

Message brokering evaluation for live spacecraft telemetry monitoring, recorded playback, and analysis

Daren Lee¹ and Marc Pomerantz²

Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109

Live monitoring and post-flight analysis of telemetry data play a vital role in the development, diagnosis, and deployment of components of a space flight mission. Requirements for such a system include low end-to-end latency between data producers and visualizers, preserved ordering of messages, data stream archiving with random access playback, and real-time creation of derived data streams. We evaluate the RabbitMQ and Kafka message brokering systems, on how well they can enable a real-time, scalable, and robust telemetry framework that delivers telemetry data to multiple clients across heterogeneous platforms and flight projects. In our experiments using an actively developed robotic arm testbed, Kafka yielded a much higher message throughput rate and a consistent publishing rate across the number of topics and consumers. Consumer message rates were consistent across the number of topics but can exhibit bursty behavior with an increase in the contention for a single topic partition with increasing number of consumers.

Nomenclature

<i>AMQP</i>	=	Advanced Message Queuing Protocol
<i>API</i>	=	Application Programming Interface
<i>JMS</i>	=	Java Messaging Service
<i>JSON</i>	=	Javascript Object Notation
<i>RAM</i>	=	Random-access Memory

I. Introduction

Live monitoring and post-flight analysis of telemetry data play a vital role in the development, diagnosis, and deployment of components of a space flight mission. For example, this telemetry data can be used to assess the real-time health of a flight system or troubleshoot deficiencies during development or post-flight analysis. Requirements for such a system include low end-to-end latency between data producers and visualizers, preserved ordering of messages, data stream archiving with random access playback, and real-time creation of derived data streams. This paper evaluates recent developments in message brokering on how well they can enable a real-time, scalable, and robust telemetry framework that delivers telemetry data to multiple consumers across heterogeneous platforms and flight projects. In Section II, message brokers are introduced along with the main telemetry features on which we evaluate this technology. In Section III, the experimental performance configurations are detailed with the results of the message brokering experiments presented in Section IV. Section V details our system design which enables real-time, interactive telemetry visualization from application or browser based clients.

II. Background

Message brokers are mediators that handle communication among applications and have been applied in domains where real-time or distributed transactions are vital, such as financial transactions, web analytics, business intelligence, and big data analysis¹⁻⁴. They have also been identified as an enabling technology for cloud based mission design and operations⁵. Message brokers typically abstract the low-level networking protocols, message queuing logic, and routing details from the applications. Incorporating message brokering into a design architecture enforces distinct modular boundaries that decouple telemetry producers from consumers. This design effectively isolates the impact of changes to each individual module, increasing the system's re-usability and interoperability

¹ Research Technologist, Robotics Modeling and Simulation Group, Mail Stop 198-219, AIAA Member.

² Member of Technical Staff, Robotics Modeling and Simulation Group, Mail Stop 198-235, AIAA Member.

among heterogeneous applications and platforms. Common features of modern message broker frameworks include high throughput, low latency end-to-end communication via a publish-subscribe methodology, scalability to distributed resources such as cloud computing, and support for common programming languages. Differences include message delivery guarantees to subscribers, message ordering guarantees, message persistence for archival purposes, and ecosystem support. For this evaluation, we chose two of the more common open source messaging systems, RabbitMQ and Kafka.

RabbitMQ (<https://www.rabbitmq.com>) is an Erlang-based Advanced Message Queueing Protocol (AMQP) broker with a mature ecosystem of tools⁶. It provides rich routing capabilities through exchanges, bindings, and queues. Messages are published to exchanges that distribute the messages to queues using rules called bindings. The messages are then either pushed to queue subscribers or pulled by consumers on demand. Exchange types include direct (unicast messaging), fanout (broadcast messaging), and topic (multicast messaging). For message delivery guarantees, the RabbitMQ broker uses acknowledgements to guarantee at-least-once delivery and without acknowledgements, guarantees at-most-once delivery. AMQP does not specify any message ordering guarantees. Distributed RabbitMQ is possible through clustering where the cluster appears as a single logical broker or federation where an exchange on one broker receives all messages from another broker.

Kafka (<http://kafka.apache.org>) is a high throughput Java-based messaging system that leverages Apache Zookeeper (<https://zookeeper.apache.org/>) for synchronous and distributed storage⁷. Unlike most other message brokering systems that only provide built-in transient messaging, Kafka provides persistent messaging through logs stored as files. Kafka achieves parallelism by dividing topic data into partitions to allow multiple consumers to simultaneously access data. Each partition is an ordered sequence of messages and each message is assigned a unique identifier called the offset. Consumers can then use the offset to achieve random access to any of the messages since Kafka retains all published messages whether or not they were consumed. Kafka guarantees that within a partition, a consumer will be delivered the messages in order. Ordering is not guaranteed across multiple partitions. Kafka does not use message acknowledgements and delivery guarantee is delegated to the consumer to keep track of its partition offset state. Kafka can run as a distributed cluster, using an explicit model where the producer knows it is dividing a topic's messages across several nodes.

In addition to low end-to-end latency and high throughput, the major capabilities needed for a multi-mission telemetry framework are preserving telemetry data order, supporting multiple consumers on heterogeneous systems, providing telemetry archiving and playback, and enabling creation of derived telemetry data. While message ordering preservation is an atypical use case for most message brokering systems that use parallelization to achieve higher throughput, supporting multiple heterogeneous consumers is a common use case for modern messaging systems to support today's wide landscape of software and hardware devices. The messaging system achieves this by specifying on-the-wire protocols rather than application programming interfaces (API) such as the Java Messaging Service (JMS). On-the-wire protocols allow greater interoperability as any programming language can directly interface with the messaging system whereas API-based systems require a translation layer from the base programming language to the target language. Messaging archiving and playback is also an atypical use case for most message broker systems where the common capability is to provide transient messages that only persist until a delivery guarantee is met. Derivative messages are out of the scope of message brokers and are typically handled through stream processing frameworks.

III. Design & Methodology

To evaluate the open source messaging systems, we acquire telemetry data generated from an actively developed robotic arm (Sampling Lab Universal Robotic Manipulator) that is part of the JPL Environmental Development Testbed for Mars 2020 Sampling and Caching System. The robotic arm produces 130 different topics of data, with a mixture of data rates of 1000Hz and 100Hz. Each topic represents a single scalar data stream. Our messaging testbed framework interfaces with the robotic arm software through C language bindings for RabbitMQ (<https://github.com/alanxz/rabbitmq-c>) and Kafka producers (<https://github.com/edenhill/librdkafka>) and publishes the telemetry in JSON serialized format. Both the producer rate of the robotic arm and the publisher rate of the messaging layer in the testbed framework can be user-defined. For example, the robotic arm data can produce data at 200Hz and the telemetry message server can publish at 60Hz by aggregating data between sends. Three computers are used for the testbed: (1) an 8-core desktop with 2.90 GHz Intel Core i7-3920XM CPUs with 32GB of RAM that hosts the RabbitMQ v3.4.2 and Kafka v0.8.1 servers, (2) a 6-core desktop with 3.33 GHz Intel Xeon X5680 CPUs with 12GB of RAM that run the consumer threads, and (3) a notebook with a 2.90 GHz i7-3920XM CPU with 16GB RAM that runs the robotic arm software to publish the telemetry data. All computers are on the same local area network to minimize any networking latencies.

The first experimental configuration compares the publishing rate of the messaging systems. For RabbitMQ, topic routing is used for more efficient bandwidth usage and message acknowledgements are disabled to favor speed over message reliability. While both messaging systems support multiple producers, in order to preserve message ordering, we chose to enforce the ordering at the broker for simplicity by using single threaded producers. Moreover, Kafka is configured to use a single partition per topic to ensure message ordering. While using a single Kafka partition is the simplest method to achieve ordering, it may introduce contention when multiple consumers attempt simultaneous access. To test the limits of using a single-threaded messaging publisher, the message publishing rate was set to mirror the producer rate which was varied from 1 to 1000Hz. In the main thread, the producer buffers the messages to a synchronized queue and in a worker thread, the publisher reads and sends the messages. As shown in Figure 1, since Kafka outperforms RabbitMQ in message throughput, the remainder of the experiments focuses on discovering the limits with a single Kafka partition configuration, specifically on how well the message latency scales with the number of consumers and topics. On the client side, the consumer rate per topic is the metric used for evaluation. On the server side, the publishing rate is used as the metric. Ideally, the consumer rate should equal the publishing rate for a topic. For the experiments below, the publishing rate is set to 60Hz so the ideal consumer rate per topic is 60 messages per second.

For the Kafka set of experiments, consumers using the Python language bindings (<https://github.com/mumrah/kafka-python>) were employed. The Python script spawns multiple consumer processes, each using a unique consumer group id to allow full access to the partition. Each consumer process can subscribe to one or many user specified topic streams. At initialization, each consumer queries the Kafka server for the latest offset metadata. Alternatively, a user specified offset can be used to retrieve archived data. Unless stated otherwise, the consumers are configured to retrieve the latest messages. For the second experiments, the effect on the consumer and publisher rates is studied by varying the number of topics for a single consumer and then the number of consumers for a single topic. In the last set of experiments, both the number of consumers and topics were varied.

IV. Results and Discussion

As shown in Figure 1, for a single-threaded producer, the message publishing rate to consumers is bound by the publisher's send rate. For RabbitMQ, 7.5k messages/sec can be delivered without any latency and for Kafka, 60k messages/sec. For our deployed telemetry system, the desired message rate with the required number of channels is 9k message/sec based on a 60Hz data refresh. For this modest message rate, RabbitMQ underperforms while Kafka performs well to deliver real-time telemetry data to the consumer clients. For this reason, the remainder of the experiments delves deeper into the Kafka performance.

As shown in Figures 2 and 4, the sustained publishing rate for Kafka is constant when varying either the number of topics or consumers. This reflects Kafka's design philosophy of not throttling the publishing rate due to latencies with the consumer, which can occur with designs that ensure reliability delivery guarantees through acknowledgements.

The effect of contentions using a single partition for each topic on the Kafka server can be seen with the increased average consumer rate and standard deviation with the number of consumers. The increase in the average consumer rate is a result of a consumer falling behind the incoming messages. At the next pull request, the Kafka server will batch and send the older messages. Since reading from the log is much faster than the live data stream rate, the consumer will receive a higher number of messages in bursts, as shown in Figure 3. The contention rate of the topic partition increases with the number of consumers, as shown with the more consistent rate with 64 consumers as compared to 1024 in Figure 3. Other observed events of latency on the Kafka server occurred when more than 512 consumers simultaneously requested meta-data information for each topic. Retrieval of the meta-data causes the server to momentarily stop processing producer messages, most likely for synchronization purposes.

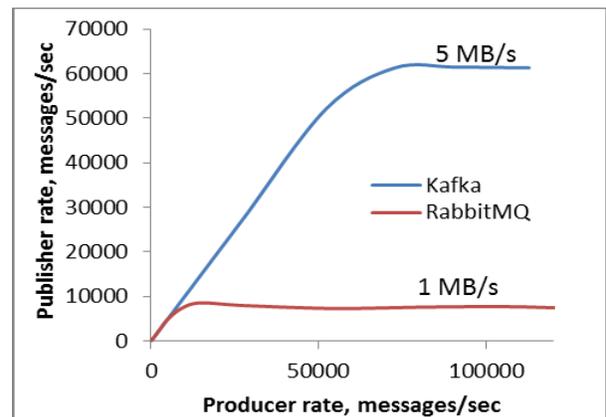


Figure 1. Comparison of publishing rates between RabbitMQ and Kafka using a single threaded producer. The robotic system has 130 different telemetry data topics with a mixture of 100Hz and 1000Hz data rates.

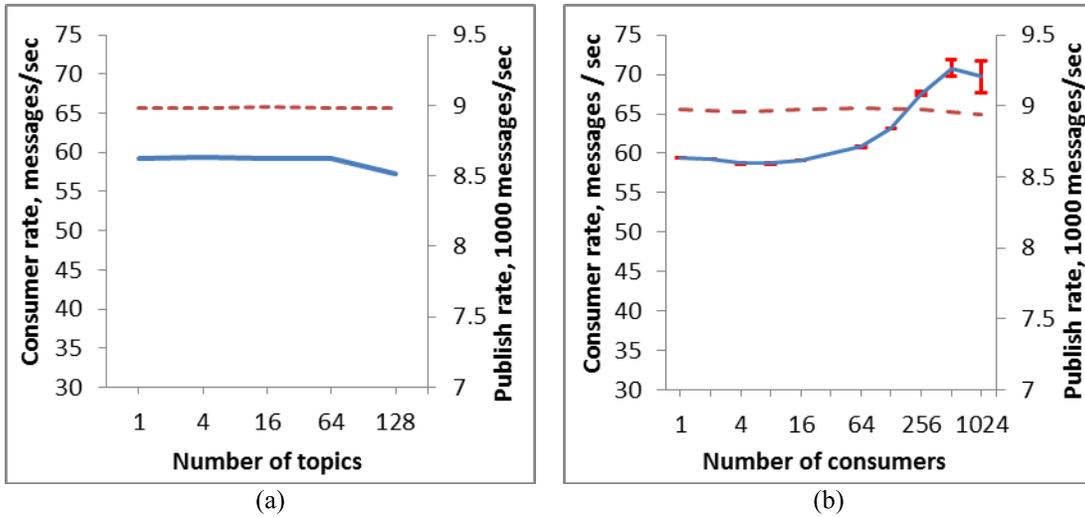


Figure 2. Sustained Kafka consumer (solid line) and publish rates (dashed line) for varying (a) number of topics with a single consumer and (b) number of consumers with a single topic. In (b), the average consumer rate is plotted with the standard deviation. The publish rate is consistent across number of topics and consumers.

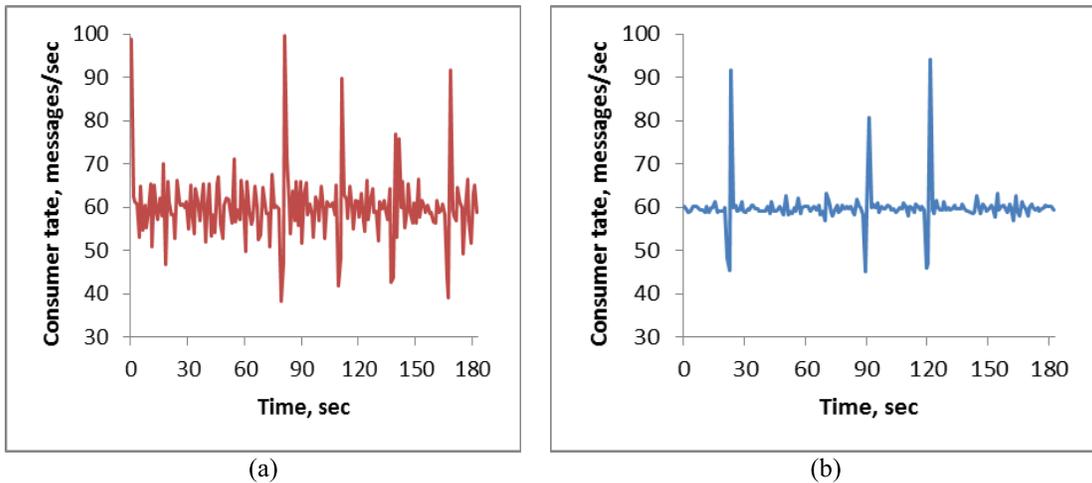


Figure 3. Detailed consumer rate for a single consumer instance in a group of (a) 1024 consumers and (b) 64 consumers, showing the effect of contentions using a single topic partition in Kafka. The message rate is more consistent with fewer consumers and contentions.

The consume rate for multiple topics in a single consumer is constant until the largest number of topics where it begins to tail off as the accumulative time to process all the messages grows, as shown in Figure 3a. With multiple topics and consumers, this effect is magnified as the CPU and memory resources are exhausted on our 6-core host. A more appropriate test would be to distribute the consumers across multiple host machines so the host resources are not limiting the consumer rates.

For multi-mission support, we believe that Kafka is a viable and scalable solution with its consistent publishing rate across topics and consumers. We recommend that the following issues are considered in the server configuration and design:

1. For publishing data, employ a work queue design pattern to separate the data producer and message publisher. To accommodate arbitrary data rates, buffer messages in the work queue so the data producer and message publisher can run at different rates. Buffering messages also helps with the bandwidth efficiency by decreasing the routing overhead per data point.
2. If message latencies arise from a single Kafka server, distribute topics across multiple Kafka servers.

3. If multiple runs of data need to be archived, unique name spaces must be created for topics. Currently, Kafka v0.8.1 does not support hierarchical topic names.

On the client side, we recommend the following design issues be considered:

1. Based on the number of topics per consumer, divide the topics among several consumers and design the client around asynchronous I/O or event-driven programming to increase the consumer rate. Parameterize these values for quick adoption to different configurations.
2. Design the client to expect bursty messages to avoid slowing the client.

An alternative to preserving message order without enforcing a single topic partition is to employ an out of order delivery scheme leveraging multiple partitions per topic. Publishers could send message to the partitions in a round-robin fashion. Consumers would then subscribe to all the partitions on a topic and then reassemble the messages in order. The tradeoff between the throughput gained using multiple partitions and the increased overhead incurred on the receiving consumer side must be carefully weighed. Also for archival playback a proper scheme to retrieve the round-robin messages must be designed since the default is to send all the data for one partition at a time.

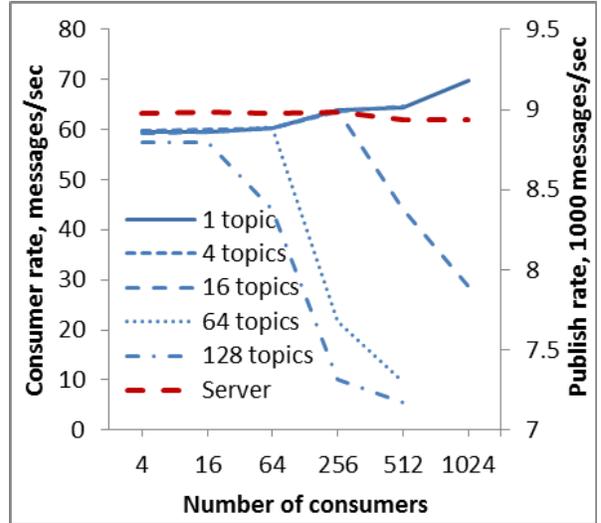


Figure 4. Sustained average consumer rates (blue) and publish rates (red) for combinations of number of topics and consumers.

V. System Architecture

A driving principle behind our testbed system architecture is to support both software and hardware heterogeneous platforms. To achieve this, our design provides both a native and a browser-based interface, as shown in Figure 5. Through the native interface, applications communicate directly with the Kafka server using language bindings, like C and Python, to create custom consumers and subscribe to topics. Supporting a browser-based interface allows built-in migration to multiple devices, such as desktop machines, tablets, and phones. For the browser-based interface, the main challenge is to overcome the uni-directionally communication of the Hypertext Transfer Protocol (HTTP). We leverage node.js⁸ to provide the middleware between web applications and Kafka

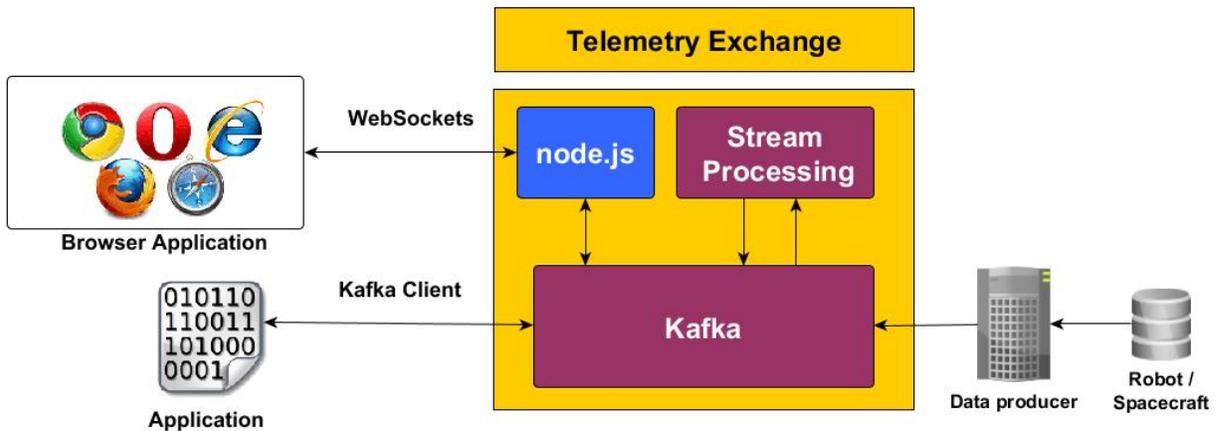


Figure 5: System architecture for multi-mission telemetry framework based on a Kafka messaging system supporting browser-based and native clients.

through kafka-node (<https://github.com/SOHU-Co/kafka-node>) and the WebSockets protocol. The WebSockets protocol allows full-duplex communication unlike traditional HTTP. Hence, a Javascript-based web application can send a subscribe request to the node.js server which creates a consumer topic instance through kafka-node. Messages from Kafka are then sent back to the node.js server, where they are forwarded to the browser via WebSockets.

An optional component of our design is the stream processing module. Stream processors, like Apache Storm (<https://storm.apache.org/>), provide real-time computation systems that consume real-time streams of data and perform arbitrary sequence of calculations specified through graph topologies. They abstract away the complexity of modeling data processing as a graph of computational nodes and scaling data processing, synchronization, and communication to large distributed systems. Stream processing frameworks have been used effectively in the field of real-time big data analysis¹. For our telemetry system, stream processing can be used to calculate derivative channels, detect anomalies, or diagnose faults.

VI. Conclusion

For a real-time telemetry monitoring and playback framework, the Kafka messaging system provides a robust solution. Its high message throughput with a single threaded producer, consistent publishing rate across topics and consumers, and built-in archiving and playback capabilities make Kafka well suited to meet the demands of a telemetry framework. From the flight project perspective, based on our experimentation, we believe that the Kafka messaging system can support testbed as well as assembly, test, and launch operations (ATLO). The main challenge is to design clients that fit the constraints of the desired data rate, required number of topics and consumers, and target processing platforms. For mission operations, we believe that Kafka's built-in replication and redundancy protocols for the server side are well suited to provide robust fail over mechanisms. However, this needs further investigation as it was not addressed in this work. Other areas of future work include performance analysis of distributed clustering and using cloud-based servers. As computing technologies and platforms evolve, choosing architectures that decouple key functionality will facilitate technology adoption for future space flight operations.

Acknowledgments

The research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

References

- ¹Marz, N., and Warren, J., *Big Data: Principles and best practices of scalable realtime data systems*, Manning Publications Co., 2015.
- ²Subramoni, H., Marsh, G., Narravula, S., Lai, P., and Panda, D.K. "Design and evaluation of benchmarks for financial applications using Advanced Message Queuing Protocol (AMQP) over InfiniBand." High Performance Computational Finance, 2008. WHPCF 2008. Workshop on. IEEE, 2008.
- ³Liu, X., Iftikhar, N., and Xie,X.. "Survey of real-time processing systems for big data." Proceedings of the 18th International Database Engineering & Applications Symposium. ACM, 2014.
- ⁴Kalashnikov, D., Bartashev, A., Mitropolskaya, A., Klimov, E., and Gusarova, N. "Cerrera: In-stream data analytics cloud platform." Digital Information, Networking, and Wireless Communications (DINWC), 2015 Third International Conference on. IEEE, 2015.
- ⁵Arrieta, J., Beswick, R., and Gerasimatos,D., "Cloud Computing for Mission Design and Operations," *AIAA SpaceOps 2012 Conference*, 2012.
- ⁶Vinoski, Steve. "Advanced message queuing protocol." IEEE Internet Computing 6 (2006): 87-89.
- ⁷Kreps, Jay, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing." *Proceedings of the NetDB*. 2011.
- ⁸Tilkov, Stefan, and Steve Vinoski. "Node. js: Using JavaScript to build high-performance network programs." IEEE Internet Computing 6 (2010): 80-83.