Experiences Building and Deploying Wireless Sensor Nodes for the Arctic Tundra

Michael J. Murphy,* Øystein Tveito,* Eivind Flittie Kleiven,†
Issam Raïs,* Eeva M. Soininen,† John Markus Bjørndalen,* and Otto Anshus*

*Department of Computer Science / †Department of Arctic and Marine Biology
UiT The Arctic University of Norway
email: { michael.j.murphy, oystein.tveito, eivind.f.kleiven }@uit.no
email: { issam.rais, eeva.soininen, john.markus.bjorndalen, otto.anshus }@uit.no

***Abstract*— The arctic tundra is most sensitive to climate change. The change can be quantified from observations of the fauna, flora and weather conditions. To do observations at sufficient spatial and temporal resolution, ground-based observation nodes with sensors are needed.**

However, the arctic tundra is resource-limited with regards to energy, data networks, and humans. There are also regulatory and practical obstacles. Consequently, observation nodes must be small and unobtrusive, have a year or longer operational lifetime from small batteries, and be able to report results and receive software updates over scarce back-haul networks.

We describe the architecture, design, and implementation of prototype observation nodes deployed to the arctic tundra for the periods August 2019 to July 2020 and August 2020 to July 2021.

For the 2019 deployment, ten nodes were each placed inside ten existing camera traps. A camera trap is a box with a wildlife camera taking pictures of rodents when they enter the box from tunnels under snow and ice. For the 2020 deployment, eight nodes were located pairwise inside four camera traps.

Each node measures carbon dioxide level and temperature inside the camera trap during the winter season. A node reports its state and observational data each night over a commercial low power IoT telecom back-haul network, if available.

We report on the issues encountered doing actual deployments of the prototype nodes. For each issue, we describe the reason for why it happened, relate it to the architecture, design and implementation, and explain what we did about it.

I. INTRODUCTION

The circumpolar arctic tundra is the Earth's terrestrial biome that is the most sensitive to climate change. The extent of projected warming is so extreme that tundra ecosystems will likely transform into novel ecological states within a few decades, potentially leading to loss of important ecological functions and biodiversity. To be able to reliably predict such transitions and their consequences, climate-ecological models need data from many observations providing for measurements of climate and ecosystem state variables. The Climate-ecological Observatory for Arctic Tundra (COAT) (http://www.coat.no) is working to make such observations.

Presently, ecological observations are still heavily dependent on data collected by field personnel traveling to the tundra to make observations in person, by hand. Technological developments include automated *camera traps*, timer- and motion-activated cameras that photograph wildlife. Such instruments must still be placed by hand, and they typically do not include any communications capability. Field personnel must return to the site in the next fieldwork season (typically a year later) to collect the captured data and reconfigure the instruments. This observe-by-travel approach limits the possible number of observations to a human scale. An observe-by-wire approach, with many automated sensor nodes that can collect and transmit data with minimal human intervention, could greatly increase the spatial and temporal coverage of the observations.

However, the conditions found on the arctic tundra present multiple challenges to such a system. Energy is scarce in winter. The sun does not rise, and though wind is plentiful, it may be inaccessible due to deep snow. Communications infrastructure is scarce. There is little to no LTE coverage, and satellite communication is expensive. Roads are scarce. Travel to the field requires specialized vehicles like ATVs, snowmobiles, or even helicopters, with the last mile on foot. Snow and ice are not scarce. Nodes will be buried in snow and ice during winter, and flooded by snow melt in the spring. Strategies to address these challenges involve difficult trade-offs. Large batteries add weight. Aggressive energy-aware task scheduling complicates systems. Powerful antennas require more energy. Masts to raise antennas or wind turbines above snow will add weight and complexity and may be barred by local regulations against obtrusive installations.

The Distributed Arctic Observatory (DAO) (https://site.uit.no/dao/) researches ways to overcome these challenges and build a comprehensive observation system for the arctic tundra.

In this paper, we present a vision for a distributed arctic observatory architecture, and we present the design and implementation of a prototype observatory system that we built and deployed to measure carbon dioxide inside existing COAT camera traps [1]. We report on our experiences deploying a ten-node system on the tundra in the far north east of Norway from the summer of 2019 to the summer of 2020. We examine issues that arose during the deployment period and take an unflinching look at their causes. For each issue, we locate its cause within the system and discuss implementation or design changes that were applied — or could be applied in future work — to address it. We distill this experience into lessons learned for future wireless sensor network development.

The structure of the paper is as follows. Section II presents related work. Section III describes the observation system at three abstraction levels: an idealized, abstract architecture (III-A), the applied CO_2 observation system design (III-B), and actual implementation details (III-C). Section IV describes the actual deployment, and Section V describes the system's operation while deployed. Section VI analyzes different issues that arose during the deployment period and discusses lessons learned and how the issues can be avoided. Finally, Section VII is the conclusion.

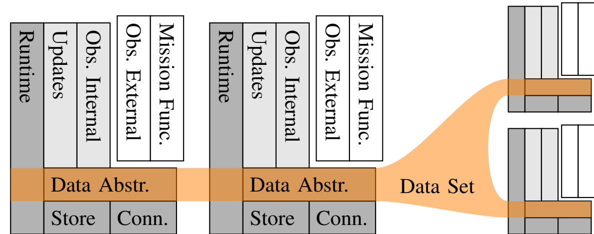


Fig. 1. Architecture for a Distributed Arctic Observatory

II. Related Works

Ecological research is increasingly embracing technology for data collection [2]. The field of biologging or biotelemetry tags animals with GPS devices that record their positions over time [3], [4]. Ecological research technology is often in the form of commercial data loggers, which can often be costly monetarily. The rise of accessible low-cost computer development kits such as the Raspberry Pi and Arduino is spurring an interest in do-it-yourself sensors for ecological research [5]–[7]. However, there are still few attempts to use DIY sensors for serious ecological work.

Wireless sensor networks (WSNs) are an active and diverse topic in computer science, with applications such as the Internet of Things (IoT) [8] and smart cities [9], which flourish with the ubiquity of modern wireless infrastructure. Further away from infrastructure are applications like smart agriculture monitoring [10], [11] which, though rural, are still tied to human activity and thus still often enjoy some access to infrastructure. Much further afield is habitat monitoring [12]. ZebraNet [13] notably used incidental contact of wandering zebras to relay data to base stations with back-haul networks. Still, even habitat monitoring systems rarely venture into the arctic.

WSNs have been deployed to demanding environments such as rivers [14], deserts [15], and even volcanoes [16]. These environments can damage sensor nodes or impact the measurements for several reasons, including humidity, dust, and temperatures. Therefore, a node must be built with its deployment environment in mind. The sensors we describe are built to be physically isolated and exposed for up to a year in the arctic tundra with low temperatures, high wind, heavy snow and ice, and high humidity.

The environment where nodes are deployed can have a significant impact on the radio propagation properties. Experiences from a system deployed inside caves document that thick rocks make communication challenging [15]. Sensors on the tundra will be covered in snow and ice. Physics literature [17] suggests that the effect of snow and ice on radio waves will vary with how densely packed the snow is. Soft snow, with a low relative permittivity ($\varepsilon_r = 4$), should have little hindering effect on radio waves. Compact wet snow ($\varepsilon_r = 50$) should have more of an effect, but still less than water ($\varepsilon_r = 80$).

Previous deployments of wireless networks in arctic conditions include ice- and snow-monitoring stations in Antarctica [18], sensors dropped into boreholes in a glacier in Norway [19], and a system to deliver internet to Sámi reindeer herders in northern Sweden [20] using delay-tolerant networking (DTN) [21]. Each of these employs larger batteries than are practical for our application, plus some combination of solar [18], [19], wind [18], or even diesel generators [20] to charge their batteries. Such charging strategies are not viable for sensors that are buried under the snow and where regulations prohibit protruding masts for antennas or wind turbines.

A current trend in sensors systems is ultra-low power, batteryless, energy-harvesting sensor motes [22]. Energy in the tundra is difficult to harvest in winter, so our sensor deployment still uses batteries for now. We also used low-cost off-the-shelf components for our prototype sensors rather than advanced systems-on-chip. However, arctic research could pose an interesting challenge for the batteryless frontier, and as ultra-low-power system-on-chip technology improves [23], batteryless arctic operation may become more feasible.

Because of the harsh conditions imposed by the arctic tundra, it is complicated to cope with current and state of the art solutions and difficult to decide which design choices could be the best in our context. To understand and study the implications for such a system, we decided to design a simple solution, helping us to keep track of the possible problems that might arise.

III. CO_2 Observation System

We describe the CO_2 observation system at three levels of abstraction: a high-level *architecture* that describes an idealized, abstract structure for a distributed arctic observatory system, an intermediate *design* that characterizes the specific CO_2 observatory application that we built, and the concrete *implementation* details of the CO_2 observatory system as built and deployed.

A. Architecture

Sensor nodes in an arctic observatory must be careful with energy and must operate on their own without assuming that a stable network connection is available. The Distributed Arctic Observatory envisions an arctic observatory made up of smart, autonomous sensor nodes called *observation units (OUs)* that are distributed across the tundra, collecting data that is part of a greater tundra *data set* and storing it until some network becomes available or some messenger arrives to carry the data away to a more secure location.

We divide the general high-level architecture of these observation units into eight fundamental abstractions. Six abstractions constitute a platform that is common to all OUs: *runtime, store, connectivity, updates,* and *observation of internal state.* The other two are customized to each application: *observation of external state* and *mission-specific functionality.* These abstract architectural components are visualized in Fig. 1 and described below.

1) Runtime: The runtime abstraction is the manager of the other component abstractions. It schedules other functionalities, and in a physical environment where it makes sense to spend most of the time hibernating, the runtime decides when to sleep and when to wake. In a sophisticated observation unit, this may be a full operating system with parallel multiprocessing and memory protection. It may schedule and customize operations dynamically based on the history of collected internal state. It could make these decisions by heuristics or even by machine learning. Or, in a minimal OU, the runtime could be as simple as a sequential task-running loop: wake, check fixed schedule, run task, go back to sleep.

2) Store: The store abstraction represents the local data storage of the observation unit. Because an OU cannot rely on a network connection, data must be stored locally by the store abstraction until it can be delivered to a more-central node for safekeeping.

3) Connectivity: An observation unit cannot assume that a network is available, but it must be ready to make use of any network that becomes available. This is the responsibility of the connectivity abstraction: to manage connections to other nodes across different communications technologies or different network topologies, to negotiate sleep times, and to navigate network partitions and failure events.

4) Data Abstraction: Bridging the store and connectivity abstractions is the data abstraction. The data abstraction links the data collected by individual observation units into a larger, distributed *data set*. It manages the flow of data between available connections, replicating data across neighboring nodes for safety and/or routing data towards a back-haul network for aggregation at a more central node. The form of such a data abstraction is part of our ongoing research and the properties of this data set are not the focus of this paper.

5) Updates: Because bugs are inevitable and requirements change, a remote observation unit must be equipped to receive and apply remote software updates. This is the responsibility of the update abstraction. It makes use of the connectivity and data abstractions to transfer the content of software updates from a central authority to the local node, and it manages the details of verifying and applying downloaded software updates.

6) Observation of Internal State: An autonomous observation unit must have some degree of self awareness to feed back into the decision-making of the runtime abstraction. This awareness comes from the OU's monitoring and recording of its own internal state, including such metrics as battery level, available networks, state of local storage, and error logs. The purpose is to provide the other abstractions with information to use to adapt the node's behavior to changing conditions, and to supply operators with information needed to diagnose errors.

7) Observation of External State: The observation of external state is the observatory's raison d'etre: to observe and record state variables related to external conditions of the arctic tundra environment. The external state abstraction encompasses the sensors needed to observe these state variables and the programming responsible for collecting and storing this data. This can include observations such as atmospheric data and weather conditions, photos or video of plants or animals, or audio recordings of e.g. bird songs. All data is to be stored by the store abstraction and guided by the data abstraction to become part of the larger data set.

8) Mission-Specific Functionality: The mission-specific functionality abstraction is for any other functionality required by the observatory application. This may include edge analytics such as compression or summarization of observed data for transmission over constrained networks.

B. Design

Our prototype application of the distributed arctic observatory architecture is a CO_2 observation system that is a companion to existing under-snow camera traps used by COAT to photograph small rodents such as lemmings that live under the tundra snows in winter [1]. The CO_2 observation unit is designed to fit into a hollow space in the wall of the existing camera trap boxes. It records an external state variable that the existing cameras do not: carbon dioxide (CO_2) levels. It also records temperature to double-check the camera's temperature measurements, and it observes the camera itself by detecting the infrared light of the camera's infrared flash. The OU records CO_2 and temperature every 30 minutes, and it attempts to transmit collected data to a central server over an LTE-M network each night. It spends the rest of its time in a deep sleep state. Detections of a camera flash trigger a brief wake-up state that increments an "observed flashes" counter, which is then recorded as part of the half-hourly CO_2 and temperature observations.

The CO_2 OU design simplifies but does not encompass all the characteristics of the prescribed architecture. This simplification was done to meet deployment deadlines. The nodes must be deployed before the arrival of winter buries the camera traps with snow and makes further deployment impossible without disturbing the under-snow habitat. In practice, these simplifications led to node behaviors and failures that are described later in the paper (Section VI). The design is visualized in Fig. 2. It compares to the prescribed architecture as follows.

1) Runtime $\rightarrow$ Task Runner: The runtime of the CO_2 OU is a simple, sequential task runner that manages the scheduling of external observations (every 30 minutes) and connectivity (every night) as its major tasks. The design is made to be simple to run on a microcontroller without an operating system. As such, it runs its tasks in sequence as scheduled subroutines, waking to run the scheduled subroutine and then

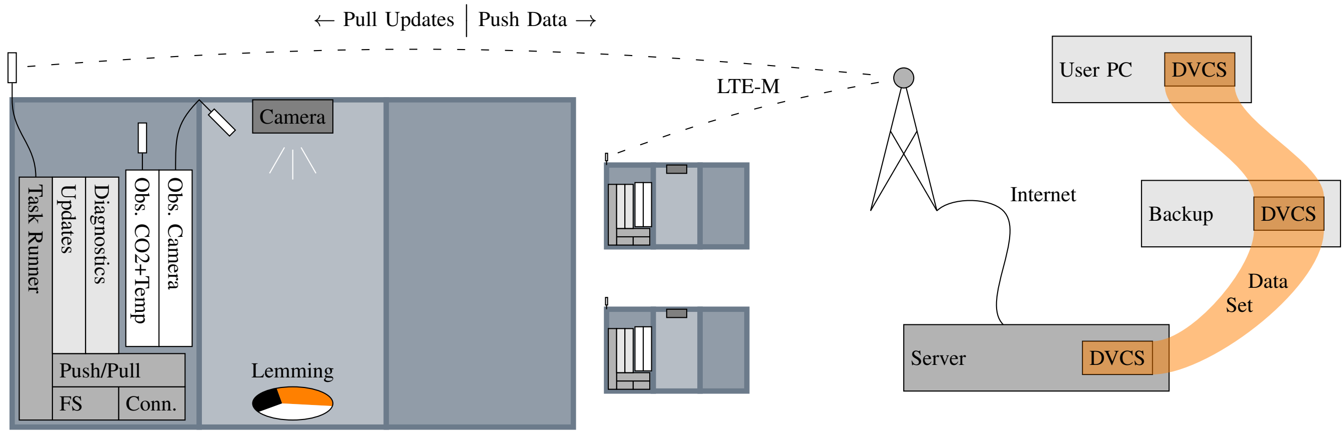


Fig. 2. Design of CO_2 observation unit system, integrated with COAT wildlife camera boxes.

going back to sleep until it is time to run the next scheduled task/subroutine.

The task runner also runs additional subroutines in response to specific conditions. After booting from a hard power cycle, it runs a self test subroutine that checks hardware and connectivity and then gives visual feedback to the operator via an onboard LED. After waking via a camera-flash trigger, it runs the subroutine to increment the camera-flash counter and then goes back to sleep. It runs an updates subroutine after a power cycle (after the self test) and after the connectivity routine (if updates were downloaded). It also records errors if any subroutine throws an uncaught exception, and it runs an error-recovery subroutine if it boots from an unexpected reset.

2) Store → Filesystem: The store in this case is a typical hierarchical filesystem on local storage. Internal and external observation data is stored in an append-only fashion to local files. The files are numbered sequentially (e.g. `data-0000.csv` or `log-0000.txt`). When a file grows past a certain threshold (100 KiB), a new file with an incremented name is created and used. Directories named by date under the `updates/` directory, e.g. `updates/update-2019-10-11`, are monitored by the update abstraction for installation.

3) Connectivity → LTE-M: Connectivity for the CO_2 OUs is simplified to connect to a commercial LTE-M back-haul network if available. Connection attempts are scheduled each night. Once an internet connection is established, the OU makes a direct connection to a central server via a RESTful HTTP API over TCP. The network topology is a star with the server at the center. A central server is a single point of failure that could eventually limit scaling of a large deployment, but it was a simple starting point for this early experiment.

4) Data Abstraction → Data Push/Pull: The data abstraction for the CO_2 OUs is a simplified push/pull system that is activated once the connectivity abstraction makes a connection to the central server. New observation data is pushed to the server and new updates are pulled from it.

The subroutine for pushing data relies on the fact that the data is sequential and append-only. It stores a position of the last data to be transferred to the server (a filename and file position pair). It sends data in small chunks, starting at the last position, and only advancing the position after an ACK from the server. If the OU's stored position is reset or otherwise falls out of sync from the data stored on the server, the server can send a corrected position as part of its ACK, which resets the OU's saved position.

Similarly to the pushing of data, the subroutine for pulling updates relies on a sequential naming scheme. It first asks the server for the name of the latest update. Then, if that update does not exist locally, it asks for a list of all files in the update and then downloads them one by one. It keeps the incomplete download in a holding area until all files have been downloaded. Then when the download is complete, it moves the entire directory from the holding area into the updates directory that is monitored by the updates abstraction.

On the central server, data from each OU is aggregated into the complete data set, which is then stored in a distributed version control system (DVCS). The data from the CO_2 OUs is in text-based file formats and is on the order of megabytes per year per OU, so a DVCS is a natural fit. Once in version control, the entire data set can be easily replicated and synced between user PCs or backups. Updates, once developed and ready for distribution, are committed to the file hierarchy in the DVCS. Once these commits make their way to the server, they are then available to be pulled by the OUs.

5) Updates: The updates abstraction for the CO_2 OUs works by checking the designated updates area on the filesystem for new updates. When a new directory of update files is detected, it installs them by copying the program code into the active program area of the microcontroller's flash memory. The runtime/task runner runs the update subroutine after a hard reset to detect updates that are loaded manually via SD card, and also after a connectivity cycle if new updates have been downloaded.

6) Observation of Internal State → Diagnostics: Internal state observed by the CO_2 OU is mostly diagnostic information logged to sequential `log-0000.txt` files in the store/filesystem. This information is mostly in the form of logged exceptions that were thrown by the different subroutines and caught

by the runtime/task runner. Error information is also logged when the error recovery routine is run after an unexpected reset. The connectivity routine also records the received signal strength indicator (RSSI) of the LTE-M signal each time it makes a connection.

7) Observation of External State → CO_2, Temperature, and Light: The external state recorded by the CO_2 observation units is, of course, the CO_2 level inside the camera box. This measurement is recorded once every $30\,\mathrm{min}$, along with temperature and the count of camera flashes that were observed between scheduled wake-ups. These are appended as a row of comma-separated values to the current sequential `data-0000.csv` file on the store/filesystem.

8) Mission-specific Functionality → No-op: The CO_2 observation units have no additional analytics or other mission-specific functionality.

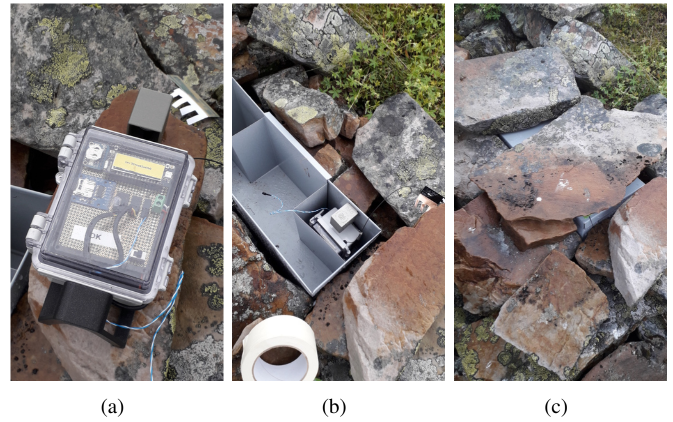
C. Implementation

The design of the CO_2 observation system is concretized by the implementation.

The CO_2 observation unit implementation is based on a Pycom FiPy microcontroller which includes an onboard LTE modem for connectivity. The *runtime* is the FiPy's MicroPython runtime environment with a custom sequential task runner written in MicroPython code. The FiPy was chosen for its low cost and for its inclusion of multiple radio technologies that can potentially be used for future expansion of the connectivity abstraction. The code is sequential, running tasks and catching exceptions. Execution is monitored by a system-wide watchdog timer, which resets the system if it is not "fed" by the task runner or its subroutines after $60\,\mathrm{s}$. The watchdog timer catches unexpected freezes and acts as a last resort against unexpected behavior.

Storage is provided by a MicroSD card and the filesystem is FAT. *Connectivity* is established via the FiPy's LTE-M modem in Cat-M1 mode. The *push/pull data abstraction* is implemented as a MicroPython routine in a straightforward coding of the behavior described in the design section. The *update* routine also works as described in the design section. It checks for downloaded MicroPython module files in the `updates/` directory of the SD card (either Python source or precompiled bytecode) and installs them into the FiPy's flash memory for execution. In terms of *observations of internal state*, the task runner writes exceptions that it catches to the log, as well as unexpected resets. The connectivity routine also records LTE RSSI as measured by the LTE modem on each connection. Log entries are timestamped using a hardware real-time clock (RTC) that is more accurate than the FiPy's internal RTC and which is set via NTP after LTE communication.

The central server is a simple RESTful HTTP server written in Python using the Flask-RESTful library, running in a Linux LXC container in our campus network. A cron job commits the collected data to a Git repository once per day and then pushes the commits to a local GitLab installation. From there, users can clone the data and examine it.



(a) (b) (c)

Fig. 3. A CO_2 observation unit being deployed. (a) Detail of OU. (b) The OU is placed in the hollow wall of the camera trap box. (c) The closed camera box is covered with stones for protection and camouflage.

Physically, all internal logic components of each OU are encased in a water-resistant enclosure, with leads for external sensors. A photo of a completed CO_2 OU is shown in Fig. 3(a). *Observations of external state* are provided by specialized hardware sensors. CO_2 is measured by a Gas Sensing Solutions ExplorIR-W-20 CO_2 sensor. It is affixed to the top of the node enclosure and is protected by a 3D-printed hood. There is also a 3D-printed stabilizing foot to lift the OU off the ground to avoid standing water. The temperature sensor is a DFRobot temperature probe made for outdoor use. It is affixed to the side of the enclosure. Camera flashes are detected by a generic IR light sensor. A flat LTE antenna is also attached outside of the enclosure.

The CO_2 OU, like the camera it compliments, is powered by twelve AA batteries. This arrangement was chosen for convenience of maintenance, so that only one type of batteries must be carried to the field. Initial energy use estimates showed that twelve AAs should be sufficient for a year of operation with measurements every half hour and up to eight minutes of LTE radio operation per day. The total hardware cost is approximately 330 USD per unit.

Complete source files for this CO_2 observation system are available online (https://github.com/arcticobservatory), including software source code, hardware parts lists and schematics, and model files for 3D-printed components.

IV. Deployment

COAT camera traps are deployed in clusters of twelve, in locations chosen to give a sampling of different elevations and terrain types. The primary terrain types are *snow beds*, rocky areas where deep snow accumulates, and *hummock* sites, marshier areas characterized by small mounds of soil. Within clusters, camera boxes are typically between $200\,\mathrm{m}$ and $600\,\mathrm{m}$ apart, and clusters are spaced at least $2.5\,\mathrm{km}$ apart. Deployment in the field is as shown in Fig. 3. The CO_2 observation unit is placed into the hollow cavity in the walls of the camera box, and the camera box is covered with stones found nearby for protection and camouflage. Care is taken

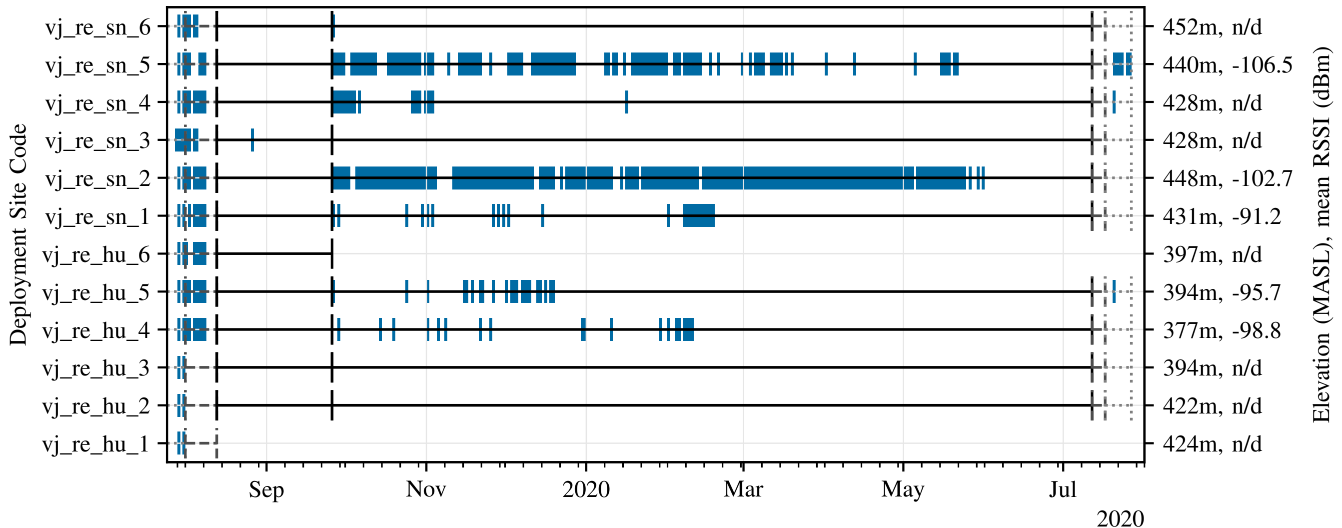


Fig. 4. Timeline of observation unit contact with back-end server during the 2019–2020 deployment.

to place the weight of the "roof" stones onto other stones rather than on the box itself. This also keeps the weight of accumulated snow off of the box during the winter.

Ten CO_2 observation units were deployed in August of 2019 to selected camera traps in the arctic tundra environment of Norway's Varanger peninsula (Varangerhalvøya), at the far north and east end of the country. The chosen cluster of camera traps are on the hill Reinhaugen (70°20′N 28°57′E) in the Vestre Jakobselv region of the peninsula. This cluster was chosen for the first deployment because mobile coverage maps [24] indicated that it should have good LTE-M coverage. Twelve OUs were originally carried into the field to match the twelve camera traps, but multiple issues (described below in Section V) reduced the number to ten.

Visiting these sites from the nearest town requires roughly four hours of travel by car and all-terrain vehicle, with the last few kilometers on foot due to regulations restricting ATV driving to designated trails. Consequently, maintenance cannot be done casually. One trip for in-field maintenance of the OUs was made in September 2019 to adjust antennas and install updated software.

During the summer of 2020, all OUs were retrieved and brought back to the lab. Memory cards were removed, and all observational data and error logs were saved. The following section describes the observation system's operation during this first year.

The OUs underwent maintenance, receiving new batteries and updated software, and then eight nodes were redeployed to Reinhaugen in August 2020. They will be operational from August 2020 until summer 2021. For this second deployment, the eight CO_2 nodes are placed two per camera trap box in four camera traps.

V. Operation

In this section, we describe the operation of the CO_2 observation system during its first year of deployment, from summer 2019 to summer 2020. The collected CO_2 data itself is erratic and inconclusive, and further analysis beyond the scope of this paper is needed to determine if it is useful. For this paper, we focus on the behavior of the CO_2 observation units themselves, leading to the analysis in the next section (Section VI) of issues encountered.

A timeline of contact from deployed CO_2 observation units during the winter 2019–2020 season is shown in Fig. 4. The left-hand labels show the site code of the camera box in which each CO_2 observation unit was deployed. These labels are comprised of region ("vj" for Vestre Jakobselv), cluster ("re" for Reinhaugen), terrain type ("sn" for snow bed, "hu" for hummock), and a number (1–6) that comes from the order in which the sites were initially surveyed and scouted to be camera trap sites. The sites are listed in descending alphanumeric order in the figure. This corresponds roughly to descending elevation above sea level of the camera traps.

Each horizontal line represents a deployed observation unit, with dashed portions representing periods of transit to and from the field, and dotted portions representing periods of testing in the lab. Shaded parts along each line show days when a node was able to reach the central back-end server, with consecutive days of contact becoming a solid bar. Black vertical lines represent changes in deployment status. These include the maintenance visit on Sept 26, 2019, and the retrieving of all OUs in July 2020. Labels on the right-hand side show the elevation in meters above sea level (MASL). Also shown is the average received signal strength indicator (RSSI) in decibel-milliwatts (dBm). The RSSI was measured using the FiPy's modem. RSSI readings are included in the average if the initial contact was successful, regardless of whether observation or log data was transferred successfully or not.

Starting at the left of the timeline, representing July and early August 2019, shading shows where the OUs made contact with the server as they were first tested in the lab and then in transit towards the Varanger Peninsula. After

Aug 13, site vj_re_hu_1 is empty due to problems with storage (Section VI-B). From Aug 13 to Sept 26, a lack of shading shows the connectivity issues (Section VI-C) that necessitated the maintenance visit on Sept 26. In this period only one contact was made between OU and server: that from vj_re_sn_3 on Aug 24. After Sept 26, site vj_re_hu_6 was also left empty due to problems with batteries (Section VI-B), while other sites show increased contact with the server.

By the time of pickup in July 2020, all OUs had ceased communicating. Examination after return to the lab showed that the OU from vj_re_sn_5 was still running on battery power and collecting data. It began communicating and transferring data again after a power cycle, which can be seen in the shaded region near the right of the figure. Of the others, 3 were damaged by flooding during the spring snow melt in late May and early June; 2 had drained batteries but were otherwise operational once connected to a power source; and 4 had drained batteries and were not operational when connected to power. The brief contact from the two operational OUs when powered on, vj_re_sn_4 and vj_re_hu_5, can also be seen in July. The four that were not operational were stuck in a "crash loop" (Section VI-B), running their watchdog timer reset routine over and over again.

Two OUs experienced data loss due to damaged SD cards. Site vj_re_sn_2's SD card was destroyed by flooding. However, it managed to transfer all of its collected data up until May 31, which is presumably the day that it flooded. Site vj_re_hu_4's SD card was too corrupted to read. It was only able to transfer part of its data. Therefore, the date of last server contact (Feb 10, 2020) is later than timestamp of the latest data received (Nov 02, 2019).

In the next section, we expand on the operational issues that arose during the deployment period and extract lessons to be learned from the experience.

VI. Forensics and Lessons Learned

Here, we describe issues encountered with the CO_2 observation system and lessons learned from it. We examine issues via the architectural components in which they originated — runtime, storage, connectivity, data abstraction, updates, or observation of internal state — and discuss how improved design and implementation could have avoided the issues.

A. Runtime

A weakness of the CO_2 observation unit's runtime is its error-handling model: it has a limited ability to respond to code that stalls during execution. This is because it runs scheduled routines sequentially, expecting them to either return or to throw exceptions. Stalls are handled by a global watchdog timer (WDT) that resets the OU into an error-handling routine after 60 seconds if the watchdog is not "fed" during forward progress.

Trouble arose when unanticipated stalls occurred during deployment. One unexpected stall occurred during the OU's power-on self test routine, sending the OU into an infinite self-test loop. As part of the self test, an NTP request was sent using a library that set a 1 s socket timeout. If the request took more than 1 s, it would time out and throw an exception for the test code to catch. What was unexpected was that this call also included a DNS lookup that was not governed by the timeout. Out in the field where signal strength was reduced, the DNS request packet was lost, and the network code stalled while waiting for it. The stall triggered the watchdog timer, and then, because operation had not progressed past the self test, the OU runtime ran the self test again, causing an infinite loop. Breaking this loop in the field required improvisation: temporarily dislodging the LTE SIM card so that the LTE network refused a connection. This caused the self test to fail in an expected manner by throwing an exception.

On an implementation level, this bug was fixed by increasing the watchdog timeout from an overly aggressive 10 s to 60 s, giving enough time for the DNS lookup to reach its implicit internal timeout and throw an exception. Also, the potential for a loop was disabled by changing the runtime to only attempt to run the power-on self test once.

However, on a design level, it should not have been possible for an unexpected failure mode in the self test to lead to an execution loop. The loop happened because of design mistakes in scheduling. The first version of the task runner checked its schedule after running each routine and before going to sleep. It then set a state variable in nonvolatile RAM for what routine to run next on wake or reset. Because this next-routine variable was followed blindly after a reset, and because the unexpected reset prevented execution from progressing past the self test, the runtime became stuck re-running the self test on every reset. To fix this, the runtime was redesigned and rewritten to instead double-check its power-up/wake reason and its schedule on every reset or wake-up, *before* executing any routines. The new design is more robust in the face of unexpected resets.

A further improvement to the design of the runtime would be to use parallelism. This would not only allow the runtime to be more flexible with scheduling, running multiple routines at once, but it would also allow a watchdog thread to monitor execution and recover control from stalled routines without resorting to a full system reset.

Lessons learned: 1) Incorporate unexpected stalls and freezes into the system's failure model. 2) Double-check state and schedule *before* running any routines. 3) Simple runtime environments are under pressure to accumulate more operating-system like features, such as time-sharing and process isolation.

B. Storage

The first problem encountered with storage was that during transit, the SD cards of three separate observation units were jostled out of their sockets. One SD card could simply be reseated, but two cracked in half and were rendered inoperative. We were carrying a spare OU, but no spare SD cards. Thus, only 11 of the 12 planned OUs were initially deployed (the number was later reduced to 10; see Section VI-C below).

Over a year-long deployment, many observation units suffered storage failures that led to mission failure for the OU. At the root of this issue is the fact that the SD cards used for storage use a simple FAT filesystem which is not resilient in the face of dirty shutdowns. Unexpected watchdog timer resets cause exactly such unsafe shutdowns, and each reset runs a risk of corrupting the filesystem. Over time this corruption occurred in at least four of the deployed OUs. The corruption then led to another unexpected stall in the code. When attempting to open a file on the corrupt filesystem for writing, the MicroPython I/O API froze. The freeze triggered a watchdog timer reset, which then triggered the error routine to record the error. Attempting to open the error log file for writing then caused another stall, which in turn caused another reset, and so on. This is the "crash loop" noted previously in Section V. Affected OUs became stuck in a loop of trying to record their own errors, unable to make progress while their batteries drained completely. This led not only to a complete failure of these OUs' missions to observe conditions on the tundra, but it also led to a lack of diagnostic information. Without evidence in the log, the diagnosis had to be inferred post-mortem from each OU's final data and error timestamps and then confirmed by manually observing the OU's behavior while hooked up to a debugging console.

On an implementation level, this bug has been mitigated by keeping a counter of consecutive watchdog resets in non-volatile RAM. If it reaches an arbitrary threshold, e.g. five consecutive resets, the runtime will not record the error but will instead try to continue back to normal scheduled operation. This breaks the loop and prevents unnecessary battery drain. From here, adding a filesystem check could repair the filesystem and allow the OU to continue normally.

A simpler alternative that would eliminate this type of error entirely would be to use a more sophisticated fail-safe filesystem in the first place. There is a fail-safe filesystem with atomic writes called littlefs [25] that is created specifically for embedded flash storage. Current builds of MicroPython support littlefs on the FiPy's internal flash storage, but not on external SD cards. A hybrid storage system design could write first to littlefs in the FiPy's internal storage, then sync to the SD card. Building such a hybrid system is future work.

Lessons learned: 1) Use a fail-safe storage medium. 2) Do not forget that removable storage like SD cards still use fragile filesystems like FAT. 3) Storage media can still break physically. Use replication and redundant storage.

C. Connectivity

The most standout issue with connectivity was a problem with LTE antenna positioning. When we first deployed the observation units, we attached the antennas directly to the top of the metal camera boxes. Later experiments in the lab showed that this placement attenuated the signal by as much as $20\ \mathrm{dBm}$ compared to a free-standing OU. This attenuation was not enough to cause problems during development in the lab, but in the field, it led to the unexpected "self-test loop" behavior (Section VI-A) and prevented nearly all communication between the OUs and the server in the first weeks of deployment (Fig. 4 and Section V). This attenuation was fixed during the September 2019 maintenance visit by adding a $1\ \mathrm{cm}$ shim between each antenna and its metal box.

At an implementation level, this was a misunderstanding of the antenna's intended mounting, but it was also a lack of internal state observations. Reading the received signal strength indicator (RSSI) earlier in the development process would have uncovered the attenuation. This is a reminder that a quantitative scalar metric gives far more information than a binary one.

On a software implementation level, many details of data transfer were left up to the HTTP libraries used to connect to the server. Optimizations such as reusing TCP connections might have improved reliability over the challenged networks by reducing round-trips for handshakes and reusing flow-control state.

However, an improved design would use protocols such as MQTT or CoAP [26] that are designed for IoT over challenged networks, transferring small chunks of data directly with small packets, rather than building a TCP stream for an HTTP connection only to send a small chunk of data.

Of course, optimizing the use of a network does not help if the network is not there at all. An observation unit needs a robust abstraction for connectivity that will use any and all links available to it. The FiPy alone includes LTE-M, Wi-Fi, LoRa, Sigfox, and Bluetooth. A flexible connectivity abstraction could use the other radios to potentially reach neighboring nodes (or passing skiers or biologger-equipped animals) and relay what data it can towards a node that does have a back-haul connection. Developing such a connectivity abstraction is part of our group's ongoing research.

Lessons learned: 1) Do not make assumptions about antenna behavior. Measure and analyze signal strength metrics. 2) Choose protocols that are designed for the application use case. 3) In challenged networking environments, use every available networking resource.

D. Data Abstraction

No serious issues originated from the simplified data abstraction itself, though it was affected by the issues in storage and connectivity. In future work, as the observation unit storage and connectivity become more advanced and data is replicated across multiple local stores (e.g. internal flash memory and external SD card) or multiple nodes (e.g. by relay across neighboring nodes), it will be the responsibility of the data abstraction to manage the consistency of these replicas.

There was an implementation bug in the pull mechanism that allowed incomplete updates to be installed. If the download of a multi-file update failed after at least one file was downloaded successfully, the code to resume the download would instead mark the whole download as complete and pass it along to the update abstraction prematurely. This issue was discovered before any updates were sent, and it was remedied by first sending a one-file update to fix the bug before sending any other remote updates.

One design issue is that the hardware ID of the OU's FiPy is used to identify that OU's subset of collected data. This seems natural at first, but it results in a coupling between the SD card and the FiPy that can cause duplication if SD cards and FiPys get mixed up. During deployment, a broken SD card was replaced with one from a spare unit. This led to the spare unit's data up to that point being duplicated as part of the deployment OU's data. This duplication of data was easy to correct manually, but if many such incidents occurred, the situation would quickly become unmanageable. If the data subset had an ID of its own that traveled with it, this issue would not have arisen. Future work on the data abstraction should include insights from the field of Information-Centric Networking [27].

Currently, the simplified data abstraction does not consider security when storing or transporting data, leaving it vulnerable to man-in-the-middle attacks. Future work to secure the data abstraction can draw inspiration from the decentralized social network Secure Scuttlebutt [28], which adds cryptographic signatures and other security considerations to a very similar data model: single-writer, append-only logs of data that are propagated via gossip protocol.

Lesson learned: A data subset should have its own ID that moves with it across devices and storage media.

E. Updates

The updates functionality was briefly affected by the bug in the data abstraction that allowed incomplete downloads to be marked as complete. That was an implementation bug in the data abstraction, but it is also a weakness of design in the oversimplified updates abstraction. The update mechanism should independently verify the completeness and integrity of an update before applying it. This could be as simple as consulting a manifest file with checksums, but a more secure design could use cryptographic code signing. Adding these features is future work, as is adding another key feature: the ability to roll back faulty updates.

Lessons learned: 1) There will be bugs. Therefore, remote update capability is a key feature of any remote computer system. 2) Update files may be incomplete, corrupted, or even malicious. Therefore, the update mechanism itself must check the integrity of updates, without leaving it to the download protocol or storage mediums.

F. Observe Internal State

Limited observation of internal state was intertwined with other issues recounted above: failure of the runtime to check run state before running code resulted in a loop of self tests; failure to measure signal strength metrics led to antenna problems; and failure to detect and respond to filesystem corruption resulted in an inability to record any further state. Addressing these problems involved adding such measurements: checking run state before running code, measuring and reporting RSSI, and keeping a count of consecutive resets in nonvolatile RAM.

Another seemingly obvious missing internal state measurement is remaining battery power. The lack of battery measurement was not an oversight but a cost trade-off. The lithium-ion batteries used to power the CO_2 osbervation units have a relatively flat discharge voltage curve [29], and the change in voltage is too slight to be detected accurately by the FiPy's onboard analog-to-digital converter pins. Adding a more sensitive ADC would have added monetary cost and design complexity to the CO_2 OU hardware, and at the time we chose not to pursue it.

Lessons learned: 1) An autonomous node in a cyber-physical system must record information about its own internal state, not only as a text log, but as machine-readable state variables that can be used for operational decisions. 2) Accurate battery-level measurements may be worth the additional cost and complexity.

VII. Conclusion

Building a robust wireless sensor network is a challenge in itself, and it is an even greater challenge when building a WSN for a hard-to-reach environment that is inaccessible for most of the year, has little-to-no network coverage, and has practical barriers to energy harvesting. These are the characteristics of the arctic tundra. The Distributed Arctic Observatory (DAO) explores, prototypes, and builds autonomous observation units to be deployed as part of the Climate-ecological Observatory for Arctic Tundra (COAT) to monitor the arctic tundra at a large scale.

In this paper we show that even the simplest of observation systems — with only a few sensors and a single network — can be surprisingly difficult to build, deploy, and debug in a demanding environment such as the arctic tundra. We have pinpointed issues in our early design and implementation and enumerated lessons learned from the experience to set the direction for future work. We hope that this experience and the lessons learned will be enlightening for others as well.

Future work includes analysis of the gathered CO_2 data. We also intend to pursue further research into overcoming the challenges of the tundra and to develop a robust observatory platform based on the architecture that we have described. We intend especially to develop a flexible connectivity abstraction that can utilize multiple radio technologies and multi-hop store-and-forward networking to relay messages to units by any and all available means, along with a robust data abstraction to manage replication and consistency across unreliable links in order to treat the scattered data holistically as a single data set.

Acknowledgements

This project is funded by the Climate-ecological Observatory for Arctic Tundra (COAT) (https://www.coat.no/en/) under the COAT Tools program, and by the Distributed Arctic Observatory (DAO) (https://site.uit.no/dao/). DAO is sponsored by the Research Council of Norway (https://www.forskningsradet.no/en/) under the IKTPLUSS programme, grant no. 270672.

The authors would like to thank Jan Erik Knutsen for his help with logistics and fieldwork; Mathias Leines Dahle for

fieldwork; Irma Trane for lending her cabin and her hospitality; Dorothee Ehrich for help with mapping; and Arne Munch-Ellingsen and Telenor Research for donating data plans for the CO_2 observation units and for navigating the red tape to provide us with useful cell tower data.

REFERENCES

[1] E. M. Soininen, I. Jensvoll, S. T. Killengreen, and R. A. Ims, "Under the snow: a new camera trap opens the white box of subnivean ecology," *Remote Sensing in Ecology and Conservation*, vol. 1, no. 1, pp. 29–38, 2015.

[2] B. M. Allan, D. G. Nimmo, D. Ierodiaconou, J. VanDerWal, L. P. Koh, and E. G. Ritchie, "Futurecasting ecological research: the rise of technoecology," *Ecosphere*, vol. 9, no. 5, p. e02163, 2018.

[3] L. E. Loe, G. E. Liston, G. Pigeon, K. Barker, N. Horvitz, A. Stien, M. Forchhammer, W. M. Getz, R. J. Irvine, A. Lee, L. K. Movik, A. Mysterud, Å. Ø. Pedersen, A. K. Reinking, E. Ropstad, L. M. Trondrud, T. Tveraa, V. Veiberg, B. B. Hansen, and S. D. Albon, "The neglected season: Warmer autumns counteract harsher winters and promote population growth in arctic reindeer," *Global Change Biology*, vol. 27, no. 5, pp. 993–1002, 2021.

[4] E. Fuglei and A. Tarroux, "Arctic fox dispersal from Svalbard to Canada: one female's long run across sea ice," *Polar Research*, vol. 38, Jun. 2019.

[5] J. G. Mickley, T. E. Moore, C. D. Schlichting, A. DeRobertis, E. N. Pfisterer, and R. Bagchi, "Measuring microenvironments for global change: DIY environmental microcontroller units (EMUs)," *Methods in Ecology and Evolution*, vol. 10, no. 4, pp. 578–584, 2019.

[6] S. Nazir, S. Newey, R. J. Irvine, F. Verdicchio, P. Davidson, G. Fairhurst, and R. v. d. Wal, "WiseEye: Next generation expandable and programmable camera trap platform for wildlife research," *PLOS ONE*, vol. 12, no. 1, pp. 1–15, 01 2017.

[7] W. J. McBride and J. R. Courter, "Using Raspberry Pi microcomputers to remotely monitor birds and collect environmental data," *Ecological Informatics*, vol. 54, p. 101016, 2019.

[8] C. Perera, C. H. Liu, and S. Jayawardena, "The emerging internet of things marketplace from an industrial perspective: A survey," *IEEE Transactions on Emerging Topics in Computing*, vol. 3, no. 4, pp. 585–598, 2015.

[9] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 22–32, 2014.

[10] L. Ruiz-Garcia, L. Lunadei, P. Barreiro, and I. Robla, "A review of wireless sensor technologies and applications in agriculture and food industry: State of the art and current trends," *Sensors*, vol. 9, no. 6, pp. 4728–4750, 2009.

[11] T. Ojha, S. Misra, and N. S. Raghuwanshi, "Wireless sensor networks for agriculture: The state-of-the-art in practice and future challenges," *Computers and Electronics in Agriculture*, vol. 118, pp. 66 – 84, 2015.

[12] 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*, ser. WSNA '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 88–97.

[13] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein, "Energy-efficient computing for wildlife tracking design tradeoffs and early experiences with ZebraNet," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. Association for Computing Machinery, 2002, pp. 96–107.

[14] E. A. Basha, S. Ravela, and D. Rus, "Model-based monitoring for early warning flood detection," in *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 295–308.

[15] M. Xia, Y. Dong, W. Xu, X. Li, and D. Lu, "MC2: Multimode user-centric design of wireless sensor networks for long-term monitoring," *ACM Trans. Sen. Netw.*, vol. 10, no. 3, pp. 52:1–52:30, May 2014.

[16] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 381–396.

[17] S. Evans, "Dielectric properties of ice and snow—a review," *Journal of Glaciology*, vol. 5, no. 42, pp. 773–792, 1965.

[18] X. Li, X. Cheng, R. Yang, H. Zhang, J. Zhang, F. Hui, and F. Wang, "A multi-interface ice and snow remote monitoring platform in the polar region," *IEEE Sensors Journal*, vol. 14, no. 11, pp. 3738–3744, Nov 2014.

[19] K. Martinez, R. Ong, and J. Hart, "Glacsweb: a sensor network for hostile environments," in *2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004.*, Oct 2004, pp. 81–87.

[20] A. Lindgren, A. Doria, J. Lindblom, and M. Ek, "Networking in the land of northern lights: Two years of experiences from DTN system deployments," in *Proceedings of the 2008 ACM Workshop on Wireless Networks and Systems for Developing Regions*, ser. WiNS-DR '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 1–8.

[21] K. Fall, "A delay-tolerant network architecture for challenged internets," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 27–34.

[22] J. Hester and J. Sorber, "The future of sensing is batteryless, intermittent, and awesome," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '17. New York, NY, USA: Association for Computing Machinery, 2017.

[23] H.-S. Kim, M. P. Andersen, K. Chen, S. Kumar, W. J. Zhao, K. Ma, and D. E. Culler, "System architecture directions for post-SoC/32-bit networked sensors," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 264–277.

[24] Telenor, "Dekning for IoT på 4G." [Online]. Available: https://www.telenor.no/bedrift/iot/dekning/

[25] C. Haster, "The design of littlefs," Document in littlefs source tree, ARM Ltd. [Online]. Available: https://github.com/ARMmbed/littlefs/blob/master/DESIGN.md

[26] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *2017 IEEE International Systems Engineering Symposium (ISSE)*, 2017, pp. 1–7.

[27] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos, "A survey of information-centric networking research," *IEEE Communications Surveys Tutorials*, vol. 16, no. 2, pp. 1024–1049, 2014.

[28] D. Tarr, E. Lavoie, A. Meyer, and C. Tschudin, "Secure Scuttlebutt: An identity-centric protocol for subjective and decentralized applications," in *Proceedings of the 6th ACM Conference on Information-Centric Networking*, ser. ICN '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–11.

[29] *Energizer L91 "Ultimate Lithium" Product Datasheet*, Energizer, form No. L91GL1218. [Online]. Available: https://data.energizer.com/pdfs/l91.pdf