IndraFlow: Seamless Data Transfer and Transformation Between Internet of Things, Robot Systems and Cloud-Native Environments

Attila Csaba Marosi[1][a] and Krisztián Póra[1,2][b]
[1]*Laboratory of Parallel and Distributed Systems, Institute for Computer Science and Control (SZTAKI), Hungarian Research Network (HUN-REN), Hungary*
[2]*John von Neumann Faculty of Informatics, Óbuda University, Hungary*

Keywords: Data Analytics, IoT, IIoT, Big Data, Data Streaming, ROS, Data Bridge.

Abstract: In this paper we present our solution which aims to be a generic streaming data bridge. It utilizes a modular architecture with current support for MQTT, ROS1, ROS2, Kafka and relational database management systems (RDBMS), such as MySQL or PosgreSQL as data sources or destinations. Our solution also supports custom transformations of messages and using multiple sources and destinations within a single bridge instance. We compare our solution to existing generic streaming solutions (such as the GUI-based Apache NiFi) and custom-made bridge codes (such as a ROS to MQTT bridge). Next, we present two use cases for our solution from different projects. In the first use case ROS messages are received from drones, transformed and sent to a cloud-based Kafka cluster. The second use case is representing an industrial IoT use case where MQTT messages are received, transformed and sent to a PostgreSQL server for persistent storage. Finally, we evaluate the performance and reliability of our solution using the second use case.

1 INTRODUCTION

The rise of the Internet of Things (IoT) has led to an era where enormous amount of data is generated by countless devices. This presents a challenge in effectively using this data, as it requires robust infrastructure and connectivity. Similarly, for robots using the Robot Operating System (ROS) (Quigley et al., 2009), there's a growing need for a strong connection between them and cloud environments.

Message consumption patterns and messaging systems in distributed systems have evolved to meet the need for efficient communication and data processing, also supporting the challenges of IoT systems. Publish-subscribe (Eugster et al., 2003) and message queuing are two important patterns that have emerged. Publish-subscribe allows for decoupling of producers and consumers, enabling scalability and a loosely-coupled architecture. Message queuing ensures reliable and asynchronous communication, with features like load balancing and message persistence. Initially, generic messaging systems like email and instant messaging allowed basic communication but lacked complex integration capabilities. Message-oriented middleware (MOM) (Curry, 2004) introduced reliable messaging and clustering provided fault tolerance and scalability. Enterprise service bus (ESB) enhanced messaging systems with centralized infrastructure for managing message routing and transformation. For messaging systems beside custom-tailored solutions (Institut für Kraftfahrzeuge, 2023) (Lourenco et al., 2021) (Chaari et al., 2019), generic distributed message systems like Apache Kafka (Kreps et al., 2011), Apache Flink (Carbone et al., 2015) or Apache Pulsar (Pulsar, 2023) have gained popularity, handling high-throughput, fault-tolerant messaging for modern architectures and with data processing capabilities (Akidau et al., 2015).

This research focuses on the need for a reliable data bridge that connects robot systems, IoT devices and cloud environments, allowing for smooth and real-time data transmission and analysis across the different connected systems. The main contributions of the paper are as follows. We present our solution, IndraFlow, a generic streaming data bridge with support for MQTT (Light, 2017), ROS1, ROS2, Kafka (Wang et al., 2021) and RDBMS (e.g., MySQL or PosgreSQL) as data sources or destinations. Our solution also supports custom transformations of mes-

[a] https://orcid.org/0000-0001-9105-6816
[b] https://orcid.org/0009-0008-2229-9521

Marosi, A. and Póra, K.
IndraFlow: Seamless Data Transfer and Transformation Between Internet of Things, Robot Systems and Cloud-Native Environments.
DOI: 10.5220/0012622600003705
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 9th International Conference on Internet of Things, Big Data and Security (IoTBDS 2024)*, pages 191-198
ISBN: 978-989-758-699-6; ISSN: 2184-4976
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

sages and using multiple sources and destinations within a single bridge instance. We compare our solution to existing streaming solutions such as the GUI-based Apache NiFi, and to use case specific custom-made bridge codes. Additionally, we present two use cases: (a) we receive ROS messages from drones, transform them and sending them to a cloud-based Kafka cluster; and (b) in an industrial IoT scenario we receive MQTT messages, transform and send them to a PostgreSQL server for permanent storage. Finally, we evaluate our solution through the second use case.

The structure of the paper is as follows. In Section 2 we are discussing related works. In Section 3 we are introducing IndraFlow and present its architecture and the design decisions we chose. Additionally, we present two use cases where IndraFlow is used. In Section 4 we present performance and reliability evaluations using the second use case. Finally, Section 5 discusses future work and concludes the paper.

2 RELATED WORKS

We discuss related works in three categories using a top-down approach. First, we discuss high-level concepts of data streaming. Second, we discuss generic data flow solutions with focus on their Robot and IoT capabilities. We include in this category complete data streaming platforms that provide capabilities both for data sources and sinks and have computation capabilities either built-in or support via a plug-in mechanism. Third, we look at custom bridge solutions serving single use-cases. These solutions are typically developed serving a single use case. Additionally, we note here, that we relate the presented related works concepts to our solution in Section 3.

First, the Lambda (Marz, 2011) and Kappa (Kreps, 2014) architectures are two popular high-level approaches for building data processing systems. The Lambda architecture was designed for handling massive amounts of data in a fault-tolerant manner using a batch, a speed and a serving layer. On the other-hand the Kappa architecture simplifies the system by eliminating the batch layer, relying solely on the speed layer for processing both real-time and historical data. The Kappa architecture focuses on stream processing and streamlining the processing pipeline, while the Lambda architecture has both a batch and a streaming pipeline.

Next, Differential Dataflow (McSherry et al., 2013; Murray et al., 2013), developed by Microsoft, is a computational framework that enhances traditional data flow systems by enabling incremental and timely processing of data. It introduces the concept of differences, allowing for efficient handling of updates to large-scale datasets. By processing incremental updates instead of recomputing the entire dataset, it achieves significant performance improvements. It offers a declarative programming model for expressing complex data processing tasks in a concise and intuitive manner. Also, it includes fault-tolerance mechanisms to ensure data consistency and reliability.

A prominent example of the second related works category, Apache NiFi is an open-source data integration tool designed to automate the flow of data between systems. NiFi provides a user-friendly interface for designing data flows, allowing users to easily collect, process, and distribute data across various environments. It supports a wide range of data sources and destinations, however it does not support ROS directly. In case of running multiple data flows (e.g., different use cases) on a single cluster, all nodes instantiate all components. This means that the CPU and memory footprint has always a factor that is determined by only the complexity of the data flow and cannot be reduced by scaling the cluster. Apache NiFi MiNiFi (MiNiFi, 2023) is a sub-project of Apache NiFi and provides a complementary, low-footprint data collection approach. It functions as an agent near or adjacent to data sources such as sensors, edge devices or other systems. It has a low footprint, but can support only a subset of NiFi processors.

Logstash is an open-source data processing pipeline tool that allows users to collect, transform, and store data from various sources. It is part of the Elastic Stack (Elastic, 2023), along others such as the data collectors called as 'Beats' and Elastic Agent used for data ingestion. Logstash does not support MQTT and ROS as input or output. MQTT support is available via Filebeat, but only as a data source.

Integration frameworks (Ibsen and Anstey, 2018) (MuleSoft, 2023) (Google, 2023) enable the connection of different systems for data consumption and production. These support multiple data formats like such as XML, JSON, CSV and AVRO, and allow the transformation of messages between different systems and protocols. They typically provide a developer environment, where custom components can be developed (e.g., via Java code or XML descriptors) and deployed. These solutions can be considered as low-level, but highly configurable.

Fluvio (Fluvio, 2023) is an open-source streaming platform with built-in computation capabilities. Its composed of dedicated components for processing and stream handling. It supports horizontal scaling and is backed by a distributed key-store via an universal interface that currently works with Etcd (Etcd, 2023) and native Kubernetes (Brewer, 2015) key-value stores. Fluvio follows the unified cluster approach, where all components for streaming and stateful computation are integrated. On the other hand, Arroyo (Arroyo, 2023) is a distributed stream processing engine developed for the execution of stateful computations on different data streams. Its core is a distributed data flow engine. Its goal is to achieve efficient processing of both bounded and unbounded data sources. In contrast to traditional batch processors, Arroyo operates on data streams of varying sizes, enabling the immediate generation of results as soon as they are available and pipelines are defined using SQL. Also, it provides a web based user interface for creating pipelines, monitoring and configuration.

Finally, in the third category we discuss custom solutions for bridging different systems. For example (Institut für Kraftfahrzeuge, 2023) provides a bi-directional bridge between ROS1/ROS2 and MQTT. It is written in C++ and uses a YAML-based configuration file for defining topic mappings between source and destination systems. A similar solution is (GROOVE X, 2016). In (Lourenco et al., 2021), authors present a solution that enables the communication between Apache Kafka and ROS2 using their custom bridge code. The authors report the number of messages they produced and the number of messages stored in Kafka, however they do not report the rate of messages produced. The presented solution allows bi-directional data flow, but does not support any security features of Kafka (authentication, authorization and encryption). In (Chaari et al., 2019), authors present an architecture that allows computation offloading to cloud based environments for robotics environments. Here ROS is used for communication framework between different robots, Apache Kafka is used as an intermediary buffer and as a translation layer between the cloud and robot environment, and Apache Storm is used for computation. The presented solution allows bi-directional data flow. The solution is ROS1 based and relies on Rosbridge (Crick et al., 2017) that allows sending to and collecting data from ROS in JSON format using Websockets. The authors evaluated their solution using message rates up to 25 messages/second (25Hz).

3 IndraFlow DESIGN

We can see that on the higher-level of the spectrum of the related works we can find generic solutions such as Apache NiFi, Apache Camel, Fluvio, etc. These are highly customizable, but also require a steep learning curve and typically lack support for ROS based use cases. On the lower part of the spectrum we can see custom solutions such as (Lourenco et al., 2021) that are tailored for a specific use case or provide bridging capabilities between two solutions (e.g., bridging between ROS and MQTT) with no or limited transformation capabilities.

The aim of IndraFlow is to provide a middle ground. The first goal of IndraFlow is to offer a generic, but still highly customizable solutions with support for cloud, IIoT and Robot based use cases. The second goal is to provide light-weight transformation capabilities for changing the structure or format of the messages between source and destination systems. Additionally, we aim to provide an a modularized structure, rather than an integrated all-in-one solution such as Fluvio. Furthermore, we use a descriptive configuration (i.e., YAML-based) such as in (Institut für Kraftfahrzeuge, 2023) and a simplified setup, thus a single instance focuses on data transfer between a single source and destination.

Figure 1 presents the IndraFlow architecture. On the left side of the architecture we observe the data source modules. Currently ROS, ROS2, MQTT and Kafka sources are supported out of the box. However, data sources are connected via modules that rely on a well-defined interface. Thus, adding new sources is a straight-forward process. Similarly, on the right side of Figure 1, the available data destinations are shown. Currently IndraFlow has modules for Kafka, RDBMS (relational databases supported via SQLAlchemy) and MQTT data destinations. The list of outputs is also extensible with new destination modules that implement a well-defined interface.

The Intermediary Storage and Processing (ISP, see in Figure 1) is a special module as it can act both as a data source and destination. Its role is to decouple other data sources and other data destinations. Next, in IndraFlow the basic data flow is represented by a span. A span is similar to an input → filter → output pipeline in Logstash. However, IndraFlow allows custom transformations and supports ROS and MQTT based data sources and destinations. Figure 2 depicts a single span denoted by N as follows (see Figure 1):

$$N = \langle S, T(t_1, ..., t_n), D \rangle \tag{1}$$

Here S represents a data source, $T(t_1, ..., t_n)$ denotes the transformation functions used in the span, and D is the destination for the data. Both source S and destination D can be any of the supported plugins or the ISP. Any number of transformations are allowed, however, currently a span is executed within a single process, thus, increasing the number of transformations will increase the latency of messages.

Next, lets consider D as a set of points. This set of points represent the available data sources and des-

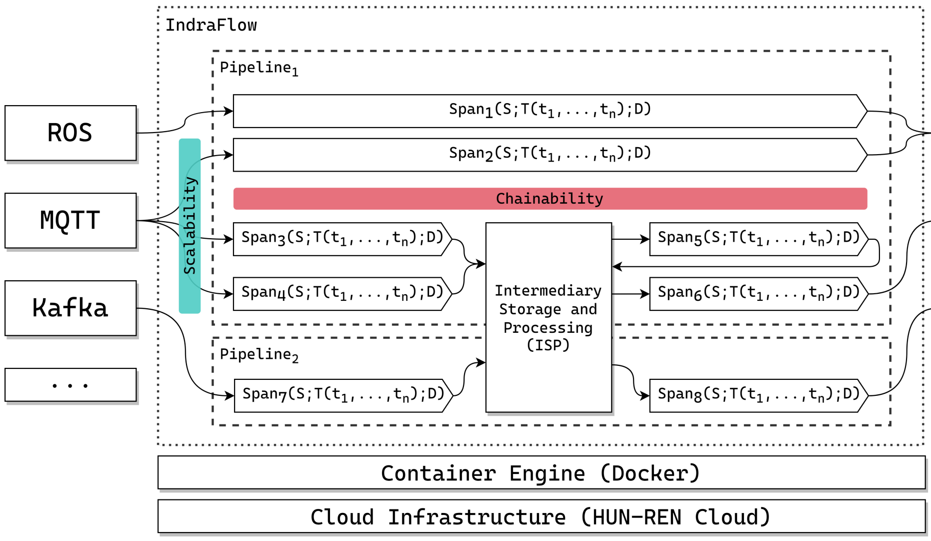


Figure 1: General architecture of IndraFlow.

tinations within an environment or use case. A span e_1 is an edge between data source $v_{s1} \in D$ and destination $v_{d1} \in D$ with the weight $w_1 = T(t_{1,1},...,t_{1,n})$. Using the notation from equation 1:

$$e_1 = \langle v_{s1}, w_1, v_{d1} \rangle \tag{2}$$

Furthermore, a pipeline P is can be constructed using arbitrary number of spans (edges) (see Figure 1):

$$P = (e_1, e_2 ..., e_n) \tag{3}$$

A pipeline denotes and end-to-end data flow, however it does not need to be continuous. Spans within a pipeline behave independently, they can be created, started and stopped as needed. They might form a path (a continuous data flow) within a directed graph (e.g., $e_1 \rightarrow e_2 \rightarrow e_3$). In this case the destination of e_1 is the data source of e_2. Additionally, each edge (span) in a pipeline has an additional weight p that denotes the multi-span configuration (see section 3), where multiple parallel instances of the same span are started to load balance the transfer of messages.

IndraFlow runs on top of a container orchestrator such as Kubernetes or Docker Compose. Currently, each span is implemented as a separate container, and pipelines are scaled and orchestrated via standardized tools such as Docker Compose and Helm (Helm, 2023). Currently, the ISP is a containerized Apache Kafka cluster. These tools allow to deploy, manage and scale IndraFlow on a component by component basis. For example in case of Apache NiFi when running multiple data flows (e.g., different use cases) on a single cluster, all nodes instantiate all components. This means that the CPU and memory footprint has always a factor that is determined by only the complexity of the data flow and cannot be reduced by scaling the cluster. The best-practice for NiFi is to separate each data flow to its dedicated cluster (Cloudera, 2023). However, this would increase the administrative burden or require further automation.

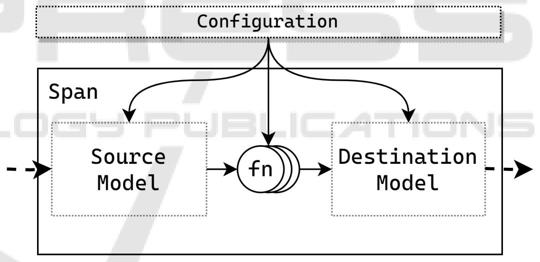


Figure 2: Structure of a single IndraFlow Span.

In the following sections we introduce two real-world use cases where IndraFlow is actively being used. The first use case involves bi-directional data flow between drones and autonomous vehicles, and cloud-based environments. Within the National Laboratory for Autonomous Systems (abbreviated as ARNL in Hungarian) and the TKP2021-NVA-01 project ("Research on Hydrogen-Powered, Cooperative Autonomous Remote Sensing Devices and Related Data Processing Framework") in Hungary IndraFlow is used for data bridging between ROS1/ROS2 based drones and autonomous vehicles (Németh and Gáspár, 2021) and a cloud-based data platform (Marosi et al., 2022). This involves multiple data flows between (i) bi-directional ROS and Apache Kafka, and (ii) ROS and TimescaleDB/PostgreSQL. Originally the flow of data was ROS $\rightarrow$

MQTT (via custom bridge code) $\rightarrow$ Apache Kafka (via Apache NiFi) $\rightarrow$ TimescaleDB (via Apache NiFi). With the help of IndraFlow this is simplified to (a) ROS $\rightarrow$ TimescaleDB/ PostgreSQL or (b) ROS $\rightarrow$ Kafka $\rightarrow$ TimescaleDB/ PostgreSQL, depending on the message rate (see section 4 for details).

The second use case is based on Industrial IoT (IIoT) data collection and enrichment in a robotic assembly scenario (Beregi et al., 2019) (Beregi et al., 2021). This scenario involves a service-oriented manufacturing execution system, in which the collaborative robotic arm near real time-data provision provides the foundation necessary for a better understanding of the entire process. In this use case, the robots used provide data on their physical operations at up to 125Hz via an MQTT interface. This data is subsequently stored in TimescaleDB (TimescaleDB, 2023), a time-series database based on PostgreSQL. Here similar Apache NiFi based data flows were used as in the previous use case, and they are being phased out in favor of IndraFlow based ones.

This use case is actively utilized within ARNL in Hungary, and is detailed in (Marosi et al., 2022). However, for the evaluation, we opted to use a publicly available dataset to support the reproducibility of our research. Therefore, we selected an environmental dataset from Kaggle (Stafford, 2020).

Furthermore, in contrast to the original use case where TimescaleDB is actively used, we chose to use the standard PostgreSQL for the evaluation. This decision was made to eliminate an external variable (TimescaleDB) from the assessment.

4 EVALUATION AND DISCUSSION OF RESULTS

In order to evaluate the scalability and reliability of our solution, numerous performance benchmarks simulating the previously described IIoT use case were performed. As detailed previously, the experiments were based on a public dataset of sensor telemetry data (Stafford, 2020). For the sake of systematic and reproducible experimentation, we developed an automated benchmarking framework capable of starting and terminating components, logging utilization, gathering results, and generating reports.

The experiments presented were performed on HUN-REN Cloud (Héder et al., 2022), a federated, community cloud based on OpenStack. We utilized the resources of a single m2.2xlarge flavoured virtual machine instance, featuring a 16-Core virtual CPU (Intel Xeon Gold 6230R) and 32 GB of memory. All components were deployed using Docker.

Three containers were launched in order to act as parts of a sensor network, and provide a stable stream of time-series data. Under the hood, the sensor containers were reading rows from the dataset and transforming them to JSON objects before forwarding them to the message broker. In order to enable the evaluation of the scalability of IndraFlow, a delay parameter was implemented, which determines the rate at which the sensors send out messages. The sensors were generating data throughout a five minute benchmark period in all presented cases. The first step in the message flow was an open source MQTT message broker, Mosquitto (Light, 2017). The main aim of this use case is to provide long-term storage for the messages received by the broker, namely to store them in a PostgreSQL (Stonebraker and Rowe, 1986) database, which serves as the final destination for the flow of data. IndraFlow interconnects the source and destination systems and handles the delivery of messages and the transformation of time-series data into records of a relational database. The executed benchmarks were scaled twofold: the delivery rates, throughput, and latency of IndraFlow were measured with message frequencies from 120Hz to up to 845Hz, and with pipelines consisting of 1 to up to 16 Spans. While in the 1-Span case all messages were handled by the single Span, in Multi-Span configurations the messages are balanced among the Spans using a shared subscription model.

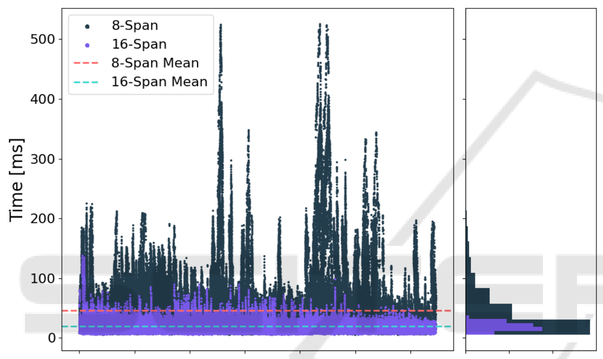
We present reliability measurements in Table 1. The first row of the table describes the destination of the messages. We performed experiments with the PostgreSQL database at increasing frequency values, and repeated with a Dummy destination at the highest frequency. The two rows below describe the parameters of the sensor network. The frequency of messages, denoted by f, is calculated as follows:

$$f = \frac{1}{\delta + \theta} \times N \tag{4}$$

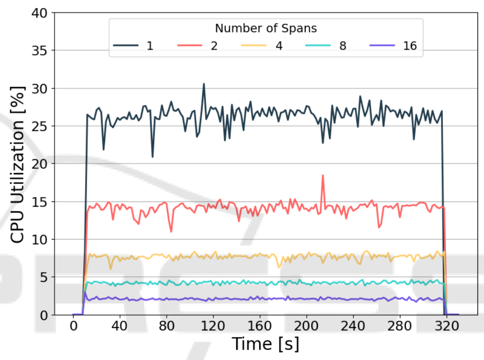
Where δ stands for the message delay, and N is the number of sensors transmitting messages. θ represents the overhead of publishing messages, which is increasingly significant as the delay is lowered: at a delay of 0.025 seconds, the sensors are able to publish messages without a noticeable overhead, and achieve a frequency of 120 Hz. In theory, halving the delay would mean twice the frequency, however, as it can be observed, due to the effect of the overhead, with a delay of 0.0125 seconds messages are published at a frequency of 235 Hz instead of the expected 240 Hz. The message delay is set up before running the benchmark, and then forwarded to the sensor containers at runtime. The rows below show the achieved results of 1- to 16-Span configurations of IndraFlow.

Table 1: Measurements of scalability and reliability focusing on the delivery rate and latency achieved by different pipeline configurations under increasing message frequency. The 845Hz configuration was also tested with a Dummy destination model in order to verify the database adapter as a source of bottleneck.

Destination		PostgreSQL				Dummy
Message	**Frequency [Hz]**	120	235	450	845	845
	Delay [s]	0.025	0.0125	0.00625	0.003125	0.003125
1-Span	**Delivery Rate**	100%	30.4%	11.7%	7.3%	100%
	Latency [ms]	11	21800	14318	14886	12
2-Span	**Delivery Rate**	100%	100%	48.9%	19.7%	100%
	Latency [ms]	8	9	14973	18508	15
4-Span	**Delivery Rate**	100%	100%	100%	39.8%	100%
	Latency [ms]	8	9	13	17392	12
8-Span	**Delivery Rate**	100%	100%	100%	100%	100%
	Latency [ms]	8	9	13	28	9
16-Span	**Delivery Rate**	100%	100%	100%	100%	100%
	Latency [ms]	8	9	13	19	9



(a) Latency of messages at 850Hz.



(b) CPU Utilization of a single Span at 120Hz.

Figure 3: Latency and CPU utilization measurements with different configurations.

The delivery rate represents the percentage of messages successfully stored. The latency, measured in milliseconds, shows the amount of time between the sensors publishing a message, and it being stored by the PostgreSQL database. The presented values were taken as the median of three 5-minute benchmark executions. In case of the latency, the median value for all messages sent is first selected for each run. As it can be seen, an IndraFlow pipeline consisting of just a single Span handles a 120 Hz message flow without errors, however becomes highly unreliable beyond this point, storing smaller and smaller fractions of total messages sent as the frequency increases. It can also be observed that the message loss is accompanied by latency values that are several orders of magnitude larger. The measurements show that scaling the pipelines, namely increasing the number of Spans improves the reliability of the solution, enabling low-latency, lossless message delivery even for data streams with significantly higher frequency. IndraFlow is able to achieve lossless message delivery at the last measurement point, 845 messages per second, while utilizing 8 or 16 Spans.

We found that the main problem occurring in high-frequency benchmarks is that due to a bottleneck in the destination model, the employed Spans are simply too busy with storing data, and cannot keep up with the velocity of messages. In the current iteration of IndraFlow, messages are handled one-by-one, meaning that the database adapters in the RDBMS model were forced to perform many costly single insertions, which becomes a limiting factor at higher frequencies. In order to verify this, additional benchmarks were executed with a Dummy destination model, which simply drops messages after the Span forwards them for storage. The results in the last column of the table demonstrate that without the bottleneck of the database adapters, even a single Span is able to receive all messages at a frequency of 845Hz.

Figure 3a shows the latency of messages measured over the duration of a benchmark conducted with 845 messages sent per second. While it was shown previously that both 8- and 16-Span configurations managed to deliver all the messages from source to destination at this frequency, it can be observed on the figure that the 16-Span pipeline achieved this with significantly lower latency. The fastest message took 5 milliseconds in both cases, while to slowest took 526 milliseconds using 8 Spans, and 138 milliseconds using 16. The mean latency for 8- and 16-Span pipelines were 45.66 and 19.18 milliseconds respectfully. Based on the histogram on the right side of the plot, we can conclude that while major spikes in latency occur (especially in the 8-Span case), most of the messages are delivered with a latency of less then 100 milliseconds. These results nicely demonstrate why scaling even beyond configurations achieving a 100% delivery rate might be beneficial in use-cases with time-critical systems.

Lastly, tendencies in the CPU utilization of IndraFlow were also examined. Measurements of the CPU utilization percentage of a single Span in different configurations are presented on Figure 3b. As indicated by the results, scaling the data pipelines to multiple Spans can be a major factor in not only reliability and performance, but also in the efficient utilization of computational resources. As doubling the number of utilized Spans nearly halves the CPU utilization per Span, scaled IndraFlow pipelines can be considered as beneficial options in use cases where the available processing power in individual devices might be limited (e.g., Edge and Fog computing).

5 CONCLUSIONS AND FUTURE WORK

In this paper, we presented our generic data bridge, IndraFlow. It features a modular architecture, supporting currently different protocols and systems such as MQTT, ROS1, ROS2, Kafka and various RDBMS. Furthermore, it supports custom message transformations and multiple sources and destinations within a single bridge instance. Our evaluation has shown that the scalability of our solution enables it to provide reliable and and low latency transmission at high message frequencies. Additionally, it was demonstrated that increasing the number of Spans is beneficial both in terms latency and the resource utilization of single Spans, making IndraFlow a promising option in time- and resource-critical systems.

For future we would like to focus on enhancing the orchestration of spans, particularly for use cases requiring multiple spans and more intricate workflows (graphs), and enabling components of spans (source, transform, destination) to run on Function-as-a-Service (FaaS) platforms such as AWS Lambda or OpenFaaS (OpenFaaS, 2023).

ACKNOWLEDGEMENTS

The research was partially supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Autonomous Systems National Laboratory Program. Project no. TKP2021-NVA-01 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme, as well as under OTKA Grant Agreement No. K 132838. On behalf of projects ”ARNL: GPU-enabled Cloud-based Big Data/AI Research Platform” and ”Research on Hydrogen-Powered, Cooperative Autonomous Remote Sensing Devices and Related Data Processing Framework” we thank for the usage of HUN-REN Cloud (Héder et al., 2022) (https://science-cloud.hu/) that significantly helped us achieving the results published in this paper.

REFERENCES

Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., and Whittle, S. (2015). The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803.

Arroyo (2023). Arroyo: Cloud-native stream processing. https://www.arroyo.dev/. Accessed: 2023-12-01.

Beregi, R., Pedone, G., Háy, B., and Váncza, J. (2021). Manufacturing execution system integration through the standardization of a common service model for cyber-physical production systems. *Applied Sciences*, 11(16).

Beregi, R., Pedone, G., and Mezgár, I. (2019). A novel fluid architecture for cyber-physical production systems. *International Journal of Computer Integrated Manufacturing*, 32(4-5):340–351.

Brewer, E. A. (2015). Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC ’15, page 167, New York, NY, USA. Association for Computing Machinery.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, 38(4).

Chaari, R., Cheikhrouhou, O., Koubaa, A., Youssef, H., and Hmam, H. (2019). Towards a Distributed Computation Offloading Architecture for Cloud Robotics. In

2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC), pages 434–441, Tangier, Morocco. IEEE.

Cloudera (2023). Cloudera dataflow for the public cloud (cdf-pc) datasheet. https://www.cloudera.com/content/dam/www/marketing/resources/datasheets/cloudera-dataflow-datasheet.pdf. Accessed: 2023-12-26.

Crick, C., Jay, G., Osentoski, S., Pitzer, B., and Jenkins, O. C. (2017). Rosbridge: Ros for non-ros users. In *Robotics Research: The 15th International Symposium ISRR*, pages 493–504. Springer.

Curry, E. (2004). Message-oriented middleware. *Middleware for communications*, pages 1–28.

Elastic (2023). Elastic stack: Elasticsearch, kibana, beats & logstash. https://www.elastic.co/elastic-stack/. Accessed: 2023-12-27.

Etcd (2023). Etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/. Accessed: 2023-12-26.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131.

Fluvio (2023). Fluvio: The programable data streaming platform. https://www.fluvio.io/. Accessed: 2023-12-01.

Google (2023). Google Cloud: Apigee API Management. https://cloud.google.com/apigee. Accessed: 2023-12-31.

GROOVE X, I. (2016). mqtt_bridge: functionality to bridge between ROS and MQTT in bidirectional. https://github.com/groove-x/mqtt_bridge. Accessed: 2023-12-31.

Héder, M., Rigó, E., Medgyesi, D., Lovas, R., Tenczer, S., Farkas, A., Emődi, M. B., Kadlecsik, J., and Kacsuk, P. (2022). The past, present and future of the elkh cloud. *INFORMÁCIÓS TÁRSADALOM: TÁRSADALOMTUDOMÁNYI FOLYÓIRAT*, 22(2):128–137.

Helm (2023). Helm: The package manager for kubernetes. https://helm.sh/. Accessed: 2023-12-31.

Ibsen, C. and Anstey, J. (2018). *Camel in action*. Simon and Schuster.

Institut für Kraftfahrzeuge, RWTH Aachen, i. (2023). MQTT_client: ROS / ROS 2 C++ Node for bi-directionally bridging messages between ROS and MQTT. https://github.com/ika-rwth-aachen/mqtt_client. Accessed: 2023-12-31.

Kreps, J. (2014). Questioning the lambda architecture. https://www.oreilly.com/radar/questioning-the-lambda-architecture/. Accessed: 2023-12-31.

Kreps, J., Narkhede, N., Rao, J., et al. (2011). Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece.

Light, R. A. (2017). Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, 2(13):265.

Lourenco, L. L., Oliveira, G., Mea Plentz, P. D., and Roning, J. (2021). Achieving reliable communication between kafka and ROS through bridge codes. In *2021 20th International Conference on Advanced Robotics (ICAR)*, pages 324–329, Ljubljana, Slovenia. IEEE.

Marosi, A. C., Emődi, M., Farkas, A., Lovas, R., Beregi, R., Pedone, G., Németh, B., and Gáspár, P. (2022). Toward reference architectures: A cloud-agnostic data analytics platform empowering autonomous systems. *IEEE Access*, 10:60658–60673.

Marz, N. (2011). How to beat the cap theorem. http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html. Accessed: 2023-12-31.

McSherry, F., Murray, D., Isaacs, R., and Isard, M. (2013). Differential dataflow. In *Proceedings of CIDR 2013*. Proceedings of cidr 2013 edition.

MiNiFi, A. N. (2023). Apache nifi minifi:a subproject of apache nifi to collect data from the point of origin. https://nifi.apache.org/minifi/. Accessed: 2023-12-01.

MuleSoft (2023). MuleSoft Anypoint Platform: Enterprise Hybrid Integration Platform. https://www.mulesoft.com/platform/enterprise-integration. Accessed: 2023-12-31.

Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. (2013). Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, Farminton Pennsylvania. ACM.

Németh, B. and Gáspár, P. (2021). The design of performance guaranteed autonomous vehicle control for optimal motion in unsignalized intersections. *Applied Sciences*, 11(8).

OpenFaaS (2023). OpenFaaS: Serverless Functions, Made Simple. https://www.openfaas.com/. Accessed: 2023-12-31.

Pulsar (2023). Apache Pulsar: Cloud-Native, Distributed Messaging and Streaming. https://pulsar.apache.org/. Accessed: 2023-12-31.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A. Y., et al. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan.

Stafford, G. (2020). Environmental sensor telemetry data. https://www.kaggle.com/datasets/garystafford/environmental-sensor-data-132k. Accessed: 2023-12-01.

Stonebraker, M. and Rowe, L. A. (1986). The design of postgres. *ACM Sigmod Record*, 15(2):340–355.

TimescaleDB (2023). TimescaleDB: Time-series data simplified. https://www.timescale.com. Accessed: 2023-12-31.

Wang, G., Chen, L., Dikshit, A., Gustafson, J., Chen, B., Sax, M. J., Roesler, J., Blee-Goldman, S., Cadonna, B., Mehta, A., Madan, V., and Rao, J. (2021). Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2602–2613, New York, NY, USA. Association for Computing Machinery.