UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Study of the energy consumption and duration of a Cyber-Physical System reconfiguration in the Arctic Tundra: from experiments on real infrastructure to extensive simulations

Antoine Omond

Philosophiae Doctor in Science, May 2025

Cover photo: Ny-Ålesund, Svalbard, Brage B. Hansen (https://www.coat.no/en/Climate)

Abstract

Cyber-physical systems deployed in scarce-resource environments like the Arctic Tundra face extreme conditions. Nodes deployed in such environments have to carefully manage a limited energy budget, forcing them to alternate long sleeping periods and brief uptime periods. While sleeping, nodes save energy but cannot communicate with other nodes. During uptimes, nodes can collaborate for data exchanges or computations by providing services to other nodes. Due to lack of network infrastructure, nodes have to rely on peer-to-peer connections to communicate. Nodes are able to communicate only during uptime, when their uptime periods overlap. When dealing with short and non-synchronised uptimes, opportunities for communication are low. Due to harsh weather, physical access to the nodes deployed in the field is prevented for long periods (e.g., 6 months during winter). During these periods, nodes have to be autonomous. Nodes have to adapt to their surrounding environment by reconfiguring their systems. Due to collaboration, nodes of the CPS form a distributed system, where services can be coupled between nodes.

In a broad sense, dynamic reconfiguration of distributed systems (software, services, infrastructure) has been widely studied in the industry and academic literature. Distributed systems are characterised by their modular architecture. A modular system can be reconfigured by modification of individual modules and their composition (e.g., services). For instance, a system composed of clients connected to a database can be reconfigured by adding/removing and connecting/disconnecting clients to the database.

Solutions dealing with reconfiguration often consider a centralised paradigm. In such case, one or multiple central entities are responsible for coordinating the reconfiguration of the whole system. This approach may not be suited for contexts where nodes are most of the time isolated. When dealing with isolated nodes, a central entity may be unreachable most of the time. In such cases, a decentralised reconfiguration paradigm might be better suited. In this paradigm, each node has the ability to reconfigure itself and coordinate its reconfiguration with others. To this end, nodes should have access to a common reconfiguration solution allowing different and potentially heterogeneous nodes to coordinate their reconfiguration.

This manuscript proposes to study the decentralised reconfiguration of a Cyber-Physical system with sleeping nodes. It provides a solution to define and coordinate the execution of decentralised reconfiguration programs between sleeping nodes. Based on this solution, it aims at extensively studying the performances in terms of energy consumption and duration of such reconfiguration according to the Arctic Tundra characteristics.

To this end, this manuscript provides the following contributions. First, the design and implementation of a decentralised reconfiguration solution, named Concerto-D, allowing to coordinate a reconfiguration between sleeping nodes. Second, a validation of the capabilities of Concerto-D on different reconfiguration scenarios and an evaluation of its performance in time compared to another solution from the literature. Third, a study of the impact of non-synchronised sleeping nodes on reconfiguration duration. Fourth, a study of a trade-off between nodes' uptime duration and reconfiguration duration. Fifth and sixth, an evaluation and study of the energy consumption and duration of a decentralised reconfiguration according to different parameters relevant for CPSs deployed in the Arctic Tundra.

Contents

Chapter 1

Introduction

Cyber-physical systems (CPS) are systems where physical instruments are combined with software modules to achieve smart observations, computations, and decision-making. CPSs are typically composed of sensors to collect data, actuators, and have computing and networking capabilities. Nodes of the CPS may collaborate to provide additional services such as data aggregation, analytics, etc. For these reasons, CPSs are well suited for use cases involving large deployments for data collection and treatment directly from the environment [1].

Nodes of a CPS can be very small and non-intrusive (i.e., not very visible and not disturbing the local environment), which allows them to be deployed in different kinds of environment [2]. When deployed in urban environments such as in homes, offices, and public areas, nodes benefit from the surrounding energy and network infrastructure: access to a regular energy supply, wireless connectivity within the CPS and from outside, etc.

Wild environments, such as protected natural areas, often have fewer resources than urban environments. They are often located far from energy and network infrastructures, making it harder for nodes to have regular energy supply and connectivity [3, 4]. An extreme example of such wild area is the Arctic Tundra: a protected natural area, difficult to access, with scarce resources and a very limited energy supply. This area is currently observed by scientists of the COAT as it is one of the most sensitive areas to climate change [5]. Most of the current observations are coarse-grained satellite-based images [1]. To better understand the impact of climate change on the Arctic Tundra, scientists could benefit from a CPS deployed in sites of interest to make in-situ observations.

The Arctic Tundra is a difficult environment to monitor. It is a very large area, and has difficult weather conditions. Covering its size requires a large number of nodes on multiple observation sites. Due to strict regulations and a lack of roads, navigation in the area is difficult. Nodes deployed on the field must be small, non-intrusive, and forced to rely on batteries. Harsh weather prevents physical access to observation sites for long duration (from 6 months to a whole year). For this reason, nodes may stay on the field for long periods to gather long-term and in-situ observation. The absence of eligible energy harvesting mechanisms (e.g., sun, wind, etc.) prevents nodes from accessing a regular and reliable energy supply [6]. Network coverage is rare or absent. Nodes deployed on the field are most of the time isolated from a connectivity perspective from the external world. Communication between nodes on the field must rely most of the time on peer-to-peer connections [7].

Due to extreme energy constraints, the energy budget of nodes should be carefully

[1] Science Plan for COAT

managed. Nodes may enter long sleeping periods to save energy, and wake up for short periods of activity. This includes activities such as observation, but also communication with other nodes. While sleeping, nodes save energy but are not able to perform any other activity. The frequency of uptimes and their duration depends notably on the energy budget and the activities that the node needs to perform. These uptimes may not be synchronised, especially when nodes have different observation schedules, which is most of the time the case [8].

Due to the characteristics of the Arctic Tundra, nodes of the CPS must be autonomous. An autonomous CPS should be able to function without physical or remote intervention from humans for long periods and automatically adapt to external events. Adaptation capabilities require the node to be able to monitor these events and reconfigure its system accordingly, such as collecting and processing data from an observed event, handling node failures, etc [9]. Nodes adapt notably by reconfiguring their services and their connections (e.g., sensor activation/deactivation, disconnection from failed node, etc.). Additionally, in some cases, having the ability to quickly reach a new configuration can be beneficial for the nodes (e.g., quickly going back to sleep, treating a short-lived event, etc.). Due to collaboration, nodes of the CPS form a distributed system, where services can be coupled between nodes. Collaboration allows heterogeneous nodes to benefit from the resources and capabilities of each other. For instance, nodes with high energy budget or computing power can provide their resources to treat data extracted from other nodes. Coordinating the reconfiguration between collaborative nodes is essential to prevent failure.

In a broad sense, dynamic reconfiguration of distributed systems (software, services, infrastructure) has been widely studied in the industry [10] and academic literature [11]. Distributed systems are characterised by their modular architecture. A modular system can be reconfigured by modification of individual modules and their composition (e.g., services). For instance, a system composed of clients connected to a database can be reconfigured by adding/removing and connecting/disconnecting clients to the database.

Solutions dealing with reconfiguration often consider a centralised paradigm. In such case, one or multiple central entities are responsible for coordinating the reconfiguration of the whole system. This approach may not be suited for contexts where nodes are most of the time isolated. When dealing with isolated nodes, a central entity may be unreachable most of the time. In such cases, a decentralised reconfiguration paradigm might be better suited. In this paradigm, each node has the ability to reconfigure itself and coordinate its reconfiguration with others. To this end, nodes should have access to a common reconfiguration solution allowing different and potentially heterogeneous nodes to coordinate their reconfiguration.

When dealing with sleeping nodes, the execution and coordination of reconfiguration between nodes can only be done during uptime periods. When considering non-synchronised uptime periods between nodes, such a coordination can take a long time to converge and impose a significant energy consumption overhead on nodes. In the literature, reconfiguration and its performances in the context of sleeping nodes have not yet been studied. This manuscript aims at studying reconfiguration in such a context and its performances in terms of duration and energy consumption.

Due to scarce connectivity between nodes, this manuscript also includes a study of a potential leverage to alleviate the difficulty of nodes to communicate. This leverage is a special type of node that uses its energy supply to help other nodes to communicate. This node, called Relay Node, is considered to have sufficient energy supply to be awake more frequently and for longer duration than the other nodes. This capability allows the

Relay Node to be awake at the same time as its neighbouring nodes. It can be used as an intermediary by other nodes to store messages from senders while recipients are sleeping, and forward messages to recipients when they wake up. Such node may not be available in all observation sites. In this manuscript, the performances of reconfiguration in terms of energy consumption and duration are studied with and without the Relay Node.

Different factors may have an impact on the energy consumption of such a reconfiguration. First, due to the lack of network infrastructure, nodes embed wireless communication capabilities and have to communicate in a peer-to-peer manner using multi-hop communications. A peer-to-peer connection can be enabled when nodes are located close to each other with no obstacle preventing communication. Second, the network topology formed by nodes that are able to communicate may vary according to observation sites or environmental conditions. When dealing with collaborative nodes, the network topology can have a large impact on the coordination, especially when nodes are located at a high hop count. Finally, when considering sleeping nodes, communication between nodes during coordination may also be difficult. Studying the energy consumption of such reconfiguration can help planning future deployments of such CPSs in harsh environments such as the Arctic Tundra. This manuscript aims at studying the energy consumption and duration of a decentralised reconfiguration between non-synchronised sleeping nodes. Such a reconfiguration needs to be coordinated without relying on central authority.

Research challenges The first challenge is to design and implement a solution that would allow to execute and coordinate a decentralised reconfiguration between sleeping nodes. Each node of the CPS should be able to reconfigure its local system, and coordinate its reconfiguration with other nodes. Such a solution should provide the ability to execute a reconfiguration quickly.

The second challenge is to evaluate the energy consumption and duration of the coordination of a reconfiguration according to different scenarios. Such an evaluation can be done using theoretical study, simulation, or prototyping. Theoretical study models the system, its constraints, and the reconfiguration process through mathematical definitions, equations, etc. Simulation relies on validated simulators that model the different parts of the system and perform experiments in simulated environments. Prototyping allows to study the system on physical hardware and environment (e.g., Raspberry Pi). In cases where experiments in real environments such as the Arctic Tundra is not possible, the conditions of the target environments can be emulated (e.g., sleeping nodes, limited bandwidth, etc.) [12].

Finally, the third challenge is to study values obtained during the evaluation. This study should highlight the impact of sleeping nodes and lack of connectivity on reconfiguration duration and energy consumption. This study should include relevant parameters for CPSs deployed in the Arctic Tundra, such as the number of nodes, duration of each uptime period, direct communication between nodes or usage of a Relay Node, etc.

Contributions To deal with these challenges, the work of this manuscript includes the following contributions:

- The design and implementation of a decentralised reconfiguration solution, named Concerto-D, allowing to coordinate a reconfiguration between sleeping nodes;
- A validation of the capabilities of Concerto-D for different reconfiguration scenarios;

- A study of the impact on reconfiguration duration of sleeping nodes and scarce connectivity;
- A study of a trade-off between nodes' uptime duration and reconfiguration duration when using a Relay Node;
- A study of the energy consumption due to decentralised reconfiguration on sleeping nodes with scarce connectivity;
- A study of the impact on energy consumption of different factors relevant for a CPS deployed in the Arctic Tundra environment.

Publications This work led to three publications in conferences and one submission in journal.

International conferences

- Antoine Omond, Hélène Coullon, Issam Raïs, Otto Anshus. Leveraging Relay Nodes to Deploy and Update Services in a CPS with Sleeping Nodes. In 2023 IEEE International Conferences on Cyber, Physical & Social Computing (CPSCom) (pp. 532-539).
- Antoine Omond, Issam Raïs, Hélène Coullon. Evaluating the energy consumption of adaptation tasks for a CPS in the Arctic Tundra. In 2023 IEEE International Conferences on Green Computing & Communications (GreenCom) (pp. 681-688).
- Jolan Philippe, Antoine Omond, Hélène Coullon, Charles Prud'Homme, Issam Raïs. Fast Choreography of Cross-DevOps Reconfiguration with Ballet: A Multi-Site OpenStack Case Study. In 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).

Submission to journal

- Antoine Omond, Issam Raïs, Hélène Coullon. Quantifying the Impact of Adaptation Tasks on the Energy Consumption of a CPS in the Arctic Tundra. In 2025 Ad Hoc Networks. [Under review]

Organisation of the manuscript This manuscript is organised in seven chapters following this introduction. Chapter 2 introduces the context and use case of this work. Chapter 3 presents the state of the art in two parts. In the first part, different patterns and solutions for adaptation are presented. In the second part, observatory CPSs and energy consumption due to reconfiguration are presented. Chapter 4 presents Concerto-D. Chapter 5 presents a study on reconfiguration duration between multiple sleeping nodes. Chapter 6 studies energy consumption of such reconfiguration. Chapter 7 concludes this work and gives perspectives.

Chapter 2

Reconfiguration of autonomous Cyber-Physical Systems with sleeping nodes

Contents

Autonomous CPSs must adapt to external events by reconfiguring their systems. When nodes of the CPS must collaborate, services are provided and used by the nodes, creating a distributed system. Reconfiguration of distributed systems has to be coordinated to prevent failure. In the literature, different patterns exist for such a reconfiguration. Decentralised patterns can be relevant in contexts where nodes are most of the time isolated. This chapter presents a use case featuring such a context. Additionally, this chapter aims at introducing definitions and concepts related to autonomous CPSs, their adaptation/reconfiguration, and the potential energy consumption overhead due to reconfiguration activities.

2.1 Autonomous Cyber-Physical Systems

This section introduces definitions and concepts regarding autonomous Cyber-Physical Systems and presents a reference model to represent adaptation capabilities in such systems. This section ends with an example to illustrate the different phases of an autonomous system using this model.

Distributed system A distributed system is composed of inter-connected hardware and software modules. The connection between hardware can be done using a dedicated or shared network infrastructure. Modules communicate and coordinate their activities by exchanging messages through the network. Nodes of the system can be located far from each other up to thousands of kilometres (e.g., Internet) or close together down to a few centimeters (e.g., datacenter). Distributed systems may include nodes with heterogeneous resources in terms of computing power, network bandwidth, and energy consumption [13].

Software module Distributed systems are characterised by their modular architecture. These modules can represent hardware or software elements of the system. This manuscript focuses on the software part. Modules represent the different pieces of software hosted on the system and their interactions. A module can represent application software such as services. This includes servers, databases, data producer/consumer, but also repositories, registries, etc. A module can also represent libraries or other dependencies for the installation and configuration of other modules [14].

Cyber-Physical System A Cyber-Physical System (CPS) is a distributed system that provides seamless integration between software modules and physical processes (e.g., sensing, actuation). Inside this system, these processes continuously interact with each other to combine (i) computing, storage, networking devices and their software (i.e., Cyber) and (ii) sensors, actuators and other devices allowing interaction with the physical world (i.e., Physical). Data collected from the physical world is used by software, which in turn interacts with the physical devices, such as sensors and actuators.

Nodes of the CPSs are typically equipped with heterogeneous hardware and software capabilities. Nodes collaborate by sharing their capabilities with other nodes. Some nodes of the CPS can be producers of data while others consume this data. For instance, sensor nodes deployed in area of interest, reporting data to other nodes with higher computing capabilities for analytics or other computation tasks. Relay nodes can be deployed to ensure connectivity between the different parts of the system [1].

Observatory CPS The term observatory CPS has few occurrences in the literature. The term is often worded as an observatory, which is an implementation or an application of a CPS [15]. In this manuscript, the term observatory CPS refers to a CPS deployed in-situ with the mission of collecting and reporting data from the environment. An observation includes the collection of samples from the physical world (e.g., using sensors), the treatment of this data (e.g., filtering, analytics, or refining) and the reporting of results to scientists. For instance, using proximity cameras to take images from the local wildlife, processing such image samples to recognise animals, and reporting results back to scientists [7].

Use cases for observatory CPSs include smart-home, smart-city, agriculture, healthcare, natural environments, etc. The features provided by CPSs, such as the integration

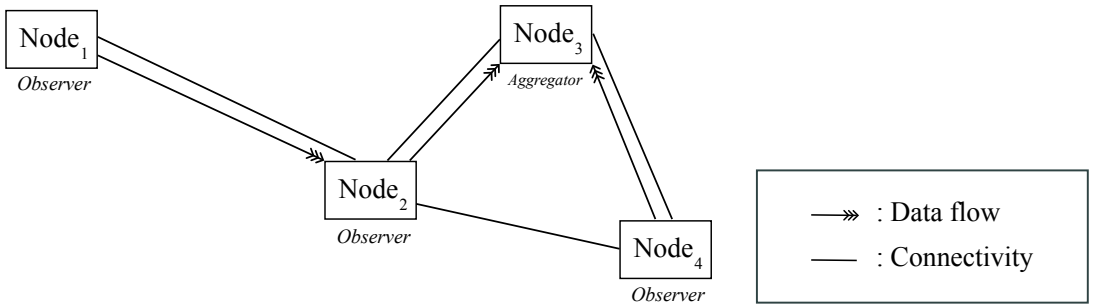


Figure 2.1: Peer-to-peer network of collaborative nodes

of heterogeneous hardware (e.g., sensors, microcontroller, single-board computers) of different sizes, make such system well suited for observation activities. Nodes can be as small as microcontroller or RFID chip. To facilitate their deployment, nodes of the CPS often rely on wireless communication. This makes CPSs easier to deploy. Finally, CPS can include mobile nodes. For instance, such nodes can have RFID chips attached to animals living in specific areas [16].

The environment where the CPS is deployed has a direct impact on the type of nodes that can be deployed or the type of application possible to run on the nodes. When deployed in urban environments (e.g., smart-city, smart-home), nodes benefit from abundant energy supply and have access to the surrounding network infrastructure. In environments that are far from energy sources (e.g., forest, mountains, tundra), nodes are forced to rely on embedded resources. Such constraints may restrict the size and resources available to the nodes and therefore to the application running on the system. Compared to applications where nodes have access to large amount of resources (e.g., Cloud), nodes deployed in observation sites may be limited to lightweight applications [1].

Peer-to-peer network In the literature, the term peer-to-peer network refers to a type of distributed network where the participants are equally privileged and share a part of their own hardware resources (processing power, storage capacity, network link capacity). These resources can be provided to other nodes. They are accessible by other peers directly, without passing through intermediary entities. A peer-to-peer network, or a network where nodes rely on their neighbours, can be used to create a network without relying on external network infrastructure, or removing single point of failure in the system [17].

In this manuscript, the term peer-to-peer is used solely to define a network created without relying on external network infrastructure. Each node part of the network provides its network capabilities (i.e., radio) to receive and forward packets to other nodes in the network. In this network, any node can be used to forward messages to any other node. However, nodes composing the network may host different services and have heterogeneous capabilities.

An example of peer-to-peer network is presented in Figure 2.1. In this figure, nodes use wireless radio technology to communicate and create a peer-to-peer network. Based on their radio range and field characteristics (e.g., no obstacle preventing communication), node 1 is able to communicate only with node 2. Nodes 2, 3, and 4 are all able to communicate. Communication between node 1 and nodes 3 or 4 is enabled through node 2. Nodes collaborate by sharing their properties with other nodes. For instance, node 3 may have access to more computing power than other nodes. Node 3 hosts a service (called *Aggregator* for the example) collecting data from nodes 1, 2, and 3.

Network topology In the rest of this manuscript, nodes are considered connected through a peer-to-peer network. In this manuscript, nodes that are in radio range, with no obstacle preventing communication are called neighbours. The network topology is defined by all neighbours. Neighbours are nodes that are able to create a peer-to-peer connection to communicate. Communication between nodes in the topology is enabled by forwarding messages between neighbours until a recipient is reached.

Information can be extracted from the network topology, such as the hop count required to reach a recipient from any node, and the number of neighbours of each node. For instance, in Figure 2.1, $Node_2$ has three neighbours, while $Node_1$ has only one. $Node_1$ is located at two hops from $Node_3$ and $Node_4$, and at one hop from $Node_2$.

In the network topology, the positions of collaborative nodes impact the hop count required to communicate. In some cases, collaborative nodes can be located close to each other (i.e., single or few hop count between nodes). In other cases, communication between nodes may require high hop count to reach a recipient. The network topology has a large impact on the energy consumption of the system, as communication between nodes is a major source of energy consumption [18].

Autonomous systems and CPSs In this manuscript, the term autonomous is used to define a system that is able to (i) function without physical or remote intervention from humans for long periods (i.e., from months to years) (ii) automatically adapt to external events [19]. In this manuscript, an autonomous CPS is a system with both these properties.

Adaptation capabilities of CPSs An adaptation is the process of adjusting the behaviour of a system according to external events to match pre-defined objectives [19]. It can be used by autonomous systems to automatically adapt to external events. The behaviour of the system defines the activities of the system for which it has been deployed (e.g., data collection, treatment, etc). The objectives include staying within thresholds such as power consumption, CPU usage, number of failures, latency, etc. These objectives ensure that the system provides sufficient quality of service, sustains for long periods, correctly handles different workloads, etc.

CPSs usually have the ability to reconfigure their cyber and physical parts. In CPSs, changes in the cyber part include changing software modules and their interactions (e.g., software installation, update, connections between modules, etc.). Changes in the physical part include the ability to interact with sensors. This can be done by using actuators. Actuators are physical entities that are able to physically interact with the CPS (e.g., turn sensors on and off). The state of the physical entities can be changed using actuators (e.g., turning sensors on and off).

Adaptation capabilities require the system to be able to monitor external and internal events, decide which changes to perform according to these events and how to perform them, and finally apply them. To model these different capabilities, it is common to refer to the MAPE-K model.

MAPE-K model The MAPE-K model is an abstraction proposed by IBM to help design autonomous systems [19]. As an abstraction, the model highlights the different phases of an adaptation, separates concerns between these different phases and describes their interactions. It does not represent how adaptation should be conducted on a system, such as concrete actions to execute at each phase of the adaptation.

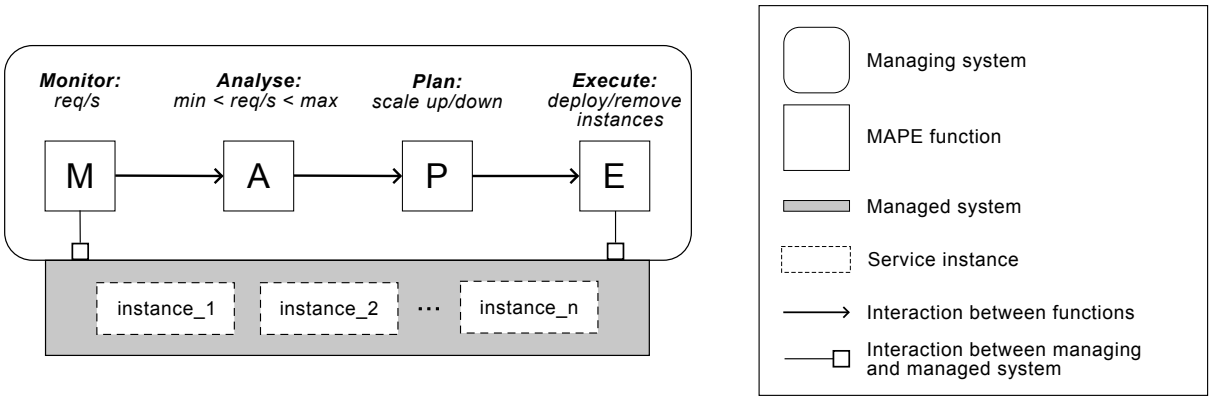


Figure 2.2: Adaptation example of an autoscaler, using the MAPE model from [20]. Knowledge is not represented for simplicity. When the *Execution* (E) phase is completed, execution goes back to the *Monitoring* (M) phase (not represented for simplicity).

The MAPE-K model is composed of four phases (MAPE) that share a common Knowledge (K). The four phases are the *Monitoring* (M), *Analysis* (A), *Planning* (P) and *Execution* (E). In the *Monitoring* phase, data is collected. This data is used in the *Analysis* phase to decide whether an adaptation is necessary. When it is, a plan is created by the *Planning* phase. It is composed of adaptation actions to execute to perform changes. These actions are executed during the *Execution* phase. When the *Execution* phase is completed, the execution goes back to the *Monitoring* phase. The Knowledge (K) represents a set of data or framework shared between MAPE phases. In this section, the Knowledge and how it is used and shared between the different MAPE phases is abstracted away. This is done, as in [20], to simplify the content of the following sections. Only data exchanges between phases not part of Knowledge are considered (e.g., data monitored sent from M to A, actions to execute sent from P to E). For this reason, the term MAPE (without K) instead of MAPE-K is used. The Knowledge will be used again in Section 2.4.

MAPE phases are executed on the managing system. The managing system drives adaptation of the managed system. The managed system represents any application or hardware being adapted. It may be composed of single or distributed nodes hosting single or multiple applications or services. It may also represent system software such as kernel or OS, or attached devices such as sensors. Note that the model does not provide information about where the managing and managed system are hosted. Both may be hosted on the same or different nodes.

Illustrative example of the MAPE model To introduce the MAPE model and each phase, an illustrative example is proposed. This example is a simple autoscaler that adapts a number of instances of a specific service according to the variation of the workload. When the workload increases above a certain threshold, additional instances are deployed to properly treat the additional workload. When the workload decreases below a certain threshold, underutilised instances are removed to free resources. In this example, the workload is evaluated using the number of requests per second that have to be treated by all instances. A simple comparison between this metric and the threshold boundaries is sufficient to decide to scale up or down the number of instances.

Figure 2.2 depicts the design and architecture of this example using the MAPE model extracted from [20]. In this figure, MAPE phases and their interactions are represented.

For simplicity, execution going back from the *Execution* to *Monitoring* phase is not represented. The following presents each MAPE phase and its responsibility.

The *Monitoring* phase is responsible for collecting data from the managed system. Such data can be extracted from the nodes (e.g., resource usage, remaining energy, network throughput/latency) and their software (e.g., software version, failure). It can also be extracted from sensors present on nodes (e.g., heat sensor). These data can be combined and transformed to construct different metrics relevant for the adaptation. Such metrics can be related to the quality of service (i.e., measure of the performance of the system), software obsolescence, energy consumption, etc. In the example, the monitored metric is the number of requests per second received by the managed system. These metrics are used by the *Analysis* phase.

The *Analysis* phase decides whether an adaptation is necessary based on metrics received from the *Monitoring* phase. Using these metrics, the *Analysis* phase computes the current state of the managed system. The *Analysis* phase usually has a set of goals to satisfy. For instance, staying within thresholds or optimising values (e.g., energy consumption, latency). An adaptation is necessary when these goals are not satisfied. In this case, a new target state allowing these goals to be satisfied is computed. In the example, the *Analysis* phase checks whether the number of requests per second is outside specified boundaries. When these boundaries are crossed, a new target state with a sufficient number of instances to handle the frequency of requests is computed. Current and target states are given to the *Planning* phase.

The *Planning* phase creates an adaptation program composed of an ordered or partially ordered set of adaptation actions. This program aims at bringing the managed system from the current to the target state given by the *Analysis* phase. These actions are concrete changes to apply on the managed system (e.g., updating software, actuating sensors). Dependencies may be expressed between actions to coordinate these changes. In the example, the plan aims at deploying/removing instances to meet the target number of instances given by the *Analysis* phase. Actions from the plan are executed during the *Execution* phase.

The *Execution* phase is responsible for the execution of actions given by the *Planning* phase. It ensures that each action is done following the order given by the plan. In the example, the execution conducts the actual deployment or removal of instances on the managed system. More details about this phase are given in Section 3.2.

Interactions between MAPE phases are depicted in Figure 2.2. These interactions represent the order of execution of each phase and the flow of data exchanged between phases. In this example, phases are executed in sequence, from the *Monitoring* to the *Execution* phase. Each phase relies on the output of the previous one. The output may also be given only to the next phase of the loop, such as the number of instances to deploy/remove. All phases may interact with the managed system. However, it is common that the *Monitoring* and *Execution* phases are the only ones interacting with the managed system, as in the example.

The nature of each MAPE phase may influence the architecture of the managing system. Each phase imposes an overhead in terms of resource usage. For instance, the computation of the optimal value of a utility function by the *Analysis* phase may be expensive in terms of CPU usage, while activating/deactivating services may be straightforward. Interactions between MAPE phases located on distributed nodes imply communications. These communications may expose the system to bottlenecks, single point of failure, high latency in case of limited connectivity and heavy resource usage for data exchange. This is notably the case in scenarios where MAPE phases rely on frequent data exchanges,

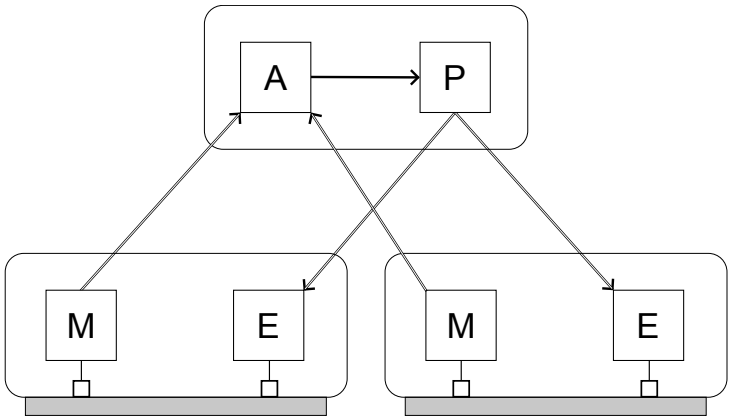


Figure 2.3: Controller/worker pattern from [20]

such as the *Monitoring* and *Analysis* phases.

2.2 Adaptation patterns for autonomous CPSs

In the literature, different MAPE patterns are proposed to fit the constraints of different use cases, including CPSs [9]. These patterns include controller/worker, hierarchical and coordinated control. The following presents each of these patterns, and gives the advantages and drawbacks when applied to CPSs.

Controller/worker In the controller/worker pattern, a single central managing system (i.e., controller) manages a set of distributed nodes (i.e., workers). Figure 2.3 depicts an illustrative example of such a pattern with two workers. Each worker executes the *Monitoring* and *Execution* phases. The *Monitoring* phase provides metrics to the controller. These metrics include health of the software (e.g., heartbeat, software failure) and quality of service (e.g., average load and latency). These metrics can be either queried by the controller or reported by the workers. These metrics are used by the controller to take decision globally about the state the system should reach (i.e., the *Analysis* phase). It then globally computes the reconfiguration program to reach this state (i.e., the *Planning* phase). This plan is then distributed to workers. Each worker is responsible for the execution on its managed system of actions given by the plan. The controller node can be informed about the success or failure of the execution of each action.

Controller and worker nodes can be located in different environments. For instance, worker nodes can be small sensor nodes deployed in observation sites, while controller nodes are located in the Cloud. This pattern is suited for use cases where latency and bandwidth between controller and worker nodes are not an issue. For instance, in smart city applications, sensor nodes can benefit from urban network and energy infrastructure. Authors in [21] manage software of distributed sensor nodes deployed in smart city. The controller node is located in the Cloud and collects data from software deployed on nodes. Software can be deployed, updated, and removed by the controller. Thanks to the network infrastructure surrounding the system, connectivity between worker sensor nodes and the controller is most of the time available.

The controller/worker is a simple architectural pattern, allowing a single controller to be responsible for the decision regarding the reconfiguration of multiple, distributed workers. However, as the controller and workers are located on different nodes, commu-

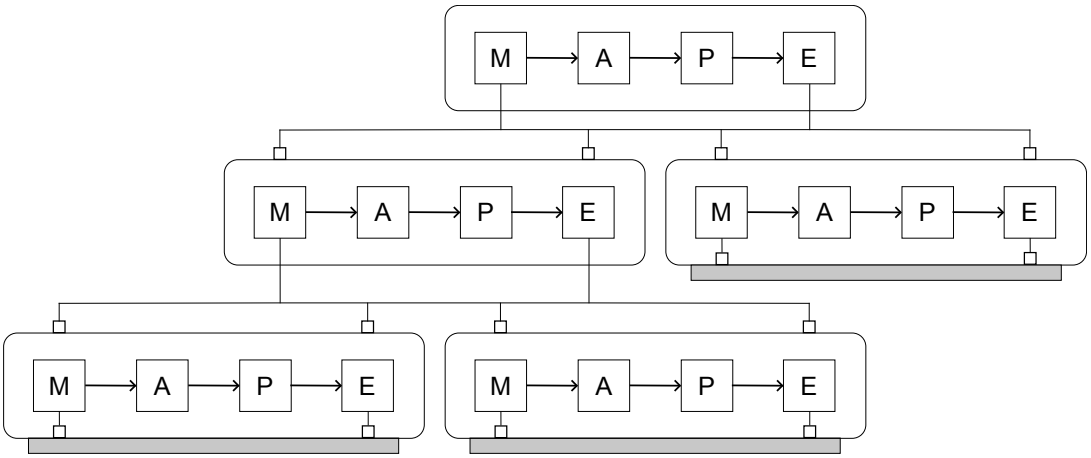


Figure 2.4: Hierarchical pattern from [20]

nication between controller and worker nodes may be difficult in environments with high latency. Such latency may be prohibitive when the decision has to be taken quickly. To tackle this problem, the ability to take decision can be directly given to nodes, while still keeping a central managing system with a global view of the system.

Hierarchical In the hierarchical pattern, the managing system is composed of multiple layers. Figure 2.4 provides an example of such a pattern with three layers. Each parent layer communicates with its child layer. Each child reports data or collaborates with its parent. Compared to controller/worker, this pattern is more flexible. Child layers can take local or latency-sensitive decisions. Parent layers can take decision for all child layers.

For instance, authors in [22] designed a system resilient to network partition between a parent layer in the Cloud and child layers at the edge. The parent and child layers are part of a single network created using a dedicated protocol. A replicated database located at the parent and child layers stores the state of the system and instructions pushed by the parent layer. This design aims at providing resilience to the system when network partition occurs between parent and child layers. When network partition occurs, the edge nodes can still work on their own and take decisions locally based on database content. When network partition is resolved, nodes from child layers pull new instructions from the parent and the parent pulls changes applied on the system during partition from child layers.

Hierarchical pattern has the benefit of being flexible while having simple pattern. Child layers cannot communicate directly between each other, which keeps the flow of information only between the parent and children. The hierarchical pattern still relies on central entity that has a global view of the system. Nodes must refer to this central entity to ensure long-term functioning. When network partitions between the parent and child layers frequently occur, latency between decision and execution of changes may be significant. As for controller/worker pattern, this may be prohibitive when decision has to be taken quickly.

Coordinated control In the coordinated control pattern, managing systems and MAPE phases are fully decentralised. Each managing system may interact with others without relying on a central authority. Figure 2.5 depicts an example with two interacting managing systems. In the coordinated control pattern, each MAPE phase hosted on a managing

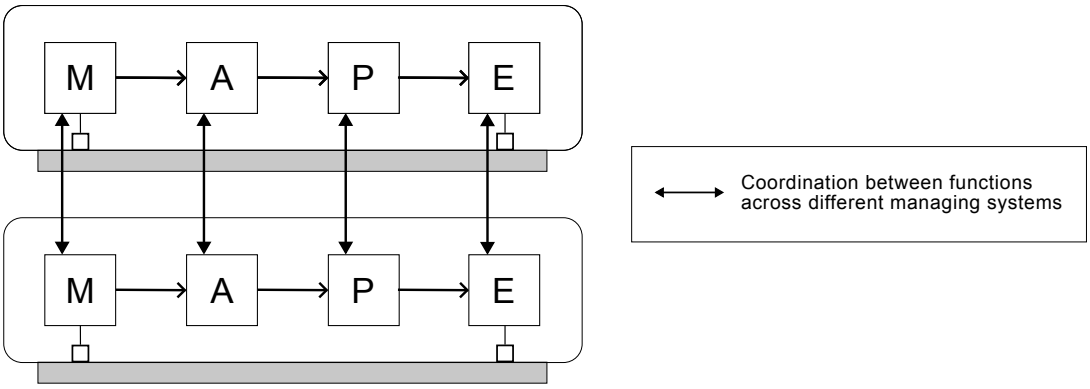


Figure 2.5: Coordinated control pattern from [20]

system may coordinate with the same phase hosted on other managing systems. To represent such interactions, a double arrow notation is added in Figure 2.5.

This pattern can allow each node of a distributed system to be autonomous while still allowing some degree of collaboration. For instance, in [23], authors present DOCMA, a decentralised managing system for Cloud and Edge applications. Edge systems aim at pushing computing and storage close to the sources of data [22]. Edge nodes can share similarities with CPSs, such as nodes with low resources. In DOCMA, nodes are organised in flat structure (i.e., without hierarchy). Each node is potentially part of the managing or managed system. Nodes hosting the managing system are automatically elected whenever a user application needs to be deployed on the system. Managing and managed systems are selected based on properties such as type and quantity of resources (e.g., energy, CPU, storage). Managing systems collaborate at the *Monitoring* and *Analysis* phases: these nodes automatically discover their neighbours and select a subset of nodes as a managed system to host user applications. Managing systems also exchange information about failure and mobility (i.e., nodes entering/leaving the network). Failure in the managing or managed systems is automatically detected and fixed by replacing failed nodes with new ones from the neighbourhood.

In [24], authors present SeMaFoR (Self-Management of Fog Resources), a vision for the decentralised and autonomous management of Fog systems. Fog systems are geo-distributed systems aiming at bringing Cloud resources closer to end users, avoiding single point of failure. SeMaFor provides an Architecture Description Language (ADL) allowing to model any kind of Fog system, including hardware and software modules and their topologies. This includes their properties such as embedded hardware (e.g., GPU, type of sensor), remaining energy, latency, etc. SeMaFor considers multiple autonomous Fog controllers that are responsible for their own local systems. A collaborative decision-making algorithm is used to take decision globally about what the target state of the different systems managed by each controller should be (*Analysis* phase). For instance, when a user wants to host an application composed of multiple services, the decision-making algorithm uses the Knowledge of each site to know where to deploy each service according to its constraints (e.g., specific hardware required, low latency, etc.). A decentralised plan computation algorithm (*Planning* phase) is then used to collaboratively compute the Plan to reach the target states computed by the *Analysis* phase.

In some use cases, nodes may also be loosely coupled, and act as autonomous and independent entities. In these cases, collaboration is optional. It only contributes to enhance the quality of service, rather than being a condition for the system to work properly. In [25], mobile sensor nodes opportunistically collaborate during random encounters. This

collaboration aims notably at improving the quality of service or fault-tolerance. During collaboration, nodes exchange data such as the node's position, available services, or quality of sensing. Nodes may then decide to temporarily use services from other nodes with better quality (e.g., data extracted from better sensor). The decision is done collaboratively with other nodes.

It is considered that coordinated control pattern would be suited for the use case of this manuscript (presented in Section 2.3). The ability to provide reconfiguration capabilities to all nodes, removing single point of failure, and enabling a degree of collaboration when necessary are relevant properties for the use case. The main drawback of the coordinated control pattern is its complexity. No single entity has a global view of the system. Each managing system has a view only of itself and its collaborators.

Works of this manuscript focus on the decentralised *Execution* phase of the coordinated control pattern. As a reminder, the *Execution* phase consists in executing actions given by the *Planning* phase. The execution of these actions means the reconfiguration of the managed system. In the following, the *Execution* phase is simply called reconfiguration.

2.3 Autonomous CPS use case: the DAO-CPS

The use case of this manuscript is the deployment of a CPS to monitor the Arctic Tundra. The Arctic Tundra is a difficult environment to monitor. It is a very large and hard-to-reach area. Covering its size requires large number of nodes on multiple observation sites. Due to strict regulation and lack of roads, navigating through the area requires specialised equipments and vehicles. Energy supply is limited. Nodes deployed on the field must be small, non-intrusive, and forced to rely on batteries. Nodes may stay for long periods with difficult environmental conditions. Harsh weather and the absence of eligible energy harvesting mechanisms (e.g., sun, wind, etc) prevent nodes from accessing a regular energy supply. Harsh weather also prevents physical access to observation sites for long duration, up to 6 months during winter. Network coverage is rare or absent. To communicate, nodes deployed on the field must rely most of the time on peer-to-peer communications.

Scientists wanting to observe such an environment could benefit from an autonomous system, to alleviate part of the maintenance and interaction [6]. Such a system can be implemented as a CPS. CPSs are widely used in a variety of use cases, including hard-to-reach environment monitoring like ocean observatories [26, 27], the Swiss Alps [28] and Sierra Nevada [29]. CPSs are composed of actuators, sensors to collect data, and have computing and networking capabilities. Sensor nodes of the CPS may collaborate to provide additional services (e.g., data aggregation [30], analytics [7]) or increase the resilience of the system (e.g., data dissemination [8]).

The DAO project The Distributed Arctic Observatory (DAO)[1] project proposes a CPS as a scientific observatory to monitor the Arctic Tundra (Figure 2.6). It is mainly composed of Observation Nodes (ON). ONs are small devices monitoring the environment through physical instruments (e.g., optical and proximity cameras, CO2 sensors). ONs have computing and networking capabilities to enable advanced capabilities such as collaboration (e.g., single-board computers and LPWAN radio technologies such as

[1] `https://site.uit.no/dao/`

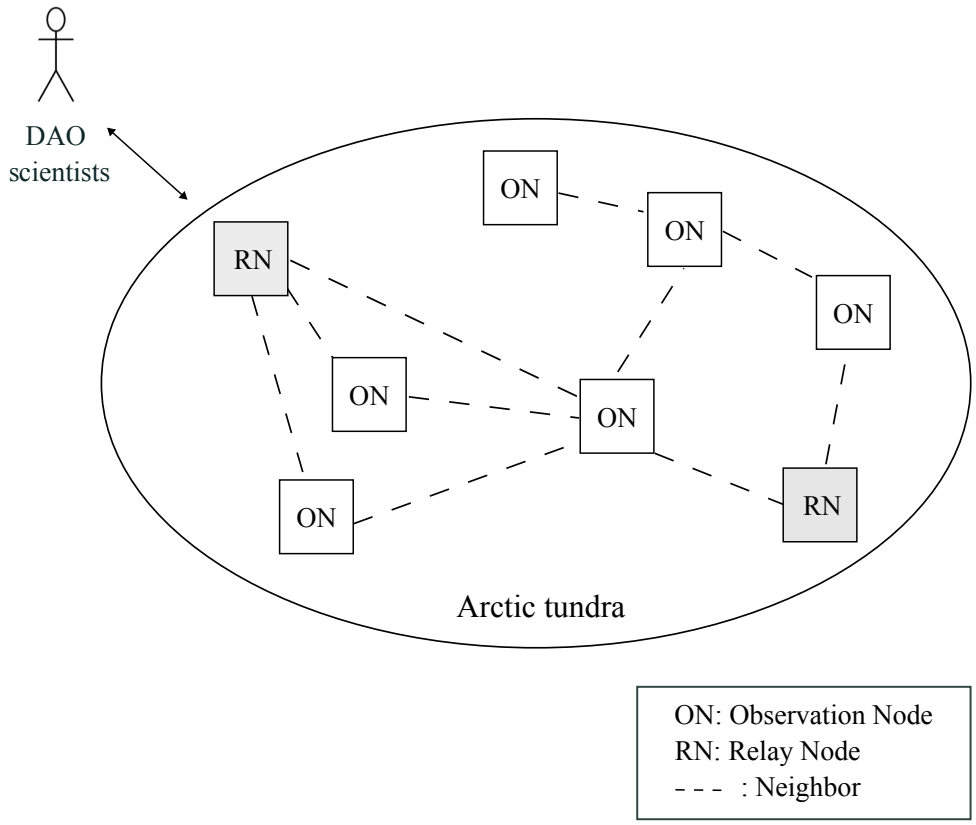


Figure 2.6: DAO-CPS in the Arctic Tundra. ONs are most of the time isolated from external communications. ONs and RNs have varying numbers of neighbours.

LoRa and NB-IoT) [7]. ONs can collaborate for observation through wireless, ad hoc and temporary peer-to-peer connections. When available, a back-haul network allows remote interactions with the system (e.g., collect data, send requests to ONs). Connectivity to this network is sporadic and not reliable. This forces ONs to be autonomous most of the time.

Uptime schedule Due to limited energy budget, ONs alternate between short uptime and long sleeping periods to increase their lifetimes. Uptime schedules are not synchronised and vary according to observation activities or energy budget. Each ON has its own observation activities which may lead to different uptime schedules between ONs. In this manuscript, ONs are assumed to wake up randomly once every hour.

While sleeping, ONs save energy but cannot interact with the rest of the CPS. During uptime, ONs perform different tasks. These include performing measurements, creating connection with neighbours, doing computations and adapting to occurring events. ONs wake up primarily to perform measurement and create connection with neighbours, which are high-priority activities.

Relay Node In the DAO-CPS, very few ONs have access to larger energy supply. In this manuscript, it is proposed that these ONs are used as Relay Node (RN). The RN is a special type of node able to be awake at the same time as its neighbours. It can be used as an intermediary to help communication between the other sleeping nodes. It can temporarily store messages while the recipient is sleeping, and forward this message to the recipient when it wakes up. To study the impact of such node on the system,

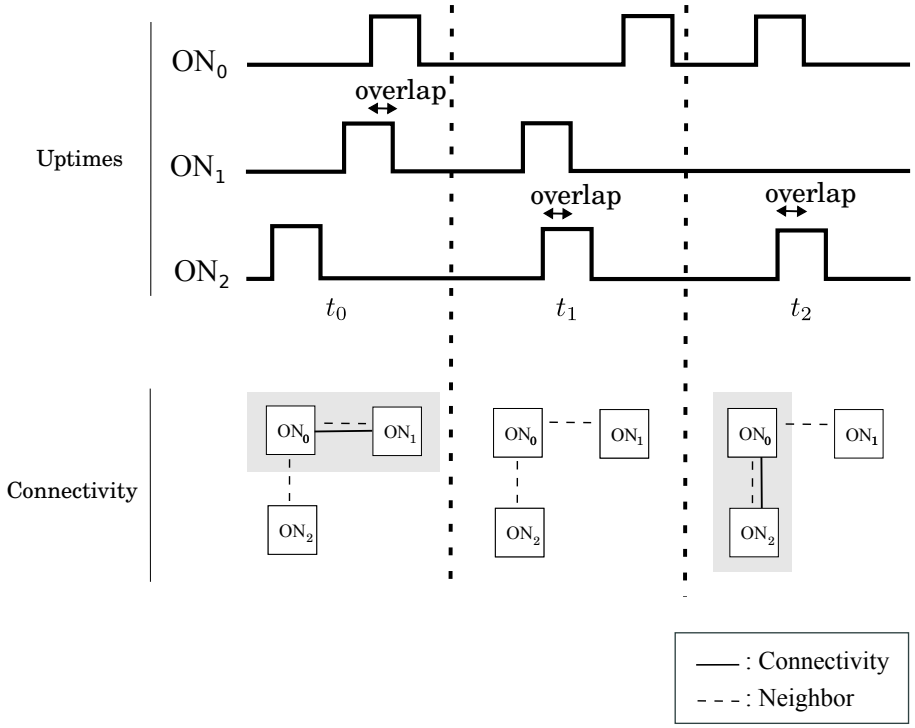


Figure 2.7: Connectivity over time between neighbours, according to ONs' uptimes overlaps.

the following hypothesis is made: the RN has full knowledge about neighbours' uptimes and has sufficient energy to be awake at the same time as its neighbours. The RN can temporarily store messages while ONs are sleeping and forward messages to ONs on wake-up.

Collaboration between ONs ONs collaborate for observation. Examples of collaboration include the treatment by ONs of collected data from the CPS such as wildlife images [7]. To enable such a collaboration, services must be exposed and provided by ONs to other ONs. For instance, ONs responsible for treatment of data must expose their services to ONs collecting data. These services must be properly configured to prevent failure. For instance, wrongly parametrised data (e.g., image encoding or compression) may be unusable for a recipient service. Trying to use an inactive remote service may lead to data loss or crash.

Communication opportunities between ONs When dealing with peer-to-peer networks composed of sleeping ONs, connectivity between ONs depends on overlaps between ONs' uptimes. Figure 2.7 shows an example with 3 ONs. ON_0 is neighbour of ON_1 and ON_2. At t_0, ON_0 and ON_1's uptimes overlap, allowing communication. At t_1, ON_1 and ON_2's uptimes overlap, but these ONs are not neighbours, preventing communication. At t_2, ON_0 and ON_2's uptimes overlap, allowing communication.

Improving connectivity with the Relay Node As stated previously, it is assumed that the RN can be awake at the same time as its neighbours using its larger energy supply. Figure 2.8 shows an example with the same 3 ONs as Figure 2.7, with the RN replacing ON_0. The RN has the same initial uptimes as ON_0. On top of it, the RN is also awake at the same time as ON_1 and ON_2. The additional and extended uptimes are

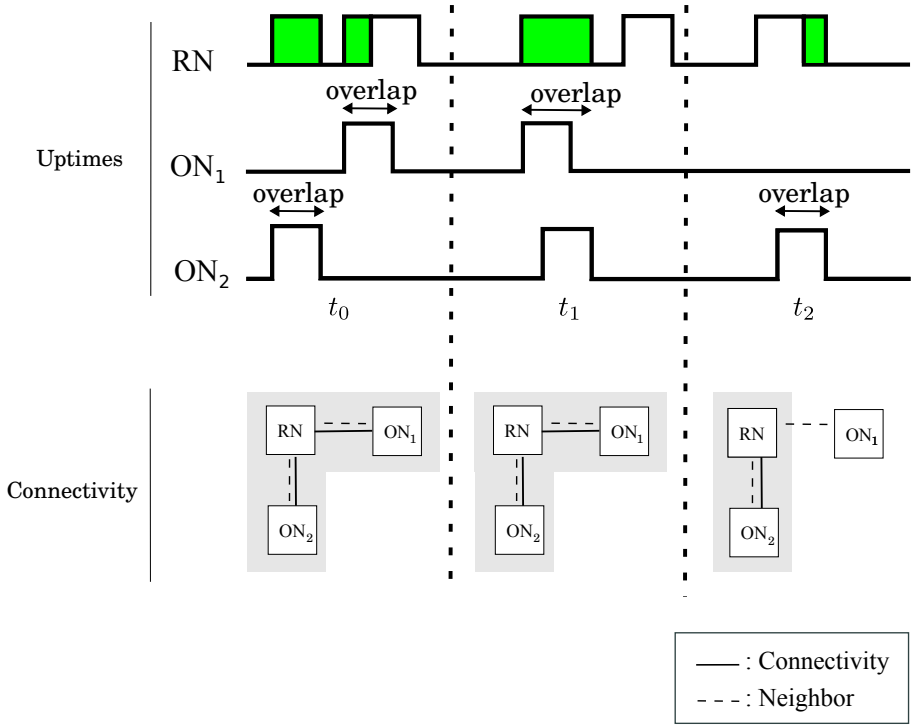


Figure 2.8: Connectivity over time between neighbours with an RN. The RN has its own observation schedule (e.g., observation schedule from ON_0 in the example). The RN has additional and extended uptimes (coloured).

represented in colour. In this manuscript, the RN is assumed to have sufficient knowledge about its neighbours' uptimes to guarantee its availability.

Reconfiguration capabilities To provide autonomy for ONs, automatic adaptation capabilities can be enabled. Such capabilities allow ONs to adapt to their environment (e.g., handle ON failure, etc.). Automatic adaptation involves different phases described in Section 3. These phases are (i) environment monitoring (ii) decision about whether an adaptation is required (iii) definition of a plan of actions to perform if an adaptation is required (iv) execution of these actions. This manuscript focuses on the Execute phase of an adaptation (i.e., reconfiguration). Therefore, this manuscript focuses on the reconfiguration of software modules deployed on ONs.

When ONs are collaborating, coordinating such a reconfiguration is essential to prevent failure. These failures and corrections may imply an additional energy consumption overhead, which ONs can hardly afford regarding their limited energy budget. However, the execution of reconfiguration and its coordination also imposes an energy consumption on ONs. This manuscript aims at studying such consumption on ONs. In the DAO-CPS, reconfigurations are usually low-priority tasks. The energy budget of ONs should be reserved for observations and computations. Thus, synchronizing the wake-up schedules of ONs to ease communications during reconfiguration is not considered in this manuscript.

Note that, in this manuscript, experiments are not conducted on real ONs deployed in the Arctic Tundra. Experiments on real hardware emulate the uptime and sleeping periods of ONs, but are not real ONs. However, for the sake of simplicity, nodes are called ONs when referring to the DAO use case (e.g., at Chapters 5 and 6).

2.4 Reconfiguration of distributed systems

This section deals with the reconfiguration of distributed systems (i.e., *Execution* phase of the MAPE-K). The Knowledge (K) is re-introduced to represent the current configuration of the managed system. The following introduces definitions and concepts to present the reconfiguration.

2.4.1 Representation of the system

Reconfiguration solutions need to have a representation of the system to manage its individual modules. Such a representation allows to know what has been deployed on the system and what is its structure.

State of the system The state of the system is composed of the modules added to the system, their connections, and the internal configuration of each module. A convenient way to represent this state is the component-based representation. The component paradigm can be used as a representation method or software development method. In this manuscript, the component-based representation is used to represent modules and their connections [31].

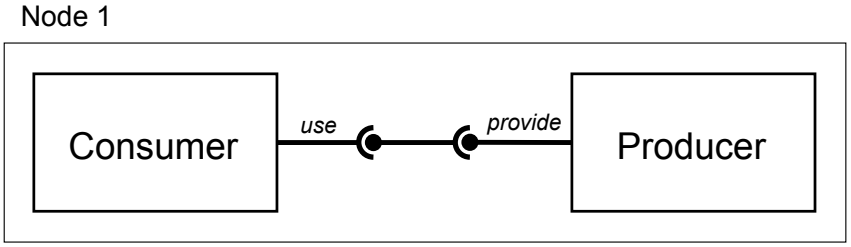
In the component-based representation, modules can be connected to each other. These connections represent relationships and interactions between the different pieces of software. To represent such relationship, the component-based representation uses ports and connections.

The reconfiguration solution needs to keep track of modules added to the system, their connections and their internal configurations. In this manuscript, it is considered that these are defined in the Knowledge of the MAPE-K loop. This is justified as phases like the *Planning* may also need to know what is present on the system to function properly. This Knowledge is used by the reconfiguration solution to keep track of the execution advancement.

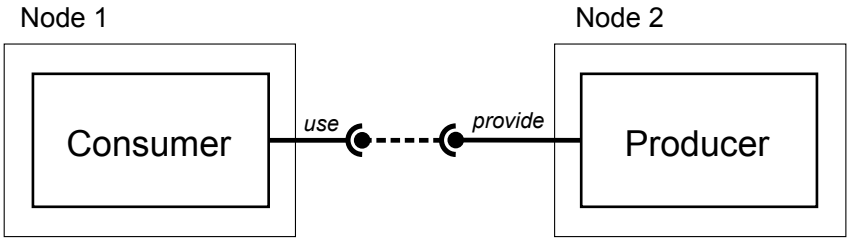
Life-cycle of modules Dealing with the internal configuration of modules means the ability to define their life-cycles. The life-cycle of a module is a representation of all configurations a module can be in, and how to go from one configuration to another. The configuration of a module represents its internal software behaviour and interactions with the external world. For instance, a data producer sending data to a consumer. The simplest life-cycle defines the existence of a module. A more complex life-cycle can define more configurations to represent more accurately the internal software, such as its installation, configuration, activation, deactivation, etc. The transitions between one configuration to the other are associated with concrete changes to perform on the module, such as executing a shell script, rewriting part of a software, etc. [11, 14].

When modules are connected, their life-cycles are not independent. Due to dependencies between modules, the execution of life-cycle on one module influences the execution of the life-cycle of another module. For instance, the consumer needs the producer to be activated before activating itself. If it is not the case, the consumer may fail when trying to request data from the producer.

Module operation To interact with the life-cycle of modules, modules are associated with module operations. A module operation is an action or command used to interact with the life-cycle of a module. A module operation makes the life-cycle transition from



(a) Centralised Knowledge. Node 1 manages both producer and consumer.



(b) Decentralised Knowledge. Node 1 only manages the consumer module. Node 2 only manages the producer modules.

Figure 2.9: Component-based representation used to represent data producer and consumer. The consumer is dependent on the producer.

one configuration to the next, which triggers the associated concrete changes. For instance, for a life-cycle composed of two configurations (module exists/not exists), module operations can be defined to create or delete such module. The creation of the module is associated with concrete changes that create the module (e.g., installation and activation of sensors). The deletion of the module is associated with concrete changes that delete the module (e.g., deactivation of sensors). Module operations can include more complex operations, such as updating the module without deleting it, temporarily suspending the activity of the module, etc.

Reconfiguration in component-based representation A reconfiguration consists in changing at runtime the configuration of a distributed system. Reconfiguration includes changing the internal behaviour of individual modules (through their life-cycles) and structural changes, such as adding/removing/connecting/disconnecting modules. To do so, reconfiguration actions are executed. Reconfiguration actions include the addition/removal of modules to the system, connection/disconnection of modules and change of the internal configuration of each module using module operation. During reconfiguration, the dependencies between modules must be respected.

2.4.2 Decentralised reconfiguration

Decentralisation of a reconfiguration involves decentralisation of the Knowledge of the system (i.e., state) and coordination of reconfiguration actions between managing systems.

Decentralised vs distributed The distinction and definitions of decentralised compared to distributed are inspired by [20].

The term distributed refers to the location of software on nodes of the system. A software is said to be distributed if its modules are located on multiple nodes connected via

the network. A software is not distributed if all its modules are present on a single node. This can be useful to describe the managing and managed system. A managed system can be distributed while the managing system is not. In this case, the managing system is present on a single node and coordinates a reconfiguration for multiple distributed managed nodes. Conversely, the managing system can be located on distributed nodes while the managed system consists of a single node (e.g., multiple sensor nodes to monitor the activity of a single node).

The term decentralised refers to how control decisions of the different components of an autonomous system are coordinated. This is independent of how these different components are distributed. Applied to the managing system of the MAPE-K model, having a decentralised MAPE-K phase means that multiple instances of this phase coordinate their activities. Each of these instances is responsible for its own part of the system and applies changes locally, but coordinates these changes with other instances with which it collaborates.

With these definitions, it is possible for example to have decentralised instances of the *Analysis* phase on a single node. It is also possible to have distributed but independent instances of the *Analysis* phase on different nodes. These distributed instances do not collaborate, and are not considered decentralised. For example, this can be used when these instances are replicated to treat an increased workload.

Decentralised state of the system Figure 2.9a is a representation of centralised Knowledge of both producer and consumer modules. Using this Knowledge, the node can reconfigure these modules locally. In Figure 2.9b, the consumer and producer's Knowledge is split between Node 1 and Node 2. Producer and consumer modules are still connected, but remotely, as represented by the dashed line. Reconfiguring these modules requires coordination between Nodes 1 and 2.

In this manuscript, a connection between modules defined within the same Knowledge is called local connection. A connection between modules defined on different Knowledge is called remote connection. From the point of view of a reconfiguration, a module accessible by local Knowledge is called a local module. A module defined on another node with different Knowledge is called remote module.

Decentralised reconfiguration When dealing with decentralised reconfiguration, each managing system only has local Knowledge about its own modules. When modules and connections are defined locally, the reconfiguration is the same as centralised one. When modules and connections are defined remotely (i.e., on collaborative nodes), communication during reconfigurations must occur between collaborative nodes. These communications are notably used to respect dependency between modules (see Figure 2.9b).

2.5 Energy consumption due to reconfiguration of CPSs with sleeping nodes

Nodes have to spend resources to execute reconfiguration actions. This includes resources spent for the execution and coordination of reconfiguration actions between collaborative nodes. When dealing with decentralised reconfiguration, collaborative nodes must communicate to coordinate the execution of these actions. As this manuscript deals with

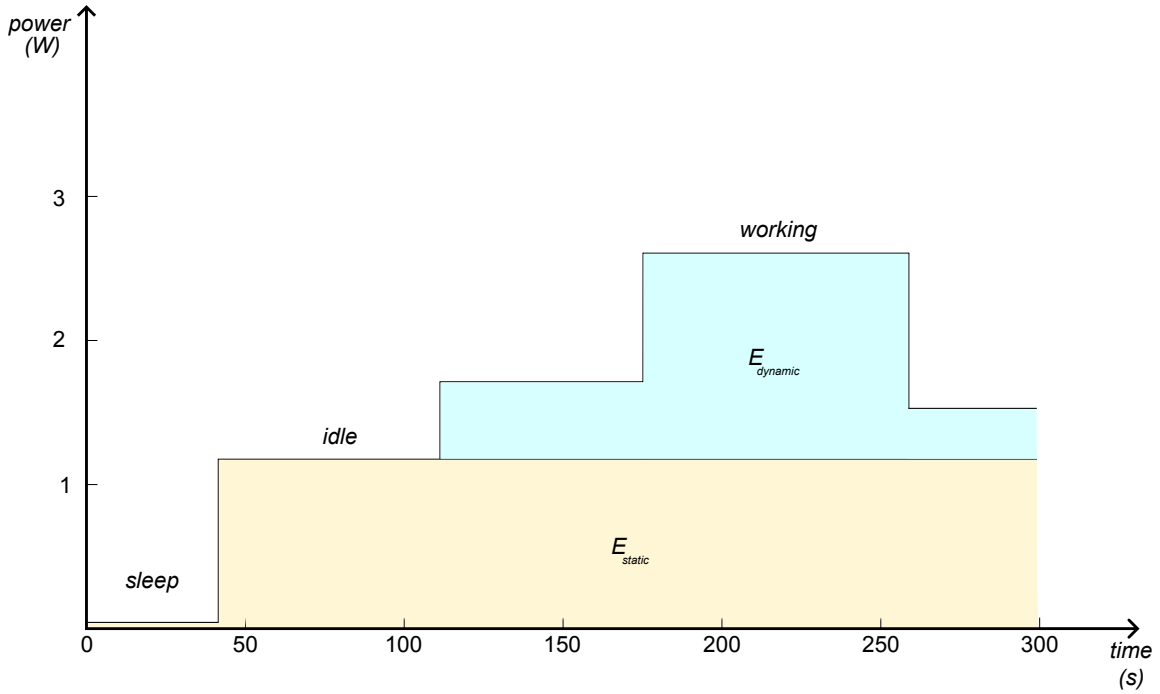


Figure 2.10: Simplified representation of static and dynamic power consumption for a node such as Raspberry Pi. When sleeping, the static power is very low and the dynamic power is null. When up but without any workload (i.e., idle), the static power increases significantly, but the dynamic energy is still null. When a workload is imposed on the system, dynamic power is consumed.

CPSs deployed in scarce-resource environments, coordination of these actions may be difficult. Nodes of the CPS may enter sleeping periods to save energy, and communication must be wireless and done in peer-to-peer manner. During sleeping periods, nodes have to suspend any on-going reconfiguration process and cannot communicate with other nodes. This section presents definitions and concepts to describe such CPSs and the energy consumption of CPS nodes due to reconfiguration.

Power and energy consumption The power is the rate at which the work is performed on a given system. The power consumption of a system can be classified in two main categories: static and dynamic. The static category corresponds to the power passively consumed by the node independently of running activities. It represents the fixed part of the power consumption. The dynamic category is the variable part of the power consumption and depends on the workload put on the system (e.g., executed processes, activity of peripherals connected to the system, etc.). The total power consumed is the sum of the static and dynamic power.

The energy consumption of a system corresponds to the power usage over time. For a given interval T_1 to T_2 with $T_1 < T_2$, the energy consumption E of a system can be obtained by computing its power usage P between T_1 and T_2 [32]:

$$E(T_1, T_2) = \int_{T_1}^{T_2} P(t)dt \tag{2.1}$$

A simplified representation of the energy consumption of a system is depicted on Figure 2.10. In this figure, the static and dynamic energy consumption are represented. While sleeping, the system consumes very low amount of static energy, and no dynamic

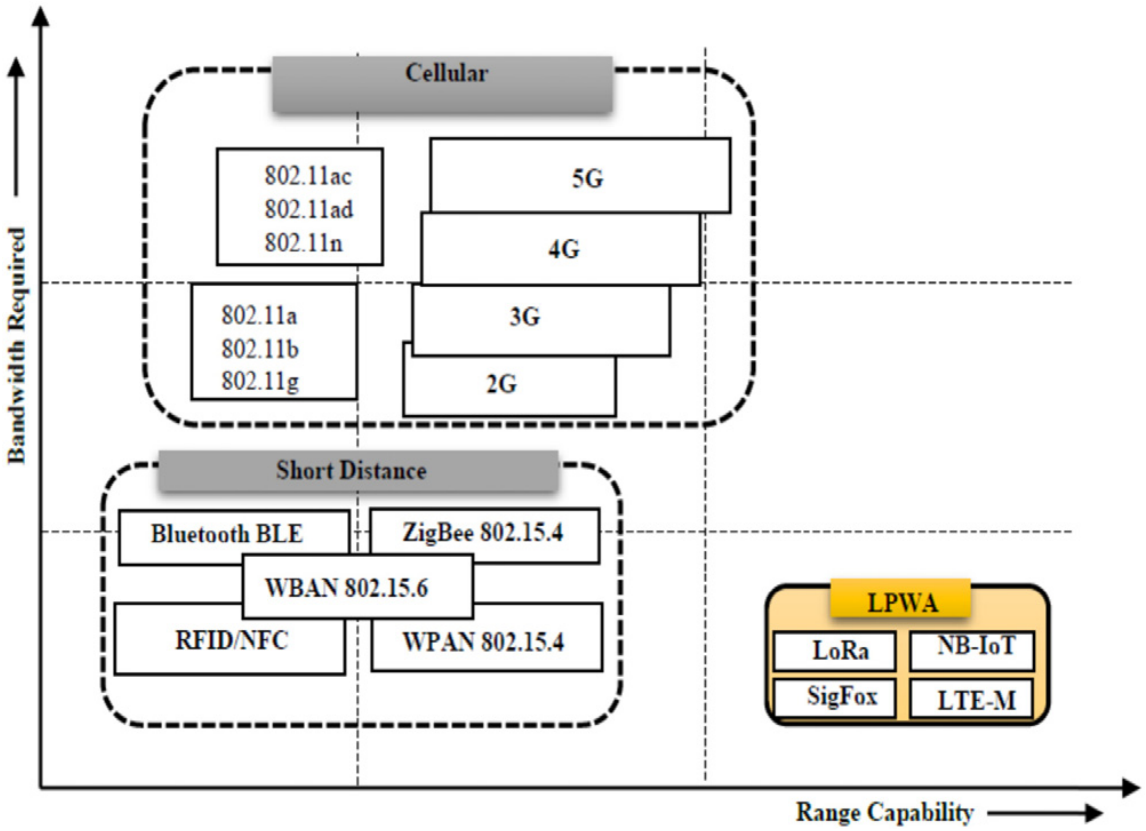


Figure 2.11: Comparison of wireless technologies according to their range and bandwidth. LPWAN technologies such as LoRa and NB-IoT enable long range and short bandwidth, while having low power consumption [33].

energy. When up and idle, the static energy consumption increases. The dynamic energy consumption depends notably on the intensity of the workload.

Sleep mechanism In CPSs with low energy supply, nodes may enter sleep periods to reduce their activity and save energy. In this manuscript, sleeping makes the node suspend all its activity. This includes the ability to reconfigure or communicate with other nodes.

Energy consumption due to reconfiguration In this manuscript, it is considered that the dynamic energy consumption due to reconfiguration is composed of (i) the energy due to communication required for the coordination, and (ii) the energy due to reconfiguration actions. The reconfiguration involves change in the software present on the node and its configuration. In this case, the node must re-write data to memory or disk. When runtime must be changed (e.g., processes, sensor activity), the node must create or remove processes, reconfigure sensors, etc. Finally, when the managing system is hosted on the node, activities related to the managing system also involve an energy consumption overhead on nodes (e.g., related to each MAPE phase).

When communication is necessary, nodes consume energy by sending or receiving data from other nodes. This includes software data such as updates, which can be large depending on the size of the update. This also includes commands to change runtime activities. Finally, data used for coordination between different managing systems, acknowledgment, health check, etc., can also impose energy consumption overhead on nodes.

Energy consumption due to wireless communication Nodes deployed in hard-to-reach, scarcely and limited resource environments such as the Arctic Tundra are often equipped with wireless radio technologies [8]. Range of communication and transmission's data rate depends on the radio technology used by the node. Radio technologies allowing long range and high throughput consume more energy. Such technologies do not suit use cases with energy-constrained nodes. For such nodes, LPWAN (Low Power Wide Area Network) technologies are more suited. These technologies enable long range communication and low energy consumption at the cost of low data rate. Examples of LPWAN include LoRa and NB-IoT (see Figure 2.11).

The next chapter is dedicated to the state of the art regarding the reconfiguration solutions for distributed systems (which include CPSs), and which solution could be relevant for the DAO use case (i.e., decentralised reconfiguration). This state-of-the-art includes review of observatory CPSs and their adaptation, and the energy consumption overhead due to reconfiguration activities (i.e., execution of reconfiguration actions and wireless communication).

Chapter 3

Decentralised reconfiguration of CPSs and its energy consumption: State of the art

Contents

This chapter presents the state of the art regarding decentralised reconfiguration, the reconfiguration in CPSs, and the impact of reconfiguration on the energy consumption of the system. In Section 3.1 solutions for the decentralised execution of a reconfiguration are introduced. Centralised solutions are also proposed, which provide relevant properties for the reconfiguration. Section 3.2 presents the positioning of this manuscript regarding observatory CPSs in the literature and their reconfigurations. Section 3.3 deals with the energy consumption due to a reconfiguration, especially on systems with low resources such as CPSs.

3.1 Reconfiguration solutions

The component-based representation provides a convenient way to study reconfiguration solutions. In such a representation, the managed system is composed of modules, with their topologies (e.g., dependencies) and configurations. Reconfiguration aims at performing changes on the managed system by executing the ordered or partially ordered set of actions given by the *Planning* phase [34]. These actions aim at creating and updating modules, by changing their dependencies and internal configuration at runtime. To prevent failure, these actions have to be coordinated (e.g., to respect dependency between modules). Depending on the architecture of the managing system, such a coordination can be local to single node (e.g., centralised) or handled by multiple nodes (e.g., decentralised).

Changing the internal configuration of modules can be done by executing module operations. These operations define what is possible to reconfigure on modules. Most solutions provide CRUD operations allowing to create, update, or delete modules. These operations provide simple interfaces with modules but are limited to few operations. Some solutions provide the ability to define more complex operations to define additional actions such as suspension of activity. Such actions allow the definition of more complex life-cycles, as operations represent more accurately the different configurations the module can be in (see Section 2.4).

Module operations can be executed individually on modules or coordinated between modules, when modules are connected. Depending on the dependencies defined between modules, the execution of module operations can be executed sequentially or in parallel. Theoretically, a parallel execution of module operation should allow an overall faster reconfiguration execution.

This section focuses on reconfiguration solutions according to the capabilities in terms of decentralisation, type of module operation, and level of parallelism. In this manuscript, the decentralisation capabilities are the most important. However, as few contributions from the literature provide decentralised reconfiguration, centralised solutions and their capabilities in terms of type of module operation and level of parallelism are also reviewed.

3.1.1 Reconfiguration paradigms for component-based representation

Different paradigms exist in the literature and industry dealing with component-based representation.

Component-based software oriented reconfiguration Component-based software systems are designed such that the application programming is split into two phases: the writing of business code, and the composition phase. The component models provide a structured programming paradigm, and ensure program re-usability. Dependencies between components can be defined to represent functionalities provided by these components through use and provide ports. Some component models additionally keep an internal representation of components and their dependencies at runtime. Knowing how components are composed and being able to modify this composition at runtime allows the component system to be reconfigured [11].

Infrastructure as Code The Infrastructure as Code (IaC) paradigm considers the management of applications using the same practices as software development. In IaC,

modules are represented using specific language (DSL) or extension of existing languages (e.g., C, C++, Python). The IaC paradigm does not consider the internal software of each module. It considers only their life-cycles. Using IaC solutions, modules can be declared and connected using these languages. Lots of these solutions are used in the industry to facilitate the management of applications and infrastructure where these applications are hosted.

There are multiple categories of IaC solutions. First, some IaC solutions implement a whole MAPE-K loop. These are solutions that are able to take parameters regarding the number of modules to be deployed and can provide features such as the adjustment of the number of modules according to the workload, the redeployment of failed modules, etc. It corresponds to the *Analysis* phase of the MAPE-K (see Figure 2.2). Second, some IaC solutions are specialised in configuration. In these solutions, the modules take the form of high-level scripts that can be applied to the managed system. Modules in these cases correspond to these scripts. Third, some IaC solutions provide declarative languages to declare the modules and compose them following a similar representation as component-based representation. Typically, these modules are declared in the source file. A process takes this file as input and automatically computes the different module operations and in which order actions must be executed. It corresponds to the *Planning* phase of the MAPE-K (see Figure 2.2). The Knowledge of modules and their connections is typically kept in a state file, which is updated every time a change occurs in the managed system.

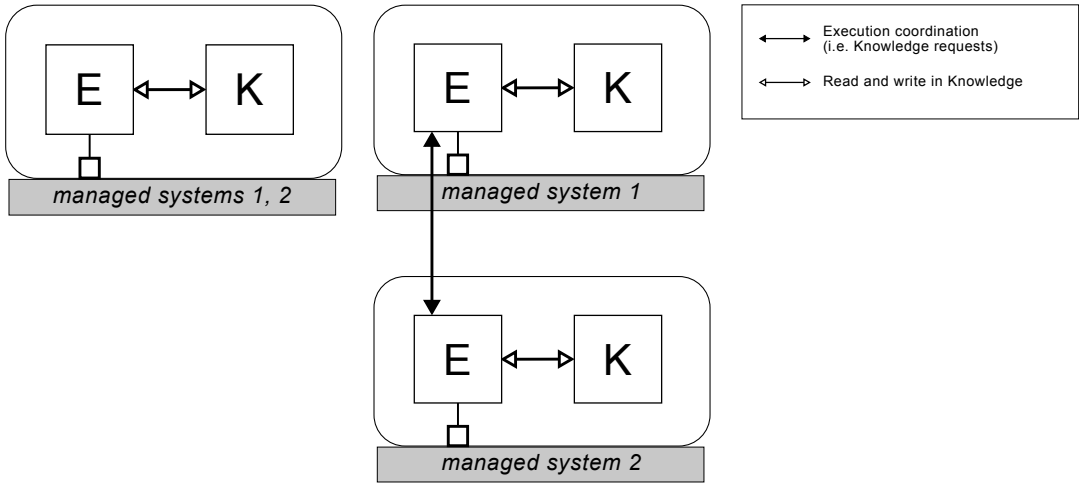
Dynamic Software Update In the Dynamic Software Updating (DSU) paradigm, the running processes of applications can be updated without interruption [35]. In DSU, modules represent software at low level. Instead of representing service or infrastructure, modules represent complete or parts of the source or compiled code. These can be individual functions, classes, or whole files. Dependencies between two modules can be represented by function calls or variable access [36]. Additionally, DSU solutions often deal with single node reconfiguration. Therefore, due to the specificities of DSU compared to component-based software and IaC, DSU solutions are considered outside of the scope of this manuscript, which focuses on component-based representation. Note that some solutions try to bring component-based representation to DSU. One of them is Coqcots, which aims at integrating DSU techniques in component models. In this solution, the internal configuration of components represents the code under execution, which can be connected with use and provide ports [37].

3.1.2 Comparison criteria

This subsection presents three relevant criteria for the comparison of the different reconfiguration solutions from the literature. These criteria are the solution allows decentralised execution of reconfiguration, the type of module operation, and the level of parallelism provided by the solution.

Decentralised execution As a reminder, the Knowledge is used by the reconfiguration solution to track changes made to the configuration of the managed system (see Section 2.1). This configuration includes modules added to the configuration, their connections, and the progression of the execution of module operations.

Figures 3.1a and 3.1b provide a representation of centralised and decentralised *Execution* phases (i.e., reconfiguration) along with their respective centralised and decentralised Knowledge. On Figure 3.1a, a single, centralised managing system is responsible for the



(a) Centralised *Execution* phase. Knowledge and *Execution* phase are both on the same managing system.

(b) Decentralised *Execution* phases. Knowledge and *Execution* phases are on different managing systems. Each managing system is responsible for its own managed system.

Figure 3.1: Centralised and decentralised *Execution* phases.

whole managed system. On Figure 3.1b, the managed system is split into two parts (1 and 2). Each part is managed by its own managing system. The depicted *Execution* phases keep their own Knowledge locally on their system, but communicate when coordinating their changes. Such communication is used notably to extract data from the Knowledge of the other system, such as which modules have been deployed, what reconfiguration actions have been executed, etc. Note that this communication is necessary only when modules are connected across two different managing systems.

Solutions from this state-of-the-art are classified according to these two patterns (centralised and decentralised). Note that some centralised pattern may have distributed but independent *Execution* phases (see "distributed vs decentralised" in Section 2.4). In such cases, *Execution* phases are not able to collaborate, such as in the controller/worker pattern.

Types of operation Having the ability to manage a system on the long run involves the ability to perform different kinds of action on the system. Most of these actions are CRUD operations that allow to create, update, and delete modules. Some module operations allow to define custom changes on the module (e.g., suspend). Modules can be connected and disconnected implicitly (e.g., directly in the declaration of the module) or explicitly, by calling a specific operation (e.g., connect). Some solutions allow advanced operations on modules such as scale or migrate, however, these operations often are composition of more simple CRUD operations such as create and delete. The following describes each of the basic module operations.

The *create* action adds a new module in the architecture. This may be the execution of a script, the deployment of a container, the provisioning of a piece of infrastructure such as virtual machine, etc.

The *delete* action removes a module from the architecture. This action can be used

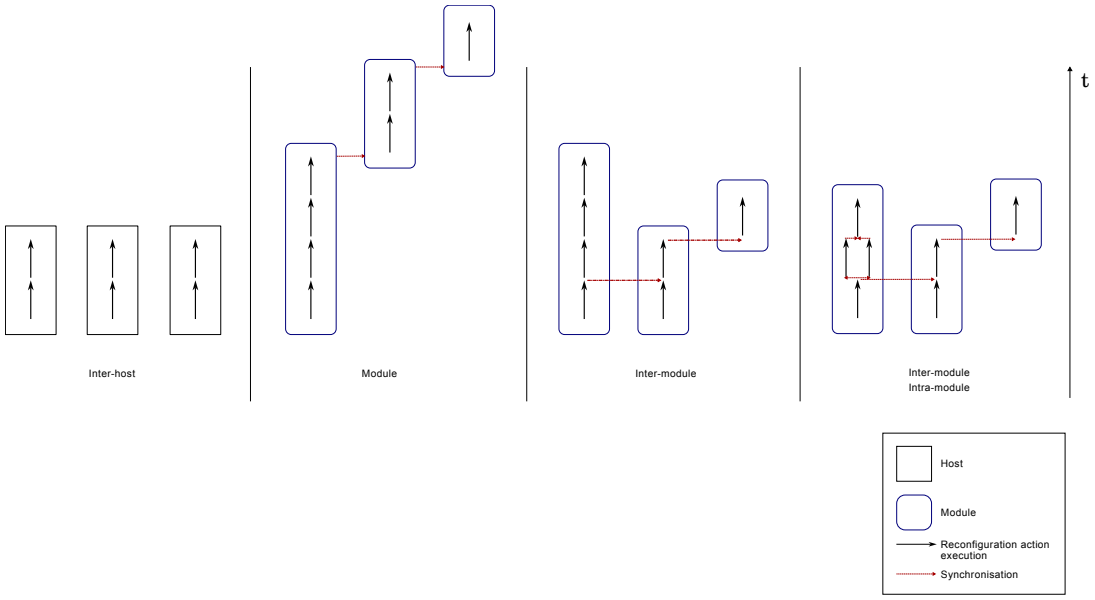


Figure 3.2: Depiction of the four levels of parallelism studied in this manuscript. Note that the dashed red arrows only represent temporal execution, not the exchange of data. This is inspired by the state of the art of [38].

to remove everything associated with this module, or keep some data, such as cache data. Note that, in some cases, this action may not be available. This may be the case for solution dealing with application of scripts (which do not have an action planned to remove the modification done by the script).

The *re-create* action re-creates the module with new parameters, such as software version. It is the combination of *delete* and *create* actions. It is typically done when the configuration of a module needs to change, and the module needs to be re-created to perform such change. For instance, changing the size of a virtual machine.

The *update* action does not need to re-create the module to update it. Changes are directly applied on the module. For instance, dynamically loading a configuration file into a service.

The *custom* action refers to any type of change defined additionally to the *create*, *remove*, *re-create* and *update* actions. For instance, suspension of the activity of the module without its removal.

Level of parallelism Modules and their life-cycles may have dependencies between each other. The execution of these actions should respect the dependency order. For instance, an action specifying the activation of a service may be executed after an action specifying the installation of such service. These actions may be executed sequentially, respecting dependency order.

Modules may also be independent of each other. The execution of independent actions does not impact the execution of others. Theoretically, having the possibility to execute these actions in parallel should decrease reconfiguration duration. Realistically, this depends on different factors, such as the underlying hardware. In case of CPSs with constrained nodes, the parallelism may be possible only for limited amount of actions, and for a subset of nodes (e.g., single-board computers with several CPU cores).

In the context of reconfiguration, having the possibility to define independent actions depends notably on the ability to define fine-grained life-cycle of modules. A fine-grained

definition of life-cycle of modules allows to define fine-grained dependencies between modules. For instance, the installation of consumer and producer may occur in parallel, while the activation of the producer should occur before the consumer.

Four levels of parallelism are considered: *intra-module*, *inter-module*, *module* and *inter-host*. Figure 3.2 depicts these four levels and gives an example for each one. Note that the red arrows represent the temporal execution of modules and not exchange of data. Note that the location of modules and resolution of dependencies are abstracted. Two dependent modules can be located on the same or different nodes.

The *inter-host* level allows to execute identical module operations on different hosts in parallel. All hosts execute the same module operation in parallel. When all hosts finish the execution, the next operation is executed.

The *module* level allows to express and resolve dependencies at the level of whole modules. When a dependency exists between two modules, all actions on the first module must be complete before the second module can execute its own.

The *inter-module* level allows a parallelism at the level of individual transitions of the life-cycles between different modules. It is a finer-grained parallelism compared to *module* level. A dependent module has to wait only for a subset of changes to be executed instead of all. An example is depicted in Figure 3.2. In the second module, only the first action from the first module is required. Subsequent actions can be executed in parallel between the first and second modules.

The *intra-module* level allows a parallelism between transitions within the same module. It is complementary compared to *inter-module* level of parallelism.

3.1.3 Decentralised solutions

This subsection presents decentralised reconfiguration solutions from the literature. Each solution is described according to criteria presented in Section 3.1.2. Note that solutions presented in coordinated control pattern of Section 2.1 are not presented in this subsection, as they focus on higher steps of the MAPE-K loop (M and A).

Decentralised dynamic software update As stated in Section 3.1.1, DSU contributions are outside of the scope of this manuscript. However, some contributions aim at bringing some decentralisation capabilities to individual DSU processes. In [39], authors bring decentralisation to the execution of a reconfiguration to multiple, distributed DSU processes. In this contribution, changes are coordinated between all collaborative nodes, using a plan issued by an initiator node. The initiator node has all knowledge about the system and dependencies between modules. This means that, while the execution itself is decentralised, the Knowledge and therefore the solution is centralised.

Decentralised deployment using reactive agents In [40], authors use a set of reactive agents to automatically coordinate a set of actions given by an external *Planning* phase. The plan is computed by a central node and distributed to the different agents. Using this plan, modules can be created for each agent. Modules have dependencies between each other for a single agent or between different agents.

Each agent is located on its own node. Agents autonomously coordinate the execution of reconfiguration actions without relying on central authority. Agents have self-healing capabilities. Once the coordination is complete, agents continuously monitor the activity of their modules. When a crash is detected, agents automatically react and apply correction.

Module operations are executed sequentially for each module. Therefore, this solution has *module*-based parallelism. It is however limited to deployment as it only covers the creation of each agent and maintenance of the deployed software. This means that it is only able to *create*.

Decentralised deployment between organisations In [41] authors address the reconfiguration of an application managed by different private entities (e.g., companies). Application modules and dependencies are defined to represent pieces of infrastructure or software. The entities aim at keeping part of the system private, such as the underlying infrastructure hosting the software. The solution aims at connecting pieces of software running on different entities while abstracting the underlying infrastructure.

The solution uses an algorithm to compute and distribute reconfiguration plans to the distributed managing systems hosting *Execution* phases (i.e., handling the reconfiguration). These *Execution* phases then automatically coordinate the reconfiguration. The execution of the reconfiguration starts by a global synchronisation. Each *Execution* phase exchanges n-1 messages with its peers (n being the number of other *Execution* phases of the coordination). The level of parallelism is *module* level. The solution covers only the deployment of different modules. This means that the available module operations are *create*.

Muse: decentralised reconfiguration In [42], authors present Muse, a decentralised solution for the management of distributed applications spanning across multiple teams in a company. This solution is presented with more details as it will be used in Section 5 for evaluation. Each team hosts a Muse instance to manage its local application. Applications can be connected to applications of other teams, for instance, to share a service (e.g., database). Muse automatically coordinates changes between applications belonging to different teams. Muse is built on top of Pulumi (presented in Section 3.1.4) and uses it to model and execute changes. Muse inherits the ability to perform all types of module operation except *custom* and inherits Pulumi's *module*-based parallelism.

Each module has the ability to provide or use service from other modules. Modules can be connected together inside and between applications using Wish and Offer. A wish expresses the need for a module to use another module. An Offer expresses the fact that a module provides a service to another module. Modules exchange information such as the type of data provided/used and the state of the module (i.e., active/inactive). Such information is exchanged at runtime, when modules are reconfigured.

The state of the deployed application is represented using a Directed Cyclic Graph. Vertices represent modules and their parameters. Edges represent dependencies between modules. This state represents the source of truth of the deployed application. It is saved in a file stored locally or in a remote location accessible by Muse. Whenever an update needs to be performed on the deployed application, the state is checked to compute changes to perform. Changes in the applications are tracked by updating the graph. The graph only represents local application of each team. Each team has its local state. Dependencies between different applications can be expressed using inter-graph dependencies.

Muse uses reactive engines to perform reconfiguration. Each team has a local engine which reacts to changes from engines of other teams. When changes occur in one of the teams, the engines from other teams react and dynamically compute a plan to locally reconfigure according to these changes. For instance, the engine computes actions to *delete* all local modules relying on a deleted module from another application. This is

transitive across all reactive engines. Changes made by the engine on one team trigger changes in other teams and so on. Communication between agents is done using client-server gRPC requests. The reactive engine can be gracefully exited by suspending ongoing reconfiguration executions.

Modules can be manipulated using general-purpose languages. This is inherited from Pulumi. Supported languages include Python, C#, and Typescript. Modules are declared and connected with their respective parameters using a declarative syntax. Whenever the declaration of module changes (e.g., modules added/removed, parameter changed), the new declaration is compared with application state. Reconfiguration actions are generated to change the current deployed application to the new one.

Having pre-defined and fixed CRUD interface for all modules allows reconfiguration to be simpler and more predictable. Modules always have the same module operation independently of executed concrete changes. This is paid by not being able to separate and execute concrete changes individually. For instance, the *create* action can sequentially download, install, configure and start service in a single action.

3.1.4 Centralised solutions

This subsection presents and compares solutions that can be relevant for their types of module operations and level of parallelism. Most of these solutions and the study on their life-cycles are inspired by the state of the art of [14]. While these solutions are centralised, some of these provide high level of parallelism in the execution of module operations.

Ansible [43] It is an IaC tool allowing the definition and execution of modules on a distributed system. It is widely used in the industry for a variety of use cases, notably system management. It is serverless, and relies only on ssh connections with the nodes of the target system.

In terms of module operation, Ansible is only able to *create* modules. Note that, in Ansible, modules represent scripts to be executed. The *create* operations do not necessarily have a *delete* operation. In terms of parallelism, Ansible can target multiple distributed target nodes. Each module operation is executed in parallel on each target node. The execution of each module operation must be completed before going to the next. This means that Ansible has *inter-host* parallelism. The solution can provide idempotence for some modules (but not all). For these modules, calling multiple times the *create* operation has no effect.

Chef, Saltsack, Puppet [1], [2], [3] They are IaC tools also designed to manage changes on distributed systems. Contrary to Ansible, these tools rely on a controller/worker architecture. A central controller gives the definition of the target infrastructure for all workers. The target infrastructure is defined using a declarative syntax. Chef and Puppet use Ruby files, SaltStack uses YAML and Python files. Each client periodically pulls this definition from the server. Clients compute the differences between what is present on the system and what is new, and apply updates.

[1] `https://docs.chef.io/platform_overview/`
[2] `https://docs.saltproject.io/en/latest/contents.html`
[3] `https://www.puppet.com/`

In each of these tools, a module is able to define its own actions. Most modules implement the *create* and *delete* actions, but custom actions can also be implemented [4]. Modules can have dependencies with other modules. These dependencies can be either to ask an action to execute or listen to another module and perform action when the other module's state changes. Thus, all module operations can be done on Chef, SaltStack, and Puppet. As dependencies can be expressed between modules, the parallelism is *module*-based. Finally, while clients pull and execute their modules independently of other clients, the architecture relies on centralised servers, which makes the solution *centralised*. Note that, compared to Ansible, the execution is orchestrated by the clients and not by the server. Clients have to pull once from the server before applying the update. This should intuitively require less communication between managing and managed systems compared to Ansible. This is with the exception of SaltSack which requires a persistent TCP connection between controller and worker nodes.

Terraform and Pulumi [44], [45] These are IaC solutions widely used in the industry. These are specialised in provisioning pieces of infrastructure such as virtual machines or containers, but also extend to other software such as databases. These integrate with solutions such as OpenStack [5], AWS [6], Kubernetes, etc. A common API is provided to integrate all these solutions together. Modules with their respective properties and dependencies can be declared using a declarative DSL or general-purpose languages (e.g., Javascript, Python). The state of the system is represented as a graph of modules. Each module in the graph is deployed in the system. Each edge represents a dependency between deployed modules. A central managing system is responsible for applying changes on the target system and updating the graph accordingly. Pulumi has predefined modules for a wide variety of targets, such as Cloud providers, container applications, Git repositories, etc. Users can also define custom modules with custom associated with the CRUD interface.

In Pulumi and Terraform, *create*, *delete*, *re-create* and *update* operations are available. Relationships such as dependencies between modules can be represented using connections. Additional information about the kind of data provided by the dependency to other modules can be implemented. This means that Terraform and Pulumi support all types of module operations. As dependencies can be expressed between modules, Terraform and Pulumi support *module*-based parallelism. Finally, in both solutions, different orchestrators can be defined for a single infrastructure. This is possible as both solutions can put the state of the infrastructure in remote locations. However, for the same infrastructure, only one orchestrator can be executed at a time. This means that the execution is *centralised*.

Figaro [46] It is a lightweight reconfiguration framework optimised for very constrained systems. The solution relies on the Contiki OS. Each module is mapped directly to the underlying OS through a Contiki service. Modules may have dependencies with other modules at the scope of a single node. Dependencies provide information such as minimum version of the module, mandatory or optional dependency, static or dynamic dependency. Available module operations are *create*, *delete*, *re-create*. Modules also have the *custom* action *suspend*. A module is suspended when the component providing

[4] `https://docs.chef.io/custom_resources_notes/#custom-resources`

[5] `https://www.openstack.org/`

[6] `https://aws.amazon.com/`

services is under reconfiguration. A component is running when all of its dependencies are resolved. The solution provides *module*-based parallelism. When a new component is added, it waits until a timeout that all its mandatory dependencies are satisfied. After the timeout, it is discarded. A component can be updated if the new component version is greater than the old one, and the old component has no dependent services that have static dependencies.

Juju [7] It is an IaC solution where modules of the systems can be represented using component representation. It can be used to provision infrastructure or software, such as virtual machines, containers, databases, networks, etc. These components can be connected together. Juju provides workflows to coordinate and execute reconfiguration actions. Juju provides an interface to apply *create* and *delete* module operations on components. In terms of parallelism, it *module* parallelism.

Aeolus [47] It is a component-based reconfiguration solution using component-based representation to model modules and their dependencies. Using the component model, life-cycle of individual modules can be expressed. This life-cycle is customisable. The developer can add as many configurations as the module can be in. Dependencies between modules can be done specified at the scope of individual transitions. This means that Aeolus allows *inter-module* parallelism. Module can be created and deleted. As life-cycle can be customised, the *re-create*, *update* and *custom* operations are also supported.

Concerto [38] It is a component-based reconfiguration solution using component-based representation that emphasises parallelism when executing module operation. A more complete presentation of the solution is done in Section 4. The following only highlights the main properties relevant for the state of the art. In Concerto, the life-cycle of individual modules is fully customisable. Module operations are also customisable. Therefore, all types of module operations can be executed on modules. Contrary to the previously presented solutions, connections between modules can be done at the level of individual transitions between modules and inside the same module. This allows the *inter-module* and *intra-module* parallelism.

3.1.5 Discussion

Table 3.1 sums up the solutions presented in this state of the art and their properties. As seen in the table, Muse provides at the same time decentralised reconfiguration capabilities and almost all types of module operations (except custom). However, Muse has the following limitations for the use case of this manuscript (e.g., CPS with sleeping nodes).

Muse limitations Muse is limited to *module* parallelism. This limitation doesn't allow experiments using *intra* or *inter-module* levels of parallelism. Having a high level of parallelism should theoretically allow faster reconfiguration execution. Such level of parallelism could be relevant for the use case of this manuscript (DAO).

Muse doesn't cover the *custom* type of module operation. Such an operation can be beneficial when designing the modules of the nodes. The ability to precisely express which changes to perform on the nodes would allow time and resource saving. For instance, the ability to temporarily suspend the activity of a module without completely removing it.

[7] `https://jujucharms.com/`

	Action type	Level of parallelism	Centralised/decentralised
Ansible [43]	*create*	module (different hosts)	centralised
Chef [48]	all-{*custom*, *update*}	module	centralised
Terraform [44]	all-{*custom*}	module	centralised
Pulumi [45]	all-{*custom*}	module	centralised
FiGaRo [46]	all-{*update*}	module	centralised
Juju [49]	all	module	centralised
Aeolus [47]	all	inter-module	centralised
Concerto [38]	all	intra-module/inter-module	centralised
Muse [42]	all-{*custom*}	module	decentralised
Wild, K. et al. [41]	*create*	module	decentralised
Herry, H. et al. [40]	*create*	module	decentralised

Table 3.1: Summary of reconfiguration solutions and their properties. In the "Action type" column, "all" means that the solution covers all types of module operation. The notation "all-{*operation*}" means that the solution is able to perform all operations minus the ones mentioned between braces.

The Muse prototype relies on Pulumi, which mainly targets Cloud environments. Muse prototype has been validated on the AWS Cloud on a specific use case [42] and non-constrained nodes. Currently, the Muse prototype imposes a high workload on the CPU, which is limiting for nodes that are highly limited in resources such as CPU power and energy. This is described in Chapter 5, where experiments are conducted on Raspberry Pis using the Muse prototype. Due to the high workload imposed by Muse on the CPU, the performance of the reconfiguration decreases and interferes with the results.

Concerto's properties Concerto provides interesting properties for the use case of this manuscript. Its design allows it to have the highest level of parallelism among all solutions and is able to execute all types of module operations. However, the solution is centralised. In this manuscript, a decentralised version of Concerto is proposed. This solution inherits the properties of Concerto (level of parallelism and types of module operation). The solution is called Concerto-D and is presented in Chapter 4. A comparison of Muse and Concerto-D is presented in Chapter 5.

3.1.6 Conclusion

Lots of reconfiguration solutions allow users to design distributed software using modules. Modules can be created and connected to represent such software. Connections are used to represent dependencies between modules. Reconfiguration of modules is coordinated according to connections. Most of the time, a central controller is responsible for the execution and coordination of changes in modules.

While limited, literature on decentralised reconfiguration provides key concepts that can be used to design decentralised reconfiguration solution. The main concept is the ability to connect modules belonging to different systems. These connections share the same characteristic as connections between local modules, except they need network communication to be resolved.

Concepts learned from the state of the art are used to design a decentralised reconfiguration solution named Concerto-D. This solution is based on Concerto, a centralised reconfiguration solution providing high level of parallelism and all types of module operations. This solution is presented in Chapter4.

3.2 Reconfiguration of observatory CPS

As stated in Section 2.1, the environment where CPSs are deployed can have an impact on available resources for CPS nodes. Nodes of the CPS may have different sizes and have different amounts of available resources depending on observation sites [3]. This section aims at presenting examples of observatory CPS and their associated constraints in terms of energy and connectivity. Then, it gives examples of reconfiguration capabilities of such CPSs. Section 3.2.1 provides examples of observatory CPSs deployed in environments with different constraints. Section 3.2.2 shows typical reconfiguration capabilities of CPS nodes.

3.2.1 Observatory CPS environments

Environments where observatory CPSs are deployed vary in terms of constraints imposed on the system. This subsection focuses on energy and network related constraints.

Urban environments CPSs deployed in urban environments (e.g., cities, homes) usually benefit from their rich network and energy infrastructures [50]. The access to the energy grid is enabled notably through power sockets. Using such power sockets, batteries of CPS nodes can be regularly recharged. For instance, in [51], sensor nodes are attached to public electrical bicycles to collect usage data. These sensors rely on batteries shared with the bicycle. These batteries are replenished each time the bicycle plugs in a docking station. When not docked, low-power protocol is used for communication (LoRaWAN). When docked, sensors report all collected samples to a central station using Wi-Fi communications using Wi-Fi endpoints provided by the urban environments.

Another example is given in [52]. A set of sensors is attached to wristbands measuring health data. These wristbands provide energy supply to sensors, which can be manually recharged by users. Sensors report to beacons spread across the medical building. These beacons have constant energy supply and are able to receive data from sensors at short distance. These beacons collaborate by creating and maintaining a mesh network across the building. This allows data received by any beacon to be forwarded in the network until reaching a gateway.

Wild environments Wild environments such as protected natural areas, forests, oceans, mountains, etc. have lower amount of resources compared to urban environments [50]. This usually leads to more constrained CPS deployments in terms of network connectivity and energy supply. To alleviate such constraints, different methods can be used. This includes deploying sources of energy supply, providing dedicated or shared network infrastructure, deploying nodes using few computing or networking resources, and putting nodes to sleep.

Sources of energy supply include batteries. These batteries can be embedded in nodes to provide limited energy supply for a certain period of time. These batteries can be regularly replaced by physical intervention on the system. When possible, these batteries can also be refilled using energy harvested from the environment. For instance, solar energy can be harvested using solar panels. These panels can provide energy according to the exposition of the site to sun, and amount and size of solar panels [3]. Energy can also be harvested from the wind using turbine. These panels can provide energy according to the frequency and intensity of the wind, and the amount and size of these

turbines [6]. Other sources of energy can be harvested to a lower extent, such as energy from radio frequency (e.g., extracted from passive RFID tags such as in [4]).

Providing network infrastructure includes the deployment of dedicated nodes for communications, or the ability for nodes to work as a peer-to-peer network. These approaches can be combined, where different parts of the network form a peer-to-peer network (e.g., sensor nodes), and the other part forms a relay network to enable connectivity between the CPS and the external world. In [29], a network of relay nodes is deployed and links together multiple observation sites. Nodes in these observation sites are located close together and can communicate using peer-to-peer connections. Relay nodes are connected to a base station. This allows scientists to send commands to the CPS and collect observation data. This CPS, coupled with solar panels to refill batteries, can be autonomous for more than two years.

Nodes of the CPS may not have sufficient harvestable energy available or proper tools to harvest it. For instance, in ocean observatories, some nodes are immersed at the bottom of the sea to perform seafloor observation [53]. In some cases, underwater nodes may be cabled to a shore station, such as in [27]. The cabled infrastructure provides connectivity and regular energy supply to the system. In cases where nodes cannot be cabled, nodes have to rely on limited energy budget and cannot recharge their batteries. In [54], underwater nodes are located far from the shore, and immersed at locations up to 2000 m for more than a year. Underwater nodes are not fully isolated as they can communicate using acoustic waves. A surface buoy translates acoustic into electromagnetic signal. This buoy may be equipped with solar panel to increase its lifetime. The buoy itself is connected to a shore station, providing connectivity between the CPS and the external world.

Another example is [28]. In this paper, a CPS is deployed in Swiss Alps in high position, up to 3500 m on the cliff of mountains. Due to extreme temperatures, especially in winter (-30°C), nodes are physically unreachable between 7 and 8 months. This imposes severe constraints on nodes' activity to limit energy consumption, especially in terms of communications and computing. Few nodes of the CPS are equipped with GSM antennas. These nodes serve as gateway between the system and the external world. Nodes with GSM antennas propagate information to other nodes in multi-hop manner. Nodes have a very low duty-cycle (nodes are up 0.003% of the time). This limits the amount of data nodes can exchange within the CPS and with the external world.

In the Arctic Tundra, nodes do not have access to regular network coverage such as GSM or other cellular networks. From a connectivity perspective, this makes them isolated most of the time from the external world. Network infrastructure is absent and forces nodes to rely on peer-to-peer communications. Nodes must rely on small batteries with no external energy source. Most nodes deployed in observation sites have limited computing, networking, and storage resources. These nodes include few single-board computers and microcontrollers. Nodes should adopt policies to extend their lifetimes, such as entering long sleeping periods.

3.2.2 Reconfiguration capabilities of CPSs

Reconfiguration can be done manually by humans through remote access to the system or automatically by the nodes. Reconfiguration capabilities range from small parameter change to replacement of the whole software running on nodes. The following categorises reconfiguration capabilities in two categories. The first category corresponds to changes in parameters or configuration, such as sampling frequency or activation/deactivation of

sensor. It aims at providing adjustments that do not require updating existing software or deployment of new software. The second category is the deployment of new software or update of existing software. It aims at sending the new or updated software to nodes of the CPS, which receive and install this software locally on their system.

Changing parameters or configurations For sensor nodes, parameters change include sampling rate, sample value threshold, etc. For instance, in [30], authors deploy a sensor network to monitor the habitat of wildlife fauna. Each sensor extracts samples from the environment (e.g., temperature) and is accessible from outside. Sensors can be individually selected and parameters to change the rate at which temperature is sampled, and stop sampling when temperature is too low (e.g., at night). The commands to change these parameters are encapsulated in network maintenance packets that are extracted on reception by nodes of the CPS to save bandwidth and energy [30].

Parameter change can be executed on nodes directly by sending scripts to nodes. In [28], a CPS is deployed in the Swiss Alps for monitoring the environment. Each sensor node can be reconfigured through the transmission of scripts. These scripts allow to control sensor activity, such as activating/deactivating sensor and halt/resume the transmission of its samples. These parameters are typically related to saving of energy. Halting the transmission of samples allows each node of the CPS to save energy by not transmitting data.

Actuators can be deployed along with sensor nodes when reconfiguration requires physical intervention. In [55], nodes of the CPS are used to monitor and control the population of possums in sites located in New Zealand. Toxin delivery devices are deployed in an area and distributed toxin dose to regulate the possum population. Actuators allow control of these devices, and activate or not the delivery of the toxin. Additionally, the system provides a graphical interface to users to ease monitoring and reconfiguration. Nodes repeatedly report data about number of individual toxin deliveries, remaining gas level for toxin shots and time between toxin delivery events. These data are collected in a centralised framework in the Cloud that exposes these data to users through a graphical interface. Using these data, users can send commands to put sensor nodes to sleep or change the frequency of reporting of collected data. A similar feature is present in [29]. In this system, each node can have multiple sensors attached to it. These sensors are controlled individually using a dedicated programmable system on chip.

Rewriting software When more drastic changes must occur, the software present on nodes may need to be rewritten. Monolithic software has to be updated as a whole [30]. In this case, the whole software is sent and rewritten on the nodes. While simple, this approach may result in lots of wasted resources (e.g., energy consumption, bandwidth), as part of the software that does not need to be updated is rewritten as well [56]. This is why lots of reconfiguration solutions adopt module reconfiguration, where only part of the software can be changed [57].

An example is Remora [58], which proposes a reconfigurable module-based software architecture. Software deployed on nodes is divided into modules that can be individually reconfigured (see Section 3.1). Module-based architecture allows fine-grained reconfiguration of the application compared to full software rewriting (including OS). Authors compare energy spent for full software rewriting (34 KB) compared to individual modules of 1 KB. Results show that the energy spent is an order of magnitude lower. The energy overhead paid for restructuring the modules (e.g., updating links between old and new modules) is much lower compared to the replacement of the full software.

Before its installation, a module must be transmitted and received by the node. Solutions have been developed to inject modules in the network and automatically distribute these modules to targeted nodes of the CPS (called dissemination). Nodes automatically distribute these data to target nodes. Target nodes include nodes with outdated software [59], nodes with sufficient energy to perform update [46], etc. Techniques have been studied to minimise the impact of such distribution on nodes, by reducing the size of update [56] or reducing redundancies during dissemination [60].

Nodes of the DAO-CPS would not be accessible from outside for long periods. For this reason, reconfiguration has to be conducted and coordinated directly from the CPS. Having the ability to define modules would allow fine-grained reconfiguration of software present on nodes. This reconfiguration includes rewriting the software of the individual module or changing their configuration. Such a reconfiguration has an energy consumption overhead. The next section describes different sources of energy consumption due to reconfiguration and presents different techniques to study this energy consumption.

3.3 Energy consumption due to decentralised reconfiguration of CPSs with sleeping nodes

As seen in Section 3.2.2, reconfiguration capabilities on CPSs often involve a central entity distributing update data to nodes of the CPS. This data can be simple parameter change or large update of software modules. This manuscript aims at studying the coordination of a reconfiguration where a central entity is not involved. It doesn't deal with the distribution of large update data, but only with the coordination of changes by the nodes of the CPS. Additionally, it aims at studying such a reconfiguration in a context where nodes sleep. Sleeping allows to extend nodes' lifetimes by saving energy, but may decrease the connectivity of nodes and increase the duration of the reconfiguration. In this manuscript, a potential leverage to help sleeping nodes to communicate is the Relay Node (see Section 2.3). In this manuscript, the Relay Node is a special type of node able to be awake at the same time as its neighbours. It can be used as an intermediary to help communication between the other sleeping nodes. It can temporarily store messages while the recipient is sleeping, and send these messages to the recipient when it wakes up. This section aims at providing a literature review regarding the impact on energy consumption of the execution of reconfiguration actions, communication, disruptive network, sleeping node, radio technology, and Relay Node usage to help other nodes communicate. Studying such an impact can be done using different techniques, such as physical prototyping and simulation.

3.3.1 Impact of decentralised reconfiguration on energy consumption of CPSs with sleeping nodes

Coordinating a decentralised reconfiguration between nodes includes the execution of reconfiguration actions and communications between nodes due to coordination. In peer-to-peer networks, such communication is enabled by forwarding data between nodes until a recipient is reached. When dealing with sleeping nodes, forwarding data is difficult, as network links between nodes are frequently disrupted. Different policies have been studied to ease communication due to disruption caused by sleeping nodes. Additionally, the radio technology used by the nodes to communicate also has an impact on energy consumption.

Impact of the execution of reconfiguration actions As a reminder, reconfiguration actions consist in changing software modules running on the system (see Section 2.4). These actions include changing nodes' runtimes (e.g., activating/deactivating services, sensors), changing parameters (e.g., sampling rate), rewriting software on RAM or disk, etc. The energy consumption due to these activities can vary. This depends on the workload put into the different devices responsible to handle reconfiguration actions. Rewriting software is one of the most energy-consuming reconfiguration actions [58]. The energy consumption depends on the intensity of the workload. For instance, large software to rewrite imposes high activity on the RAM or storage device [56].

In this manuscript, the energy consumption due to the execution of reconfiguration actions is abstracted. Coarse-grained energy consumption values for the execution of reconfiguration actions are considered. In the energy consumption evaluation part of this manuscript (Section 6), it is considered that each reconfiguration action fully stresses the RAM for the duration of the action. This is due to the fact that rewriting software during reconfiguration is very common [58].

Impact of communication during decentralised reconfiguration When dealing with decentralised reconfiguration, nodes must communicate to coordinate the execution of their actions. These communications can be simple messages sent directly to the node such as in client/server architecture [42], or more complex procedures involving middleware such as in publish/subscribe architecture [61]. The impact of such communication on energy consumption during decentralised reconfiguration has not yet been studied. This is, to the best of my knowledge, due to the few contributions dealing with decentralised reconfiguration.

In the literature, the impact of communication on energy consumption during reconfiguration has been studied mainly for the distribution of large update data to nodes of the CPS. This is notably the case for contributions dealing with the dissemination of these data to nodes. These contributions are centralised, and consider a central base station that injects the update, which is propagated to other nodes of the network [57]. These contributions deal notably with reliable and efficient dissemination of these data by reducing the size of the data [56] or automatically detecting and requesting out-of-date software [60]. While these papers are relevant to assess the energy consumption due to the communication of large update data, they do not address the impact of communication inside the CPS due specifically to the coordination of reconfiguration actions. Studying the energy consumption overhead of communications due to coordination is part of the contributions of this manuscript.

Impact of radio technology The radio technology is used by nodes to wirelessly send and receive messages from other nodes. A node sending messages is received by all nodes in range, when no obstacle prevents communication (see Section 2). The range, bandwidth, and energy consumption of the radio vary according to the type of radio technology. Some radio technologies have low-power and long-range features to the cost of low bandwidth (e.g., LoRa and NB-IoT, see Section 2.5) [33]. These characteristics can be interesting in reducing energy consumption due to communication while allowing the deployment of peer-to-peer networks in large areas. This is to the cost of having potentially slower communications due to low bandwidth. For this reason, simulations realised in Section 6 of this manuscript are calibrated using the characteristics of either LoRa or NB-IoT to study this trade-off.

LoRa operates at low power in the unlicensed sub-GHz. This frequency allows long-range communications and a degree of obstacle penetration [33]. This improves the reachability of ONs located far from each other and covered in snow, ice, rock, etc [62]. NB-IoT also operates at low power, in the licensed radio spectrum. NB-IoT has slightly more range than LoRa and higher throughput, but consumes more energy.

Impact of sleeping node on connectivity Works regarding Delay/Disruptive Tolerant Networks (DTN) address challenges involving nodes with intermittent connectivity that need to communicate. These works include studies on routing protocols and their efficiency in terms of energy consumption and latency [63]. Such papers helped to choose an epidemic-like communication pattern for the contributions of this manuscript (see Section 6). This choice is due to the reliability of the epidemic-like pattern when having no knowledge about the network [64]. Papers related to DTNs often deal with finding energy consumption/duration trade-offs using different uptime schedules. In the experiments of this manuscript, the uptime schedule of nodes is mainly considered random and non-modifiable. However, contributions from the DTN could be relevant in future works for scenarios where uptime schedules of nodes are considered modifiable.

Some papers dealing with the DAO use case have been studying different policies for uptimes to increase connectivity between nodes by having extended or additional uptimes while keeping the initial ones. In [65], authors conducted experiments for disseminating update data across sleeping nodes. Experiments show that extending uptime duration decreases the amount of transmission failure and allows the dissemination to finish faster at the cost of increased uptime duration. In [8] authors studied different policies for data dissemination in networks with non-synchronised sleeping nodes. As in [65], a source node is sending 1 MB of data to 12 target nodes. Policies are: extending the uptime duration of both sender and receiver until the end of transmission, or/and hinting to receivers of the next uptime of the sender, to facilitate future transmission. Both policies allow to reduce the amount of failure, at the cost of increased energy consumption due to additional uptimes.

Impact of the usage of a Relay Node to ease communication In the literature, Relay Nodes (RN) are often nodes with different properties compared to other nodes [66]. This includes longer radio range, higher energy budget, lower hop count to other nodes, etc [67]. RNs are essential in many CPSs relying on cluster architectures [66]. In these architectures, RNs are used as cluster heads to gather messages from other nodes and forward them to the base station. RNs can be chosen as the result of an election by other nodes (e.g., nodes with the best properties for being an RN) [68]. This manuscript does not deal with cluster or clustering algorithms. It considers the RN already defined, and chooses to study directly the benefits of using an RN in terms of energy consumption and reconfiguration duration (see Section 6). Studies such as RN election could be done as future work.

3.3.2 Techniques to study energy consumption of distributed systems

Different techniques have been developed to study energy consumption. It can either be physically measured using dedicated hardware or estimated using energy models [69].

The former technique requires physical prototyping, the latter simulation or analytical study.

Physical prototyping Platforms and tools have been designed to study energy consumption for physical measurement (i.e., where a node in the experiment is studied using real hardware) [70]. By doing so, instruments can be deployed along the node to monitor its energy consumption. These instruments can be physical instruments such as multimeters or direct current sensing devices [71]. These instruments can also be software-based and hosted on the node they monitor. In this case, they monitor the activity of the node and use energy models to estimate its energy consumption [72, 73].

Experiments using real hardware allow to have high accuracy in measurements, such as high sampling frequency (e.g., kHz) and high precision (e.g., sub-μJ) [74]. Some testbeds propose outdoor and wireless setups for experiments under real-world conditions [69, 75]. Such features are relevant when studying observatory CPSs. However, for scenarios involving several months of deployments with a high number of nodes, real environments are expensive in time and resources. This may not be practical when trying to study specific algorithms involving lots of experiments.

Simulated environment Simulators can be a relevant and complementary alternative to physical prototyping. They can use the energy consumption values measured using physical hardware to calibrate their energy models and estimate energy consumption of simulated nodes. Simulators provide time-simulated environments. Simulators provide a framework to simulate node's activities such as sending and receiving messages. Large number of nodes can be quickly simulated, up to hundreds or thousands depending on the performance of the simulator. Nodes are represented as software components which alleviates the user from handling specificities of the underlying hardware [70].

Simulators can provide models to simulate the system and estimate energy consumption due to nodes' activities. For instance, in [76] a model for wired and wireless communications is used to conduct simulations. This model allows to parametrise latency and bandwidth of individual links between nodes. An energy model can be used to estimate energy consumption due to the sending and receiving of data. These models must be calibrated using values from relevant sources or measured by the experimenter for specific hardware.

Contributions of this manuscript use simulation to evaluate the energy consumption for the DAO use case. This is mainly due to the very long sleeping periods of nodes. In such a context, several years of experiments can be reduced to a few hours of simulation. This allows to experiment with lots of different scenarios.

3.3.3 Conclusion

Reconfiguration of distributed systems involves energy consumption due to execution and coordination of reconfiguration actions. Evaluation of this energy consumption can be done notably using physical prototyping or simulated environments. Simulated environments allow performing experiments with a high control over important parameters, such as the number of nodes, the network topology, bandwidth/latency between network links, etc. Simulations run in time-simulated environments, allowing to reduce years of experiments in real environments to few hours of simulations. Simulators can use energy model to estimate energy consumption of simulated nodes. With proper calibration, such models can yield accurate energy estimations.

The impact of the execution of reconfiguration actions on nodes varies significantly. For instance, it can vary according to the size of software data to rewrite. For this reason, experiments in this manuscript consider that reconfiguration actions fully stress the RAM of the node. The energy consumption due to this stress can be given by measurement obtained using physical prototyping (i.e., measuring energy consumption when fully stressing the RAM).

The impact of sleeping nodes and disruptive networks has been studied mainly for controllable uptimes. This may not be suited for the DAO use case where each node has its own and not predictable uptime schedules. The Relay Node introduced in the DAO use case is used to help nodes with scarce connectivity to communicate. Studying the impact of having an RN during the coordination of a reconfiguration is part of the contributions of this manuscript. However, the election of this RN is considered out of scope.

Papers dealing with reconfiguration often deal with dynamic energy consumption, related to activities such as communication and execution of reconfiguration actions. In the DAO-CPS context, static energy, related to nodes' uptimes is also an important metric. Waking up nodes solely for reconfiguration may have a significant impact on energy consumption. Evaluations performed in this manuscript should take into account this type of energy to fully grasp the total energy consumption spent for reconfiguration.

With these design aspects in mind, experiments have been designed and described in Chapter 6. Results are presented and compared depending on different parameters related to the DAO use case. These experiments use a decentralised reconfiguration solution as reference. This solution is Concerto-D, which is presented in the next chapter.

Chapter 4

Concerto-D, decentralised reconfiguration solution

Contents

This chapter presents Concerto-D, a solution to provide decentralised reconfiguration capabilities to a system. Concerto-D allows the execution of a reconfiguration program to be suspended and resumed, allowing reconfiguration on sleeping nodes. The implementation also features different communication paradigms, which are client/server (e.g., direct) [42] and publish/subscribe (e.g., indirect) [61].
Concerto-D is based on Concerto, a reconfiguration solution for distributed systems [14]. This allows Concerto-D to inherit from Concerto capabilities, which features a language and model to express generic reconfiguration actions and execute them with a high-level of parallelism. Concerto-D aims at extending the language and model of Concerto to allow multiple instances hosted on different nodes to coordinate a reconfiguration.

4.1 Introduction

As seen in Section 3.1, very few contributions deal with decentralised reconfiguration. Among these solutions, none provides a high level of parallelism in the execution of module operations. This chapter presents Concerto-D, a solution enabling decentralisation and high level of parallelism. To this end, Concerto-D is based on Concerto. Concerto is a centralised reconfiguration solution allowing fine-grained definition of the execution of module operation using a reconfiguration language. This chapter describes how Concerto-D extends Concerto and its language to provide decentralisation capabilities. An overview of Concerto is given in Section 4.2. Concerto-D is presented in Section 4.3. The validation of decentralisation and high level of parallelism capabilities of Concerto-D is presented in Section 4.4.

4.2 Concerto overview

The content of this subsection is inspired by the PhD thesis [14] and paper [38] that present Concerto. Concerto is a centralised reconfiguration solution used to model module life-cycles, and dynamically change the modules state and architecture. In Concerto, a module is called a component. A reconfiguration language with a specified semantics is provided to modify components and their interactions. For the interested reader, the full formal semantics of Concerto has been detailed by its authors in [38]. This section focuses on the concepts of Concerto required to understand Concerto-D. Notably, component definition, architecture, life-cycle, and actions issued from the Concerto language.

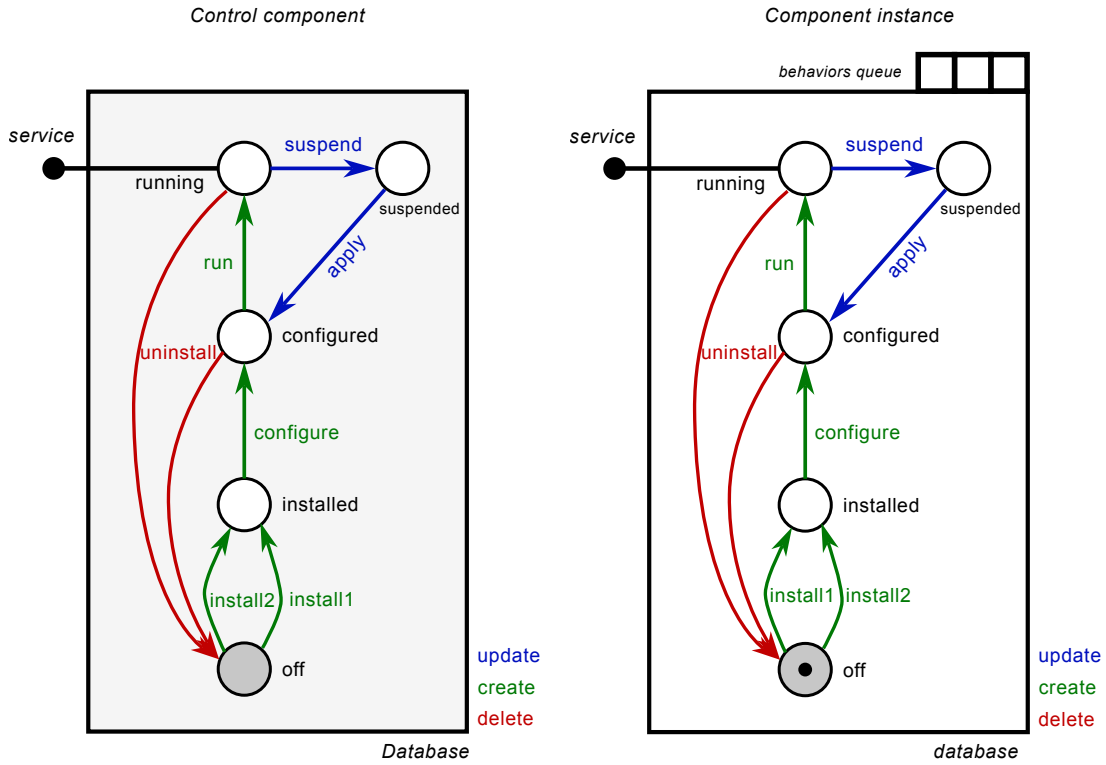
4.2.1 Control component

A module in Concerto is modelled as a *control component*. A *control component* defines the life-cycle of the module and its use/provide ports. Use and provide ports are used to connect different *control components* together. This is detailed later in the subsection.

The representation of a *control component* is presented in Figure 4.1a. In Concerto, places represent the different milestones in the reconfiguration of the component. The depicted *control component* has five places, depicted as circles: `off`, `installed`, `configured`, `running` and `suspended`. The initial place is `off` and depicted in grey. It is the place in which the *control component* is when instantiated.

To reach the different places, Concerto uses transitions. Each transition is associated with a set of concrete changes to execute to reach the next place. For instance, execution of a shell script to install or activate a service. These transitions are represented by arrows that connect the different places together. For instance, in Figure 4.1a, the transitions *install1* and *install2* start at the `off` place and end at the `installed` place. The `installed` place can be reached from the `off` place by executing both *install1* and *install2* transitions.

In Concerto, transitions are grouped by behaviour. Behaviours are interfaces that allow user to interact with the component (similar to CRUD actions in solutions such as Pulumi or Terraform as seen in Section 3.1). Activating a behaviour triggers the execution of its associated transitions. This provides a convenient way to apply changes without having to know the internal life-cycle of components. In Concerto, transitions and their



(a) A database represented as a CONCERTO *control component*. It is extracted from Listing 4.1. The module operations are called behaviours. The depicted *control component* has three behaviours: *update*, *create* and *delete*.

(b) Instance of a database named *database*. When first instantiated, a CONCERTO component has its token located in the initial place. A behaviour queue located at the top right indicates the queued behaviour to be executed. Behaviours are executed from left to right.

Figure 4.1: Representation of a database using a CONCERTO component.

associated behaviours are represented using colors. For instance, in Figure 4.1a, the *create* behaviour is associated with the *install1*, *install2*, *configure* and *run* transitions. This behaviour is represented in green. When the *create* behaviour is activated, its associated transitions are executed following the order of the arrows.

Listing 4.1 depicts the CONCERTO code for the declaration of the Database *control component*. Each transition is associated with a Python function (listed at the end of the listing). Users can define the concrete change to execute for each transition in each function.

The *control component* can be instantiated to a *control component instance*. The *control component instance* represents the piece of software running on the infrastructure. For the sake of simplicity, a *control component instance* is simply called component. The instantiation of the *control component* in Figure 4.1a is presented in Figure 4.1b. The instance features three additional elements. First, a unique ID for identification purposes (e.g., database). Second, a behaviour queue. The behaviour queue is used to register behaviours to execute and execute them in FIFO order. It is represented as three squares at the top right of the component. Third, a token created in the initial place (black dot). In CONCERTO, tokens are used to track the state of the reconfiguration for each component. The state of the reconfiguration for each component is composed of the reached places and transitions being executed. Multiple tokens can be used to represent parallel execution of transitions (detailed in the next subsection).

```
class Database(Component):
  def create(self):
     self.places = [
        'off',
        'installed',
        'configured',
        'running',
        'suspended',
     ]

     self.transitions = {
        'install1': ('off', 'installed', 'create', 0, self.install1),
        'install2': ('off', 'installed', 'create', 0, self.install2),
        'configure': ('installed', 'configured', 'create', 0, self.cfg)
   ,
        'run': ('configured', 'running', 'create', 0, self.run)
        'suspend': ('running', 'suspended', 'update', 0, self.suspend)
        'apply_u': ('suspended', 'configured', 'update', 0, self.
   apply_u)
     }

     self.dependencies = {
        'service': (DepType.PROVIDE, ['running']),
     }

     self.initial_place = 'off'

 # Implement concrete changes for each transition
  def install1(self): ...
  def install2(self): ...
  def configure(self): ...
  def run(self): ...
  def suspend(self): ...
  def apply_u(self): ...
```

Listing 4.1: CONCERTO code of the Database control component from Figure 4.1a. Each transition is associated with a Python function. Python function apply concrete changes on the node.

To execute a behaviour, this behaviour has to be pushed in the behaviour queue. The current behaviour is the first behaviour starting from the left of the behaviour queue. The component executes transitions associated with the current behaviour. For instance, Figure 4.2 depicts the creation of the *database* component. The behaviour *create* has been pushed into the queue. Starting from the initial place, the transitions associated with the *create* behaviour are executed until none are left. This leads to the token reaching successively the `off`, `installed`, `configured` and `running` places (depicted by the red tokens). The behaviour is then popped from the queue, and the next behaviour in the queue, if it exists, takes its place. The execution of a transition follows different steps described thereafter.

4.2.2 Execution of transitions

This subsection describes the execution of transitions in CONCERTO components. First, it describes such execution for single transition. Second, it describes a parallel execution

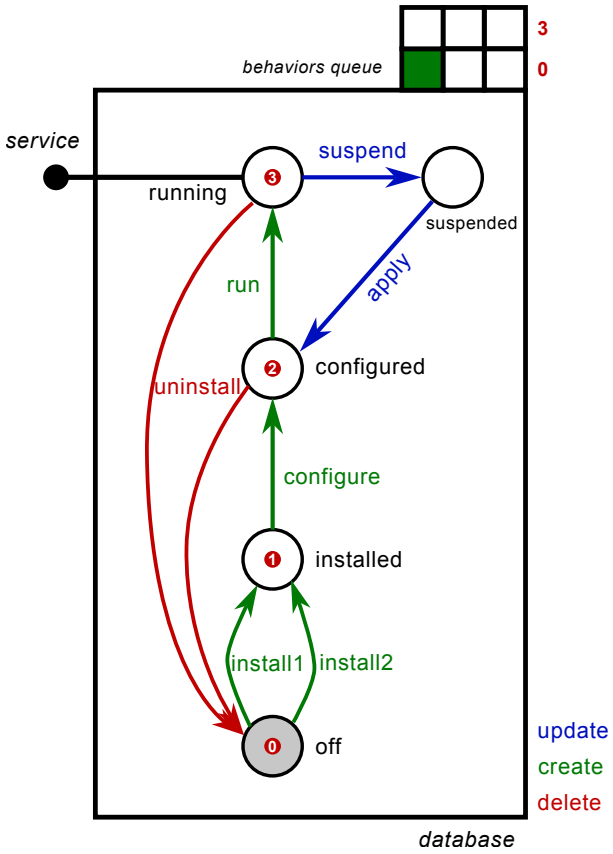


Figure 4.2: Creation of the *database* component.

of transitions.

Single transition execution Figure 4.3a provides a detailed view of the execution of a transition in a CONCERTO component. This figure focuses on the execution of the `run` transition of the database component, going from the `configured` to the `running` places. The movement of the token is represented using numbers. Each number is associated with the name of the steps. The following describes each step.

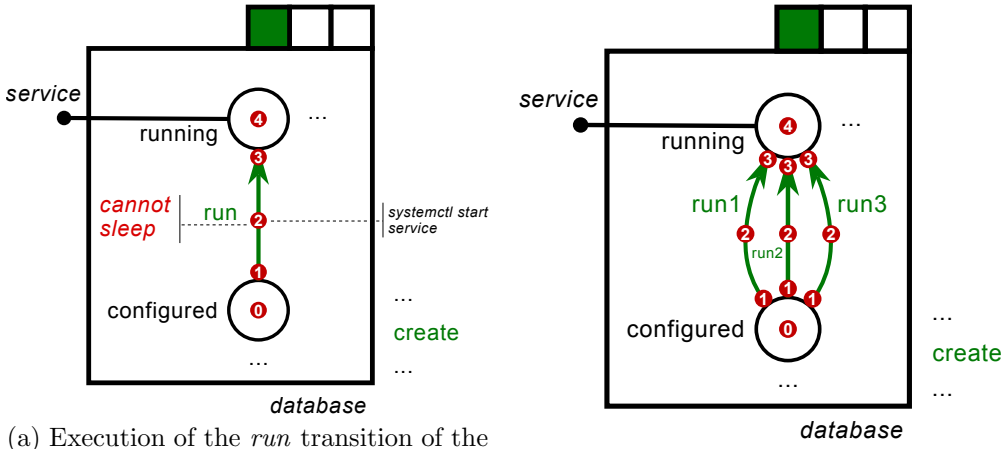
0. The token is in the `configured` place. The current behaviour is *create*. The `run` transition is associated with the *create* behaviour and originates from the `configured` place. This transition must be executed.

1. The token leaves the `configured` place.

2. Concrete changes associated with the transition are executed. In this example, it starts a service using systemd. The "cannot sleep" notation is used for CONCERTO-D, presented in Section 4.3.

3. Execution of concrete changes is complete. The token wait before entering the `running` place.

4. When the token enters the `running` place, no more transitions associated with the *create* behaviour remain. This behaviour is popped from the behaviour queue. The database is considered running and can provide service to potential clients.



(a) Execution of the *run* transition of the database component.

(b) Example of parallel execution of three transitions.

Figure 4.3: Execution of transitions

Multiple transitions, parallel execution Figure 4.3b depicts an example of parallel transition execution. In this case, multiple tokens are used to represent the state of the execution of each transition. The following describes each step of the execution.

0. The token is in the `configured` place. The current behaviour is *create*. The `run1`, `run2` and `run3` transitions are associated with the *create* behaviour and originate from the `configured` place. All three transitions must be executed.

1. The token leaves the `configured` place and gets split into three to track the state of the execution for each transition.

2. Concrete changes associated with each transition are executed. Note that, in real conditions, transitions may not always be executed at the same time. For instance, if resources cannot be allocated for the execution of all three transitions at the same time (e.g., number of cores).

3. The execution of all three transitions must be complete before entering the `running` place.

4. When the execution of all three transitions is completed, their associated tokens are merged into a single one, which enters the `running` place. No more transitions associated with the *create* behaviour remain. This behaviour is popped from the behaviour queue. The database is considered running and can provide its service.

4.2.3 Use/provide port and assembly

In a distributed system, nodes may use and provide services to each other. For instance, the database component presented above provides an endpoint for communication to one or multiple clients. Clients connect to this endpoint to store or query data. Concerto represents relationships between components with use and provide ports. A provide port represents an endpoint through which other components can connect. Use ports represent the active utilisation of an endpoint by a client. Provide ports can also be used to represent software dependencies such as a library. Ports can be activated to represent availability of service or data (provide port) or its usage by other modules (use

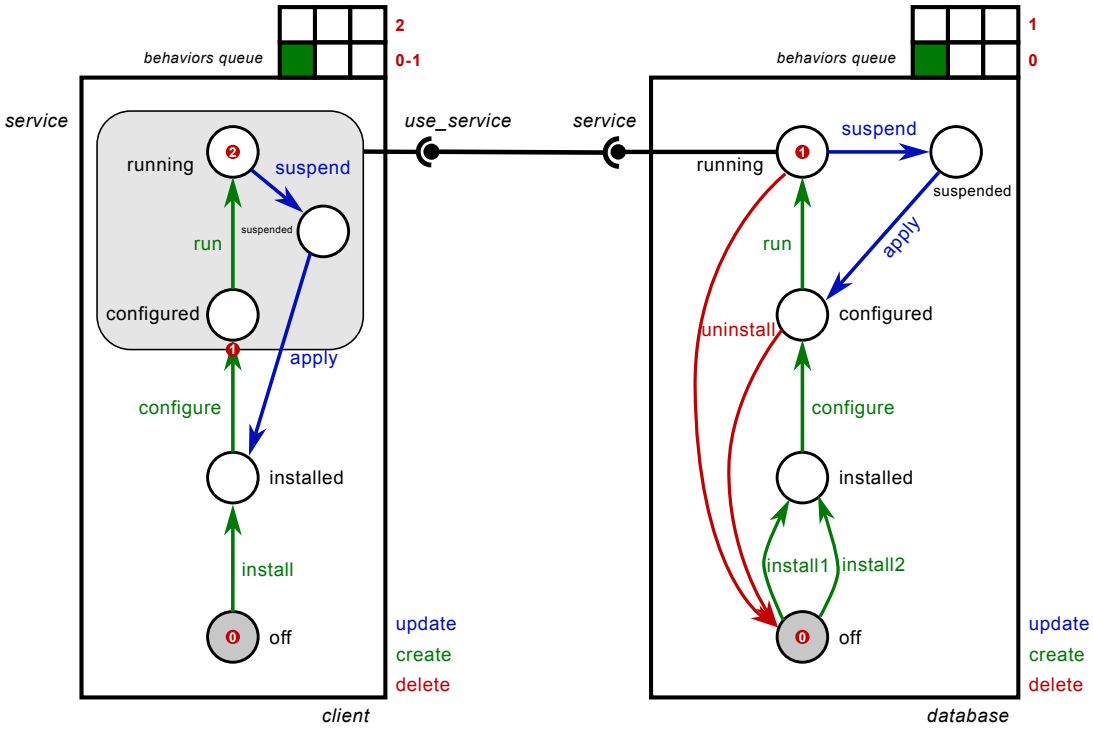


Figure 4.4: Assembly representing a client component connected to a database component. The client's use port is connected to the database's provide port. Note that the client's use port is used by both `configured` and `running` places. This is represented by the area grouping both places. The *create* coordination of both components is depicted.

port). Finally, a use port can be connected to a provide port. Multiple use ports can be connected to the same provide port.

In each component, ports are bound to one or more places. When ports are associated with multiple places, a group syntax is used for simplicity. A port is active when the token is located in at least one of its bound places. For instance, the service provided by the *database* component is activated when the component's token is in the `running` place. When the token leaves the place, the provide port is deactivated. A use port can be activated only when its connected provide port is activated. A provide port can be deactivated only when all its connected use ports are deactivated.

Figure 4.4 depicts an example of two connected components. A client component is connected to the database through its *use_service* use port. This port is associated with `running`, `suspended` and `configured` places. This is depicted graphically by having the *use_service* port attached to the group including these three places (grey area). Due to this connection, the *client* component cannot be in the `configured`, `running` or `suspended` places while the *database* component is not in its `running` place. However, both components can execute transitions prior to reaching these places.

The ability to define dependencies at place level allows to have *inter-module* parallelism (see Section 3.1). This level of parallelism allows two connected components to execute their transitions in parallel until a dependency is encountered. In the example of Figure 4.4, the *client* component can execute its `install` and `configure` transitions regardless of the place reached by the *database* component. However, it has to wait for the *database* to be in the `running` place to be able to execute its *run* transition. This example is given with more details in the next subsection.

In Concerto, components, places and connections form an assembly. Assemblies

```
sc = Assembly("client_database")
sc.add("client", Client)
sc.add("database", Database)
sc.connect("client", "use_service", "database", "service")
sc.push_b(f"client", "create")
sc.push_b(f"database", "create")
sc.wait(f"client")
sc.wait(f"database")
```

Listing 4.2: Reconfiguration program for the creation of the assembly depicted in Figure 4.4.

can be changed using the CONCERTO language. Components can be added or removed, connected or disconnected. Transitions of each component can be executed by pushing behaviours in the behaviour queue of each component. The assembly keeps track of the execution of each behaviour.

4.2.4 Concerto language

CONCERTO is equipped with a small reconfiguration language to control the assembly of components and triggers reconfiguration. This language provides six actions which are described in the following.

- add(id_c, t): add an instance of a *control component* of type t. Each instance is given a unique identifier id_c. The instance is added to the assembly. Once created, this component is in its initial place and has its behaviour queue empty. It is initially not connected to any other component;
- del(id_c): remove a component based on its id_c from the assembly;
- con(id_u, u, id_p, p): connect the use port u of component id_u to the provide port p of component id_p;
- dcon(id_u, u, id_p, p): disconnect the use port u of component id_u from the provide port p of component id_p;
- pushB(id_c, b): push the behaviour b to the behaviour queue of component id_c. It is asynchronous. It immediately returns after having pushed the behaviour;
- wait(id_c): action used for synchronisation purpose. This action pauses the execution of reconfiguration program until the component instance id_c has no more behaviour in its queue. Note that, when a component has no more behaviour in its queue, it is said to be idle.

Figure 4.4 provides an example of the deployment of a client and a database using CONCERTO actions. Listing 4.2 depicts actions used for this purpose. The client and database components are created from their respective *control components* (Client for client, Database for database). Usage of the database by the client is specified using the *connect* action. Then, the *create* behaviours are pushed in the queues of both components. The program waits for both components to finish the execution of these behaviours.

Execution of the program is shown in Figure 4.4. The following describes each step of the execution.

0. The client and database components are added to the assembly and connected. The *create* behaviour is pushed in both client and database components' behaviour queues. The reconfiguration process waits for both components to complete the execution of their respective behaviours.

1. The database component can execute all its transitions associated with the *create* behaviour. When it reaches the `running` place, the *service* port attached to the place becomes activated. The *create* behaviour is popped from *database*'s queue. At the same time, the client component can only execute the `install` and `configure` transitions before the *database* reaches the `running` place. It cannot enter the `configured` place without having the *database* reach the `running` place.

2. After the *database* reaches the `running` place, the *client* can execute the *run* transition and reach its own `running` place. The *create* behaviour is popped from *client*'s queue. As both *database* and *client*'s queues are empty, the *wait* actions of the reconfiguration process return, and the reconfiguration terminates.

As stated previously, the execution of Concerto is centralised. This means that the execution of each Concerto action is done by a single entity, which has knowledge about the whole assembly. In the following subsection, Concerto-D is presented. The main contribution of Concerto-D is the decentralisation of the execution of Concerto programs. This means the concurrent execution and coordination of multiple Concerto programs and their associated assemblies.

4.3 Concerto-D: decentralised version of Concerto

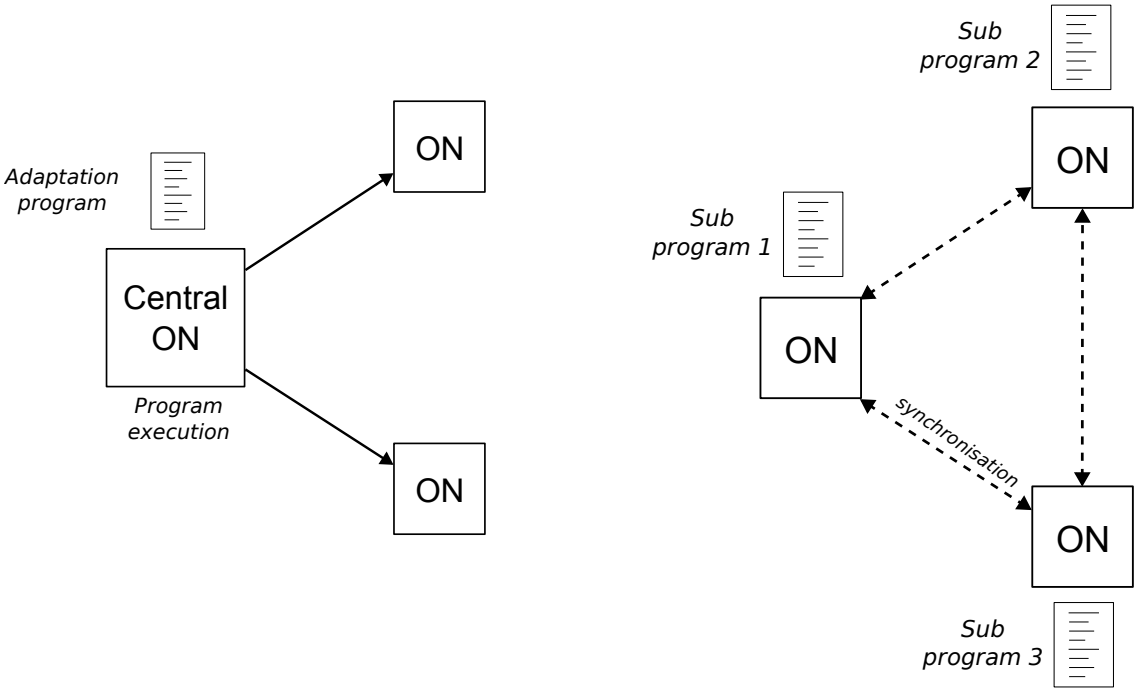
Concerto-D extends the semantics of Concerto language to allow execution and coordination of programs executed across multiple instances (Figure 4.5). To this end, it provides the Concerto actions and execution engine the ability to synchronise the execution across multiple instances. Communication between instances is enabled using client/server or publish/subscribe paradigms. Concerto-D can suspend and resume its execution, so that it can be hosted on sleeping nodes. A prototype of Concerto-D has been developed as a fork of Concerto integrating these elements.

4.3.1 Assumption

Concerto-D only deals with the execution and not the creation of the reconfiguration plan. In this subsection, it is assumed that the plan is known, correct, and has been distributed on nodes. Note that the computation of the plan can be done either by splitting and distributing an existing centralised plan or by computing this plan in a decentralised manner [77]. Such contribution is outside of the scope of this PhD.

Nodes have access to an inventory with the address (e.g., ip address) of other nodes, and which node hosts which Concerto-D components. When nodes try to contact other nodes, such as when trying to find the status of a connected port, they can use this inventory.

Failure of the reconfiguration is not covered. It is assumed that all nodes involved in the coordination of the reconfiguration are available at some point for the coordination to complete. All reconfiguration actions and execution of transitions are executed without having the node crash during their executions. In case of failure, it is considered that



(a) Concerto centralised reconfiguration plan execution.

(b) Concerto-D decentralised reconfiguration plan execution. In this example, the centralised program has been decentralised and distributed to individual nodes.

Figure 4.5: Reconfiguration plan execution for Concerto and Concerto-D.

the higher phases of the MAPE acknowledge the failure and deal with it. Notably, recomputation of a new plan to apply correction.

Finally, Concerto-D has the ability to control when and how the program is suspended. The user can decide when the program should be suspended, such as exiting after a deadline or on a specific software interrupt. Unanticipated interruption during reconfiguration, which causes the reconfiguration process to fail, is not covered in this manuscript.

4.3.2 From Concerto to Concerto-D

This part covers the evolution of the design of Concerto to reach Concerto-D. Changes applied to the model and language are described.

Distributed assemblies Assemblies in Concerto-D are identical to Concerto, the only difference being the ability to connect components in assemblies present on different nodes together. Connections between components follow the same principle as Concerto. A use port can be connected to a provide port, and multiple use ports can be connected to the same provide port. As for Concerto, assemblies in Concerto-D can represent the software present on a single or distributed nodes. This allows a flexible architecture, as a node hosting Concerto-D is able to manage its own local software but also the software of other nodes.

Due to the distributed assemblies, new terms are introduced to describe the location of the different nodes, components, ports or connections. From a node's standpoint, the

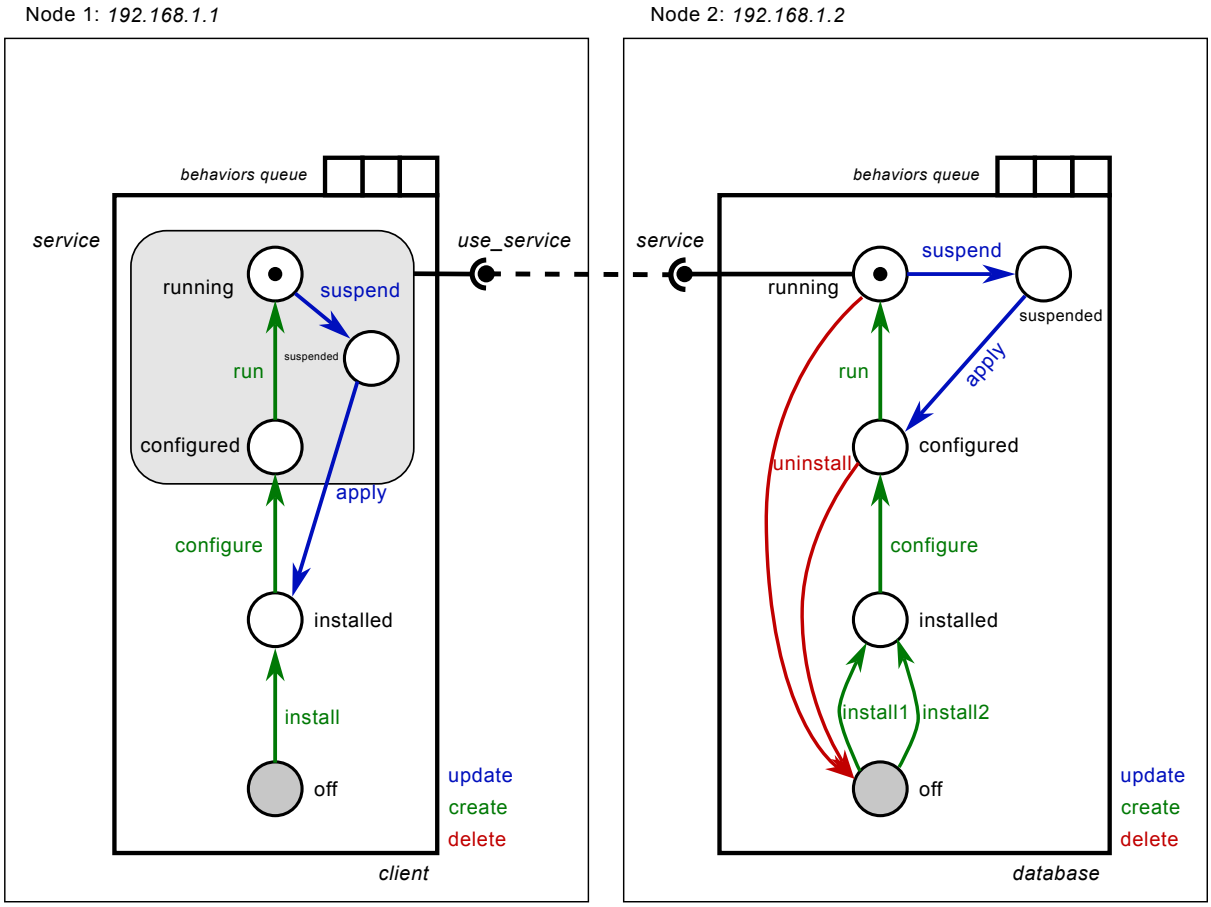


Figure 4.6: Client and database components on different nodes. The local connection is changed to a remote connection. It corresponds to the execution of Listing 4.3 and 4.4. The ip address of each node is depicted in italic.

term local is used to refer to any component or port present locally to the node. The term remote refers to any component or port present on a different node. A local connection is a connection between two local ports. A remote connection is a connection between a local and a remote port.

Figure 4.6 shows two assemblies present on different nodes. Components present in the assembly are similar to components presented in Section 4.2. The only difference is the remote connection between client and database instead of local. Due to this connection, information about port status is split between the two assemblies. From Node 1's standpoint, information about Node 2's port status must be requested and vice versa. Nodes know whether a component is local or remote by checking if the component is part of the local part of the assembly or not. When not part of the local part of the assembly, it means that the component is remote.

Language evolution Due to the decentralised execution of reconfiguration programs, additional synchronisations are required to coordinate the execution of these programs. Notably, due to the remote connection between components, information about remote port status and whether a remote connected component is idle are not known locally. Synchronisation between programs' executions is required for nodes to exchange such information. For these reasons, changes to the semantics of some of Concerto actions have been applied to take into account these synchronisations. The new semantics of

```
### inventory_client.yaml
database: "192.168.1.2:5000"

### reconf_client.py
sc = Assembly("client_assembly")
sc.add("client", "Client")
sc.connect(
 "client", "use_service",
 "database", "service"
)
sc.push_b(f"client", "create")
sc.wait(f"client")

```

Listing 4.3: Reconfiguration program of Node 1 for the creation of the client (Figure 4.6) using direct communication (see Section 4.3.3). An inventory containing the host name and port of the CONCERTO-D instance hosting the database component is assumed

```
### inventory_database.yaml
client: "192.168.1.1:5000"

### reconf_database.py
sc = Assembly("database_assembly"
 )
sc.add("database", "Database")
sc.connect(
 "database", "service",
 "client", "use_service"
)
sc.push_b("database", "create")
sc.wait("database")

```

Listing 4.4: Reconfiguration program of Node 2 for the creation of the database (Figure 4.6) using direct communication (see Section 4.3.3). An inventory containing the host name and port of the CONCERTO-D instance hosting the client component is assumed

CONCERTO-D for the general case have been defined and formalised in [78]. However, in this manuscript, changes to the semantics have been applied to simplify the reconfiguration when the reconfiguration programs don't involve the same behaviour being pushed multiple times. The following describes the new semantics of each action, and whether the semantics of the command has been simplified.

- add(id_c, t): this action doesn't change, as it is the responsibility of each CONCERTO-D instance to manage its local components;
- del(id_c): same as add;
- con(id_u, u, id_p, p): it can now create remote connection. In case of remote connection, this action registers the remote port and remote component to which the local port is connected. This information is used later in the program to retrieve information about the status of the connected remote port. In the CONCERTO-D semantics for the general case, the con ;
- dcon(id_u, u, id_p, p): it can now remove remote connection;
- pushB(id_c, b): the pushB action doesn't change, as it is the responsibility of each CONCERTO-D instance to manage its local components. In the formalised semantics, the pushB assigns an id to the behaviour in case of duplicates;
- wait(id_c): the wait action can now wait for the behaviour queue of a remote component to be empty. In the formalised semantics, the action waits a specific behaviour by its id to be popped from the queue, instead of waiting for the whole behaviour queue (in case of duplicates).

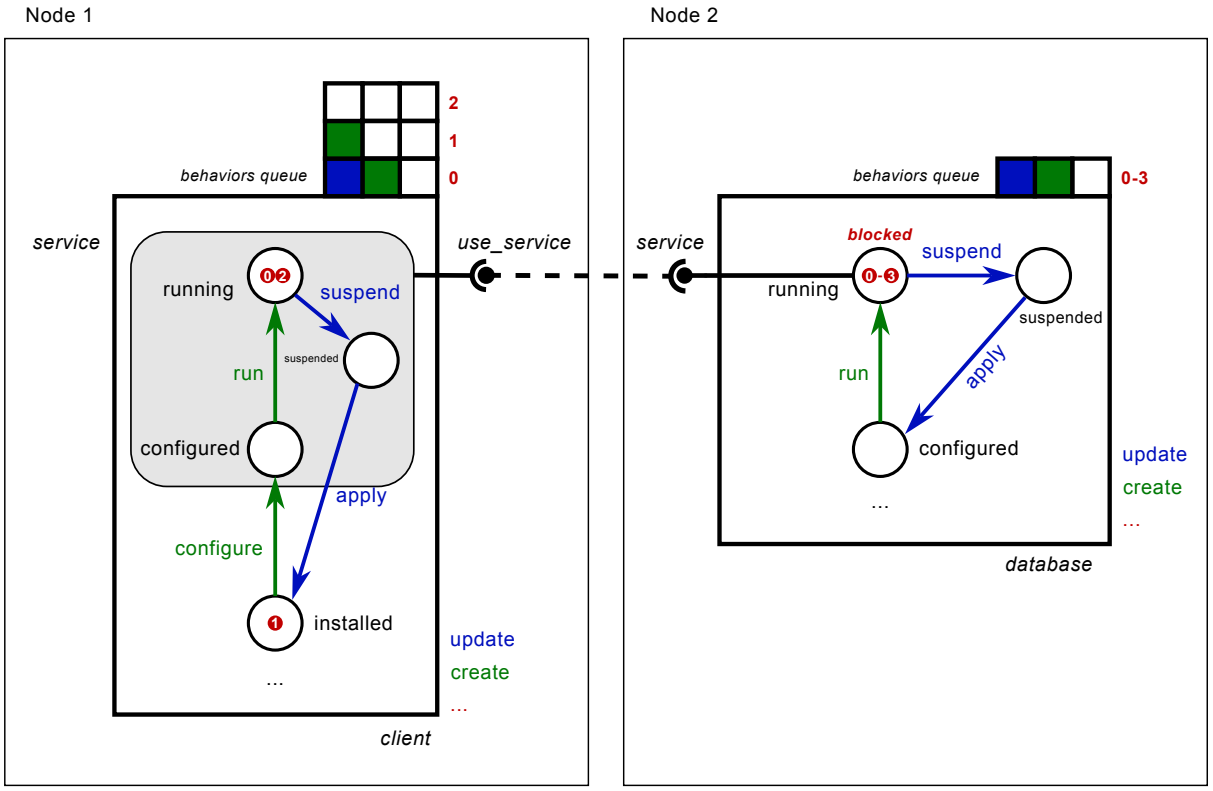


Figure 4.7: Update of client and database components. The client may entirely complete its update before the database component has a chance to leave the `running` place.

From a Concerto program to Concerto-D programs The programs depicted in Listings 4.3 and 4.4 are decentralised versions of the program in Listing 4.2. The add, pushB and wait actions are distributed according to the node. Node 1 manages the client component. Node 2 manages the database component. The connect action is duplicated in both programs as each needs to register the remote port and component.

Concerto-D programs' executions coordination Coordinating the execution of the different Concerto-D programs requires synchronisations between programs' executions. This includes the execution of the wait actions, and the activation or deactivation of ports connected to remote components. When targeting remote components, the wait action needs information from the remote component about whether its behaviour queue is empty. When encountering use or provide ports, nodes need to exchange information about the status of the remote port. Note that, in the formalised Concerto-D semantics, the dcon action also needs synchronisation. As this action is not used in the use cases of this manuscript, it is not discussed.

The need for synchronisation between use and provide ports follows the semantics of use and provide port of Concerto (which is inherited by Concerto-D). In Concerto, a use port can be activated only when its connected provide port is activated. A provide port can be activated only when no connected use port is activated. When ports are remotely connected, synchronisations are needed to check for port status when (1) a use port must be activated and is connected to a remote provide port and (2) a provide port must be deactivated and is connected to one or multiple remote use ports. In case a provide port is connected to multiple use ports, it has to request the status of each remote use port.

```
sc.push_b("client", "update")
### prevent deadlock
sc.wait("database")
###
sc.push_b("client", "create")
sc.wait("client")

```

Listing 4.5: Update of the client. Deadlock is prevented using an additional wait action before pushing the *create* behaviour in the queue (see Figure 4.7).

```
sc.push_b("database", "update")
sc.push_b("database", "create")
sc.wait("database")

```

Listing 4.6: Update of the database (see Figure 4.7)

Deadlock prevention In some scenarios, the rule imposed by use and provide ports can lead to a deadlock. Such scenario can happen notably when the provide port needs to be deactivated and reactivated in the same program. In the following paragraphs, an example of the update of the client and the database is given to illustrate the deadlock problem and its resolution with the wait action.

Listings 4.5 and 4.6 provide an example of the update of the client and database. The *update* and *create* behaviours have been pushed in both components' queues. As seen in Listing 4.5, a wait action is added after the *update* behaviour has been pushed to the client's behaviour queue.

The execution of these programs is depicted in Figure 4.7. Due to the connection between the client and database, the database has to wait for the client to release its provide port before executing its update. However, the client may execute its *update* and *create* behaviours before the database has a chance to leave the `running` place. This would result in the impossibility of the database component to ever execute its update.

This case is depicted on the figure. The client component is able to execute its update in steps 0 to 2. What could happen is that the database is only able to start its update at step 3 (due to factors such as sleeping period, prioritisation of other processes than reconfiguration, etc.). In this case, the token never has a chance to leave the `running` place of the database before the client reactivates its use port. This results in a deadlock.

To prevent this from happening, Concerto initially provides an additional port status called refusing. For a given provide port, the refusing status prevents any subsequent usage of the port until the refusing status is cleared. This means that the `use_service` port of the client couldn't be reactivated until the database completes its update. This is possible thanks to the local availability of information about port status due to the knowledge of the whole assembly in Concerto.

In Concerto-D, the refusing status of the provide port is not available locally by the client. The issue is that the database component may not have set the refusing status yet. Therefore, during execution, the client could request an unset refusing status and consider that it could go on with the execution.

To tackle this problem, an additional wait action is added to the client's reconfiguration program to emulate the refusing status. Using this action, it would be possible to temporarily suspend the execution of behaviours by not pushing them in the queue. In the example, the wait action is added just after pushing the *update* behaviour of the client. The wait action is used to explicitly wait for the database to execute all its behaviours. This would allow both components to finish their execution and terminate the reconfiguration.

```
components:
 client:
  act_places: [installed]
  behaviour_queue: [create]
  connections: [client-use_service/database-service]
nb_executed_actions: 1

```

Listing 4.7: Exceprt of the saved state of the execution when Node 1 suspends the execution of its reconfiguration program while at step 1 (see Figure 4.7). The reconfiguration program is presented in Listing 4.5

This additional wait action is added whenever the execution of a behaviour could result in a deadlock. This means a situation where the execution of a reconfiguration program would lead to the deactivation and reactivation of a use port connected to a remote provide port that also needs to be deactivated and reactivated.

Reconfiguration execution suspension Nodes' sleeping behaviour needs to be taken into account when designing Concerto-D. When going to sleep mode, the node must interrupt all its activities including reconfiguration [79]. Concerto-D allows temporary suspension of reconfiguration program. When a program is suspended, the state of the program execution is saved in a persistent file.

The state of the reconfiguration is presented in Listing 4.7. It is composed of the state of the local part of the assembly and the state of the local reconfiguration program. The state of the local part of the assembly includes the component instances and their active places (act_places), their behaviour queue, and their local or remote connections. The places reached give the status of attached ports. The state of the local reconfiguration program is given by the number of actions already executed by the program. Resuming an execution is done by reading the file and populating values in Concerto-D which makes it continue at the point where it left. Using the number of already executed actions, it skips all the reconfiguration program actions until reaching the actions that were being executed before suspending the execution.

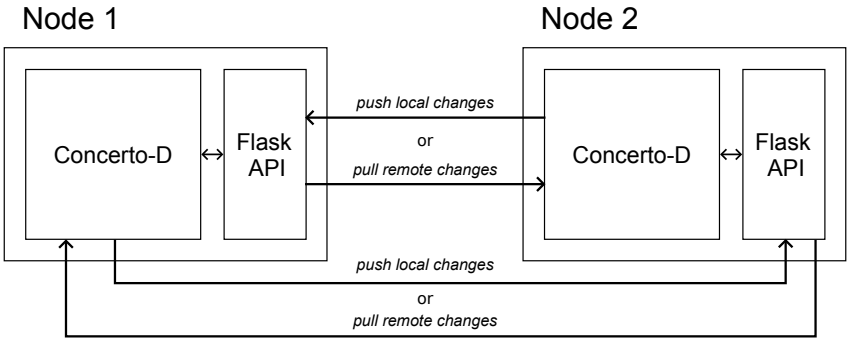
Execution must be gracefully suspended to prevent inconsistencies. A graceful suspension in Concerto-D is possible when not having any ongoing transition, as the interruption of an incomplete transition can lead to an inconsistent state. In this manuscript, it is assumed that Concerto-D can delay an interruption request and finish any ongoing transition before exiting.

Figure 4.3a shows an example of when the node can go to sleep and when it can't. A node must stay awake any time a transition has been fired and did not end yet. Note that, to cope with this restriction, components can be designed using transitions lasting for short time according to their uptime period. Long transitions could be divided into multiple smaller transitions to reduce a potential uptime overhead due to unfinished transitions.

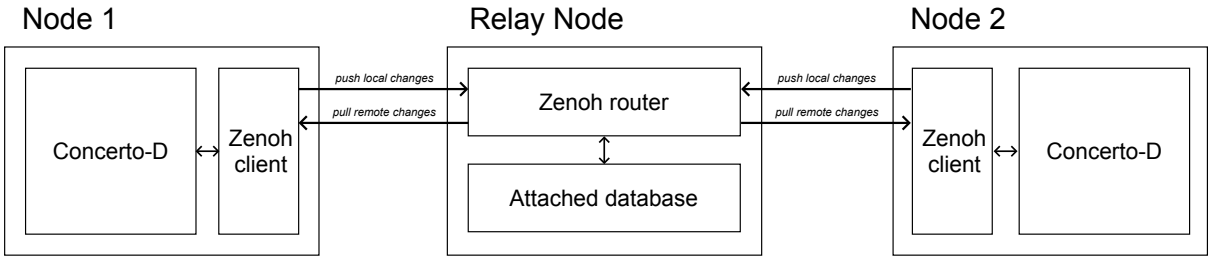
4.3.3 Implementation evolution

The Concerto-D prototype is a fork of Concerto prototype. As for Concerto, the source code of Concerto-D is written in Python (CPython) and publicly available [1].

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



(a) Direct communication representation



(b) Indirect communication representation

Figure 4.8: Direct and indirect communications representation. In the direct communication, each node hosts a Flask API. CONCERTO-D is used as a client to pull or push changes using the Flask API hosted by the other nodes. In the indirect communication, a Relay Node hosts a Zenoh router to handle communication between nodes. The Zenoh router has an attached database to store data pushed by nodes. Each node hosts a Zenoh client to communicate with the router (i.e., push and pull changes).

Implementation of CONCERTO-D serves two purposes: integrate new design elements of CONCERTO-D, and conduct experiments for the use case of this PhD.

Design elements integration Integration of the new semantics to the implementation is relatively small. As stated previously, the semantics of most CONCERTO actions does not change. The main changes are to provide nodes ways to request execution status from other nodes.

Use and provide ports status checking has been updated. When checking the status of connected port, the CONCERTO-D prototype checks whether the connected port is associated with a local component. In case of a remote component, the prototype tries to fetch port status from an external source. The implementation of the wait action also has evolved to take into account remote components. In case of a remote component, the component status is fetched from an external source.

CONCERTO-D has been designed with two communication paradigms: direct and indirect. Direct communication means the ability for a node hosting a CONCERTO-D instance to request data from another node hosting a CONCERTO-D instance. In indirect communications, a middleware is responsible for handling communication between CONCERTO-D instances. Nodes hosting CONCERTO-D instances request and send data to this middleware, which stores and forwards these data to recipients.

Direct communications Direct communication is implemented using a Flask server hosted on each CONCERTO-D instance [2]. It is depicted in Figure 4.8. This server exposes HTTP endpoints allowing nodes to request or send data related to port or component status. The ability to either send or request data enables either push or pull paradigms. In the pull paradigm, CONCERTO-D instances request data to other instances. In the push paradigm, CONCERTO-D instances send data to other instances. In the experiment of this manuscript, the pull paradigm is used (Chapters 5 and 6).

Inventories are available for each CONCERTO-D instances with the location of each component. These inventories have one line per component with the format "*component_id*:
ip_address:*port_num*". The *component_id* corresponds to the id of the component. The *ip_address* is the address of the node hosting the CONCERTO-D instance responsible for the component *component_id*. The *port_num* is the port number through which the Flask server is accessible. This allows, during program execution, to know where each connected remote component is located.

The server is launched at the start of a CONCERTO-D instance in a separate thread from the execution of the reconfiguration program. The following endpoints are exposed.

- get_port_status(*component_id, port_id*): Pull the port status of *port_id* of component *component_id*. Returns 1 if *port_id* is active, else return 0;
- set_port_status(*component_id, port_id, status*): Push the port status. Set the port *port_id* to *status*. *Status* can be 1 or 0;
- get_behaviour_state(*component_id, behaviour_id*): Return "ACTIVE" if the behaviour *behaviour_id* pushed in *component_id* is in the behaviour queue, "INACTIVE" if not.

Indirect communications Indirect communication is implemented using a middleware to handle communication between nodes. This middleware is Zenoh. Zenoh is a lightweight decentralised publish/subscribe communication protocol. It is lightweight as it consumes few resources (at minimum, runs on the data link layer with down to 4 bytes overhead besides user payload). Its decentralisation capabilities allow it to be resilient to network partition. It is also well suited for mobility as nodes entering the network are automatically integrated in the network [3]. Finally, it can be deployed to heterogeneous devices in terms of resources, including single-board computers such as Raspberry Pi or microcontroller (using dedicated solution called Zenoh-pico [4]).

In Zenoh, data can be stored and retrieved using key/value pairs. Zenoh supports multiple formats for value as long as it is serialisable as a bytes buffer. This includes string and integer. The key takes the form of a string with hierarchical syntax. This string aims at naming and categorising the value. Name and categories are separated with "/" as in Unix filesystems (e.g., "/category1/subcategory1/name").

Using this format, CONCERTO-D instances can communicate by sending and retrieving port and behaviour status according to their component id. For simplicity, it is assumed that the component id is unique across the whole system. This can be done by having an external entity distributing ids to each component when instantiating them.

[2] `https://flask.palletsprojects.com/en/stable/`
[3] `https://zenoh.io/`
[4] `https://github.com/eclipse-zenoh/zenoh-pico`

Note that, if component ids were not unique, a node identifier could be used additionally to the component id. The following present the format of each key used to store or retrieve data from Concerto-D instances.

- Port status key: "/port_status/<component_id>/<port_id>". Value stored as integer (0 or 1);
- Behaviour state key:: "/behaviour_state/<component_id>/<behaviour_id>". Value stored as string ("ACTIVE" or "INACTIVE").

The publish/subscribe architecture has the benefits of delegating the responsibility of enabling communication between nodes. This removes the requirement for nodes to build an inventory with the location of components on nodes, as data can be directly retrieved from the middleware.

In this manuscript, a Zenoh instance with an attached database is placed on the Relay Node (see Section 2). When a port or behaviour status change on a Concerto-D instance, this information is pushed and stored on the Relay Node. When a Concerto-D instance encounters a synchronisation point, it tries to pull data from the Relay Node.

While only a single Zenoh instance is used in this manuscript, using Zenoh opens the door to have a decentralised storage with no single point of failure. The decentralisation capabilities of Zenoh allow multiple Zenoh instances to collaborate for data storage and retrieval. When a key/value pair is stored on a Zenoh instance, this data is propagated to all Zenoh instances. When a new Zenoh instance enters the network, it automatically synchronises with other databases to keep its data up to date.

4.4 Concerto-D capabilities validation: OpenStack use case

This section aims at validating the capabilities of Concerto-D by comparing it to an existing solution from the literature. This section describes the experimental setup and evaluation of Concerto-D to validate both its decentralisation and high level of parallelism capabilities. It aims at comparing Concerto-D with Muse, another solution from the literature presented in the state of the art. Muse also has decentralisation capabilities which have been validated by its authors in [42]. As stated in Section 3.1, Muse has *module* parallelism. Comparing the time performance of a reconfiguration using *module* parallelism of Muse and *intra* and *inter-module* parallelism of Concerto-D would allow to validate the high level of parallelism of Concerto-D which allows faster reconfiguration.

Decentralisation capability validation Programs are created both for Concerto-D and Muse. These programs aim at performing the same reconfiguration using either Concerto-D or Muse. These programs have to be executed in a decentralised manner. For Concerto-D, programs are created and executed using Concerto-D actions. For Muse, an equivalent of these programs is created and executed using Muse's declarative syntax. The ability for both Concerto-D and Muse to execute these programs would validate the ability of Concerto-D to coordinate decentralised reconfiguration programs.

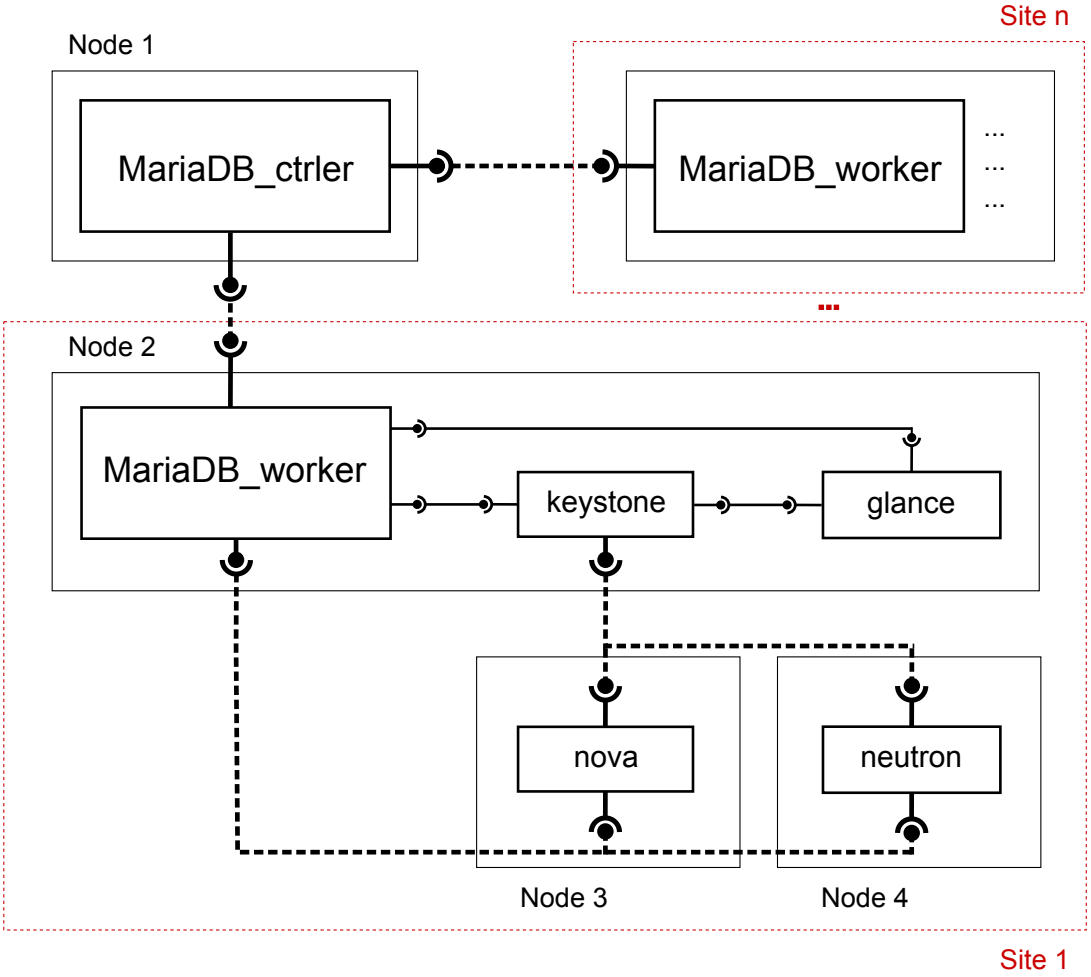


Figure 4.9: Galera cluster overview. Each site is composed of three nodes. One node controls a `MariaDB worker` and `Keystone`, `Glance` services. One node controls the `Nova` service and one node controls the `Neutron` service. Only *Site 1* is represented for simplicity. A `MariaDB controller` collaborates with n `MariaDB workers` distributed over n sites [77].

High level of parallelism validation The reconfiguration programs and components are designed such that the high level of parallelism capabilities of CONCERTO-D can have an impact on reconfiguration duration. To do this, the reconfiguration using CONCERTO-D involves *intra* and *inter-module* parallelism. Muse only has module parallelism. During experiments, the reconfiguration duration of CONCERTO-D and Muse is evaluated. Theoretically, it is expected that the reconfiguration duration using CONCERTO-D is shorter than using Muse. This would validate the ability of CONCERTO-D to have a high level of parallelism, therefore faster reconfiguration compared to a lower level of parallelism.

4.4.1 OpenStack use case

The use case used to compare CONCERTO-D and Muse is a decentralised version of OpenStack [5] on multiple sites. This use case deals with Cloud environments, and not CPS or the Arctic Tundra. The reason is that Muse relies on Pulumi, which has been designed primarily to target Cloud environments. For this reason, Muse has difficulties running

[5] `https://www.openstack.org/`

on constrained hardware (see Section 5). Experiments on this type of environment allow to limit factors that slow down the execution of Muse during reconfiguration. Comparing Muse and CONCERTO-D in Cloud environments and not CPSs ensures accurate comparison of performances of both solutions.

OpenStack is a standard and open-source solution to manage infrastructure in the Cloud (i.e., computing, storage and networking resources). OpenStack can manage large-scale and geo-distributed applications, that may expand on Edge infrastructure (i.e., with potentially high latency). As stated in Section 3.1, environments with high latency may benefit from a decentralised pattern to manage the system, which is the motivation of this use case for this section.

OpenStack is composed of multiple services interacting together. One way to have multiple OpenStack instances collaborating together is to decentralise the MariaDB database that handles data required by the authentication service (i.e., `Keystone`) and to replicate the other services interacting with it. In this section, a Galera cluster is considered. A Galera cluster is composed of one `MariaDB controller` and multiple `MariaDB workers`. Each `MariaDB worker` is connected to different services, which are `Keystone`, `Glance`, `Nova` and `Neutron`. As stated above, `Keystone` is the authentication service, handling permissions and access [6]. `Glance` is responsible to provide metadata on where data assets can be uploaded and used, and handle the virtual machine images to be deployed on the infrastructure [7]. `Nova` aims at providing computation resources (in the form of virtual machines or bare-metal servers) [8]. `Neutron` aims at providing networking resources [9]. This use case was showcased during the 2018 Vancouver OpenStack summit [10] [11].

Figure 4.9 depicts an overview of this use case. In this figure, a `MariaDB controller` collaborates with n `MariaDB workers` distributed over n sites. Only *Site 1* is represented for simplicity. Each site is composed of three nodes. One node controls a `MariaDB worker` and `Keystone`, `Glance` services. One node controls the `Nova` service and one node controls the `Neutron` service. Each node is responsible for the reconfiguration of its own hosted services. The number of nodes accounts for 1+n*3 nodes in total. This means that, for this use case, 1+n*3 concurrent reconfiguration programs are executed [77].

Components in Concerto-D Figure 4.10 gives the representation in CONCERTO-D of one site of the Galera cluster. The `MariaDB worker` (*mariadb_worker*) has a remote dependency with the `MariaDB controller` (*mariadb_ctrler*). The *mariadb_worker* can enter the `restarted` place only when the *mariadb_ctrler* is in the `deployed` place. The *mariadb_worker* provides service to all four other components. The *mariadb_worker* has local dependencies with *keystone* and *glance*, and remote dependencies with *nova* and *neutron*. The *keystone* component provides service to *glance*, *nova* and *neutron*. It has local dependency with *glance*, and remote dependencies with *nova* and *neutron*.

In Muse, local and remote dependencies between components are the same as CONCERTO-D (i.e., following the architecture of Figure 4.9). However, it is not possible to reproduce the fine-grained life-cycle of CONCERTO-D components in Muse. Muse components have been designed from CONCERTO-D components by reducing the number of transition into

[6] `https://docs.openstack.org/keystone/latest/`
[7] `https://docs.openstack.org/glance/latest/`
[8] `https://docs.openstack.org/nova/latest/`
[9] `https://docs.openstack.org/neutron/latest/`
[10] OpenStack summit video
[11] blog video

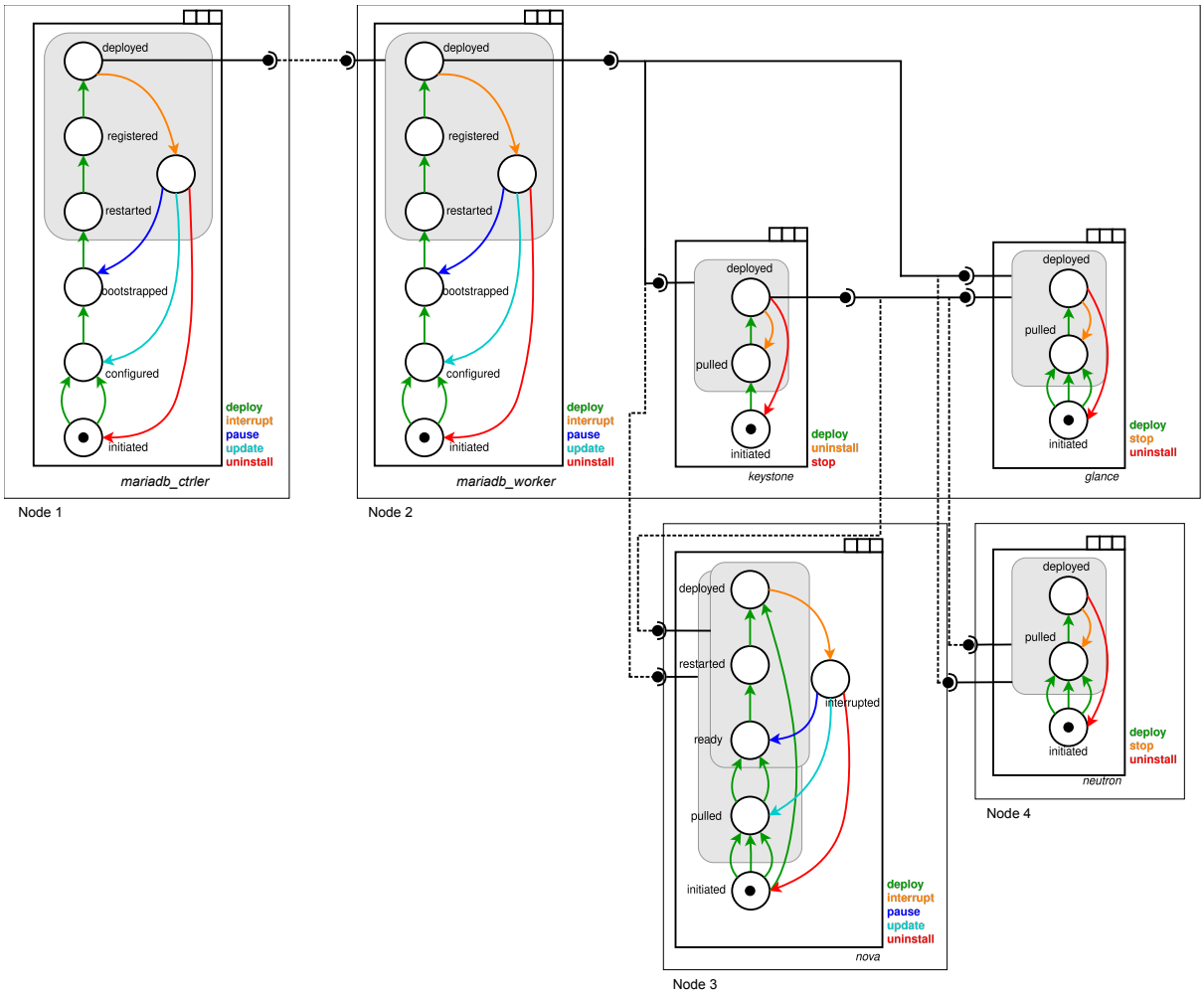


Figure 4.10: CONCERTO-D components of the `MariaDB controller` and one site of the Galera cluster. Local and remote dependencies are depicted for each component [77].

```
sc.add(
 "mariadb_ctrler",
 "MariaDBCtrler"
)
sc.connect(
 "mariadb_ctrler", "service",
 "mariadb_worker", "use_service",
)
sc.push_b(
 f"mariadb_ctrler",
 "deploy"
)
sc.wait(f"mariadb_ctrler")
```

Listing 4.8: Exceprt of the deployment of the MariaDB controller.

```
sc.add(
 "mariadb_worker",
 "MariaDBWorker"
)
sc.connect(
 "mariadb_worker", "use_service",
 "mariadb_ctrler", "service",
)
sc.push_b(
 f"mariadb_worker",
 "deploy"
)
sc.wait(f"mariadb_worker")
```

Listing 4.9: Exceprt of the deployment of the MariaDB worker.

individual ones according to the module operation. For instance, for the *create* operation of Muse, the sum of the duration of all transitions associated to the *deploy* behaviour for a specific component is taken. The same principle is applied to all components.

```
const mariadb_ctrler = new
Resource(
 'mariadb_ctrler',
 {
  timeCreate: t1,
  timeDelete: t2,
  offers: []
 }
)

```

Listing 4.10: Excerpt of the declaration of the MariaDB controller in Muse.

```
const mariadb_worker = new
Resource(
 'mariadb_worker',
 {
  timeCreate: t3,
  timeDelete: t4,
  offers: [mariadb_ctrler.offer]
 }
)

```

Listing 4.11: Excerpt of the declaration of the MariaDB worker in Muse.

Scenario	# Sites	Concerto-D	Muse	Gain
Deploy	1	306.02s	536.57s	42.7%
	2	306.09s	536.69s	42.6%
	5	306.19s	537.09s	42.7%
	10	306.14s	538.13s	42.7%
Update	1	416.84s	555.56s	24.4%
	2	416.92s	555.70s	24.2%
	5	417.17s	556.08s	24.0%
	10	417.46s	556.77s	24.0%

Table 4.1: Reconfiguration duration for Concerto-D and Muse for the deployment and update of nodes of the Galera cluster for different number of sites. The gain allowed by Concerto-D compared to Muse is depicted on the last column.

Reconfiguration program Listings 4.8 and 4.9 are excerpts of Concerto-D programs used to deploy *mariadb_ctrler* and *mariadb_worker*. In these programs, the components are connected and the *deploy* behaviour is pushed for each component. The same actions are used to add, connect and *deploy* the *keystone*, *glance*, *nova* and *neutron* components. Listings 4.10 and 4.11 are excerpts of the equivalent reconfiguration programs in Muse. Muse has a declarative syntax to declare components and dependencies. Each component is declared and parametrised by the duration of *create* and *delete* operations.

For the update reconfiguration program, it is assumed that nodes are initially in the `deployed` places in Concerto-D, and that the components have been *created* in Muse. For Concerto-D, the update consists in pushing and waiting for the *interrupt*, *update* and *deploy* behaviours for *mariadb_master*, *mariadb_worker* and *nova* components. For *keystone*, *glance* and *neutron*, it consists in pushing the *uninstall* and *deploy* behaviours. For Muse, the update consists in deleting and creating again the components. In Muse, for the *mariadb_master*, *mariadb_worker* and *nova* components, the time required to *delete* each component is equal to the sum of the time required to execute transitions associated with the *interrupt* and *update* behaviours. For the *keystone*, *glance* and *neutron* components, the time required to *delete* each component is equal to the sum of the time required to execute transitions associated with the *uninstall* behaviour.

```
sc.push_b(
 f"mariadb_ctrler",
 "interrupt"
)
sc.push_b(
 f"mariadb_ctrler",
 "update"
)
sc.push_b(
 f"mariadb_ctrler",
 "deploy"
)

```

Listing 4.12: Exceprt of the update of the MariaDB controller.

```
sc.push_b(
 f"mariadb_worker",
 "interrupt"
)
sc.push_b(
 f"mariadb_worker",
 "update"
)
sc.wait(f"mariadb_ctrler", "
 deploy")
sc.push_b(
 f"mariadb_worker",
 "deploy"
)

```

Listing 4.13: Exceprt of the update of the MariaDB worker.

4.4.2 Evaluation

This subsection aims at giving results on the duration of the execution of reconfiguration programs for the deployment and update of the Galera cluster, for Concerto-D and Muse. Results are presented in Table 4.1. Based on these results, the validation of decentralisation capabilities and high level of parallelism of Concerto-D are given.

Experimental setup For the sake of simplicity and reproducibility, traces of third-party real experiments on OpenStack and Galera have been used to calibrate the experiments [38]. These traces are available online [12]. Experiments are performed on a real infrastructure offered by the experimental platform Grid'5000. Evaluations are conducted on the Gros and Paravance clusters [13]: Gros is composed of hosts equipped with one 18-core Intel Xeon Gold 5220 CPUs. Paravance has hosts equipped with two 8-core Intel Xeon E5-2630. Network in these clusters has a high bandwidth (at Gbps) and ensures stable connectivity between nodes. EnosLib [80] scripts have been used as a front interface for Concerto-D instances [77].

Decentralisation capability validation Concerto-D and Muse successfully terminate the execution of their reconfiguration programs. This validates the ability of Concerto-D to execute decentralised reconfiguration programs for the deployment and update of distributed services.

High level of parallelism validation Reconfiguration duration results for Concerto-D are consistently lower than Muse for deployment and update. This is mainly due to the opportunities for Concerto-D to execute transitions in parallel. This is especially the case for deployment. As seen in Figure 4.10, lots of transitions associated with the *create* behaviour can be executed using *intra-module* parallelism. This is especially the case for transitions originating from the initial places. This allows Concerto-D to significantly reduce the reconfiguration duration compared to Muse. As a reminder, the duration of

[12] https://doi.org/10.5281/zenodo.10472116

[13] 4https://www.grid5000.fr/w/Hardware

the *create* behaviour using Muse is equal to the sum of the duration of all transitions associated with *create* behaviour in Concerto-D.

Update is longer than deployment for Concerto-D and Muse. This is expected for Muse, as the theoretical update duration is the sum of *create* and *delete* duration. This duration should be higher than the duration for only the execution of *create*. For Concerto-D, it is slightly different. While it has to execute transitions associated with interruption or uninstallation of components, its fine-grained life-cycle allows for some components to skip the transitions starting from the initial places (for *mariadb_master*, *mariadb_worker* and *nova*). Transitions duration for the interruption or uninstallation of components are relatively fast compared to the whole reconfiguration duration (e.g., up to 10s). However, update duration for Concerto-D is about 110s higher than deployment. One possible explanation is the additional wait action that is added before the action that pushes the *deploy* behaviour (see Listings 4.12 and 4.13). This wait prevents the *deploy* behaviour from being pushed, meaning that it delays the execution of transitions for the *deploy* behaviour, including the ones that are not blocked by an unresolved dependency. Future works could be proposed to remove this wait action and speed up the reconfiguration.

4.5 Limitations

This subsection discusses the limitations of the current Concerto-D design and prototype.

Inventory and location of remote components When using direct communication, nodes are assumed to have an inventory with the location of each remote component on the other Concerto-D instances. The computation and maintenance of such inventory is not considered in this manuscript. Such an inventory could be given directly by the *Planning* phase of the MAPE-K along with the reconfiguration program. It could also be dynamically updated using discovery algorithms [81].

An alternative would be to delegate the responsibility of communication to a dedicated middleware. In this manuscript, this middleware is Zenoh. To communicate, nodes only need to be part of the Zenoh network, which handles all discovery and communication concerns. This removes the need for nodes to maintain an inventory. Additionally, this allows to remove any concerns about the actual location of remote components. Their locations are transparent to each Concerto-D instance.

Unavailability of remote Concerto-D instance during coordination During coordination, the unavailability of one of the Concerto-D instance prevents the coordination to be completed. Currently, Concerto-D doesn't have a way to detect and report such failure. For instance, implementing a heartbeat with the nodes involved in the coordination and considering a certain threshold of missed heartbeats as a crash of the node. Such a heartbeat could be calibrated depending on the latency of the environment where the system is running (e.g., milliseconds in Cloud, hours in the DAO).

Checkpoint system Crash (e.g., power failure) is not considered in the design of Concerto-D. Dealing with crash during process execution has been studied in the intermittent computing literature. A notable feature is the checkpoint system. A checkpoint system allows to frequently save the state of a system while it is running. When

the system crashes in a way that could lead to an inconsistent state, the saved state can be used to restore the system [82].

A checkpoint system could easily be implemented at the scope of Concerto-D program execution. Concerto-D currently supports the saving of the execution state when the program is suspended before sleeping. Such feature could be extended by saving the state more frequently (e.g., every time a transition finishes, a place is reached, etc.). When the execution of Concerto-D crashes, the execution could continue from the last saved state. However, when dealing with reconfiguration, the benefits of having a checkpoint system only for Concerto-D is limited. As a reminder, Concerto-D is the managing system. The transitions executed by Concerto-D components apply changes to the managed system. If Concerto-D crashes during the execution of a transition, such a transition only partially executes changes on the managed system. Such changes cannot be tracked by Concerto-D. Concerto-D only considers that the transition hasn't been completed before the crash and that it needs to execute it again. For this reason, additional work should be done to be able to fully rollback the managing and managed systems.

4.6 Conclusion

In this chapter, Concerto-D's design and implementation are presented. Concerto-D is based on Concerto, a reconfiguration solution providing high expressiveness in terms of parallelism. Concerto-D extends the semantics of Concerto to allow model and execution of decentralised reconfiguration across distributed Concerto-D instances. Its decentralisation and high level of parallelism capabilities have been validated for the deployment and update of distributed services. These capabilities have been validated by comparing the execution of Concerto-D with the execution of Muse, an existing decentralised solution from the literature. Experiments have been conducted in an environment with sufficient resources (e.g., computing power, stable connectivity) where both Concerto-D and Muse can be executed without being slowed down by the environment.

Additionally, Concerto-D has been designed to be able to suspend and resume its execution, and is equipped with direct (i.e., client/server) and indirect (i.e., publish/subscribe) communication capabilities. Such capabilities make Concerto-D suitable for experiments involving sleeping nodes.

In the next chapter, Concerto-D is used as a decentralised reconfiguration solution to gather results of the reconfiguration in the context of sleeping nodes and disruptive network. It is deployed on a set of sleeping nodes hosting different services and coordinating their reconfiguration in a decentralised manner.

Chapter 5

Impact of sleeping nodes on reconfiguration duration

Contents

Sleeping nodes are only able to execute tasks during their uptimes. This includes execution and coordination of reconfiguration actions with other nodes. When dealing with short and non-synchronised uptimes, coordinating a reconfiguration can take a long time. This chapter aims at studying the time performance of the coordination of a decentralised reconfiguration in the context of sleeping nodes. To this end, theoretical study and experiments using the reconfiguration solutions are provided. These experiments are conducted in the context of the DAO-CPS (see Section 2). This chapter deals with

Observation Node (ON) and Relay Node (RN). CONCERTO-D and Muse are both used for experiments to provide results on the decentralised reconfiguration duration in such a context. Both solutions are able to suspend and resume their execution, making it possible to host them on sleeping nodes.

5.1 Introduction

To the best of my knowledge, no contribution has yet studied decentralised reconfiguration involving disruptive networks and sleeping nodes. Contributions dealing with reconfiguration of distributed systems usually consider environments with sufficient resources (e.g., energy, network) to ensure stable connectivity between nodes. This chapter aims at studying reconfiguration in the context of disruptive networks and sleeping nodes. To this end, it considers the DAO use case presented in Section 2.3. As a reminder, this use case considers a deployment of ONs in an environment with scarce energy and network resources. ONs are forced to sleep most of the time and wake up for short duration to increase their lifetime. ONs' uptimes are not synchronised. Additionally, the absence of network infrastructure forces ONs to rely on peer-to-peer network to communicate. In this chapter and Chapter 6, the term ON is used to refer to a node with the properties of a node deployed in the Arctic Tundra and part of the DAO-CPS. It doesn't refer to real ONs (i.e., physically deployed in the Arctic Tundra).

The first objective of this chapter is to provide results on decentralised reconfiguration duration considering (i) sleeping nodes and (ii) scarce connectivity due to non-synchronised uptime periods and peer-to-peer network. As ONs only execute reconfiguration during their uptimes, having long sleeping periods can significantly delay the overall reconfiguration duration. Additionally, ONs are forced to rely on peer-to-peer connections, as they cannot rely on external network infrastructure (see Section 2.3). Due to remote dependencies between modules defined in reconfiguration programs, ONs need to communicate in peer-to-peer manner to resolve such dependencies during reconfiguration. For such communication to happen, both nodes need to be awake at the same time (i.e., having their uptimes overlap). To study the reconfiguration duration due to these aspects, two parameters representing the characteristics of ONs deployed in the Arctic Tundra are studied: the number of overlaps between ONs, and the order of uptimes of each ON (i.e., uptime schedule).

The second objective of this chapter is to emphasise the existence of a trade-off between uptime dedicated to reconfiguration and total reconfiguration duration. This trade-off is studied in a context where the Relay Node is available to handle communications. As stated in Chapter 2, reconfiguration is a low priority task compared to observation activities. ONs spend their uptime primarily for observation purposes. Additional uptime implied by reconfiguration activities could have a significant impact on ONs' energy budget. Due to rare connectivity between ONs, a large portion of uptime could be spent waiting for an opportunity to communicate. This waiting period forces the ON to be awake, consuming energy. This chapter explores a potential trade-off to reduce the amount of time spent waiting by increasing reconfiguration duration. This is done by allowing ONs to sleep instead of staying awake during waiting periods. This allows shorter uptimes to the cost of potentially missed communication opportunities.

Direct and indirect communication are used to conduct experiments. Using direct communication, ONs can communicate only when they are awake at the same time (i.e., uptime overlap). Using indirect communications, a Relay Node hosting a Zenoh router

is responsible for handling communications between ONs (see Section 4.3.3). This removes the necessity of ONs to overlap to communicate as they can rely on the RN to communicate.

This Chapter aims at answering the following research questions:

- RQ1: How the number of overlaps and uptime order influence reconfiguration duration, with and without using RN for communication?
- RQ2: To what extent using an RN for communication can reduce reconfiguration duration?
- RQ3: To what extent the RN can be leveraged to decrease uptime duration of ONs while not significantly increasing reconfiguration duration (i.e., trade-off between uptime duration and reconfiguration duration)?

These questions are studied theoretically and experimentally. Subsection 5.2 presents two reconfiguration scenarios used to conduct experiments. Subsection 5.3 provides a formalisation/modelling of the use case and research questions. Subsection 5.4 is dedicated to experiments using reconfiguration prototypes. Experiments are conducted using Concerto-D and Muse, which are decentralised reconfiguration solutions presented in Chapter 4. Finally, discussions and conclusions are given.

5.2 Reconfiguration scenarios for the DAO-CPS

This section presents reconfiguration scenarios used in experiments. These scenarios are examples of coordinated reconfiguration between ONs. The coordination is described in Subsection 5.2.1. It shows the individual actions to execute on the different ONs and their coordination. The implementation in Concerto-D of such coordination is presented in Subsection 5.2.2.

5.2.1 Deployment and update of distributed services

Scenarios in this chapter deal with the reconfiguration of coupled modules hosted on a set of ONs. They are inspired by existing CPS deployments previously done in the Arctic Tundra [6, 7]. Modules hosted on ONs represent the collection and analytics of data extracted from the field. Data are collected by a set of ONs, which are analysed by another ON.

n observers are considered, sending observations to an *aggregator*. In this chapter, these modules are hosted on ONs. The RN is only used for communication purposes. The *aggregator* needs outputs from all *observers* to be functional. When changes happen either in the *aggregator* or *observers* (e.g., type of observation changed, service has been interrupted), ONs need to share data and coordinate their changes. Two coordination cases, *deploy* and *update*, are considered and depicted in Figure 5.1.

Deploy: It takes the form of two synchronisation barriers. First, the *aggregator* fetches configurations from all *observers*, to do its calibration. Second, it waits for notification that all *observers* are actively sensing and performing observations. Then, it starts to listen and process observations, shared by *observers*.

Update: It takes the form of one synchronisation barrier. The *aggregator* stops listening for observation and waits for each *observer* to update before applying its own

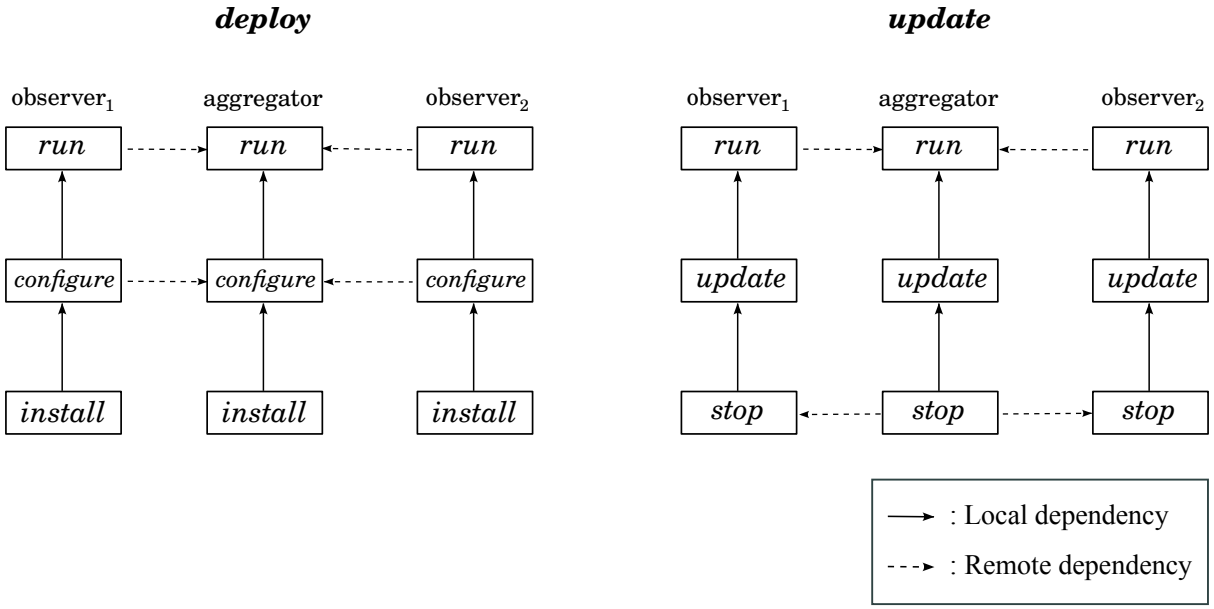


Figure 5.1: *Deploy* and *update* coordinations. Only two *observers* are depicted for simplicity. The *observers* and *aggregator* are not necessarily neighbours. Actions and their local or remote dependencies are depicted. An action can be completed only when all of its dependencies are resolved.

update. Each *observer* updates upon receiving confirmation that the *aggregator* stopped listening for observation.

In experiments, real actions are not executed on ONs. Instead, a random duration is generated for each module's transition. This allows to cover a large spectrum of cases by having more control over the range of duration that each transition can have. Durations are generated over a range based ONs' uptime duration values from [8]. Each transition is assigned to a random duration, following a log-normal distribution bounded between 1s and 30s, where low values are more represented.

5.2.2 Implementation in Concerto-D and Muse

As for Chapter 4.4, experiments are conducted using Concerto-D and Muse. The following describes the implementation for both solutions.

Reconfiguration programs The Concerto-D assemblies corresponding to the reconfiguration use cases are depicted on Figure 5.2. Only two observers are depicted for simplicity. Each observer has 2 connections with the aggregator.

Deploy reconfiguration programs are depicted in Listings 5.1 and 5.2. The aggregator needs to synchronise with all observers before entering `configure_n` places. This is due to the *cfg_obs_i* use ports connected to the *observer* components. The aggregator needs additional synchronisations with all observers before entering the `running` place, due to the *srv_obs_i* use ports.

Update reconfiguration programs are depicted in Listings 5.3 and 5.4. Reconfiguration programs are executed simultaneously on aggregator and observers. Before leaving their own `running` places, observers need the aggregator to execute the `suspend_n` transitions and leave the `suspended` place. Then, before pushing its *create* behaviour, the aggregator needs all observers to finish their updates (by waiting for each observer as noted in line

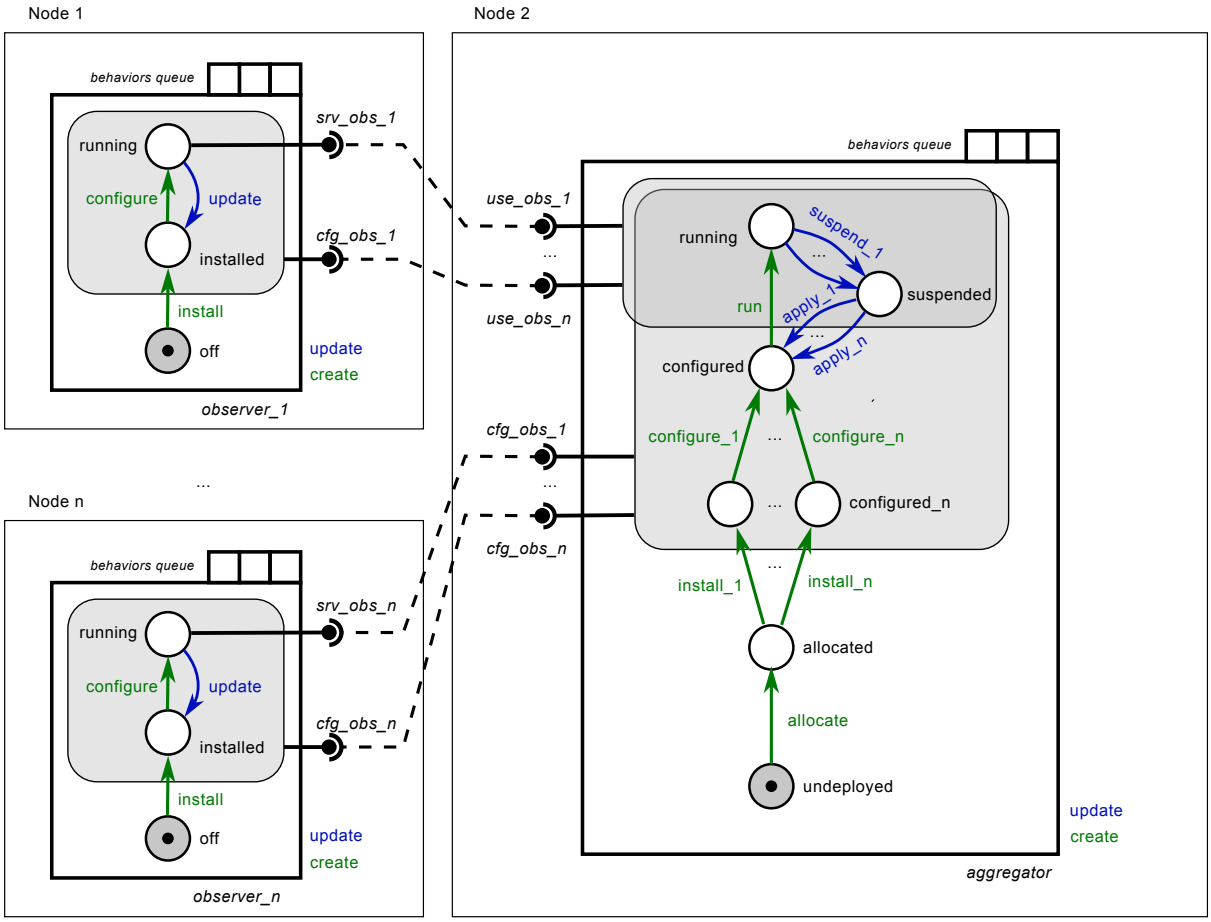


Figure 5.2: Representation of an aggregator and observers. The aggregator collaborates with n observers. Two observers are depicted for simplicity.

4 of Listing 5.3).

Deploy and update programs are distributed on each ON prior to the experiments. Reconfiguration programs are executed simultaneously on aggregator and observers. Uptime and sleeping periods are simulated by running and exiting the execution of the program. When an ON "goes to sleep", the execution of the reconfiguration program is gracefully suspended and the reconfiguration state is saved on disk. When a ON "wakes up", it reloads the state and continues where it left.

Experiments using Muse Experiments have also been realised using Muse. As for experiments in Chapter 4.4, the main difference between reconfiguration programs of Muse and CONCERTO-D is the fact that CONCERTO-D provides *intra* and *inter-module* parallelism contrary to Muse (see Section 3.1). This is notably the case of the `install_n`, `configure_n`, `suspend_n` and `apply_n` transitions of the aggregator component.

As for Section 4.4, in Muse, transitions have been merged into single ones. In Muse, the *create* module operation duration corresponds to the sum of the duration of CONCERTO-D transitions associated with the *create* behaviour. The *delete* module operation duration corresponds to the sum of the duration of transitions associated with the *update* behaviour of CONCERTO-D. The remote connections between components are the same in Muse.

```
sc = Assembly("agg_assembly")
sc.add_component("agg", "Aggregator")
for i in range(nb_obs):
  sc.connect("agg", f"cfg_obs_{i}", f"cfg_obs_{i}", f"obs_{i}")
  sc.connect("agg", f"use_obs_{i}", f"srv_obs_{i}", f"obs_{i}")
sc.push_b("agg", "create")
sc.wait("agg")

```

Listing 5.1: Deployment of the aggregator.

```
sc = Assembly("obs_n_assembly")
sc.add_component(f"obs_{n}", "Observer")
sc.connect(f"obs_{n}", f"cfg_obs_{n}", "agg", f"cfg_obs_{n}")
sc.connect(f"obs_{n}", f"srv_obs_{n}", "agg", f"use_obs_{n}")
sc.push_b(f"obs_{n}", "create")
sc.wait(f"obs_{n}")

```

Listing 5.2: Deployment of observer n

5.3 Theoretical study

Before doing experiments with Concerto-D and Muse, the coordination of the reconfiguration is studied theoretically. A generic model of both deployment and update coordinations between ONs is given. This model leads to an analytical answer to RQ1, RQ2, and RQ3, experimentally validated in the rest of the section.

5.3.1 Deploy and update modelling

Many languages exist in the literature to model the configuration of a system and apply changes (see Section 3.1). Reconfiguration languages can be used to write either deployment or update procedures. In Concerto, a reconfiguration is modelled as a graph of actions to apply on a system. The following modelling follows this idea for Concerto-D.

A deployment or an update procedure on multiple ONs can be modelled as a partially ordered set $(A_r, \prec, \preccurlyeq)$. $\prec$ is the order relation used between two actions on the same ON. $\preccurlyeq$ is the order relation used between actions executed on two different ONs. $\preccurlyeq$ models a required communication between two ONs of the CPS to solve a remote dependency.

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

Example: Figure 5.1 and Listing 5.5 give a simplified example of the reconfiguration use case represented in Figure 5.2. Figure 5.1 depicts the simplified DAG modelling, while Listing 5.5 represents a subpart of the partially ordered set modelling (with two ONs). Actions are denoted $install_i$, $config_i$, and run_i, where i represents the ON number on which the action is executed.

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

```
sc = Assembly("agg_assembly")
sc.push_b("agg", "update")
for i in range(nb_obs):
 sc.wait(f"obs_{i}")
 sc.push_b("agg", "deploy")
 sc.wait("agg")

```

Listing 5.3: Update of the aggregator.

```
sc = Assembly("obs_n_assembly")
sc.push_b(f"obs_{n}", "update")
sc.push_b(f"obs_{n}", "create")
sc.wait(f"obs_{n}")

```

Listing 5.4: Update of observer n.

$LP(A_r, D_r \cup D_r^*)$ [38].

The duration of each edge in D_r^* is $W(a_i \lessdot a_j)$. $W(a_i \lessdot a_j)$ is the waiting duration before the ON responsible for a_j is informed that the dependency $a_i \lessdot a_j$ is solved. W is the element of interest in this manuscript. Notably, using an RN or not impacts W, while other weights of the graph are unchanged.

5.3.2 ONs and overlap modelling

Considering the infinite time as $\mathbb{R}^+$, and the sets of sleeping and uptime periods of an ON $\mathfrak{S}$, $\mathfrak{U}$ 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 ONs (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 ON i, and $(s_j, e_j) \in \mathfrak{U}_\mathfrak{j}$ on ON 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 $\min(e_i, e_j) - \max(s_i, s_j)$.

Communications between two ONs happen if and only if there exists at least one overlapping period $\mathfrak{o} \in \mathfrak{O}$ between the two ONs i, j, denoted $\mathfrak{o}(i, j)$.

Figure 5.3 illustrates an example of uptime and sleeping periods as well as overlapping periods between two ONs.

5.3.3 Waiting duration modelling

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

The instant where action a_i is finished on ON i is denoted t_{a_i}. The instant where ON j needs a_i to be finished on ON i before being able to apply action a_j is denoted $t_{a_i \lessdot a_j}$. The waiting duration induced by the dependency $a_i \lessdot a_j$ without the RN is

$$W(a_i \lessdot a_j) = \begin{cases} s_{\mathfrak{o}_{i,j}(t_{a_i})} - t_{a_i \lessdot a_j} & t_{a_i} > t_{a_i \lessdot a_j} \\ s_{\mathfrak{o}_{i,j}(t_{a_i \lessdot a_j})} - t_{a_i \lessdot a_j} & \text{otherwise} \end{cases} \tag{5.1}$$

$$\begin{aligned}
&\text{observer1} \quad (1): \; install_1 \prec configure_1, configure_1 \prec run_1 \\
&\text{aggregator} \quad (2): \; install_2 \prec configure_2, configure_2 \prec run_2 \\
&\text{observer2} \quad (3): \; install_3 \prec configure_3, configure_3 \prec run_3^3 \\
&\\
&\text{Between ONs:} \quad configure_1 \lessdot configure_2, run_1 \lessdot run_2 \\
&\qquad\qquad\qquad\quad configure_3 \lessdot configure_2, run_3 \lessdot run_2
\end{aligned}$$

Listing 5.5: Deployment modelled as a strict partially ordered set with two order relations $\prec$ and $\lessdot$

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 ONs 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 ON j has to wait from the instant it requires a_i to be finished on ON 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 ON i is not blocked and can continue its local execution while j is waiting.

When RN is present, ONs do not have to overlap to communicate. In this case, the waiting duration induced by the dependency $a_i \lessdot a_j$ is

$$W(a_i \lessdot a_j) = \begin{cases} s_{\mathfrak{u}_j(t_{a_i})} - t_{a_i \lessdot a_j} & t_{a_i} > t_{a_i \lessdot a_j} \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

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

Information that a_i is finished on ON i is stored in the RN (considered available). If ON j needs a_i to be finished after ON i ends a_i (i.e., $t_{a_i} > t_{a_i \lessdot a_j}$), the waiting duration is null for j. If ON j needs a_i to be finished before ON i ends a_i, ON 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 (5.1) and (5.2) the following statements stand:

- RQ1: without RN, the waiting duration depends on the frequency of overlaps between ONs ($s_{\mathfrak{o}_{i,j}}$ (Eq. 5.1). With RN the waiting duration depends on the uptime order between ONs (i.e., $s_{\mathfrak{u}_j(t_{a_i})}$ (Eq. 5.2);
- RQ2: relations $s_{\mathfrak{o}_{i,j}(t_{a_i})} \geqslant s_{\mathfrak{u}_j(t_{a_i})}$ and $s_{\mathfrak{o}_{i,j}(t_{a_i \lessdot a_j})} \geqslant s_{\mathfrak{u}_j(t_{a_i})}$ are always true. In the best case, the next uptime instant of j is also an uptime period for i. Thus, the waiting duration without RN should always be greater than or equal to the waiting duration with RN. Consequently, the deployment and update durations should always be faster when using an RN for communication.

5.3.4 Increasing the sleeping time of ONs

Increasing sleeping time of ONs can be done by reducing uptime duration of ONs. The current uptime period of a ON j is denoted $\mathfrak{u}_j^c \in \mathfrak{U}$. Reducing $\mathfrak{u}_j^c$ reduces energy spent by the ON being up.

An ON can go back to sleep if and only if it has no ongoing measurements, backup, or computations performed on the ON, and if the ON is currently waiting for information

from other ONs. In other words, reducing the current uptime period is possible if and only if the ON is *inactive*.

Reducing $\mathfrak{u}_j^c$ also reduces the probability of overlap with other ONs. A shorter $\mathfrak{u}_j^c$ could miss some reconfiguration information that could be received during the initial $\mathfrak{u}_j^c$. For this reason, there is a trade-off between reducing uptime periods to save energy on ONs and increasing the global reconfiguration duration.

More formally, for a dependency $a_i \lessdot a_j$, and with t_{a_i} (as previously defined) the point in time when a_i is finished on ON i, the information that a_i is finished can be missed when reducing $\mathfrak{u}_j^c$ if $t_{a_i} \in \mathfrak{u}_j^c$.

$R(\mathfrak{u}_j^c)$ is the reduction rate of $\mathfrak{u}_j^c$. When $R(\mathfrak{u}_j^c) = 100\%$, the ON goes back to sleep as soon as it is inactive. This reduces as much as possible $\mathfrak{u}_j^c$. When $R(\mathfrak{u}_j^c) = 50\%$ the ON goes back to sleep when inactive and when 50% of $\mathfrak{u}_j^c$ has elapsed.

When reducing $\mathfrak{u}_j^c$ with the rate $R(\mathfrak{u}_j^c)$, ON j may miss and delay the information that a_i is finished (i.e., potentially increasing the reconfiguration duration) when

$$t_{a_i} \geqslant e_{\mathfrak{u}_j^c} - R(\mathfrak{u}_j^c) d_{\mathfrak{u}_j^c} \tag{5.3}$$

where $e_{\mathfrak{u}_j^c}$ is the initial ending point in time of $\mathfrak{u}_j^c$ and $d_{\mathfrak{u}_j^c}$ its initial duration.

Note that if t_j^{in} denotes the instant in $\mathfrak{u}_j^c$ when ON j is inactive, if $t_j^{in} \geqslant e_{\mathfrak{u}_j^c} - R(\mathfrak{u}_j^c) d_{\mathfrak{u}_j^c}$, the ON immediately goes back to sleep when inactive.

From Equation (5.3) the following points stand:

- when having a low overlap rate between ONs, the probability that this equation is satisfied is low as it requires ON i to be awoken during $\mathfrak{u}_j^c$ on ON j;
- when having a low overlap and using synchronous communications between ONs, the consequences when satisfying this equation may be severe on the reconfiguration duration. In fact, the next overlap period has to be reached to solve the dependency;
- when having a low overlap and using asynchronous communications, the consequences when satisfying this equation have a smaller impact on the reconfiguration duration. In fact, it is only required to wait for the next uptime period of ON j.

For this reason, uptime reduction is only considered when using an RN.

When using an RN, especially if the overlap rate is low, the sleeping time of ONs can theoretically be increased while not significantly increasing the reconfiguration duration (RQ3).

5.4 Impact of number of overlaps and uptime orders

This section describes experiments and results that allow to experimentally validate the theoretical results for RQ1 and RQ2. The following describes the reconfiguration program and uptime scenarios used to conduct experiments. It presents the metric used to measure reconfiguration duration and infrastructure used to conduct experiments. Results are given and discussed.

5.4.1 Uptime scenarios

Studying the impact of non-synchronised uptimes on reconfiguration duration using reconfiguration prototypes is challenging. ONs wake up randomly each hour for few minutes

at best [8]. Experiments in such a context could last for days or weeks, which is not affordable for the kind of experiments conducted here (study of the influence of different parameters on results). Chapter 6 aims at using simulation to deal with this issue. In this section, experiments have been designed to be terminated within 3h at maximum. This allows to obtain valuable results while staying within reasonable boundaries.

During experiments, ONs follow predefined scenarios made of uptime and sleeping periods. Sleeping periods are much shorter than the ones in the DAO context (around 200s sleeping duration instead of an hour). Uptime duration is set to 50 seconds [8]. During each experiment, ONs wake up in predefined timeslots. ONs wake up 45 times at most. ONs have to finish reconfiguration during these uptimes. To generate scenarios, a combination of two parameters is used. First, the total number of overlaps that ONs hosting a *measurement* module have with the *aggregator* (*Nb.Ovlp.* for short). This parameter influences $s_{\mathfrak{o}_{i,j}}$ of Eq. 5.1. Second, the order of uptime periods between ONs hosting a *measurement* module and the *aggregator* (*Upt.Order* for short). This parameter influences $s_{\mathrm{u}_j(t_{a_i})}$ of Eq. 5.2. Both uptimes and overlaps are uniformly distributed throughout the time slots of the experiment.

Figure 5.3 illustrates how these parameters influence the generation of scenarios. In the base scenario, $ON1$ always wakes up before $ON0$. There are also two overlaps in the time slots (TS) 2 and 4. From the base scenario, two other categories of scenarios have been generated. The first one modifies the number of overlaps. This is illustrated by *Scenario 1*, which shows a change with three overlaps instead of two. This should have an impact when not using an RN. The second one modifies the uptime order. This is illustrated by *Scenario 2*, where the order of uptime between $ON0$ and $ON1$ is changed. This should have an impact when using an RN (i.e., *rn*).

The number of overlaps parameter is called *Nb.Ovlp.*. The number of overlaps can take three values: 7, 15, or 30. The uptime order is called *Upt.Order*. Three different orders have been randomly generated uto_0, uto_1, uto_2.

5.4.2 Metric and Infrastructure

The total reconfiguration duration is measured. This corresponds to the time at which all ONs have completed their reconfiguration. It is composed of all uptime and sleeping periods needed for all ONs to complete the reconfiguration. This includes time spent to complete actions, waiting duration, and sleeping periods.

Impact of low bandwidth or packet loss on the reconfiguration is not measured. Experiments have been conducted on a cluster of seven Raspberry Pi 4, equipped with a Cortex-A72 CPU. These Raspberries have been connected to Grid5000[1], a scientific test-bed.

To simulate the uptime periods of ONs, Concerto-D instances are launched using a timer. This timer is used by Concerto-D to track when the execution of reconfiguration program should be suspended. When the timer goes off, Concerto-D suspends the execution, saves its state, and exits. The Concerto-D instance is launched again at the next uptime period. Each Concerto-D instance manages one module (i.e., *observer* or *aggregator*). An external controller is responsible to orchestrate the launch of each Concerto-D instance and gathering metrics for each. The code of this orchestrator and of the Concerto-D scripts are available here [2].

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

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

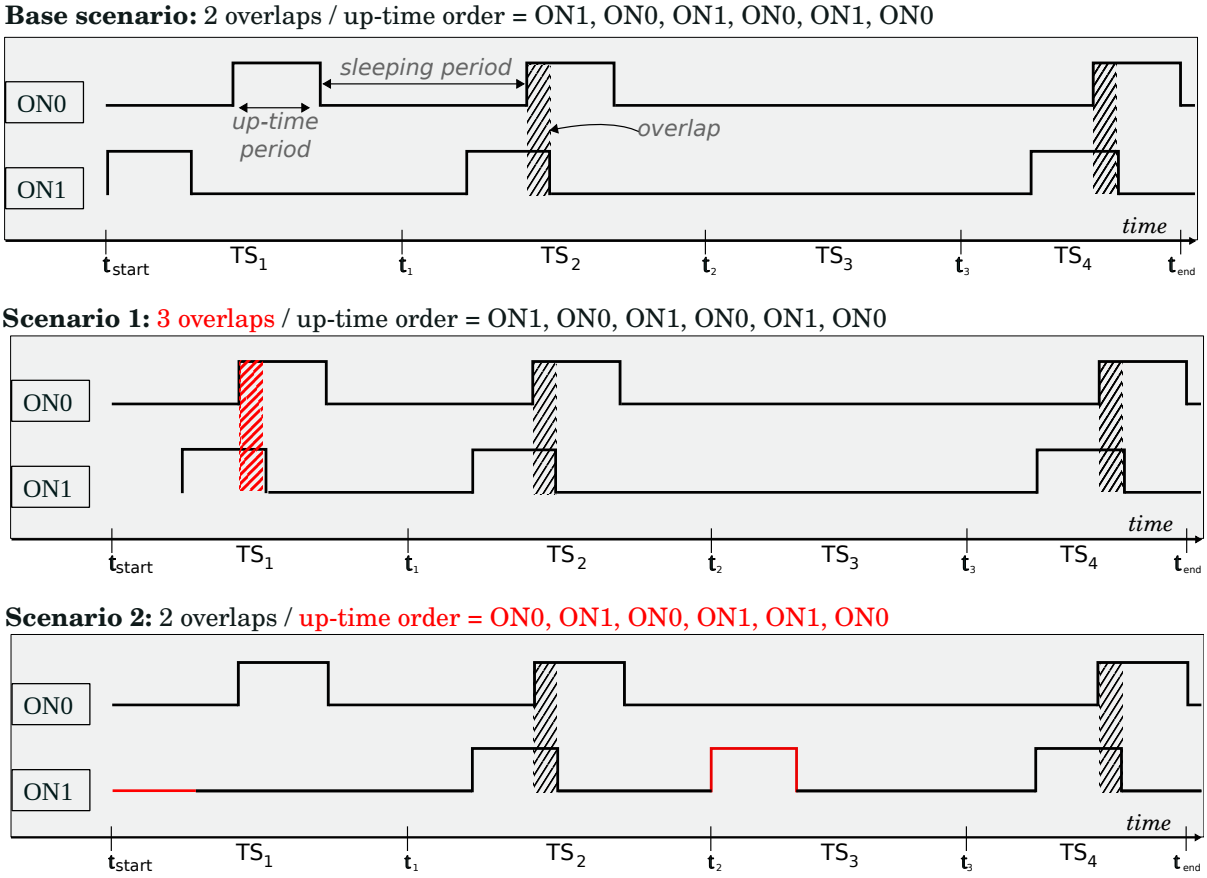


Figure 5.3: Illustration of three different uptime scenarios with two ONs. A scenario is divided into time slots (TS). The first scenario is the base to compare to others. The second scenario differs from the base in the number of overlaps, while the third differs in the order of uptime periods.

5.4.3 Experimental results

Table 5.1 and Table 5.2 depict experimental results obtained by varying uptime order and number of overlaps, respectively. Each experiment has been executed four times and a mean is given. A maximum standard deviation of 0.06% for CONCERTO-D and 7% for Muse (due to a fully loaded CPU, as discussed later) have been observed.

Table 5.1 presents the results obtained by varying the uptimes' orders and fixing the number of overlaps to 15. Overall, reconfiguration duration when using RN is lower than without using RN, which is expected from the analytical study (see Section 5.3). Using RN reduces reconfiguration duration from 19% (from 1036.81s to 841.01s) to 90% (from 2604.93s to 267.04s).

Reconfiguration durations without using RN remain stable among the three uptime orders. The main reason being that waiting for the next overlap overshadows the uptime order. On the other hand, the results are sensitive to the uptime order when using an RN. Reconfiguration duration for deployment varies from 257.41s to 645.38s (151% variation). For update, reconfiguration duration varies from 259.88s to 841.01s (224% variation). This is expected from the analytical study. Reconfiguration duration using RN is mainly sensitive to the uptime order (Equation 5.2).

Table 5.2 presents results obtained by varying the number of overlaps during experiments and fixing the uptimes order to uto_0.

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 5.1: Deployment and update durations (in seconds) when varying uptimes' orders: results for Concerto-D with/without RN and Muse, 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	$Not finished$	6090,51	274,50	3737,02	1832,47	257,41	2280,93	656,48	240,05
	ard_1	$Not finished$	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 5.2: Deployment and update durations (in seconds) when varying number of overlaps: results for Concerto-D with/without RN and Muse, for two random draws of actions duration ard_0 and ard_1.

Using an RN reduces deployment and update duration from 13% (from 269.64s to 233.20s) to 96% (from 6441.27s to 266.99s). When not using an RN, reducing the number of overlaps has a high impact on the reconfiguration duration. Reconfiguration duration can vary up to 693% for *deploy* (from 812.35s to 6441.27s), and 869% (from 629.66s to 6100.69s) for *update*. For 7 overlaps, Muse does not finish its reconfiguration for *deploy*. When using an RN, the number of overlaps has a much lower impact compared to without using an RN. From 30 to 7 overlaps, the reconfiguration duration when using an RN varies from 240.05s to 274.50s at most (14% increase).

These results give an experimental answer to RQ1 and RQ2. Deployment and update durations when using RN are always shorter than without using RN. When using an RN, the reconfiguration duration reduction ranges from 13% to 96% in scenarios with varying number of overlaps and uptime orders. The uptime order has an impact mainly when using an RN, with a maximum variation of 224% across the three different uptime orders. The number of overlaps has a low impact, up to 14% from 30 to 7 overlaps. Without using an RN, the reconfiguration duration is mostly sensitive to the number of overlaps, with a maximum variation of 869% across different numbers of overlaps.

5.5 Impact of uptime reduction rate

The second experiment aims at experimentally validating the theoretical results obtained for RQ3. This experiment has been realised using the first version of the Concerto-D prototype. The first version of the Concerto-D prototype has several differences from the version presented in Chapter 4 (second version). The first prototype relies on more synchronisations to coordinate the execution of reconfiguration programs compared to the second version. The differences between the two are described in the following.

```
sc = Assembly("agg_assembly")
sc.add_component("agg", "Aggregator")
for i in range(nb_obs):
  sc.connect("agg", f"cfg_obs_{i}", f"cfg_obs_{i}", f"obs_{i}")  # sync
  sc.connect("agg", f"use_obs_{i}", f"srv_obs_{i}", f"obs_{i}")  # sync
sc.push_b("agg", "create")
sc.wait("agg")

# Global synchronisation
for i in range(nb_obs):
 sc.wait(f"obs_{i}")

```

Listing 5.6: Deployment of the aggregator.

```
sc = Assembly("obs_n_assembly")
sc.add_component(f"obs_{n}", "Observer")
sc.connect(f"obs_{n}", f"cfg_obs_{n}", "agg", f"cfg_obs_{n}")  # sync
sc.connect(f"obs_{n}", f"srv_obs_{n}", "agg", f"use_obs_{n}")  # sync
sc.push_b(f"obs_{n}", "create")

# Global synchronisation
sc.wait("agg")
for i in range(nb_obs):
 sc.wait(f"obs_{i}")

```

Listing 5.7: Deployment of observer n

5.5.1 Differences between first and second Concerto-D prototypes

The two main differences impacting reconfiguration duration for the first prototype compared to the second are presented. These differences are the synchronisations for *con* and *dcon* actions and addition of global synchronisation at the end of the programs.

***con* and *dcon* synchronisations** In the first Concerto-D prototype, synchronisation were added for *con* and *dcon* actions. The different program executions had to synchronise with their peers when reaching *con* or *dcon* actions. The main reason was to keep the semantics of decentralised actions consistent with the centralised ones. In the centralised version, actions, including *con* and *dcon*, are sequentially executed. In the decentralised version, *con* and *dcon* actions are duplicated. This means that, to ensure a similar execution, synchronisations were added to ensure that both programs completed these actions before continuing.

Global synchronisation Reconfiguration programs used in experiments are modified versions of *deploy* and *update* programs used in first experiment. They are depicted on Listing 5.6, 5.7, 5.8, 5.9. The main difference is the global synchronisation at the end of each program. Each ON waits for every other ON to finish their reconfiguration before terminating. Additionally, each *con* action has to be synchronised between *aggregator* and *observer*. This results in a much higher constrained coordination than that of the

```
sc = Assembly("agg_assembly")
sc.push_b("agg", "update")
for i in range(nb_obs):
 sc.wait(f"obs_{i}")
 sc.push_b("agg", "deploy")
sc.wait("agg")

# Global synchronisation
for i in range(nb_obs):
 sc.wait(f"obs_{i}")

```

Listing 5.8: Update of the aggregator.

```
sc = Assembly("obs_n_assembly")
sc.push_b(f"obs_{n}", "update")
sc.push_b(f"obs_{n}", "create")

# Global synchronisation
sc.wait("agg")
for i in range(nb_obs):
 sc.wait(f"obs_{i}")

```

Listing 5.9: Update of observer n.

first experiment.

5.5.2 Uptime scenarios

Uptime scenarios have been generated similarly as in the first experiment. ONs wake up for short duration and sleep for approximately 200s between each awakening. During experiments, ONs wake up 60 times at maximum.

Scenarios have been generated using similar but different parameters than the first experiment. Overlaps between aggregator and observer are expressed as percentage instead of number. The overlap rate represents the average percentage of time the aggregator and each observer overlap according to the total aggregator uptime. For instance, if the aggregator overlaps 10s with obs_1 and 20s with obs_2 during 150s of *aggregator* uptime, the overlap rate is equal to $((10 + 20)/2)/150 = 10\%$. Overlap rates are given as range (2-5%, 20-30% or 50-60%).

5.5.3 Experimental setup and infrastructure

In this experiment, *deploy* and *update* programs are executed sequentially. ONs first deploy *aggregator* and *observer* then update them. The maximum uptime duration is the maximum amount of time spent by ONs to complete both programs. For instance, if the aggregator spent 120s uptime and one observer 110s uptime, the maximum uptime duration is equal to 120s. The maximum sleeping duration is the maximum amount of time spent by ONs sleeping. The maximum reconfiguration duration is equal to the maximum of the sum of uptime and sleeping duration. Note that, as an ON could have a maximal value for one metric but not the others, the equation $uptime + sleeping = reconfiguration$ is not always true.

Overlap rate		2-5%				
Reduction rate		100%		50%		0% (ref)
		time (s)	gain (%)	time (s)	gain (%)	time (s)
ard_0	uptime duration	56.11	-88	271.3	-40	451.32
	sleeping duration	2230.36	+45	1998.72	+30	1543.04
	reconf duration	2254.58	+13	2255.5	+13	1995.09
ard_1	max uptime duration	56.08	-88	271.16	-40	451.4
	sleeping duration	2220.99	+44	1996.69	+29	1543.45
	reconf duration	2253.51	+13	2255.64	+13	1995.52
Overlap rate		20-30%				
Reduction rate		100%		50%		0% (ref)
		time (s)	gain (%)	time (s)	gain (%)	time (s)
ard_0	uptime duration	54.84	-86	247.39	-37	391.25
	sleeping duration	1971.38	+50	1799.81	+37	1312.75
	reconf duration	2002.49	+17	2034.82	+19	1704.53
ard_1	uptime duration	54.67	-86	250.34	-36	391.12
	sleeping duration	1976.56	+50	1843.63	+40	1313.36
	reconf duration	2002.4	+17	2079.43	+22	1705.05
Overlap rate		50-60%				
Reduction rate		100%		50%		0% (ref)
		time (s)	gain (%)	time (s)	gain (%)	time (s)
ard_0	uptime duration	58.22	-71	166.1	-17	199.41
	sleeping duration	2612.01	+303	1196.86	+85	648.6
	reconf duration	2642.20	+218	1348.25	+62	829.77
ard_1	uptime duration	58.65	-73	168.11	-23	218.54
	sleeping duration	2646.90	+253	1196.91	+60	750.33
	reconf duration	2673.87	+178	1350.09	+40	961.57

Table 5.3: Comparison of results for different uptime reduction rates using an RN, taking the rate 0% as a reference (in grey).

Experiments are conducted using an RN. Experiments are conducted on Grid5000, a scientific test-bed, to facilitate reproducibility of experiments. Reconfiguration programs are executed by ONs from the UvB cluster hosted in Sophia-Antipolis, France. ONs of the cluster have 2 Intel Xeon X5670 CPUs 4.

5.5.4 Experimental results

Results are compared for different uptime reduction rates. This rate is denoted $R(\mathfrak{u}^t)$ (see Section 5.3). When $R(\mathfrak{u}^t) = 0\%$ ON is awake until the end of initial uptime period. When $R(\mathfrak{u}^t) = 100\%$, ONs go back to sleep as soon as entering an inactive state. $R(\mathfrak{u}^t) = 50\%$ is a compromise.

Table 5.3 gathers results for three overlap rates (2-5%, 20-30%, 50-60%), two random draws of transition duration (ard_0 and ard_1) and the three reduction rates $R(\mathfrak{u}^t)$ (0%, 50% and 100%). Impact of using reduction rate of 50% or 100% compared to 0% is shown in green and red. Green and red colours are used to express whether a difference is beneficial or not. A difference is beneficial if the sleeping duration increases, the uptime duration decreases, and the reconfiguration duration decreases.

Sleeping as soon as the ON is inactive ($R(\mathfrak{u}^t) = 100\%$), drastically decreases uptime

duration and increases sleeping duration in all cases. It also increases reconfiguration duration in all cases. These results are expected. Reducing uptime duration reduces the amount of time ONs have to do reconfiguration.

While experiments are conducted using an RN, reconfiguration duration is still sensitive to the overlap rate. This challenges the conclusions made in the first experiment. In the first experiment, it is observed that the number of overlaps does not have a significant impact on reconfiguration duration when using RN. However, it is shown that a lower overlap rate increases reconfiguration duration, for all reduction rates and both transition durations. For 2-5% overlap rate, reconfiguration duration is about 2255s for 100% and 50% uptime reduction rate, and 1995s for 0% rate.

An explanation of this phenomenon could be that the impact of overlap rate is more visible due to the additional synchronisations of reconfiguration programs. This is especially the case due to the high number of synchronisations due to connections. During deployment, the aggregator and each observer must complete two connections, one after the other. When not having overlapping uptimes, the aggregator and observers must wait for the next uptime to receive information for the first connection, then the next uptime again to receive information for the second connection. They need at least two uptimes to complete both connections. When having overlapping uptimes, both connections can be made during the same overlap, during a single uptime. This phenomenon should be very visible when comparing high overlap rate (50-60%) with low overlap rate (2-5%). Reconfiguration duration for the 50-60% rate is less than half the amount of duration for the 2-5% rate. Therefore, the overlap rate may still have an effect on reconfiguration duration, even when using an RN. This phenomenon may be less visible when the number of synchronisations is low (as seen in Section 5.4).

Impact of reduction rate on results varies significantly according to the overlap rate. For 2-5% overlap rate, reconfiguration duration is increased by 13%. For 50-60% rate, it is increased by 218%. This is expected from the theoretical study. With an initially low overlap rate, sleeping as soon as possible degrades the initial overlap rate to a small extent. With initially high overlap rate, sleeping as soon as possible degrades the overlap rate to a large extent. Notably, with 100% reduction rate, uptime, sleeping and reconfiguration duration have values in similar order of magnitude regardless of the overlap rate or transition duration.

Reducing uptime by 100% seems to be better than 50% for 2-5% and 20-30% overlap rates. For about the same reconfiguration duration, uptime duration is reduced by larger amount for 100% than 50%. For 50-60% overlap rate, results are more nuanced. Reconfiguration duration increases to an additional order of magnitude for 100% compared to 50%. Uptime duration decreases by large amounts.

Overall, results provide observations favourable to a reduction of initial uptime duration in the presence of an RN when overlap rate is low. Low overlap rate is the most plausible scenario for the DAO. When RN is present, reducing an already low overlap rate has little impact on reconfiguration duration. This could significantly reduce the amount of uptime required for reconfiguration, even for reconfiguration that requires lots of synchronisations between ONs (56s uptime in this case).

5.6 Discussion

This section discusses the results obtained using Muse and its performances on constrained hardware. It also discusses how the waiting duration (i.e., waiting for a depen-

dency resolution) could overshadow the impact of parallelism during reconfiguration.

Muse and Concerto-D comparison Results obtained using Muse follow the same trend as Concerto-D without using RN. As for Concerto-D, reconfiguration duration with Muse is sensitive to the number of overlaps but not to the uptime order. This supports results obtained from the implementation of Concerto-D. However, conclusions on performances of Concerto-D compared to Muse cannot be drawn. Notably, the fact that Concerto-D uses intra-module parallelism and Muse uses *module* parallelism. These performances may be overshadowed by implementation-related issues. During experiments, Muse has been used as-is, without modification to the prototype or the core engine. Muse relies on Pulumi to conduct reconfiguration (see Section 3.1). Pulumi is a reconfiguration solution that targets Cloud environments, containing a large amount of resources. This solution may not be suited for constrained nodes such as Raspberry Pi.

During experiments, it was observed that Muse imposes a high CPU load on Raspberry Pis. CPU load was measured to be at 100% most of the time. Analysis of the log exposed a very long initialisation time of Muse for each uptime period (e.g., up to 20s). This long initialisation prevented Muse to receive or send messages during these periods. This leads to Muse periodically missing overlaps, because it is busy initialising. This increases the waiting duration of Muse for all experiments, thus the overall reconfiguration duration. A better comparison of Muse and Concerto-D is done in Section 4.4.

5.7 Conclusion

In this Chapter, it is shown that decentralised reconfiguration on ONs can significantly delay the duration of the reconfiguration. This is done due to the fact that reconfiguration can only progress during uptime, and because non-synchronised uptimes lead to low opportunities for synchronisation. Results show that the number of overlap and uptime orders both have an impact on the reconfiguration duration. Results are validated both theoretically and experimentally. Experiments have been conducted using Concerto-D and Muse.

Additionally, results show that the uptime spent waiting for a dependency to be resolved can be decreased, while not significantly increasing the overall reconfiguration duration. This can be done by preemptively putting the ON back to sleep after a certain period of time. Putting an ON to sleep quicker allows to reduce uptime duration, but decreases the opportunities of ONs to communicate. To this end, the Relay Node is used to carry communication between ONs. Using this strategy, the ON can be awake strictly for reconfiguration purposes (execution transitions or synchronise with other ONs).

The next chapter aims at completing the results obtained on the duration of the reconfiguration by providing results on the energy consumption of such reconfiguration. To this end, experiments are conducted in a simulator, equipped with energy models to estimate the energy consumption due to the reconfiguration. Concerto-D is used as a reference to simulate the activity of the nodes during reconfiguration (e.g., execute transition, send/receive messages during coordination).

Chapter 6

Energy consumption evaluation of a decentralised reconfiguration

Contents

This chapter aims at evaluating and studying the energy consumption due to the coordination of a decentralised reconfiguration of ONs of the DAO-CPS. It aims at studying different parameters relevant for the DAO use case. Simulation is used to allow high control over these parameters, large-scale experiments, and energy consumption estimation using energy models. A wireless communication model is used to simulate different network topologies. The uptime and sleeping periods of ONs are simulated according to the

DAO use case (i.e., random wake-up once every hour). The execution and coordination of reconfiguration actions are also simulated. The energy consumption due to ONs' uptime/sleeping periods, the execution of reconfiguration actions and the communication due to coordination is evaluated. Results of these evaluations are compared and discussed according to each parameter relevant for the DAO use case. At the end of this chapter, conclusions about the work of this chapter and its limitations in terms of assumptions and realism are presented.

6.1 Introduction

Reconfiguration involves computing and networking activities besides CPS's main activities. These activities lead to an energy consumption overhead (see Section 3.3). The study of such overhead has been widely studied in the literature for different use cases. These use cases often consider a central entity distributing update data to nodes of the CPS, which is installed locally and independently on each node. Nodes often do not sleep, and the connectivity between nodes is stable. The literature doesn't extensively study the reconfiguration in environments with scarce energy and connectivity. This chapter aims at studying the energy consumption due to reconfiguration in such an environment. Such reconfiguration is coordinated between sleeping nodes without a central entity (i.e., decentralised reconfiguration).

Experiments are conducted in a simulated environment. The simulation of the reconfiguration is based on experiments conducted using the Concerto-D prototype under the characteristics of the Arctic Tundra (see Section 5.2.1). The *deploy* and *update* coordinations from Section 5.2.1 are simulated. During simulation, ONs wake up randomly once every hour for short duration. The wireless radio technologies used by ONs to communicate are simulated. Due to the low energy budget of ONs, LoRa and NB-IoT are simulated (see Section 3.3). The peer-to-peer network topology is simulated using a wireless network model. The Relay Node can be present in the topology to help other ONs communicate. Finally, different configurations of the service topology are considered. The service topology is defined by the position of each collaborative ONs in the network topology (i.e., *aggregator* and *observers*).

This chapter aims at evaluating and studying the impact on energy consumption of a decentralised reconfiguration according to the (i) number of ONs (ii) ONs' uptime duration (iii) simulated radio technology (LoRa or NB-IoT) (iv) usage of an RN to ease communications (v) network topology (vi) RN's position in the network topology and (vii) service topology.

6.2 Experimental parameters

This section presents the different parameters used to conduct simulation. Some of these parameters are extracted from contributions dealing with the DAO use case.

Type of coordination The types of coordination considered in this chapter are the ones described in Chapter 5. These types of coordination are the coordination of the deployment and update of the *aggregator* and *observers*, which are typical types of reconfiguration coordination. These coordinations are referred to as *deploy* and *update*. They are depicted in Figure 5.1.

Uptime duration To save energy, ONs alternate between short uptime and long sleeping periods. Considering peer-to-peer networks, ONs can only communicate when their uptimes overlap. In the simulations, it is considered that ONs wake up randomly every hour for a short duration, which is representative of a real ON deployed in the Arctic Tundra [8]. In these conditions, the uptime duration has a significant impact on connectivity between ONs. Having longer uptimes could increase the number and duration of ONs' uptime overlaps. To highlight and evaluate this impact, the uptime duration of ONs is set to 1, 2, or 3min. This duration is considered homogeneous among ONs for each individual simulation.

Simulated radio technology Constrained and remote environments such as the DAO reduce the range of usable radio technology. ONs in the DAO-CPS are located hundreds of meters away from each other and have limited energy budget [6]. High-range and low-power technologies are relevant to allow large-scale and long-running deployments. LPWAN radio technologies combine high-range and low-energy consumption at the cost of lower bandwidth (see Section 3.3). In this category, two radio technologies provide relevant features: LoRa and NB-IoT.

Due to the small opportunities to communicate, ONs have to spend a significant amount of time trying to communicate with neighbours. This may account for a large part of the energy consumption. Simulations using LoRa or NB-IoT allow to see the impact of the different technologies on energy consumption due to communication.

Relay Node, network and service topologies During the coordination of *deploy* and *update*, communication between *aggregator* and *observers* is enabled by forwarding messages between neighbours (see Chapter 2). The hop count required to reach a neighbour depends on the network and service topologies.

As a reminder, the network topology is formed by all the peer-to-peer network links that can be created by ONs. To be able to create such links, ONs must be in range, with no obstacle preventing communication (i.e., neighbours). The service topology corresponds to the position of collaborative ONs in the network topology. In this chapter, it corresponds to the locations of the *aggregator* and *observers* in the network topology.

In this chapter, network and service topologies relevant for the DAO use case are explored. Network topologies that could suit the Arctic Tundra context are extracted from the literature (i.e., clique, star, grid, ring, chain [83–87]). These topologies are studied individually, but may be combined to create larger and more complex ones. In these topologies, communication between *aggregator* and *observers* is enabled by forwarding data between ONs in the network topology until a recipient is reached.

Figure 6.1 shows the network and service topologies studied in this chapter. *Aggregator* and *observers* are hosted on ONs or RN. In the clique, all ONs are neighbours. This topology suits small areas with few obstacles, where all ONs are in range of each other. In the star topology, a central ON is neighbour to multiple, isolated ONs. In the grid topology, ONs have between two and four neighbours. ONs located at the corner of the grid may have more difficulty to communicate with other ONs than those located near the center. In the ring topology, each ON has two neighbours. This topology can impose a high hop count for ONs to communicate when located at opposite sides of the ring. In the chain topology, ONs also have two neighbours with the exception of the ONs located at the extremities of the chain. This topology can impose a higher hop count than ring for ONs to communicate, when ONs that want to communicate are located near the extremities of the chain.

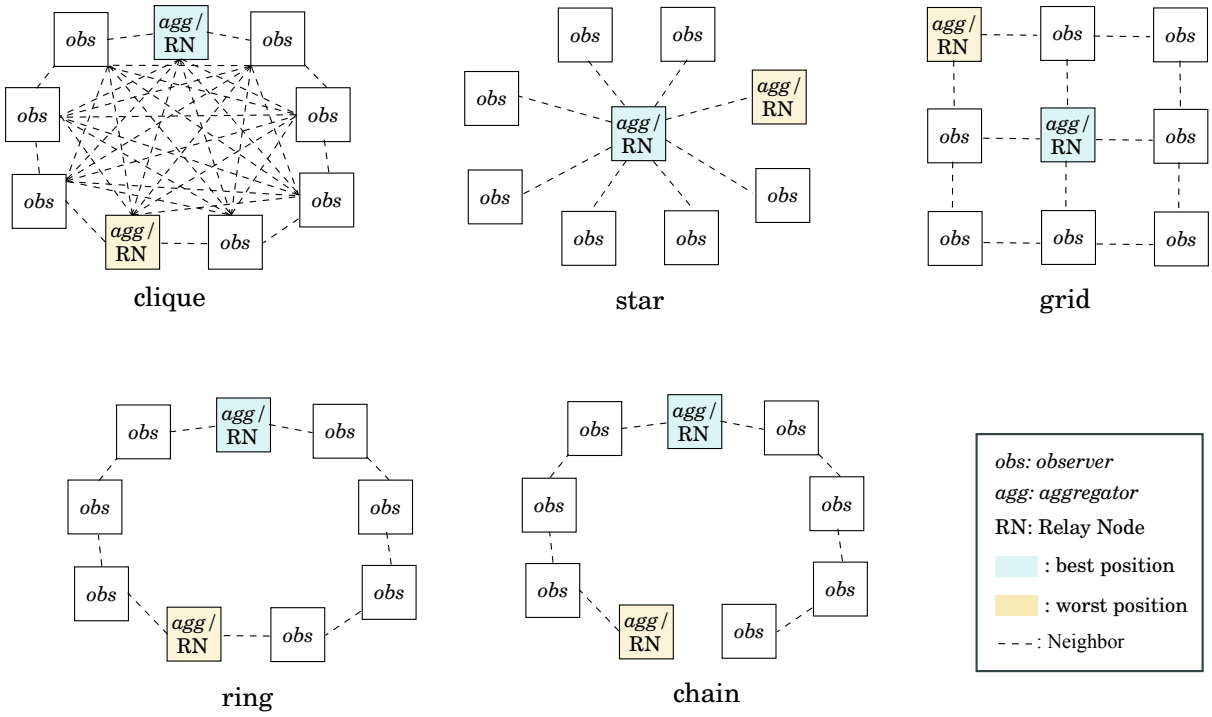


Figure 6.1: Studied network and service topologies. *Observers* represented on the figure are hosted on ONs. Best and worst *aggregator* and RN's positions are represented. The *aggregator* is hosted on the RN when both share the same position. Otherwise, the RN hosts an *observer*.

The service topology is defined by the positions of the *aggregator* and *observers* in the network topology. Due to the environment, the *aggregator* and *observers*' positions may vary. A high hop count between the *aggregator* and *observers* may imply longer and more energy consuming communications compared to a low hop count.

Additionally, the position of the RN in the network topology is also considered. The impact of the RN on the coordination's energy consumption may vary depending on its position in the network. For instance, having few neighbours decreases the number of ONs to which the RN can be made available. In the following, only the *aggregator* and RN's positions are considered for simplicity.

To cover most cases, the best and worst *aggregator* and RN's positions are studied. In this chapter, ONs' (and RN's) positions are characterised using two factors: the ON's degree (i.e., number of neighbours), and the ON's maximum hop count to other ONs. The maximum hop count of an ON is the maximum number of hops required by this ON to reach any other ON. For instance, the maximum hop count of the ON at the center of the star is one. The maximum hop count of an ON located at a branch of the star is two. A position with a high degree and low maximum hop count allows the *aggregator* to reach lots of *observers* in a few hops. Such a position also allows the RN to be available to lots of ONs (due to high degree). For instance, the RN placed at the center of the star can be available to all other ONs. The RN located at a branch of the star can be available only to the center of the star. For a given network topology, an ON is considered in the best position when it has maximum degree and minimum hop count to other ONs. An ON is considered in the worst position when it has minimum degree and maximum hop count to other ONs. The service topology is considered best when the *aggregator* is in the best position. The service topology is considered worst when the *aggregator* is in the

worst position.

The service topologies and their respective best and worst positions are summarised in Figure 6.1. Clique's and ring's ONs' degree and maximum hop count are homogeneous. For these topologies, best and worst positions are equivalent. For other topologies, the best positions are the center of the star and grid, and the middle of the chain. The worst positions are the branch of the star, the corner of the grid, and the extremity of the chain. Note that the *aggregator* is hosted on the RN when both share the same position. Otherwise, the RN hosts an *observer*.

6.3 Experimental setup

Experiments are conducted in simulated environments. This section presents the simulation of ONs and their reconfiguration activities, and the experimental setups used for studying the impact on energy consumption of the different parameters presented in Section 6.2.

6.3.1 Use case simulation

Experiments are conducted on ESDS [76], a simulator using flow-level wireless network model publicly available[1]. Simulation provides flexibility and saves time compared to setting up and executing on a real infrastructure. The equivalent of several years of real execution time is reduced to few hours of simulation. Simulation allows for easier reproducibility, as executions and results are not bound to a specific hardware platform.

ESDS has been used in papers with similar use case (i.e., non-synchronised sleeping ONs simulated in the observed conditions of the Arctic Tundra [8] [88]). It provides energy models to predict energy consumption estimations from simulated ON. Communication models provided by ESDS allow to simulate different network topologies with parametrised wireless communications (e.g., bandwidth, latency). Each ON can be simulated as a Python function, allowing great flexibility to simulate activity on ONs.

In ESDS, an API is provided to simulate ONs activity and to retrieve the energy consumption of each ON. For each ON, the sleeping and uptime periods, the execution of *deploy/update* module operations, and communications are simulated.

Each ON can go through 4 states: *off*, *idle*, *stress* and *pull*. In the *off* state, the ON sleeps. It cannot receive or send messages or execute reconfiguration actions. In the *idle* state, the ON is awake. It is not executing reconfiguration actions or sending messages, but it can receive messages. The *stress* state simulates the execution of individual transitions in the life-cycle of Concerto-D components. In this state, the ON is awake and executes these transitions. Finally, in the *pull* state, the ON is awake but requires coordination data for one or multiple reconfiguration actions.

In the *stress* state, only the load generated on the ON to execute transition is considered. The load varies depending on the concrete change triggered by this transition. This change can be a simple activation of service to a heavy installation of software on the ONs. In the simulations, the worst-case is considered: each transition fully stresses the ON. As for Chapter 5, the duration of each transition is assigned to a random duration, following a lognormal distribution bounded between 1s and 30s, where low values are more represented.

[1] `https://gitlab.com/manzerbredes/esds`

Table 6.1: Power calibration and bandwidth

Power values	P_{idle}	1,339W [90]
	P_{stress}	2,697W [90]
	P_{pull} (LoRa)	+0.16W [8]
	P_{pull} (NB-IoT)	+0.65W [8]
Bandwidth (latency)	LoRa	50 kbps (0s) [8]
	NB-IoT	200 kbps (0s) [8]

Table 6.2: Simulation parameters. In the first experiment, the network and service topologies are fixed. In the second experiment, the uptime duration, coordination name, and simulated radio technology are fixed.

	Experiment 1	**Experiment 2**
# of ONs	{5+1, 15+1, 30+1}	{8+1, 15+1, 24+1} [65]
Relay Node position	{NA, best}	{NA, best, worst}
Upt duration	{1min, 2min, 3min}	1min [8]
Coordination name	{*deploy*, *update*}	*update*
Radio technology	{LoRa, NB-IoT}	LoRa [8]
Network topology	*star*	{*clique*, *star*, *grid*, *ring*, *chain*}
Service topology	best	{best, worst}

In the *pull* state, ONs request data once per second. This high frequency is motivated by the uptime characteristics of the use case. ONs wake up once every hour for very short periods, leading to very few overlaps. Such high frequency reduces the probability of missing an overlap. In this chapter, coordination data is assumed to be a simple flag. Each request has a measured fixed size of 257 bytes, close to the lower bound of common request sizes[2].

Table 6.1 summarises power calibration and radio technology parameters. The power calibration is done from the energy consumption of a Raspberry Pi-based ON, previously used in papers dealing with CPS deployments in the Arctic Tundra [65,89]. The extreme values measured in [90] are used (1.339W for P_{idle}, and 2.697W for P_{stress}). When an ON is sleeping, its energy consumption is assumed to be null. Two suitable radio technologies for the DAO-CPS [8] are simulated: LoRa and NB-IoT. The communication cost (send or receive) is calibrated from [8] (additional 0.16W for LoRa, and 0.65W for NB-IoT). Bandwidths are 50kbps for LoRa and 200kbps for NB-IoT. Latency is assumed to be null. Simulations are run with and without an RN, and the number of ONs can vary up to 31 ONs [65].

Table 6.2 summarises simulation parameters for both experiments. The setup for both experiments is described in the following.

6.3.2 Study of the impact of uptime duration, radio technology and usage of RN

In the first experiment, the impact of uptime duration, radio technology and the usage of an available RN for communications is studied. The network and service topologies are

[2]https://www.chromium.org/spdy/spdy-whitepaper/

fixed to the *star* and best topologies, respectively. Simulations are run using 1 *aggregator* and 5, 15, or 30 *observers*. ONs' uptime duration is set either at 1, 2 or 3 min [8]. Two coordination cases are simulated: *deploy* and *update*. Finally, LoRa and NB-IoT are simulated.

ONs communicate at 1-hop without propagating requests. ONs simulation states and coordination duration are computed using a dependency graph based on the life-cycles of the *aggregator* and *observer* modules (see Chapter 5). Each arc represents a transition (from life-cycle of module), directed either toward the next transition on the same ON, or toward a transition on the life-cycle of a remote ON. Computing a graph traversal gives the ONs states and their duration for each individual ON. Computing the longest path gives the total coordination duration. The graph creation is inspired by [38].

The ON states, their duration, and the coordination duration are given as input to ESDS. For each ON, ESDS simulates these states and gives the energy consumed. The draw of transition duration, uptime schedules, ONs states, and simulation results are available online.[3].

6.3.3 Study of the impact of network topology, service topology, and RN's position

In the second experiment, the impact of the network and service topologies, and RN's position in the topology is studied. To focus on these parameters, some parameters from the first experiment have been fixed. The type of coordination is fixed to *update* as it is considered more frequent than *deploy* (which is an initial deployment of modules). Radio technology is fixed to LoRa, as it has lower power consumption than NB-IoT. ONs' uptime duration is fixed to 1 min as it corresponds to the lowest uptime in [8]. *Clique*, *star*, *grid*, *ring* and *chain* network topologies are simulated. In these topologies, best and worst RN's positions (when present) and service topologies are simulated. To ensure a consistent comparison between topologies, a square shape for *grid* should be conserved. Simulations are run using a square number of ONs: 1 *aggregator* and 8, 15 or 24 *observers*.

In this experiment, requests need to be propagated between ONs to reach a recipient. ONs simulation states and coordination's duration are computed directly by ESDS. No knowledge of the network or service topology, or ONs' uptime schedules is assumed. Epidemic routing is used to propagate requests through the network until the recipient is reached. Each non-recipient ON propagates requests to each of its neighbours. Epidemic routing allows for better reliability and is well-suited for use cases with nodes with limited knowledge but implies a lot of duplicated messages [64]. In this use case, messages are very small, which should limit the energy consumption overhead due to duplicated messages. Simulation and results are available online[4][5].

For the first and second experiments, 200 random uptime schedules are generated to get a representative set of results. During each schedule, ONs wake up randomly once every hour. During experiments, a scenario represents a combination of parameters. Each scenario is run on each uptime schedule, for a total of 200 runs per scenario.

[3] `https://github.com/aomond-imt/reconfiguration-esds/releases/tag/greencom_2023`
[4] `https://github.com/aomond-imt/journal-pdc-experiments`
[5] `https://github.com/aomond-imt/journal-pdc-results`

6.3.4 Metrics

The accumulated energy consumed by all ONs under each scenario and each uptime schedule is composed of static and dynamic energy consumption. The static energy consumption is the energy passively consumed by ONs during uptime. The dynamic energy consumption is the energy consumed for the execution and coordination of reconfiguration actions (see Section 2.5). For 200 runs, the means of the accumulated static, dynamic and total energy consumed by all ONs under a scenario s are denoted $\overline{eStatic}(s)$, $\overline{eDynamic}(s)$ and $\overline{e}(s)$, respectively. $\overline{e}(s)$ is given by

$$\overline{e}(s) = \overline{eStatic}(s) + \overline{eDynamic}(s) \tag{6.1}$$

Distinguishing between $\overline{e}$ and $\overline{eDynamic}$ allows to consider when ONs wake up specifically for *deploy/update* and when ONs take advantage of existing uptimes. When ONs wake up specifically for *deploy/update*, $\overline{e}$ is considered. When ONs take advantage of existing uptimes, only $\overline{eDynamic}$ is considered.

$\overline{eComms}(s)$, included in $\overline{eDynamic}$, represents the mean of the accumulated energy consumed by communications for 200 runs under a scenario s. This allows to compare the coordination's energy consumption when using LoRa or NB-IoT. Finally, $\overline{t}(s)$ represents the mean of the duration of *deploy/update*, for 200 runs under a scenario s.

For each metric m, the percentage of variation $\%\Delta m(s_1, s_2)$ quantifies the variation of m from scenario s_1 to s_2.

$$\%\Delta m(s_1, s_2) = \frac{m(s_1) - m(s_2)}{m(s_1)} * 100, \tag{6.2}$$

where m can be either $\overline{e}$, $\overline{eDynamic}$, $\overline{eComms}$ or $\overline{t}$. The percentage of variation is used to study the impact of varying simulation parameters on energy consumption and coordination duration.

6.4 Impact of uptime duration, radio technology and usage of RN

This section presents and compares the energy consumed by ONs during coordination according to uptime duration, radio technology and the usage of an available RN for communication. Scenarios are created from the Cartesian product of simulation parameters in the "Experiment 1" column of Table 6.2.

In the following, the energy consumption is first studied when ONs wake up specifically for *deploy* and *update* ($\overline{e}$ and $\overline{t}$ are evaluated). Second, the energy consumption is studied when ONs take advantage of existing uptimes ($\overline{eDynamic}$ and $\overline{eComms}$ are evaluated).

6.4.1 Energy consumed when ONs wake up specifically for deploy and update

Table 6.3 presents $\overline{e}$ and $\overline{t}$ values for *deploy* and *update* according to the number of ONs, the uptime duration and usage of RN.

Table 6.3: Total energy consumption $\overline{e}$ and coordination duration $\overline{t}$ values for *deploy* and *update* according to the number of ONs, uptime duration, and usage of RN. Standard deviations are shown in parentheses. $\%\Delta\overline{e}$ and $\%\Delta\overline{t}$ quantify the variation of $\overline{e}$ and $\overline{t}$, between scenarios with and without an RN.

deploy						
6 ONs						
	$\overline{e}$			$\overline{t}$		
	no RN (kJ)	RN (kJ)	$\%\Delta\overline{e}$	no RN (hours)	RN (hours)	$\%\Delta\overline{t}$
1min uptime	63.82 (24.18)	1.39 (0.31)	97.82	143.19 (54.37)	1.46 (0.34)	98.98
2min uptime	61.46 (24.68)	2.51 (0.62)	95.92	68.96 (27.71)	1.42 (0.38)	97.94
3min uptime	59.78 (23.64)	3.40 (0.92)	94.31	44.71 (17.70)	1.34 (0.41)	97.00
16 ONs						
	$\overline{e}$			$\overline{t}$		
	no RN (kJ)	RN (kJ)	$\%\Delta\overline{e}$	no RN (hours)	RN (hours)	$\%\Delta\overline{t}$
1min uptime	245.94 (61.29)	3.68 (0.73)	98.50	207.05 (51.67)	1.51 (0.31)	99.27
2min uptime	248.08 (67.00)	6.61 (1.49)	97.34	104.42 (28.25)	1.48 (0.33)	98.58
3min uptime	240.56 (62.96)	9.31 (2.09)	96.13	67.49 (17.69)	1.50 (0.34)	97.78
31 ONs						
	$\overline{e}$			$\overline{t}$		
	no RN (kJ)	RN (kJ)	$\%\Delta\overline{e}$	no RN (hours)	RN (hours)	$\%\Delta\overline{t}$
1min uptime	572.02 (123.12)	6.91 (1.29)	98.79	248.65 (53.60)	1.53 (0.30)	99.38
2min uptime	566.06 (113.78)	11.82 (2.49)	97.91	122.98 (24.76)	1.49 (0.31)	98.79
3min uptime	578.35 (134.04)	16.50 (3.63)	97.15	83.76 (19.46)	1.52 (0.34)	98.19
update						
6 ONs						
	$\overline{e}$			$\overline{t}$		
	no RN (kJ)	RN (kJ)	$\%\Delta\overline{e}$	no RN (hours)	RN (hours)	$\%\Delta\overline{t}$
1min uptime	39.18 (17.67)	2.12 (0.46)	94.59	87.61 (39.68)	2.30 (0.55)	97.37
2min uptime	35.29 (18.19)	3.90 (1.02)	88.95	39.47 (20.42)	2.24 (0.58)	94.32
3min uptime	34.49 (17.75)	5.87 (1.40)	82.98	25.76 (13.31)	2.30 (0.54)	91.07
16 ONs						
	$\overline{e}$			$\overline{t}$		
	no RN (kJ)	RN (kJ)	$\%\Delta\overline{e}$	no RN (hours)	RN (hours)	$\%\Delta\overline{t}$
1min uptime	150.81 (44.89)	5.94 (0.95)	96.06	126.70 (37.83)	2.50 (0.43)	98.03
2min uptime	141.18 (43.27)	10.46 (2.20)	92.59	59.27 (18.24)	2.39 (0.52)	95.97
3min uptime	126.80 (49.71)	15.13 (2.99)	88.07	35.44 (13.97)	2.43 (0.48)	93.14
31 ONs						
	$\overline{e}$			$\overline{t}$		
	no RN (kJ)	RN (kJ)	$\%\Delta\overline{e}$	no RN (hours)	RN (hours)	$\%\Delta\overline{t}$
1min uptime	350.34 (107.06)	11.11 (1.66)	96.83	152.02 (46.58)	2.55 (0.40)	98.32
2min uptime	321.13 (83.48)	19.38 (3.66)	93.97	69.58 (18.15)	2.49 (0.47)	96.42
3min uptime	307.41 (88.20)	26.98 (4.59)	91.22	44.38 (12.78)	2.50 (0.42)	94.37

Coordination without using an RN For the smallest cluster size (6 ONs) and 1 min uptime duration, $\overline{e}$ is equal to 63.82 kJ for *deploy*, and to 39.18 kJ for *update*. For the biggest cluster size (31 ONs), $\overline{e}$ reaches 572.02 kJ for *deploy*, and 350.34 kJ for *update*. These high energy consumptions are expected due to the characteristics of the use case. Scarce connectivity between ONs leads to a significant amount of time required for coordination convergence. This is illustrated by $\overline{t}$ values. For 6 ONs and 1 min uptime

duration, *deploy* and *update* take in average 143.19 hours and 87.61 hours respectively. For 31 ONs, *deploy* and *update* take in average 248.65 hours and 152.02 hours respectively.

Increasing uptime duration Modifying uptime duration leads to a trade-off between energy saved from faster convergence and energy spent at each uptime.

For *deploy* and the smallest cluster size, increasing uptime duration from 1 to 3 min decreases $\overline{e}$ by 6.33% (from 63.82 to 59.78 kJ). For the largest cluster size, increasing uptime duration from 1 to 2 min slightly decreases $\overline{e}$ by 1.04% (from 572.02 to 566.06 kJ). When increasing uptime duration from 1 to 3 min, $\overline{e}$ slightly increases by 1.11% (from 572.02 to 578.35 kJ).

For *update* and the smallest cluster size, increasing uptime duration from 1 to 3 min decreases $\overline{e}$ by 11.97% (from 39.18 to 34.49 kJ). For the highest cluster size, $\overline{e}$ decreases by 12.25% (from 350.34 to 307.41 kJ).

Increasing uptime duration allows for a strong reduction in $\overline{t}$. For *deploy* and the smallest cluster size, increasing uptime duration from 1 to 3 min decreases $\overline{t}$ by 68.78% (from 143.19 to 44.71 hours). For *update*, $\overline{t}$ decreases by 70.60% (from 87.61 to 25.76 hours). For *deploy* and the largest cluster size, $\overline{t}$ decreases by 66.31% (from 248.65 to 83.76 hours). For *update*, $\overline{t}$ decreases by 70.81% (from 152.02 to 44.38 hours).

Coordination using an available RN When using an RN, ONs energy consumption drastically decreases: for 1 min uptime duration and all cluster sizes, $\overline{e}$ decreases by more than 97% for *deploy*, and by more than 94% for *update*. These reductions are expected, as using an RN drastically reduces *deploy*/*update* duration: for 1 min uptime duration having an RN reduces $\overline{t}$ by more than 97% under any scenario.

Finally, combining both increasing uptime duration and using RN increases energy consumption under all scenarios. When using an RN for *deploy*, increasing uptime duration from 1 to 3 min for the smallest cluster size increases $\overline{e}$ from 1.39 to 3.40 kJ. For the largest cluster size, $\overline{e}$ increases from 6.91 to 16.50 kJ. Similar variations can be observed for *update*. For the smallest cluster size, increasing uptime duration also increases $\overline{e}$ from 2.12 to 5.87 kJ. For the largest cluster size, $\overline{e}$ increases from 11.11 to 26.98 kJ. These variations are explained by the full availability of the RN to ONs. In such conditions, the impact of increasing uptime duration to reduce the *deploy*/*update* coordination duration is minimized.

6.4.2 Energy consumed when ONs take advantage of existing uptimes for deploy and update

In this section, only $\overline{eDynamic}$ is considered, as the *deploy*/*update* coordination is considered to be a task running among others, on ONs. The RN is also included in this assumption, as ONs use the RN not only for *deploy*/*update* but also for any type of collaboration. Table 6.4 presents $\overline{eDynamic}$ values for *deploy* and *update* according to the number of ONs, the uptime duration and usage of RN.

Coordination without using an RN For 1 min uptime duration, $\overline{eDynamic}$ for *deploy* is 221.84 J for the smallest cluster size, and 1081.37 J for the highest cluster size. For *update*, $\overline{eDynamic}$ is 222.38 J for the smallest cluster size, and 1274.14 J for the highest cluster size. These values are much lower than $\overline{e}$. $\overline{eDynamic}$ represents only a slight overhead in ONs' energy consumption.

Table 6.4: $\overline{eDynamic}$ values for *deploy*/*update* according to the number of ONs, uptime duration, and usage of RN. Standard deviations are shown in parentheses. $\%\Delta\overline{eDynamic}$ quantifies the variation of $\overline{eDynamic}$ between scenarios with and without an RN.

deploy			
6 ONs			
	$\overline{eDynamic}$		
	no RN (J)	RN (J)	$\%\Delta\overline{eDynamic}$
1min uptime	221.84 (31.29)	104.24 (4.83)	53.01
2min uptime	221.99 (32.49)	107.28 (6.90)	51.67
3min uptime	224.01 (33.38)	109.81 (10.37)	50.98
16 ONs			
	$\overline{eDynamic}$		
	no RN (J)	RN (J)	$\%\Delta\overline{eDynamic}$
1min uptime	575.59 (44.23)	255.52 (6.27)	55.61
2min uptime	623.47 (58.84)	265.23 (11.18)	57.46
3min uptime	658.32 (67.97)	273.76 (17.69)	58.42
31 ONs			
	$\overline{eDynamic}$		
	no RN (J)	RN (J)	$\%\Delta\overline{eDynamic}$
1min uptime	1081.37 (66.98)	450.39 (10.66)	58.35
2min uptime	1228.90 (80.45)	467.23 (18.88)	61.98
3min uptime	1381.37 (104.40)	483.82 (28.54)	64.98
update			
6 ONs			
	$\overline{eDynamic}$		
	no RN (J)	RN (J)	$\%\Delta\overline{eDynamic}$
1min uptime	222.38 (47.24)	111.13 (3.47)	50.03
2min uptime	222.48 (52.29)	121.88 (8.96)	45.22
3min uptime	219.72 (47.56)	136.31 (11.57)	37.96
16 ONs			
	$\overline{eDynamic}$		
	no RN (J)	RN (J)	$\%\Delta\overline{eDynamic}$
1min uptime	620.27 (87.62)	269.24 (9.53)	56.59
2min uptime	668.75 (92.83)	298.41 (26.61)	55.38
3min uptime	690.84 (116.43)	343.76 (36.19)	50.24
31 ONs			
	$\overline{eDynamic}$		
	no RN (J)	RN (J)	$\%\Delta\overline{eDynamic}$
1min uptime	1274.14 (141.42)	505.62 (19.83)	60.32
2min uptime	1476.30 (163.06)	568.08 (55.87)	61.52
3min uptime	1664.77 (177.83)	665.37 (81.07)	60.03

Increasing uptime duration When uptime duration increases, under most scenarios $\overline{eDynamic}$ increases, especially for large cluster sizes. For *deploy* and the smallest cluster size, increasing uptime duration from 1 to 3 min slightly increases $\overline{eDynamic}$ by 0.98% (from 221.84 to 224.01 J). For the largest cluster size, $\overline{eDynamic}$ increases by 27.74% (from 1081.37 to 1381.37 J). For *update* and the smallest cluster size, increasing uptime duration from 1 to 3 min slightly decreases $\overline{eDynamic}$ by 1.20% (from 222.38 to 219.72 J). For the largest cluster size, $\overline{eDynamic}$ increases by 30.66% (from 1274.14 to 1664.77 J). Increasing uptime duration leads to larger and more frequent overlaps between ONs. More overlaps lead to more ONs receiving communications, including non-intended transmissions. Receiving non-intended communications can add a significant overhead to $\overline{eDynamic}$.

Table 6.5: $\overline{eComms}$ values for *deploy*/*update* according to the number of ONs, the uptime duration, and usage of RN. Standard deviations are shown in parentheses. $\%\Delta\overline{eComms}$ quantifies the variation of $\overline{eComms}$ between scenarios using LoRa and scenarios using NB-IoT.

deploy			
6 ONs			
	LoRa (J)	NB-IoT (J)	$\%\Delta\overline{eComms}$
1min uptime	98.34 (30.95)	115.24 (36.08)	-17.19
2min uptime	98.75 (32.25)	115.16 (37.65)	-16.62
3min uptime	101.02 (33.35)	117.95 (38.99)	-16.76
16 ONs			
	LoRa (J)	NB-IoT (J)	$\%\Delta\overline{eComms}$
1min uptime	271.80 (43.50)	374.44 (58.07)	-37.76
2min uptime	320.73 (58.66)	441.04 (77.78)	-37.51
3min uptime	356.79 (65.93)	487.75 (88.15)	-36.71
31 ONs			
	LoRa (J)	NB-IoT (J)	$\%\Delta\overline{eComms}$
1min uptime	513.75 (65.13)	815.97 (95.02)	-58.83
2min uptime	665.64 (79.76)	1055.40 (115.45)	-58.55
3min uptime	823.80 (102.39)	1293.10 (149.06)	-56.97
update			
6 ONs			
	LoRa (J)	NB-IoT (J)	$\%\Delta\overline{eComms}$
1min uptime	121.43 (47.24)	136.81 (53.31)	-12.67
2min uptime	121.53 (52.29)	137.09 (59.07)	-12.80
3min uptime	118.77 (47.56)	133.45 (53.55)	-12.36
16 ONs			
	LoRa (J)	NB-IoT (J)	$\%\Delta\overline{eComms}$
1min uptime	376.87 (87.62)	456.47 (104.75)	-21.12
2min uptime	425.35 (92.83)	512.65 (110.47)	-20.52
3min uptime	447.44 (116.43)	538.26 (136.96)	-20.30
31 ONs			
	LoRa (J)	NB-IoT (J)	$\%\Delta\overline{eComms}$
1min uptime	812.87 (141.42)	1030.84 (174.41)	-26.81
2min uptime	1015.03 (163.06)	1284.27 (202.64)	-26.53
3min uptime	1203.50 (177.83)	1511.64 (216.01)	-25.60

Coordination using an available RN For *deploy*, 1 min uptime duration and the smallest cluster size, using an RN decreases $\overline{eDynamic}$ by 53.01% (from 221.84 to 104.24 J). For the largest cluster size, $\overline{eDynamic}$ decreases by 58.35% (from 1081.37 to 450.39 J). For *update*, 1 min uptime duration and the smallest cluster size, using an RN decreases $\overline{eDynamic}$ by 50.03% (from 222.38 to 111.13 J). For the largest cluster size, $\overline{eDynamic}$ is reduced by 60.32% (from 1274.14 to 505.62 J).

As for Section 6.4.1, combining the utilisation of an RN with longer uptime duration increases $\overline{eDynamic}$ in all scenarios. For the smallest cluster size, going from 1 to 3 min while using an RN increases $\overline{eDynamic}$ from 104.24 to 109.81 J for *deploy*, and from 111.13 to 136.31 J for *update*. For the largest cluster size, $\overline{eDynamic}$ increases from 450.39 to 483.82 J for *deploy*, and from 505.62 to 665.37 J for *update*.

Simulated radio technology Table 6.5 presents $\overline{eComms}$ values for *deploy* and *update*, according to the number of ONs, the uptime duration and the radio technology (i.e., LoRa, NB-IoT).

For *deploy*, the smallest cluster size and 1 min uptime duration, using NB-IoT instead

of LoRa increases $\overline{eComms}$ by 17.19% (from 98.34 to 115.24 J). For the largest cluster size, $\overline{eComms}$ increases by 58.83% (from 513.75 to 815.97 J). For *update*, the smallest cluster size and 1 min uptime duration, using NB-IoT instead of LoRa increases $\overline{eComms}$ by 12.67% (from 121.43 to 136.81 J). For the largest cluster size, $\overline{eComms}$ increases by 26.81% (from 812.87 to 1030.84 J).

NB-IoT has a higher consumption than LoRa under any scenario. The *deploy*/*update* duration using LoRa or NB-IoT is not shown in this chapter, but NB-IoT's higher bandwidth implies in average a negligible reduction of *deploy*/*update* duration (less than 3% in few scenarios). This is due to the small size of requests.

6.4.3 Discussion

Waking up ONs specifically for coordination implies very high energy consumption. Increasing ONs uptime duration to speed up coordination convergence can decrease energy consumption, by up to 12.25% at best. Having an available RN allows drastic energy consumption reduction, by up to 97% at best. However, in the Arctic Tundra, the availability of an RN to all neighbours may not always be guaranteed.

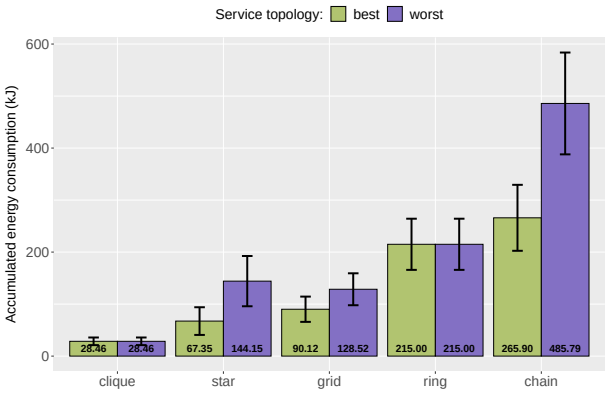
When ONs execute reconfiguration actions alongside their observation activities, only the dynamic energy used for the coordination of reconfiguration is taken into account. The dynamic energy represents only a fraction of the total energy consumption. Increasing the uptime duration only increases energy consumption, by up to 30.66% at worst. Having an RN is also always favourable with regard to energy consumption (up to 60.32% reduction). Finally, LoRa has a lower consumption than NB-IoT, in all scenarios.

Due to scarce connectivity between ONs, having an available RN to carry communications is an interesting option. Results show a very high energy consumption reduction in all scenarios. Realistically, additional issues should be dealt with to have such nodes in a system. Notably, building knowledge about RN's neighbour uptimes and dealing with clock drift. This could be the subject of future works. When an RN is not available, increasing uptime duration is only interesting when ONs are forced to wake up for reconfiguration. It allows to reduce the total energy consumption and the coordination duration, but increases dynamic energy consumption in all scenarios. As the dynamic energy is very low compared to the total energy consumption, it is preferable to do reconfiguration actions besides observation activities. Finally, due to the very small size of messages, using LoRa instead of NB-IoT is always a better choice.

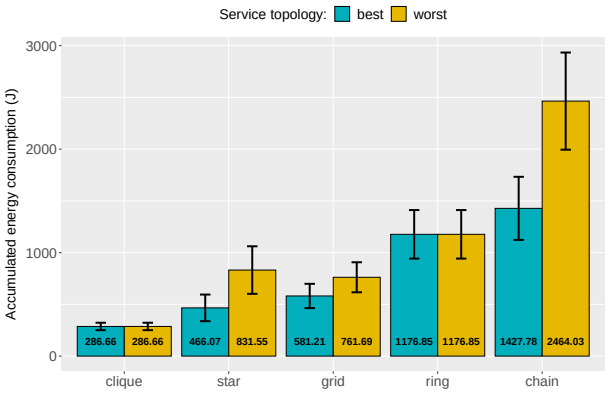
6.5 Impact of network topology, service topology and RN's position

This section presents and compares the energy consumed by ONs during coordination according to network topology, service topology, and RN's position. Scenarios are created from the Cartesian product of simulation parameters in the "Experiment 2" column of Table 6.2. Note that some parameters are fixed. The type of coordination is fixed to *update* as it is considered more frequent than *deploy*. Radio technology is fixed to LoRa as the first experiment shows better results using LoRa compared to NB-IoT. Finally, ONs' uptime duration is fixed to 1 min as it corresponds to the lowest uptime in [8].

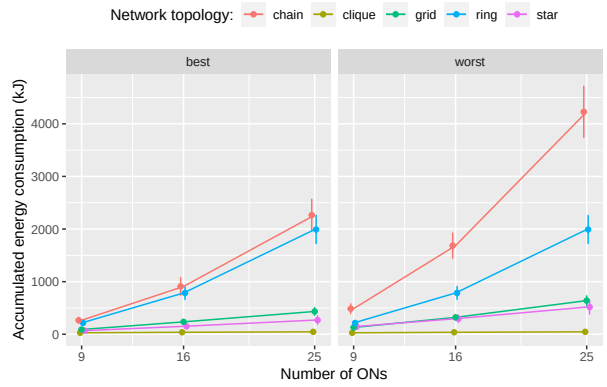
First, scenarios where an RN is not present are studied. Second, scenarios where an RN is present are studied. As for the first experiment, the total and dynamic energy consumption are studied.



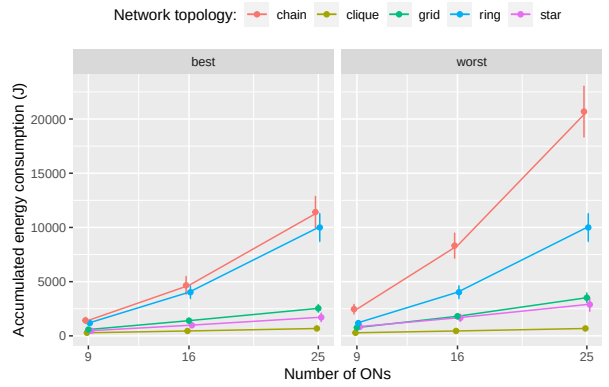
(a) $\overline{e}$ results for 9 ONs



(b) $\overline{eDynamic}$ results for 9 ONs



(c) $\overline{e}$ results according to the number of ONs



(d) $\overline{eDynamic}$ results according to the number of ONs

Figure 6.2: Simulation results without using an RN. $\overline{e}$ and $\overline{eDynamic}$ values are depicted according to the network and service topologies. Figures 6.2a and 6.2b show values for 9 ONs. Figures 6.2c and 6.2d show values according to the number of ONs.

6.5.1 Coordination without an RN

In the following, the RN is not present. ONs have to rely on direct communications with neighbours when their uptimes overlap. Figure 6.2 shows $\overline{e}$ and $\overline{eDynamic}$ results, according to the network and service topologies. Figures 6.2a and 6.2b show results for 9 ONs. Figures 6.2c and 6.2d show results according to the number of ONs. As a reminder, network topologies and best/worst service topologies are depicted in Figure 6.1. The *aggregator* and *observer's* reconfigurations are described in Section 5.

Impact of network and service topologies Figures 6.2a and 6.2b show the impact of network and service topologies on the total and dynamic energy consumption for 9 ONs. In *clique*, due to ONs' maximum degree (i.e., number of neighbours), requests propagate the fastest. Request propagation doesn't significantly increase energy consumption due to the small size of requests. This allows *clique* to consume the least amount of energy among all network topologies. $\overline{e}$ and $\overline{eDynamic}$ values are 28.46 kJ and 286.66 J respectively. The best and worst service topologies of *clique* have the same results, due to homogeneous ONs' degree and hop count.

In *star*, requests propagation is more limited compared to *clique*. In the best service topology, the *aggregator* has all *observers* as neighbours. In this case, $\overline{e}$ and $\overline{eDynamic}$

values are 67.35 kJ and 466.07 J, respectively. In the worst service topology, the *aggregator* only has the central *observer* as neighbour. In this case, $\overline{e}$ and $\overline{eDynamic}$ values almost double, and reach 144.15 kJ and 831.55 J respectively.

In the best service topology of *grid*, the *aggregator* has a lower degree and increased maximum hop count compared to *star*. $\overline{e}$ and $\overline{eDynamic}$ values are 90.12 kJ and 581.21 J, respectively, which are higher than the best service topology of *star*. For worst service topology, *grid*'s results are slightly lower than *star*'s. In this case, $\overline{e}$ and $\overline{eDynamic}$ values reach 128.52 kJ, and 761.69 J, respectively.

Ring's ONs have a much higher maximum hop count compared to other topologies. Requests are forced to propagate along the ring. *Ring* consumes more energy than *clique*, *star*, and *grid*, for best and worst service topologies. $\overline{e}$ and $\overline{eDynamic}$ values reach 215.00 kJ and 1176.85 J, respectively. As for *clique*, the best and worst service topologies of *ring* have the same results, due to homogeneous ONs' degree and hop count.

The best position of *chain* has the same degree and maximum hop count as *ring*. However, in *chain*, requests cannot propagate from one side to the other. For the best service topology, on average, this makes *chain* consume more energy than *ring*. $\overline{e}$ and $\overline{eDynamic}$ values are 265.90 kJ and 1427.78 J, respectively. For worst service topology, maximum hop count strongly increases, forcing some requests to travel through the whole chain. In this case, the energy consumption strongly increases and reaches 485.79 kJ for $\overline{e}$ and 2464.03 J for $\overline{eDynamic}$.

Scalability impact on energy consumption Figures 6.2c and 6.2d show the evolution of the total and dynamic energy consumption when the number of ONs increases. As depicted on the figures, from 9 to 25 ONs, $\overline{e}$ and $\overline{eDynamic}$ follow a similar evolution.

Clique's $\overline{e}$ and $\overline{eDynamic}$ values increase the slowest according to cluster size, as ONs' maximum degree is conserved when *clique*'s size increases. For 25 ONs, $\overline{e}$ and $\overline{eDynamic}$ values are 46.53 kJ and 682.87 J, respectively. It is followed by *star* and *grid*. *Star*'s results for best service topology grow slower than *grid*'s, as the central ON's maximum degree is also conserved when adding more branches to *star*. For 25 ONs, $\overline{e}$ and $\overline{eDynamic}$ are respectively 269.21 kJ and 1704.57 J for *star*, and 433.12 kJ and 2536.08 J for *grid*. For worst service topology, the evolution of *grid*'s and *star*'s energy consumption is closer. $\overline{e}$ and $\overline{eDynamic}$ are respectively 517.11 kJ and 2891.00 J for *star*, and 639.58 kJ and 3518.97 J for *grid*.

Ring's and *chain*'s energy consumption increase much faster according to cluster size, as adding more ONs linearly increases maximum hop count in both topologies. Considering the best service topology, *chain* grows slightly faster than *ring*. For 25 ONs, $\overline{e}$ and $\overline{eDynamic}$ values are respectively 1991.04 kJ and 9996.62 J for *ring*, and 2265.03 kJ and 11417.57 J for *chain*. However, considering the worst service topology, *chain* grows much faster than *ring*. In this case, $\overline{e}$ and $\overline{eDynamic}$ values are respectively 4227.51 kJ and 20686.73 J.

6.5.2 Coordination with an RN

In the following, an RN is present and wakes up at the same time as its neighbours. This eases communication, but implies additional RN's uptimes. Table 6.6 shows $\overline{e}$ and $\overline{eDynamic}$ results for 9 ONs. The energy consumption is depicted according to the network topology, service topology, and RN's position. Energy consumption variations between scenarios without an RN and scenarios with an RN are depicted. Figure 6.3 shows results for each topology according to the number of ONs, for best and worst RN's

Table 6.6: $\overline{e}$ and $\overline{eDynamic}$ results for 9 ONs, according to the network topology, service topology and RN's position. rn_{NA}, rn_{best} and rn_{worst} represent, respectively, scenarios without an RN, with an RN in the best position and with an RN in the worst position. For scenarios with best and worst RN's positions, $\%\Delta$ computes the energy consumption variation compared to scenarios without an RN.

$\overline{e}$ (kJ)					
Service topology	best				
Network topology	clique	star	grid	ring	chain
rn_{NA}	28.46 (7.33)	67.35 (26.61)	90.12 (24.29)	215.0 (49.18)	265.9 (63.42)
rn_{best}	1.37 (0.05)	1.37 (0.05)	39.78 (19.09)	192.62 (49.93)	239.73 (71.19)
rn_{worst}	4.07 (0.79)	71.47 (30.25)	93.93 (29.4)	195.31 (50.76)	268.43 (77.5)
$\%\Delta(rn_{NA}, rn_{best})$	**-95%**	**-98%**	**-56%**	**-10%**	**-10%**
$\%\Delta(rn_{NA}, rn_{worst})$	**-86%**	**6%**	**4%**	**-9%**	**1%**
$\overline{e}$ (kJ)					
Service topology	worst				
Network topology	clique	star	grid	ring	chain
rn_{NA}	28.46 (7.33)	144.15 (48.31)	128.52 (30.7)	215.0 (49.18)	485.79 (97.9)
rn_{best}	4.07 (0.79)	4.05 (0.71)	88.72 (35.78)	195.31 (50.76)	431.95 (96.38)
rn_{worst}	1.37 (0.05)	72.25 (30.42)	104.25 (33.39)	192.62 (49.93)	470.75 (98.09)
$\%\Delta(rn_{NA}, rn_{best})$	**-86%**	**-97%**	**-31%**	**-9%**	**-11%**
$\%\Delta(rn_{NA}, rn_{worst})$	**-95%**	**-50%**	**-19%**	**-10%**	**-3%**
$\overline{eDynamic}$ (J)					
Service topology	best				
Network topology	clique	star	grid	ring	chain
rn_{NA}	286.66 (35.91)	466.07 (128.61)	581.21 (117.3)	1176.85 (234.54)	1427.78 (304.81)
rn_{best}	150.67 (0.31)	150.34 (0.29)	445.83 (147.18)	1390.37 (322.23)	1699.97 (461.69)
rn_{worst}	169.85 (5.54)	551.02 (173.8)	741.87 (190.55)	1273.05 (280.99)	1672.24 (436.73)
$\%\Delta(rn_{NA}, rn_{best})$	**-47%**	**-68%**	**-23%**	**18%**	**19%**
$\%\Delta(rn_{NA}, rn_{worst})$	**-41%**	**18%**	**28%**	**8%**	**17%**
$\overline{eDynamic}$ (J)					
Service topology	worst				
Network topology	clique	star	grid	ring	chain
rn_{NA}	286.66 (35.91)	831.55 (229.8)	761.69 (145.13)	1176.85 (234.54)	2464.03 (469.51)
rn_{best}	169.85 (5.54)	168.8 (5.22)	733.31 (259.64)	1273.05 (280.99)	2645.47 (585.01)
rn_{worst}	150.67 (0.31)	555.53 (174.76)	820.54 (217.2)	1390.37 (322.23)	2828.63 (560.72)
$\%\Delta(rn_{NA}, rn_{best})$	**-41%**	**-80%**	**-4%**	**8%**	**7%**
$\%\Delta(rn_{NA}, rn_{worst})$	**-47%**	**-33%**	**8%**	**18%**	**15%**

positions and service topologies. As stated in Section 6.2, the RN can be either the *aggregator* or an *observer*. The RN is the *aggregator* when both share the same position (best or worst). Otherwise, the RN is an *observer*.

Impact of the RN's position and service topology on energy consumption Table 6.6 shows the impact of RN's position and service topology on total and dynamic energy consumption. On *clique*, the RN is synchronised with all ONs, due to ONs' maximum degree. This allows drastic energy consumption reduction. When the RN is the *aggregator*, it can consistently coordinate with each *observer* within the first hour. In this case, $\overline{e}$ and $\overline{eDynamic}$ are reduced by 95% and 47%, respectively. Results' consistency is depicted by the small standard deviation. When the RN is an *observer*, the *aggregator* can coordinate with only part of the *observers* within the first hour. It needs a second hour to coordinate with the other part. In this case, $\overline{e}$ and $\overline{eDynamic}$ are reduced by 86% and 41%. Results are also slightly more inconsistent, as shown by the slightly bigger standard deviations.

On *star*, the impact of the service topology on energy consumption is largely overshadowed by the RN. The best position of *star* has maximum degree. Using an RN on the best position allows *star* to consistently have the same energy consumption values as *clique*, for the same parameters. For the best service topology, $\overline{e}$ and $\overline{eDynamic}$ are reduced by 98% and 68% respectively. For the worst service topology, $\overline{e}$ and $\overline{eDynamic}$ are reduced by 97% and 80%. In the worst position, the RN is only synchronised with the central ON, reducing its impact. For best service topology, the RN is an *observer*. Using an RN only adds an energy consumption overhead: $\overline{e}$ and $\overline{eDynamic}$ increase by 6% and 18% respectively. For the worst service topology, the RN is the *aggregator*. This helps communications between the *aggregator* and central ON. $\overline{e}$ and $\overline{eDynamic}$ are reduced by 50% and 33% respectively.

On the best position of *grid*, the RN is synchronised with fewer ONs than *clique* and *star*, due to a lower degree. This position still allows decent energy consumption reduction. The service topology also has more impact on energy consumption than *clique* and *star*. For best service topology, using an RN allows to reduce $\overline{e}$ and $\overline{eDynamic}$ by 56% and 23%, respectively. For worst service topology, $\overline{e}$ is reduced by 31%, and $\overline{eDynamic}$ is reduced by 4%. In the worst position, the RN's degree decreases, reducing its impact. For the worst service topology, $\overline{e}$ decreases by 19%, but $\overline{eDynamic}$ increases by 8%. For the best service topology, $\overline{e}$ and $\overline{eDynamic}$ both increase by 4% and 28%.

On *ring*, the RN's benefits are limited by ONs' low degree and high maximum hop count. The RN is synchronised only with the previous and next ON of *ring*. It still allows to decrease $\overline{e}$, but $\overline{eDynamic}$ increases in all cases. When the RN is the *aggregator*, $\overline{e}$ is reduced by 10%, but $\overline{eDynamic}$ increases by 18%. When the RN is an *observer*, $\overline{e}$ is reduced by 9%, but $\overline{eDynamic}$ increases by 8%. Note that, when the RN is the *aggregator*, $\overline{eDynamic}$ increases more than when the RN is an *observer* (18% against 8%). The *aggregator* needs to communicate with every ON, while *observers* need to communicate only with the *aggregator*. For this reason, when the RN is the *aggregator*, $\overline{eDynamic}$ increases more due to communications than when the RN is an *observer*.

On *chain*, the impact of the RN is also limited and heavily overshadowed by the impact of the service topology. On the best position, the RN is synchronised with the previous and next ON. For best service topology, it has a similar impact as for *ring*: $\overline{e}$ is reduced by 10%, while $\overline{eDynamic}$ increases by 19%. For worst service topology, $\overline{e}$ decreases by 11%, while $\overline{eDynamic}$ increases by 7%. In the worst position, the RN is only synchronised with the next ON. For best service topology, $\overline{e}$ slightly increases by 1%, and $\overline{eDynamic}$ increases by 17%. For worst service topology, $\overline{e}$ is reduced by 3%, while $\overline{eDynamic}$ increases by 15%.

Scalability impact on energy consumption variation Figure 6.3 shows the evolution of the total and dynamic energy consumption when the number of ONs increases. Additional ONs added to *clique* are synchronised with the RN. The benefits of the RN remain when *clique*'s size increases, for best and worst service topologies. RN's maximum degree allows $\overline{e}$ to grow very slowly compared to scenarios without RN. $\overline{eDynamic}$ grows at a close rate compared to scenarios without RN, due to the energy consumed for the execution of reconfiguration actions.

Similarly to *clique*, in the best position, additional branches added to *star* are also synchronised with the RN. For best and worst service topologies, *star*'s $\overline{e}$ and $\overline{eDynamic}$ grow at relatively the same rate as *clique*'s. In the worst position, $\overline{e}$ and $\overline{eDynamic}$ grow much faster.

When *grid*'s size increases, the service topology has slightly more impact than RN's

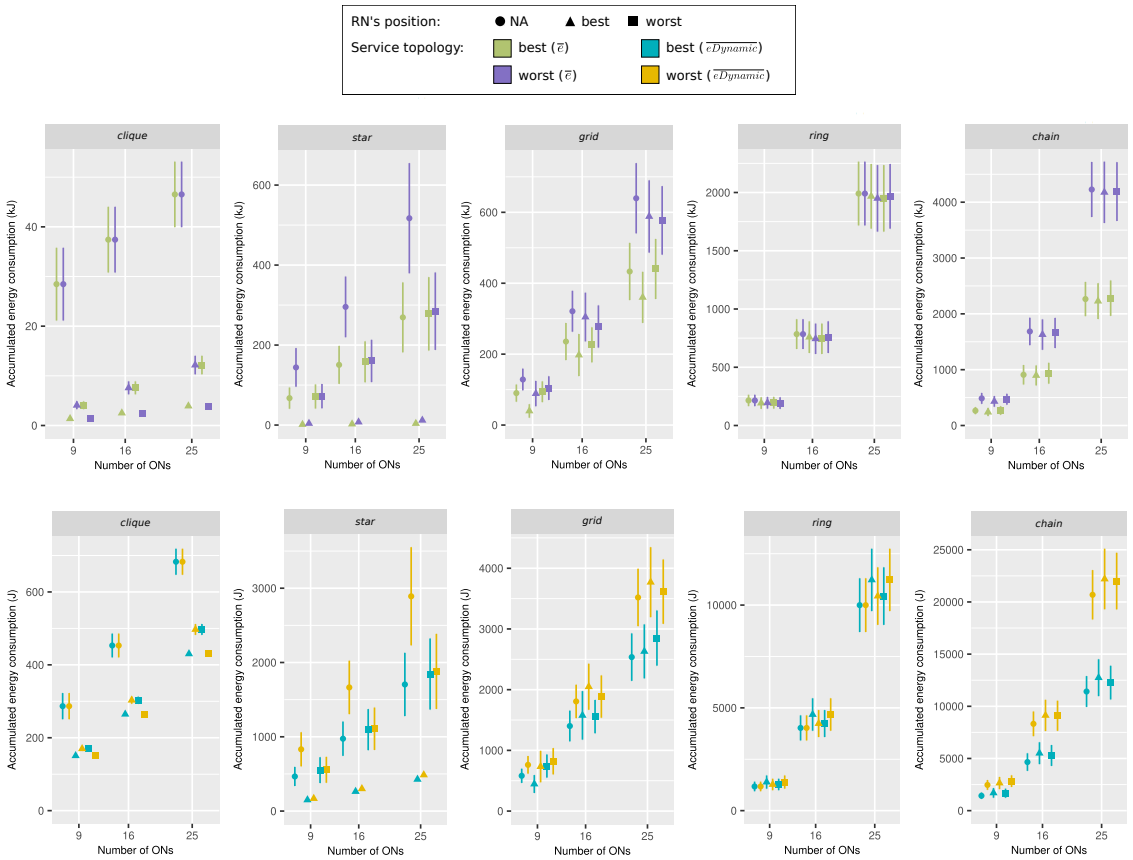


Figure 6.3: Comparison between scenarios without using RN and scenarios using an RN, according to the RN's position and the number of ONs.

position. Figures show a faster evolution of $\overline{e}$ and $\overline{eDynamic}$ values for worst service topology compared to best. With more ONs on *grid*, RN's degree remains and its maximum hop count increases. While $\overline{e}$ decreases in most cases when using an RN, $\overline{eDynamic}$ only increases, especially when the RN is in the best position. In the best position, the RN has a higher degree, thus wakes up more often and exchanges more data with neighbours, compared to the worst position.

Finally, for *ring* and *chain*, RN's energy consumption reduction stays relatively stable. $\overline{eDynamic}$ increases in most cases while $\overline{e}$ is slightly reduced.

6.5.3 Discussion

As for the first experiment, the dynamic energy consumption represents only a fraction of the total, due to the long coordination duration. Thus, doing coordination alongside observation activities is a better option in this chapter.

Clique is the most favourable network topology in terms of energy consumption, under any scenario. Most of the time, *star* consumes less than *grid*, especially for a large number of ONs. *Ring* and *chain* consumes the most energy.

The impact of service topology varies across network topologies. On homogeneous network topologies like *clique* and *ring*, the coordination's energy consumption is not sensitive to the service topology. For other network topologies, the service topology has a significant impact. For *star* and *chain*, on average, the worst service topology almost doubles the energy consumed by ONs. For *grid*, on average the worst service topology

also strongly increases energy consumption.

Using an RN with a high degree can significantly reduce energy consumption. On *clique*, and the central ON of *star* and *grid*, using an RN drastically decreases energy consumption. With a low degree, or when the maximum hop count is high, the benefits of the RN are more limited. In some cases, the impact of the service topology is stronger than the impact of the RN. This is notably the case for *grid* and *chain*, especially when the number of ONs increases. Due to more frequent RN's uptimes compared to ONs, the coordination's energy consumption may increase. For *ring* and *chain*, using an RN slightly decreases the total energy consumption, but increases the dynamic energy consumption in all cases. For *star* and *grid*, having the RN in the worst position increases energy consumption when the service topology is best. When service topology is worst, having an RN can help to reduce both total and dynamic energy consumption altogether. This is especially the case for *star*.

Results show a clear distinction between the energy consumption of the different network topologies. However, not all topologies are suited for all contexts. For scenarios where small areas need to be covered, *clique* has better energy consumption results than *star*. For scenarios where large areas need to be covered, *grid*, *ring* and *chain* may be better choices compared to *clique* and *star*, to the cost of more energy consumption by the ONs. *Grid's* energy consumption is at the same order of magnitude as *star*. This offers an interesting trade-off between the ability to cover large areas and the energy consumption due to coordination. For scenarios where *ring* and *chain* topologies are imposed, *ring* may be preferable when service topology is worst. For best service topology, *chain* and *ring* may both be used, as *chain* has similar energy consumption results as *ring*.

6.6 Limitations

The work of this section is based on different assumptions. This subsection discusses limitations due to these assumptions and potential future works to handle them.

Relay Node's assumptions In this manuscript, the RN has sufficient energy, accuracy and knowledge about neighbours' uptimes to be available to its neighbours during the whole reconfiguration duration. These properties may be hard to consistently achieve in constrained environments such as the Arctic Tundra. Complementary work can be done using relaxed assumptions to study a less optimal RN: (i) having partial knowledge about neighbours' uptimes: for example, an RN could know or work for only a subset of neighbours, or a subset of uptimes per neighbour. (ii) Building knowledge about neighbours' uptimes: for instance, the RN could get and send hints about next uptimes of ONs and itself before going back to sleep. (iii) Including the possibility of clock drift: the RN may miss some of its neighbours' uptimes due to clock drift leading to shorter or missed overlaps with its neighbours. Alternatively, the RN may wake up for longer duration to compensate for any potential drift, leading to additional energy consumption. In all previously proposed weaker assumptions, energy consumption due to knowledge building should be evaluated and taken into account.

Coarse-grained simulation results Energy consumption during simulations is calibrated using extreme values measured on real hardware (Raspberry Pi). This includes energy consumption due to the execution of reconfiguration actions. In real scenarios,

execution of reconfiguration actions may consume less energy than these values. Calibrating the simulations using energy consumption of real reconfiguration actions could refine energy consumption results by making them closer to reality.

Additionally, size, frequency, and transmission of messages between ONs during simulation are not optimised. ONs send messages every second and disseminate these messages in the CPS using an epidemic-like dissemination pattern. Additional work could be done to optimise energy consumption due to communication.

Crash and failures In the simulations, ONs are assumed to have no crash nor failure. For each simulation, the reconfiguration is able to terminate with no error. This may also be hard to consistently achieve in the Arctic Tundra. An ON's failure during reconfiguration may put the system in an inconsistent state. Such failure must be detected and appropriate correction should be applied. Additionally, different strategies could be studied to mitigate the impact of such failure (for instance using loosely coupled services to decouple ONs, or tolerate certain types of failure).

Note that simulating crash and failure can be easily done using the same setup as experiments done in this chapter. A crash can be simulated as an ON that stops waking up until the end of the simulation. A failure can be simulated as an ON that goes to sleep during the execution of a reconfiguration action.

6.7 Conclusion

In this chapter, the energy consumption when coordinating reconfiguration actions between ONs according to different scenarios is studied. Two coordination cases (*deploy* and *update*) and plausible parameters (number of ONs, uptime duration, radio technology, network topology, service topology, and usage of Relay Node) are simulated.

Uptime schedules are generated to simulate ONs' sleeping behaviours. To get a representative set of results, each scenario is run over 200 uptime schedules. For each scenario, an average of the energy consumption is given and discussed.

Results show high energy consumption variations. Increasing uptime duration can reduce energy consumption due to faster coordination convergence, up to 12% of the total energy. Due to longer ONs' uptimes, the dynamic energy consumption may also increase, up to 31% in worst scenarios. LoRa is always a better choice than NB-IoT due to small packet sizes. In most favourable network topologies, ONs consume up to 98% less energy than others (i.e., clique compared to chain). The best service topology decreases energy consumption compared to the worst, up to 53%. Using an RN can drastically reduce energy consumption with a high number of neighbours, up to 98%. Due to additional RN's uptimes, energy consumption may also increase, up to 28% in worst scenarios (particularly when the RN has few neighbours). In scenarios where the RN has a lot of neighbours, the impact of the RN overshadows the impact of the service topology. Finally, it is shown that it is beneficial to do reconfiguration actions while overlapping with existing uptimes (i.e., reserved for observation activities). Results show high energy consumption in scenarios where ONs wake up specifically to *deploy*/*update*.

Chapter 7

Conclusion and perspectives

7.1 Conclusion

The DAO project proposes a CPS composed of Observation Nodes (ON) deployed in the Arctic Tundra to perform in-situ and long-term observations. The Arctic Tundra is a scarce-resource environment which imposes extreme constraints on the CPS, such as lack of energy supply and network coverage. ONs deployed on the field have to carefully manage a limited energy budget. The low sun exposition and absence of infrastructure to harvest energy such as wind makes this energy budget non-rechargeable. To save energy and extend their lifetimes, ONs enter long sleeping periods and short uptime periods. Due to lack of network infrastructure, ONs have to rely on peer-to-peer network connections. Non-synchronised uptime periods between ONs result in very few opportunities for ONs to communicate.

Due to harsh weather conditions, physical access to observation sites is prevented for 6 months during winter. This forces nodes to be autonomous for extended periods. Providing reconfiguration capabilities to such ONs can improve their autonomy. ONs should be able to automatically adapt to external events by reconfiguring their system (e.g., update software, activate/deactivate service, etc.). Collaboration between ONs is enabled to allow heterogeneous ONs to benefit from the resources and capabilities of each other. Due to such collaboration, reconfiguration should be coordinated between ONs to prevent failure. To this end, ONs should be able to coordinate the execution of reconfiguration with other ONs (i.e., decentralised reconfiguration).

Due to scarce connectivity between nodes, this manuscript also includes a study of a potential leverage to alleviate the difficulty of nodes to communicate. This leverage is a special type of node that uses its energy supply to help other nodes to communicate. This node, called Relay Node, is considered to have sufficient energy supply to be awake more frequently and for longer duration than the other nodes. This capability allows the Relay Node to be awake at the same time as its neighbouring nodes. It can be used as an intermediary to store messages from senders while recipients are sleeping, and forward messages to recipients when they wake up. Such node may not be available in all observation sites. In this manuscript, the performances of reconfiguration in terms of energy consumption and duration are studied with and without the Relay Node.

Decentralised reconfiguration between sleeping ONs can impose high energy consumption overhead on ONs and take a long time to converge. The impact of sleeping node on decentralised reconfiguration duration and energy consumption hasn't yet been studied in the literature. This manuscript provides six contributions aiming at providing a decentralised reconfiguration solution, and results about the decentralised reconfiguration

duration and energy consumption.

The first contribution is the design and implementation of Concerto-D, a solution to model and execute decentralised reconfiguration across sleeping ONs. It is based on Concerto, a centralised reconfiguration solution. Concerto-D inherits the properties of Concerto, such as genericity of module operations and high level of parallelism. The decentralisation of Concerto has been done by extending the semantics of Concerto actions, to enable the expression of dependencies between multiple reconfiguration programs, and between modules defined on different Concerto-D instances. The execution of a reconfiguration program using Concerto-D can be suspended and resumed, making Concerto-D suitable to be hosted on sleeping ONs. Finally, a Concerto-D instance is able to communicate with other instances, using client/server or publish/subscribe paradigms.

The second contribution is the validation of the decentralisation and high level of parallelism capabilities of the Concerto-D prototype on two use cases. The first use case is the deployment and update of an OpenStack cluster in the Cloud. This use case allows to validate both the decentralisation and high level of parallelism capabilities of Concerto-D. In this use case, Concerto-D successfully executes and coordinates decentralised reconfiguration programs across different Concerto-D instances. The reconfiguration duration using Concerto-D has been compared to the duration using another solution from the literature. It is shown that the execution of decentralised reconfiguration is faster using Concerto-D than the other solution, thanks to the high level of parallelism of Concerto-D (between 24% and 42% faster on the OpenStack use case). The second use case is the decentralised reconfiguration of modules deployed on sleeping ONs. Concerto-D programs could be successfully executed on multiple sleeping ONs, by temporarily suspending and saving the state of the execution on disk when the ON goes to sleep. Reconfiguration programs have been executed either client/server or publish/subscribe communications, which validates the capability of Concerto-D of using either of these paradigms for communication.

The third contribution aims at studying the impact of non-synchronised sleeping ONs on reconfiguration duration. Experiments have been conducted for different degrees of connectivity between ONs (using the number of overlaps between ONs' uptimes) and uptime orders between ONs. Experiments have been conducted with and without using a Relay Node. Results show that having a low number of overlaps significantly increases the reconfiguration duration when the Relay Node is not present. When varying the number of overlaps from 7 to 30, the reconfiguration duration increases by up to 869%. When ONs have access to a Relay Node, the uptime order has a high impact on reconfiguration duration (up to 224% variation).

The fourth contribution of this manuscript is the study of a trade-off between uptime duration and reconfiguration duration when using a Relay Node. This trade-off has been studied in the context of sleeping ONs for different overlap rates between ONs. During a decentralised reconfiguration involving sleeping ONs, ONs spend a large portion of time waiting to coordinate with another ON. This study shows that, making ONs sleep as soon as possible (i.e., reducing the waiting duration to the maximum) reduces uptime duration from 71% to 88% for high and low overlap rates. The trade-off with reconfiguration duration varies according to the overlap rate. When the overlap rate is high (e.g., 50-60%), making ONs sleeping as soon as possible can significantly increase reconfiguration duration (up to 218%). When the overlap rate is low (e.g., 2-5%), making ONs sleeping as soon as possible doesn't increase the reconfiguration by a lot (e.g., by 13%).

The fifth and sixth contributions are the evaluation and study of the impact of sleep-

ing ONs according to different parameters on reconfiguration's energy consumption and duration. These parameters are the number of ONs, uptime duration, radio technology, usage of a Relay Node for communication, network topology, and service topology. Simulations have been used to conduct this study. The implemented simulator uses validated energy and communication models to simulate different network topologies and estimate energy consumption due to the activities simulated on the system. Using this simulator, uptime, sleeping periods, and reconfiguration activities can be simulated. Results show that having a higher uptime duration decreases the reconfiguration duration but may increase its energy consumption due to reconfiguration activities (up to 31% increase in dynamic energy). LoRa has better energy consumption results than NB-IoT under any scenario. Network topologies with a high number of neighbours and low hop count between ONs allow to decrease energy consumption and duration compared to topologies with low number of neighbours and high hop count between ONs (up to 98% for network topology). The service topology can also have a large impact on energy consumption, and reduce the energy consumption by up to 53% for the best service topology compared to the worst. The Relay Node impact on energy consumption decreases in topologies and positions where it has few neighbours and high hop count. However, when it is neighbour to all other nodes, it can reduce energy consumption by up to 97%.

7.2 Discussion

This section puts in perspective the work done in this PhD. Particularly, the benefits of high level of parallelism given by Concerto-D.

Benefits of high level of parallelism in high latency environments The choice of Concerto as the basis for the design of Concerto-D was motivated by its high level of parallelism and genericity in terms of module operations. Concerto-D inherits *intra* and *inter-module* parallelism capabilities of Concerto, which are higher level of parallelism compared to *module*. As a reminder, *intra-module* parallelism allows parallelism at the scope of transitions within the life-cycle of the same module. *Inter-module* parallelism allows parallelism at the scope of transitions between different modules.

The *intra-module* parallelism could theoretically be beneficial both for energy consumption and reconfiguration duration. This type of parallelism allows local reconfiguration to be faster, which can be valuable in environments such as the Arctic Tundra (i.e., going back to sleep quicker, reach a configuration to treat a short-lived event, etc.).

The *inter-module* parallelism may have the same benefit as *intra-module* parallelism (i.e., quicker reconfiguration) when modules are defined in the same managing system. In such case, the reconfiguration can also be faster due to the ability to execute transitions on different modules in parallel. When modules are defined on different managing systems hosted on different nodes (i.e., remote modules), synchronisations over the network are required to resolve dependencies between remote modules. These synchronisations have a limited impact in a context of low latency between nodes (as in the OpenStack use case of Section 4.4). However, when latency between nodes is very high, such as in the DAO-CPS, such synchronisations can impose a significant overhead on the energy consumption and duration of the reconfiguration. In such case, the waiting duration due to dependency resolution could overshadow the benefits provided by *inter-module* parallelism. Therefore, increasing the level of parallelism between remote modules using *inter-module* parallelism in high-latency environments could be counterproductive and potentially increase the

reconfiguration duration and energy consumption instead of decreasing them. The study of the impact of *inter-module* parallelism between remote modules on reconfiguration duration according to latency between nodes could be the subject of a dedicated study.

Coordination involving large number of sleeping nodes Coordination involving non-synchronised sleeping nodes such as ONs can reach very high energy consumption and duration. While the individual transitions have a duration of seconds, the coordination of the reconfiguration can last from hours at best to a whole month at worst. When a Relay Node is available to few or no ONs, the duration of a decentralised reconfiguration can last for weeks. Even when considering scenarios with few ONs (e.g., 6), the reconfiguration duration can last for days. On the long term, this may impose a high energy consumption due to communication between nodes, and uptime spent waiting for other nodes.

For these reasons, in the context of scarce connectivity between nodes, having lots of nodes involved in a coordination may not be a good design choice when fast and energy-efficient reconfiguration needs to be considered. Having the ability to reconfigure modules independently from others could be significantly helpful for the energy consumption and the ability of nodes to reconfigure fast.

7.3 Perspectives

The following describes research perspectives to extend contributions of this manuscript.

Concerto-D impact on energy consumption In the experiments of this manuscript, the managing and managed system are co-hosted on the same node. Each Concerto-D instance (managing system) is responsible for the reconfiguration of its local system (managed system). The energy consumption overhead due to the execution of module operations and communication with other Concerto-D instances has been evaluated and studied in Chapter 6. However, the execution of the Concerto-D instance itself can also have an energy consumption overhead which has not been studied in this manuscript (e.g., impact of reconfiguration actions, saving and restoring the state, launching a Flask server and answering requests, etc.).

Design of a low-power decentralised reconfiguration As seen in results, energy consumption due to reconfiguration can reach very high values (up to several weeks of energy consumption). These values are notably related to communications. Nodes try to pull data every second whenever a synchronisation is needed. This is highly inefficient due to the high amount of failed attempts to pull data. Data are propagated in the network in an epidemic manner. This leads to lots of redundant transmissions. Reducing the amount of transmission and redundancy could significantly improve energy consumption. Additionally, when lots of synchronisations are required, reconfiguration programs could be significantly delayed.

Current Concerto-D prototype is a fork of Concerto prototype implemented in CPython. Compared to other Python implementations or lower level languages, CPython may impose an overhead in resource consumption at runtime, due to its abstraction of memory management, dynamic typing, etc. Such an overhead hasn't been studied in this manuscript, but could be the subject of future works. For instance, experiments using lower level languages or implementations than CPython could highlight the different performance improvements the current Concerto-D implementation could have. The

DAO-CPS can also feature devices such as microcontrollers. Implementation of Python such as MicroPython [1] could be relevant to make Concerto-D deployable on such nodes.

Prediction of energy consumption and duration of a reconfiguration Nodes deployed in environments such as the Arctic Tundra have to manage a limited energy budget. Providing nodes the ability to predict resource usage and duration of a specific reconfiguration can be useful. As seen in the experiments, coordinating a reconfiguration can impose a high energy consumption overhead on nodes and last for a long time. The prediction of the energy consumption and duration of a reconfiguration could help nodes to not engage in a reconfiguration that is not affordable in terms of energy, or would be too long to provide sufficient value to the system (e.g., necessity to treat an event urgently).

Concerto-D inherits an interesting feature from Concerto, which is a dependency graph. When given a reconfiguration program, Concerto is able to compute a dependency graph based on this program. Each edge of the graph is an action to be performed by Concerto, including the execution of transitions triggered by module operations (via pushB). The execution of each individual transition can be assigned a duration. Based on this, the expected duration of a given reconfiguration program can be estimated. Such graph could be extended to include new Concerto-D's language semantics by including remote dependencies. The duration of these dependencies could also be used as input of the graph and could be heavily impacted by latency between nodes (as seen in the experiments of this manuscript).

Additionally, the new performance graph of Concerto-D could also be extended to include energy consumption values besides duration. Such an energy consumption could be measured individually for each action or transition before deployment. Values obtained from measurements could be used as input of the performance graph to estimate the energy consumption of a reconfiguration program. This approach is challenging and may lack accuracy, as the energy consumption of a same action or transition may vary according to different factors (e.g., parallel execution, varying workload imposed on the node, etc.). Another technique is the usage of software-defined power monitoring systems. These systems can be hosted besides each Concerto-D instance and actively monitor its activity to provide estimation of its energy consumption in real time. Some power monitoring systems can improve their accuracy over time by recalibrating their estimations according to the activity of the monitored system (i.e., Concerto-D). However, the activity of these power monitoring systems can also have an energy consumption overhead on the system, which should also be taken into account in the total energy consumption of the managing system [72, 73].

Energy consumption impact and duration of remote dependency resolution could also be challenging to measure. Lots of different factors need to be taken into account. Non-synchronised uptime schedules add large unpredictability regarding the time at which a remote dependency can be resolved. Size of exchanged data varies depending on the data required by the provide/use ports. Different radio technologies have different bandwidth and energy consumption. Finally, the network and service topology may impact the number of retransmissions and redundancies.

[1] `https://micropython.org/`

List of Figures

List of Tables

Listings

Bibliography

[1] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: the next computing revolution," in *Proceedings of the 47th design automation conference*, 2010, pp. 731–736.

[2] J. Shi, J. Wan, H. Yan, and H. Suo, "A survey of cyber-physical systems," in *2011 international conference on wireless communications and signal processing (WCSP)*. IEEE, 2011, pp. 1–6.

[3] 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.

[4] S. Naderiparizi, Z. Kapetanovic, and J. R. Smith, "Wispcam: An rf-powered smart camera for machine vision applications," in *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ser. ENSsys '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 19–22. [Online]. Available: https://doi.org/10.1145/2996884.2996888

[5] Å. Ø. Pedersen, J. Stien, S. Albon, E. Fuglei, K. Isaksen, G. Liston, J. U. Jepsen, J. Madsen, V. T. Ravolainen, A. K. Reinking *et al.*, "Climate-ecological observatory for arctic tundra (coat)," *State of Environmental Science in Svalbard (SESS) Report 2019*, pp. 58–83, 2020.

[6] M. J. Murphy, O. Tveito, E. F. Kleiven, I. Raïs, E. M. Soininen, J. M. Bjørndalen, and O. Anshus, "Experiences Building and Deploying Wireless Sensor Nodes for the Arctic Tundra," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2021, pp. 376–385.

[7] I. Raïs, O. Anshus, J. M. Bjørndalen, D. Balouek-Thomert, and M. Parashar, "Trading data size and CNN confidence score for energy efficient CPS node communications," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pp. 469–478.

[8] I. Rais, L. Guegan, and O. Anshus, "Impact of loosely coupled data dissemination policies for resource challenged environments," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. Taormina, Italy: IEEE, May 2022, pp. 524–533. [Online]. Available: https://ieeexplore.ieee.org/document/9826006/

[9] R. Ben Halima, M. Hachicha, A. Jemal, and A. Hadj Kacem, "Mape-k patterns for self-adaptation in cyber-physical systems," *The Journal of Supercomputing*, vol. 79, no. 5, pp. 4917–4943, 2023.

[10] K. Morris, *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.", 2016.

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

[12] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab *et al.*, "Grid'5000: A large scale and highly reconfigurable experimental grid testbed," *The International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[13] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design.* pearson education, 2005.

[14] M. Chardet, "Reconciling parallelism expressivity and separation of concerns in reconfiguration of distributed systems," Ph.D. dissertation, Ecole nationale supérieure Mines-Télécom Atlantique, 2020.

[15] P. Favali and L. Beranzoli, "Seafloor observatory science: A review," *Annals of geophysic*, 2006.

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

[17] R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," in *Proceedings first international conference on peer-to-peer computing.* IEEE, 2001, pp. 101–102.

[18] R. Shanmugapriya and S. V. N. Santhosh Kumar, "Comprehensive survey on data dissemination protocols for efficient reprogramming in internet of things," *Concurrency and computation: practice and experience*, vol. 34, no. 26, p. e7280, 2022.

[19] J. Kephart and D. Chess, "The vision of autonomic computing," vol. 36, no. 1, pp. 41–50.

[20] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka, "On Patterns for Decentralized Control in Self-Adaptive Systems," in *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, ser. Lecture Notes in Computer Science, R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds. Berlin, Heidelberg: Springer, 2013, pp. 76–107. [Online]. Available: https://doi.org/10.1007/978-3-642-35813-5_4

[21] S. Muralidharan, G. Song, and H. Ko, "Monitoring and managing iot applications in smart cities using kubernetes," *Cloud Computing*, vol. 11, 2019.

[22] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with KubeEdge," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 373–377.

[23] L. L. Jiménez and O. Schelén, "DOCMA: A Decentralized Orchestrator for Containerized Microservice Applications," in *2019 IEEE Cloud Summit*, Aug. 2019, pp. 45–51.

[24] A. Alidra, H. Bruneliere, H. Coullon, T. Ledoux, C. Prud'Homme, J. Lejeune, P. Sens, J. Sopena, and J. Rivalan, "Semafor-self-management of fog resources with collaborative decentralized controllers," in *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2023, pp. 25–31.

[25] I. Gerostathopoulos, D. Skoda, F. Plasil, T. Bures, and A. Knauss, "Tuning self-adaptation in cyber-physical systems through architectural homeostasis," *Journal of Systems and Software*, vol. 148, pp. 37–55, 2019.

[26] 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.

[27] F. Qu, Z. Wang, H. Song, Y. Chen, and L. Yang, "A study on a cabled seafloor observatory," *IEEE Intelligent Systems*, vol. 30, no. 1, pp. 66–69, 2015.

[28] 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.

[29] Z. Zhang, S. Glaser, T. Watteyne, and S. Malek, "Long-Term Monitoring of the Sierra Nevada Snowpack Using Wireless Sensor Networks," *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 17 185–17 193, Sep. 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8977506

[30] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, 2002, pp. 88–97.

[31] C. Szyperski, D. Gruntz, and S. Murer, *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[32] I. Rais, "Discover, model and combine energy leverages for large scale energy efficient infrastructures— theses. fr," Ph.D. dissertation, Lyon, 2018.

[33] R. S. Sinha, Y. Wei, and S.-H. Hwang, "A survey on LPWA technology: LoRa and NB-IoT," *ICT Express*, vol. 3, no. 1, pp. 14–21, Mar. 2017. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2405959517300061

[34] A. Computing *et al.*, "An architectural blueprint for autonomic computing," *IBM White Paper*, vol. 31, no. 2006, pp. 1–6, 2006.

[35] H. Seifzadeh, H. Abolhassani, and M. S. Moshkenani, "A survey of dynamic software updating," *Journal of Software: Evolution and Process*, vol. 25, no. 5, pp. 535–568, 2013. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1556

[36] V. Ilvonen, P. Ihantola, and T. Mikkonen, "Dynamic software updating techniques in practice and educator's guides: A review," in *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, 2016, pp. 86–90.

[37] J. Buisson, F. Dagnat, E. Leroux, and S. Martinez, "Safe reconfiguration of coqcots and pycots components," *Journal of Systems and Software*, vol. 122, pp. 430–444, 2016.

[38] M. Chardet, H. Coullon, and S. Robillard, "Toward Safe and Efficient Reconfiguration with Concerto," *Science of Computer Programming*, vol. 203, p. 1, Mar. 2021. [Online]. Available: https://hal.inria.fr/hal-03103714

[39] M. Weißbach, N. Taing, M. Wutzler, T. Springer, A. Schill, and S. Clarke, "Decentralized coordination of dynamic software updates in the Internet of Things," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, Dec. 2016, pp. 171–176.

[40] H. Herry, P. Anderson, and M. Rovatsos, "Choreographing configuration changes," in *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*. IEEE, 2013, pp. 156–160.

[41] K. Wild, U. Breitenbücher, K. Képes, F. Leymann, and B. Weder, "Decentralized Cross-organizational Application Deployment Automation: An Approach for Generating Deployment Choreographies Based on Declarative Deployment Models," in *Advanced Information Systems Engineering*, ser. Lecture Notes in Computer Science, S. Dustdar, E. Yu, C. Salinesi, D. Rieu, and V. Pant, Eds. Cham: Springer International Publishing, 2020, pp. 20–35.

[42] D. Sokolowski, "Deployment coordination for cross-functional DevOps teams," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 1630–1634. [Online]. Available: https://doi.org/10.1145/3468264.3473101

[43] https://www.ansible.com/.

[44] https://www.terraform.io/.

[45] https://www.pulumi.com/.

[46] L. Mottola, G. P. Picco, and A. Amjad Sheikh, "Figaro: Fine-grained software reconfiguration for wireless sensor networks," in *European Conference on Wireless Sensor Networks*. Springer, 2008, pp. 286–304.

[47] R. Di Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro, "Aeolus: A component model for the cloud," *Information and Computation*, vol. 239, pp. 100–121, 2014.

[48] https://www.chef.io/.

[49] https://juju.is/juju-architecture.

[50] N. Hamed, O. Rana, C. Perera, P. Orozco Ter Wengel, and B. Goossens, "Open data observatories: a survey," *none*, 2021.

[51] V. Kumar, S. Gunner, M. Pregnolato, P. Tully, N. Georgalas, G. Oikonomou, S. Karatzas, and T. Tryfonas, "Sense (and) the city: From internet of things sensors and open data platforms to urban observatories," *IET Smart Cities*, 2024.

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

[53] D. Raj, P. A. Prabha, A. S. Pai, and N. Hridul, "A comprehensive survey on iot ocean observatory systems," in *2024 10th International Conference on Communication and Signal Processing (ICCSP)*. IEEE, 2024, pp. 130–135.

[54] *ORION-GEOSTAR-3: A Prototype of Seafloor Network of Observatories For Geophysical, Oceanographic And Environmental Monitoring*, ser. International Ocean and Polar Engineering Conference, vol. All Days, 05 2004.

[55] A. Ghobakhlou, X. Wang, P. Sallis, S. Inder, and S. Blok, "Using wsn for possum management," in *2015 9th International Conference on Sensing Technology (ICST)*. IEEE, 2015, pp. 689–693.

[56] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "Flexcup: A flexible and efficient code update mechanism for sensor networks," in *Wireless Sensor Networks: Third European Workshop, EWSN 2006, Zurich, Switzerland, February 13-15, 2006. Proceedings 3*. Springer, 2006, pp. 212–227.

[57] S. Brown and C. J. Sreenan, "Software updating in wireless sensor networks: A survey and lacunae," *Journal of Sensor and Actuator Networks*, vol. 2, no. 4, pp. 717–760, 2013.

[58] A. Taherkordi, F. Loiret, R. Rouvoy, and F. Eliassen, "Optimizing sensor network reprogramming via in situ reconfigurable components," *ACM Transactions on Sensor Networks (TOSN)*, vol. 9, no. 2, pp. 1–33, 2013.

[59] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *Proc. of the 1st USENIX/ACM Symp. on Networked Systems Design and Implementation*, vol. 25, 2004, pp. 37–52.

[60] j. w. hui and d. culler, "the dynamic behavior of a data dissemination protocol for network programming at scale," in *proceedings of the 2nd international conference on embedded networked sensor systems*, ser. sensys '04. new york, ny, usa: association for computing machinery, 2004, p. 81–94. [Online]. Available: https://doi.org/10.1145/1031495.1031506

[61] I. Alfonso, K. Garcés, H. Castro, and J. Cabot, "Self-adaptive architectures in iot systems: a systematic literature review," *Journal of Internet Services and Applications*, vol. 12, pp. 1–28, 2021.

[62] 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.

[63] A. Verma, Savita, and S. Kumar, "Routing Protocols in Delay Tolerant Networks: Comparative and Empirical Analysis," *Wireless Personal Communications*, vol. 118, no. 1, pp. 551–574, May 2021. [Online]. Available: https://doi.org/10.1007/s11277-020-08032-4

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

[65] R. Tollefsen, I. Rais, J. M. Bjørndalen, P. Hoai Ha, and O. Anshus, "Distribution of Updates to IoT Nodes in a Resource-Challenged Environment," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2021, pp. 684–689.

[66] E. L. Lloyd and G. Xue, "Relay node placement in wireless sensor networks," *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 134–138, 2007.

[67] S. Santhosh Kumar, Y. Palanichamy, M. Selvi, S. Ganapathy, A. Kannan, and S. P. Perumal, "Energy efficient secured k means based unequal fuzzy clustering algorithm for efficient reprogramming in wireless sensor networks," *Wireless Networks*, vol. 27, pp. 3873–3894, 2021.

[68] M. Sadrishojaei and F. Kazemian, "Clustered routing scheme in iot during covid-19 pandemic using hybrid black widow optimization and harmony search algorithm," in *Operations Research Forum*, vol. 5, no. 2. Springer, 2024, p. 47.

[69] R. Van Glabbeek, E. H. Teshome, D. Deac, T. Jemal, J. Tiberghien, and K. Steenhaut, "A building block for internet of things prototyping," in *IECON 2022 – 48th Annual Conference of the IEEE Industrial Electronics Society*, 2022, pp. 1–6.

[70] A.-S. Tonneau, N. Mitton, and J. Vandaele, "How to choose an experimentation platform for wireless sensor networks? a survey on static and mobile wireless sensor network experimentation facilities," *Ad Hoc Networks*, vol. 30, pp. 115–127, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570870515000451

[71] L. Guegan, S. Tofaily, and I. Raïs, "Design and evaluation of single-board computer based power monitoring for iot and edge systems," in *2023 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing & Communications (GreenCom) and IEEE Cyber, Physical & Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*. IEEE, 2023, pp. 716–723.

[72] G. Fieni, R. Rouvoy, and L. Seinturier, "Smartwatts: Self-calibrating software-defined power meter for containers," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 479–488.

[73] G. Fieni, R. Rouvoy, and L. Seiturier, "Selfwatts: On-the-fly selection of performance events to optimize software-defined power meters," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021, pp. 324–333.

[74] R. Trüb, R. Da Forno, L. Sigrist, L. Mühlebach, A. Biri, J. Beutel, and L. Thiele, "Flocklab 2: Multi-modal testing and validation for wireless iot," in *3rd Workshop on Benchmarking Cyber-Physical Systems and Internet of Things (CPS-IoTBench 2020)*. OpenReview. net, 2020.

[75] P. Tian, X. Ma, C. A. Boano, Y. Liu, F. Yang, X. Tian, D. Li, and J. Wei, "Chirpbox: An infrastructure-less lora testbed," 02 2021.

[76] L. Guegan, I. Rais, and O. Anshus, "Validation of esds using epidemic-based data dissemination algorithms," in *2023 19th Annual International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, 2023.

[77] J. Philippe, A. Omond, H. Coullon, C. Prud'Homme, and I. Raïs, "Fast choreography of cross-devops reconfiguration with ballet: A multi-site openstack case study," in *SANER 2024-IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2024.

[78] F. Arfi, H. Coullon, F. Loulergue, J. Philippe, and S. Robillard, "An overview of the decentralized reconfiguration language concerto-d through its maude formalization," *arXiv preprint arXiv:2412.08233*, 2024.

[79] 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.

[80] R.-A. Cherrueau, M. Delavergne, A. van Kempen, A. Lebre, D. Pertin, J. R. Balderrama, A. Simonet, and M. Simonin, "Enoslib: A library for experiment-driven research in distributed computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1464–1477, 2022.

[81] J. A. Khan, R. Pujol, R. Stanica, and F. Valois, "On the energy efficiency and performance of neighbor discovery schemes for low duty cycle iot devices," in *Proceedings of the 14th ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, & Ubiquitous Networks*, ser. PE-WASUN '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 63–70. [Online]. Available: https://doi.org/10.1145/3134829.3134835

[82] J. Hester, K. Storer, and J. Sorber, "Timely Execution on Intermittently Powered Batteryless Sensors," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017.

[83] S. Sharma, D. Kumar, and K. Kishore, "Wireless Sensor Networks-A Review on Topologies and Node Architecture," 2013. [Online]. Available: https://www.researchgate.net/publication/309180675_Wireless_Sensor_Networks-A_Review_on_Topologies_and_Node_Architecture

[84] L. Moharana, B. K. Biswal, R. Raj, and S. Naik, "Comparison of Performance Metrics of Star Topology and Ring Topology in Wireless Sensor Network," in *Advances in Intelligent Computing and Communication*, ser. Lecture Notes in Networks and Systems, M. N. Mohanty and S. Das, Eds. Singapore: Springer, 2020, pp. 122–134.

[85] D. ARI, M. ÇIBUK, and F. AĞGÜN, "The Comparison of Energy Consumption of Different Topologies in Multi-hop Wireless Sensor Networks," in *2018 International Conference on Artificial Intelligence and Data Processing (IDAP)*, Sep. 2018, pp. 1–5. [Online]. Available: https://ieeexplore.ieee.org/document/8620903/

[86] B. G. Awatef, N. Nejeh, and K. Abdennaceur, "Impact of Topology on Energy Consumption in Wireless Sensor Networks," 2014. [Online].

Available: https://www.researchgate.net/profile/Nasri-Nejah/publication/321485180_RP_Journal_2246-137X_124/data/5a24654caca2727dd87e50b9/RP-Journal-2246-137X-124.pdf

[87] A. Salhieh, J. Weinmann, M. Kochhal, and L. Schwiebert, "Power efficient topologies for wireless sensor networks," in *International Conference on Parallel Processing, 2001.*, Sep. 2001, pp. 156–163, iSSN: 0190-3918. [Online]. Available: https://ieeexplore.ieee.org/document/952059/

[88] L. Guegan, I. Rais, and O. Anshus, "A large-scale study of the impact of node behavior on loosely coupled data dissemination: The case of the distributed arctic observatory," *Journal of Parallel and Distributed Computing*, vol. 197, p. 105013, 2025.

[89] I. Raïs, J. M. Bjørndalen, P. Hoai Ha, K.-A. Jensen, L. S. Michalik, H. Mjøen, O. Tveito, and O. Anshus, "UAVs as a Leverage to Provide Energy and Network for Cyber-Physical Observation Units on the Arctic Tundra," in *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, May 2019, pp. 625–632, iSSN: 2325-2944.

[90] S. Tofaily, I. Rais, and O. Anshus, "Quantifying the variability of power and energy consumption for iot edge nodes," in *2023 19th Annual International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, 2023.