Replacing DDS with Apache Kafka as middleware technology for the Rubin Observatory Control System

Tiago Ribeiro[1], Russell E. Owen[2], David J. Mills[1], Michael A. Reuter[1], Andy W. Clements[1], and William O'Mullane[3]

[1]Vera C. Rubin Observatory Project Office, 950 N. Cherry Ave., Tucson, AZ 85719, USA
[2]University of Washington, Dept. of Astronomy, Box 351580, Seattle, WA 98195, USA
[3]Vera C. Rubin Observatory, Avenida Juan Cisternas #1500, La Serena, Chile

May 29, 2024

ABSTRACT

Once in operation, the Vera C. Rubin Observatory will execute a 10-year-long survey of the Southern sky known as the Legacy Survey of Space and Time (LSST). The Rubin Observatory Control System (Rubin-OCS) is a distributed system with each component in charge of a particular sub-system e.g.; the mount, the M1M3 mirror support system, etc. Each component is designed as an independent part of the system and they must work together during operations. Communication between the components is done by means of a software middleware.

The software middleware is the backbone of the system, allowing components to communicate with each other in a seamless way. The highly distributed nature of the Rubin-OCS places tight constraints in terms of latency, availability, and reliability for the middleware. The baseline implementation of the Rubin-OCS adopts the Data Distribution Service (DDS) technology for the middleware.

In the Rubin-OCS, the middleware is encapsulated with a layer of abstraction known as the Service Abstraction Layer (SAL), which currently uses the ADLink-OpenSpliceDDS implementation of the Data Distribution Service (DDS) message passing system.

Recently we performed a study of using Apache Kafka to replace DDS as the middleware technology for the Rubin-OCS. This study was motivated by the middleware-related challenges we faced while integrating the system as well as the recent announcements indicating that the adopted library may be deprecated during the lifespan of the project. The study involved throughput and latency studies and a proof of concept of our core libraries. Overall Kafka proved to be a suitable replacement for DDS.

Keywords: Vera C. Rubin Observatory, observatory control, middleware, Kafka, Data Distribution Service, DDS

1. INTRODUCTION

Middleware typically denotes software applications utilized for enabling communication in parallel and distributed computing.[1] This technology became popular with the advent of distributed systems, which come as a solution to the problem of parallel computation. As the complexity of software and hardware systems increase, it becomes impractical to develop and execute single processes on a single node. Hence, software evolved from monolithic applications, where a single program executes in a single process or node, to distributed applications, where the system is divided into a number of smaller applications; each running as their own process possibly on their own node. For these applications to work together coherently they must be able to communicate with each other, thus giving origin to middleware technologies.

How a distributed system is broken down into smaller pieces is heavily dependent upon the problem. Some systems are only broken down into a small number of components each still in charge of large contexts, others are

---

broken down into many small applications that are in charge only of small simple tasks. The latter has gained substantial popularity recently and is commonly referred to as *microservice architecture.* This architecture is behind many popular large services in use today like those from Google and Amazon.

The architecture of distributed systems can take many shapes and forms. For instance, some systems are designed to emulate monolithic applications. The application is composed of a number of smaller applications but there is a hierarchical organization, with components at the top, in charge of communicating and operating components at the bottom. The advantage of these systems is that they are easier to understand and to maintain. Since each component is isolated from the rest of the system, and only communicates with a components on the top and bottom of the hierarchical chain, adding new components is relatively easy and have minimal impact in the system. The disadvantage is that the system is more vulnerable to outages, if a component on the top of the hierarchical chain becomes unavailable those components at the bottom also become unavailable.

More modern distributed systems have favor less hierarchical approaches, following the principles of reactive systems. In these systems each component is designed as an independent entity that reacts to input data, be it from other components or external data services. Since these systems are designed with separation of concerns in mind (e.g. each component must be able to act independently), reactive systems are usually extremely resilient. If one component becomes unavailable, the others are expected to continue to operate, taking precautions to deal with the missing agent. At the same time, it can become quite burdensome to update and grow these systems as their complexity increases exponentially with the number of different components.

In any case, the middleware plays a crucial role in any kind of distributed system, acting as the glue that binds the system together.

The Vera C. Rubin Observatory[2] Control System (Rubin-OCS) is designed following the principles of a distributed, reactive architecture. The system is composed of a number of independent components that work together to execute cohesive operations. The middleware is encapsulated in the Service Abstraction Layer (SAL), which currently uses the ADLink-OpenSpliceDDS implementation of the Data Distribution Service (DDS) message passing system.[3,4] Like similar projects before us,[5] we experienced considerable challenges with DDS while rolling out the system into production. While we were able to resolve the majority of those issues by fine tuning ADLink-OpenSpliceDDS configuration, a number of recurrent issues still remain, which are hard to track down and resolve due to their non-repeatable nature. However, one of the most critical aspects, that inspired us to look for a replacement is that the adopted library is currently bound for sunsetting.

With all that in mind we decided to explore alternative solutions for the Rubin-OCS. After exploring a number of alternatives,[6] we decided to adopt Kafka as the message passing system, with AVRO-schema encoding for the topics schema.[7]

In the following sections we describe the work done to validate the adoption of Kafka and the initial results obtained with the system running in one of our test stands. In Section 2 we provide a comparison between DDS and Kafka in the context of the Rubin-OCS, including a description of their similarities, differences and a performance review. In Section 3 we describe our initial experience deploying the full system on a test stand, followed by conclusions in Section 4.

2. COMPARISON BETWEEN DDS AND KAFKA

In this section we present a comparison between DDS and Kafka in the context of the Rubin-OCS.

2.1 Similarities

One of the main similarities between DDS and Kafka, which weighed considerably when we decided to explore Kafka as a viable replacement, is the fact that both use a publish-subscribe model to transmit and receive messages. Each message is associated with a "topic", which can be understood as a communication channel to exchange a particular type of message.

In both cases, the message transport happens over TCP/IP protocol.

Both DDS and Kafka have support for durability, which is the capability to retrieve historical data (often referred to as "late-joiner"), and our control system architecture relies heavily on this feature. As a node starts

up, it reads the most recent historical message for each event it subscribes to in order to get the current state of the system, without having to resort to requesting information to be republished.

2.2 Differences

One of the main distinctions between these two systems is that Kafka is brokered, whereas DDS is distributed, with no central broker.

With DDS each node discovers the other nodes, using UDP multicast, as the application starts up. Durability is achieved by the election of a main node which stores all the historical data and distributes them as needed, switching which node acts as the primary source of data if one fails.

With Kafka each node connects to a broker (and schema registry); there is no discovery phase and no need for UDP multicast. Kafka writers write messages to the broker, and readers read messages from the broker. This is a much simpler topology than DDS, but it does require that each message be sent to the broker and then again to each subscriber, which may result in increased latency. Durability is a natural attribute of the system as the brokers store messages for distribution by design. Reliability and performance can be increased by running multiple brokers in parallel, and by using more than one partition for each topic.

2.2.1 Topic Schema and Schema Evolution

In DDS each individual topic has a specific pre-defined schema. In the specification that we are using, DDS does not support schema evolution. This means that, once a topic is created all further readers and writers must have the same topic schema. Thus, in order to prevent collisions between old topic schemas and new topic schemas, we include a hash to the topic name that changes whenever the schema changes. Our high-level software hides that hash from users. Although this helps us avoid schema collisions in the running system, this causes silent errors if systems are brought up with different topic schemas. We have managed to avoid this issue by enforcing a tight deployment procedure that guarantees all systems are running with the same schema version.

Kafka topics on their own have no concept of schema, working more like a communication channel than as a specific message exchange. However, Kafka ships with a schema register that can be used to register and validate topic schema. The schema register can be configured with different levels of support for schema evolution and supports a variety of schema specifications. For the Rubin-OCS we adopted AVRO-schema, in part because it was already in use in the project but also because it provides great level of flexibility. For the time being we have only made limited use of all the capabilities provided by an advanced schema evolution mechanism. In the future we expect to be able to allow systems to update their interfaces more freely, considerably improving the timeline to deploy new features to the system.

2.2.2 Subtle Differences Between DDS and Kafka

DDS topics have the concept of "quality of service" (QoS) settings. There are many QoS settings, and the settings must match quite closely between publishers and subscribers of a particular topic, or else the topic cannot be constructed. This has been a source of much frustration over the years, though we ultimately found QoS settings that work.

The settings for publishing Kafka topics are far simpler than for DDS, and subscribers have no settings. Kafka publishers can specify the number of partitions (increasing the number of partitions increases throughput) and how many ACKs to listen for when writing a message (a tradeoff between latency and reliability).

DDS topics have the concept of "liveness" (one of the many QoS settings), whereas Kafka topics do not. We have configured our DDS topics such that messages are no longer alive when a CSC exits, in order to avoid collisions between older and newer versions of topic QoS when upgrading control software. Changes to a topic's schema are handled by changing a hash field in the DDS topic's name, as mentioned in Section 2.2.1. The result is a completely different topic, as far as DDS is concerned. Until we learned to do this, we frequently ran into a problem where we could not run a new version of a component, because it could not create its topics due to an incompatibility with cached "zombie" data. But marking topics as dead when an application exits means that historical data for a component that have quit is not unavailable from DDS.

Kafka has no concept akin to "liveness", neither does it need it. In most cases we can simply run the new application. If a topic's schema has changed in some incompatible way then we might have to clean up the broker and schema registry.

2.3 Performance

One of our main concerns in adopting Kafka as a replacement for DDS was to ensure its performance was on par with our system requirements. Our primary performance requirements are:

- Average latency better than 25ms.

- Latency standard deviation better than 3ms.

- Throughput better than X MB/s per topic.

- Consumers must be able to keep up with the data they read.

- Must be able to handle the full system throughput.

A major aspect of adopting Kafka, is that it is already in use in the DDS-based control system to feed data to the Engineering Facility Database (EFD). We have been operating the system with this architecture from the very beginning, which allowed us not only to gain experience with Kafka, but also to demonstrate that Kafka is able to handle the load. Overall, the only aspect we have not investigated is the latency in which messages are delivered from publishes to subscribers.

Thus for measuring performance we concentrated on the first four items.

We tested performance using three topics:

- MTM1M3 summaryState: One of the smallest topics.

- MTMount trackTarget: The command which we most care about latency. This is the command sent by the pointing component to the mount with on sky pointing demands. It is this operation that derives the latency values above.

- MTM1M3 forceActuatorData: The largest topic in the systems, which is also published at 50Hz. This topic sets the per topic throughput requirement.

For Kafka we performed the tests using two different settings:

- `ack=0`, meaning the writers will not wait for confirmation from the broker that the topic was received. This is not considered safe, and is provided purely to show how much latency is due to waiting for acknowledgement.

- `ack=1`, meaning the writers will wait for 1 acknowledgement from the broker. This is a very common configuration that is considered safe.

For DDS we used our standard configuration. All tests were executed on a dedicated node with the same configuration we use for our production environment.

The results are presented in the following sections.

System	ACKs	Topic	Latency (ms)			
			mean	stdev	min	max
DDS	n/a	summaryState	1	0	1	1
DDS	n/a	trackTarget	1	0	1	2
DDS	n/a	forceActuatorData	5	0	4	7
Kafka	0	summaryState	2	0	2	6
Kafka	0	trackTarget	2	0	2	4
Kafka	0	forceActuatorData	3	0	3	8
Kafka	1	summaryState	2	1	2	27
Kafka	1	trackTarget	2	1	2	25
Kafka	1	forceActuatorData	3	0	3	23

Table 1. Summary of the latency measurements performed with DDS and Kafka for three different topics. For Kafka we performed latency measurements with two different setups; `ack=0` and `ack=1`. When `ack=0` the writer will not wait for confirmation from the broker that the topic was written before proceeding, whereas with `ack=1` it will wait for at least 1 receive acknowledgement. The three different topics used in this test represents the smallest (summaryState) and the largest (forceActuatorData) topics in our system, plus an intermediary case that requires the lowest latency possible (trackTarget).

2.3.1 Latency

Latency was estimated by writing 2000 messages at 20 Hz and measuring the time between when each message was written and when the reading process received that message. The results are shown in Table 1.

Latency is only an issue for tracking demands, which are sent at approximately 20 Hz and specify position, velocity, and time. Tracking demands are presently sent at least 50 ms in advance, so it is possible that an occasional tracking command be late, due to an extreme outlier latency. This should not be a problem, because our specifications allow us to lose up to three sequential tracking commands, so we can afford to ignore (with a warning) the occasional late command.

We could also send tracking commands a bit farther in advance. This would decrease the responsiveness to offsets and new slews by the same small amount.

2.3.2 Write Speed

Write speed was measured by having two separate processes, a writer and a reader, writing and reading messages as quickly as possible. The test executed until the reader had received 10,000 messages. We decided to use the number of received messages as the stopping point in order to improve metrics in situations where there is data loss. The results are shown in Table 2.

System	ACKs	Topic	Write Speed messages/s	Read Speed messages/s	Messages lost
DDS	n/a	summaryState	16,553	16,580	0
DDS	n/a	trackTarget	13,557	13,577	0
DDS	n/a	forceActuatorData	1,912	1,492	2,632
Kafka	0	summaryState	4,730	4,739	0
Kafka	0	trackTarget	4,379	4,385	0
Kafka	0	forceActuatorData	3,267	3,262	0
Kafka	1	summaryState	2,065	2,066	0
Kafka	1	trackTarget	1,841	1,842	0
Kafka	1	forceActuatorData	1,652	1,652	0

Table 2. Write speed measurements for DDS and Kafka. As with the latency measurements in Section 2.3.1 the measurements were performed with 2 different configurations for the Kafka writer and for three different topics. See Table 1 for more details.

These throughputs all easily meet our requirements.

Note that, alongside write speed and message loss, we also report read speed. It is worth noting that this is not a measure of maximum read speed, as this is limited by the write speed. In general, as long as the reader can keep up with the writer, read speed should be approximately the same as write speed. It is not clear to us why the reported read speed is sometimes slightly higher than write speed. We believe that this might be because the writers starts shortly before the reader and it might have time to write some messages before the readers starts to read them. This would cause the reader to quickly read messages previously in the historical queue and suggests that it is possible to consume messages faster than we can produce it which is, in general, a desirable property in publish-subscribe systems.

The fact that some DDS messages were lost shows that our DDS testing harness can write large messages faster than it can read them. This is not a cause for concern, as long as we keep up with the actual rate at which messages are written at operational regimes (e.g. 50Hz).

3. SYSTEM WIDE DEPLOYMENT

Following the encouraging performance results we obtained with Kafka, we started a system-wide adoption project. We commenced by updating our high level Python library to replace DDS with Kafka. During this period we wanted to maintain compatibility between both versions of the library to ease the transition. Overall, the worked pretty well and we are able to provide compatible versions of the library that can be seamlessly swapped.

Although the majority of our components are written using the high level Python library, some of the most critical components are written in C++ and Java, namely; the pointing component, the M1M3 support system (both in C++) and the cameras control system (in Java). The second phase of the process was then to update our C++ and Java components.

Before we can finally run the system at the Summit, driving the observatory systems, we need to ensure it works as expected. With all our systems able to run with Kafka, we made a full deployment of all the components in one of our Test Stands. This is possible because all our systems are developed with a simulation mode that closely emulate the actual component behavior. We regularly use our components in simulation mode on our Test Stands for integration tests and for testing operations before deploying them at the Summit. In Figure 1, we show a screenshot of the user interface of our system running Kafka.
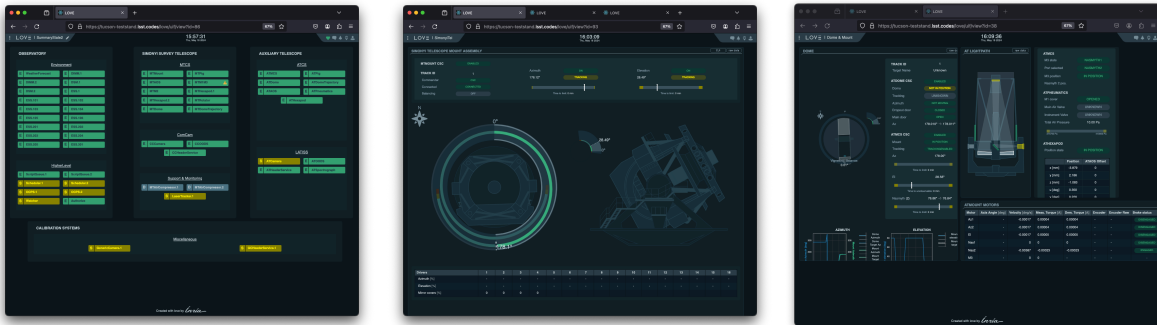

Figure 1. Screenshot of some of the main views of the LSST Operator Visualization Environment (LOVE) for the system running on Kafka at the Tucson Test Stand (TTS). The left hand panel shows the state of all the components currently running; green means the system is in Enabled state which is when they are operational, yellow systems are in Standby which means the system is alive but not able to perform any action and blue means the system is in Disable which means the system is configured and ready to operate but will not perform any action. Two states are not represented; Offline and Fault, which are states for components that are not running or when they experience unexpected behavior. The middle panel shows the view of the Simonyi telescope status, showing the current position of the telescope and the state of the different axis. The right hand panel shows the same view for the Auxiliary Telescope.

4. CONCLUSIONS

We have demonstrated the feasibility of replacing DDS with Kafka as the middleware technology for the Rubin-OCS. Our initial latency and throughput benchmarks have given us confidence in implementing the update, and our Service Abstraction Layer facilitated the overall process. We have started to execute more system-wide testing on one of our test stands, which have allowed us to gain more confidence and experience with the system. Migration to the new Kafka-based system is expected to happen in the near future once testing has concluded and the commissioning schedule allows it. Once migration is complete we plan to take advantage of schema evolution capabilities enabled by Kafka to improve rollout of components interfaces, reducing some of the burden on configuration management imposed by DDS.

ACKNOWLEDGMENTS

REFERENCES

[1] Buyya, R., Vecchiola, C., and Selvi, S. T., "Chapter 2 - principles of parallel and distributed computing," in [*Mastering Cloud Computing*], Buyya, R., Vecchiola, C., and Selvi, S. T., eds., 29–70, Morgan Kaufmann, Boston (2013). https://www.sciencedirect.com/science/article/pii/B9780124114548000024.

[2] Ivezić, Ž., Kahn, S. M., Tyson, J. A., Abel, B., Acosta, E., Allsman, R., Alonso, D., AlSayyad, Y., Anderson, S. F., Andrew, J., Angel, J. R. P., Angeli, G. Z., Ansari, R., Antilogus, P., Araujo, C., Armstrong, R., Arndt, K. T., Astier, P., Aubourg, É., Auza, N., Axelrod, T. S., Bard, D. J., Barr, J. D., Barrau, A., Bartlett, J. G., Bauer, A. E., Bauman, B. J., Baumont, S., Bechtol, E., Bechtol, K., Becker, A. C., Becla, J., Beldica, C., Bellavia, S., Bianco, F. B., Biswas, R., Blanc, G., Blazek, J., Blandford, R. D., Bloom, J. S., Bogart, J., Bond, T. W., Booth, M. T., Borgland, A. W., Borne, K., Bosch, J. F., Boutigny, D., Brackett, C. A., Bradshaw, A., Brandt, W. N., Brown, M. E., Bullock, J. S., Burchat, P., Burke, D. L., Cagnoli, G., Calabrese, D., Callahan, S., Callen, A. L., Carlin, J. L., Carlson, E. L., Chandrasekharan, S., Charles-Emerson, G., Chesley, S., Cheu, E. C., Chiang, H.-F., Chiang, J., Chirino, C., Chow, D., Ciardi, D. R., Claver, C. F., Cohen-Tanugi, J., Cockrum, J. J., Coles, R., Connolly, A. J., Cook, K. H., Cooray, A., Covey, K. R., Cribbs, C., Cui, W., Cutri, R., Daly, P. N., Daniel, S. F., Daruich, F., Daubard, G., Daues, G., Dawson, W., Delgado, F., Dellapenna, A., de Peyster, R., de Val-Borro, M., Digel, S. W., Doherty, P., Dubois, R., Dubois-Felsmann, G. P., Durech, J., Economou, F., Eifler, T., Eracleous, M., Emmons, B. L., Fausti Neto, A., Ferguson, H., Figueroa, E., Fisher-Levine, M., Focke, W., Foss, M. D., Frank, J., Freemon, M. D., Gangler, E., Gawiser, E., Geary, J. C., Gee, P., Geha, M., Gessner, C. J. B., Gibson, R. R., Gilmore, D. K., Glanzman, T., Glick, W., Goldina, T., Goldstein, D. A., Goodenow, I., Graham, M. L., Gressler, W. J., Gris, P., Guy, L. P., Guyonnet, A., Haller, G., Harris, R., Hascall, P. A., Haupt, J., Hernandez, F., Herrmann, S., Hileman, E., Hoblitt, J., Hodgson, J. A., Hogan, C., Howard, J. D., Huang, D., Huffer, M. E., Ingraham, P., Innes, W. R., Jacoby, S. H., Jain, B., Jammes, F., Jee, M. J., Jenness, T., Jernigan, G., Jevremović, D., Johns, K., Johnson, A. S., Johnson, M. W. G., Jones, R. L., Juramy-Gilles, C., Jurić, M., Kalirai, J. S., Kallivayalil, N. J., Kalmbach, B., Kantor, J. P., Karst, P., Kasliwal, M. M., Kelly, H., Kessler, R., Kinnison, V., Kirkby, D., Knox, L., Kotov, I. V., Krabbendam, V. L., Krughoff, K. S., Kubánek, P., Kuczewski, J., Kulkarni, S., Ku, J., Kurita, N. R., Lage, C. S., Lambert, R., Lange, T., Langton, J. B., Le Guillou, L., Levine, D., Liang, M., Lim, K.-T., Lintott, C. J., Long, K. E., Lopez, M., Lotz, P. J., Lupton, R. H., Lust, N. B., MacArthur, L. A., Mahabal, A., Mandelbaum, R., Markiewicz, T. W., Marsh, D. S., Marshall, P. J., Marshall, S., May, M., McKercher, R., McQueen, M., Meyers, J., Migliore, M., Miller, M., Mills, D. J., Miraval, C., Moeyens, J., Moolekamp, F. E., Monet, D. G., Moniez, M., Monkewitz, S., Montgomery, C., Morrison, C. B., Mueller, F., Muller, G. P., Muñoz Arancibia, F., Neill, D. R., Newbry, S. P., Nief, J.-Y., Nomerotski, A., Nordby, M., O'Connor, P., Oliver, J., Olivier, S. S., Olsen, K., O'Mullane, W., Ortiz, S., Osier, S., Owen, R. E., Pain, R., Palecek, P. E.,

Parejko, J. K., Parsons, J. B., Pease, N. M., Peterson, J. M., Peterson, J. R., Petravick, D. L., Libby Petrick, M. E., Petry, C. E., Pierfederici, F., Pietrowicz, S., Pike, R., Pinto, P. A., Plante, R., Plate, S., Plutchak, J. P., Price, P. A., Prouza, M., Radeka, V., Rajagopal, J., Rasmussen, A. P., Regnault, N., Reil, K. A., Reiss, D. J., Reuter, M. A., Ridgway, S. T., Riot, V. J., Ritz, S., Robinson, S., Roby, W., Roodman, A., Rosing, W., Roucelle, C., Rumore, M. R., Russo, S., Saha, A., Sassolas, B., Schalk, T. L., Schellart, P., Schindler, R. H., Schmidt, S., Schneider, D. P., Schneider, M. D., Schoening, W., Schumacher, G., Schwamb, M. E., Sebag, J., Selvy, B., Sembroski, G. H., Seppala, L. G., Serio, A., Serrano, E., Shaw, R. A., Shipsey, I., Sick, J., Silvestri, N., Slater, C. T., Smith, J. A., Smith, R. C., Sobhani, S., Soldahl, C., Storrie-Lombardi, L., Stover, E., Strauss, M. A., Street, R. A., Stubbs, C. W., Sullivan, I. S., Sweeney, D., Swinbank, J. D., Szalay, A., Takacs, P., Tether, S. A., Thaler, J. J., Thayer, J. G., Thomas, S., Thornton, A. J., Thukral, V., Tice, J., Trilling, D. E., Turri, M., Van Berg, R., Vanden Berk, D., Vetter, K., Virieux, F., Vucina, T., Wahl, W., Walkowicz, L., Walsh, B., Walter, C. W., Wang, D. L., Wang, S.-Y., Warner, M., Wiecha, O., Willman, B., Winters, S. E., Wittman, D., Wolff, S. C., Wood-Vasey, W. M., Wu, X., Xin, B., Yoachim, P., and Zhan, H., "LSST: From Science Drivers to Reference Design and Anticipated Data Products," *ApJ* **873**, 111 (Mar 2019). DOI: https://doi.org/10.3847/1538-4357/ab042c.

[3] Mills, D., Schumacher, G., and Lotz, P., "LSST communications middleware implementation," in [*Ground-based and Airborne Telescopes VI*], Hall, H. J., Gilmozzi, R., and Marshall, H. K., eds., *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* **9906**, 99065C (July 2016). DOI: https://doi.org/10.1117/12.2233099.

[4] Mills, D. and Schumacher, G., "Middleware design and implementation for LSST," in [*Software and Cyber-infrastructure for Astronomy*], Radziwill, N. M. and Bridger, A., eds., *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* **7740**, 77402C (July 2010). DOI: https://doi.org/10.1117/12.857233.

[5] Maartens, D. S. and Brink, J. D., "Selecting a simple, natively implemented middleware solution for the SALT control system," in [*Software and Cyberinfrastructure for Astronomy V*], Guzman, J. C. and Ibsen, J., eds., **10707**, 107070E, International Society for Optics and Photonics, SPIE (2018). https://doi.org/10.1117/12.2313106.

[6] Ribeiro, T., Clements, A., Mills, D., Reuter, M., and Owen, R., "The past, present and future of the Vera Rubin Observatory Control System Middleware," (August 2022). Vera C. Rubin Observatory TSTN-028, https://tstn-028.lsst.io/.

[7] Owen, R., "Exploring Kafka for Telescope Control," (April 2022). Vera C. Rubin Observatory TSTN-033, https://tstn-033.lsst.io/.