MoDD: A Model-Driven Framework For Data Collection In Drone-Based Systems

Manele Ait Habouche, Mickaël Kerboeuf, Goulven Guillou, Jean-Philippe Babau

MoDD : A Model-Driven Framework For Data Collection In Drone-Based Systems

Manele Ait Habouche*, Mickaël Kerboeuf†, Goulven Guillou‡ and Jean-Philippe Babau§
Université de Bretagne Occidentale, Brest, France
{*Manele.AitHabouche, †Mickael.Kerboeuf, ‡Goulven.Guillou, §Jean-Philippe.Babau}@univ-brest.fr

***Abstract*—Nowadays, Cyber-Physical Systems (CPS), particularly drones, play a pivotal role in environmental research. Scientists depend on these platforms to monitor various sensor data and ensure comprehensive data archiving. However, despite their advantages, researchers encounter several challenges, including communication limitations and the complexity of setting up systems tailored to their needs. To address these issues, we propose MoDD, a model-driven data collection framework based on a customized publish/subscribe model. MoDD simplifies the development and configuration of data collection systems. It offers scientists a solution that meets their specific needs, allowing them to focus on high-level requirements while the framework manages the underlying complexities. We demonstrate the effectiveness of MoDD through practical evaluations on an actual Unmanned Surface Vehicle. Additionally, results show a 79% reduction in throughput (drone to base station link) compared to existing publish/subscribe systems.**

***Index Terms*—model-driven engineering, drone, data communications, architecture, optimization**

I. Introduction

Drones such as Unmanned Surface Vehicles (USV) are Cyber-Physical Systems (CPS) that integrate computational processes (*e.g.*, control, processing, and communication) into physical devices [1]. Drones are gaining prominence in environmental monitoring [2]. In this context, scientific observation missions raise two main issues: vehicle control and sensor data monitoring [3]. Our focus is on the latter.

Scientists aim to monitor sensor measurements at various frequencies to get an overall understanding of the environment throughout the mission. In our context, we consider sensors that produce numerical values, either in simple or hierarchical formats, such as navigation data or physical parameters. This allows them to guide the mission better, *e.g.*, by adjusting the drone's trajectory to investigate a specific area of interest. The monitoring process is not critical and operates on a communication link (data link) distinct from drone control to ensure uninterrupted navigation [4], [1]. Thus, disruptions and delays in monitoring communications are tolerated, provided that all acquired data are stored on-board. Besides, due to battery limitations, efficient energy management is crucial [4]. Therefore, data monitoring must be carried out without straining the network or the drone's resources, specifically by minimizing data throughput.

The publish/subscribe model has proved to be an effective solution in data transmission due to its inherent decoupling characteristics [5]. Indeed, it is widely used in robotics [6], [7], [8], IoT [9], [10], [11], and Wireless Sensor Networks (WSN) [12], [13], [14] as it ensures that data producers (sensors) and consumers operate independently. This level of separation provides the flexibility to add, remove, or change sensors and data consumers with minimal impact on the rest of the system.

In a resource-constrained context, scientists need to monitor sensor data without overloading the network. To address this, it is essential to minimize communication overhead between the drone and consumers, while also ensuring persistent data storage. This is achieved by reducing the frequency of communications and enabling data aggregation. Additionally, scientists face challenges in developing and integrating the necessary software for different missions, underscoring the need for a flexible approach.

Existing publish/subscribe models like ROS [6] support message filtering to control how often data is received. However, this filtering only occurs on the receiving end, meaning data is still transmitted between the drone and subscribers at the publisher's frequency. ROS provides a mechanism to limit bandwidth usage (topics_throttle), but this involves discarding packets to meet the throttle rate, which is not suitable for scientists who need to preserve all data. Some industrial tools like Kafka [15] provide flow control mechanisms, such as controlling message size and throughput and limiting the number of unacknowledged messages. However, it requires nodes to actively request data since it relies on a pull-based system. This continuous polling can consume significant network bandwidth, which might not be optimal in resource-constrained environments. Hence, optimizing a publish/subscribe system for each mission proves challenging. Moreover, developing a data collection solution in this environment requires expertise across various technologies (embedded programming, communication protocols, and performance optimization). The data collection system needs to be easily configured and adapted to suit both the platform's characteristics and the clients' data monitoring requirements, as well as to support sensor integration. Employing Model-Driven Engineering (MDE) holds promise in simplifying the complex development process. By automating code generation from high-level models [16], [17], [18], MDE can help build configurable and optimized systems.

To summarize, it is clear that end users require a simple approach to develop a data collection solution that is both optimized and easy to set up for each mission.

In this paper, we introduce MoDD, a model-driven data collection framework based on a customized publish/subscribe

2376-9521/24/$31.00 ©2024 IEEE
DOI 10.1109/SEAA64295.2024.00013

architecture. MoDD's objectives are twofold: (1) to provide scientists with a straightforward development process and (2) to generate optimized configurations tailored to scientists' requirements. To tackle the first challenge, we rely on MDE to simplify the configuration of MoDD. This flexibility is crucial as it allows the system to adapt to varying mission requirements, allowing scientists to focus on their research activities. To address the second challenge, we propose a publish/subscribe architecture that allows subscribers to specify their data ingestion frequency while maintaining the publisher's frequency, thus minimizing throughput. This feature is handled at the broker rather than the application level.

The remainder of this paper is organized as follows. Key concepts are introduced in Section II. Section III describes MoDD's architecture. Section IV details MoDD's metamodel and the proposed domain-specific language. The approach is validated in Section V. Section VI presents related work, and Section VII concludes and opens some perspectives.

II. Background

A. Publish/subscribe model

The publish/subscribe messaging model [5] is a communication paradigm where entities, referred to as *subscribers*, express their interest in consuming specific information. This information is produced by *publishers*. The strength of this model is its ability to completely decouple publishers and subscribers in terms of space and synchronization [5], [19]. Participants communicate in a non-blocking manner through a *broker*, a component responsible for routing messages from publishers to the appropriate subscribers.

There are four variants of the pub/sub scheme [20], namely channel-based, content-based, type-based, and topic-based. In the more general-purpose channel-based scheme, publishing an event implies its broadcasting to all subscribers without filtering. The topic-based scheme introduces the concept of *topics*, which represent the individual subjects to which participants can subscribe or publish. In MoDD context, the use of topics is sufficient to organize the various sensor parameters without the need for additional filtering criteria. Therefore, the content-based and type-based schemes would not be relevant in this case. Moreover, the topic-based model is effective for message routing, which meets resource constraints. Additionally, it is implementation-independent, unlike the type-based model.

B. Double buffering

Double buffering is a technique that involves using two buffers to parallelize read and write operations. The first buffer, often called *back buffer*, is used to store new data, while the other one (*front buffer*) is being processed. Once the write operation is complete and the first buffer is full, the roles of the two buffers are swapped. This technique is particularly useful in scenarios where there is a continuous flow of data being produced and consumed at different rates, as it helps in maintaining smooth processing.

III. MoDD Architecture

MoDD complies with the usual scheme of topic-based publish/subscribe. In our context, scientists may track raw data at the frequency set by sensors. However, in most cases, aggregated data (such as average, min, or max) at a reduced frequency are sufficient for continuous monitoring. Therefore, within the MoDD approach, topics encompass both raw data sourced from sensors and aggregated data.

The overall MoDD architecture, illustrated in Figure 1, can be broken down into two components: an on-board segment and a base station. The on-board segment consists of a drone equipped with sensors. Sensors produce measurements at regular intervals and publish them to a *PubBroker* deployed on the on-board computer. The PubBroker stores these data locally and relays them to the subscribed end users at the base station through the *SubBroker*. In this paper, the terms subscriber and subscription are employed interchangeably to denote a conceptual end user subscription rather than a physical entity.

MoDD's architecture supports data aggregation and user-specified data ingestion rates (best effort policy) distinct from publishers' frequencies. This is achieved through the use of double buffering to better manage the periodic backups and guarantee that end users can maintain uninterrupted access to the measurements, even if they are not the most recent. This design prevents the need for continuous transmission of all data to subscribers and reduces data throughput between the drone and the base station. In the following, we provide more details on each element of the architecture.

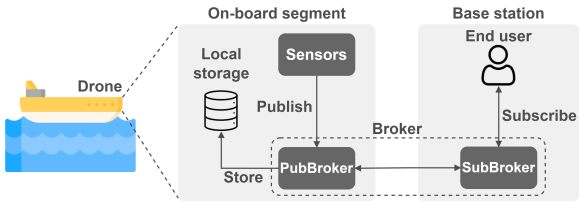

Fig. 1: MoDD architecture

A. MoDD components

1) Sensors: Sensors act as publishers, periodically sending measurements (raw data). For instance, an accelerometer that measures acceleration along three axes (x, y, z) disseminates its readings to three distinct topics, *e.g.*, accel_x, accel_y, and accel_z. Each measurement is sent by the sensor driver to the PubBroker using the publish() primitive (Figure 2). This will be further detailed in Section IV-C.

2) End users: End users play the role of subscribers, registering their interest in specific topics, *i.e.*, raw or aggregated data they wish to monitor. These subscriptions are characterized by a period and an aggregate function. The period determines the frequency at which subscribers receive updates; given that sensors can have high sampling rates, subscribers may opt for less frequent updates. However, it is important to mention that this frequency is achieved *at best*, but is not guaranteed due to factors such as network latency and the

capacity of the double buffer. Additionally, subscribers can process the incoming data stream by specifying an aggregate function such as the average, providing a more synthesized view rather than raw sensor readings. This function can also be the identity function.

3) PubBroker: Upon receiving the published measurements, the PubBroker temporarily stores them. To prevent any delays for end users caused by periodic backups, for each measurement, a double buffer of size n_i is allocated (Figure 2). As such, each received measurement is added to its respective double buffer, and once it reaches capacity, its content is saved to the disk. The measurements are then either directly transmitted (case of raw data topic) or undergo sampling and/or aggregation before transmission (case of aggregated data topic). In both cases, the transmission follows the computed message emission schedule described in Section III-C. The messages are then sent to the SubBroker, which in turn delivers them to the subscribed end users.

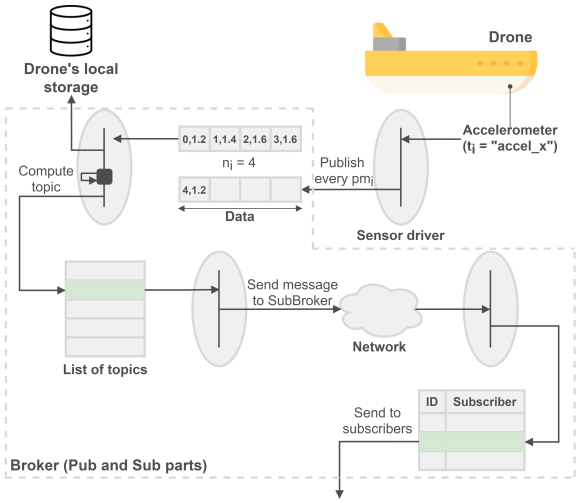

Fig. 2: Broker architecture

4) SubBroker: The SubBroker's main purpose is to reduce the PubBroker's workload and preserve the resources of the drone by allowing the PubBroker to communicate with a single node, *i.e.*, the SubBroker, rather than multiple subscribers. It is tasked with managing subscribers and ensuring the efficient delivery of topics to the relevant subscribers. To enhance communication efficiency, the SubBroker only processes the data relevant to the subscribers' needs.

B. Communication between the PubBroker and the SubBroker

1) Static part: From the information provided by publishers and subscribers, a timed sequence of message emissions is computed. The goal is to minimize throughput and merge measurements into one single message whenever possible. The generation of this schedule is detailed in Section III-C. Both the PubBroker and SubBroker adhere to the same schedule. It is worth mentioning that the PubBroker does not have access to subscribers' details. The task of identifying the subscribers meant to receive the measurements falls on the SubBroker.

The communication begins with the SubBroker initiating a connection with the PubBroker. Once connected, a ping message is sent from the SubBroker to the PubBroker. This ping determines the timeout period needed to compute buffer sizes in Section III-D. Following that, the list of required topics and the computed schedule are sent to the PubBroker.

2) Dynamic part: After a topic communication schedule has been established, subscribers can request measurements dynamically. However, the SubBroker must first assess its ability to fulfill these requests, as the data related to these dynamic subscribers were not considered during the computation of the optimized message emission schedule. The dynamic subscription process begins when a subscriber sends a request. This request includes the topic t to monitor, the subscription period ps, and the chosen aggregation function f. Subsequently, the SubBroker determines if the specified period and aggregation function can be accommodated within the current schedule. This assessment involves verifying if the requested period ps is a multiple of the existing one and if the aggregation function f can be computed from the received data. For instance, deriving a minimum value from all the data is acceptable, but extracting data solely from minimal values is not. If the request is feasible, the subscriber's ID is added to the list of targeted subscribers. Otherwise, a NOT_AVAILABLE message is sent back to the inquiring subscriber.

C. Communication optimization

The PubBroker transmits data to the SubBroker following a pre-computed schedule (a timed sequence of messages), minimizing communication overhead between the SubBroker and the PubBroker. The schedule is designed to group topics whenever possible. In Algorithm 1, the first step is to compute the *hyperperiod*, defined as the least common multiple of all subscribers' periods. The hyperperiod represents the basic cycle for sending messages, repeated periodically. Next, the algorithm identifies the dates within the hyperperiod that align with each subscriber's active period. It then schedules the transmission of their corresponding topics accordingly. The resulting schedule is a sequence of lists containing (topic, aggregation) tuples, each associated with a specific date.

The PubBroker reads the measurements published by sensors and assigns them to the appropriate double buffer. It also updates aggregation values as necessary, thus optimizing the number of messages sent according to the topics needed. Concurrently, the PubBroker processes the schedule, batching topics linked to a specific date into a single message before sending it to the SubBroker. The SubBroker then forwards these topics to the appropriate subscribers.

D. Buffer optimization

The buffer size n_i is a parameter that needs to be determined for each topic i. Its value is influenced by the periods at which measurements are produced, *i.e.*, pm_i, and by the subscription period of each subscriber j, denoted by ps_j. Our goal is to

Algorithm 1: Compute message emission schedule

Input: subPeriods: Subscriber period list,
subTopics: Subscriber topic list,
subAggregs: Subscriber aggregation list,
J: Number of subscribers
Output: schedule: Array of list of (topic, aggregation),
H: Hyperperiod of the schedule

```
H = lcm(subPeriods)
schedule[1 to H] = NULL
for j = 1 to J do
    time = subPeriods[j]
    while time ≤ H do
        schedule[time].add(subTopics[j], subAggregs[j])
        time = time + subPeriods[j]
```

determine the optimal buffer sizes for every topic in order to minimize the delay experienced by the subscribers. This delay is defined as the time difference between data generation and subscriber reception, estimated as $|ps_j - n_i \times pm_i|$ for each subscriber j associated with topic i. This problem is formulated as a simple Mixed-Integer Linear Program (MILP). The collective delay across all subscribers is defined as the sum of the absolute delays between the periods of the subscribers j and the time required to fill each buffer i. Due to the non-linearity of the absolute value, a continuous decision variable is introduced $z_{j,i}$ as shown in Equations (2–3). $z_{j,i}$ is a continuous variable that represents the absolute delay experienced by subscriber j associated with topic i.

Minimizing this sum does not guarantee the minimization of the delay for each individual subscriber. Hence, an additional variable w is introduced, representing the greatest delay any subscriber might encounter. The objective becomes twofold: to minimize w and to ensure the delay $z_{j,i}$ for each subscriber does not exceed w. The two objectives are combined in Equation (1) using a weighted sum, where α and β act as weights that determine the relative importance of the cumulative delay and the maximum delay. I represents the number of topics and $S(i)$ is the set of subscribers associated with topic i.

In Equation (5) and Equation (7), the buffer sizes n_i are constrained by MAX_BUFFER, and the delays $z_{j,i}$ must not exceed the timeout period, which is evaluated as three times the ping time computed in Section III-B1. This problem is solved using the CBC solver.

$$\text{Minimize:} \quad \alpha \times \sum_{i=1}^{I} \sum_{j \in \mathrm{S}(i)} z_{j,i} + \beta \times w \tag{1}$$

$$z_{j,i} \geq ps_j - n_i \times pm_i \tag{2}$$

$$z_{j,i} \geq n_i \times pm_i - ps_j \tag{3}$$

$$z_{j,i} \leq w \tag{4}$$

$$n_i \leq \text{MAX\_BUFFER} \tag{5}$$

$$n_i \geq 1 \tag{6}$$

$$z_{j,i} \leq 3 \times \text{PING\_TIME} \tag{7}$$

IV. MoDD Metamodel And Configuration

A. Process overview

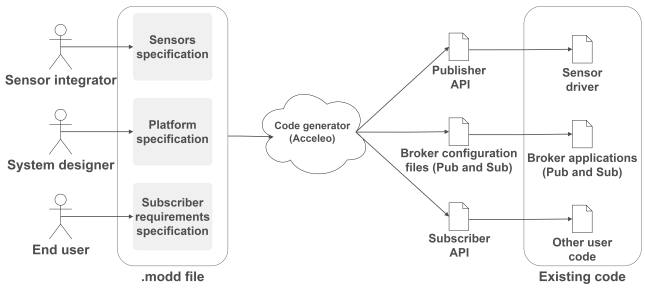

Fig. 3: MoDD configuration process overview

One of our main objectives is to provide a data collection system that effortlessly integrates various heterogeneous sensors. Deploying such a system involves three key roles: (1) the system designer, responsible for integrating all embedded software; (2) the end user, who monitors parameters during a mission; and (3) the sensor integrator, responsible for managing sensors and writing and deploying their specific drivers.

As depicted in Figure 3, inputs from all involved roles are used to configure the data collection system to meet the specific requirements of each mission. This is done through a common description (a .modd file) using a Domain-Specific Language (DSL) based on a metamodel. Both the DSL and the metamodel are detailed in Section IV-B.

As usual with generative models, the MoDD specification is used to generate specific code. Three categories of code are generated for different purposes. First, for sensor data publication, the sensor integrator needs to integrate existing sensor drivers with MoDD architecture. Second, since the applications of the PubBroker and SubBroker are already provided by MoDD, configuration files are generated. Last, for end users, an API is generated for integration into a client viewer or analysis tool. Further details on the code generation process are provided in Section IV-C.

B. MoDD metamodel and Domain-Specific Language

The MoDD metamodel, depicted in Figure 4, is structured into three parts, corresponding to the three roles described in Section IV-A. The first part describes sensors and topics, the second part provides details on the platform, and the last part outlines the subscribers' requirements.

A MoDD data collection system features a Drone that hosts a PubBroker and several Publishers. The drone is equipped with various Sensors. Each sensor captures a set of measurable physical quantities or parameters referred to as MeasuredVariables. These measured variables represent the topics to which publishers disseminate data every period seconds. The PubBroker saves the published measurements at a location determined by the logPath attribute. On the other

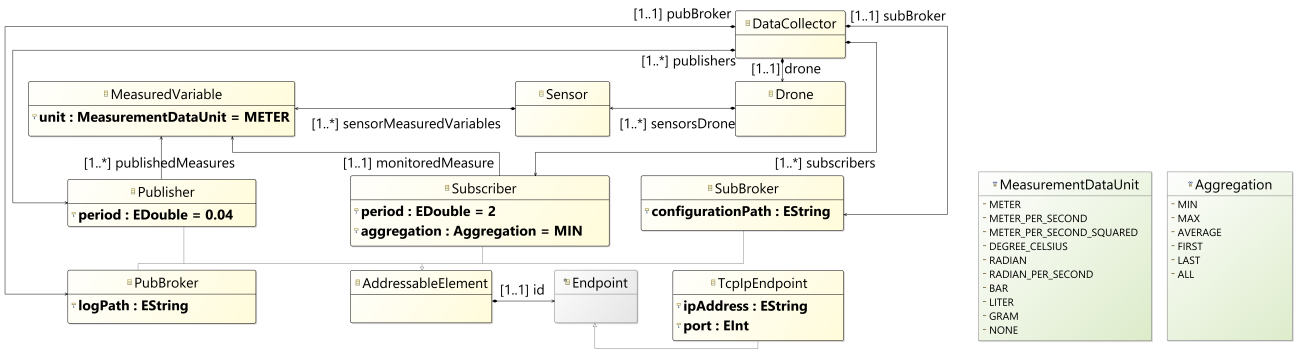

Fig. 4: MoDD metamodel

hand, the base station consists of a SubBroker and several Subscribers. The generated configuration files, as described in Section IV-C, are stored at a location defined by the configurationPath attribute. Subscribers are characterized by a subscription period and the type of aggregation they require for their data. The PubBroker, SubBroker, Publishers, and Subscribers are addressable and can be accessed through an Endpoint. In our context, an endpoint refers to TCP/IP identifiers that include both an ipAddress and a port number.

This metamodel enables the various stakeholders to model their needs. To make this process more user-friendly, a textual grammar is derived from MoDD metamodel using Xtext. This DSL features an intuitive textual syntax that simplifies the description of model elements.

An example is provided in Listing 1. The language defines multiple blocks. The Drone block allows for the specification of the equipped Sensors, with each sensor defining its associated topics. For instance, the drone named usv has an accelerometer that measures acceleration along the x-axis. These measurements are recorded under the topic accel_x and expressed in $\mathrm{m\,s^{-2}}$. The PubBroker and SubBroker blocks set up endpoints for their respective components. For instance, the SubBroker is reachable at 192.169.1.163:7272. Additionally, the PubBroker block specifies the directory path for logging measurements (/path/to/logs), while the SubBroker specifies the storage path for the generated configuration files (/path/to/conf). In the Publishers block, each publisher specifies the topics they cover and the frequency of their publications. For instance, pub_1 disseminates x-axis accelerometer data on topic accel_x every 0.04 s. Finally, the Subscribers block details the features of each subscription, namely, the subscription topic, the subscription period given in seconds, and the aggregate function. Here, sub_1 fetches the average (AVERAGE) x-axis acceleration every 2 s.

```
1  Drone usv {
2      Sensors {
3          Sensor accelerometer measures {
4              accel_x(METER_PER_SECOND_SQUARED);
5          }
6      }
7  }
8  PubBroker("192.168.1.128":7171, "/path/to/logs");
9  SubBroker("192.168.1.163":7272, "/path/to/conf");
10
11 Publishers {
12     Publisher pub_1("192.168.1.128":7373, 0.04, "usv.accelerometer.
           ↪ accel_x");
13 }
14 Subscriptions {
15     Subscriber sub_1("192.168.1.173":7474, "usv.accelerometer.accel_x
           ↪ ", 2.0, AVERAGE);
16 }
```

Listing 1: Example of a *.modd* file

C. Code generation

The code generation process leverages Acceleo, a template-based engine designed for Model-To-Text (M2T) transformations.

For broker configuration, four configuration files are generated from the .modd instance. These files contain endpoint details for both brokers, the PubBroker's log path, and information about topics and subscribers stored in the SubBroker's designated configuration path. The SubBroker uses this information to determine double buffer sizes and compute the message emission schedule. Data are then transmitted to the PubBroker.

For sensor driver integration, a C++ API designed for data publication is generated. This API includes a Publisher class that implements the publish(topic, timestamp, value) primitive, and provides an initial code skeleton that initializes a Publisher instance. The sensor integrator simply needs to insert a publish call in the data acquisition code, typically right after new sensor data is received. This is illustrated in Listing 2, which represents a code snippet from the generated C++ skeleton integrated with an SBG driver that collects IMU data. The Publisher instance is created in line 3, enabling pub_1 to connect to the PubBroker using the endpoint specified in the .modd file. The sensor integrator then effortlessly adds the publish primitive where appropriate. In the example, the publish method is invoked in line 10 to disseminate z-axis acceleration data to topic accel_z.

Similarly, a ready-to-use C++ API is generated to set up each subscriber instance with the characteristics defined in the .modd file.

```
#include "Publisher.h"
// Initialize a publisher instance
std::unique_ptr<Publisher>publisher =
      std::make_unique<Publisher>("pub_1");

SbgErrorCode on_log_received(SbgEComHandle* p_handle,
      ↪ SbgEComClass message_class, SbgEComMsgId message,
      ↪ const SbgBinaryLogData* p_log_data, void* p_user_arg) {
    // Any preliminary code
    switch (message) {
        case SBG_ECOM_LOG_IMU_DATA: {
            // Publish measurements to the right topic
            publisher->publish("accel_z",
      p_log_data->imuData.timeStamp,
      p_log_data->imuData.accelerometers[2]);
            break;
        }
    }
    return SBG_NO_ERROR;
}
```

Listing 2: Code snippet illustrating data publication

V. Evaluation

A. Experiment

To evaluate MoDD[1], an experiment was conducted in a pool measuring $50\,\mathrm{m} \times 12.5\,\mathrm{m}$, with a depth ranging from $10\,\mathrm{m}$ to $20\,\mathrm{m}$.

The experiment involved a USV equipped with various sensors, including a GNSS (Global Navigation Satellite System) module with RTK (Real-Time Kinematic) positioning for location tracking, and an Ellipse INS (Inertial Navigation System) for orientation and navigation. Communication between the drone and the base station was achieved through two links: a control link (RC transmitter) and a long-range Wi-Fi data link.

Fig. 5: USV facing waves

The evaluation is based on two scenarios. In the first one, a publisher disseminates data across three topics to seven subscribers. Details about subscription periods and topics are presented in Table I. The data aggregations requested by each subscriber are: average for subscribers 1 to 4, minimum for subscriber 5, and maximum for subscribers 6 and 7. The results of the MILP for setting up double buffer sizes n_i are given in the last row. The constants for the problem formulation are as follows: MAX_BUFFER is set to 1000, the obtained PING_TIME is $0.83\,\mathrm{s}$, and both coefficients α and β are set to 1. For topic accel_z, a size of 6 is adequate for subscribers 5 to 7 since they have the same subscription period ps_j. In the case of topics accel_x and accel_y, a size of 28 accommodates the different periods of the targeted subscribers.

[1]The GitHub repository provides the MoDD C++ API, the metamodel, the textual grammar, the Acceleo templates, and a user manual. Link: https://github.com/manele-ah/modd/

Topics	accel_x	accel_y	accel_z
Subscribers	sub_{1, 3}	sub_{2, 4}	sub_{5, 6, 7}
pm_i (s)	0.04	0.04	0.04
ps_j (s)	{2, 0.25}	{2, 0.25}	{0.25, 0.25, 0.25}
$\boldsymbol{n_i}$	**28**	**28**	**6**

TABLE I: Double buffer settings for scenario 1

In the second scenario, a publisher sends data across 19 topics every $0.04\,\mathrm{s}$. There are 19 subscribers, each subscribing to a distinct topic with a period of $1\,\mathrm{s}$. The specified data aggregation for all these subscribers is the min function. The MILP constants are the same as in the previous scenario. The resulting double buffer sizes are all set to 25 since every subscriber has the same period and subscribes to one topic.

In both scenarios, the PubBroker and publisher were deployed on the drone, while the SubBroker and subscribers were set up on a computer that communicates with the USV.

B. Code generation

We evaluate the ease and effort required to set up optimized code for each scenario. This assessment is based on the number of lines of code (LOC) as shown in Table II. To accommodate each mission's needs, the generated code consists of 275 and 287 LOC, respectively. For data dissemination, the sensor integrator adds 3 LOC in scenario 1 to publish on the three targeted topics, and 19 LOC to handle the 19 topics in scenario 2. When combined with .modd files, user LOC amount to 29 and 57, respectively, representing an average of 15% of the total generated LOC and 2% of MoDD API (2404 LOC). The added LOC are straightforward and user-centered, meeting scientists' needs for flexible software development that accommodates the various requirements of different missions.

Scenarios	Scen. 1	Scen. 2
MoDD API	2404	
.modd file	26	38
Generated files	275	287
Added lines in sensor drivers	3	19

TABLE II: Lines of code in MoDD deployment scenarios

C. Data transmission efficiency

The architecture of MoDD is designed to minimize the communication overhead between the broker and subscribers. Its effectiveness is first assessed by comparing it with three pub/sub models based on the same implementation. Then, MoDD is evaluated against three existing tools. This comparative analysis focuses on the throughput of messages exchanged between the drone and the base station. The throughput is computed using a hybrid approach: initially using Wireshark traces to determine packet sizes, and then proceeding with

analytical calculations to derive the throughput. This allows us to isolate and focus on the relevant (publish and subscribe) packets. The packet sizes include both the message size at the application layer and the headers of the TCP and IP layers.

1) Evaluation of MoDD: MoDD is compared with three publish/subscribe models: (S1) a traditional pub/sub system that does not include a SubBroker and is devoid of any mechanisms for managing subscriber frequency and data aggregation; (S2) a pub/sub system that includes a SubBroker with a message batching feature; and (S3) a pub/sub system that integrates mechanisms for managing subscriber frequency and data aggregation but does not include a SubBroker.

The results illustrated in Table III show that data publication volumes remain unchanged across all strategies since we do not intervene at that level. Instead, our focus is on the data exchanged between the drone and the base station, specifically between the broker (S1, S3)/PubBroker (S2) and subscribers (S1, S3)/SubBroker (S2). MoDD reduces the volume to 1,553.5 B/s and 1,284 B/s in scenarios 1 and 2, resulting in a decrease of approximately 96%. This efficiency is attributed to two factors. First, MoDD allows subscribers to specify their data reception frequency, therefore creating a direct correlation between the saved percentage and the subscribers' chosen frequencies, as illustrated by the 92% gain achieved with S3 in comparison to S1. Second, introducing a SubBroker enables message batching, eliminating the need to transmit duplicate information to multiple subscribers. This is highlighted by the 54% gain achieved with S2 as opposed to S1.

	Throughput (bytes/second)			
	Publisher → Broker		Drone → Base station	
	Scen. 1	Scen. 2	Scen. 1	Scen. 2
S1	12675	79825	29575	79825
S2	12675	79825	14850	32100
S3	12675	79825	3549	3193
MoDD	12675	79825	**1553.5**	**1284**

TABLE III: Data transmission volume - MoDD evaluation

2) Comparison with existing tools: In Table IV, MoDD is compared with Kafka, MQTT (Mosquitto), and ROS 2. Kafka follows the pub/sub paradigm but operates on a pull-based approach where subscribers actively pull data as needed. Subscribers send a dynamic request each time they need data, and the requested data is received through a fetch response. The publishing process also involves the same mechanism. While Kafka does not allow frequency specification, it provides parameters to set the minimum message size per subscriber and the maximum server response delay for fetch requests, which we adjusted accordingly ($\mathrm{fetch.max.wait.ms}$ and $\mathrm{fetch.min.bytes}$). MQTT relies on a push-based system and does not allow subscribers to specify a frequency. However, this protocol is widely used in IoT. ROS also operates on a push-based system and provides a mechanism ($\mathrm{TimeSequencer}$ from $\mathrm{message\_filters}$) for subscribing nodes to control the frequency at which they receive messages. However, this is handled at the subscriber level. Since ROS does not use a broker, messages are transmitted directly between publishers and subscribers. As a result, the publish volume is not reported for ROS.

Results show that Kafka's data throughput in drone-to-subscriber communication scenarios is, on average, 6.98 times greater than MoDD's. This is due to Kafka's dynamic and distributed design, which results in messages containing extensive information (partitions, offsets, etc.). Despite MQTT being a more lightweight protocol, as evidenced by its publish volume being 2.65 times lower than MoDD's, the throughput on the sensitive drone-subscriber link is still, on average, 4.43 times higher with MQTT than with MoDD. This difference arises because MQTT brokers lack mechanisms for managing subscriber frequency and data aggregation. Regarding ROS, the throughput on the drone-subscriber link is on average 11.45 times greater than MoDD's. This is because even though $\mathrm{message\_filters}$ allow subscribers to specify the frequency of message reception, this filtering occurs after the data has been transmitted. Indeed, the received packets are buffered on the subscriber side (the buffer size is also given as a parameter), and they are emitted at the desired frequency after transmission between the drone and the subscribers.

The results presented in Sections V-C1 and V-C2 underline how the approach optimizes communication between the drone and the base station, a sensitive link within the architecture that also impacts the drone's autonomy [4].

	Throughput (bytes/second)			
	Publisher → Broker		Drone → Subscribers	
	Scen. 1	Scen. 2	Scen. 1	Scen. 2
MoDD	12675	79825	**5102.5**	**4477**
Kafka	15675	77575	19410	45505
MQTT	**4800**	**29950**	11200	29950
ROS 2	-	-	28700	77450

TABLE IV: Data transmission volume - Comparison with existing tools

3) Double buffer overhead: To assess the performance impact of using double buffers, a comparative analysis is conducted between MoDD's PubBroker and MQTT broker. Two metrics are evaluated: data latency and physical memory usage. This analysis is based on averaged measurements from 10 test runs, each lasting 1 minute. Data latency is defined as the time elapsed between the publication of the first message corresponding to the first element in the double buffer, and the reception of aggregated data by subscribers. This excludes messages sent at subscribers' desired frequency but containing no new information due to ongoing writes to the double buffer. In the case of MQTT, latency is measured from message publication to subscriber delivery. In scenario 1, MoDD shows lower latency, primarily because subscribers 5 to 7 track data on a double buffer corresponding to their specified period. The standard deviation observed is due to the higher latency experienced by subscribers 1 to 4 compared to subscribers 5 to 7. In scenario 2, MoDD shows higher latency because it processes 19 topics, posing a challenging scenario for data

monitoring. However, MoDD maintains a memory footprint that is on par with MQTT. Overall, the impact of double buffers is justified by the data throughput benefits it provides compared to MQTT. This is particularly relevant since the monitoring process is not subject to real-time constraints, and end users can tolerate delays of a few seconds. For future work, it would be interesting to study the variability of the double buffer size and its impact on data latency. Our intuition suggests that larger buffer sizes increase latency, as data must accumulate in the buffer before transmission of new measurements, and vice versa.

	Data latency (s) (+/- std. dev)		Mem. usage (%) (+/- std. dev)	
	Scen. 1	Scen. 2	Scen. 1	Scen. 2
MoDD	0.87 (0.64)	3.14 (0.31)	0.08 (10^{-4})	0.09 (10^{-3})
MQTT	1.49 (0.03)	1.52 (0.03)	0.09 (10^{-5})	0.09 (10^{-5})

TABLE V: Measuring double buffer overhead

VI. Related Work

A. Publish/subscribe model

Publish/subscribe models are widely adopted in many fields, both in academia and industry. In the context of wireless sensor networks, many solutions are proposed [20]. Mires [12] is a pub/sub middleware designed to simplify the creation of WSN applications by encapsulating network protocols and providing a high-level API. Other solutions, such as TinyCOPS [21] and TinyDDS [13], rely on a content-based scheme and aim to provide interoperability across different protocols based on the requirements of application designers. PRISMA [14] is another pub/sub middleware based on REST. It supports communication and topology control services while providing QoS mechanisms to meet quality constraints.

Although these solutions offer many capabilities, they still require additional programming efforts to properly configure and optimize them. Middleware configuration is mainly aimed at developers, without directly involving end users. In contrast, MoDD enables users to describe the system in a natural language. Granted, sensors need to be integrated appropriately into the system, but development efforts are significantly reduced. Besides, these solutions are more focused on control aspects. Our goal is different. We aim to enable multiple scientists to monitor data at various frequencies, while also attempting to minimize throughput.

In IoT, pub/sub models are heavily used through MQTT [9]. MQTT is a lightweight network protocol that aims for efficient message exchange between constrained devices over low-bandwidth networks. The Object Management Group (OMG) has also integrated this model into the Data Distribution Service (DDS) [10], which defines interfaces for developing pub/sub systems with QoS management capabilities [22], [23].

For drones and robotics, the pub/sub model is mainly implemented through ROS (Robot Operating System) [6], [24], as highlighted by several works [7], [8]. ROS uses this model for inter-process communication, fostering a loosely coupled and distributed system. Additionally, many industrial messaging brokers implement the pub/sub pattern such as Kafka [15], ActiveMQ [25], and RabbitMQ [26]. Although these technologies are mature and offer a wide range of capabilities, they are not fully optimized for efficient communication between the broker and users. These systems often transmit data at the publisher's set frequency, which can lead to unnecessary transmissions. Methods to limit bandwidth typically involve discarding packets, which is not suitable for scenarios requiring complete data preservation. Some systems provide rate-limiting mechanisms but rely on a pull-based approach that requires constant data polling.

B. MDE in IoT and robotics

In [27], the authors discuss the role of MDE in simplifying complex robotics software development. The analysis reveals that the majority of studies have concentrated their methods on ground robot and robotic arm applications, with a comparatively limited application in the context of aerial vehicles, and no referenced works address USV. [16] proposes AutoIoT, a framework developed for generating IoT server-side applications. IoT scenarios are described through a metamodel, and users define their IoT systems with a JSON file. Model-to-model and model-to-text transformations are used to output the final application. [28] presents a model-based approach aimed at generating efficient APIs to handle communication between resource-constrained devices. The described method relies on ThingML models [29] to define the messages that need to be exchanged, as well as some network features such as acknowledgments, timeouts, and message retransmission. [17] introduces CyprIoT, a framework devised for modeling and controlling network-based IoT applications. It addresses the challenge of connecting diverse IoT devices and provides a networking language as well as a rule-based policy language to control the behavior of the network.

While present frameworks adeptly serve the needs of robotics and a range of IoT applications, a distinct gap remains in addressing the specific challenges associated with data collection for drones.

VII. Conclusion

The development of a data collection system for autonomous marine systems requires a comprehensive understanding of various technologies. MDE simplifies this by providing a modeling tool that allows users to overlook low-level details and focus on high-level concepts. MoDD captures user requirements for both platform and data and addresses the needs of scientists in terms of data collection. This is achieved through an architecture that optimizes data transfer, ensuring efficient monitoring and archiving of data. This is demonstrated through a case study conducted with a USV equipped with multiple sensors. The framework's ease of use was assessed in real-world scenarios and the data transmission volume showed a 79% decrease compared to existing publish/-subscribe systems.

Future research will focus on incorporating energy awareness into MoDD architecture and balancing the trade-offs between double buffer overhead and throughput. We plan to approach this as a multi-objective problem, considering energy consumption, latency, and throughput as metrics. Additionally, we aim to study the variability introduced by double buffering and its impact on data latency. Another direction for improvement would be to extend the metamodel to include details for specific targets beyond our custom publish/subscribe system, such as ROS, for the embedded part of the system (*i.e.*, PubBroker). Future work could also benefit from a comprehensive qualitative evaluation, involving collecting feedback from developers and end users through questionnaires to assess the usability of the framework.

The architecture targeted by our framework is designed for aiding scientists in data collection and analysis. For multi-drone scenarios, we could envision a more distributed framework across several nodes. Our proposal would be to distribute the broker across several nodes, but this raises issues of coordination and efficient routing. Additionally, we could propose a modular approach, where sensor integration can be handled through plug-and-play modules. However, it still requires further customization to incorporate the specifics of each sensor.

ACKNOWLEDGMENTS

This work was supported by the ISblue project, "Interdisciplinary Graduate School For The Blue Planet" (ANR-17-EURE-0015), and co-funded by a grant from the French government under the program "Investissements d'Avenir" embedded in France 2030.

REFERENCES

[1] H. Wang and H. e. a. Zhao, "Survey On Unmanned Aerial Vehicle Networks: A Cyber Physical System Perspective," *IEEE Commun. Surv. & Tutorials*, vol. 22, no. 2, pp. 1027–1070, 2020.

[2] S. K. Khaitan and J. D. e. a. McCalley, "Design Techniques And Applications Of Cyberphysical Systems: A Survey," *IEEE Syst. J.*, vol. 9, pp. 350–365, June 2015.

[3] A. Zolich and D. e. a. Palma, "Survey On Communication And Networks For Autonomous Marine Systems," *J. Intell. & Robot. Syst.*, vol. 95, pp. 789–813, Sept. 2019.

[4] Y. Zeng and R. e. a. Zhang, "Wireless Communications With Unmanned Aerial Vehicles: Opportunities And Challenges," *IEEE Commun. Magazine*, vol. 54, pp. 36–42, May 2016.

[5] P. T. Eugster and P. A. e. a. Felber, "The Many Faces Of Publish/Subscribe," *ACM Comput. Surv.*, vol. 35, pp. 114–131, June 2003.

[6] M. Quigley and K. e. a. Conley, "ROS: An Open-Source Robot Operating System," in *ICRA Workshop Open Source Softw.*, vol. 3, (Kobe, Japan), p. 5, IEEE, 2009.

[7] J. Mendoza-Chok and J. C. C. e. a. Luque, "Hybrid Control Architecture Of An Unmanned Surface Vehicle Used For Water Quality Monitoring," *IEEE Access*, vol. 10, pp. 112789–112798, 2022.

[8] L. Shi and N. J. H. e. a. Marcano, "A Review On Communication Protocols For Autonomous Unmanned Aerial Vehicles For Inspection Application," *Microprocessors And Microsystems*, vol. 86, p. 104340, Oct. 2021.

[9] A. Stanford-Clark and U. Hunkeler, "MQ Telemetry Transport (MQTT)," *Online. https://mqtt.org*, vol. 22, p. 2013, 1999.

[10] G. Pardo-Castellote, "OMG Data-Distribution Service: Architectural Overview," in *23rd Int. Conf. Distrib. Comput. Syst. Workshops, 2003. Proc..*, pp. 200–206, May 2003.

[11] A. Lazidis and K. e. a. Tsakos, "Publish–Subscribe Approaches For The IoT And The Cloud: Functional And Performance Evaluation Of Open-Source Systems," *Internet Of Things*, vol. 19, p. 100538, Aug. 2022.

[12] E. Souto and G. e. a. Guimarães, "Mires: A Publish/Subscribe Middleware For Sensor Networks," *Personal And Ubiquitous Comput.*, vol. 10, pp. 37–44, Feb. 2006.

[13] P. Boonma and J. Suzuki, "TinyDDS: An Interoperable And Configurable Publish/Subscribe Middleware For Wireless Sensor Networks," in *Wireless Technologies: Concepts, Methodologies, Tools And Appl.*, pp. 819–846, IGI Global, 2012.

[14] J. R. Silva and F. C. e. a. Delicato, "PRISMA: A Publish-Subscribe And Resource-Oriented Middleware For Wireless Sensor Networks," in *Proc. 10th Advanced Int. Conf. Telecommun., Paris, France*, vol. 2024, p. 8797, Citeseer, 2014.

[15] "Apache Kafka." https://kafka.apache.org/.

[16] T. Nepomuceno and T. e. a. Carneiro, "AutoIoT: A Framework Based On User-Driven MDE For Generating IoT Applications," in *Proc. 35th Annu. ACM Symp. Appl. Comput.*, SAC '20, (New York, USA), pp. 719–728, ACM, Mar. 2020.

[17] I. Berrouyne and M. e. a. Adda, "CyprIoT: Framework For Modelling And Controlling Network-Based IoT Applications," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, SAC '19, (New York, USA), pp. 832–841, ACM, Apr. 2019.

[18] M. Ait Habouche, M. Kerboeuf, G. Guillou, and J.-P. Babau, "FaST: An Efficient Framework For Visualizing Large-Scale Time Series," in *2022 IEEE Int. Conf. Big Data*, (Osaka, Japan), pp. 3745–3754, IEEE, Dec. 2022.

[19] C. Esposito, D. Cotroneo, and S. Russo, "On Reliability In Publish/-Subscribe Services," *Computer Networks*, vol. 57, pp. 1318–1343, Apr. 2013.

[20] T. R. Sheltami and A. A. e. a. Al-Roubaiey, "A Survey On Developing Publish/Subscribe Middleware Over Wireless Sensor/Actuator Networks," *Wireless Networks*, vol. 22, pp. 2049–2070, Aug. 2016.

[21] J.-H. Hauer and V. e. a. Handziski, "A Component Framework for Content-Based Publish/Subscribe In Sensor Networks," in *Wireless Sensor Networks* (R. Verdone, ed.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 369–385, Springer, 2008.

[22] P. Bellavista and A. e. a. Corradi, "Quality Of Service In Wide Scale Publish—Subscribe Systems," *IEEE Commun. Surv. & Tutorials*, vol. 16, no. 3, pp. 1591–1616, 2014.

[23] G. Sciangula and D. e. a. Casini, "Bounding The Data-Delivery Latency of DDS Messages In Real-Time Applications," in *35th Euromicro Conf. Real-Time Syst.* (A. V. Papadopoulos, ed.), vol. 262 of *Leibniz Int. Proc. In Informatics*, (Dagstuhl, Germany), pp. 9:1–9:26, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[24] Y. Maruyama and S. e. a. Kato, "Exploring The Performance Of ROS2," in *Proc. 13th Int. Conf. Embedded Softw.*, EMSOFT '16, (New York, NY, USA), pp. 1–10, ACM, Oct. 2016.

[25] "Apache ActiveMQ." https://activemq.apache.org/.

[26] "RabbitMQ." https://www.rabbitmq.com/.

[27] E. de Araújo Silva and E. e. a. Valentin, "A Survey Of Model Driven Engineering In Robotics," *J. Computer Languages*, vol. 62, p. 101021, 2021.

[28] F. Fleurey and B. e. a. Morin, "MDE To Manage Communications With And Between Resource-Constrained Systems," in *Model Driven Engineering Languages And Syst.* (J. Whittle, T. Clark, and T. Kühne, eds.), Lecture Notes In Computer Science, (Berlin, Heidelberg), pp. 349–363, Springer, 2011.

[29] B. Morin and N. e. a. Harrand, "Model-Based Software Engineering to Tame The IoT Jungle," *IEEE Softw.*, vol. 34, pp. 30–36, Jan. 2017.