# Dynamic Reconfiguration of Wireless Sensor Networks to Support Heterogeneous Applications

Marcin Szczodrak
Columbia University
msz@cs.columbia.edu

Omprakash Gnawali
University of Houston
gnawali@cs.uh.edu

Luca P. Carloni
Columbia University
luca@cs.columbia.edu

***Abstract*—As larger numbers of Wireless Sensor Network (WSN) applications get deployed in our homes and offices, it is desirable to use the same network to run different applications. We present and analyze the problem of scheduling and supporting the execution of multiple heterogeneous applications on top of the same WSN. First, we establish that using the same MAC or network protocol is not sufficient to obtain acceptable performance across a set of applications that require different types of communication services from the protocol stack (e.g., low-rate reliable many-to-one collection vs point-to-point low-latency bulk-data streaming). Hence, we propose a framework to dynamically reconfigure the WSN and adapt its power consumption, transmission reliability, and data throughput to the different requirements of the applications. The framework makes it possible to specify, at design time, distinct network, MAC and radio protocols for each application as well as the events and policies triggering the WSN reconfigurations. At run-time, the WSN automatically reconfigures itself in response to these events and according to these policies. Through experiments on a 119-node testbed, we show that the proposed approach can reconfigure the whole network in few hundreds of milliseconds while incurring little memory and control overhead.***

## I. Introduction

Indoor climate monitoring and control, intrusion detection, and energy-use monitoring are examples of Wireless Sensor Network (WSN) applications being deployed in large numbers. Often, each new application requires installation of a dedicated WSN. However, researchers have realized that it is infeasible to deploy a separate WSN for each application.

We propose a WSN framework that supports the execution of different applications at different times. We motivate and demonstrate our framework by presenting the combined deployment of two heterogeneous applications for indoor monitoring of a building environment on the same WSN. The deployment must satisfy the following requirements:

1) Minimize the number of WSN nodes deployed in the building.
2) During normal operation, the network must reliably collect climate data (e.g., temperature) to a single server while remaining energy efficient. We call this application *Collection*.
3) When an emergency event occurs in a particular zone of the building (e.g. a smoke sensor goes off), the network must rapidly transmit a sequence of images from this zone to the server. We call this application *Firecam*.

The first requirement (1), common to many other WSN applications, stems from physical, logistical, and cost considerations. While compressive-sensing and optimal sensor placement partly address this requirement, our approach shares nodes across applications to reduce the number of required nodes. The second (2) and third (3) requirements are specific to the *Collection* and *Firecam* applications.

In our desire to leverage prior work to meet the requirements of our target applications we focus on a few choices:

- Run different dataflow programs corresponding to *Collection* or *Firecam* at different times on top of the same network, link, and physical layer protocols, as in Tenet [8].
- Re-program the WSN using systems such as Deluge [13] when we switch from *Collection* to *Firecam*.
- Re-configure the MAC parameters with systems such as pTunes [26] to optimize performance as the traffic pattern changes between running *Collection* and *Firecam*.

Tenet and pTunes do not allow the two applications to run on their preferred protocol stack. Deluge, however, takes several minutes to reprogram the nodes, leading to unacceptable delays while transitioning from *Collection* to *Firecam*. We also find and show (later in the paper) that the selection of different protocols or tuning the parameters of a single layer (e.g., MAC) misses the opportunity to comprehensively optimize network performance at each operational phase.

To overcome these limitations we developed Fennec Fox, a framework to dynamically reconfigure a WSN to support different applications at different times. To perform optimally, these applications depend on different network and MAC protocols. By providing a way to dynamically select and configure each component of the protocol stack, Fennec Fox allows us to leverage these existing works and support execution of heterogeneous applications on a single WSN.

Our approach consists of two steps. At design time, for each application we can specify a distinct protocol stack (consisting of a network, a MAC, and a radio protocol) as well as the policies that govern the WSN reconfigurations and the events that trigger them. Then, at run-time, the WSN automatically reconfigures itself in response to these events and according to these policies. These two steps result in the dynamic scheduling of the execution of different applications, each supported by its own optimized network, MAC, and radio protocols.

Our main contributions include:

- Design of a language and tool-chain to configure a network protocol stack to support execution of an application and the conditions under which different applications with their corresponding protocols should execute.
- Implementation of Fennec Fox to demonstrate the feasibility of executing *Collection* and *Firecam* on top of their preferred protocol stack in a single WSN.
- Evaluation of Fennec Fox on a 119-node testbed, showing that dynamic reconfiguration is not only feasible, but also quick, efficient, and extendible to a large number of applications and various protocols.

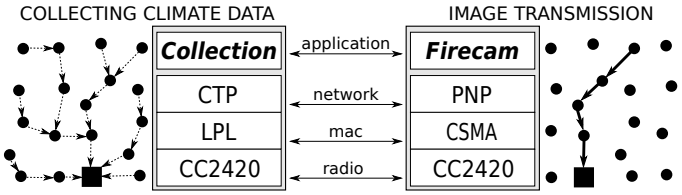



Fig. 1. Two different applications and their corresponding protocol stacks. In the sequel we will use the term *CTP stack* and *PNP stack* to denote the protocol stacks required by *Collection* and *Firecam*, respectively.

## II. One Network, Two Applications

We want to run two different sensor network applications in our office building.

*Collection* is the default WSN application, in the sense that is executed "continuously" during the normal network operations: it monitors the building's environment by collecting various kinds of information (e.g. temperature, humidity, light, and, possibly, also people occupancy through PIR and camera sensors) from the WSN nodes that are distributed almost uniformly across all the various zones in which is partitioned the building. The collected information can be used for various purposes, such as improving the operation of the HVAC system or saving the power consumed by the building. As part of this application, every WSN node periodically sends a message with the collected sensors' sample to a collective node hosted on a server (the *sink*.) These messages are small, just a few tens of bytes, but transmission reliability is important, i.e the sink is expected to receive them with a high delivery ratio[1]. The period of the transmission may vary depending on the size of the building and the required granularity of the measurements. Typical values of the period are between 1 and 3 minutes. When the nodes are not transmitting data, the WSN is duty-cycled to minimize power consumption.

The second application of interest, *Firecam*, is executed much more rarely as a consequence of an emergency event. Specifically, when a smoke detector or a security sensor goes off in a particular zone of the building, a node (or a very limited number of nodes) in that zone takes a series of pictures and sends this stream of pictures to a sink node, which may or may not coincide with the same sink node of the other application. Thus, differently from the previous application, in the case of *Firecam* we are interested in the transmission of large amounts of data (the size of a single picture is about 76k bytes) on a point-to-point connection between two nodes that are typically located in two distant zones of the building. One of the purposes of *Firecam* is to assist emergency/security personnel to immediately assess the gravity of the potential problem[2].

Fig. 1 illustrates the main characteristics of the two applications in terms of communication patterns across the WSN. It also shows the choice of the protocols that are optimized to execute each of them. In particular, the state-of-the-art data-collection protocol CTP [9] is best suited to support the *Collection* application because it achieves a delivery ratio close to 100% while being also very power efficient. It does so by establishing a network-tree topology and routing

[1]The delivery ratio is defined as the ratio of total number of received messages over total number of sent messages.

[2]This is indeed a practical issue since many emergency alarms are often the results of false positive sensor readings. It is thus helpful to have a mechanism that can quickly confirm the occurrence of real problems through the real-time delivery of pictures of the particular zones.

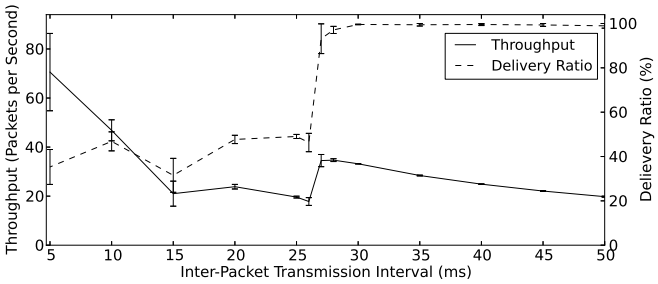



Fig. 2. Throughput and Delivery Ratio during operation of the PNP stack.

the packets with the applications' small messages from the various nodes toward the sink. CTP is a multi-hop network protocol that relies on the services provided by the MAC and radio protocols, which focus on a single transmission between two nodes. CTP was designed to run on top of a CSMA MAC protocol, which attempts to avoid transmission collisions by sensing presence of other radio communications and introducing random transmission delays. Also, the CSMA MAC protocol is typically augmented with a Low Power Listening (LPL) mechanism to duty-cycle the WSN. The quality of a single-hop transmission is further supported by radio services that provide clear channel assessment (CCA), auto acknowledgement, and automatic CRC error-detection.

In the case of *Firecam*, instead, the efficient transmission of a stream of pictures from one particular node to the sink requires to quickly establish a multi-hop path between them. Assuming that a picture size is $240 \times 320$ pixels and that each pixel is encoded as a single byte, the transmission of a single uncompressed picture requires the transfer of 76800 bytes. We can partition such picture in 768 packets each storing 100 bytes of picture data and a 4-byte sequence number that is necessary to allow the picture reconstruction at the sink. The *Parasite Network Protocol* (PNP) is a network protocol that can efficiently support the *Firecam* application by forwarding the packets at a constant rate over a fixed path. Similarly to the protocols proposed by Kim et al. [15] and Österlind et al. [21], PNP relies on the presence of another protocol that establishes the multi-hop path and, in order to achieve a high-throughput, assumes the absence of other network traffic. Also, PNP works more efficiently on top of a simplified MAC protocol, where most of the CSMA functionality is disabled, without CCA and CRC checks, and with a radio protocol where the auto acknowledgement is also disabled.

Next, we discuss experimental results that confirms the following important fact about the protocol stacks shown in Fig. 1: *each of them supports well the corresponding application, for which it has been optimized, while supporting poorly the other application.*

**Experimental Setup.** The School of Computing building at the National University of Singapore is a three-floor building that has been instrumented with a WSN testbed called Indriya [4], which consists of 119 active TelosB motes [22]. TelosB has a CC2420 radio, 8 MHz CPU, 10 KB RAM, 48 KB of program memory, and is a widely used hardware platform in WSN research. In the first set of experiments, which are discussed in this section, we remotely programmed the Indriya motes to support the *Firecam* and *Collection* applications separately without WSN reconfiguration (the reconfiguration experiments with Fennec Fox are discussed in Section IV.) In all our experiments we assumed that the sink node is located

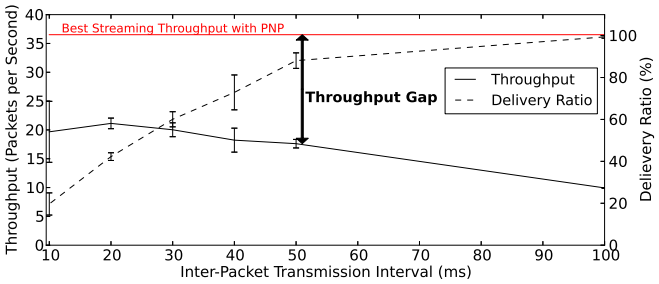



Fig. 3. CTP with CSMA does not support high point-to-point throughput.

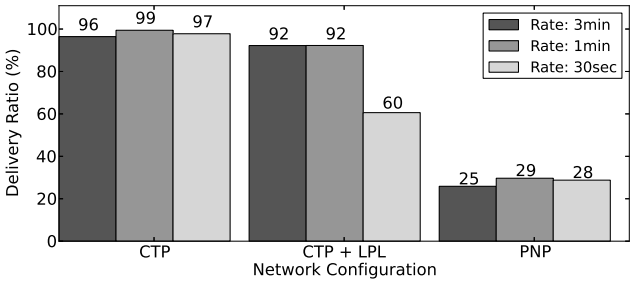



Fig. 4. PNP cannot support the same many-to-one delivery ratio as CTP.

at the corner of the first floor. For *Collection*, all remaining 118 nodes send data to the sink, while in the case of *Firecam*, the picture is streamed from a node located at the opposite corner of the building, on the third floor. The path between two opposite corners requires 7 to 9 hops. All experiments have been completed multiple times, over a period of two-weeks, during day and night hours, in midweek and weekend days.

**PNP Works Better than CTP to Support *Firecam*.** Fig. 2 shows how fast a picture from *Firecam* application can be streamed over a WSN with the PNP stack discussed above. In particular, it reports the results of multiple experiments to show how the network throughput (measured as the number of packets received at the sink per unit of time) and the delivery ratio (as previously defined) vary as function of the inter-packet transmission interval, which is varied at the step of 5ms in the range [5ms, 50ms]. For inter-packet transmission intervals equal or more than 30ms, the packets arrive with delivery ratio close to 100%. While the delivery ratio is still 97.4% for an interval value equal to 28ms, it drops to 50%, due to transmission collisions, for a 26ms interval value. The network throughput values are 33.96, 35.63, and 35.32 packets per second for 30, 28, and 27ms interval values, respectively. In summary, we consider a 28ms inter-packet transmission interval as the most adequate to transmit a picture as it allows a successful transfer within 21.5 seconds.

Next, we study how fast a picture from the *Firecam* application can be streamed over a network running the CTP stack with a CSMA MAC using 10 jiffies random backoff, CCA, CRC, and radio's auto acknowledgment. Notice, that we purposely disabled the LPL mechanism because it does not provide any help for the type of transmission that characterizes the *Firecam* application. Fig. 3 shows the corresponding experimental results in terms of network throughput and delivery ratio as the inter-packet transmission interval is varied at the step of 10ms in the range [10ms, 100ms]. It is clear that with this WSN configuration the *Firecam* application suffers a low delivery ratio. While streaming a packet every 100ms yields a delivery ratio close to 100%, this drops considerably to 88% and 60% for lower inter-packet transmission intervals equal to 50ms and 30ms, respectively. As highlighted in Fig. 3, there is a clear throughput gap between the performance of the two protocol stacks of Fig. 1 when running the same *Firecam* application. The top line marks the best throughput achieved by the PNP stack, which delivers over 97% of packets with a throughput of 36.52 packets per second. The CTP stack, instead, can only achieve 88% delivery ratio at the 50ms inter-packet interval, with a throughput of 17.61 packets per second: at this rate, a picture is transmitted in 38.4 seconds, which is more than twice as long as taken when streaming it with the PNP protocol.

In summary, based on the results of Fig. 2 and Fig. 3, we conclude that the *Firecam* application clearly benefits from a WSN deployment that uses the PNP stack. Next, it is natural to study how well this protocol stack can support the very different *Collection* application.

**CTP Works Better than PNP to Support *Collection*.** Fig. 4 compares the packet delivery ratio for the messages of the *Collection* application for three different configurations of the WSN protocol stack: CTP with CSMA and radio support, CTP with CSMA and Low Power Listening (LPL) duty-cycling at 100ms, and the PNP stack discussed above (i.e. without CSMA, CCA, CRC and acknowledgements.) Data are reported for three different transmission rates: 3 minutes, 1 minute, and 30 seconds. As expected, CTP achieves close to 100% delivery ratio. Even when LPL is enabled, CTP still performs well unless the sending rate becomes too high (the delivery ratio drops to 60% only when the 119 nodes are sending sensor measurements every 30 seconds.) The network configuration with PNP, instead, struggles to successfully deliver messages, as more than 70% of packets are lost. We conclude that traditional WSN applications for collecting sensor information cannot be effectively supported by the PNP stack.

**The Need for Dynamic Reconfiguration.** The above empirical study shows that two applications which have very different traffic characteristics require two different protocol-stack configurations in order to be properly supported. While all the experiments discussed so far have been run separately, we are interested in understanding to which extent the same WSN can effectively support two different applications such as *Firecam* and *Collection*. Running such heterogeneous applications with different network communication requirements is difficult because there is no WSN system that allows switching MAC protocols at runtime[3]. On the other hand, we are focusing on a heterogeneous application scenario that does not require simultaneous execution of the two applications. Instead, we are interested in a WSN that can run *Collection* as the default application and switch to running *Firecam*, which has a higher priority, only when an emergency event occurs. In other words, we want to deploy a WSN that: (i) can support multiple applications at different times and (ii) at any given time it uses the protocol stack configured to run those network, MAC, and radio protocols that are optimized for the current application. Since these protocols are different for different applications, the WSN needs to dynamically reconfigure the protocol stack to support their execution. For the case of our building environment application, the *Collection* application

[3]pTunes [26] only allows to reconfigure MAC parameters and Deluge [13] can change the MAC by reprogramming the whole sensor node firmware.

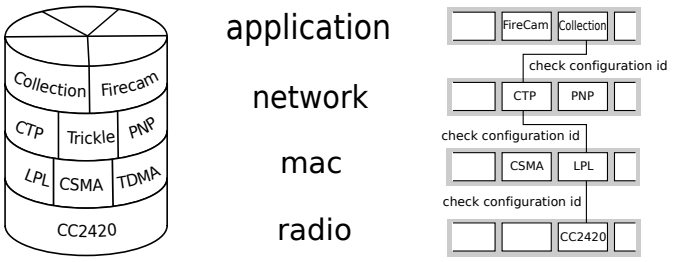



Fig. 5. The Fennec Fox four-layer protocol stack.

runs on top of the CTP stack, but when an emergency event occurs, the network reconfigures to the PNP stack to support the *Firecam* application. When the emergency is over, the network reconfigures back to run *Collection*.

## III. The Fennec Fox Framework

To support the dynamic reconfiguration of WSNs we developed the Fennec Fox framework that consists of a runtime infrastructure built around a layered protocol stack and a programming language to specify the various WSN configurations and the policy to switch among them.

**Framework Definitions.** Fig. 5 shows the four layers of the stack: radio, MAC, network, and application. Each layer provides a set of *services* that are used by the layer immediately above. Each layer contains one or more modules. A *module* is a software program that provides an implementation of the services of its layer. This implementation is typically optimized with respect to some metric, such as power consumption, reliability, throughput, network routing topology, etc. Hence, depending on the particular layer, a module can be: (1) an application such as *Firecam* or *Collection*; (2) a network protocol such as CTP or PNP; (3) a MAC protocol such as CSMA or TDMA; and (4) a driver of a particular radio. A module accepts zero or more parameters, whose values have impact on the module's execution. A *module instance* is a module with a specified set of values for its parameters. Two module instances are equivalent when they are both instantiated with the same parameter values.

A *protocol stack configuration*, or simply *configuration*, is a set of four module instances executing on the four-layer stack, one module for each layer. Each network stack configuration of a given WSN has a static, globally unique *configuration identifier* (id) defined at the WSN design time. Two configurations are equivalent when their module instances are equivalent.

A *network reconfiguration* is the process during which the WSN switches its execution between two non-equivalent configurations, i.e., two different stacks. A node starts this process either in response to a reconfiguration request from another node or by itself as a result of an internal event, sensor readings, or an an occurrence of a periodic event.[4] Once initiated, the nodes continue with reconfiguration by requesting surrounding nodes to reconfigure as well. During reconfiguration, a node stops all the modules running across the layers of the stack and starts execution of the modules defined in the new configuration.

**Framework Implementation.** The Fennec Fox software running on each node is implemented in nesC [7] on top of the TinyOS operating system [17]. The software stores information about various protocol stack configurations, events triggering network reconfiguration, statically linked layers' modules and information about parameters' values that are passed to each module when it starts execution. Each module has to comply with the Fennec Fox standardized interfaces, i.e. a module must have a management interface allowing the framework to start and stop execution of the module and it must comply with the interfaces of its layer.

[4]In this paper, we do not focus on how the nodes decide to initiate reconfiguration. Fennec Fox provides mechanism to reconfigure the stack once such a decision is made by a node or a group of nodes.

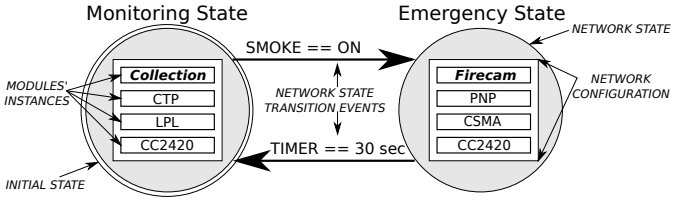



Fig. 6. A FSM model of a WSN supporting the *Collection* and *Firecam*.

The network protocol stack is implemented as a set of switch statements, which direct function calls and transfer packets among the modules based on the configuration id, as shown in Fig. 5. The id determines every function call made outside of the module's layer. To allow a radio driver to dispatch packets to the appropriate MACs, each radio defines location in a packet where it stores the configuration id, e.g. CC2420 radio driver stores the id's value in the Personal Area Network field of the IEEE 802.15.4 header [14]. The value of the packet's id is set to the id of the configuration of the stack in which that packet was created.

**Modeling Reconfigurations with FSMs.** The evolution of the behavior of a WSN that can dynamically reconfigure itself through the Fennec Fox framework can be captured in a simple way by using the Finite State Machine (FSM) model of computation. In particular, each protocol stack configuration can be modeled with a distinct state of the FSM and the process of reconfiguring the WSN between two particular configurations can be modeled with a transition between the corresponding states.

For example, Fig. 6 shows the FSM that models the reconfiguration of a WSN supporting the two applications as discussed in Section II with the optimized stacks shown in Fig. 1. The FSM has two states. The *Monitoring* state, which is also the initialization state, models the execution of the *Collection* application on top of the CTP stack, with the MAC and radio configured to minimize power dissipation and to avoid packet collisions. The *Emergency* state models the execution of the *Firecam* application on top of the PNP stack, with the MAC and radio configuration aimed at minimizing transmission delay and maximizing throughput. Further, the state transitions model the conditions that govern the reconfiguration of the WSN. The transition from *Monitoring* to *Emergency* specifies that this reconfiguration must occur when the smoke detector of a WSN node goes off so that the *Firecam* can start streaming a picture from the corresponding zone in the building. The transition from *Emergency* to *Monitoring* specifies that the opposite reconfiguration must occur when a certain time period has passed since the network has switched to the *Emergency* state: in this example, after a period of 30 seconds the network is brought back to execute the *Collection* application.

**High-Level Programming of WSN Reconfigurations.** To simplify the deployment of reconfigurable WSNs, we

```
1 # Definition of network configurations
2 # configuration <conf_d> [priority level] {<app> <net> <mac> <radio>}
3 configuration Monitoring {collection(2000, 300, 1024, NODE, 107)
4                          ctp(107) lpl(100, 100, 10, 10) cc2420(1, 1, 1)}
5 configuration Emergency L3 {firecam(1000, 28) parasite()
6                                                csma(0, 0) cc2420(0, 0, 0)}
7 # Events: event-condition <event_id> {<source> <condition> [scale]}
8 event-condition fire {smoke = YES}
9 event-condition check_if_safe {timer = 30 sec}
10
11 # Policies: from <conf_id> to <conf_id> when <event_id>
12 from Monitoring goto Emergency when fire
13 from Emergency goto Monitoring when check_if_safe
14
15 # Definition of the initial state: start <conf_id>
16 start Monitoring
```

Fig. 7. Swift Fox program reconfiguring WSN between two applications.

developed Swift Fox, a new domain-specific high-level programming language that has its formal foundation on the simple FSM model described above. Using Swift Fox, it is possible to specify at design time the behavior of a self-reconfiguring WSN, by scheduling the execution of each application and indicating the corresponding supporting stack configurations. A Swift Fox program allows us to control the four stack layers for each configuration by instantiating modules, initializing module parameters, and assigning unique ids to each configuration. Further, for each configuration we declare a *configuration priority level*, which plays an important role when multiple distinct reconfigurations occur at the same time in the network, as discussed below.

The semantics of the Swift Fox language supports the declaration of the sources of reconfiguration events and the threshold values that must be matched for an event to fire. The source of an event may come from a timer or a sensor. Boolean predicates can be specified using the basic relational operators (e.g. $==$, $<$, $>$) to compare sensor measurements and timer values with particular threshold values. The event-condition predicates are compiled into code that at runtime periodically evaluates the expression value. When the value is *true*, the occurrence of the event is signaled. The network FSM model is programmed by combining network state declarations with the event-conditions to form policy statements. Each policy statement specifies two network configurations and an event triggering network reconfiguration from one configuration to another. A Swift Fox program is concluded with a statement that specifies the initial configuration of the WSN.

The Fennec Fox software infrastructure relies on the definitions of the network configurations written in the Swift Fox program. This includes not only the logic to capture possible reconfigurations but also the list of modules that are executed across the layers of the stack for each particular configuration, i.e. which application and which network, MAC, and radio modules together with the values of their parameters. The Swift Fox programs are compiled into nesC code that links together all the modules that are specified for a given configuration and generates switch statements that direct function calls and signals among the modules.

As an example, Fig. 7 shows a Swift Fox program for a WSN that reconfigures between the *Monitoring* and *Emergency* states according to the state transition diagram of the FSM of Fig. 6 in order to support the execution of the *Collection* and *Firecam* applications, respectively. Lines 3-6 declare the two network configurations with ids *Monitoring* and *Emergency*. The *Monitoring* configuration consists of the *Collection* application module that starts sensing after *2000ms* since the moment it receives the start command on the management interface. From every *NODE*, the module sends messages with the sensors' measurements at the rate of *300* seconds (*1024ms*). The messages are sent to a sink node whose address is *107*. Indeed, the configuration uses the *ctp* module, which runs the CTP network protocol with a root node at the address *107*. The *Collection* configuration runs also a MAC protocol with Low-Power Listening (*lpl*), a *100ms* wakeup period and stay-awake interval, together with *10jiffies* random backoff, and *10jiffies* minimum backoff CSMA's parameters. The configuration is supported by a radio driver enabling all three services: auto-acknowledgements, CCA and CRC. The specification of the *Emergency* configuration is similar but it is characterized by a higher priority level (set to 3 while the default is 1) and by the use of PNP with all MAC and radio services disabled. Lines 8-9 declare two reconfiguration events. The first event, *fire*, occurs when a sensor detects the presence of smoke. The second event, *check_if_safe* takes place *30 seconds* after it is initiated. Lines 12-13 declare the network state reconfiguration policies: the network reconfigures from *Monitoring* to *Emergency* when *fire* occurs; similarly, it reconfigures from *Emergency* back to *Monitoring* when the *check_if_safe* occurs. Line 16 sets *Monitoring* to be the initial state.

The same Swift Fox program is deployed on every node of the given WSN. While Swift Fox allows us to program a reconfigurable WSN, the language does not allow programmers to specify how a particular node detects an event, how it reconfigures itself, and how it can trigger the reconfiguration of all the other nodes in the network. Indeed, Swift Fox is meant to provide a high-level abstraction that intentionally hides the underlying mechanisms governing the WSN reconfiguration.

**Runtime Network Reconfiguration.** A node decides to reconfigure when the result of an event matches the reconfiguration policy in the Swift Fox program. Then, the node requests other nodes to reconfigure by broadcasting a *Control Messages* (CM), a single 4-byte packets that contains the id and the *sequence number* of the new configuration. The sequence number is incremented by one after each network reconfiguration. Based on the sequence number, nodes can distinguish a new configuration from an old one. As a result of the nodes re-broadcasting CM packets to other nodes during reconfiguration, the whole WSN reconfigures itself.

The network reconfiguration process requires dissemination of CM packets in the presence of various MAC protocols scheduled to run on the stack at a given time. CM packets are distinguished from other packets by their own configuration id, which allows radio drivers to dispatch CM packets to Fennec Fox. To enable transmission of CM packets during operation of other MACs or radio duty-cycling, Fennec Fox monitors the radio status together with function calls and packets crossing the layers of the stack, making decision on when CMs should be transmitted such that other nodes will receive the message, i.e. the CM broadcasts are suspended when a radio is turned off or other transmissions are ongoing.

The CM dissemination process has been successfully tested to reconfigure the network among TDMA, CSMA, and duty-cycling versions of these protocols However, TDMA to TDMA reconfigurations may not be successful when both MACs duty-cycle with the same period but end up at different offset. This

**Algorithm 1** Broadcast Control Process (BCP)

```
1: retry ← r
2: while retry > 0 do
3:     counter ← 0
4:     WAIT(d)
5:     if counter < t then
6:         BROADCAST_CM
7:     end if
8:     retry ← retry − 1;
9: end while
```

**Algorithm 2** Processing Received Control Message

```
1: Input: msg
2: if !CRC(msg) || msg.state ∉ ALL_STATES then
3:     EXIT
4: end if
5: msg_version ← concat( msg.sequence, PRIORITY(msg.state) )
6: node_version ← concat( node.sequence, PRIORITY(node.state) )
7: if msg_version < node_version then
8:     BCP ; EXIT
9: end if
10: if msg_version > node_version then
11:     RECONFIGURE ; EXIT
12: end if
13: if msg.state = node.state then
14:     counter++
15: else
16:     node.sequence += RANDOM
17:     BCP
18: end if
```

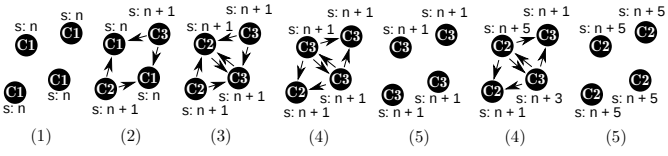
problem is mitigated by introducing a transition configuration with a CSMA MAC that runs between the two TDMA-based configurations.

The CM broadcast functionality, which co-existing with other MACs, supports network reconfiguration operating on a modified Trickle [18] algorithm. First, no messages are disseminated when network reconfiguration does not take place and all nodes in the network run the same configuration. Second, to ensure that all nodes run the same stack after switching their state, not only the sequence number but also the content of the CM is used during network reconfiguration.

To distribute CMs a node follows the *Broadcast Control Process* (BCP), which is specified as Algorithm 1. In particular, the node attempts to broadcast the CM every $d$ ms (line 4,6). A node abstains from broadcasting when it receives $t$ identical CMs sent by other nodes within the last $d$ ms (lines 5-7)[5]. The BCP terminates after $r$ broadcasts attempts (lines 1-2, 8-9).

A node enters the BCP as a result of one of three possible situations. First, after a node has completed a stack reconfiguration it enters BCP to request other nodes to switch to the same configuration. Second, when a node receives a data packet with a configuration id that is different from its current configuration, it assumes that it either missed the last network reconfiguration or the node transmitting the packet has missed it; to resolve this situation the node enters BCP[6]. Third, the reception of a CM may lead also to the execution of BCP, depending on the values of the configuration id of the new state and sequence number in the control message as well as the corresponding current values stored in the node; all these values are processed by the node executing Algorithm 2.

Algorithm 2 specifies the decision process followed by a node after receiving a new CM. First, this message is validated by checking its CRC code and the value of the configuration id that it carries. If either CRC fails or the configuration id




(a) Two events (b) Definite (c) Random

Fig. 8. Network synchronization: (b) deterministic and (c) non-deterministic.

value is different from all known configuration ids, which are specified at design time, then CM is ignored (lines 2-4). The algorithm decides to run BCP or trigger node reconfiguration by comparing CM's configuration version with node's configuration version, which are computed by concatenating the sequence number and priority of the configuration id from the CM and node, respectively (lines 5-6). The comparison of both CM and node configuration versions leads to the following decisions. If CM has a sequence number less than the node's sequence number or the sequence numbers are equal but the CM's configuration has lower priority than the node's configuration, then the node enters BCP (lines 7-9). If CM has a sequence number higher than the node's sequence number, or the sequence numbers are equal but the CM's configuration has higher priority than the node's configuration, then the node switches to the new configuration (lines 10-12). If a node has the same sequence number and configuration id as CM has, then the node increases $counter$ by 1 (line 14).

When CM and a node have equal sequence numbers but different configuration ids then the network is unsynchronized. This situation occurs when two nodes simultaneously decide to run different configurations. Then these nodes start BCP with the same sequence number, but different configuration ids. This is illustrated in Fig. 8(a) showing two nodes in the corners of the network reconfiguring from a configuration with id $C_1$ and sequence number $n$ to two different configurations $C_2$ and $C_3$, both with sequence $n+1$. The nodes in the middle of the network detect reconfiguration inconsistency. If the configuration ids of the conflicting CMs have different priorities, then the network is deterministically synchronized to the configuration with the higher priority (lines 7-12). This is shown in Fig. 8(b) where the conflict between $C_2$ and $C_3$ is solved by synchronizing to $C_3$ because Priority($C_3$) > Priority($C_2$). When the network is unsynchronized among states with undefined[7] or equal priorities then the nodes that detect the conflict increase their sequence numbers by a random value and start BCP while keeping their current configuration (lines 16-17). As shown on Fig. 8(c) where Priority($C_2$) = Priority($C_3$), after randomly increasing sequence number ($+5$ and $+3$, for $C_2$ and $C_3$ respectively), the node that broadcasts CMs with the highest sequence ($5 > 3$) will synchronize the rest of the network to its own configuration, i.e. $C_2$.

In summary, the Fennec Fox network reconfiguration mechanism has the following properties: (1) it controls the execution of the four-layer stack and applies the specification of the network behavior given in the Swift Fox program; (2) it has zero overhead when no reconfiguration takes place; (3) the network reconfiguration does not require any hardware support; and (4) it is guaranteed to resolve any possible reconfiguration conflict that may arise given the distributed nature of the mechanism.

[5]This transmission suppression avoids unnecessary radio broadcasts, in a way similar to the Trickle protocol [18].

[6]Recall that every packet carries its configuration id and therefore every packet can be used to detect network configuration inconsistency.

[7]Recall that the Swift Fox language allows, but not mandates, programmers to specify the network configuration's priority level. By default each configuration priority level is set to the lowest possible value.

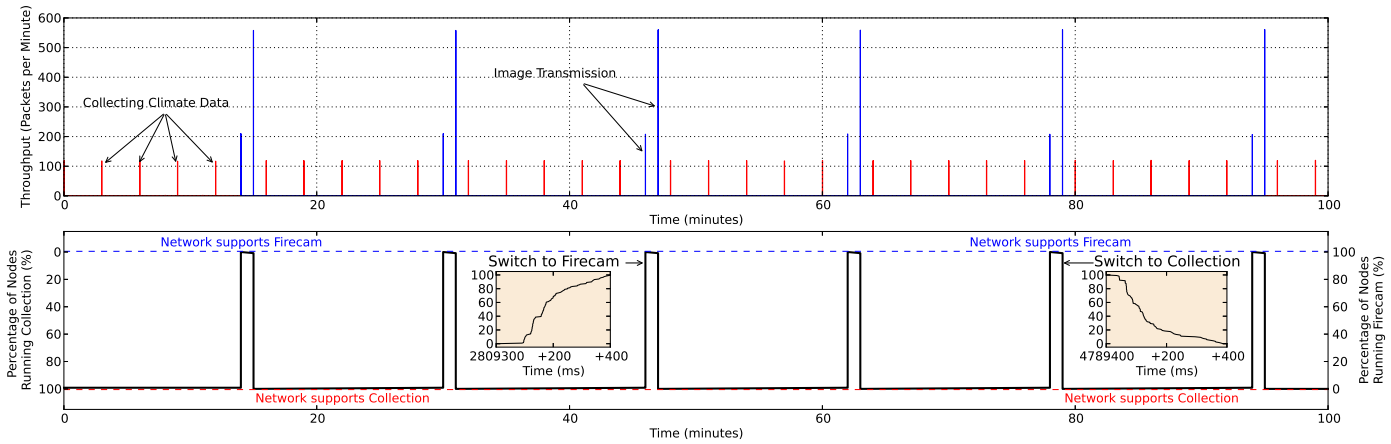



Fig. 9. A 100 minute run of a network reconfiguring between the *Collection* and *Firecam* applications.

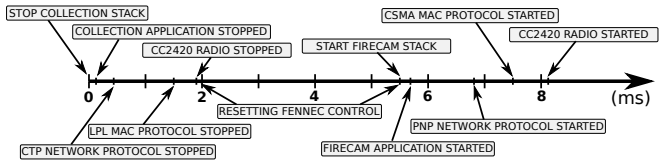



Fig. 10. Protocol stack reconfiguration from *Collection* to *Firecam*.

## IV. Evaluation

The goal of our experiments is to study the feasibility and performance of dynamic WSN reconfiguration. We show the memory overhead and time that it takes to reconfigure the protocol stack on a single node. We present network reconfiguration experiments with the setup as described in Section II. We demonstrate the feasibility of our approach and measure the overhead of reconfiguration between stacks running different MACs. We also study BCP algorithm configurations that successfully disseminate CM packets.

**Code and Memory Overhead.** On TelosB, the reconfiguration protocols and mechanisms introduce an overhead of 4.7 KB of ROM and 0.2 KB of RAM. The Swift Fox program with both *Collection* and *Firecam* requires 28.9 KB of ROM and 5.6 KB of RAM.

**Single-node Reconfiguration Delay.** Fig. 10 shows all the events and their timings when a node switches from running *Collection* to running *Firecam*. Once reconfiguration is initiated, Fennec Fox first stops the currently running *Collection* application module and then stops the CTP network protocol, LPL MAC, and CC2420 radio modules. This process requires a total of 1.969ms. Next, the reconfiguration engine is reset, which takes 3.469ms, of which 2.9375ms is spent resetting the radio device. Finally, Fennec Fox starts the CC2420 radio, CSMA MAC, PNP network protocol, and *Firecam* application, which takes a total of 2.686ms. The whole network stack reconfiguration takes 8.125ms. We observe similar reconfiguration delays among other WSN configurations.

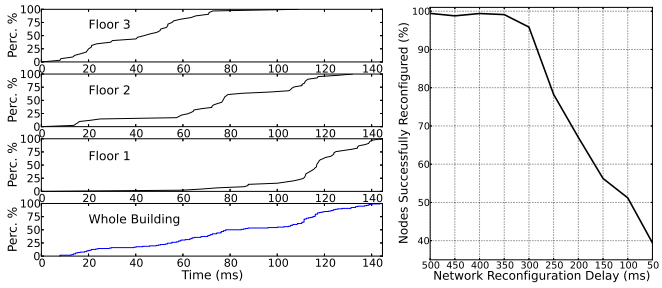
**WSN Switching between *Collection* and *Firecam*.** We first determine if it is feasible to reconfigure a network between *Collection* and *Firecam* applications correctly, quickly, and efficiently using the proposed Fennec Fox framework. In these experiments, the BCP algorithm (Algorithm 1) runs with $d = 18$, $t = 2$ and $r = 1$. We set node 107 located at one corner of the testbed building to be the sink node.

*Feasibility of Reconfiguration*. First, we run repeatedly (36 times) the following experiment: after *Collection* has executed for 5 minutes, a random node on the network triggers reconfiguration to *Firecam*. The results show that on average 99.98% of the nodes complete reconfiguration by successfully switching from *Collection* to *Firecam*. This demonstrates the feasibility of our approach. In fact, as discussed below, even in a duty-cycled network, 99.5% of the nodes are successfully reconfigured.

*Multiple Reconfigurations*. The next question is whether our system is robust enough to perform multiple reconfigurations. We performed the following sequence of tasks for 100 minutes: run *Collection* for 15 minutes before letting a node trigger reconfiguration to *Firecam*; then, after 1 minute, the network is reconfigured back to run *Collection* and the process is repeated. The lower graph in Fig. 9 shows the percentage of nodes executing *Firecam* at a given time: except during the transition between the configurations, all the nodes are running either *Collection* or *Firecam*. This transition occurs 12 times as shown in Fig. 9. As the network transitions between execution of *Collection* and *Firecam*, we expect to see the network throughput observed from the sink to transition between low and high throughput. This is confirmed by the results shown in the upper graph of Fig. 9. reporting the throughput sampled every minute. The timing of these transitions match the timing of reconfigurations. This provides additional evidence that the reconfigurations indeed make the network transition between two completely different applications; furthermore, the applications and their protocol stack are not impaired by the proposed reconfiguration mechanism.

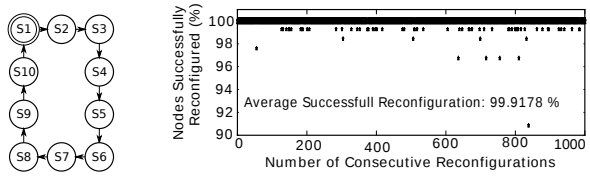
*Network-wide Reconfiguration Delay*. The graphs that are embedded in the lower graph of Fig. 9 show the network reconfiguration at time scale of milliseconds, thereby highlighting how much time it takes for the network to transition between two configurations. The first embedded graph shows the number of nodes executing *Collection* just before the reconfiguration: a rapid reconfiguration of 80% of nodes occurs within less than 100ms, while the rest of the nodes transition in the next 200ms to start *Firecam*. Similarly, the second embedded graph shows that close to 80% of the nodes reconfigure quickly, while the remaining 20% of transitions happen with the next 200ms, to switch from *Firecam* back to *Collection*.

The reconfiguration delay depends on the network distance between the nodes being reconfigured. Fig. 11(a) shows average results from 50 experiments where a node, located in a



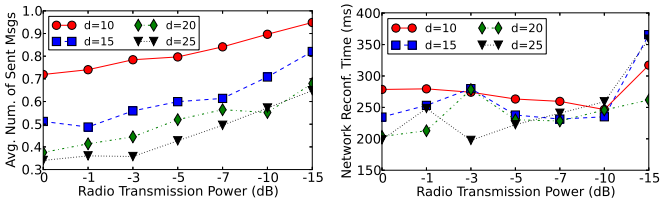

(a) Building Reconfiguration (b) Transition Rate

Fig. 11. Reconfiguration performance with radio duty-cycled.



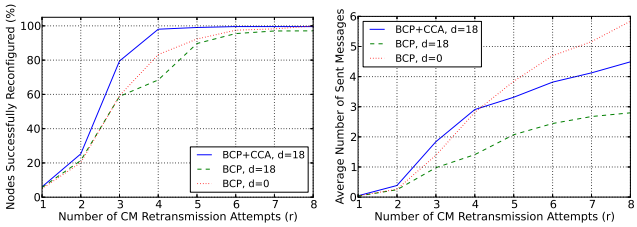

(a) FSM (b) 1000 Reconfigurations

Fig. 12. Network reconfiguration firing every 500ms.




(a) CM Broadcasts (b) State Transition Delay

Fig. 13. Radio TX power impact on reconfiguration overhead and delay.




(a) Successful Reconfigurations (b) CM Broadcasts

Fig. 14. Reconfiguration from a network with duty-cycling MAC protocol.

corner of the 3rd floor, initiates a reconfiguration every minute. For each floor, reconfigurations occur in bursts: this is due to a single broadcast packet initiating the process and being able to trigger reconfiguration on all the nodes that receive that broadcast. After running multiple experiments with nodes initiating reconfiguration placed all over the building, we found that those located on the same floor as the node that starts the process switch to the new configuration within 49.81-71.54ms; instead, the nodes in the adjacent floor take a time within 95.67-153.67ms, and the nodes that are two floors away need 134.41-141.01ms.

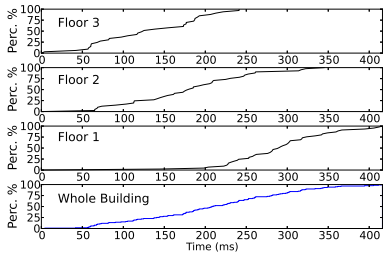
**Maximum Reconfiguration Rate.** Fig. 11(b) shows the percentage of nodes that successfully reconfigure as function of the network reconfiguration delay, varies between 50ms and 500ms. With the reconfiguration delay not less than 350ms, almost 100% of the nodes reconfigure on time. However, as the reconfiguration delay decreases further, the percentage of nodes that successfully reconfigure also decreases. These results demonstrate that it is feasible to reconfigure a network successfully, if necessary, multiple times, and quickly.

**Beyond Two Applications.** We wrote a Swift Fox program with 10 configurations of simple functionality to emulate 10 different applications. Fig. 12(a) shows the model of the network with a reconfiguration event fired every 500ms. We ran an 8-hour experiment from which the first 1000 network reconfigurations are shown in Fig. 12(b), marking the percentage of nodes successfully reconfigured at particular transition. On average, 99.91% of network reconfigurations are successful. In another experiment we let the same network reconfigure at the rate of 350ms for 5289 times. In that experiment, 99.68% of nodes successfully follow each network configuration, sending 0.4 messages per configuration transition.

**Factors Impacting Reconfiguration.** The network-reconfiguration success rate and overhead, which is the number of CM broadcasts and the time reconfiguration delay, depend on multiple-factors: network density, BCP's parameters and the MAC protocols that are scheduled to run at a given configuration. We evaluate the network reconfiguration performance with various MAC protocols: IEEE 802.15.4-complaint CSMA, LPL duty-cycling with a sleep interval of 100ms, a TDMA duty-cycling MAC protocol, and a NULL MAC that transmits without regard to other possible transmissions in the network. In particular, TDMA presents characteristics that match the most challenging aspects that one encounters in the Fennec Fox framework: specifically, the fact that outside the designated time slots packets cannot be received (e.g., CM packets) complicates the operations of the Fennec Fox network-reconfiguration mechanism. Still, in the sequel we present experiments showing how Fennec Fox can handle the presence of the TDMA-style packet timing and, indeed, allow a network to switch between CSMA and TDMA and vice versa.

**Network Density.** To emulate networks with different densities, we set the CC2420 radio on TelosB motes to transmit at power of $0dB$ to $-15dB$. We found that the reconfiguration is uniformly successful across all the densities, i.e., experiments spanning the entire range of transmission power, except with large values of reconfiguration delay. Network density, however, has a more visible impact on other metrics. Fig. 13(a) shows that the efficiency of reconfiguration increases at higher density. This is because at higher densities the reconfiguration algorithm suppresses more CM broadcasts in a way similar to the Trickle protocol. Depending on the $d$ value, at the highest density there are 42.1% fewer broadcast transmissions compared to the experiment with the lowest density. Fig. 13(b) shows that a network with higher density (but the same number of nodes) reconfigures faster. This trend, however, is not observed across all density values.

**CM Broadcast delay - $d$.** By delaying the CM broadcast by $d$ ms, we allow nodes to suppress their transmission in case other nodes in the neighborhood are already transmitting the reconfiguration information. As expected, we found that a longer broadcast delay leads to reconfiguration with less CM transmissions. Depending on the density, with a broadcast delay of 10ms, there are on average 0.32 to 0.45 fewer transmissions than with a broadcast delay of 25ms as shown in Fig. 13(a). Smaller broadcast delays increase network reconfiguration by as much as 178ms due to the higher probability of transmission collisions. This result, however, is not conclusive for the experiment with the lowest density having lower packet collision probability. The higher CM broadcast rate and the longer reconfiguration time impact positively the success reconfiguration rate, which for $d$ =10ms, 15ms, 20ms and 25ms is on average 99.9%, 99.39%, 98.54%, and 97.94%,




| Duty-Cycle | Retry (r) | CM TXs | Delay (ms) |
|---|---|---|---|
| 0 | 1 | 0.4 | 80 |
| 100 | 8 | 6.4 | 209 |
| 200 | 14 | 9.2 | 278 |
| 300 | 15 | 9.9 | 387 |
| 400 | 21 | 13.1 | 449 |
| 500 | 25 | 15.1 | 559 |

Fig. 15. LPL impact on network reconfiguration.

respectively. In actual deployments we set $d = 18$. Fig. 14(a), shows that with $d = 18$ and while using CCA before CM transmission, we can reconfigure almost 100% of the nodes for $r$ greater than 5. With $d = 0$, the success rate is 94-99% for the same range of $r$. On the other hand, Fig. 14(b) suggests to avoid CCA and setting $d = 18$ as such CM dissemination consistently requires less CM broadcasts. Because we favor reconfiguration success rate over the reconfiguration transmission overhead, in actual deployments we use CCA.

**Impact of Radio Duty-Cycling on the Number of Broadcast Attempts - $r$.** Fig. 14 shows that to reconfigure WSN between CTP stack and PNP stack, BCP should be set with $r = 5$ and $d = 18$ while using CCA. Without CCA and $d = 0$, setting $r = 8$ successfully reconfigures the network. When $r = 1$ at most 6.06% of the network successfully reconfigures. As $r$ increases, the number of nodes that successfully switch between two configurations also increases. Not surprisingly, as the number of CM broadcast attempts grows, the actual number of radio transmissions grows as well (Fig. 14(b)): from close to 0 for $r = 0$ to about 2-4 transmissions per node for $r = 6$. In this case, however, larger values of $r$ are still desirable because these additional transmissions contribute to reach a success rate close to 100%.

In many experiments where the radio is always turned on we notice that setting $r = 1$ is sufficient to achieve reconfiguration rate close to 100%. This shows that the presence of other MAC protocols, i.e. MAC protocols that duty-cycle radio operation, has a significant impact on the network reconfiguration. In Fig. 15 we show how BCP with $d = 18$, $r = 6$ and CCA[8] reconfigures the network across the floors of the testbed building in the presence of a stack configuration with LPL MAC. Examining this graph together with Fig. 11(a) shows how duty-cycling impacts the latency. With duty-cycling, the complete network reconfiguration takes more than twice as long. Fig. 15 shows how the network reconfigures progressively, instead of in bursts as in the experiment without LPL. The linear progress of network reconfiguration is the results of radio's LPL with unsynchronized wakeup interval.

We further explore the reconfiguration performance in the presence of LPL MAC with sleep interval beyond 100ms. Fig. 15 shows the results for the case of a network that switches between two configurations every 5 minutes while using duty-cycling with different sleep intervals. The table reports the minimum values of $r$ required for high success rate. The results show that longer sleep intervals require more retransmission attempts, up to $r = 25$ when radio periodically sleeps for 500ms. As the number of CP broadcasts attempts grows, the actual transmission count stays around 60-66% of $r$. As the sleep period increases, the WSN reconfiguration takes longer.

[8]With those parameters we are optimizing for WSN success reconfiguration rate over the number of CM broadcasts and the WSN reconfiguration delay.

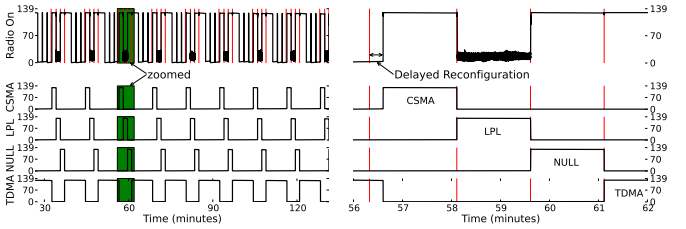



Fig. 16. Reconfiguration among various MACs.

The delay more than doubles between 200 and 500ms.

**Multiple MAC protocols.** The presence of other MAC protocols have a large impact on the performance of WSN reconfiguration. We have already discussed how protocols such as LPL complicate network reconfiguration. Therefore, we conclude the evaluation section with the presentation of an 8-hour experiment, where the network is reconfigured among four network stacks: one using CSMA, one LPL (200ms), one Null MAC, which simply forwards traffic between the network and the radio layer, and the final one, which uses duty-cycling TDMA MAC[9]. Fig. 16 reports results from the run with the network reconfigured among these four MACs. The left side of the figure shows a sample part of the experiment while the right side shows zoomed-in green section of the left side. The red vertical lines mark moments when events triggering a network reconfiguration take place. The 4-bottom graphs show how many nodes are running with each MAC protocol. Specifically, the upper graph shows how many nodes at the given time have their radio turned on. When CSMA is scheduled to run, all nodes keep their radio on, as expected. When LPL is scheduled, the nodes turn the radio on only periodically and their periods are not synchronized. When TDMA is running, the nodes stay on for a while to synchronize their global time and then periodically turn the radio off and on, according to the time schedule.

Fig. 16 (right) shows how the reconfiguration mechanism handles the situation when the network is in a synchronized sleep mode. The figure starts with all the nodes in the TDMA configuration and their radios turned off. Then, an event occurs that triggers network reconfiguration. Instead, of starting CM broadcast immediately after the event, sending a packet that no other node would receive, the CM broadcast transmission is delayed until the TDMA protocol starts radio again. This delay prevents the broadcasting of CM messages when not a single node would receive CM because the whole network is synchronously shutdown. These experiments show that Fennec Fox can reconfigure a WSN among the four MAC protocols (CSMA, LPL, Null, TDMA), which by themselves could not co-exist in the same WSN.

## V. Related Work

A major approach to WSN reconfiguration is based on distributing fragments of code that are loaded and executed on the sensor nodes. TinyCubes [19], Enix [3], BASE [11], and ViRe [1] are examples of such systems. Works such as Deluge can perform full-program update on the nodes [13]. Some systems allow users to program WSNs at run-time. Maté [16] allows dissemination of code to be executed in a virtual machine. Tenet [8] allows sending data-flow programs to be executed in the network. Our approach does not send code

[9]This experiment is performed on the same testbed described in Section II but after its expansion to 139 nodes.

updates at runtime. Instead, the Fennec Fox framework allows users to specify many applications, each with a dedicated protocol stack, and the conditions under which the WSN self-reconfigures from one application to another.

While previous works show that it is possible to perform incremental or wholesale code updates in the network, these updates must still be disseminated efficiently among the nodes. Efficiency can be achieved by: selecting the subset for update (as in FiGaRo [20]), using a shared infrastructure [24], or minimizing redundant broadcast transmissions as done by Trickle [18]. Fennec Fox uses an approach similar to Trickle to perform efficient network-wide self-reconfiguration.

In the early days of WSN research, most of the protocols were cross-layered, making them difficult to reuse across the applications. Nowadays, there are many sensor network stacks such as Rime [6], uIP [5], and TinyOS IP stack [12] that try to follow the layered protocol model. Besides these complete stacks, there are now layer-specific protocols (e.g., CTP [9], XMAC [2]) that are designed to allow different protocols at the higher or lower layers of the stack. We leverage the design and implementation of these protocols and provide a framework that allows applications to compose their own stack using protocols of their choice at each layer.

MultiMAC [25] shows that different MACs can be implemented on top of the same radio driver. The pTunes project [26] shows the need for runtime MAC's parameters adjustment and demonstrate it on one protocol. Fennec Fox takes these concepts further by scheduling execution of different MACs that can be initialized with various parameters.

Fennec Fox has been demonstrated to work with other radios than CC2420, i.e. CC1000 [10] and UWB-IR [23].

## VI. Conclusions

We study the problem of executing a set of heterogeneous applications with different communication requirements on a single WSN. Our solution consists in the dynamic self-reconfiguration of the WSN such that it runs the combination of network and MAC protocols that suits well a given application. To do so, we developed the Fennec Fox framework composed of a runtime infrastructure built around a layered protocol stack and a programming language to specify the various WSN configurations and the policy to switch among these in response to various events. Our experimental evaluation showed that our approach can successfully reconfigure a large WSN in few hundreds of milliseconds while incurring little control overhead.

While we demonstrated that is possible to have the WSN self-reconfigure between different MAC protocols such as a CSMA MAC and a TDMA-like MAC, future work needs to address the challenge of switching effectively large WSNs between multiple TDMA protocols which may disable the radio for periods of time that may never overlap. Supporting frequency-hopping and nodes with multiple radios are other research venues to be pursued with Fennec Fox. To truly support a large number of complex applications operating on a single WSN supported with multiple protocols, we need WSN nodes with larger RAM and ROM memories than those available in current mote-class platforms. As more resourceful platforms are already on the horizon, we believe that our approach offers a new path to investigate a new generation of heterogeneous WSN applications.

**Acknowledgements:** We thank Prof. Ping Ji from John Jay College of Criminal Justice (CUNY) for early discussion on reconfigurable network protocols. We thank Manjunath Doddavenkatappa from the National University of Singapore for providing support with the Indriya testbed. This project is partially supported by the National Science Foundation under Awards #644202 and #931870 and by an ONR Young Investigator Award.

## References

[1] R. Balani et al. Vire: Virtual reconfiguration framework for embedded processing in distributed image sensors. In *APRES Work.*, Apr. 2008.

[2] M. Buettner et al. X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks. In *Proc. of the ACM SenSys Conf.*, pages 307–320, Nov. 2006.

[3] Y.-T. Chen, T.-c. Chien, and P. H. Chou. Enix: a lightweight dynamic operating system for tightly constrained wireless sensor platforms. In *Proc. of the ACM SenSys Conf.*, pages 183–196, Nov. 2010.

[4] M. Doddavenkatappa, M. C. Chan, and A. Ananda. Indriya: A low-cost, 3D wireless sensor network testbed. In *TRIDENTCOM*, pages 302–316, Apr. 2011.

[5] A. Dunkels. Full TCP/IP for 8 bit architectures. In *Proc. of the MobiSys Conf.*, pages 85–98, May 2003.

[6] A. Dunkels, F. Österlind, and Z. He. An adaptive communication architecture for wireless sensor networks. In *Proc. of the ACM SenSys Conf.*, pages 335–349, Nov. 2007.

[7] D. Gay et al. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the PLDI*, pages 1–11, May 2003.

[8] O. Gnawali et al. The Tenet architecture for tiered sensor networks. pages 153–166, Nov. 2006.

[9] O. Gnawali et al. Collection tree protocol. In *Proc. of the ACM SenSys Conf.*, pages 1–14, Nov. 2009.

[10] M. Gorlatova et al. Prototyping energy harvesting active networked tags: Phase II mica mote-based devices. In *MobiCom10*, Sept. 2010.

[11] M. Handte et al. The BASE plug-in architecture - composable communication support for pervasive systems. In *Proc. of the ICPS Conf.*, page 443, July 2010.

[12] J. Hui and D. Culler. IPv6 in low-power wireless networks. *Proc. of the IEEE*, 98(11):1865–1878, Nov. 2010.

[13] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of the ACM SenSys Conf.*, pages 81–94, Nov. 2004.

[14] Institute of Electrical and Electronics Engineers. IEEE 802.15: Wireless Personal Area Networks (PANs). [Online] http://standards.ieee.org/about/get/802/802.15.html.

[15] S. Kim et al. Flush: a reliable bulk transport protocol for multihop wireless networks. In *Proc. of the ACM SenSys Conf.*, pages 351–365, Nov. 2007.

[16] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proc. of the ASPLOS Conf.*, pages 85–95, Oct. 2002.

[17] P. Levis et al. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–144, Nov. 2004.

[18] P. Levis et al. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. of the NSDI Symp.*, pages 15–28, Mar. 2004.

[19] P. J. Marrón et al. TinyCubus: a flexible and adaptive framework sensor networks. In *Proc. of the EWSN Conf.*, pages 278–289, Jan. 2005.

[20] L. Mottola, G. P. Picco, and A. A. Sheikh. FiGaRo: Fine-grained software reconfiguration in wireless sensor networks. In *Proc. of the EWSN Conf.*, pages 286–304, Jan. 2008.

[21] F. Österlind and A. Dunkels. Approaching the maximum 802.15.4 multi-hop throughput. In *Proc. of the HotEmNets*, June 2008.

[22] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Proc. of the IPSN Conf.*, pages 364–369, Apr. 2005.

[23] G. Stanje et al. Demo: Organic solar cell-equipped energy harvesting active networked tag (EnHANT) prototypes. In *Proc. of the ACM SenSys Conf.*, pages 385–386, Apr. 2011.

[24] A. Tavakoli, A. Kansal, and S. Nath. On-line sensing task optimization for shared sensors. In *Proc. of the IPSN Conf.*, pages 47–57, Apr. 2010.

[25] D. van den Akker and C. Blondia. MultiMAC: A multiple MAC network stack architecture for TinyOS. In *Proc. in the ICCCN Conf.*, pages 1–5, Aug. 2012.

[26] M. Zimmerling et al. pTunes: Runtime parameter adaptation for low-power MAC protocols. In *Proc. of the IPSN Conf.*, pages 173–184, Apr. 2012.