A Comparison of Data Streaming Frameworks for Anomaly Detection in Embedded Systems

Murray Dunne, Giovani Gracioli, and Sebastian Fischmeister
University of Waterloo, Canada
{mdunne,g2gracio,sfischme}@uwaterloo.ca

***Abstract*—As IoT devices are integrated into our daily lives, verification and security become of increasing concern. Using anomaly detection methods, we can identify damaged and compromised devices by examining traces of their activity. Collecting these traces with minimal overhead is a core requirement of any anomaly detection system. We evaluate four publish-subscribe broker systems on their viability for trace collection in the context of IoT devices. Our comparison considers ordering and delivery guarantees, client language support, data structure support, intended use case, and maturity. We run each system on original Raspberry Pis and collect network performance statistics, measuring their capability to collected traces in a resource-constrained embedded systems environment. We conclude with recommendations for designing an anomaly detection system for IoT devices.**

I. Introduction

Embedded systems, such as Internet-of-Things (IoT) and autonomous vehicles, are present in our daily lives. Such systems interact with the environment through several sensors and actuators, usually controlling the operation of critical processes. Moreover, these systems generate a huge volume of data, which makes the task of verifying the system specification difficult.

In this context, trace-based anomaly detection can monitor the system behavior and prevent or/and recover from failures [1]. Anomaly detection aims at detecting execution patterns that do not conform with the expected system behavior. It can be done online (during run-time) or offline (by analyzing recorded traces). Online anomaly detection usually receives a stream of data as input and incrementally adapts anomaly scores for the analyzed system, thus providing early detection of an anomaly (when compared to the offline approach).

Trace-based online anomaly detection requires a data streaming infrastructure with minimal performance overhead. Examples of general-purpose streaming frameworks are Zmq, Mqtt, ActiveMQ, Apache Spark, Redis, NATS, Apache Kafka, and RabbitMQ. Other data streaming frameworks, such as ROS, Polysync, Qnx PPS, OpenDDS, RTI Connext, and OpenSplice are designed as a for building an entire product, rather than as an ancillary monitoring application.

Figure 1 shows an overview of a general data streaming framework infrastructure for anomaly detection in embedded systems. Sources are different embedded systems, such as a smart building or an autonomous vehicle, and generate streams of data at run-time. Data from these systems is sent to a data streaming framework. Processors connect to the framework, receive and process data. The processor output is either written back to the framework (to be consumed by another processor) or indicates an anomaly. Finally, sinks receive data and can perform an action, such as storing the data into a file [2].

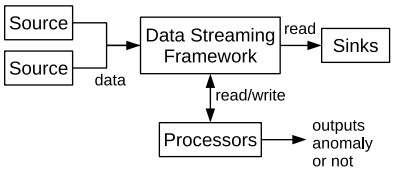


Fig. 1: Overview of general data streaming framework organization.

Several data streaming frameworks have been proposed recently. However, to the best of our knowledge, a comparison among them targeting embedded systems has not been made yet. Two metrics are important for data streaming framework performance: (i) low run-time latency (the time difference between the instant a data is generated by sources and the instant it is received by processors or sinks); and (ii) throughput, the rate of data transmission a framework can support.

In this paper, we present a comparison of existing data streaming frameworks focusing on embedded systems. We compare Redis [3], Kafka [4], NATS Streaming Server [5], and RabbitMQ [6] in terms of characteristics and performance (latency and throughput). We choose them because they have been receiving wide attention by the scientific community and represent different classes of streaming framework (*i.e.*, implemented with different languages, targeted at different platforms, and supporting different features). The performance comparison is carried out using a standard embedded system platform (Raspberry Pi). Our results indicate that Redis and RabbitMQ are suitable frameworks for embedded systems in both features and performance.

The rest of this paper is organized as follows. Section II overviews the main features of the analyzed frameworks. Section III presents the performance evaluation. Section IV discusses related works and Section V concludes the paper.

II. Overview of Data Streaming Frameworks

Redis. REmote DIctionary Server (Redis) is an open-source in-memory key-value database. It supports several data structures, including list, sets, maps, and bitmaps. Redis can be used

as a data streaming framework because it provides a publish-subscribe interface. Redis clients publish data into channels using the REdis Serialization Protocol (RESP). Instances that subscribe to channels receive data in the same order it was published. Redis also supports the integration with on-disk databases. Moreover, it has a small memory consumption; in a 64-bit system, 1 million keys (hash values), representing an object with five fields, use around 160 MB of memory [3]. This small memory consumption is adequate for embedded systems, which usually have limited memory. Redis provides a replication mechanism based on master-slave, in which slave server instances are exact copies of master servers. To reduce the network Round Trip Time (RTT) latency over transmitted messages, Redis implements Pipelining, making it possible to send multiple commands to the server without waiting for individual replies [3]. These replies are instead batched together into a single response.

Kafka. Apache Kafka is a distributed streaming platform written in Java. Kafka runs as a cluster of one or more servers. The cluster stores streams of records (key, value, and a timestamp) in topics. A topic is a category or a name in which records are published. For each topic, the Kafka cluster maintains a structured commit log, formed by partitions. Each partition within a topic is an ordered sequence of records that is continually appended to the structured commit log. Log partitions can be distributed over the cluster servers, providing fault tolerance. Clients subscribe to topics to receive/write real-time streams using a binary protocol over TCP. Kafka provides APIs for sources, processors, and sinks. Moreover, Kafka provides persistent storage by writing topic records to the disk. As Kafka is written in Java, it requires a Java virtual machine (JVM). This may not be appropriate for resource-constrained embedded systems due to JVM memory requirements [4].

NATS Streaming Server. NATS is an open-source data streaming server written in Go. NATS streaming server embeds a NATS server. Thus, the streaming server is not a server, but a client to a NATS server. Clients also communicate with the streaming server through the NATS server. All the communication uses a NATS streaming protocol based on protocol buffers. NATS streaming server provides a publish-subscribe interface based on channels. Clients send and receive messages to/from channels. Messages can be stored in memory or disk files. NATS provides a message logging mechanism to save all messages produced in a channel, allowing historical message replay by subject. Clients may specify a playback start position in the stream of messages stored for the subscribed subjects channel. NATS streaming server does not support clustering of servers. However, it supports fault tolerance by allowing the initialization of a group of servers. Within the group, only one server answers to clients requests, while the others monitor the main server. When the main server fails, another one takes control and acts as the main server [5].

RabbitMQ. RabbitMQ is an open source message brokering server maintained by Pivotal software. It implements the Advanced Message Queuing Protocol (AMQP) (ISO/IEC 19464:2014), a standardized protocol for message brokering services [6]. AMQP defines a two-stage architecture where messages are first transmitted to an exchange which forwards them to different queues depending on the exchange selected. Exchanges exist for broadcasting copies to multiple queues, addressing queues by name, or pattern matching. Messages in AMQP queues are acknowledged upon receipt by the server, and clients must acknowledge a message before it is removed from the queue. Queues must be declared before use and may be saved to disk, so their contents are not lost on restart. There is no inbuilt mechanism for replaying a message history, but RabbitMQ may be configured to store logs of message activity. RabbitMQ supports clustering where queues exist on only one node at a time, but are reachable from all nodes. A configuration option enables replication of entire queues. Nodes that store their data entirely in memory are available.

Table I summarizes the discussed features for each framework. The four framework are mature; they all provide clients in several languages, support disk storage, publish-subscribe interface, and message ordering. They differ in how they organize and process data internally, the languages in which they are written, and communication protocol.

III. Performance Evaluation

A. Experiment Description

The evaluation compares the four frameworks on publisher-to-subscriber latency and throughput. We compare messages differing in size and frequency. To mimic an IoT installation we use two first version Raspberry Pis (single 700MHz ARM11 core, 512M RAM) for the subscriber and the publisher. The server is a Raspberry Pi Version 2 (quad-core ARM Cortex-A7, 1G RAM). Figure 2 details the experimental setup.

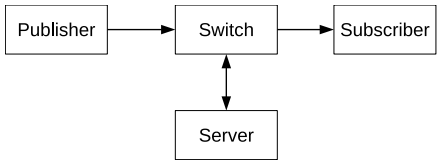


Fig. 2: Overview of the experimental setup.

We consider message sizes of 256 bytes, 1 KiB, 100 KiB, and 1 MiB. The 256 byte messages are analogous to command or status update packets from simple IoT devices such as thermostats or light bulbs. The larger messages represent outputs from more complex devices, such as cameras, lidar, and smart sensors for Industry 4.0. We generate messages with a CSPRNG to eliminate effects from any internal compression. We publish messages at 30Hz, 60Hz, and 100Hz. The lower frequencies resemble routine status updates from passive IoT devices (which are often much slower than 30Hz). The 60Hz rate is a common camera FPS measure, and the 100Hz rate may be used for high-frequency sensors.

We consider three factors: (i) choice of framework; (ii) message size; and (iii) message frequency. All frameworks are configured with persistence disabled because we are targeting online anomaly detection. The clients and server are synchronized using PTP, a network level time synchronization protocol

TABLE I: Features comparison among the analyzed frameworks.

Feature / Framework	Redis	Kafka	NATS	RabbitMQ
Supported data structures	strings, hashes, lists sets, bitmaps	Structured commit log	Queue	Queue
Message ordering	Yes	Yes	Yes	Yes
Client-side languages	About 49 different languages	About 17 different languages	C#,Go,Java,Node.js,Python	About 30 languages
Storage	In-memory dataset and saving in disk	Disk	In-memory or disk	In-memory (saving logs in files)
Written in	C	Scala/Java	Go	C
Message publication	Pub-Sub	Pub-Sub	Pub-Sub	Pub-Sub
Replication	Master-slave	Replicated cluster	Fault Tolerance/Partitioning	Clustering
Protocol	REdis Serialization Protocol (RESP)	Binary protocol over TCP	Based on Google protocol buffer	AMQP

capable of microsecond accuracy [7]. Latency is measured by including the timestamp at which a message is sent within the message itself. The subscriber then notes the timestamp at which it receives the message and subtracts to find the latency. Throughput is measured by multiplying the current message size by the time it takes all the messages in a single run to arrive, then dividing by the total time for that trial.

One sample of our experiment consists of one idealized minute of message transmission. That is, at 100 Hz we expect 6000 messages to be transmitted in a minute. If transmission takes longer than a minute, the experiment waits for all messages to be transmitted. For each configuration ($4 \times 3 \times 4 = 48$), the experiment is run five times. Due to the large startup time of these services, experiments on each framework were run in order: Redis, Kafka, NATS, and then RabbitMQ. Therefore, this is not a fully randomized experiment.

B. Results and Discussion

Consider the latency and throughput results in Figures 3 and 4. In our setup, both Kafka and NATS were unable to transmit large messages. The subscribers hung after a handful of messages were received for all 1 MiB messages to Kafka and NATS, and also for the 100 KiB messages at 100 Hz to Kafka. At high frequencies, Kafka dropped 15% of messages (note that Kafka was configured to keep messages 1 ms, so they would not persist to disk) and NATS dropped 3%. Redis also dropped 0.16% of the 1 MiB messages at 60 Hz and 100 Hz. RabbitMQ dropped no messages.

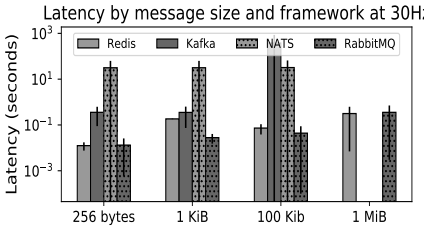
Redis and RabbitMQ behave comparably in all analyzed scenarios for both metrics. At small message sizes (256 bytes and 1 KiB) throughput is comparable across all four systems. This points to the networking capability of the platform as the primary bottleneck for small messages. Even at high message sizes, the throughput values remain comparable for all frameworks except NATS. This is likely due to the NATS Streaming server connecting to the main NATS server, adding a level of indirection. This indirection also impacts the latency of messages sent through NATS. The primary differences between the frameworks are observed in the latency results. RabbitMQ and Redis maintain lower and more consistent latencies across all message sizes; Kafka is significantly slower but still consistent, but NATS is more than an order of magnitude slower and more variable than the other frameworks for all message sizes. We also observe an increase in the variance of message latencies as the data size increases. Especially for Kafka and NATS at 100 KiB and 1 MiB. This is likely due to longer waiting times in buffers.

As an application of a data streaming framework in embedded systems, consider the LIDAR sensor on an autonomous vehicle. The autonomous vehicle could publish the current power consumption of the sensor into the framework and a detector would continuously analyze its state. When the power state goes to off and other sensors are still on, then an anomaly is reported. Alternately, the vehicle could publish the gear pattern from the autonomy software into the data streaming framework. Detectors would then monitor this stream for driving irregularities. The data streaming server would not run on the same platform as the autonomy software; it would run on a smaller, low power system solely tasked with motoring the vehicle. The choice of embedded system to run the framework is constrained by the operating system requirements, the amount of data collected, and the complexity of the anomaly detector. A cloud server may be needed as the number of clients and detectors grows.

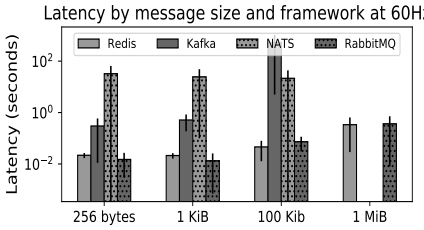
IV. Related Work

There are comparisons between data streaming frameworks available online, but they often lack scientific rigor (*i.e.*, do not entirely describe the experimental environment) and do not target embedded systems. For instance, Yigal discusses the throughput in Kafka and Redis but does not execute both on the same platform [8]. Treat compares the throughput and latency in Kafka and NATS, using a high-performance server. They find that Kafka and NATS present a similar performance in both metrics [9].

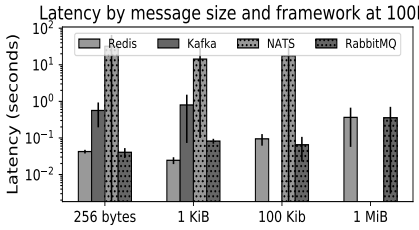
Data streaming frameworks have been used in several works to detect errors and anomalies in different systems. Lopez et al. discuss the characteristics and compare three stream processing platform (Apache Spark, Flink, and Storm) in terms of throughput using a threat detection application [10]. Solaimani et al. used Apache Spark to detect anomaly for multi-source VMware-based cloud data center [11]. Subramaniam et al. proposed a framework to online detect anomalies (outliers detection) in wireless sensors networks [12]. However, the authors only implemented the framework in a simulator. Du et al. proposed a network anomaly detector based on Apache



(a) Latency at 30Hz.

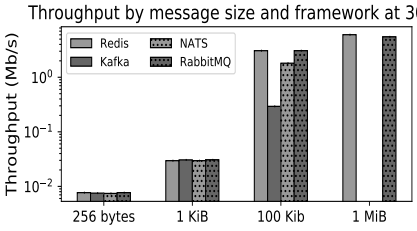


(b) Latency at 60Hz.

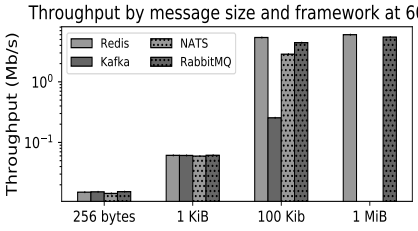


(c) Latency at 100Hz.

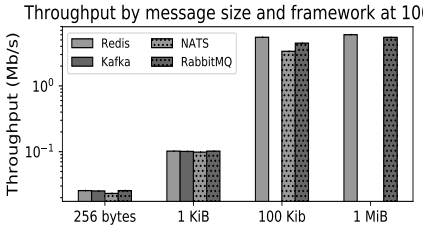
Fig. 3: Experimental latency results.



(a) Throughput at 30Hz.



(b) Throughput at 60Hz.



(c) Throughput at 100Hz.

Fig. 4: Experimental throughput results.

Storm [13]. Shi et al. implemented an online fault diagnosis system based on Apache Spark for power grid equipment [14].

V. Conclusion

As security becomes a growing concern in IoT systems, we turn to anomaly detection techniques to monitor the correctness of devices. Collecting traces for such systems requires an efficient data collection framework that will run on embedded devices. We compared Redis, Kafka, NATS, and RabbitMQ as publish-subscribe brokers on original Raspberry Pis.

Both Redis and RabbitMQ performed nearly identically. They are both C programs designed with the specific goal of lightweight message transmission, making them suitable for embedded systems. Clients for both Redis and RabbitMQ are widely available and require little more than an open network socket. We would recommend either of these systems for supporting trace-based anomaly detection in embedded systems. However, we would not recommend Kafka or NATS. Both are designed for web-based usage, focusing on delivery, concurrency, and fault-tolerant guarantees, rather than raw performance. These guarantees may not be required for an anomaly detection system for IoT devices.

As future work, we plan to integrate and evaluate several embedded system anomaly detectors, such as SiPTA [15] and arrival curves [16] in a data streaming infrastructure based on Redis targeting IoT and embedded systems. Other future work could consider additional low-level publish-subscribe brokers, and use a true real-time operating system.

References

[1] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection for discrete sequences: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 24(5):823–839, May 2012.

[2] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[3] Redis website, Jan 2018. Available online: https://redis.io/.

[4] Apache kafka website, Jan 2018. Available online: https://kafka.apache.org/.

[5] Nats website, Jan 2018. Available online: https://nats.io/.

[6] Rabbitmq website, Jan 2018. Available online: http://www.rabbitmq.com/.

[7] J. Han and D. K. Jeong. A practical implementation of ieee 1588-2008 transparent clock for distributed measurement and control systems. *IEEE Trans. on Inst. and Meas.*, 59(2):433–439, Feb 2010.

[8] Asaf Yigal. Kafka vs. redis: Log aggregation capabilities and performance, Nov 2016. Available online: https://logz.io/blog/kafka-vs-redis/.

[9] Tyler Treat. Benchmarking nats streaming and apache kafka, Dec 2016. Available online: https://dzone.com/articles/benchmarking-nats-streaming-and-apache-kafka.

[10] M. A. Lopez, A. G. P. Lobato, and O. C. M. B. Duarte. A performance comparison of open-source stream processing platforms. In *2016 IEEE GLOBECOM*, pages 1–6, Dec 2016.

[11] M. Solaimani, M. Iftekhar, L. Khan, B. Thuraisingham, J. Ingram, and Sadi E. Seker. Online anomaly detection for multi-source vmware using a distributed streaming framework. *Softw. Pract. Exper.*, 46(11):1479–1497, November 2016.

[12] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos. Online outlier detection in sensor data using non-parametric models. In *Proc. of the 32Nd VLDB*, pages 187–198, 2006.

[13] Y. Du, J. Liu, F. Liu, and L. Chen. A real-time anomalies detection system based on streaming technology. In *2014 Sixth IHMSC*, volume 2, pages 275–279, Aug 2014.

[14] W. Shi, Y. Zhu, T. Huang, G. Sheng, Y. Lian, G. Wang, and Y. Chen. An integrated data preprocessing framework based on apache spark for fault diagnosis of power grid equipment. *Journal of Signal Processing Systems*, 86(2):221–236, Mar 2017.

[15] Mohammad Mehdi Zeinali Zadeh, Mahmoud Salem, Neeraj Kumar, Greta Cutulenco, and Sebastian Fischmeister. Sipta: Signal processing for trace-based anomaly detection. In *Proc. of the EMSOFT*, pages 1–6, New Dehli, India, Oct. 2014.

[16] M. Salem, M. Crowley, and S. Fischmeister. Anomaly detection using inter-arrival curves for real-time systems. In *ECRTS*, France, 2016.