Leveraging Relay Nodes to Deploy and Update Services in a CPS with Sleeping Nodes

Antoine Omond*†, Hélène Coullon†, Issam Rais* and Otto Anshus*
*Department of Computer Science, UiT The Arctic University of Norway, Tromsø, Norway
†IMT Atlantique, Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, F-44000 Nantes, France
Emails: antoine.s.omond@uit.no, helene.coullon@imt-atlantique.fr, issam.rais@uit.no, otto.anshus@uit.no

***Abstract*—Cyber-physical systems (CPS) deployed in scarce resource environments like the Arctic Tundra face extreme conditions. Nodes in such environments are forced to rely on batteries and sleep most of the time to maximize their lifetime. While being autonomous, nodes have to collaborate to make scientific observations. As a result, their deployments and updates are subject to coordination (e.g., prevent interruption of a service used by other nodes). In this paper, we study analytically and experimentally, to what extent using relay nodes for communications reduces the deployment and update durations, and which factors influence this reduction. Intuitively, as dealing with sleeping nodes with low chances of uptime overlaps (no uptime synchronization), a coordinated deployment or update takes very long to finish if nodes have to communicate directly.**

***Index Terms*—CPS, Deployment, Update, Coordination.**

I. Introduction

Cyber-physical systems (CPS) are systems where physical instruments are combined with digital devices and software components to achieve smart observations, computations, and decision-making. They are used in a large variety of use cases such as health care [1], agriculture [2], and environmental monitoring [3].

The Arctic Tundra is one of the most sensitive ecosystems to climate change. While studying this environment is of high importance, its monitoring is nowadays very limited. The Distributed Arctic Observatory (DAO)[1] project proposes a CPS observing the Arctic Tundra. It is mainly composed of *Observation Nodes* (ONs), responsible for monitoring the ecosystem through physical instruments, gathering data, and running small computations. ONs can also collaborate for observations when located close to each other through local or temporary network connections.

This use case imposes extreme constraints on the CPS. Because the Arctic Tundra is a protected area, no infrastructure is present on the field. Nodes have to rely on batteries exclusively. Due to bad weather conditions and very short sun exposition during winter, swapping batteries or harvesting energy on a regular basis are not plausible solutions. Thus, to maximize their lifetimes, each node has its own frequency of short uptime and long sleeping periods. In this paper, we consider the realistic scenario where ONs are not synchronized in their uptime periods, as synchronizing them has an important cost in a dynamic environment. We want to study the case where ONs wake up when required for scientific observations only, i.e., randomly according to dynamic requirements from scientists, or according to dynamic external events (e.g., a moving animal).

Because nodes are collaborating, available services on a node might have dependencies with services hosted on other nodes. When such services have to be deployed or updated, coordination is required to reach the new target configuration.

Intuitively, considering direct communications between ONs and due to the very low chances of uptime overlaps between ONs, deploying or updating services in the DAO might take a very long time to finish. However, specific nodes (denoted *relay nodes*) of the DAO can be equipped with powerful batteries, thus being more frequently available to relay messages between nodes. Leveraging those specific nodes for exchanging messages between ONs offers an opportunity for indirect asynchronous communications which should intuitively speed up deployments and updates.

In this paper, we want to study how direct or indirect communications (with and without a relay node) affect the deployment and update durations. More precisely, we answer the following research questions.

RQ1: When deploying or updating interdependent services hosted on different sleeping ONs, to what extent does using indirect communications, by leveraging relay nodes, reduces the duration of the process compared to direct communications between nodes?

RQ2: Which factors influence the deployment and update duration, in both cases?

The contributions of the paper are: (1) the modeling of our case study, and an analytical answer to our questions; (2) an experimental evaluation conducted on a real infrastructure with Raspberry Pis on both the deployment and update to validate our analytical study. The rest of this paper is organized as follows. Section II details the considered CPS in the Arctic Tundra. Section III presents the modeling of a deployment or an update when having sleeping nodes with and without relay nodes. In Section IV the experimental setup is detailed, and in Section V are presented the results of our experiments. Section VI presents the related work. Finally, Section VII concludes this work, and opens to some perspectives.

[1] https://en.uit.no/project/dao

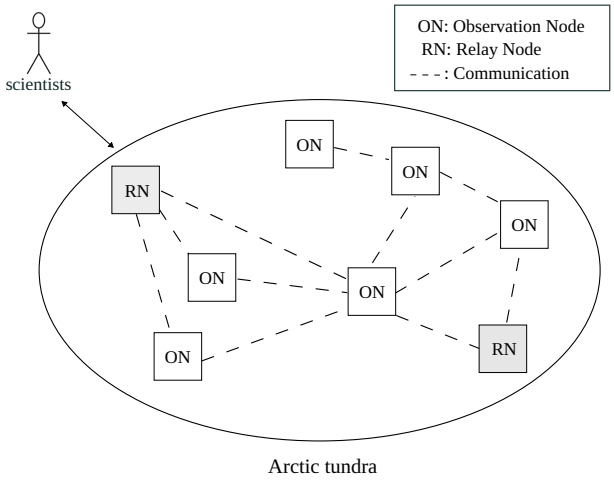

Fig. 1: DAO-CPS in the Arctic Tundra.

II. The Arctic Tundra Use-Case

The Arctic tundra is a very large, hard-to-reach, and potentially dangerous ecosystem. Presently, much less than 1% of the Arctic tundra is monitored. Therefore, to accurately study climate change, larger observations of the Arctic tundra are needed. Figure 1 gives an overview of a DAO-CPS observing the Arctic Tundra [4]. For our study, we put the nodes in two categories: the observation nodes (ON) and the relay nodes (RN).

First, ONs typically embed computing capabilities and physical instruments to observe the environment and are deployed at scale in the Tundra. When waking up, an ON can perform several actions including performing measurements, doing small computations, deploying new services for future measurements, or updating existing services. The uptime frequency of each ON is not known in advance and is considered random [4]. Indeed, ONs wake up according to dynamic requirements from scientists, or according to events occurring in the Tundra (e.g., moving animals, CO2 threshold). It is essential to notice that each ON can perform multiple types of measurements and computations and may wake up dynamically according to their own local events and decisions. In cases where services on multiple nodes are interdependents (i.e., collaborating), nodes have to synchronize their deployments and updates.

Second, RNs are notably used to help ONs communicate, rather than compute and observe. RNs are also under a limited energy budget but are equipped with more powerful batteries, making them more likely to be reachable by ONs. In particular, we consider that an RN has the knowledge and enough energy to be awakened simultaneously as the ONs connected to it [5]. RNs can be costly as they need powerful batteries. Consequently, the number of RNs to deploy in the Tundra has to be studied and limited. Such a detailed study will be the subject of another paper. In the rest of the paper, we consider one clique of multiple ONs around one RN. This paper exclusively focuses on the study of deployment and update durations when introducing one RN.

$$
\begin{aligned}
&\text{ON1}: \ install_1^1 \prec config_1^2, config_1^2 \prec run_1^3\\
&\text{ON2}: \ install_2^1 \prec config_2^2, install_2^1 \prec config_2^3,\\
&\qquad config_2^2 \prec run_2^4, config_2^3 \prec run_2^4,\\
&\text{ON3}: \ install_3^1 \prec config_3^2, config_3^2 \prec run_3^3\\
&\text{Between nodes}: \ config_1^2 \preccurlyeq config_2^2, run_1^3 \preccurlyeq run_2^4\\
&\qquad config_3^2 \preccurlyeq config_2^3, run_3^3 \preccurlyeq run_2^4
\end{aligned}
$$

Listing 1: Deployment of Figure 2 modeled as a strict partially ordered set with two order relations $\prec$ and $\preccurlyeq$

III. Modeling

In this section, we give a generic model of both a deployment and update process when facing sleeping nodes. This model leads to an analytical answer to RQ1 and RQ2, experimentally validated in the rest of the paper.

A. Deploy and update modeling

Many languages exist in the literature to model the configuration of a system and apply changes to this configuration, i.e., reconfiguration [6]. The goal of such languages is to offer the programming support to write well-structured programs that express how to move from the current state of the system to a new desired state. Reconfiguration languages can be used to write either deployment or update procedures. In [7], [8] a reconfiguration is modeled as a graph of actions to apply on a system. We follow this idea in the following modeling.

A deployment or an update procedure on multiple nodes can be modeled as a partially ordered set $(A_r, \prec, \preccurlyeq)$ where $\prec$ is the order relation used between two actions on the same node, and $\preccurlyeq$ is the order relation used between actions executed on two different nodes.

The $\preccurlyeq$ relation models a required communication between two nodes of the CPS to solve the remote dependency.

A deployment or an update procedure, as being a strict partially ordered set, can also be modeled as a directed acyclic graph (DAG) $(A_r, D_r \cup D_r^*)$ with A_r the nodes of the graph representing actions, D_r the edges representing dependencies between actions on the same node, and D_r^* the edges representing dependencies between actions on two different nodes.

Example: Figure 2 and Listing 1 give an example of deployment with three nodes: Figure 2 depicts the DAG modeling, while Listing 1 represents a subpart of the partially ordered set modeling (with two nodes). The actions are denoted $install_i^j$, $config_i^j$, and run_i^j, where i represents the ON number on which the action is executed, and j identifies the action to execute.

The duration of a deployment or an update is the time spent to execute all actions on all nodes. In other words, the duration is the longest path, LP, in $(A_r, D_r \cup D_r^*)$: $LP(A_r, D_r \cup D_r^*)$ [7].

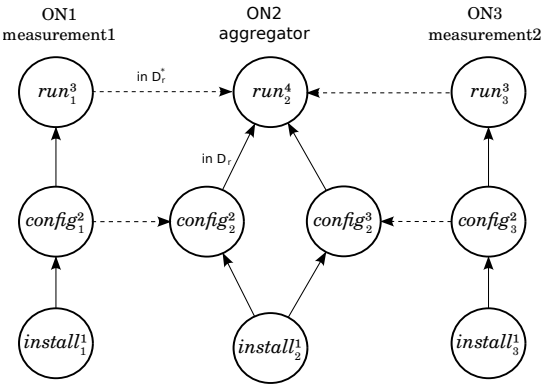

Fig. 2: A deployment modeled as a DAG partitioned into three subgraphs, one for each ON. Plain edges represent dependencies between actions of a single ON in D_r. Dashed edges represent dependencies between actions of two different ONs in D_r^*. This also illustrates our deployment case study of Section V (three services distributed on three ONs).

The duration of each edge in D_r^* is $W(a_i \lessdot a_j)$: the waiting duration before the node responsible for a_j is informed that the dependency $a_i \lessdot a_j$ is solved. W is in particular the element of interest in this paper. Indeed, if using direct (without RN) or indirect (with RN) communication between ONs, W is impacted, while other weights of the graph are unchanged.

B. Sleeping nodes and overlap modeling

In this subsection, we introduce a model of the sleeping nodes of our Arctic Tundra use case. An ON of the DAO-CPS alternates sleeping and uptime periods, without a regular frequency (see Section II).

Considering the infinite time as $\mathbb{R}^+$, and the sets of sleeping and uptime periods of a node $\mathfrak{S}$, $\mathfrak{U}$ such that $\mathfrak{U} \cup \mathfrak{S} = \mathbb{R}^+$ and $\mathfrak{U} \cap \mathfrak{S} = \varnothing$. A period in either $\mathfrak{U}$ or $\mathfrak{S}$ is a pair (s, e), where s, e are respectively the starting and ending points of the period. The period duration is $d = e - s$.

There exists an overlapping period $\mathfrak{o} \in \mathfrak{O}$ between two nodes (here i, j) if and only if it exists at least one pair of uptime periods, $(s_i, e_i) \in \mathfrak{U}_\mathfrak{i}$ on node i, and $(s_j, e_j) \in \mathfrak{U}_\mathfrak{j}$ on node j, such that $s_i \leqslant s_j \leqslant e_i$ or the opposite $s_j \leqslant s_i \leqslant e_j$. The duration of $\mathfrak{o}$ is then $\min(e_i, e_j) - \max(s_i, s_j)$.

Direct communications between two nodes of the CPS can happen if and only if there exists at least one overlapping period $\mathfrak{o} \in \mathfrak{O}$ between the two nodes i, j, denoted $\mathfrak{o}(i, j)$.

Figure 4 illustrates an example of uptime and sleeping periods as well as overlapping periods between two nodes.

C. Waiting duration modeling

In this subsection, we link the waiting duration W of a deployment or an update duration to the sleeping frequency of ONs in the CPS.

We denote t_{a_i} the instant where the action a_i is finished on node i, and $t_{a_i \lessdot a_j}$ the instant where node j needs a_i to be finished on node i before being able to apply action a_j.

The waiting duration induced by the dependency $a_i \lessdot a_j$ with direct communications between nodes (i.e., without the RN) is

$$W(a_i \lessdot a_j) = \begin{cases} s_{\mathfrak{o}_{i,j}(t_{a_i})} - t_{a_i \lessdot a_j} & t_{a_i} > t_{a_i \lessdot a_j} \\ s_{\mathfrak{o}_{i,j}(t_{a_i \lessdot a_j})} - t_{a_i \lessdot a_j} & \text{otherwise} \end{cases} \quad (1)$$

where $s_{\mathfrak{o}_{i,j}(t_{a_i})}$ and $s_{\mathfrak{o}_{i,j}(t_{a_i \lessdot a_j})}$ are, after t_{a_i} (resp. $t_{a_i \lessdot a_j}$), the starting points of the current or next overlap period between nodes i and j, with in both cases $s_{\mathfrak{o}(i,j)} \geqslant t_{a_i}$ and $s_{\mathfrak{o}(i,j)} \geqslant t_{a_i \lessdot a_j}$.

Intuitively, the node j has to wait from the instant it requires a_i to be finished on node i ($t_{a_i \lessdot a_j}$), until the overlap period between i and j ($s_{\mathfrak{o}(i,j)}$). In one case this overlap period should happen after t_{a_i}, in the other case after $t_{a_i \lessdot a_j}$. Note that node i is not blocked and can continue its local execution while j is waiting.

We now study the waiting duration in case of indirect communications between nodes through an RN, meaning that nodes do not have to overlap to communicate.

The waiting duration induced by the dependency $a_i \lessdot a_j$ with an indirect communication is

$$W(a_i \lessdot a_j) = \begin{cases} s_{\mathfrak{u}_j(t_{a_i})} - t_{a_i \lessdot a_j} & t_{a_i} > t_{a_i \lessdot a_j} \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

where $s_{\mathfrak{u}_j(t_{a_i})}$ is, after t_{a_i}, the starting point of the next uptime period of node j.

With indirect communication by using the RN, as soon as a_i is finished on node i the information is stored in the RN (which is considered awakened). Hence, on the one hand, if node j needs a_i to be finished after node i ends a_i (i.e., $t_{a_i} > t_{a_i \lessdot a_j}$), the waiting duration is null for j. On the other hand, if node j needs a_i to be finished before node i ends a_i, node j has to wait from the instant it requires a_i ($t_{a_i \lessdot a_j}$), until an uptime period after t_{a_i} ($s_{\mathfrak{u}_j(t_{a_i})}$).

By analytically comparing the equations (1) (i.e., direct communication without RN) and (2) (indirect communication with RN), the following statements stand:

- RQ1: by definition, we always have the relations $s_{\mathfrak{o}_{i,j}(t_{a_i})} \geqslant s_{\mathfrak{u}_j(t_{a_i})}$ and $s_{\mathfrak{o}_{i,j}(t_{a_i \lessdot a_j})} \geqslant s_{\mathfrak{u}_j(t_{a_i})}$ because in best case the next uptime instant of j is also an uptime period for i. Thus, the waiting duration in the direct case should always be greater than or equal to the indirect case. Consequently, the deployment and update durations should always be faster if using an RN to communicate between ONs.
- RQ2: from both equations, we can extract that the waiting duration, thus the deployment and update duration, depends either on the frequency of overlaps between nodes ($s_{\mathfrak{o}_{i,j}}$ in Eq. 1) when using direct communications, or on the uptime order between nodes (i.e., $s_{\mathfrak{u}_j(t_{a_i})}$ in Eq. 2) when using indirect communications (i.e., an RN is used).

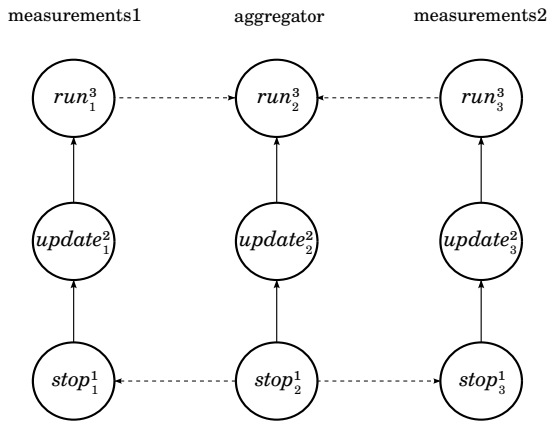

Fig. 3: Graph of actions to *update* our three services distributed on three ONs.

IV. Experimental setup

In the rest of this paper, we will validate the analytical expectations concerning our research questions through experiments. We first describe the experimental setup.

A. Deployment and update language

To experiment and answer our research questions, we need a language that is able to (1) model and execute deployment and update across multiple nodes in a decentralized manner, and (2) handle communications between nodes either directly or indirectly.

In the literature, as far as we know, only a few contributions offer a language able to model and execute deployment and update in a decentralized manner (see Section VI). Among the decentralized approaches we have found, μs [9] (called *Muse* in the rest of this paper) is a decentralized orchestrator that relies on the DevOps tool Pulumi. It aims to help DevOps teams to coordinate their changes in the cloud. Muse, unlike its concurrent solutions [10], [11] is not restricted to deployments and can handle more advanced changes such as updates, scaling, etc. Muse handles direct communications between nodes: each node hosts a gRPC server and communicates with others by making requests to remote APIs. For these reasons, in addition to experimenting with Muse, we have chosen to implement a decentralized version of the reconfiguration language Concerto [8] that embeds both direct and indirect communications between nodes[2]. The indirect version of this code uses the decentralized message broker Zenoh[3]. We call these two versions of the decentralized Concerto *direct* for direct communications without RN, and *rn* for indirect communications with one RN.

B. Use cases

Our use cases are inspired from [4] and from [8]: one ON is running an *aggregator* service and relies on the output of n others ONs running *measurement* services. Five ONs are running *measurement* services. We have dependencies between the *aggregator* and the *measurement* services.

In this context, we experiment with two different procedures: *deploy* and *update*. Figure 2 and 3 respectively illustrate the dependency graph of actions for *deploy* and *update* procedures, with only two *measurement* services and the *aggregator* (i.e., three nodes). For interested readers, the procedures are detailed in [8] as well as in the publicly available source code[4]. During the deployment, the *aggregator* has two synchronizations with each *measurement* service (or node): one at the configuration level of the services and one to start the services. During the update, all services have to suspend, do the update, then run again. Two synchronizations are also required here: *measurement* services need to wait for the *aggregator* to stop using their services before stopping their own; then, before running again the *aggregator* needs to wait for *measurement* services to ensure that they are running again before being able to use them.

Note that these synchronizations are automatically handled by Muse, *direct* and *rn*, but we detail them to give an intuition on the level of coordination required by each case, thus the required communications between nodes.

To cover a larger spectrum of cases, to avoid favorable cases, and to favor reproducibility, we do not execute real commands within our actions but instead randomly generate action duration, as stated in [4]. To be realistic, the duration of actions is generated between 1 and 30 seconds, using a *lognormal* distribution. In other words, low values are more represented than high values. Our experiments use two different random draws denoted ard_0 and ard_1 in the rest of the paper.

C. Uptime scenarios

During one experiment, nodes follow a predefined scenario made of uptime and sleeping periods. To be faithful to reality [4], we set the uptime duration of nodes at 50 seconds. Our experiments have a deadline of 3 hours and each node wakes up a total of 45 times. To generate the scenarios, we use a combination of two parameters: (1) the total number of overlaps that nodes hosting a *measurement* service have with the *aggregator* (*Nb.Ovlp.* for short), this parameter influences $s_{\mathfrak{o}_{i,j}}$ of Eq. 1; and (2) the order of uptime periods between nodes hosting a *measurement* service and the *aggregator* (*Upt.Order* for short), this parameter influences $s_{\mathfrak{u}_j(t_{a_i})}$ of Eq. 2. Both uptimes and overlaps are uniformly distributed throughout the time slots of the experiment.

Figure 4 illustrates how these parameters influence the generation of scenarios. In the base scenario, we see that $ON1$ always wakes up before $ON0$, and that there are two overlaps in the time slots (TS) 2 and 4. *Scenario 1* shows a change with three overlaps instead of two. This should have an impact when using direct communications (i.e., *direct*). In *Scenario 2*, the order of uptime between $ON0$ and $ON1$ is changed. This

[2]https://github.com/Concerto-D/concerto-decentralized/tree/cpscom2023

[3]https://zenoh.io/

[4]https://github.com/Concerto-D/evaluation/tree/cpscom2023

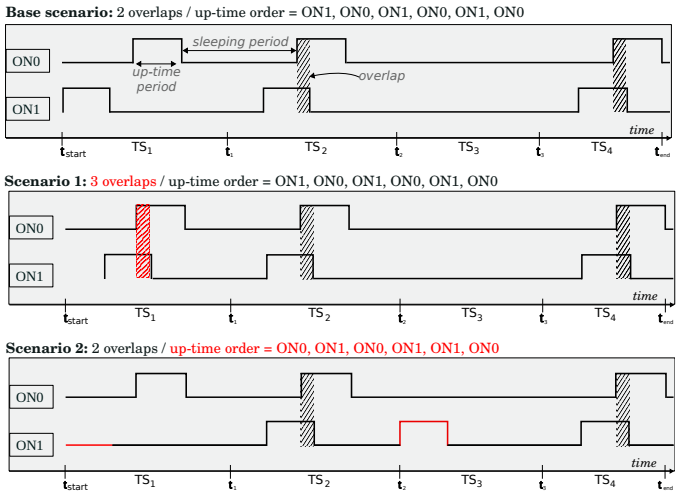

Fig. 4: Illustration of three different uptime scenarios with two ONs. A scenario is divided into time slots (TS). The first scenario is the base to compare to others. The second scenario differs from the base in the number of overlaps, while the third differs in the order of uptime periods.

should have an impact when using indirect communications (i.e., *rn*). We generated two categories of scenarios that aim at studying the sensitivity of direct and indirect communications to these two different parameters (*Nb.Ovlp.* and *Upt.Order*). The number of overlaps can take three values: 7, 15, or 30. Regarding the uptime order, we have triggered three different orders randomly uto_0, uto_1, uto_2.

D. Metric and Infrastructure

In our experiments, we measure the overall deployment and update duration, i.e., the maximum total uptime and sleeping periods needed by the nodes to complete the procedure. This includes the time spent to complete actions, the waiting duration between nodes, and the sleeping periods.

Our experiments are conducted on a cluster of seven Raspberry Pi 4, running on a Cortex-A72 CPU. These Raspberries have been connected to the Grid5000[5] network, a scientific test-bed to run and facilitate the reproducibility of experiments. Note that this paper does not study the impact of low bandwidths or packet losses on the procedure duration. This is the subject of future work.

V. Results

In this section, we discuss the results of our experiments. Table I and Table II respectively gather the results of the experiments that vary the parameters *Upt.Order* and *Nb.Ovlp*. Each of our experiments has been executed four times and a mean is represented in tables. We have observed a maximum standard deviation of 0,06% for Concerto-D and 7% for Muse (due to a fully loaded CPU, as discussed later).

[5]https://www.grid5000.fr/w/Grid5000:Home

A. Results on uptimes' orders

Table I presents the results obtained by varying the uptimes' orders and fixing the number of overlaps to 15. Overall, the deployment and update duration using *rn* (i.e., indirect communications) is lower than using *direct*, which is expected from our analytical study (see Section III). When comparing *rn* with *direct*, the minimum reduction is 19% (from 1036,81s to 841,01s), while the maximum is 90% (from 2604,93s to 267,04s). When comparing *rn* with Muse, the minimum reduction is 50% (from 1674,58s to 841,01s) while the maximum is 93% (from 3746,20s to 267,04s).

On one hand, the durations when using *direct* remain stable among the three uptime orders. Even if the uptime order is supposed to affect more the procedures with indirect communications, we may expect that the results with direct communications are also affected by the uptime order, which is not the case here. It is due to the fact that waiting for the next overlap (required in the *direct* case only) largely overshadows the uptime order parameter.

On the other hand, for *rn*, the results are very sensitive to the uptime order. For *deploy* the maximum variation is 151% (from 257,41s to 645,38s). For *update* the maximum variation is 224% (from 259,88s to 841,01s). This is expected from our analytical study, as the deployment and update duration using indirect communications is mainly sensitive to the uptime order (Equation 2).

B. Results on the number of overlaps

Table II presents the results obtained by varying the number of overlaps during experiments and fixing the order of uptimes to uto_0.

Compared to *direct*, the results of *rn* reduce the deployment and update durations by a minimum of 13% (from 269,64s to 233,20s), and a maximum of 96% (from 6441,27s to 266,99s). For *7 overlaps*, on the deployment case, Muse reaches the three-hour limit before finishing its procedures. Notice that, when using *30 overlaps*, the *direct* results get closer to the *rn* results. However, regarding the Arctic Tundra use-case, this is a very high number of overlap for an ON, thus not very realistic. On one hand, for *rn*, the *deploy* and *update* durations remain stable for *7*, *15* and *30 overlaps*. On the other hand, for *direct*, the results vary drastically: the maximum variation is 693% (from 812,35s to 6441,27s) for *deploy*, and 869% (from 629,66s to 6100,69s) for *update*.

In all the above experiments, Muse is very slow to complete deployments and updates compared to *direct* while both versions use direct communications between nodes. Our goal in this paper is not to compare tools but to compare deployment and update durations when using direct and indirect communications. However, we observed during our experiments that the Raspberry Pi's CPU was loaded to 100% in the case of Muse. This is not the case for Concerto which has its CPU loaded at 30% maximum during the execution on Raspberry, leading to the observation that Muse puts drastically more stress on the

Upt.Order.		uto_0			uto_1			uto_2		
Version		muse	direct	rn	muse	direct	rn	muse	direct	rn
$Deploy$	ard_0	3737,02	1832,47	257,41	3744,02	1832,52	645,38	3740,68	1830,66	450,39
	ard_1	3746,20	2604,93	267,04	3753,42	2604,83	654,12	3749,90	2602,96	459,12
$Update$	ard_0	1661,14	1035,20	250,72	1661,23	1036,73	831,84	1657,52	1033,60	829,88
	ard_1	1674,74	1036,71	259,88	1674,58	1036,81	841,01	1670,92	1034,97	828,44

TABLE I: Deployment and update durations (in seconds) when varying uptimes' orders: results for Muse, *direct* and *rn* versions, for two random draws of actions duration ard_0 and ard_1.

Nb.Ovlp.		7			15			30		
Version		muse	direct	rn	muse	direct	rn	muse	direct	rn
$Deploy$	ard_0	$Notfinished$	6090,51	274,50	3737,02	1832,47	257,41	2280,93	656,48	240,05
	ard_1	$Notfinished$	6441,27	266,99	3746,20	2604,93	267,04	2577,25	812,35	248,74
$Update$	ard_0	6465,42	2623,03	250,72	1661,14	1035,20	250,72	1643,79	269,64	233,21
	ard_1	6479,01	6100,69	259,86	1674,74	1036,71	259,88	1656,98	629,66	242,34

TABLE II: Deployment and update durations (in seconds) when varying number of overlaps: results for Muse, *direct* and *rn* versions, for two random draws of actions duration ard_0 and ard_1.

CPU than Concerto when executing deployments and updates. We did not dig further into this result but Muse is a declarative tool that compares the current and targeted states and infers the set of actions within the procedures while running them, while Concerto takes these actions as an input (an inference tool exists but is external [12]).

These results give an experimental answer to our research questions. On **RQ1**, the deployment and update durations when using indirect communications are always shorter than using direct communications, with a reduction going from 13% to 96%. On **RQ2**, we see that *indirect* is sensitive to the uptime order with a maximum variation of 224% across the three different uptime orders, while *direct* and Muse are sensitive to the number of overlaps. The maximum variation for the number of overlaps for *direct* is 869%.

VI. Related work

This section is divided into three parts: (1) our related work regarding particular CPS deployment for scientific observations; (2) a related work on delay-tolerant-networks and intermittent computing that can be compared to our case study; and (3) our related work on CPS reconfiguration.

A. Observatory CPS

Cyber-physical systems are used in a wide variety of use-cases such as health care [1], agriculture [2], monitoring the environment [3]. However, very few environments impose hard constraints on nodes composing the CPS. Notably, in [13], authors present a design and architecture to use IoT for animal ecology. Nodes are accessible through satellite or base stations and can potentially have solar panels to refill their batteries. Configuration of observation nodes should be done manually by experts of the domain, dedicating each node to a specific observation. In [14], authors study vegetation using an IoT deployment. Again, nodes are fully accessible through common networks and batteries can be replenished using solar panels. No possibility of reconfiguration is presented. In [15], authors consider that ocean observatories are deployed in a "harsh environment". The solution promises automatic integration of nodes in the existing system, by being able to configure the network of nodes, dynamically. However, most nodes in this observatory (except autonomous underwater vehicles) seem to be accessible from a network (at least satellite) and have access to a reliable energy source (as they come with large batteries when they are not connected by wire to the electrical grid). In a sensibly similar hard-to-reach environment as the arctic tundra, in [16] authors contributed a stand-alone geo-monitoring system for harsh environments, in the Alps. In this deployment, observation nodes form a wireless sensor network where simple sensor measurements are done on the local node. At least one node has access to a base station through a GPRS network. Sensor nodes are pre-configured before deployment and can duty cycle to save energy with synchronization on wake-up.

Compared to the combined constraints seen and imposed by the Arctic Tundra, the environments previously discussed can be considered relaxed. In particular, nodes composing the observation system do not need to both: sleep most of the time; and be adapted through time (i.e., be subject to dynamic deployments and updates). Furthermore, none of these works study how direct and indirect (i.e., with an RN) communications can affect the duration of the procedures.

B. Delay-tolerant-networks and intermittent computing

Works toward delay-tolerant networks (DTNs) and intermittent computing do not specifically address the deployment and update problems in a system with collaborative nodes. However, these domains handle specific constraints on network and computing with sporadic availability that could be useful when facing sleeping nodes in the Arctic Tundra.

Regarding DTNs first, in [17] the authors propose a survey for the routing strategies to propagate messages from sources to destinations in networks with scarce communications. Works mentioned in the survey consider the message-passing aspect between nodes, but not the actual coordination when deploying and updating multiple services across the

network. Of course, we could have used some of these results to exchange messages between ONs, this could be the subject of future work. We have instead favored a contribution at the application level, thus being much more specific to our case study, instead of using a generic protocol-oriented approach. Indeed, in DTNs researchers are working mostly on new network protocols rather than at the application level of the OSI model. The work most relevant to what is presented here is in [18]. The authors aim to adapt a protocol for distributed transactions to make it suitable for DTNs. A coordinator node is introduced to store, carry, and forward protocol messages during the transaction. This contribution is comparable to our indirect vision with one clique of ONs communicating through one RN. In [19], authors surveyed the challenges of intermittent computing including energy harvesting, power failure, data consistency, programming support, and distribution. However, it appears that the current challenges addressed in this field focus on the hardware and software design of an "intermittent" device (e.g., ensuring execution progress, maintaining a coherent memory state) rather than on the coordination problem between multiple "intermittent" devices.

C. Reconfiguring CPS

In the literature when dealing with deployments and updates, one interesting domain is the domain of component-based reconfigurations [6]. However, as far as we know, only a few contributions offer a language able to model and execute deployments and updates in a decentralized manner. Most of the time the reconfiguration programs are centralized, and a single authority handles the execution of actions by sending them to the nodes. This is notably the case in many decentralized orchestration tools based on Kubernetes [20]–[22], but also in more generic reconfiguration languages [6], [8], [23]. Some generic reconfiguration frameworks or reconfiguration languages are adapted to (or specifically designed for) IoT and CPS systems. In [24], [25] an extension of the well-known component model BIP with reconfiguration capabilities is presented, namely DR-BIP for dynamic reconfigurable BIP. The concepts introduced in DR-BIP make it a suitable candidate to reconfigure various IoT systems made of components organized in different motifs (topologies). Each motif is able to reconfigure locally and to apply coordinated reconfiguration rules with other motifs which makes this solution close to a decentralized reconfiguration. However, communications are not directly handled by the model and are not discussed. In [26], [27] is presented the tool R-Mozart to design and reconfigure IoT applications while verifying some formal properties of their behaviors. However, unlike DR-BIP the execution of a reconfiguration with R-Mozart is centralized with a deployment manager thus not adapted to more autonomous agents in a CPS system, particularly with intermittent connectivity and sleeping nodes. In [28] an extension of the component model SCA is presented to handle dynamic QoS of IoT applications, thus their reconfiguration. As for R-Mozart, this solution is centralized with the concept of a Middleware manager, responsible for sending execution orders to nodes' daemons.

Another related domain of the literature is the concept of choreography, i.e., decentralized deployments, and updates. The literature on choreographies is, as far as we know, quite small and none of the existing contributions tackle the specific case of CPS with both hard energetic and networking constraints such as the Arctic Tundra [9]–[11].

Overall, none of the above reconfiguration or choreography approaches have considered IoT and CPS systems with constraints as hard as in the Arctic Tundra use case (energy and network). In particular, none of them have explicitly studied reconfigurations with sleeping nodes, and consequently how direct (i.e., without an RN) and indirect (i.e., with an RN) communications could be leveraged to speed up reconfigurations.

VII. Conclusion

Cyber-physical systems deployed in scarce resource environments, like the Arctic Tundra, face difficult conditions. Nodes deployed in such environments are forced to sleep most of the time to save energy and rarely overlap. In such conditions, the coordination of a service deployment or update might take a long time to complete. The considered CPS has a few nodes (RNs) equipped with more powerful batteries. These nodes could be leveraged to relay messages, allowing asynchronous and indirect communications.

In this paper, we conduct an analytical and experimental study to understand which parameters of sleeping scenarios influence the time needed for deployment and update to complete, using either direct (i.e., without an RN) or indirect (i.e., with an RN) communications. Our results show that indirect communications always perform better than direct communications. Moreover, we show that while the uptime order influences the durations when using indirect communications, the number of overlaps has only consequences on the procedures using direct communications.

In future work, we plan on measuring the energy cost of using RNs for communications instead of direct communications to have a better understanding of the existing trade-off between time and energy consumption. Moreover, in this paper, one clique is considered with one node to carry asynchronous communications. We plan to extend this work to more cliques by using a decentralized broker (Zenoh[6]) on RNs and to measure the energy consumption of such a solution. Finally, it could be interesting in future work to use protocol-oriented contributions of DTNs to solve communications between sleeping nodes at the protocol level and compare results to our application-level solution.

Acknowledgment

The DAO project is supported by the Research Council of Norway (RCN) IKTPluss program, project number 270672.

[6]https://zenoh.io/

REFERENCES

[1] T. e. a. Adame, "Cuidats: An rfid–wsn hybrid monitoring system for smart health care environments," *Future Generation Computer Systems*, 2018.

[2] C.-R. e. a. Rad, "Smart monitoring of potato crop: a cyber-physical system architecture model in the field of precision agriculture," *Agriculture and Agricultural Science Procedia*, 2015.

[3] I. Rodero and M. Parashar, "Data cyber-infrastructure for end-to-end science: Experiences from the nsf ocean observatories initiative," *Computing in Science & Engineering*, 2019.

[4] O. A. Issam RAIS, Loic Guegan, "Impact of loosely coupled data dissemination policies for resource challenged environments," in *2022 IEEE/ACM 22st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022.

[5] L. S. Michalik, L. Guegan, I. Raïs, O. Anshus, and J. M. Bjørndalen, "Loralite: Lora protocol for energy-limited environments," in *2022 30th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2022, pp. 73–80.

[6] H. Coullon, L. Henrio, F. Loulergue, and S. Robillard, "Component-based distributed software reconfiguration: A verification-oriented survey," *ACM Comput. Surv.*, 2023.

[7] M. Chardet, H. Coullon, and C. Pérez, "Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto," in *CCGrid 2020 - 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 2020.

[8] M. Chardet, H. Coullon, and S. Robillard, "Toward safe and efficient reconfiguration with concerto," *Science of Computer Programming*, 2021.

[9] D. Sokolowski, P. Weisenburger, and G. Salvaneschi, "Automating serverless deployments for devops organizations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[10] H. Herry, P. Anderson, and M. Rovatsos, "Choreographing configuration changes," in *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*, 2013.

[11] K. e. a. Wild, "Decentralized cross-organizational application deployment automation: An approach for generating deployment choreographies based on declarative deployment models," in *Advanced Information Systems Engineering*, 2020.

[12] S. Robillard and H. Coullon, "SMT-Based Planning Synthesis for Distributed System Reconfigurations," in *FASE 2022 : 25th International Conference on Fundamental Approaches to Software Engineering*, 2022.

[13] S. e. a. Guo, "The application of the internet of things to animal ecology," *Integrative zoology*, 2015.

[14] N.-S. Kim, K. Lee, and J.-H. Ryu, "Study on iot based wild vegetation community ecological monitoring system," in *Seventh International Conference on Ubiquitous and Future Networks*, 2015.

[15] S. Lin, F. Lyu, and H. Nie, "An automatic instrument integration scheme for interoperable ocean observatories," *Sensors*, 2020.

[16] I. Talzi, A. Hasler, S. Gruber, and C. Tschudin, "Permasense: investigating permafrost with a wsn in the swiss alps," in *Proceedings of the 4th workshop on Embedded networked sensors*, 2007.

[17] E. Jones and P. Ward, "Routing strategies for delay-tolerant networks," 2006.

[18] I. Carreras, J. Rana, and L. Telesca, "Coordination protocol for financial application in delay tolerant networks," in *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08)*, 2008.

[19] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent Computing: Challenges and Opportunities," in *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, B. S. Lerner, R. Bodík, and S. Krishnamurthi, Eds. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[20] M. A. e. a. Tamiru, "mck8s: An orchestration platform for geo-distributed multi-cluster environments," in *2021 International Conference on Computer Communications and Networks (ICCCN)*, 2021.

[21] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with kubeedge," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018.

[22] L. L. Jiménez and O. Schelén, "Docma: A decentralized orchestrator for containerized microservice applications," in *2019 IEEE Cloud Summit*, 2019.

[23] R. e. a. Di Cosmo, "Aeolus: a component model for the Cloud," *Information and Computation*, 2014.

[24] M. B. J. S. Rim El Ballouli, Saddek Bensalem, "Dr-bip - programming dynamic reconfigurable systems," Tech. Rep., 2018.

[25] R. E. Ballouli, S. Bensalem, M. Bozga, and J. Sifakis, "Programming dynamic reconfigurable systems," *Int. J. Softw. Tools Technol. Transf.*, 2021.

[26] F. Durán, A. Krishna, M. Le Pallec, R. Mateescu, and G. Salaün, "R-MOZART: A Reconfiguration Tool for WebThings Applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2021.

[27] F. Durán, A. Krishna, M. Le Pallec, R. Mateescu, and G. Salaün, "Seamless Reconfiguration of Rule-Based IoT Applications," in *SEAMS 2021 - 16th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2021.

[28] A. Agirre, J. Parra, A. Armentia, E. Estévez-Estévez, and M. Marcos, "Qos aware middleware support for dynamically reconfigurable component based iot applications," *Int. J. Distributed Sens. Networks*, 2016.