MC^2: Multi-Mode User-Centric Design of Wireless Sensor Networks for Long-Term Monitoring

MING XIA, Zhejiang University of Technology
YABO DONG, Zhejiang University
WENYUAN XU, University of South Carolina
XIANGYANG LI, Illinois Institute of Technology
DONGMING LU, Zhejiang University

Real-world, long-running wireless sensor networks (WSNs) require intense user intervention in the development, hardware testing, deployment, and maintenance stages. A majority of network design is network-centric and focuses primarily on network performance, e.g., efficient sensing and reliable data delivery. Although several tools have been developed to assist debugging and fault diagnosis, it is yet to systematically examine the underlying heavy burden that users face throughout the lifetime of WSNs. In this paper, we proposed a general Multi-mode user-CentriC (MC^2) framework that can, with simple user inputs, adjust itself to assist user operation and thus to reduce the users' burden at various stages. In particular, we have identified utilities that are essential at each stage and grouped them into *modes*. In each mode, only the corresponding utilities will be loaded, and modes can be easily switched using the customized MC^2 sensor platform. As such, we reduce the run-time interference between various utilities and simplify their development as well as their debugging. We validated our MC^2 software and the sensor platform in a long-lived microclimate monitoring system deployed at a wildland heritage site, Mogao Grottoes. In our current system, 241 sensor nodes have been deployed in 57 caves, and the network has been running for over five years. Our experimental validation showed that the MC^2 framework shortened the time for network deployment and maintenance, and made network maintenance doable by field experts (in our case historians).

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*User-centered design*

General Terms: Design, Experimentation, Human Factors, Performance

Additional Key Words and Phrases: Wireless sensor networks, user-centric design, MC^2 framework

ACM Reference Format:
Ming Xia, Yabo Dong, Wenyuan Xu, Xiangyang Li, and Dongming Lu, 2013. MC^2: Multi-mode user-centric design of wireless sensor networks for long-term monitoring. *ACM Trans. Sensor Netw.* 9, 4, Article 39 (March 2010), 30 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

A preliminary result of this work was appeared in *ACM SIGMOBILE Mobile Computing and Communications Review* [Xia et al. 2010].
This work was supported in part by NSF under grants CNS-0923313, GEO-1124657, CNS-0832120, CNS-1035894 and ECCS-1247944, the National Natural Science Foundation of China under grants 61170216 and 61228202, the Natural Science Foundation of Zhejiang Province under grant Y1110649, the Research Program of the Department of Science and Technology of Zhejiang Province under grant 2012C33073.
Author's address: M. Xia (corresponding author), College of Computer Science and Technology, Zhejiang University of Technology; email: xiaming@zjut.edu.cn; Y. B. Dong and D. M. Lu, College of Computer Science and Technology, Zhejiang University; W. Y. Xu, Department of Computer Science and Engineering, University of South Carolina; X. Y. Li, (Current address) School of Software and TNLIST, Tsinghua University.

1. INTRODUCTION

Wireless sensor networks (WSNs) have changed the way people interact with the physical world. Without frequent field visits, scientists are able to glean real time data over a wide variety of environments, especially from remote places where no power or communication infrastructure is available. Considerable effort has targeted at building reliable data collection and delivery mechanisms using low-cost and fault-prone sensor devices. Essentially, the design principle of those approaches is *network-centric* [Estrin et al. 1999], attempting to design reliable yet energy-efficient strategies to sense, process, and relay data using wireless communication. Such a design principle certainly plays an important role towards building autonomous sensor networks. However, as WSNs have been widely used by the experts in various disciplines and been commercialized as commodity devices, developers can no longer afford to design, deploy, and maintain networks themselves. As a result, the network design has to consider an important aspect that affects the successful deployment and long-lived operation of WSNs: *users*, which include field experts, technical support staff, network developers, etc. Our experiences [Xia et al. 2012] in building long-term microclimate monitoring WSNs in a wildland cultural heritage site, Mogao Grottoes [Getty 2010], as well as experiences reported by other studies [Bai et al. 2009; Mainwaring et al. 2002; Buonadonna et al. 2005; Dyo et al. 2010] have revealed heavy user involvement and distinct operational requirements at various stages of WSNs' entire lifespan. Thus, it is advantageous to take a *user-centric* viewpoint for system design.

Heavy User Intervention. In WSN systems, the level of user intervention involved in various stages of the entire lifespan of a WSN is significantly heavier than what researchers desire. As shown in Figure 1, a typical life cycle of a WSN consists of five stages: *development*, *hardware testing*, *deployment*, *network operation*, and *maintenance*. Except for the network operation stage, all other stages involve user intervention, and those users may have little knowledge about networks or sensors, making it challenging to perform tasks. For instance, quality control (QC) personnel in a hardware factory may test hardware; technical support staff may deploy or diagnose WSNs, and field experts may maintain the deployed WSNs or even perform simple fault diagnosis.

Known deployment practice, such as redundant [Deb et al. 2003] or random deployment [Akyildiz et al. 2002], is not always applicable as a means to reduce the amount of time or effort for deployment, because of the environmental restriction. For instance, in heritage sites, the available locations for mounting sensor nodes are limited due to the concern of damaging original scenes.

For maintenance in long-running WSNs, system faults caused by battery depletion or hardware failures are common. Although it is desirable to have networks gracefully recover themselves in the presence of network exceptions, lack of redundant backup nodes means that maintenance, such as battery replacement, sensor calibration, abnormal-symptom diagnosis, or network hardware repair, has to be handled by users. The overhead of such manual maintenance could be prohibitive in a network with a large number of nodes, and a user-centric framework that can facilitate such operation is needed.

Different User Operational Requirements at Various Stages. An increasing amount of effort has been devoted to address user-related issues. For instance, NodeMD [Krunic et al. 2007] and PAD [Liu et al. 2008] can assist fault detection and diagnosis; Deluge [Hui and Culler 2004] and SDRP [He et al. 2012] provide over-the-air programming (OAP); WASP [Bai et al. 2009] is a user-friendly programming language designed for users with little programming experiences. However, each of those research projects addresses challenges caused by the heavy user intervention in some

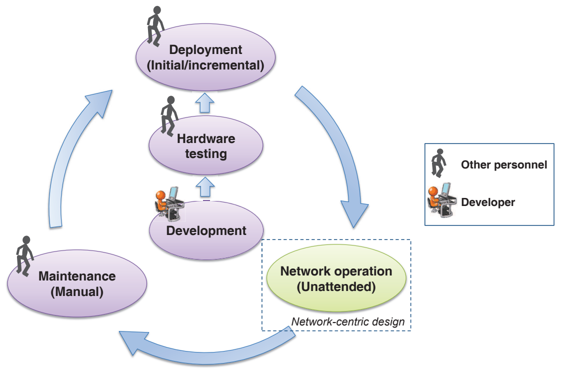


Fig. 1. The life cycle of a WSN (other personnel include QC personnel, technical support staff and field experts).

stages not all of them. In this paper, we propose a user-centric framework that is able to address various user-related challenges arising at *all* stages throughout the lifetime of WSNs, and the framework is extensible to accommodate emerging issues.

The design of a general-purpose user-centric framework is challenging, because the users' requirements are rather different at various stages and some of them may be interfering with each other. For instance, the primary focus of the network operation stage is to fulfill the application's requirement (e.g., microclimate monitoring) with minimum energy consumption. The maintenance oftentimes requires to obtain detailed current status of each node as soon as possible for identifying what causes faults, which is difficult if the network is running at a low duty cycle.

Furthermore, as the program size and complexity increase, so does the difficulty of debugging and managing correct control flow for tiny embedded systems [Dunkels et al. 2006]. A faulty application or program module may monopolize the CPU and prevent the node from processing any further control requests [Gu and Stankovic 2006].

To balance the conflicting relationship between comprehensiveness and complexity, we designed a general Multi-mode user-CentriC (MC^2) framework, which will load a subset of utilities (e.g., programs) according to the current *mode*. In particular, we identified utilities that are essential in each stage and grouped them into one or several modes. Each utility is loaded only if it belongs to the current mode. As such, we reduce the number of concurrent utilities. For instance, the over-the-air programming utility shall be loaded during hardware testing or maintenance but not in the regular network operation stage. To switch between modes according to users' need, a user-friendly input means is required. Since no existing off-the-shelf sensor platforms can take users input easily, we designed our own hardware. To our best knowledge, this is the first attempt to design a user-centric framework that can quickly organize itself to reduce the level of user intervention required at each stage. We believe this multi-mode user-centric framework can be widely used in many long-running sensor networks. The main contributions of this work are as follows:

— We emphasized the necessity of the user-centric design principle, based on the experiences gained from our early in-situ experimental exploration. The goal of user-centric design is to reduce the level of user-intervention in each stage of WSN life cycles.
— Guided by the user-centric design, we devised the MC^2 framework, which can tailor itself to meet users' requirements in different stages of the WSNs' lifespan.

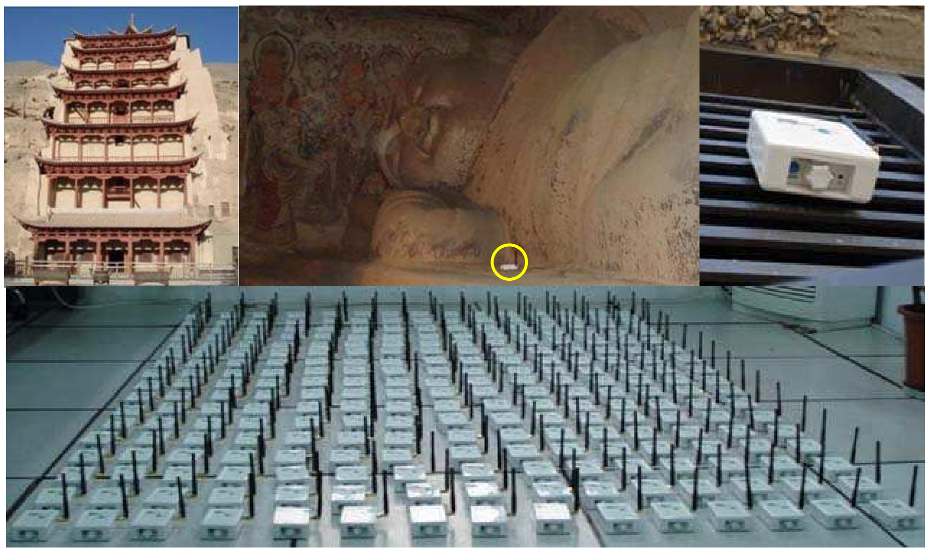


Fig. 2. Microclimate monitoring at Mogao Grottoes: [top left] a representative building, [top middle] a sensor deployed inside Cave C158, [top right] a sensor deployed at the cave entrance, [bottom] 300 sensor nodes.

— To facilitate the mode switching of the MC^2 framework, we developed supporting software and hardware: the software component of the MC^2 framework, reusable sensor nodes, and a deployment assistant, which we call SensorMate.
— We have validated the MC^2 framework in a microclimate monitoring system deployed at Mogao Grottoes. Our validation effort showed that the MC^2 framework shortened network deployment and maintenance time, and made the network maintenance manageable even by field experts.

The user-centric design principle is motivated by our ongoing microclimate monitoring project deployed at Mogao Grottoes, a world heritage site containing 492 decorated caves with murals and sculptures, as shown in Figure 2. We have installed over two hundred sensor nodes to measure temperature, humidity, carbon dioxide (CO_2) density, etc. Those measurements are used to study the impact of microclimate on mural deterioration, and they are also used to detect harmful environmental changes to enforce site conservation policies.

Early in our project, we found that many well-known approaches are not applicable. For instance, node redundancy is not practical due to our limited budget and the request to minimize the impact to the original scene. Random deployment is not an option, because the location of each sensor node must be carefully selected to avoid damaging already deteriorated murals, and to preclude easy access from tourists. During deployment, we observe heavy multi-path effects causing highly unpredictable radio propagation. Serious radio irregularity, combined with the constraint of low redundancy, make our deployment extremely laborious. Further, the manual maintenance during the network operation phase demands frequent field visits, requiring a roughly 4-day traveling time plus the unpredictable time that needs to be spent on-site. Without the MC^2 framework, it is time-consuming to deploy and to maintain the network.

The remainder of the paper is organized as follows. We give an overview of the problems users may encounter during the hardware testing, deployment and maintenance stages in Section 2. Then, we present the MC^2 software architecture in Section 3, and user-centric hardware design in Section 4. Section 5 discusses a case study of applying the MC^2 framework in a long-term microclimate monitoring system which is deployed at Mogao Grottoes. Finally, we review the related work in Section 6 and conclude the paper in Section 7.

2. PROBLEM OVERVIEW

In total, we deployed our systems in two rounds — the first round adopted the network-centric design, and the second round used the user-centric design. In this section, we outline the type of applications that we focus on in this paper, and share the experiences and lessons learned from our initial explorative deployment, which suggest the need of a user-centric design.

2.1. Our Application Paradigm

In this paper, we focus on the sensing applications with the following features:

Long Running. An important class of sensing applications are those monitoring a valuable asset. Such asset monitoring applications are expected to operate for a long time. For instance, our microclimate monitoring system at Mogao Grottoes should continuously provide real time measurement as long as the site is open to tourists, which could be twenty years or more. We note that within the lifetime of the network (e.g., twenty years), node failures such as battery exhaustion will occur. Thus, users have to maintain the network periodically to sustain long-term monitoring.

Low Redundancy. Applications are deployed with low redundancy with two reasons. 1) Limited budget. The price of sensor nodes is far from predicted few dollars each piece, beyond the budget of WSN projects in developing countries. 2) Limited deployment locations. The available spots to place sensor nodes are limited. Low redundancy means that when a sensor node fails, a standby node is not always available to mitigate the impact of the node failure. Thus, quick run-time diagnosis and fault recovery become critically important.

Real Time. Many sensing applications require obtaining the sensing data in real time. For instance, our microclimate monitoring system was required to report data within a minute, so that the caves with dangerous levels of humidity and CO_2 can be closed for tourists immediately.

Scalable and Extensible. In a long-running system, the network may require to evolve as users demand to collect new types of data or to cover new areas. For instance, we were requested to measure tourist numbers after our system has been stably operated for one year. The possibility of incremental deployment over time urges us to take extensibility and scalability into consideration at the initial design phase.

Deployed in Harsh Environment. WSN systems are frequently deployed in harsh environments, e.g., in an active volcano [Werner-Allen et al. 2006], or in rivers [Basha et al. 2008]. The environment of our system is a desert. Such a harsh environment can quickly wear out the electronic devices on sensor nodes, e.g., relative humidity and temperature sensors are typically vulnerable to dust. Therefore, hardware must be customized to suit for targeted sites and be maintained periodically after deployment.

Additionally, the deployment environment may have unusual radio propagation properties. For instance, our microclimate monitoring project is deployed inside caves at Mogao Grottoes. The thick rocks between caves make the nodes located in different caves impossible to communicate. In addition, the shapes of caves, and the number of people and their positions in caves create a highly irregular radio environment, making the network deployment challenging.

2.2. First Round Exploration

In our first round exploration, we deployed 40 customized sensor nodes in 10 caves at Mogao Grottoes, and performed a one-year in-situ study. This first attempt adopted the network-centric design principle, and it primarily focused on the correctness of network operations. We defer our discussion on hardware design to Section 4, and report the experiences we encountered in sequence.

Hardware Testing. After developing hardware and software for climate monitoring in the development stage, we started the hardware testing phase, which consists of two tasks: verifying the correctness of hardware and uploading the customized software to all nodes. In total, we had to upload two different programs to each node, a program designed to perform hardware self checking and a program for the sensing application. The most time-consuming step in this phase was uploading codes to each node, because it requires to open enclosures, plug/unplug the sensor nodes to/from the programming board for each uploading, and finally close the node enclosures. We spent approximate 9 hours to ensure that all 40 nodes were ready for deployment.

Deployment. The main tasks in the deployment stage include sensor calibrations to ensure accurate sensor readings, sensor node placement to guarantee good network connectivity, and burn-in to detect any early in-use system failures.

Many types of sensors require in-site calibration before placing in service. For instance, the Telaire 6004 CO_2 sensor [Telaire 2004] in our system requires calibration once the elevation changes. The 1300-meter elevation difference between our lab and Mogao Grottoes demands calibration. To calibrate one Telaire 6004 CO_2 sensor, we have to unplug the sensor from the original node, plug it to a dedicated circuit board (which is designed to bridge the communication between a CO_2 sensor and a laptop through an RS232 serial port), perform the calibration, and repack the sensor. It took us about one hour to calibrate only four CO_2 sensors, with most of the time spent in unscrewing and screwing the enclosures.

Deploying sensor nodes in caves turned out to be difficult because the metal doors, thick stone walls, and central columns form an indoor environment full of reflections, multipath, and other radio propagation disturbances. Meanwhile, the communication quality between nodes was affected by the location and the number of people (e.g., tourists) inside caves, as well as by whether the metal doors at the entrances of the caves were open or closed. The resulted communication range of nodes became highly unpredictable inside caves, which, combined with the limited number of deployable locations, forced us to take extra effort to assure stable network communication. During deployment, each node underwent an initial placement and a burn-in process.

During the initial placement phase, we deployed as few nodes as needed for satisfying the monitoring requirements, and then tested the communication quality of each node by measuring the data delivery ratio at the normal working cycle (i.e., 1 minute). We created a few scenarios for evaluating the network connectivity of the deployment: no people or a few people standing in front of the deployed nodes; the metal door was closed or open. If the network connectivity was unstable in one of the test scenarios, we re-positioned the nodes until finding a satisfying deployment, or until exhausting all possible spots and having to insert an extra node. Because of the low data rate, although we tried as many test scenarios as the time permitted during the deployment phase, we were only able to test a small subset of possible scenarios that will occur in the network operation phase. On average, we spent 40 minutes in deploying a few nodes for each cave.

Shortly after the initial deployment phase, we found out that the network in many caves failed. In our system, sensor nodes will retransmit the data if no acknowledgement is received. After trying to retransmit for the maximum allowed times, the sensor nodes will cache the data, wait until the next sensing cycle and try again. As a result, an unstable network connection will introduce a large data collection delay, which not only causes wasted energy on retransmissions but also prevents historians from closing caves for site protection in real time. We show a node's data collection delay in

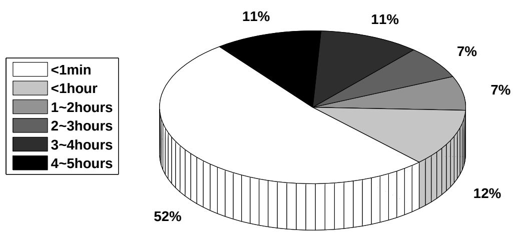


Fig. 3. Data delay distribution of an inappropriately deployed node.

Figure 3. Only 52% of the time, the communication was stable enough for the node to report its data within a minute. 48% of the time, the communication of the network has been constantly disturbed. The longest delay could be up to 5 hours. Such communication quality difference between the deployment phase and the network operation phase was caused by the insufficient deployment test.

To assure the network connectivity, burn-in process is necessary, which is essentially a prolonged node placement phase for detecting any early in-use system failures. During this processing, we were forced to revisit the cave frequently to adjust the placement of the nodes that had unstable communication quality. This process became laborious and costed us 3 days to adjust the node placement so that the connectivity became stable in our first round deployment.

Maintenance. In our explorative system, we designed a preliminary node monitoring system which notifies users once a node failure occurs without reporting node status information. However, status information is essential in recovering from the node failure. For instance, a node may stop reporting data to the data server because of a failed flash memory or because of the poor link quality. Replacing the failed node with a new one can solve the problem caused by a failed flash memory but cannot address the problem of poor link quality, which requires to re-position the node or to add an extra node. Without tools to collect real time node status information, it is impossible for a field expert (e.g. historians in our case) to diagnose the fault type or to repair the system, and we had to fly to Mogao Grottoes in person for diagnosis and repairing whenever a system fault occurs. The travel overhead was overwhelming and the repair operation frequently took a long time because of the lack of tools to collect node status information.

In a long-lived WSN, besides the aforementioned fault-recovery maintenance, another type of maintenance that has not receive its due attention is periodical maintenance. Periodically, the drained batteries of sensor nodes have to be replaced, sensors need to be calibrated, and the worn-out sensors have to be replaced to assure sensing accuracy. All those procedures can be time-consuming using the existing sensor framework.

2.3. Utility Wish List

The intense user intervention we encountered in testing, deploying, and maintaining our explorative network calls for a user-centric framework for WSNs. The goal is to reduce the workload of developers, field experts, QC personnel, or technical support staff, rather than only focusing on improving the network performance. Thus, it is desirable to have the following utilities supported by the framework:

Local Hardware Verification. The hardware verification procedure we conducted in the hardware testing phase turns out to be helpful to detect node failures in an early stage. We envision that in a large-scale WSN, the QC personnel in hardware manufactory but not developers will verify hardware. Because the QC personnel may have little knowledge on WSNs, the user-centric framework should integrate a hardware verification utility that can automatically check the correctness of all hardware components and immediately return a short summary of verification results.

Wireless Code Distribution. The most time-consuming step in the hardware testing phase is to upload programs to each sensor node, because it requires a serial of mechanical operations, e.g., screwing enclosures or plugging nodes to programming boards. To avoid such laborious yet low-tech operations, propagating code over the air is desirable to accelerate the speed of code distribution.

Wireless Configuration. In-situ configuration gives users the flexibility to adjust networks according to field situation. For instance, users may need to configure each node with a unique ID or a different working cycle. Similar to code uploading, to avoid arduous mechanical jobs when deploying networks, the user-centric framework should support node configuration through wireless communication.

Online Calibration. Sensor calibration should be performed without dismounting the sensing board. Towards this goal, the wireless modules should be able to directly access calibration-related parameters and to perform calibration. For instance, we should customize our software and hardware to calibrate Telaire 6004 CO_2 sensors.

Fast Deployment Validation. The amount of time required to search for a good spot for placing a node is large. To speed up the deployment, a special software module targeting at testing the deployed network stability within a short period of time should be supported.

Remote Fault Diagnosis. In the network operation stage, it is desirable to periodically monitor networks and notify users network failures remotely. Because of the constrained energy budget or inability to communicate in a faulty scenario, only a limited amount of status information can be reported remotely. Nevertheless, remote fault diagnosis can help users to discover network failure and to plan for the on-site failure repairing.

On-Site Fault Diagnosis. Due to the limited node status information, remotely identifying fault types can be challenging. To assist failure identification, the on-site fault diagnosis module allows an on-site user to collect a status report with greater details for a thorough diagnosis.

2.4. Utility Integration

The above-mentioned utilities exhibit diverse behaviors and may have conflicting requirements. For instance, an energy-efficient monitoring program is typically run at a low duty cycle for data sampling and reporting. The utility for fast deployment validation, however, demands sensor nodes to communicate at a much higher rate to reduce the time for testing link quality. Additionally, utilities that support wireless code distribution, wireless configuration, online calibration, and on-site fault diagnosis require the sensor nodes to respond to the user's queries immediately (demanding nodes to keep awake), yet they are not needed during the network operation stage. Thus, it is difficult and complicated to have all utilities running simultaneously.

To integrate all utilities seamlessly without sacrificing the network normal operation performance, we developed a $\underline{\text{M}}$ulti-mode user-$\underline{\text{C}}$entri$\underline{\text{C}}$ (MC^2) framework, which organizes utilities according to *modes*. The MC^2 framework can cycle through various modes based on users' input, but can only be in one mode at a time. In each mode, only those corresponding utilities with similar behaviors and requirements are executed. To maneuver mode-switching and utility-loading, the MC^2 framework consists of MC^2

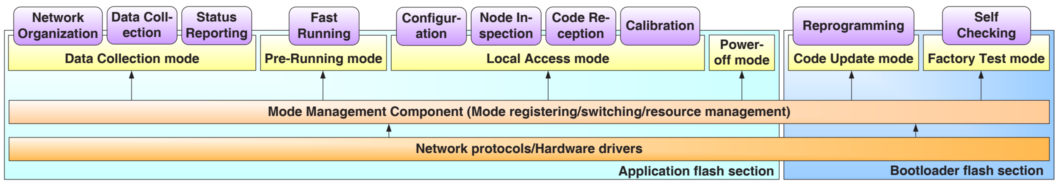


Fig. 4. The high-level software architecture of the MC^2 framework.

software and matching hardware. The hardware includes customized sensor nodes, a TestKit, and a handheld device, which we call *SensorMate*. SensorMate serves as the bridge to facilitate the direct user-node interaction in the field. We present the design details of the MC^2 software in Section 3, and discuss the hardware component of the MC^2 framework in Section 4.

3. MC^2 SOFTWARE

The MC^2 software component provides a user friendly interface between the hardware and the users, and it incorporates many utilities that can reduce the level of user intervention throughout the lifespan of WSNs. One natural way to incorporate all utilities is to implement each utility as a module and build one program that contains all modules. However, such a strategy makes all modules share the same memory and have the same privilege. As a result, the execution of those modules/utilities will be highly interleaved, which makes it difficult to manage the correct control flow of the program. Such a task is especially difficult in sensor platforms where event-driven programming model is typically used. Developing one program containing all utilities is further complicated by the fact that some utilities behave distinctively different, while some may share similar behaviors. Thus, the key idea of the MC^2 framework is to provide an abstracted container, which we call *mode*, to group utilities with similar behaviors, and a general *Mode Management Component (MMC)* to control mode switching. As such, unrelated utilities are fully decoupled to avoid interference between each other, and we can target at maximizing their run-time efficiency individually without compromising the performance of others.

3.1. Modes

Instead of grouping utilities purely based on the stage, we classify them based on factors that will affect their behaviors, e.g., the duty cycle and boot-related characteristic. Typically, a micro controller unit (MCU), such as ATmega128L [Atmel 2013], includes an application flash section and a bootloader flash section. When booting up, the MCU first executes the programs stored in the bootloader flash section and then loads programs stored in the application flash section. Reprogramming codes in the bootloader flash section requires a special hardware, e.g. JTAG, while the programs stored in the application flash section can be reprogrammed without any dedicated devices. Thus, the program that may require updating must be stored in the application section.

Based on the required characteristics, we group the utilities into five modes: Data Collection, Pre-Running, Local Access, Code Update, and Factory Test. Under each mode, one or a few modules will be running simultaneously to support multiple utilities, as depicted in Figure 4. Additionally, we introduce a special mode called Power-off mode, which does not correspond to any utility, but provides a useful mode for transportation. We summarize the relationship between modes, utilities and stages as well as the software configuration and hardware resources for each mode in Table I. We note that we developed one module for each utility except for the wireless code distribution utility, which is implemented in two modules, and we will discuss the reason in

Table I. The relationship between modes, utilities, and stages. The software configuration and hardware resources used by each mode.

Mode	**Utility**	**Stage**	**Software configuration**		**Hardware resources**
			Routing	**Duty cycle**	
Data Collection	Monitoring	Network operation	Multi-Hop, Tree	Low	Sensors, RF, etc.
	Remote fault diagnosis	Network operation			
Pre-Running	Fast deployment validation	Deployment	Multi-Hop, Tree	High	Sensors, RF
Local Access	Wireless code distribution	Hardware testing	Single-Hop, Star	100%	Sensors, RF, External storage, etc.
	Wireless configuration	Deployment			
	Online calibration	Deployment, Maintenance			
	On-site fault diagnosis	Maintenance			
Code Update	Wireless code distribution	Hardware testing	N/A	100%	External storage
Factory Test	Local hardware verification	Hardware testing	Single-Hop, Point-to-Point	100%	Sensors, RF, External storage, Serial port, etc.
Power-off	N/A	Prior to deployment	N/A	0%	None

Section 3.1.4. We emphasize the difference between modes and modules — a module is a program implementing a utility, and a mode is a virtual container that supports concurrent execution of one or more modules.

3.1.1. Data Collection Mode. The Data Collection mode is designed for the normal network operation stage. In this stage, a sensor node typically runs at a low duty cycle to conserve energy, and reports sensed data along a routing tree in a multi-hop manner. Many modules belonging to this stage are application specific. For instance, our microclimate monitoring application at Mogao Grottoes requires to sense and report climate-related data once every minute, but a road traffic monitoring application may require the network to continuously monitor the drive-by cars. Since the application requirements may change over time and these modules may need to be reprogrammed from time to time, they are stored in the application flash section.

The Data Collection mode includes three modules: `Network Organization`, `Data Collection`, and `Status Reporting`. The `Network Organization` module handles routing, time synchronization, and other network related operations. The `Data Collection` module samples, transmits, and relays data towards the data server. These two modules are designed to realize the functionality of the applications, while the `Status Reporting` module is designed for remote fault diagnosis. It periodically reports the node status to the data server, and the status report typically contains a hardware status summary (e.g., battery voltage and the status of sensors) and a network status summary (e.g., the parent node ID and the hop count of the node).

Essentially, most existing sensor nodes in the literature can be treated as working in the Data Collection mode only, while our MC^2 framework provides a richer set of working modes to meet various user requirements.

3.1.2. Pre-Running Mode. The Pre-Running mode is designed for assisting network deployment so that the deployed network exhibits stable communication quality. The Pre-Running mode contains one module called `Fast Running`. Similar to those modules reside in the Data Collection mode, the `Fast Running` module resides in the application flash section. This mode is essentially a fast-forwarding version of the Data Collection mode with slight differences. For instance, in the Pre-Running mode, a node will report the transmission delay at each hop, while in the Data Collection mode, only end-to-end delay is reported. Such hop-by-hop delay information can help users to identify the weak link and adjust the locations of nodes accordingly. To quantify the network stability, we use packet delivery ratio (PDR): a PDR higher than a threshold is considered stable (e.g., 97% in our case). In total, each sensor node reports the network stability in two ways: blinking the LEDs on its control panel and reporting the PDRs to the data server. Thus, we can verify the network stability either visually in the field or by analyzing the reported data at the data server. The `Fast Running` module helps us to quickly make a good deployment decision. Because the `Fast Running` module lets nodes exchange packets at a high frequency, it enables us to quickly emulate many scenarios such as tourists blocking the communication propagation path in a short period of time. Thus, the connectivity of the deployed networks will be relatively stable in the network operation stage.

3.1.3. Local Access Mode. The Local Access mode aims at incorporating all utilities that require to fetch/feed information from/to a node directly, e.g., on-site fault diagnosis or online calibration. A node in the Local Access mode will remain awake in order to respond to users' queries or to react to users' instructions immediately. We call this mode local access, because communication involved in this mode is one hop between a sensor node and SensorMate. This is different from the one in the Data Collection or Pre-Running mode, where data are delivered through a multi-hop routing tree. One-hop communication makes the protocol between sensor nodes and SensorMate simple yet reliable, e.g. independent of other nodes' status.

The Local Access mode contains four modules running simultaneously: `Code Reception`, `Configuration`, `Node Inspection` and `Calibration`. Each module handles one of the following utilities.

Wireless Code Distribution. The `Code Reception` module keeps waiting for code blocks that might arrive at the RF interface. Once it receives the code blocks distributed by SensorMate, it checks the correctness of every block and stores the block in an external nonvolatile memory. If a code block is lost or damaged, the node will blink its red LED to indicate a code reception failure, and users can rebroadcast the code. The `Code Reception` module is usually used during the hardware testing stage, the simple network communication pattern and "always on" feature of sensor nodes in this mode ensure the high efficiency in code distribution.

Wireless Configuration. The `Configuration` module is designed to fulfill the task of wireless configuration. As soon as the module receives parameters of sensor nodes wirelessly, such as network ID and duty cycle, it stores them in MCU's EEPROM, where the `Data Collection` and other modules will fetch relevant parameters. The `Configuration` module is also able to return the system current parameters to the user. We note that the configurable parameters can easily be extended by increasing editable fields inside EEPROM.

On-Site Fault Diagnosis. User can perform the on-site fault diagnosis by leveraging the `Node Inspection` module. Once received commands sent from SensorMate, this module conducts user-intended inspections on peripheral circuit components, such as flash writing and reading, sensor data sampling, etc., and reports a detailed inspection result to SensorMate.

Online Calibration. The `Calibration` module starts to calibrate after receiving the calibration command sent by SensorMate, and then returns a calibration result to SensorMate. The advantage of the `Calibration` module is that it eliminates the need to plug sensors to a dedicated device, and the module can handle most calibration calculation tasks automatically. A user only needs to provide necessary calibration parameters. For instance, the `Calibration` module for the Telaire 6004 CO_2 sensor requires taking the zero and span values for calibration. A detailed example about zero and span calibration is discussed in Section 5.2.2.

3.1.4. Code Update Mode. The Code Update mode is a special mode designed to complete the wireless code distribution process. The first step of wireless code distribution involves propagating the code blocks to a node. Once the entire codes are successfully received and stored in the external non-volatile storage space during the Local Access mode, the node will start the second step: reboot and enter the Code Update mode. This mode has one module, `Reprogramming`, which reads the code from the external storage and replaces the current program that is stored in the application flash section of MCU. We place the `Reprogramming` module in the bootloader flash section, since only routines running in the bootloader flash section can rewrite the application flash section of MCU. Unless necessary, we do not store modules in the bootloader flash section because of its limited size. Since the `Code Reception` module requires routing and MAC protocols, it is relatively large and is stored in the application flash section.

3.1.5. Factory Test Mode. The Factory Test mode is meant for hardware verification in the hardware testing stage, and it includes one module, `Self Checking` module. This module automatically conducts a comprehensive hardware check on each component of a sensor node, and returns a hardware status summary to a user. For example, to test the RF chip, the `Self Checking` module first sends several testing packets. A spectrum analyzer verifies whether the RF transmission (TX) power meets the requirements and returns an acknowledgement packet containing the verification result. Then, the `Self Checking` module turns the node into listening state, and we transmit several testing packets to the node to verify the RF receiving (RX) sensitivity. The `Self Checking` module will blink the green LED to indicate success, and blink the red LED if an error occurs in any step of verification.

We note that this module is different from the `Node Inspection` module running in the Local Access mode, since they are designed for different stages with different user focuses. In the hardware testing stage, a user typically has to verify a large number of sensor nodes, and the `Self Checking` module is targeted at maximizing the speed of retrieving compact hardware status summaries of a large collection of nodes. Thus, the `Self Checking` module is located in the bootloader flash section for fast checking. In comparison, during in-situ maintenance, a user is interested in searching for detailed hardware status to assist diagnosis. Thus, the `Node Inspection` module emphasizes on providing a highly detailed status report of the interested component in a well-organized format for easy access.

3.1.6. Power-off Mode. A node may need to stay in a deep sleep mode to reduce unnecessary power consumption when it is transported from a lab to a field. The Power-off mode is designed to assist fast node shut-down. Without this mode, to turn off a sensor node, one may need to open the enclosure to take off batteries. Later on, to resume the node, a battery installing is required. Although one can avoid opening enclosure by adding a hard switch on the enclosure, such operation is still time-consuming and not scalable, and makes the enclosure hard to seal. By introducing a Power-off mode, a user can turn off a large collection of sensor nodes via the MMC without physical contacts. Such soft switches greatly reduce the overhead involved in the hardware testing and deployment stages. To turn on a node, a user can press the button on the enclosure.

3.2. Mode Management Component (MMC) Overview

The Mode Management Component (MMC) fulfills two main functions: managing the list of supported modes of the MC^2 framework and switching the current working mode. To accomplish them, the MMC contains three key routines:

— *a mode register* for adding/removing modes and corresponding modules,
— *a mode switching manager* for changing the current working mode according to mode switching commands,
— *a mode resource manager* for managing hardware resources for each mode.

Meanwhile, the MMC accepts three types of mode switching commands from various interfaces (details will be discussed in Section 3.3):

— *internal switching* commands that are issued by a program,
— *button switching* commands that are issued by users through input devices (i.e., a hardware button),
— *wireless switching* commands that are issued by users wirelessly.

The entire workflow of the MMC consists of an initialization phase and a mode-switching phase. Once a sensor node boots up, the MMC immediately starts initialization by registering modules and modes, and records information essential for mode switching in a *mode resource table*. The mode resource table contains a complete list of modules that shall be loaded for each mode, the hardware resources that shall be used for each mode (e.g., sensors, RF, or external storage), and the software configuration information (e.g., duty cycles). One of the biggest benefits of dynamically registering modules and modes during the initialization phase is that it makes the MMC compact and mode modification flexible, i.e., modifying modes does not require to change the MMC but changing the mode register. After initialization, the MMC enters the mode-switching phase. Every time the mode switching manager receives a mode switching command, either from users or programs, the mode switching manager will stop all running modules, request the mode resource manager to revoke the allocated resources for the old mode, allocate resources required for the new mode, and load the set of modules for the new mode.

Instead of walking through a complete workflow of the MMC in the MC^2 framework at the full scope, we illustrate the overall work process of the MMC via a simplified yet representative example that supports two modes and three modules. As shown in Figure 5, during the initialization phase, Modules 1 and 2 were registered under Mode 1, and Module 3 under Mode 2. Then, both modes were registered. After initialization, either a user or the application issued a mode switching command for entering Mode 1, and the MMC set the current working mode to Mode 1 with Modules 1 and 2 running. In response to the second command for switching to Mode 2, the MMC stopped Modules 1 and 2, revoked resources for Mode 1, allocated corresponding resources for Mode 2 and loaded Module 3. At this point, the MC^2 framework entered Mode 2.

Our MC^2 framework currently supports six modes (listed in Table I). Similar to the above example, during the initialization phase, a mode resource table that records configuration information for all six modes is created when all modes and corresponding modules are registered one by one.

3.3. Mode Switching Prototype

In total, three types of the mode switching commands can initiate a mode switching: internal switching commands, button switching commands, and wireless switching commands. However, not every switching command is applicable in each mode. In

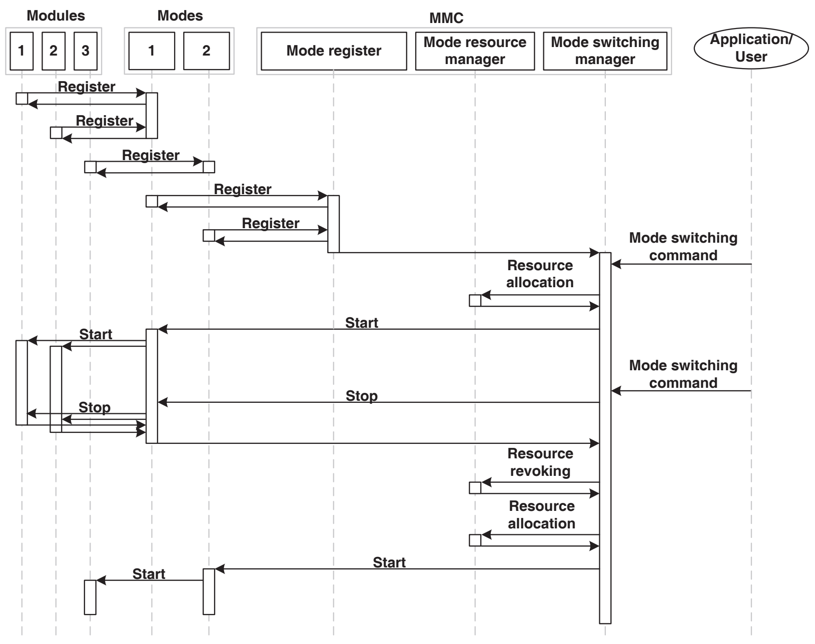


Fig. 5. A simplified workflow of the MMC in the MC^2 framework.

this section, we elaborate the relationship between mode switching commands and the applicable modes.

One of the main factors that determines whether a switching command is able to initiate a mode switch is the mode's flash section. Thus, we divide the mode switching scenarios into two categories: the set of mode switching commands that control the modes in the application flash section, and the ones controlling the bootloader flash section. Figure 6 depicts the mode switching flow chart with regard to the flash sections, and it summarizes the applicable mode switching commands for triggering switches.

3.3.1. Mode Switching Management within the Application Flash Section. The Power-off mode, Pre-Running mode, Local Access mode, and Data Collection mode are implemented in the application flash section. To switch modes among them, both the button switching and the wireless switching can be used.

Button Switching. A mode switch button is designed on the enclosure for an easy access. By pressing the button, a user can cycle through four modes. Given that the initial mode is the Power-off mode, a short pressing of the button will make the node change from the Power-off mode into the Data Collection mode. As the user keeps holding the button, the node will enter the Local Access mode and the Pre-Running mode in sequence. The user can enter the Local Access mode or the Pre-Running mode by releasing the button once the LEDs on the node indicate entering that mode. Additionally, a node can return to the Power-off mode by pressing the button. The advantage of button switching is that the node will immediately switch to the intended mode. However, a physical access to the node is mandatory, which might create scalability issues to switch a large number of nodes. To address these issues, we designed wireless switching.

Wireless Switching. Wireless switching relies on SensorMate to broadcast mode switching commands. To initiate the mode switching, SensorMate broadcasts a spe-

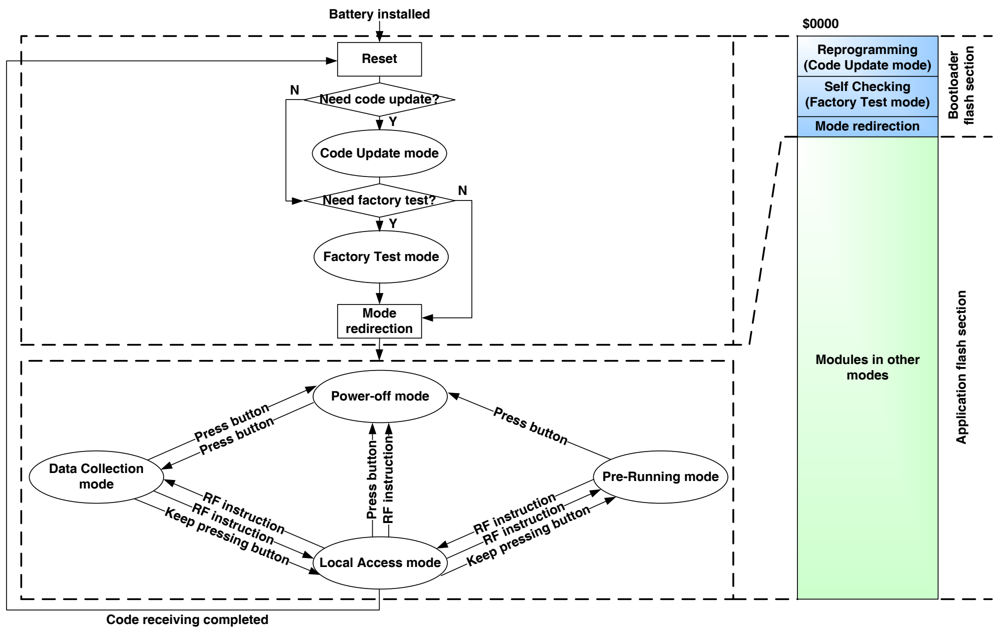


Fig. 6. The flow chart of mode switching.

cial control packet while sensor nodes are awake, e.g., in a non Power-off mode. The nodes that receive the control packet will immediately switch to the Local Access mode, and reply an acknowledgement to SensorMate. From the Local Access mode, a user can instruct a node to switch to any other three modes via SensorMate. The wireless switching is useful when nodes are difficult to access or the number of nodes is large.

During deployment or maintenance, switching between those four modes are adequate most of the time. The only exception is that after the `Code Reception` module in the Local Access mode successfully receives a new program, the MMC will mark the reprogramming flag in the external storage and reset the nodes. We now discuss how to enter the Code Update mode and the Factory Test mode.

3.3.2. Mode Switching Management within the Bootloader Flash Section. Both the Code Update and Factory Test modes are stored in the bootloader flash section. To enter either mode, a node reset is needed. Both the battery installation and code update will cause a node to reset. After being reset, the node will switch its mode according to the predefined internal switching sequence. That is, the node will try to enter the mode implemented in the bootloader flash section one by one. The MMC first checks if reprogramming is required by examining the reprogramming flag in the external storage. If the reprogramming flag is set, the node will enter the Code Update mode to finish the reprogramming task. Otherwise, the MMC determines whether the node need to enter the Factory Test mode by checking whether a special control pin on the mother board of the node has been set to high. The TestKit, which will be introduced in Section 4, can be connected with the mother board to set the pin to high. A typical hardware arrangement is displayed in Figure 7(b).

Once finishing executing the modules stored in the bootloader flash section, the MMC will enter the application flash section. In particular, the MMC will execute a special routine, *mode redirection*, to determine which mode in the application flash

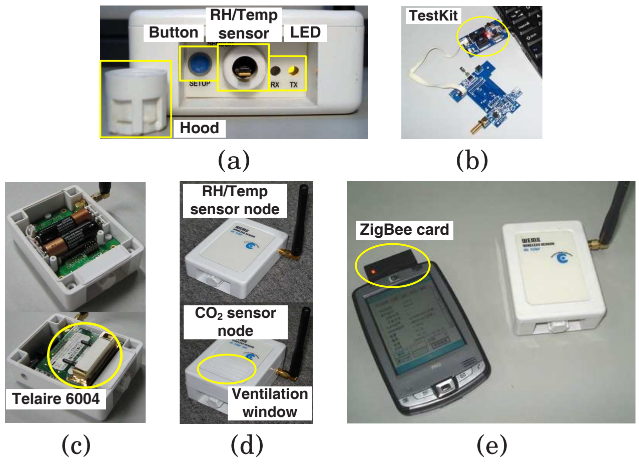


Fig. 7. Customized MC^2 hardware: (a) the control panel of a sensor node, (b) TestKit, (c) sensor nodes' internal structures, (d) sensor nodes' appearances, and (e) SensorMate (left).

section it should redirect to. The MMC stores the node current mode in a nonvolatile storage once a mode switching in the application flash section occurs. As such, the mode redirection routine will resume the same working mode prior to resetting. This routine is designed to reduce the manual operation of battery replacement, since in most cases nodes are expected to continue running under the same old mode after battery replacement.

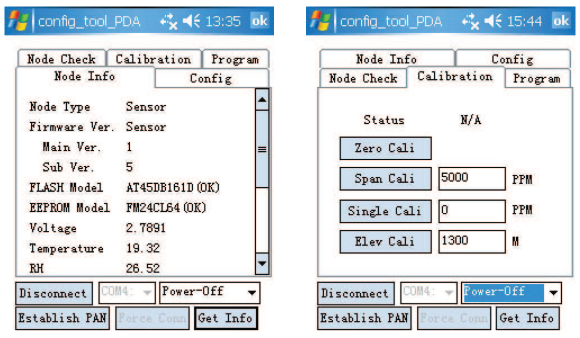
4. MC^2 HARDWARE

Existing WSN hardware designs involve sealing the node within an enclosure to achieve better protection against a harsh wildland environment, but typically at the cost of operation convenience. In contrast, we take the viewpoint that the WSN hardware should provide a user-friendly operational interface while assuring the device resilience against a harsh environment. We design our WSN hardware to support the MC^2 software and thus to reduce the amount of time and effort that users have to spend in hardware testing, deployment, and maintenance.

4.1. Sensor Node

One of the key features that makes our sensor nodes user-friendly is the control panel on the weatherproof enclosure, as shown in Figure 7(a). In particular, we have included two unique features. Firstly, the control panel contains a button for local mode switching, and two LEDs for displaying the current mode and data transmission results (succeeded/failed). Together, the mode switching button and LEDs facilitate users to maneuver through the MC^2 working modes without opening the enclosures.

Secondly, we have designed a layout that can facilitate pluggable sensor integration, which makes sensor replacement and encapsulation easy. Our system uses humidity and temperature (RH/Temp) sensors. The accuracy of RH/Temp sensors will gradually degrade and need to be replaced every few years. Replacing RH/Temp sensors at a large scale can be laborious. Thus, we chose Sensirion's SHT75 [Sensirion 2011b] sensor module instead of the most widely used SHT11 [Sensirion 2011a], since SHT75 provides a 4-pin, fully pluggable interface while SHT11 must be soldered on a circuit board. This 4-pin interface enables us to easily replace the sensor module. In addition, we placed the plug interface of SHT75 on the control panel and designed a dust-resistant, ventilated hood to protect it, as shown in Figure 7(a). As such, to re-



Node Type	Sensor
Firmware Ver.	Sensor
Main Ver.	1
Sub Ver.	5
FLASH Model	AT45DB161D (OK)
EEPROM Model	FM24CL64 (OK)
Voltage	2.7891
Temperature	19.32
RH	26.52

Status N/A

Span Cali 5000 PPM

Single Cali 0 PPM

Elev Cali 1300 M

(a) (b)

Fig. 8. The screen snapshots of SensorMate: (a) node information summary, and (b) sensor calibration GUI.

place the SHT75 sensor, a user only needs to open the hood rather than unscrewing the enclosure, greatly reducing the maintenance time.

In addition to common humidity and temperature sensors, many monitoring applications may require to use a special sensor. For instance, the microclimate monitoring at Mogao Grottoes requires CO_2 sensing. Thus, we integrated Telaire 6004 digital CO_2 sensor module [Telaire 2004] by connecting it to the sensor node's mother board through a 12-pin connector, as shown in Figure 7(c). Such a pluggable sensor integration reduces the difficulty of encapsulation. To allow the air pass through the CO_2 sensors, we added a ventilation window to the enclosure specially for CO_2 sensors, as shown in Figure 7(d).

As far as the architecture of the node mother board is concerned, our node architecture is similar to the one of MICAz [Crossbow 2007], except the following improvements to support a user-friendly design: 1) a customized shape for the specially designed enclosure; 2) customized interfaces for sensor module connections; and 3) integrating AT45DB161D (a 16-Mbit flash [Adesto 2013]) and FM24CL64 (a 64-Kbit F-RAM [Ramtron 2011]) to support wireless software programming and data caching.

4.2. Testing and Diagnosis Tools

To provide a friendly user-node interaction, we have developed two pieces of hardware. One is SensorMate and the other is a TestKit.

SensorMate. We have implemented SensorMate on a HP iPAQ PDA (hx2490c [HP 2013]). We chose a PDA over a laptop, because it is lighter and its battery lasts for a longer time. Meanwhile, it provides an easy-to-operate, touch-screen-based user interface. To enable the PDA to communicate with sensor nodes directly, we extended the PDA by attaching a Telegesis's ZigBee card [Telegesis 2013] to its CF slot, as shown in Figure 7(e). Additionally, we developed a suite of GUI to facilitate user-node interaction. Figure 8 demonstrates two sample GUI snapshots of SensorMate: Figure 8(a) displays the node information page which contains the node's running status and parameters. Figure 8(b) illustrates an interface for sensor calibration.

TestKit. We designed a TestKit for two purposes: to set a node to the Factory Test mode, and to bridge the communication between a node and a laptop. The TestKit consists of a customized 12-pin wire and a converting board with a USB port. The converting board is essentially a USB/UART converter that connects to a laptop via a USB port and supplies UART output through the customized 12-pin wire to a sensor node. When the 12-pin wire is connected to a sensor node, the factory test pin on sensor node will be set to high, and the node will immediately be reset and enter the Factory Test mode. Furthermore, the sensor node can output testing results to a terminal.

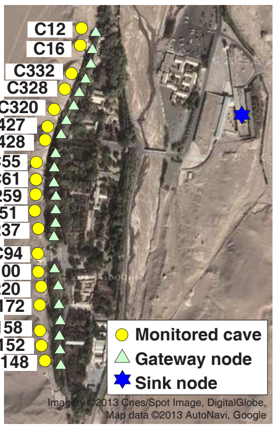


Fig. 9. The geographical layout of the deployed system (a partial view, original map courtesy of Google [Google 2013]).

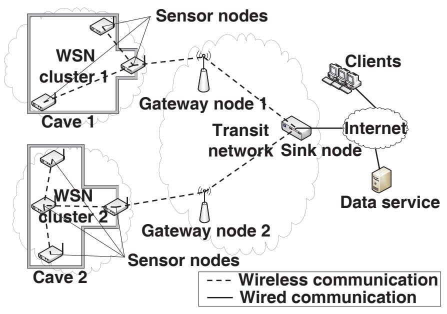


Fig. 10. Communication architecture.

5. CASE STUDY: THE MICROCLIMATE MONITORING AT MOGAO GROTTOES

In this section, we discuss our experiences in building a long-term system for Mogao Grottoes, employing the proposed MC^2 framework. The historic relics of Mogao Grottoes are deteriorating due to the inappropriate microclimate inside caves [Shi and Zhang 1997]. Our project aims to set up a long term monitoring system to collect real-time microclimate measurements (e.g., temperature, humidity and CO_2 density) that are used for scientific studies and site protection.

Battery-powered WSN technology becomes almost the only applicable solution to monitor the microclimate at Mogao Grottoes, because power and communication wiring inside caves is prohibited and solar power is not available inside closed caves. The requirements of continuous and large scale monitoring, complex landform, and minimal number of nodes in caves make the deployment at Mogao Grottoes a challenging as well as a representative long-term environmental monitoring application. We illustrate that the user-centric design helps to reduce the burden of users despite of the challenges.

We will briefly introduce our communication architecture in Section 5.1. We present our experiences on time saving by using the MC^2 framework in Section 5.2, and show the overhead and performance of the customized MC^2 framework in Section 5.3.

5.1. Communication Architecture

All caves of Mogao Grottoes are not naturally formed but were excavated into 1.6 kilometers of a cliff face in the Gobi desert. Due to the constraints of the landform as shown in Figure 9, the only place we can setup the data server is a few hundred meters away from the caves. At the time of deployment, there is no wired communication infrastructure available between the data server and the caves, and the distance between the data server and the caves is beyond sensor nodes' communication capability. Therefore, we employed a multi-tiered communication architecture to relay data from nodes located inside caves to the data server, as shown in Figure 10. In total, there are three tiers, sensor nodes, gateway nodes, and a sink node.

We deployed a group of sensor nodes in each cave. We found that the thick rock walls exhibit a high degree of radio attenuation and they can almost completely block radio propagation. Thus, we let nodes inside the same cave form a cluster. For each cluster, the sensor node located at the cave entrance is selected as the cluster header, since it is the only node that can reliably relay data to the upper tier, e.g. gateway nodes. All nodes in the same cluster will wake up and sleep synchronously, and they wake up once every minute to satisfy the monitoring requirement.

Outside the caves, we have installed several gateway nodes along the precipice to relay microclimate data between nearby cluster headers and the sink node. As shown in Figure 11, the gateway nodes are equipped with two wireless communication modules: One is a short-range communication module (CC2420 [TI 2013]), used to communicate with the WSN cluster headers; and the other is a long distance communication module, HAC-LM [HAC 2010] 433MHz, whose communication range can reach several kilometers in open space and is adequate to deliver data to the sink node. The sink node is also equipped with the long distance communication module and is connected to Internet via a wired connection.

5.2. Second Round Trial

In our second round trial, we deployed 241 sensor nodes in 57 caves. In each cave, around 3-7 sensor nodes were sufficient to form a reliable network that meets the monitoring requirement. The number of nodes deployed in the second round trial is six times of the one in our first round exploration. Fortunately, by employing the MC^2 framework, we managed to finish deploying the entire network using approximately the same amount of time. The detailed time spent in each step is listed in Table II.

5.2.1. Hardware Testing. Hardware testing involves verifying node hardware and uploading customized programs for application. In total, 300 nodes were prepared prior to deployment. The hardware verification job was done by the QC personnel of our hardware OEM factory. We provided them a simplified MC^2 software with only the Local Access, Code Update, Factory Test, and Power-off modes. Our hardware OEM factory will program the MCUs during manufacturing. They used the Factory Test mode to test hardware, and spent at most 1 minute for verifying one node. Thus, only about 5 hours were spent for verifying all 300 sensor nodes, which is a great improvement compared with spending 6 hours in testing 40 nodes in our first round exploration. The biggest saving of time was due to improved code uploading. When uploading customized programs, we spent 3 hours to program nodes one by one in our first round exploration. In our second round trial, nodes can be programmed wirelessly in a batch. Given that about one minute was needed to broadcast the code and the success rate was more than 90%, we completed software programming for the 300 nodes within 10 minutes. We note that we still needed to switch on the nodes one by one before wireless programming, which took us about 50 minutes.

5.2.2. Deployment. The online calibration feature enables us to calibrate a CO_2 sensor without taking it off the node, which was a mandatory hassle in the first round exploration. To calibrate a CO_2 sensor, we first switch the CO_2 sensor node to the Local Access mode, and peer SensorMate with the node. Then we start an elevation calibration and a two-point calibration. To perform the elevation calibration, SensorMate notifies the node the current elevation over the air. To conduct the two-point calibration, we connect the sensor to the 0 PPM standard CO_2 gas and run “zero calibration”, and then connect it to the 5K PPM standard CO_2 gas and run “span calibration”. At the same time, the `Calibration` module notifies the CO_2 sensor that the current measurements map to 0 PPM and 5K PPM respectively. The final step of calibration is to confirm that the reading of calibrated sensor is accurate. Following those steps, we only spent about 3 hours in calibrating 31 CO_2 sensor nodes[1], a big improvement from 4 sensors per hour in our first round trial.

[1]Out of 241 nodes, 31 of them contain CO_2 sensors.

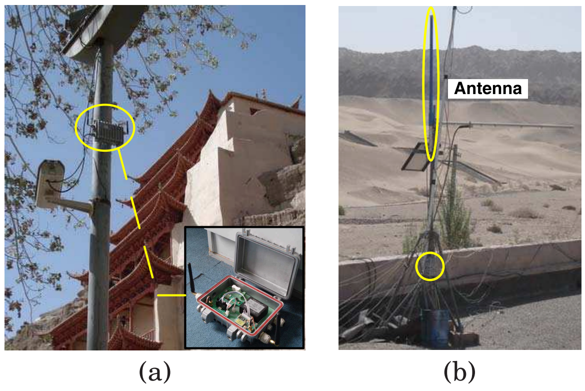


Fig. 11. (a) A deployed gateway node, and (b) the sink node.

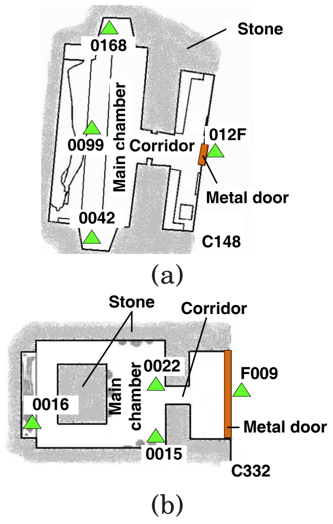


Fig. 12. The layout of nodes deployed in caves: (a) C148, and (b) C332.

The fast deployment validation and wireless configuration have shown to greatly speed up the sensor node placement process and the burn-in process. It took us 5 days in total to deploy 40 nodes in our first trial, but took about the same amount of time to deploy 241 nodes in our second trial. Similar to the first round exploration, in the second trial, we started with a few nodes and inserted new ones if the connectivity was tested as unstable. We learned the test results by visually observing the blinking LEDs, which indicate whether data transmissions are successful. Because of the `Fast Running` module, we can test a larger number of scenarios in a shorter period of time. Figure 12 shows two examples of the WSN placement in two caves: cave C148 and cave C332, whereby node 0099 and node 0022 were added to the initial deployment so that the network connectivity became stable.

In cave C332, initially we placed node 0015 and node 0016, but they were insufficient to form stable connections with the cluster header F009. Because the thick stone walls blocked the direct communication between node 0015 and the cluster header F009, their link was unstable: they failed to communicate in scenarios when several people

Table II. The time usage in each task.

Stage	**Job**	**First round (10 caves, 40 nodes)**		**Second round (57 caves, 241 nodes)**	
		Time spent	**note**	**Time spent**	**note**
Hardware testing	Hardware verification	6 hours	9 minutes per node	5 hours	1 minute per node
	Software programming	3 hours	Programming in sequence	1 hour	50 minutes for mode switching; 10 minutes for code broadcasting
Deployment	Sensor calibration	1 hour (4 nodes)	15 minutes per node	3 hours (31 nodes)	6 minutes per node
	Placement	2 days	40 minutes per cave	3 days	10 minutes per cave
	Burn-in	3 days	Redeployment in 80% caves	3 days	Redeployment in 10% caves

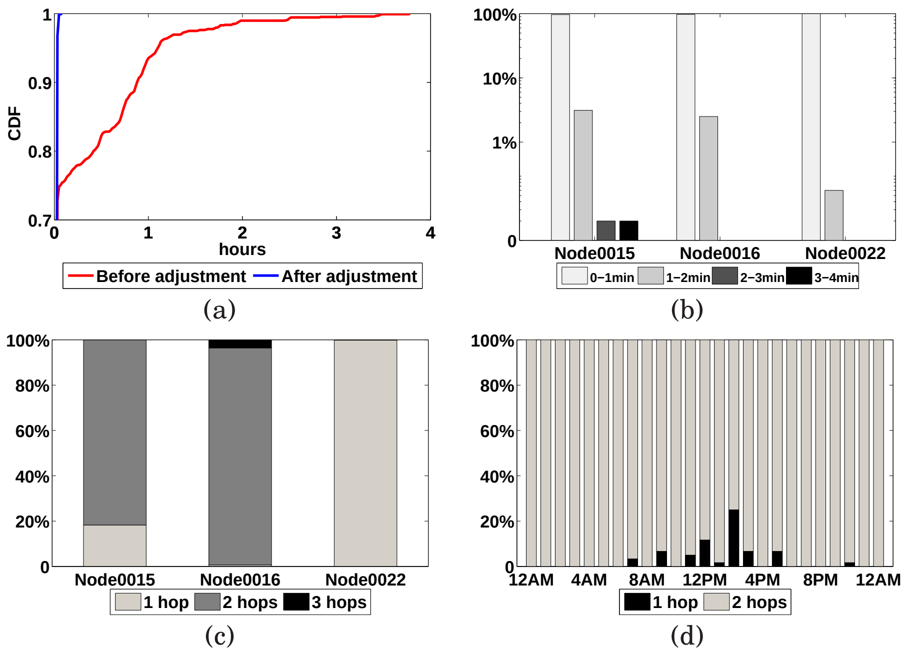


Fig. 13. Network topology plots of nodes in Cave C332: (a) the cumulative distribution of the data delivery delay between node 0015 and the cluster header F009, (b) data delay distribution, (c) hop count distribution, and (d) hop count distribution of node 0015.

stood on the signal propagation path or when the metal door was closed. The `Fast Running` module reported that the average packet delivery ratio between node 0015 and the cluster header was only about 70% on average, indicating that an additional node should be added. To improve the placement, we added node 0022 at the corridor end, as shown in Figure 12. Figure 13(a) depicts the cumulative distribution of the data delivery delay from node 0015 to the cluster header F009, i.e., the time from when the data were sampled to when the data were delivered to the cluster header. Prior to adding node 0022, 27% of the data delivered by node 0015 had a delay greater than 1 minute, which suggests that at least the first attempt to transmit data failed, and 10% of data took more than 1 hour to deliver. After adding a new node (0022), we improved the delay distribution of node 0015: most of the data from node 0015 were delivered within several minutes. A detailed analysis of data delivery delay of all nodes in C332 after adding node 0022 is given in Figure 13(b), which shows that all nodes had data delay less than several minutes. Another important metric to evaluate the stability of a link is the hop count towards the cluster header. The hop count of a stable link usually remains unchanged. Figure 13(c) shows that node 0016 and node 0022 had stable link quality as their hop counts remained the same most of the time. However, node 0015 was 1 hop 20% of the time and 2 hops 80% of the time, indicating slightly unstable network connection. A further study of the hop count distribution over a 24 hour period reveals the relationship between the position of the metal door and the network topology. As shown in Figure 13(d), when the door was closed, node 0015 could not communicate with the cluster header F009 directly, and the hop count of node 0015 became two. When the door was open in the daytime for tourists, the hop count of node 0015 dropped to 1 and they can communicate directly.

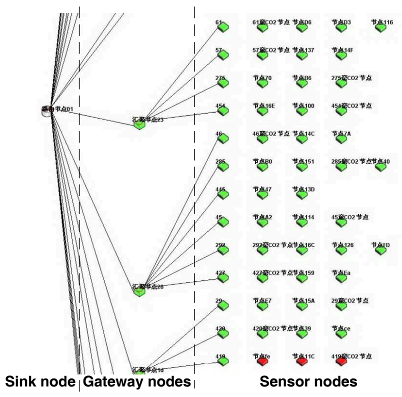


Fig. 14. Graphical system status summary (partial). Green nodes are healthy, while red nodes are faulty.

The `Fast Running` module has made the deployment process 4 times faster. Although we have spent less amount of time in placing sensor nodes, the quality of placement has been improved, because we were able to emulate a larger number of scenarios for verifying the network connectivity, and the time needed to spend on system burn-in has been reduced. Such improvement in placement is the direct payoff of the Pre-Running mode, since we are able to sample larger number of packets in a shorter period of time. The network connectivity between nodes was stable after deployment, and on average 95% microclimate data could be delivered within 1 minute.

5.2.3. Maintenance. The design goal of the MC^2 framework is to let users (historians in our case) be able to maintain the microclimate monitoring system on their own. Historians are not expected to become an expert in networking but should be able to identify the correct types of faults and repair them accordingly. Towards this goal, we have designed two utilities to help historians to make the right decision: the remote fault diagnosis utility and the on-site fault diagnosis utility.

The remote fault diagnosis utility will analyze the node status collected via the multi-hop networks and provide users fault symptom analyzing results. In particular, we extracted four fault symptoms that were encountered most often in our first deployment trial: (1) *incorrect data*, i.e., the sensed data are out of the normal range (e.g., -30°C to 70°C for temperature sensor), (2) a *high data delay* (e.g., 3 minutes on average) between a node and its cluster header, (3) a *high battery drain rate*, i.e., the drain speed of a node's battery is much faster than the empirical normal rate (e.g., 0.06V/week) , and (4) a *low battery voltage*, e.g., lower than 2.7V. The remote fault diagnosis utility consists of two parts: the `Status Reporting` module that reports node status periodically, and the symptom analysis program that runs at the data server. This program continuously analyzes the node status information and displays analyzed results graphically, as shown in Figure 14.

Different fault symptoms may be caused by different types of faults and require different ways of repairing. Table III summarizes the relationship between symptoms and fault types. For instance, a poor link quality is typically manifested by a high data delay. A high battery drain rate can be caused by several reasons: poor link quality (whereby nodes have to spend a large amount of energy on retransmission), software errors (e.g., mis-configured with a higher duty cycle than desired), or hardware errors. To repair the system, a historian has to visit the field for node examination and repairs. Based on the relationship summarized in Table III, we have designed a step-by-step maintenance manual for historians, as described below and depicted in Figure15.

Table III. Remote diagnosing.

Symptoms	Fault types	
	Poor link quality	Hardware or software error
Incorrect data		●
High data delay	●	
High battery drain rate & high data delay	●	
High battery drain rate & normal data delay		●

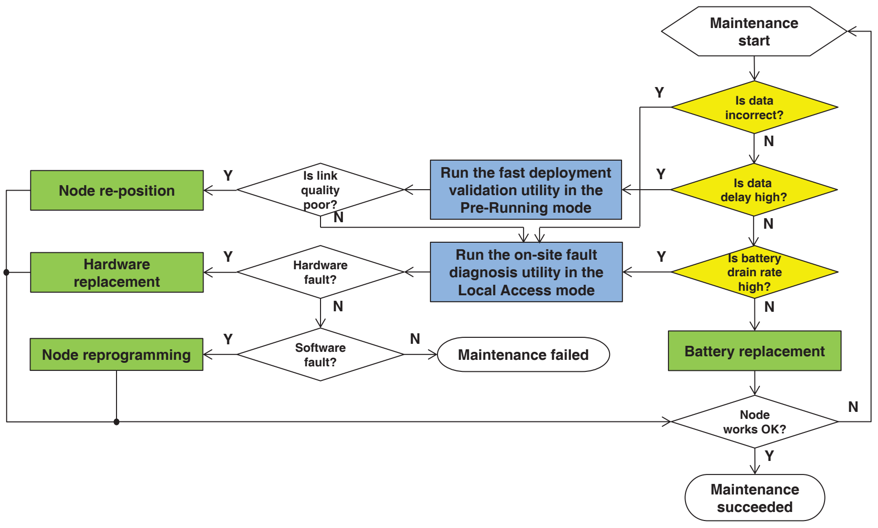


Fig. 15. The flow chart of the step-by-step maintenance manual.

(1) **Incorrect data.** The incorrect data symptom is typically caused by hardware faults (e.g., a malfunctioning sensor or flash memory). As a result, a historian will visit the field and run the `Node Inspection` module in the Local Access mode. If the inspection report shows a hardware problem of sensors, the sensor will be replaced. Otherwise the entire sensor node will be replaced.

(2) **High data delay.** A high data delay indicates that the link quality is not satisfying, and thus a historian will visit the field to adjust the node position. To improve the link quality in the field, a historian lets the faulty nodes switch to the Pre-Running mode and observe the LEDs of the nodes: a blinking *red* LED means the data transmission fails constantly and a blinking *green* LED means the data transmission is successful. The historian continues to adjust the locations of the nodes until observing a blinking green LED, just like how we deploy the network.

(3) **High battery drain rate.** If the battery voltage drops fast and the data delay is normal, there might be a hardware or software fault, and a historian will obtain local status reports when nodes are in the Local Access mode. If the report shows a hardware error, the node will be replaced. If a software error is discovered (e.g., the duty cycle is higher than required), then the node will be reprogrammed using SensorMate.

Although a historian may not be able to identify the exact root causes of faults and solve the problem permanently, the symptom reports have made it possible for a historian to keep the network operation normal. For example, the historian has observed that a few sensor nodes had high battery drain rates without the symptom of the high

Table IV. Typical time for maintenance tasks.

Task	**Average time spent per node**	
	First round (done by developers)	**Second round** (done by historians)
Node re-position	20 minutes	3 minutes
Hardware replacement	25 minutes	2 minutes
Node reprogramming	10 minutes	1 minute

data delay. Through on-site diagnosis using SensorMate, they learned that the status of flash memory on all faulty nodes was abnormal. Thus, the historian replaced those nodes and solved the problem temporarily. Later, we acquired those sensor nodes by mails. A detailed diagnosis in our lab revealed that a small set of the flash memory (AT45DB161D) chips in the system tended to run into a faulty state after repeated wakeup/sleep operations. In this faulty state, the memory chip will not respond to any READ/WRITE or even software RESET commands, but it will continuously consume high energy with a 3-4mA current draw. The only way to exit this abnormal state is to pull the hardware RESET pin of the memory chip. Because of the rare occurrence, this problem was not discovered during system deployment. Nevertheless, the historian was able to repair the network with the help of the MC^2 framework. After identifying the problem, we updated the node firmware by adding a routine for resetting the flash chip once it runs into the faulty state. The updated firmware was emailed to the historian and was distributed to nodes using SensorMate; hence, they helped to solve this type of fault permanently.

Currently, the daily system maintenance and the system repairing operations (e.g., node replacement and node placement adjustment) are conducted by the historian, which frees us from traveling to Dunhuang every time a system fault is detected. In the first round, we traveled to Mogao Grottoes three times with each trip lasted for up to 10 days. In the second round, we didn't need to visit the site, but sent new nodes and sensor parts by mails, because the historian managed to maintain the network with the help of the MC^2 framework.

Depending on the symptoms, the historian will perform one of the three on-site maintenance tasks, and Table IV summarizes the time spent for each task. For comparison, we listed the time spent by us in the first round exploration.

— **Node re-position.** The Pre-Running mode is used to examine the link stability. In our systems, 3 minutes were sufficient for the historian to search for a new location that creates stable links. In comparison, in the first round exploration, we spent 20 minutes on average to re-position one node, because the low duty cycle made it time consuming to confirm the link stability.

— **Hardware replacement.** The `Node Inspection` module in the Local Access mode checks hardware and informs the historian if there is a hardware fault. Then the historian will replace the hardware and check the network status using the Pre-Running mode. In total, 2 minutes were sufficient for the entire operation, much shorter than what was needed in the first round (e.g. 25 minutes).

— **Node reprogramming.** The `Configuration`, `Code Reception` and `Reprogramming` modules make software related repairing efficient: 1 minute on average sufficed to identify the problem and to update the nodes with a correct program, much shorter than 10 minutes where most time was spent on connecting JTAG to the nodes.

Because our MC^2 framework greatly reduces the maintenance workload, currently only one historian is in charge of maintenance. Based on his feedback, he is satisfied with the MC^2 framework and considers the maintenance manageable. A formal usability study involving a larger number of users will be our future work.

5.3. MC^2 Framework Overhead and Performance

In this subsection, we examine the overhead and performance of adopting a MC^2 framework in two aspects: the memory usage, and the loading time of the MMC and modes.

5.3.1. Memory Usage. The size of the MC^2 software is on the order of a few kilo bytes (KB). Table V summarizes the itemized binary-code sizes. In general, the MC^2 software contains three categories of source codes: modes and modules (including the MMC), the network protocol stack, and hardware drivers. The MMC contains the source code for managing modes, and its code size is about 17.91KB. The Data Collection mode and the Pre-Running mode share the same source code but have different configuration files. Since each mode will share a portion of source codes (e.g., hardware drivers), the total size of the entire framework in the application flash section (57.66KB) is smaller than the sum of the code for each mode. The other two modes resided in the bootloader flash section are smaller: the Factory Test mode and the Code Update mode only occupy 7.05KB and 4.71KB.

5.3.2. Loading time. The MMC switches between modes according to a user's input. Table V shows the loading time of the MMC and modes. When a node boots up, it will first spend about 8.4 ms to load the MMC with most time spent in hardware initialization. The time for loading the Data Collection mode is about 1.75 seconds with almost all the time spent on checking external memories, which are used to buffer data. The time for loading the Pre-Running mode and other modes are only on the order of a few milliseconds, because they do not need to check external memories. Among them, the time for the loading the Factory Test mode is the longest (e.g., 5.6 ms) because it requires initializing serial ports, which are used for reporting testing results to terminals.

In summary, the MC^2 framework incurs modest overhead and loading delay, and is applicable to sensor platforms.

6. RELATED WORK

Several sensor network systems have been deployed for monitoring purposes. Early example monitoring systems include the habitat monitoring system on Great Duck Island (GDI) [Mainwaring et al. 2002], the zebra tracking system [Juang et al. 2002], the volcano monitoring system [Werner-Allen et al. 2006], and the heritage monitoring system deployed at Torre Aquila [Ceriotti et al. 2009]. Those deployments provide precious first-hand experience towards future successful deployments. However, those systems essentially follow a network-centric design. They are primarily interested in network issues that arise in the network operation stage instead of reducing the users' burden. We believe that such a focus is because many prior work of WSNs involved deploying a few nodes and were studied intensively for a relatively short period of time, e.g., the 44-day test running on a redwood tree consisting of 33 sensor nodes [Tolle et al. 2005] and a two-month testing deployment of SensorScope on a rock glacier using 16

Table V. The MC^2 framework overhead and performance.

Name	Memory usage	Loading time
MMC	17.91KB	8.4 ms
Data Collection mode	27.94KB	1.75 s
Pre-Running mode		2.8 ms
Local Access mode	17.19KB	2.8 ms
Factory Test mode	7.05KB	5.6 ms
Code Update mode	4.71KB	0.03 ms
Power-off mode	Uncountable	0.16 ms

sensor stations [Barrenetxea et al. 2008]. In comparison, many real-world sensing applications are expected to run for a long time in various environments to provide a stream of real time data for scientific studies, e.g., an automated wildlife monitoring system lasting for one year [Dyo et al. 2010] and our deployment at a heritage site for more than five years [Xia et al. 2012]. Our work addresses the challenges caused by the growing need of user-intervention for long-term monitoring WSNs.

Much prior work does propose 'tools' to assist system deployment and maintenance. The designers of the Great Duck Island monitoring system have identified two external tools to assist deployment and maintenance: field tools running on small PDA-class devices and client tools running on PC-class devices [Mainwaring et al. 2002]. However, these tools were not their focus. Thus, they did not present the design details. The researchers from Berkeley adopted TASK (Tiny Application Sensor Kit) [Buonadonna et al. 2005] in building their microclimate monitoring system on redwoods [Tolle et al. 2005]. To enable non-network specialists to deploy and manage WSNs, TASK includes a set of tools to assist software installation, deployment, reconfigurability, and sensor nodes health monitoring, etc. Our MC^2 framework incorporates all these tools and has included additional utilities (e.g., the `Self Checking` module, the `Node Inspection` module, and the `Calibration` module). In addition, our framework coordinates the execution of these tools and enables users to quickly change nodes' behaviors according to user requirements(e.g., inspecting node status or collecting data).

Deployment assessment has been addressed in FireWxNet [Hartung et al. 2006], LUSTER [Selavo et al. 2007], and SensorTune [Costanza et al. 2010]. FireWxNet is a weather monitoring system deployed in wildland fire environments. During the deployment phase, the sensor nodes in FireWxNet exchange packets at a higher rate for verifying the communication quality than the regular runtime rate. LUSTER designed a deployment-time tool, SeeMote, which can gather a rich set of network measurement (e.g. RSSI or residual battery power) to access the health of the network. SensorTune is a mobile interface for assisting WSN deployment, and it provides audio feedback to indicate the connectivity of the network. The Pre-Running mode in our MC^2 framework is also designed to expedite deployment. The differences are that our MC^2 framework integrates modes not only for deployment but also for hardware testing, maintenance, etc., and it allows a user to easily switch between modes.

Calibration is one of the most challenging issues in real-world deployments, especially in a large scale WSN. Ramanathan *et al.* presented Suelo, which actively requests the help of a human when necessary to validate, calibrate, repair, or replace sensors [Ramanathan et al. 2009]. To alleviate the large workload in calibrating sensors, researchers have proposed pairwise calibrating or iterative calibrating [Akcan 2013], in which the calibrated sensors can be used to calibrate uncalibrated sensors as long as they can observe the same event. Similar idea was also adopted by Xiang *et al.* to improve accuracy of mobile sensing systems [Xiang et al. 2012]. Our MC^2 framework is complementary to the work because prior work requires calibrating at least a subset of sensors manually and the MC^2 framework reduces the workload of manual calibration.

Remote debugging systems, such as NodeMD [Krunic et al. 2007] and Clairvoyant [Yang et al. 2007], enable a user to remotely debug a program running on sensor nodes. Identifying sensor failures is one of the key tasks for maintenance. Sharma *et al.* focused on three types of sensor faults, and analyzed methods to detect them [Sharma et al. 2007]. Ni *et al.* employed a hierarchical Bayesian spatio-temporal (HBST) modeling approach to detect faulty data [Ni and Pottie 2012]. To cope with a broader range of system failures, Sympathy collects run-time status information of sensor nodes to detect possible system faults [Ramanathan et al. 2005]. To reduce additional traffic overhead caused by information collection, PAD employs a packet marking algorithm for detecting and diagnosing faults [Liu et al. 2008]. Similarly, DSD uses data traces to analyze the type of failure [Nie et al. 2012]. Hnat *et al.* proposed to use data losses to identify failures in their residential sensing deployments [Hnat et al. 2011]. Wu *et al.* proposed a methodology for run-time assurance (RTA), which validates whether the system can meet application requirements [Wu et al. 2010]. Khan *et al.* presented tele-diagnostic powertracer [Khan et al. 2010], which uses device-wise power measurements to determine the healthiness of an unresponsive node and deduce the most likely failure cause. Alternatively, field diagnosis tools, such as Sensible Doctor [Cha et al. 2008], enable users to monitor a subset of the deployed sensor networks in the field. Those systems are useful for system maintenance and can be integrated into our MC^2 framework.

In the area of hardware design, a group at Berkeley has proposed a building block approach to support the three phases in sensornet hardware platform development: prototype, pilot and production [Dutta et al. 2008]. Our work complements theirs, as we focus on the stages after hardware development.

The aforementioned systems cover issues associated with some stages of the WSN life cycle. In contrast, we have identified a set of utilities to address user-related issues throughout the entire WSN life cycle, and we have proposed a MC^2 framework that organizes those utilities in a way that improves the runtime efficiency of the individual utility without compromising the performance of others. We note that our framework is essentially a container, and it can be extended to incorporate many existing and future tools, such as remote fault diagnosis utilities.

The proposed MC^2 framework is similar to the concept of middleware. Middleware provides an abstract layer between the application and operating system to improve the extensibility and portability of application software. Most of WSN middlewares focus on data management (e.g., COUGAR [Bonnet et al. 2001], TINYDB [Madden et al. 2005], Mires [Souto et al. 2006], and PerLa [Schreiber et al. 2012]) and runtime node programming or configuration (e.g., Impala [Liu and Martonosi 2003], Maté [Levis and Culler 2002], Agilla [Fok et al. 2005], RemoWare [Taherkordi et al. 2013], and REED [Fei and Magill 2012]). There are also other middlewares trying to improve the simultaneousity of application programs [Horré et al. 2008] and support complex in-network processing [Lachenmann et al. 2010]. We are not aware of any middleware design that targets at reducing the levels of user intervention throughout the lifetime of WSNs.

7. CONCLUDING REMARKS

A user-centric design is critical to reduce the amount of time for deploying and maintaining large-scale long-running WSN applications. As such, we have proposed in this paper a Multi-mode user-CentriC (MC^2) framework. Based on our experiences gained in our early explorative deployment, we have identified a set of utility programs that can alleviate the burden imposed to users because of the poor user-node interface. As those utility programs have different requirements on bootstrapping and duty cycles, it is inefficient, if possible, to have them running simultaneously. Thus, we carefully examined the utilities' behaviors and their applicable scenarios, and designed the MC^2 framework that can efficiently integrate them by clustering utility programs with similar requirements into a *mode*. Our current system supports six modes, and they can be switched easily by the Mode Management Component (MMC) in the MC^2 framework. Further, we have presented our design of a matching sensor node platform which contains a user-friendly control panel to assist mode switching. We have implemented and evaluated our MC^2 framework in a long-term microclimate monitoring system deployed at Mogao Grottoes. By using the MC^2 framework, we were able to deploy 241 sensor nodes in 57 caves within one week, which is a significant improvement compared with our first round exploration where we spent the same amount of time only to deploy 40 nodes in 10 caves. Meanwhile, we showed that the MC^2 framework has greatly reduced the maintenance overhead, and enabled field experts (in our case historians) to maintain the network in the past five years.

REFERENCES

Adesto. 2013. AT45DB161D. (2013). http://www.adestotech.com/sites/default/files/datasheets/doc3500.pdf

Hüseyin Akcan. 2013. On the complexity of energy efficient pairwise calibration in embedded sensors. *Applied Soft Computing* 13, 4 (2013), 1766–1773.

Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. 2002. Wireless sensor networks: A survey. *Computer Networks* 38, 4 (2002), 393–422.

Atmel. 2013. ATmega128. (2013). http://www.atmel.com/devices/atmega128.aspx

Lan S Bai, Robert P Dick, and Peter A Dinda. 2009. Archetype-based design: Sensor network programming for application experts, not just programming experts. In *Proceedings of the 8th International Conference on Information Processing in Sensor Networks (IPSN '09)*. 85–96.

Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, Martin Vetterli, Olivier Couach, and Marc Parlange. 2008. Sensorscope: Out-of-the-box environmental monitoring. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN '08)*. 332–343.

Elizabeth A Basha, Sai Ravela, and Daniela Rus. 2008. Model-based monitoring for early warning flood detection. In *Proceedings of the 6th International Conference on Embedded Networked Sensor Systems (SenSys '08)*. 295–308.

Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. 2001. Towards sensor database systems. 1987 (2001), 3–14.

Philip Buonadonna, David Gay, Joseph M Hellerstein, Wei Hong, and Samuel Madden. 2005. TASK: Sensor network in a box. In *Proceeedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN '05)*. 133–144.

Matteo Ceriotti, Luca Mottola, Gian Pietro Picco, Amy L Murphy, Stefan Guna, Michele Corra, Matteo Pozzi, Daniele Zonta, and Paolo Zanon. 2009. Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment. In *Proceedings of the 8th International Conference on Information Processing in Sensor Networks (IPSN '09)*. 277–288.

Seunghun Cha, Hyojeong Shin, and Hojung Cha. 2008. Sensible doctor-A mobile diagnosis tool for wireless sensor networks. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN '08)*. 565–566.

Enrico Costanza, Jacques Panchard, Guillaume Zufferey, Julien Nembrini, Julien Freudiger, Jeffrey Huang, and Jean-Pierre Hubaux. 2010. SensorTune: A mobile auditory interface for DIY wireless sensor networks. In *Proceedings of the 28th ACM Conference on Human Factors in Computing Systems (CHI '10)*. 2317–2326.

Crossbow. 2007. MICAz 2.4GHz. (2007). http://bullseye.xbow.com:81/Products/productdetails.aspx?sid=164

Budhaditya Deb, Sudeept Bhatnagar, and Badri Nath. 2003. ReInForM: Reliable information forwarding using multiple paths in sensor networks. In *Proceedings of the 28th Annual IEEE International Conference on Local Computer Networks (LCN '03)*. 406–415.

Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. 2006. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*. 29–42.

Prabal Dutta, Jay Taneja, Jaein Jeong, Xiaofan Jiang, and David Culler. 2008. A building block approach to sensornet systems. In *Proceedings of the 6th International Conference on Embedded Networked Sensor Systems (SenSys '08)*. 267–280.

Vladimir Dyo, Stephen A Ellwood, David W Macdonald, Andrew Markham, Cecilia Mascolo, Bence Pásztor, Salvatore Scellato, Niki Trigoni, Ricklef Wohlers, and Kharsim Yousef. 2010. Evolution and sustainability of a wildlife monitoring sensor network. In *Proceedings of the 8th International Conference on Embedded Networked Sensor Systems (SenSys '10)*. 127–140.

Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. 1999. Next century challenges: Scalable coordination in sensor networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '99)*. 263–270.

Xiang Fei and Evan Magill. 2012. REED: Flexible rule based programming of wireless sensor networks at runtime. *Computer Networks* 56, 14 (2012), 3287–3299.

Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. 2005. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS '05)*. 653–662.

Getty. 2010. Wall paintings at Mogao Grottoes. (2010). http://www.getty.edu/conservation/our_projects/field_projects/mogao/index.html

Google. 2013. Google maps. (2013). https://maps.google.com/

Lin Gu and John A Stankovic. 2006. t-kernel: Providing reliable OS support to wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*. 1–14.

HAC. 2010. HAC-LM data RF module. (2010). http://www.rf-module-china.com/products/RF_Modules/20.html

Carl Hartung, Richard Han, Carl Seielstad, and Saxon Holbrook. 2006. FireWxNet: A multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In *Proceedings of the 4th International Conference on Mobile Systems, Applications and Services (MobiSys '06)*. 28–41.

Daojing He, Chun Chen, Sammy Chan, and Jiajun Bu. 2012. SDRP: A secure and distributed reprogramming protocol for wireless sensor networks. *IEEE Transactions on Industrial Electronics* 59, 11 (2012), 4155–4163.

Timothy W Hnat, Vijay Srinivasan, Jiakang Lu, Tamim I Sookoor, Raymond Dawson, John Stankovic, and Kamin Whitehouse. 2011. The hitchhiker's guide to successful residential sensing deployments. In *Proceedings of the 9th International Conference on Embedded Networked Sensor Systems (SenSys '11)*. 232–245.

Wouter Horré, Sam Michiels, Wouter Joosen, and Pierre Verbaeten. 2008. Davim: Adaptable middleware for sensor networks. *IEEE Distributed Systems Online* 9, 1 (2008), 1.

HP. 2013. iPAQ hx2490c pocket PC. (2013). http://h20000.www2.hp.com/bizsupport/TechSupport/Document.jsp?prodSeriesId=421399&objectID=c01330819

Jonathan W Hui and David Culler. 2004. 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 (SenSys '04)*. 81–94.

Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. 2002. 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 (ASPLOS X)*. 96–107.

Mohammad Maifi Hasan Khan, Hieu K Le, Michael LeMay, Parya Moinzadeh, Lili Wang, Yong Yang, Dong K Noh, Tarek Abdelzaher, Carl A Gunter, Jiawei Han, and others. 2010. Diagnostic powertracing for sensor node failure analysis. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN '10)*. 117–128.

Veljko Krunic, Eric Trumpler, and Richard Han. 2007. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services (MobiSys '07)*. 43–56.

Andreas Lachenmann, Ulrich Müller, Robert Sugar, Louis Latour, Matthias Neugebauer, and Alain Gefflaut. 2010. Programming sensor networks with state-centric services. In *Proceedings of the 6th International Conference on Distributed Computing in Sensor Systems (DCOSS '10)*. 306–319.

Philip Levis and David Culler. 2002. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. 85–95.

Kebin Liu, Mo Li, Yunhao Liu, Minglu Li, Zhongwen Guo, and Feng Hong. 2008. Passive diagnosis for wireless sensor networks. In *Proceedings of the 6th International Conference on Embedded Networked Sensor Systems (SenSys '08)*. 113–126.

Ting Liu and Margaret Martonosi. 2003. Impala: A middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*. 107–118.

Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2005. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems* 30, 1 (2005), 122–173.

Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. 2002. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '02)*. 88–97.

Kevin Ni and Greg Pottie. 2012. Sensor network data fault detection with maximum a posteriori selection and bayesian modeling. *ACM Transactions on Sensor Networks* 8, 3 (2012), 23.

Jiangwu Nie, Huadong Ma, and Lufeng Mo. 2012. Passive diagnosis for WSNs using data traces. In *Proceedings of the 8th International Conference on Distributed Computing in Sensor Systems (DCOSS '12)*. 273–280.

Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. 2005. Sympathy for the sensor network debugger. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys '05)*. 255–267.

Nithya Ramanathan, Thomas Schoellhammer, Eddie Kohler, Kamin Whitehouse, Thomas Harmon, and Deborah Estrin. 2009. Suelo: Human-assisted sensing for exploratory soil monitoring studies. In *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems (SenSys '09)*. 197–210.

Ramtron. 2011. FM24CL64. (2011). http://www.ramtron.com/files/datasheets/FM24CL64_ds.pdf

Fabio A Schreiber, Romolo Camplani, Marco Fortunato, Marco Marelli, and Guido Rota. 2012. Perla: A language and middleware architecture for data management and integration in pervasive information systems. *IEEE Transactions on Software Engineering* 38, 2 (2012), 478–496.

Leo Selavo, Anthony Wood, Qing Cao, Tamim Sookoor, Hengchang Liu, Aravind Srinivasan, Yafeng Wu, Woochul Kang, John Stankovic, Don Young, and others. 2007. Luster: Wireless sensor network for environmental research. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys '07)*. 103–116.

Sensirion. 2011a. SHT11 - digital humidity sensor (RH&T). (2011). http://www.sensirion.com/en/products/humidity-temperature/humidity-sensor-sht11/

Sensirion. 2011b. SHT75 - digital humidity sensor (RH&T). (2011). http://www.sensirion.com/en/products/humidity-temperature/humidity-sensor-sht75/

Abhishek Sharma, Leana Golubchik, and Ramesh Govindan. 2007. On the prevalence of sensor faults in real-world deployments. In *Proceedings of the 4th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON '07)*. 213–222.

Yucheng Shi and Jie Zhang. 1997. The Dunhuang Caves main diseases and precautions against them. *Northwestern Seismological Journal* 19, 2 (1997), 81–87.

Eduardo Souto, Germano Guimarães, Glauco Vasconcelos, Mardoqueu Vieira, Nelson Rosa, Carlos Ferraz, and Judith Kelner. 2006. Mires: A publish/subscribe middleware for sensor networks. *Personal and Ubiquitous Computing* 10, 1 (2006), 37–44.

Amir Taherkordi, Frederic Loiret, Romain Rouvoy, and Frank Eliassen. 2013. Optimizing sensor network reprogramming via in situ reconfigurable components. *ACM Transactions on Sensor Networks* 9, 2 (2013), 14.

Telaire. 2004. Telaire 6004 CO_2 module. (2004). http://www.telaire.com/oem/oem_stand.htm

Telegesis. 2013. ETRX2 'CF' compact flash card. (2013). http://www.telegesis.com/products/test_page_3.htm

TI. 2013. CC2420. (2013). http://www.ti.com/product/cc2420

Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and others. 2005. A macroscope in the redwoods. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys '05)*. 51–63.

Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. 2006. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. 381–396.

Yafeng Wu, Krasimira Kapitanova, Jingyuan Li, John A Stankovic, Sang H Son, and Kamin Whitehouse. 2010. Run time assurance of application-level requirements in wireless sensor networks. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN '10)*. 197–208.

Ming Xia, Yabo Dong, Wenyuan Xu, Dongming Lu, and Xiangyang Li. 2010. Multi-mode user-centric design of wireless sensor networks for long-term monitoring. *ACM SIGMOBILE Mobile Computing and Communications Review* 14, 1 (2010), 25–27.

Ming Xia, Yabo Dong, Wenyuan Xu, Dongming Lu, Ping Xue, and Gang Liu. 2012. Long-term microclimate monitoring in wildland cultural heritage sites with wireless sensor networks. *International Journal of High Performance Computing and Networking* 7, 2 (2012), 111–122.

Yun Xiang, Lan Bai, Ricardo Piedrahita, Robert P Dick, Qin Lv, Michael Hannigan, and Li Shang. 2012. Collaborative calibration and sensor placement for mobile sensor networks. In *Proceedings of the 11th International Conference on Information Processing in Sensor Networks (IPSN '12)*. 73–84.

Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. 2007. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys '07)*. 189–203.