeX Coordinator Service (FCS) extends the data exchange broker by managing the network infrastructure through SDN [8]. According to network constraints and subscribers' information requirements, it configures the network to enforce prioritization and packet (i.e. event) drop policies.

1.1 Application layer

The *FireDeX publishers* connect to the data exchange broker through a MQTT connection, leveraging the MQTT Paho library [14]. Currently, the data generated by the publishers is simulated via Poisson and Deterministic distributions. Future work will consider replacing it with data coming from real IoT deployments.

The *FireDeX subscribers* connect to the data exchange broker to receive relevant events. Since different events have varying importance for different subscribers, they register utility functions along with their topic subscriptions. These functions represent a quantified measure of value for that subscription. Each subscriber establishes multiple MQTT-SN connections to the broker with a client library [2]. Multiple connections allow the network layer to distinguish different event types (i.e. more/less relevant) as described in §1.3. Hence, it can apply different priorities and event drop policies to them as instructed by the FCS. For the subscribers we use MQTT-SN instead of MQTT because it relies on UDP rather than TCP. The latter's re-transmission mechanism interferes with our preemptive packet dropping approach that tolerates some loss of sensor data under constrained bandwidth. Note that UDP does not support fragmentation and reassembly of messages. Therefore, we assume that messages are never fragmented and limit their size at the application layer to 256 bytes (before packet headers) due to limitations in the MQTT-SN library. The subscribers first coordinate with the FCS to determine the connection to use for each subscription. Then they open the connections specified by the FCS to the broker and subscribe to the topics through these.

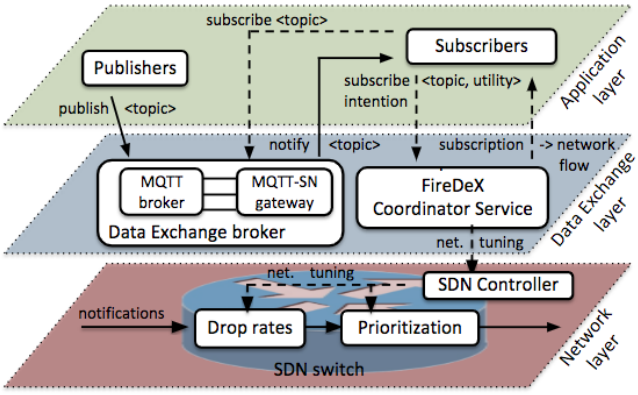

Figure 1: The FireDeX cross-layer middleware.

1.2 Data exchange layer

Data exchange broker. The data exchange layer supports the publish/subscribe paradigm for event delivery. An unmodified MQTT [11] broker facilitates this exchange between publishers and subscribers. While FireDeX supports any MQTT broker implementation, we deployed the open-source Moquette [10] broker. We also run an MQTT-SN gateway [5, 7] co-located with the MQTT broker. It translates from MQTT over TCP (publishers' protocol) to MQTT-SN over UDP (subscribers' protocol).

Middleware '18 Posters and Demos, December 10–14, 2018, Rennes, France

ACM ISBN 978-1-4503-6109-5/18/12.
https://doi.org/10.1145/3284014.3284025

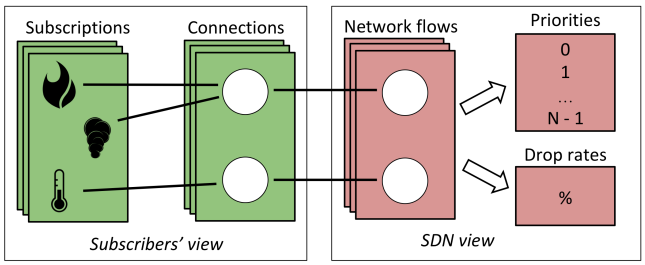

Figure 2: FireDeX assigns priorities and drop rates to subscriptions that are enforced in the network layer.

FireDeX Coordinator Service. The FCS is the "brain" of the FireDeX middleware. It manages user subscriptions by assigning priorities and drop rates as shown in Fig. 2. It runs the algorithms described in §4 of [1] to determine these policies according to users' information requirements (i.e. subscriptions' utility functions) and network constraints (i.e. limited bandwidth). The FCS physically resides either in the local network (i.e. building on fire) or a remote location. We implemented the FCS as a REST server using the Python library Flask [4]. The subscribers indicate their topics of interest and corresponding utility functions to the FCS through a HTTP request. The FCS computes priority and drop rates for each subscriber's network flow, responds with the mapping of subscriptions to connections (i.e. network flows), and then configures the network layer to enforce these policies.

1.3 Network Layer

This layer enforces event prioritization and drop rates configured through the SDN protocol OpenFlow [9]. The SDN controller configures the *Open vSwitch* (OVS) [15] software switches with these policies at the direction of the FCS. We implemented two SDN applications through the Ryu [16] SDN controller to facilitate this.

The *Topology Application* monitors network traffic to create an internal graph representation (using the NetworkX library [6]) of the network topology.

The *FireDeX Flow Application* populates the switches' flow and group tables. These tables contain OpenFlow rules that enforce the priority and drop rate policies specified by the FCS. To identify a subscriber's traffic on the network the FireDeX Flow Application matches the packet's header with the network flow information received by the FCS. Network flow information includes the subscriber's IP address and the connection's (i.e. network flow's) transport layer port number. We use *select group buckets* and bucket weights to apply the actual priority and drop rate policies. The example rules in Listing 1 match packets for the subscriber with IP address 10.0.0.1 and an MQTT-SN connection on UDP port 8888. It applies priority class 2 (i.e. queue number) and a 10% drop rate.

```
FLOW TABLE RULE: ip_address = 10.0.0.1,
    udp_port = 8888, action = (group_identifier, 1)
GROUP TABLE RULE: group_identifier = 1, buckets[
    (weight = 90, action = (queue = 2, output_port = 3)),
    (weight = 10, action = drop)]
```

Listing 1: Example rules in flow and group tables.

We configure priority queues on the switches via Linux TC [17] since OpenFlow has no unified API to support this. Furthermore, FireDeX must enforce a random per-packet selection of the buckets option (i.e. for the drop rate implementation) rather than the typical approach of hashing packet header fields. Hence, we leverage a modified OVS version [13] that implements this.

2 CONCLUSIONS AND FUTURE STUDIES

Implementation Challenges. We successfully solved the challenge of distinguishing different events at the network layer using network flows. However, other challenges remain open. For instance, the MQTT-SN control messages (e.g. subscription messages) are sent through the same UDP connection in which we apply the drop rate policies. Therefore, some of them may be dropped.
Experimental Methodology. To evaluate our approach in a real-world test-bed, we implemented an experimental framework that simulates the physical network with Mininet [12]. For each experiment, we generate random configurations, run the scenario, and calculate end-to-end response times and success rates at the end.
Towards Dynamic Studies. Our future work will leverage this prototype to study dynamic aspects of the real system: monitoring and adapting to varying bandwidth, subscriber churn, and changing priorities (i.e. utility functions).

ACKNOWLEDGEMENTS

This work was supported by: NSF award CNS 1450768, NIST Award #70NANB17H285, DARPA agreement #FA8750-16-2-0021, the Inria@SiliconValley International Lab, and the research associate team MINES. The authors of this paper also thank the authors of the Research Roadmap for Smart Fire Fighting as well as Casey Grant and Ioannis Moscholios for their contributions to FireDeX.

REFERENCES

[1] K. E. Benson, G. Bouloukakis, C. Grant, V. Issarny, S. Mehrotra, I. Moscholios, and N. Venkatasubramanian. 2018. FireDeX: a Prioritized IoT Data Exchange Middleware for Emergency Response. *ACM/IFIP/USENIX International Middleware Conference* (2018).
[2] "MQTT-SN UDP client". 2016. https://github.com/jsaak/mqtt-sn-gateway.
[3] P. T. Eugster, P. A Felber, R. Guerraoui, and A.M. Kermarrec. 2003. The many faces of publish/subscribe. *ACM computing surveys (CSUR)* 35, 2 (2003), 114–131.
[4] "Flask Web Framework". 2010. http://flask.pocoo.org/.
[5] "MQTT-SN Transparent Gateway". 2016. https://www.eclipse.org/paho/components/mqtt-sn-transparent-gateway/.
[6] A. A. Hagberg, D. A. Schult, and P. J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *SciPy*, G. Varoquaux, T. Vaught, and J. Millman (Eds.). 11–15.
[7] IBM 2013. *MQTT For Sensor Networks (MQTT-SN).* IBM.
[8] N. McKeown. 2009. Software-defined networking. *INFOCOM keynote talk* 17, 2 (2009), 30–32.
[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74. https://doi.org/10.1145/1355734.1355746
[10] "Moquette broker". 2014. https://github.com/andsel/moquette/.
[11] OASIS 2014. *MQTT Version 3.1.1.* OASIS.
[12] "Mininet: An Instant Virtual Network on your Laptop (or other PC) Mininet". 2009. http://mininet.org/.
[13] "Stochastic OVS". 2014. https://github.com/saeenali/openvswitch/wiki/Stochastic-Switching-using-Open-vSwitch-in-Mininet.
[14] "Paho Java Client". 2008. https://www.eclipse.org/paho/clients/java/.
[15] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M.ín Casado. 2015. The Design and Implementation of Open vSwitch. In *NSDI*. http://dl.acm.org/citation.cfm?id=2789770.2789779
[16] "Ryu SDN controller". 2011. https://osrg.github.io/ryu/.
[17] "Linux TC". 2001. http://lartc.org/manpages/tc.txt.