OPC UA PubSub and Industrial Controller Redundancy

Bjarne Johansson*†, Olof Holmgren*, Martin Dahl‡,
Håkan Forsberg†, Thomas Nolte†, Alessandro V. Papadopoulos†
* ABB Process Automation, Process Control Platform, Västerås, Sweden, {bjarne.johansson, olof.holmgren}@se.abb.com
† Mälardalen University, Västerås, Sweden, {hakan.forsberg, thomas.nolte, alessandro.papadopoulos}@mdu.se
‡ ABB Process Automation, Process Control Platform, Minden, Germany, {martin.dahl}@de.abb.com

Abstract—**Industrial controllers constitute the core of numerous automation solutions. Continuous control system operation is crucial in certain sectors, where hardware duplication serves as a strategy to mitigate the risk of unexpected operational halts due to hardware failures. Standby controller redundancy is a commonly adopted strategy for process automation. This approach involves an active primary controller managing the process while a passive backup is on standby, ready to resume control should the primary fail. Typically, redundant controllers are paired with redundant networks and devices to eliminate any single points of failure. The process automation domain is on the brink of a paradigm shift towards greater interconnectivity and interoperability. OPC UA is emerging as the standard that will facilitate this shift, with OPC UA PubSub as the communication standard for cyclic real-time data exchange. Our work investigates standby redundancy using OPC UA PubSub, analyzing a system with redundant controllers and devices in publisher-subscriber roles. The analysis reveals that failovers are not subscriber-transparent without synchronized publisher states. We discuss solutions and experimentally validate an internal stack state synchronization alternative.**

I. INTRODUCTION

Automation solutions are crucial in modern society and pivotal in infrastructure for critical utility services such as power and freshwater distribution. At the core of these automation solutions is the controller, which interacts with the physical environment through Input and Output (I/O) devices. The controller processes data from input devices to assess the system's status and directs output devices to achieve desired outcomes, forming what is known as the control loop. In certain domains, such as offshore oil and gas production, halts can incur significant costs, particularly unexpected halts due to hardware failures. Hence, the reliability requirements for the components constituting the control loop are high in such domains.

A widely adopted strategy to mitigate the risk of unplanned stops caused by hardware failures is the implementation of spatial redundancy. This involves duplicating critical hardware components such as devices, communication systems, and controllers. The aim is to ensure that a single fault does not lead to a system halt, effectively eliminating what is known as a Single Point of Failure (SPoF). By having redundant components, the system can continue to operate even if one component fails, thereby enhancing the system's reliability and reducing the risk of unexpected halts.

This work is funded by The Knowledge Foundation (KKS), project ARRAY, by the Swedish Research Council (VR), project PSI, and by The Swedish Foundation for Strategic Research (SSF).

The predominant industrial controller redundancy solution is standby redundancy [1]. Standby controller redundancy means that one of the controllers in the controller pair is assigned the primary role, meaning that it is the active controller driving the process [2]. The other controller in the pair is the backup, which is passive until the primary controller fails. In such a situation, the backup assumes the primary role and continues to run the control application. The same principle commonly applies to redundant devices, where one is the primary, and the other is the backup.

The automation domain is experiencing a paradigm shift driven by Industry 4.0's demand for data, increased interoperability, and interconnectivity. OPC UA is identified as the enabling standard for interoperability [3]. The PubSub part of the OPC UA standard details a publish-subscribe model suitable for cyclic real-time communication between controllers and I/O devices [4]. Further, OPC UA PubSub is the communication foundation in the OPC UA Field eXchange (UAFX) standard, targeting field device communication [5], [6].

Our contribution is the OPC UA PubSub analysis through the standby redundancy lens. Using controller and device redundancy as an analysis basis, we identify challenges in publisher failover when using the standard's normative configuration for real-time exchange. The issue is subscriber expectancy on message information populated by the publisher, which typically lacks replication in backup publishers. Based on our analysis, we propose alternative solutions and validate our findings through experiments, including a basic test where we synchronize publisher internals.

The paper is organized as follows: Sec. II present related work; Sec. III provides an overview of OPC UA, especially PubSub; Sec. IV discusses OPC UA PubSub's behavior in selected controller redundancy failure scenarios; Sec. V explores OPC UA PubSub redundancy adaptations, complemented by an experimental evaluation in Sec. VI; and finally, Sec. VII concludes with a summary and future directions.

II. Related work

This work explores OPC UA PubSub in the context of controller and device redundancy. Redundancy is a means of fault tolerance. Fault tolerance, defined by Avizienis et al. as the preservation of operation in the face of faults, is a critical aspect of dependability [7]. The field of fault tolerance research, especially in embedded and industrial systems, is extensive. For instance, Ballesteros et al. introduce an architectural model that dynamically adjusts resilience by dynamic allocation of communication and computational resources according to task criticality [8]. Vitucci et al. investigate hardware design techniques that strengthen product reliability [9]. In the flourishing field of artificial intelligence, Nouioua et al. examine the use of machine learning for network fault detection [10].

Given the reliability requirements of industrial networks, several fault tolerance approaches exist, offering various types of spatial, temporal, or informational redundancy. Álvarez et al. comprehensively survey these mechanisms in industrial networks [11] and Danielis et al. [12] survey reliability aspects of industrial, Ethernet-based, protocols. Neither of the two surveys cover OPC UA PubSub. Nast et als industrial applicability protocol survey covers OPC UA PubSub and reliability as a requirement, but without considering redundancy [13].

Regarding controller redundancy, Simion et al. note that standby modes—either hot or warm—are prevalent redundancy patterns in industrial controllers [1]. The distinction between hot and warm standby lies in the level of backup readiness. However, the backup controls the process in neither warm nor hot standby mode. Additionally, Stój et al. present a cost-effective approach to controller redundancy utilizing EtherCAT [14]. None of these controller redundancy-related works cover OPC UA PubSub.

In the context of OPC UA PubSub, Neumann et al. investigate the requirements that an OPC UA PubSub field device must meet [15]. However, their study does not address reliability aspects. Additionally, the integration of OPC UA PubSub with Time Sensitive Networks (TSN) has been examined by various researchers, demonstrating the feasibility of achieving low latency in real-time communication [16], [17], [18], [5].

Redundancy, in the context of OPC UA, is considered by Ismail et al. that describe a redundant OPC UA server architecture based on ZooKeeper as the underlying replication means [19]. Additionally, Cupek et al. detail the implementation of an OPC UA server in Java, focusing on redundancy aspects [20]. However, these works are related to redundancy for OPC UA Servers, which differs from OPC UA PubSub, further described in Sec. III.

The related work mentioned does not address OPC UA PubSub and redundancy. To our knowledge, this study is the first to examine OPC UA PubSub in the context of controller redundancy.

III. OPC UA

Established in 2008, OPC UA is a comprehensive standard for interoperability across various parts of industrial automation, encompassing machine-to-machine communication, commissioning, and engineering [21]. It introduces a platform-independent and service-oriented architecture and an information model where data and services are accessible via attributes and methods on objects within an information collection called AddressSpace. Remote access to AddressSpace exposed information typically uses OPC UA Client Server [22]. OPC UA Client Server is not designed for real-time, low-latency communication but specifies server redundancy handling. OPC UA prescribes OPC UA PubSub for cyclic, real-time communication, though it does not specify PubSub redundancy handling.

The OPC UA's PubSub part details a publish-subscribe communication model, complementing the client-server model and supporting deterministic cyclic process value exchange [5], [16], [18]. PubSub utilizes a Message Oriented Middleware (MOM) to decouple publishers and subscribers [4]. The MOM can be broker-based, where a broker connects publishers and subscribers, or broker-less, relying on network equipment to act as a broker via multicast groups. This work focuses on broker-less PubSub over User Datagram Protocol (UDP), which targets real-time cyclic data exchange.

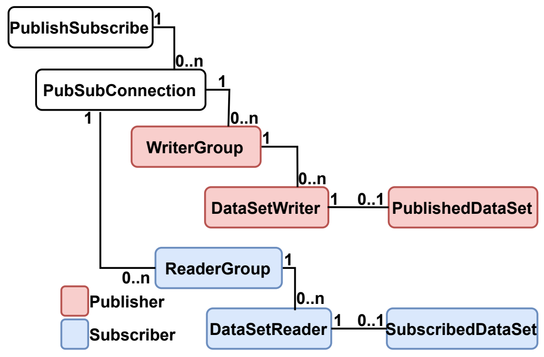
A. *OPC UA PubSub - internals*

This section outlines the internals of OPC UA PubSub as defined by the standard [4]. Fig. 1a depicts the objects and their interconnections. The PublishSubscribe object is the root for all PubSub objects, aggregating one or more PubSubConnections. A PubSubConnection defines the transport protocol, e.g., UDP, and specifies the destination address, either unicast or multicast. Each PubSubConnection may have multiple WriterGroups. WriterGroups encapsulates the data from DataSetWriter into a NetworkMessage. The DataSetWriter fetches and formats the data from PublishedDataSet for publishing. On the subscriber side, ReaderGroup is the receiving counterpart of the WriterGroup, and DataSetReader, analogous to DataSetWriter, unpacks the data and updates the SubscribedDataSet.

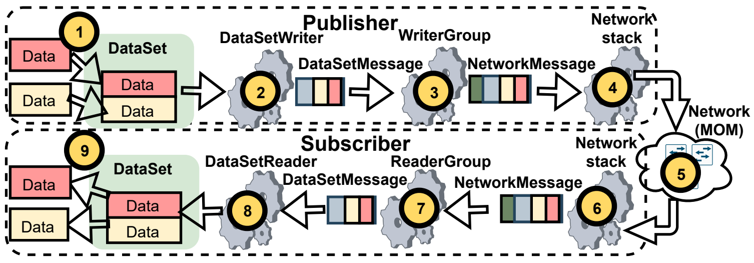
Next, to describe the OPC UA PubSub internals, we'll follow the data from a publisher to a subscriber, depicted in Fig. 1b. The initial step (1) collects the DataSet data for publication. The data acquisition method depends on the type of publisher, for example, sampling I/O values or reading a variable in a control application. The PublishedDataSet defines the data source and type of data, including the DataSetMetaData, which is essential for subscribers to interpret the received DataSetMessage.

Step (2) encapsulates the DataSet into a DataSetMessage using the DataSetWriter. The DataSetWriter DataSetMessage creation offers flexibility in fields selected for inclusion into the DataSetMessage. Which fields to include is controlled by the DataSetFieldContentMask. For example, fields like ConfigurationVersion and DataSetMessage SequenceNumber can be included, as detailed in Fig. 2.

Next, the WriterGroup (3) encapsulates the DataSetMessage into a NetworkMessage. A single WriterGroup can receive DataSetMessages from multiple DataSetWriters, allowing a



(a) OPC UA PubSub objects.



(b) OPC UA PubSub data flow from publisher to subscriber.

Fig. 1: Overview of the OPC UA PubSub internals.

NetworkMessage to carry several DataSetMessages. As the DataSetWriter, the WriterGroup offers flexibility in which fields to include in the NetworkMessage, such as sequence numbers. The PublishingInterval parameter of the WriterGroup determines the frequency of publishing.

In step (4), the WriterGroup sends the NetworkMessage to the publisher's network stack. The broker-less middleware utilizes the network for message broking (5). The publisher can target the message to a specific subscriber using a unicast IP address or address multiple subscribers simultaneously with a multicast address. The network equipment, assumed to be layer two network switches in case of real-time exchange, ensures that the published message reaches the subscribers.

Upon arrival at the subscriber (6), the network stack verifies that the message is meant for this subscriber on the node level, i.e., confirming that the destination address matches a multicast address or the subscriber's unicast address (IP or MAC address).

Next, the ReaderGroup processes the incoming NetworkMessage (7), discarding any messages not intended for this subscriber by verifying the PublisherID in the NetworkMessage. It then extracts the DataSetMessage from the NetworkMessage and forwards the DataSetMessage to the appropriate DataSetReader.

The DataSetReader (8) uses the DataSetMetaData to decode the DataSetMessage and update the DataSet with the data received. The DataSetReader monitors the interval between two DataSetMessages. Suppose no new DataSetMessage appears within the period defined by the MessageReceiveTimeout parameter of the DataSetReader. In that case, the DataSetReader enters an error state. When in error state, the DataSetReader sets the data quality on data update by the DataSetReader to bad, indicating to any dependent application that the data is outdated. Finally, the specific application determines how the received and updated data are utilized (9).

B. OPC UA PubSub protocol - UADP

As discussed, OPC UA PubSub offers various alternatives for the underlying protocol and MOM, ranging from direct, broker-less Ethernet operation to broker-based solutions. This work focuses on the UDP-based alternative, where published network messages are encapsulated in UDP packets on Ethernet, known as the Unified Architecture Datagram Protocol (UADP) [4].

UADP targets cyclic real-time data exchange, such as controller and device communication. The standard outlines recommended message layouts and header configurations. Our description adheres to these recommendations, excluding security enhancements, which are left for future exploration. Fig. 2 illustrates the normative UADP NetworkMessage fields for cyclic real-time communication, per the standard [4].

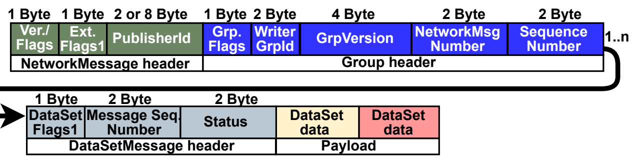
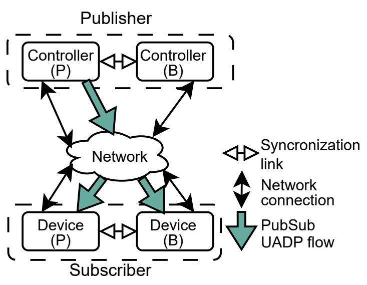


Fig. 2: UADP NetworkMessage layout, with the normative header fields for cyclic data exchange.

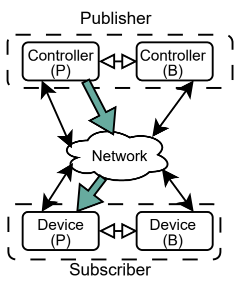
The first field in the NetworkMessage header is the Version/flags byte, which specifies the UADP version and includes flags that indicate the presence of other header fields. The standard suggests that Extended Flags 1, PublisherId, and GroupHeader are included in the cyclic real-time normative NetworkMessage.

The second field, ExtendedFlags1 (ExtFlags1), further defines which additional header fields to expect. For instance, it determines whether the PublisherId is a 16-bit or 64-bit value, with 64-bit being the default. The PublisherId is the third field, a unique publisher identifier within the MOM.

Next is the Group header, containing WriterGroup information. The first field within this header is the GroupFlags (Grp. Flags), which, similar to the NetworkMessage header, indicates the presence of certain fields in the Group header. Again, the normative fields are illustrated in Fig. 2. The WriterGroupId (WriterGrpId) uniquely identifies the WriterGroup within the publisher. The GroupVersion (GrpVersion) notes the time of the last layout change to the data encapsulated by the WriterGroup, such as changes in header fields or DataSet reconfigurations. The NetworkMessageNumber (NetworkMsgNumber) is utilized if multiple NetworkMessages are

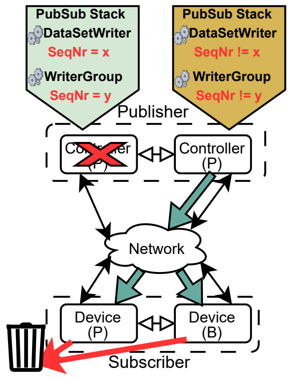


(a) Multicast.

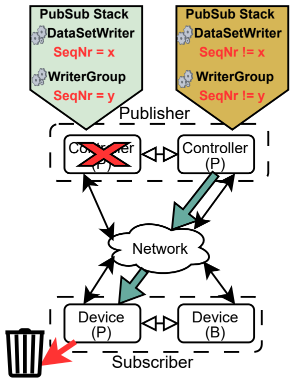


(b) Unicast.

Fig. 3: Redundant controller publishing to the redundant subscribing device.



(a) Muticast - $\mathrm{PC_M}$.



(b) Unicast- $\mathrm{PC_U}$.

Fig. 4: Primary controller failure, i.e., publisher failure.

required to transmit all DataSets managed by the WriterGroup, and the SequenceNumber increments with each message. The subscriber discards the messages that are deemed outdated by the sequence number comparisons. If no messages are received from the publisher within a time exceeding a predetermined “fail-time” (MessageReceiveTimeout), the receiver should be prepared to accept any sequence number. This mechanism ensures resilience in scenarios where the publisher fails and subsequently recovers. However, the subscriber tags the SubscribedDataSet data quality as bad since it is not updated within the MessageReceiveTimeout.

The DataSetMessage header and the DataSetMessage payloads come after the Group header and carry the published values/data. The first field in the DataSetMessage header is DataSetFlags1, specifying the subsequent header fields that are present. Next is the MessageSequenceNumber, a sequence number unique to the DataSetMessage, updated by the DataSetWriter for each DataSetMessage. The Status field follows the MessageSequenceNumber, providing quality information about the data/values within the DataSetMessage, indicating whether the data is good, bad, or uncertain. Last is the payload, comprising the application-specific data exchanged.

IV. PubSub and Controller/Device Redundancy

In this section, we analyze OPC UA PubSub in the context of controller and device redundancy, using two configurations as depicted in Fig. 3. The distinction between the two configurations is the type of UADP connection utilized for PubSub, i.e., multicast or unicast. As mentioned, the UADP PubSub configuration used is the normative for cyclic real-time data exchange as detailed in Sec. III-B.

With the two configurations illustrated in Fig. 3, we investigate failure recovery, i.e., failover, using four different failure scenarios. Those are:

- $\mathrm{PC_M}$ - Primary controller failure with multicast PubSub.
- $\mathrm{PC_U}$ - Primary controller failure with unicast PubSub.
- $\mathrm{PD_M}$ - Primary device failure with multicast PubSub.
- $\mathrm{PD_U}$ - Primary device failure with unicast PubSub.

To keep the explanation and illustration as simple as possible, we assume that the controller is the publisher and the device the subscriber, even though a controller and, likewise, a device could be both subscriber and publisher or any other possible combination. We use the four failure scenarios as the basis for the following subsections.

The assumptions for our failure consequence analysis and their justifications are as follows: the backup can, through the synchronization link, detect if the primary fails and resume the primary role. The application states, particularly the control loop states, are synchronized with the backup. However, the internal states of the OPC UA PubSub stack are not synchronized. The rationale is that control system manufacturers often develop controller runtimes, whereas communication stacks, such as OPC UA, are typically third-party software integrated into the system. Therefore, synchronizing internal stack states is not commonly practiced. Furthermore, we assume the backup is configured identically to the primary regarding OPC UA PubSub-related settings.

A. Primary controller failure with multicast PubSub - $\mathrm{PC_M}$

As illustrated in Fig. 4a and elaborated in Sec. III-A, the WriterGroup and DataSetWriter hold internal states that contribute to the composition of the NetworkMessage, such as the sequence numbers. These sequence numbers are part of the dynamic state data within the WriterGroup and DataSetWriter, and they change with each message transmitted.

When the backup controller takes over as primary, the necessary actions depend on the capabilities of the utilized stack. The backup may need to instantiate the WriterGroup and DataSetWriter, or, if the stack permits, these components could be pre-configured but inactive, allowing for a quicker transition to an operational state upon taking over as primary. Upon activation, the new primary begins publishing with the internal states and sequence numbers from its own WriterGroup and DataSetWriter, as shown in Fig. 4a. Hence, the sequence number in the messages from the new primary is not resumed from where the former primary failed, causing the subscribing device’s ReaderGroup to discard them as outdated. In a rare scenario where the backup takes over just before a sequence number wraparound, the first message from the new primary might have the expected sequence number, allowing the message to go through to the DataSetReader. However, the DataSetReader will likely reject the DataSetMessage due to an old MessageSequenceNumber.

DataSetMessages are discarded until the MessageReceiveTimeout expires. At this point, as mentioned in Sec. III-B, the DataSetReader enters an error state, marking data quality as bad but resetting the expectation for sequence numbers. The subsequent DataSetMessage from the new primary is accepted, but this acceptance comes too late for a seamless transition, as data quality has already been compromised due to the expiration of the MessageReceiveTimeout.

Although multicast allows a backup subscribing device to receive data directly from the primary controller publisher, this doesn't address the sequence number expectation mismatch between subscriber and publisher due to the publisher failover.

B. *Primary controller failure with unicast PubSub* - $\mathrm{PC_U}$

The outcome of the $\mathrm{PC_U}$ failure scenario is identical to that of $\mathrm{PC_M}$, because the failure originates at the publishing end in both examples, and the subscriber's sequence number expectations are the same. The only distinction is that in the $\mathrm{PC_U}$ scenario, only the primary device receives the published message. Nevertheless, this difference does not affect the outcome of the failure scenario; see Fig. 4b.

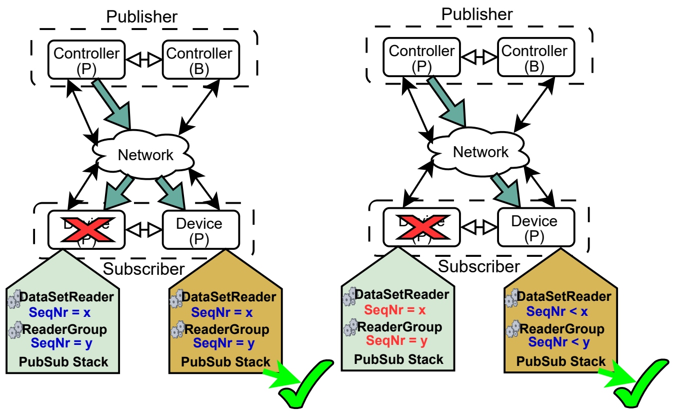
C. *Primary device failure with multicast PubSub* - $\mathrm{PD_M}$

This scenario covers the failure of a subscribing primary device in a multicast configuration as depicted in Fig. 5a. We assume both devices are appropriately configured and have their OPC UA PubSub stacks initialized to subscribe to the multicast published data. Hence, the primary and backup devices receive the data published by the primary controller. However, this requires the pair to ensure consistency. One alternative could be to discard the updated values on the backup when they reach the application layer where the redundancy roles are known. An alternate strategy would be to prevent the backup device from receiving any updates by only activating its subscription once it is required to take over as the primary. This approach would mirror the $\mathrm{PD_U}$ scenario.

In this, the $\mathrm{PD_M}$ scenario, the new primary's ReaderGroup and DataSetReader are already aligned with the former primary's since both devices have been receiving the same messages. Therefore, when the backup takes the primary role, it can seamlessly accept and process the published values from the controller.

D. *Primary device failure with unicast PubSub* - $\mathrm{PD_U}$

In contrast to $\mathrm{PD_M}$, in this scenario, the backup device, when stepping into the primary role due to the failure of the former primary, hasn't received the latest values published by the controller. Upon starting to receive messages, the subscriber—now the new primary—has no preconceived expectations regarding the sequence numbers. Specifically, the ReaderGroup, having not received any NetworkMessage from the publishing controller previously, holds no anticipation about the sequence numbers from the publishing controller's WriterGroup. As a result, it would accept the incoming NetworkMessage and forward the contained DataSetMessages to the DataSetReader. Similarly, the DataSetReader, with no prior expectations regarding the MessageSequenceNumber in the DataSetMessage, would also accept the incoming message. In this scenario, the transition to receiving subscribed data by the new primary device would be seamless, ensuring no interruption in data reception despite the role change.



(a) Multicast.   (b) Unicast.

Fig. 5: Primary device failure, i.e., subscriber failure.

E. *Summary*

Table I summarizes the redundancy and failure scenarios discussed above. It shows that a primary subscriber's failure recovery ($\mathrm{PD_M}$, $\mathrm{PD_U}$) in a redundant pair can be transparent to the OPC PubSub data using layers in the device. However, the recovery of a publisher failure ($\mathrm{PC_M}$, $\mathrm{PC_U}$), e.g., the controller in our discussion, results in bad data quality status, which is undesirable.

TABLE I: Failure and recovery scenario summary.

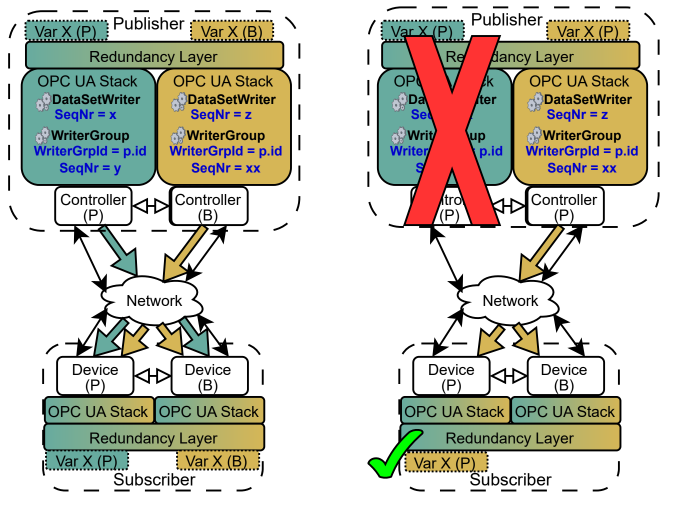
Scenario:	$\mathrm{PC_M}$	$\mathrm{PC_U}$	$\mathrm{PD_M}$	$\mathrm{PD_U}$
Result:	FAIL	FAIL	OK	OK

V. Improvement Alternatives

As summarized in Table I, a publisher failure causes the subscriber to reject DataSetMessages and NetworkMessages from the new primary's DataSetWriter and WriterGroup. This section examines three strategies for seamless publisher failover: (i) the PubSub redundancy layer, (ii) stack synchronization, and (iii) standard extension alternatives, detailed further in subsequent sections. While our examples use multicast publishers, the strategies also apply to unicast publishers.

A. *PubSub redundancy layer*

In the PubSub redundancy layer alternative, PubSub-related redundancy management occurs in a layer above OPC UA PubSub, termed the redundancy layer. As depicted in Fig. 6a, each controller within the pair establishes a WriterGroup and DataSetWriter, while each device in the redundant pair configures a corresponding ReaderGroup and DataSetReader. Synchronization between the pair is managed at the redundancy layer, not within the OPC UA PubSub stacks.



(a) Before failure.

(b) After failure.

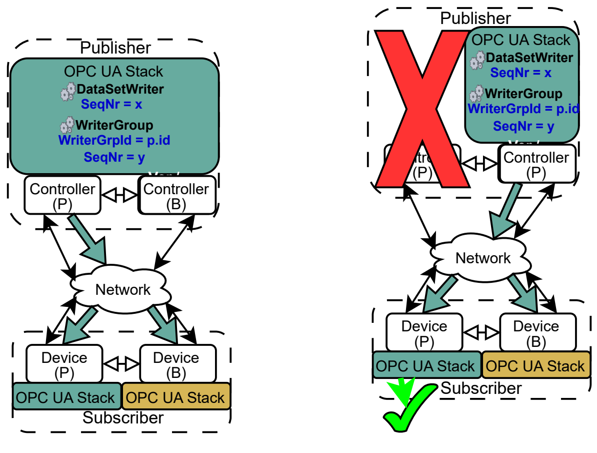
Fig. 6: PubSub redundancy layer establishing parallel publications.

Several approaches exist for the redundancy layer. One method involves embedding redundancy state information within the transmitted data. Embedding redundancy state information in the published data is similar to PROFINET's redundancy approach, which creates parallel logical connections between controller and device, distinguishing one as primary and the others as backups [23]. At the subscriber, the redundancy layer can opt to process data solely from the primary, similar to PROFINET's strategy, potentially minimizing the backup's activity. Note that the redundancy state information is carried in the application-specific data, whereas in PROFINET, this information is part of the protocol.

Another approach allows both controllers in the redundant pair to publish updates and actual data, enabling the subscriber to utilize data from either the primary or the backup, with the option to switch based on error indications from the corresponding DataSetReader or ReaderGroup. This necessitates the backup being up-to-date and publishing data at appropriate intervals, as illustrated by variable X in Fig. 6a.

A third approach is to avoid parallel publications, accept the delays, and hide potential quality degradation resulting from MessageReceiveTimeout expiration in the redundancy layer. With this approach, the redundancy layer would manage data updates. This approach hides the resumption of subscription from the new primary publisher within the redundancy layer. This strategy is most suitable for RawData since the standard does not prescribe quality handling for RawData. A MessageReceiveTimeout would otherwise lead to subscribed data being marked with bad quality.

As exemplified, the redundancy layer is realizable in various incompatible ways. Hence, the redundancy layer alternative will likely need standardization to maintain interoperability between vendors.



(a) Before failure.

(b) After failure.

Fig. 7: Stack synchronization in the publisher creates a seamless appearance, making the stack instances appear as one to the subscriber, hence depicted as a single entity. On the subscriber side, there's no necessity for internal stack state synchronization; thus, we depict it as two distinct instances.

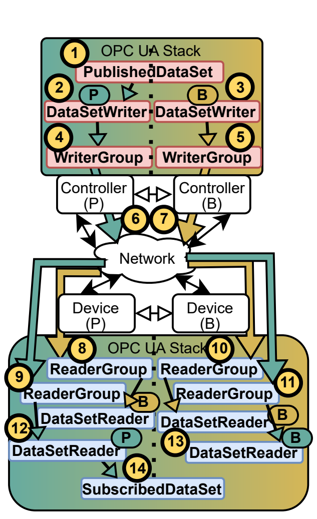
B. Stack synchronization

The stack synchronization entails synchronizing the internal states of the OPC UA PubSub stack from the primary publishing controller to the backup publisher, as illustrated in Fig. 7. The synchronization allows the stack instance running on the backup to resume with the latest state of the primary stack instance. Specifically, sequence numbers need to be synchronized to the backup before transmitting the message. With this strategy, the backup publisher can continue publishing using the same internal state as the former primary's DataSetWriter and WriterGroup. Therefore, from the subscriber's perspective, the failover due to the failure is transparent. This approach's advantage is its transparency to the subscriber. The downside, however, is the need for synchronization support within the OPC UA PubSub stack's internal workings.

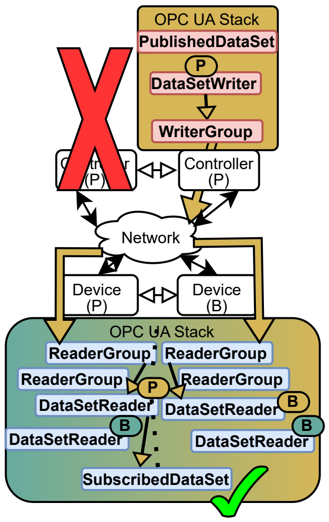
C. Standard extension

As mentioned in Sec. V-A, PROFINET achieves redundancy through parallel logical connections between controllers and devices, designating a primary connection for data exchange and monitoring others to prevent failures leading to undetected redundancy deterioration [23]. A redundancy layer manages these connections, seamlessly switching the primary connection as needed, thereby decoupling redundancy management from the application layer. The OPC UA PubSub standard could similarly incorporate a redundancy model like PROFINET. On a high abstraction level, this section presents one alternative to integrate similar redundancy features into OPC UA PubSub.

The extension includes (i) a RedundancyState field in the DataSetMessageHeader to indicate if the message is from a primary publisher and (ii) a redundancy state for DataSetWriter and DataSetReader. DataSetFlags2, as DataSetFlag1, indicates field presence. A bit in DataSetFlags2 will represent the presence of the RedundancyState field, with



(a) Before failure.



(b) After failure.

Fig. 8: Publisher and subscriber redundancy example with the described OPC UA PubSub standard extension.

DataSetFlag1 indicating the presence of DataSetFlags2. The extension is detailed in Fig. 8a.

The PublishedDataSet (1) represents the data set to be published, synchronized between the primary and backup. The primary DataSetWriter (2), labeled P, creates the DataSetMessage, setting the introduced RedundancyState field to primary in the DataSetMessageHeader. A backup DataSetWriter (3) in the backup controller also publishes, duplicating the data or sending a placeholder, with RedundancyState field set as backup.

The primary controller's WriterGroup (4) embeds the primary DataSetWriter's DataSetMessage into a NetworkMessage. Similarly, the backup controller's WriterGroup (5) encapsulates the DataSetMessage into a NetworkMessage. The primary and backup WriterGroups have distinct WriterGroupIds that ideally follow a convention that allows corresponding pairs to be identified.

The NetworkMessages are multicast (6)(7), ensuring both devices in the redundant pair receive all messages. Alternatively, additional WriterGroups using unicast could also ensure that both devices in the redundant pair receive all messages without using multicast.

ReaderGroups (8-11) on the redundant device receive NetworkMessages from both controllers, allowing connection monitoring and means to prevent undetected redundancy deterioration. The ReaderGroups forward DataSetMessages to the DataSetReaders (12-13). Each device has two ReaderGroups to handle messages from both publishing controllers in the redundant controller pair. Only the primary state DataSetReader updates the SubscribedDataSet (14), and only with DataSetMessages where the RedundancyState field equals primary.

If the primary publisher fails, the backup DataSetWriter becomes the primary, continuing to publish the PublishedDataSet, as shown in Fig. 8b. The specifics of this transition, particularly changing the DataSetWriter's redundancy state, are likely implementation-dependent.

VI. Experimental evaluation

Sec. IV looked at OPC UA PubSub in a redundancy context, identifying that publisher failovers aren't inherently transparent to subscribers, see Table I. Consequently, Sec. V explored alternatives, identifying the synchronization of internal stack states as the approach that maintains standard compatibility without necessitating specialized handling by the subscriber. This section experimentally tests these findings, employing the multicast scenarios $\mathrm{PC_M}$ and $\mathrm{PD_M}$, using the transport protocol and UADP message configurations outlined in Sec. III-B and Fig. 2.

For the experiment, we use the open source OPC UA stack open62541 [24], [25] running on Ubuntu 20.04.6 LTS using VMWare. One Virtual Machine (VM) acts as the publishing controller, and the other as the redundant subscriber device. We simulate publisher failure recovery (failover) by halting and restarting the publisher in the same VM and process. Further, we simulate subscriber failure and recovery by restarting the subscriber. The exchanged data consists of ten four-byte values the publisher publishes every 100 ms. The test implementation and modifications to the open62541 stack are available on GitHub [26].

A. Implementation

The open62541 PubSub implementation doesn't verify sequence numbers. It checks neither the Group header nor the DataSetMessage sequence numbers. To address this, we added checksum verification as per the standard. We added checking of the Group header sequence number, updated by the WriterGroup and checked by the ReaderGroup, and checking of the DataSetMessageHeader sequence number, updated by the DataSetWriter and verified by the DataSetReader. Messages with incorrect checksums are discarded.

For stack synchronization, we implemented a simple yet representative solution allowing a resumed instance to continue with the sequence number last used. This implementation is sufficient for our experiment, where we simulate the failure by stopping the publisher and resuming a new one in the same VM and process. For more details, refer to the implementation [26].

B. Experiment and Result

We conducted the experiments using the setup previously described and three different variants of the open62541 stack: (i) the original open62541 version - ORG, (ii) sequence number adherence - SEQ, and (iii) synchronization and sequence number adherence - SYNC. Table II displays the results, with OK indicating a seamless recovery from the subscriber's perspective and FAIL indicating a non-seamless recovery.

The results, as detailed in Table II, reflect the analysis from Sec. IV highlighting the challenges with publisher failures. While the original open62541 version (ORG) shows OK for publisher failure scenarios ($\mathrm{PC_M}$), this is attributed to the stack's non-adherence to sequence numbering; it ignores them. In the SEQ variant, publisher recovery is not transparent to the subscriber; the new primary publisher uses sequence numbers perceived as outdated by the subscriber, leading to message rejection. Conversely, the SYNC variant enables the new primary publisher to resume with sequence numbers aligned with subscriber expectations, resulting in a successful, seamless recovery.

TABLE II: Failure recovery result for different scenarios and stack variants.

Scenario	**Stack variant**		
	ORG	**SEQ**	**SYNC**
$\mathrm{PC_M}$	OK[1]	FAIL	OK
$\mathrm{PD_M}$	OK	OK	OK

VII. Conclusion and future work

This work has examined OPC UA PubSub within the context of controller and device redundancy, focusing on the standard's recommended messaging configuration for real-time, cyclic data exchanges. The type of exchange that is typical in industrial settings. We explored four failure scenarios in a redundant controller and device setup using OPC UA PubSub, assessing the transparency of failovers, where the backup should take over seamlessly without impacting the application.

Our analysis revealed that the publisher redundancy is not transparent—highlighting a gap where the redundant publisher fails. We proposed three alternatives to address this in order to provide a seamless publisher failover. Further, we conducted experiments using the open62541 stack, implementing one of the suggested alternatives to validate our discussions. We conclude that achieving publisher redundancy in a way that is transparent to subscribers is feasible but requires stack support.

Future work includes adding general platform-independent redundancy support in open62541, as well as an in-depth evaluation and implementation of a redundancy layer similar to that of PROFINET, allowing for status monitoring to prevent redundancy deterioration from going undetected.

References

[1] A. Simion and C. Bira, "A review of redundancy in plc-based systems," *Advanced Topics in Optoelectronics, Microelectronics, and Nanotechnologies XI*, vol. 12493, pp. 269–276, 2023.

[2] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," *Distributed systems*, vol. 2, pp. 199–216, 1993.

[3] D. Bruckner, R. Blair, M. Stanica, A. Ademaj, W. Skeffington, D. Kutscher, S. Schriegel, R. Wilmes, K. Wachswender, L. Leurs, *et al.*, "Opc ua tsn a new solution for industrial communication," *Whitepaper. Shaper Group*, vol. 168, pp. 1–10, 2018.

[4] "Opc 10000-14 - ua specification part 14: Pubsub 1.05.03." https://reference.opcfoundation.org/Core/Part14/v105/docs/. Accessed: 2024-03-26.

[5] S. Grüner, A. E. Gogolev, and J. Heuschkel, "Towards performance benchmarking of cyclic opc ua pubsub over tsn," in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, IEEE, 2022.

[1]OK due to stack not adhering to the standard prescribed sequence number verification.

[6] "Opc 10000-80 - uafx part 80: Overview and concepts 1.00." https://reference.opcfoundation.org/UAFX/Part80/v100/docs/. Accessed: 2024-03-31.

[7] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dep. and Sec. Comp.*, vol. 1, pp. 11–33, Jan 2004.

[8] A. Ballesteros, M. Barranco, J. Proenza, L. Almeida, F. Pozo, and P. Palmer-Rodríguez, "An infrastructure for enabling dynamic fault tolerance in highly-reliable adaptive distributed embedded systems based on switched ethernet," *Sensors*, vol. 22, no. 18, p. 7099, 2022.

[9] C. Vitucci, D. Sundmark, M. Jägemar, J. Danielsson, A. Larsson, and T. Nolte, "Fault management impacts on the networking systems hardware design," in *IECON 2023-49th Annual Conference of the IEEE Industrial Electronics Society*, pp. 1–8, IEEE, 2023.

[10] M. Nouioua, P. Fournier-Viger, G. He, F. Nouioua, and Z. Min, "A survey of machine learning for network fault management," *Machine Learning and Data Mining for Emerging Trend in Cyber Dynamics: Theories and Applications*, pp. 1–27, 2021.

[11] I. Álvarez, A. Ballesteros, M. Barranco, D. Gessner, S. Djerasevic, and J. Proenza, "Fault tolerance in highly reliable ethernet-based industrial systems," *Proc. IEEE*, vol. 107, no. 6, pp. 977–1010, 2019.

[12] P. Danielis, J. Skodzik, V. Altmann, E. B. Schweissguth, F. Golatowski, D. Timmermann, and J. Schacht, "Survey on real-time communication via ethernet in industrial automation environments," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pp. 1–8, IEEE, 2014.

[13] M. Nast, H. Raddatz, B. Rother, F. Golatowski, and D. Timmermann, "A survey and comparison of publish/subscribe protocols for the industrial internet of things (iiot)," in *Proceedings of the 12th International Conference on the Internet of Things*, pp. 193–200, 2022.

[14] J. Stój, "Cost-effective hot-standby redundancy with synchronization using ethercat and real-time ethernet protocols," *IEEE Trans. on Autom. Science and Eng.*, vol. 18, no. 4, pp. 2035–2047, 2020.

[15] R. Neumann, C. von Arnim, M. Neubauer, A. Lechler, and A. Verl, "Requirements and challenges in the configuration of a real-time node for opc ua publish-subscribe communication," in *2023 29th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, pp. 1–6, IEEE, 2023.

[16] J. Pfrommer, A. Ebner, S. Ravikumar, and B. Karunakaran, "Open source opc ua pubsub over tsn for realtime industrial communication," in *2018 IEEE 23rd international conference on emerging technologies and factory automation (ETFA)*, vol. 1, pp. 1087–1090, IEEE, 2018.

[17] A. Eckhardt and S. Müller, "Analysis of the round trip time of opc ua and tsn based peer-to-peer communication," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 161–167, 2019.

[18] P. Denzler, T. Frühwirth, D. Scheuchenstuhl, M. Schoeberl, and W. Kastner, "Timing analysis of tsn-enabled opc ua pubsub," in *2022 IEEE 18th International Conference on Factory Communication Systems (WFCS)*, pp. 1–8, IEEE, 2022.

[19] A. Ismail and W. Kastner, "Coordinating redundant opc ua servers," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, IEEE, 2017.

[20] R. Cupek, K. Folkert, M. Fojcik, T. Klopot, and G. Polakow, "Performance evaluation of redundant opc ua architecture for process control," *Transactions of the Institute of Measurement and Control*, vol. 39, no. 3, pp. 334–343, 2017.

[21] "Opc 10000-1 - ua specification part 1: Overview and concepts." https://reference.opcfoundation.org/Core/Part1/v105/docs/. Accessed: 2024-03-26.

[22] "Opc 10000-4 - ua specification part 4: Services 1.05.03." https://reference.opcfoundation.org/Core/Part4/v105/docs/. Accessed: 2024-03-26.

[23] "High availability - guideline for profinet version 1.2 - date feburary 2020 - order no.: 7.242."

[24] F. Palm, S. Grüner, J. Pfrommer, M. Graube, and L. Urbas, "Open source as enabler for opc ua in industrial automation," in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pp. 1–6, IEEE, 2015.

[25] "Open 62541 - project page." https://www.open62541.org/. Accessed: 2024-04-05.

[26] "The experiment source code - fork of open62541." https://github.com/Burne77a/open62541-red-inv. Accessed: 2024-04-18.