A System Architecture for Managing Complex Experiments in Wireless Sensor Networks

Jianjun Wen, Zeeshan Ansar, Waltenegus Dargie

Chair for Computer Networks
Faculty of Computer Science
Technical University of Dresden
01062 Dresden, Germany
Email:{jianjun.wen, zeeshan.ansar, waltenegus.dargie}@tu-dresden.de

***Abstract*—Several reproducible experiments are required before actual deployment of wireless sensor networks takes place if stable and predictable outcomes of protocols and data processing algorithms are desired. Considering the typical size of wireless sensor networks and the number of parameters that can be configured or tuned, conducting repeated and reproducible experiments can be both time consuming and costly. The conventional way of evaluating the performance of different protocols and algorithms under different network configurations is by changing the source code and reprogramming the testbed, which requires some effort. In this paper, we propose a traffic flow control management system that facilitates the execution of repeated experiments in an efficient and flexible way. We implemented our system on top of TinyOS for the TelosB platform and demonstrated the scope and usefulness of the system by conducting several experiments in two real testbeds.**

***Index Terms*—Experiment, experiment management, testbed, TFCP, wireless sensor networks**

I. Introduction

In order to obtain predictable performance and reproducible results from wireless sensor networks, complex and repeated experiments should be conducted with testbeds before actual deployments take place. Since most applications have their unique characteristics and requirements, the testbeds should be flexible and effectively separate the experiment phase from the application development and network management phases. In the past decades, the research community has made a considerable progress in developing reliable and flexible testbeds. Some of these provide web interfaces so that application developers can install program images on remotely available networks and execute code. Then experiment results (sensed as well as performance related data) can be extracted from the networks and delivered to the developers for off-line analysis and debugging. Some of these testbeds, besides providing common services, such as infrastructure management, experiment control (experiment scheduling and resource reservation), and data collection, enable also the inclusion of domain specific services, such as sensor data profiling [4], mobility management [10], and distributed and on-line tracing/debugging [16].

In most testbeds experiment procedures are embedded into the application logic and, hence, intra-experiment activities (such as an activation or deactivation of the collision avoidance functionality of a MAC protocol or the modification of network parameters) have to be carefully planned before a program is compiled and flashed to individual nodes. Arbitrary configurations cannot be carried out without affecting the execution of the application logic. Furthermore, the specification of complex procedures comes at the price of developing complex application-layer services. If application developers wish to introduce new procedures unforeseen at the time of uploading their image the only option they have is modifying their image and reinstalling it, which is a tedious and time consuming process. Furthermore, embedding experiment procedures in the application logic have another side effect, namely, experiment execution times will be subject to timers' error due to drift. To alleviate this problem time synchronisation should be necessarily a part of the application logic. Otherwise, experiments may not be reproducible. Finally, most existing testbeds provide experiment data management at the server side but not for individual nodes. There are no common interfaces or library files available for application developers to seamlessly gather data and performance indicators from individual nodes.

In this paper we propose a comprehensive traffic flow controller which integrates a set of toolkits for seamlessly performing changes to and configure protocols and algorithms during experiments. Our contributions can be summarised as follows:

1) We define a set of primitives to control experiments as they are being conducted. These primitives provide simple and enhanced controlling strategies.
2) We propose a light-weight protocol to communicate commands that control experiments at runtime.
3) We propose system architecture to integrate, process, and manage the traffic flow control commands.
4) We implemented the system architecture on top of TinyOS for the TelosB platform and employed it for different testbeds to demonstrate the usefulness of our approach.

The remaining part of this paper is organized as follows: In Section II, we review related work and position our own work. In Section III, we provide an example to highlight the difficulty associated with conducting moderately complex experiments in wireless sensor networks. In Section IV, we propose a traffic flow control protocol to exchange experiment commands and to collect experiment-related data from the network. In Section V, we present our system architecture for managing experiments. In Section VI, we demonstrate how we employed our system to conduct experiments in two different testbeds. Finally, in Section VII we give concluding remarks and outline future work.

II. Related Work

Testbeds are intended to efficiently test wireless sensor networks before actual deployments. Compared to the area or volume an actual deployment occupies, testbeds are considerably compact, so that they can be installed in labs or in areas which are easily accessible. This means, some communication parameters are intentionally scaled and events can be deliberately injected into the network to suit the test setting and to emulate actual events.

There are several testbeds, most of which are available for public use. Some of these are TWIST [11], WISEBED [6], MoteLab [20], and TempLab [4]. These testbeds share similar design principles. As far as hardware is concerned they provide additional wired or wireless interface (USB, Ethernet or Wi-Fi) as backbone channels for stable programming, controlling and data logging. As far as software is concerned, they provide (a) web-based interfaces to remotely access the testbeds and to manage experiments; and (b) mechanisms to automatically program, configure, and run the testbeds according to specific requirements. Rakotoarivelo et al. [15] meaningfully separate the software services into three logical services: control, management, and measurement. The experiment services of existing or proposed testbeds can be classified into two broad categories, inter-experiment and intra-experiment management services.

A. *Inter-experiment Management*

Almost all publicly available testbeds provide inter-experiment management services [10][12]. These testbeds provide web-interfaces to enable users to install their own program images on the testbeds and to specify experiment procedures remotely. Combined with different scheduling polices (for example, priority-based [20] or microeconomic processing [7]), physical resources can be reserved and experiments can be conducted automatically. During the experiment execution, the data collection service actively gathers in the background application-, protocol-, or network-specific data and store them in a local or remote database. After the execution of the experiments, users can download the data and perform off-line analysis.

B. *Intra-experiment Management*

WISEBED [6] is the first collective effort of nine European universities to build a heterogeneous wireless sensor network testbed. It provides a group of command line scripts to help users to manage their experiments [1]. Users can send arbitrary binary messages to individual nodes at runtime via a web interface or a script to interact with the experiment. This feature is useful and flexible both to retrieve and modify the state of execution, nevertheless, requires elaborate design and specification of experiment procedures (users are required to define and implement their own experiment control protocol).

In contrast, FlockLab [13] uses a hardware input/output mechanism (GPIO) to interrupt and control the experiment execution at a node level (which is more efficient than software-based interruption), however, it requires a dedicated hardware platform with an interface board. RadiaLE [3] is an application specific framework which aims to facilitate the design and implementation of link quality estimators (LQE) in wireless sensor networks. It consists of one control station (a PC) and 49 TelosB nodes which are connected via USB cables and hubs to form a radial topology. The control station has the ability to configure the network parameters and to initiate data transmission by sending command to specific nodes, according to the desired traffic pattern. The control method, however, is basic and application-dependent (can be used for studying LQE only).

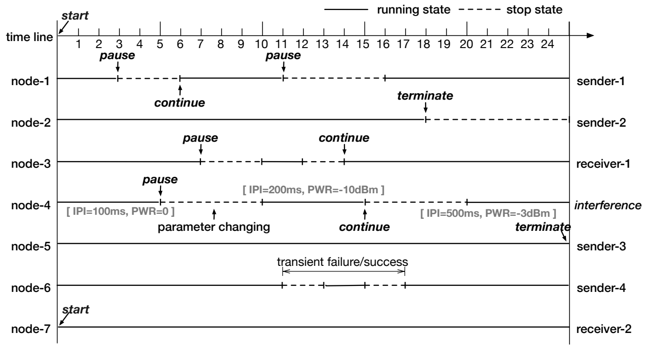
Minerva [16] is a distributed debugging testbed and provides python script interfaces to reset, halt and resume the execution of nodes which can be used for intra-experiment management. But, the testbed requires a special hardware support (a debugging board connected to a sensor node via JTAG interface) limiting its usefulness to carry out complex experiments in different testbeds.

In this paper we aim to augment the existing testbeds by separating experiment execution procedures (both intra- and inter-experiment activity sequences) from the application logic and by transferring them to the server side. This frees application developers from the burden of specifying and coding into their application logic detailed and inflexible procedures. Our approach enables the execution and control of experiments remotely. Should experiment execution logics be changed or modified at runtime, they can be done without interrupting the experiment or affecting the application logic or the need to reprogram nodes.

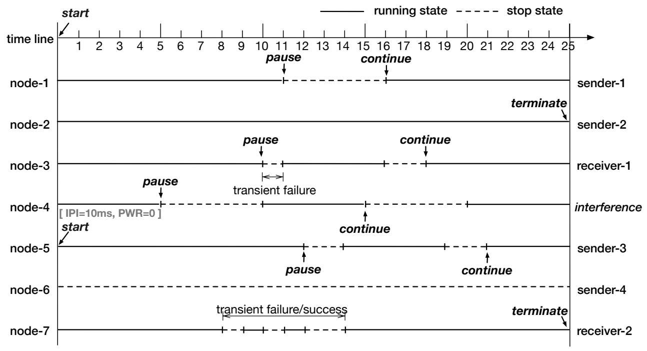
III. Challenges and Requirements

In order to illustrate the difficulty of conducting moderately complex experiments with existing testbeds, we provide an example using seven wireless sensor nodes (Figure 1). The experiment is intended to investigate the effect of random interference on the reliability of the network (measured in terms of overall packet loss). The experiment schedules specify the beginning and end of transmission times; inter-packet intervals (IPI) with which packets should be transmitted; communications types (unicast or broadcast); and a transient failure of nodes. The schedules also define the transmission patterns of nodes (when and for how long they should transmit).

In the first schedule, node 1 and 2 communicate with node 3; node 4 broadcasts to all nodes; and node 5 and 6



(a) schedule-1



(b) schedule-2

test case 1				
node	channel	tx-power (dBm)	IPI (ms)	receiver
node-1	26	0	20	node-3
node-2	26	0	50	node-3
node-3	26	0	100	-
node-4	26	0	100	broadcast
node-5	26	-10	100	node-7
node-6	26	-10	200	node-7
node-7	26	-10	250	-

(c) parameters for test case 1

test case 2				
node	channel	tx-power (dBm)	IPI (ms)	receiver
node-1	24	-10	20	node-3
node-2	24	-10	50	node-3
node-3	24	0	100	-
node-4	24	0	10	broadcast
node-5	24	-3	100	node-7
node-6	-	-	-	-
node-7	24	-3	250	-

(d) parameters for test case 2

Fig. 1: An example experiment: (a) and (b) Scheduling the transmission time and duration of nodes – The activity state of each node is represented by a solid line whereas the inactivity state is represented by a dashed line.. (c) and (d) Fixing protocol parameters for two test cases.

communicate with node 7. The Table in Figure 1 (c) displays the configuration of some of the physical-layer parameters for schedule-1. In the second schedule, node 6 is entirely terminated and transient failure is introduced to node 7 (which is a receiver). The Table in Figure 1 (d) displays the configuration of parameters for schedule-2.

The simplest way to perform the above experiments is to embed the experiment flow (schedules) into the application logic using timers to control the experiment at runtime. This, however, introduces some challenges. Firstly, a time synchronisation protocol has to be implemented at the application layer to synchronise the timers of all nodes. Otherwise the discrepancy in time drift in each node may lead to incongruity of schedule execution. Secondly, the integration of the time synchronisation protocol at the application layer violates the principle of separation of concern, because the application developer is concerned not only with the development of the application logic but also with network management. Thirdly, suppose our initial plan was to run schedule-1 only but after having observed the experiment results, we decided to modify the first schedule to produce the second and rerun the experiment. In this case, the application logic has to be modified in the source code, recompiled, and flashed to all the nodes. Reprogramming nodes not only is time consuming but also decreases the lifetime of the hardware (the number of erase/reprogram cycles is limited in most existing flash memories; for example, for the MSP430 MCU (used in the TelosB platform), the operation is limited to 10,000 times [18]). Additionally, the whole network needs to be reset manually for each round of the experiments resulting in unnecessary user intervention. In order to address these challenges, we propose a traffic flow control framework having the following features.

Integrability. Our framework is easily integrable with testbeds or application-dependent infrastructures. Researchers can use the framework to build and control their own experiment with little or no modification to their testbeds. Experiments can rerun multiple times to extract reproducible results without any manual involvement.

Scalability. As different infrastructures contain different number of nodes (from tens up to hundreds) [3] [11] [9], and different experiments require different combinations of nodes, our framework is both scalable and adaptive.

Reconfigurability. Most experiments are performed multiple times, not only under the same configuration but also with different parameter settings, to test the effect of different configurations on performance, network lifetime, and energy consumption. Some of the parameters that should be adjusted at runtime are (1) the duty cycle of MAC protocols [5] [8], (2) communication channels and transmission power levels to study link quality fluctuations [17] [19]; and (3) entries in routing tables. Thus, the framework should enable the configuration of these parameters without the need to reprogram the network. Additionally, in some experiments, event injections such as mimicking temporary node failures is useful to support.

Automatic execution. In existing testbeds, experiments automatically begin as soon as the nodes are active. Manual intervention is required to stop and restart experiments. In some cases advanced devices are used to remotely control experiments, but this makes experimentation unnecessarily expensive. Our framework employs software-controlled mechanism to manage experiments automatically.

Seamless data collection. One important and imperative process during experiment execution is extracting data and performance indicator metrics (such as RSSI, SNR, and timestamps) from the network. This process, however, should not interfere with the normal operation of the network (for example, by taking away precious bandwidth or communication time). Therefore, the framework should be able to seamlessly gather these data, temporarily store them locally, and enable the efficient collection at a most convenient time for on-line as well as off-line analysis and debugging.

IV. Traffic Flow Control Protocol

In order to cleanly separate experiment execution from experiment management, we propose a server-client architecture. The server specifies experiment schedules and dispatches them; individual nodes execute experiments and provide feedback. This simple approach relieves individual nodes from the burden of managing and executing experiments at the same time. Our approach requires a system architecture to orchestrate experiment procedures and a communication protocol to communicate commands, feedbacks, and experiment data. In this section we introduce the protocol and in the next the system architecture.

We propose a traffic flow control protocol (TFCP) in order to facilitate the remote management of inter- and intra-experiment executions. We define a set of control primitives which can be exchanged by the control protocol.

A. *Traffic Flow Control Primitives*

The traffic flow control primitives abstract a set of commands which control the execution of experiments in an application-independent manner. We classify our primitives into basic and enhanced primitives, according to their control granularity in an experiment execution. These primitives reside on top of any of the existing communication protocols and serve as agents for exchanging messages between experiment controller (server) and sensor nodes (client).

The basic primitives consist of *start* and *stop* commands, which are used to begin and stop an experiment. We keep them to two in order to limit the number of overhead messages that should be exchanged between the server and the nodes. The enhanced primitives, on the other hand, enable the execution of more complex experiments and provide fine-grained control. They consist of the following commands: *pause*, *continue*, *terminate*, *reset*, *clear*, and *read*. The combination of *pause* and *continue* can be used to suspend the execution of an experiment at an individual node for an arbitrary time, while

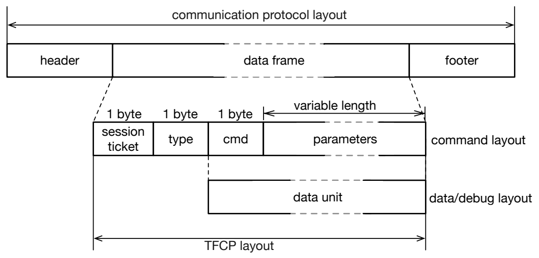


Fig. 2: TFCP encapsulated as the payload of an existing communication packet.

the *terminate* command can be used to break an experiment entirely. The last three primitives are useful for managing logged data and local resources.

The enhanced primitives can also be used for event injection into a network. Suppose we wish to test how a routing protocol copes with the dynamic behaviour of a network. The dynamic behaviour of individual nodes, such as when they leave and join a network or become temporarily unavailable causes a topology change which in turn affects the performance of the routing protocol significantly. It is not unusual to observe in real wireless networks arbitrary appearance and disappearance (failure) of nodes. The enhanced primitives provide adequate mechanisms to emulate and test these types of failures. Using *pause* and *continue* transient failures can be introduced whereas *terminate* can be used to emulate permanent failures. Compared with mechanically turning on and off nodes, which is presently the most frequently used approach to emulate node availability and to introduce transient failures, our approach is more efficient because it enables nodes to retain runtime states even when they are no longer available as active nodes.

B. *Protocol Design*

Our protocol can be encapsulated inside the payload of the backbone communication network (such as Ethernet and IEEE 802.11). This way it can easily be ported to or integrated with different backbone networks. This is shown in Figure 2. The TFCP layout is composed of a *session ticket*, *type*, and *command + parameters* or *data units* which are of variable length. The session ticket is a 1-byte-length random number which is used to identify and separate different test rounds. We categorise the messages exchanged between the experiment controller (server) and nodes into four groups, based on their QoS requirement and functionality. Table. I summarises how the primitives are categorised. The commands in the “setup”, “control” and “manage” groups require reliable communication (feedback is required to indicate failure or the successful execution of experiments). The commands in “control” group can be executed exactly once within an experiment, while the ones in the other two groups can be executed multiple times. Unreliable communication (no explicit feedback is required) is sufficient to extract data from individual nodes.

TABLE I: TFCP message types and commands.

type	QoS	command	parameters	initiator	description
setup	reliable, once or more	*init*	application dependent	server	setup the parameters of test round
control	reliable, exactly once	*start*	none	server	initiate the test round
		stop	none	node	notify finish of test round
		pause	none	server	suspend execution
		continue	none	server	resume execution
		terminate	none	server	stop execution permanently
manage	reliable, once or more	*clear*	none	server	erase data storage
		reset	none	server	reset the node
		read	block id	server	retrieve data from the local storage
data	unreliable	-	-	node	data report from nodes to server

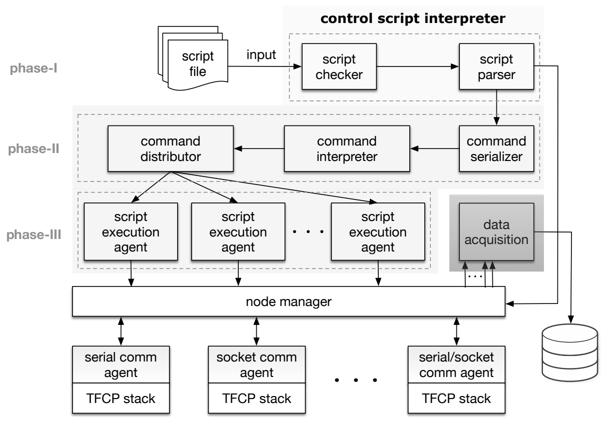


Fig. 3: The software architecture of the experiment controller.

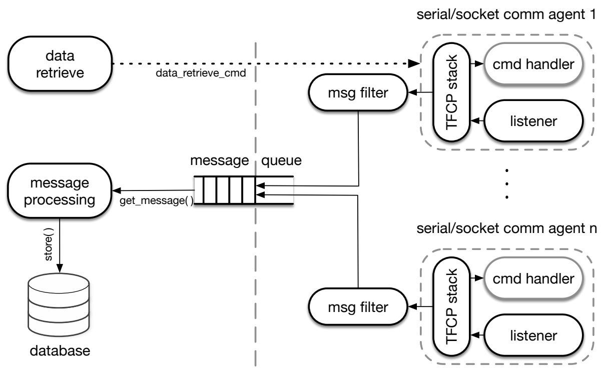


Fig. 4: The software architecture of the data acquisition module.

V. SYSTEM ARCHITECTURE

The system architecture for controlling advanced experiments consists of two major building blocks, the server-side block and the client-side block. The server-side block is useful for (1) defining experiment schedules and procedures, (2) dispatching the schedules, and (3) collecting relevant data from the network for analysis and debugging. The client-side block is a middleware that intercepts control packets from the network stack and control the execution of experiments locally.

TABLE II: keywords used in control script

keyword	function	description
init	section id	initial parameter section
flow-x	section id	flow definition section, x is flow id
node-x	node id	node identity, x is the address
flow	definition	define sequence of commands
repeat	counter	the execution number of flow
wait	timer	gap between two consecutive execution
start	command	start group of nodes immediately
stop	command	wait until receiving stop signal
pause	command	pause the execution of specific node(s)
continue	command	continue to run the specific node(s)
terminate	command	force the node(s) to stop

A. Server-Side Block

The system architecture of the server-side block is depicted in Figure 3. The control script interpreter and the data acquisition component are two separated entities. Each exchanges messages directly with the node manager. The node manager is a proxy for exchanging information between the server and sensor nodes and maintains a communication and a command execution agent for each active node, i.e., for each node which is involved in an experiment.

At the server-side, experiments are executed in three phases. First, users of the testbed specify their experiment scenario (procedure) using a list of key words we have defined (these are listed in Table II). An experiment procedure should specify initial parameters (such as the communication channels and transmission power levels of individual nodes), the sequence of actions, the number of times each test case should be repeated, and the intermission duration between experiments. These aspects are categorised into *init* (section ID) and *flow-x*. The *init* section contains the initial parameters of nodes involved in the experiment and the *flow-x* section contains the sequence of actions each node performs during the experiment. Experiment scripts are supplied to a script interpreter which prepares and dispatches procedures to the network. It consists of six sub-blocks and carries out dispatching in three phases.

Phase-I: This phase is responsible for script level validation and pre-preparation. When a control script is supplied to the control script interpreter, it first scans the entire file and checks whether the script is valid, based on predefined rules. For instance, the script must contain one *init* section and at least one *flow* section. If any of the rules are not fulfilled, the system raises an error flag and notifies the user and stops the execution. If the validation is successful, a script parser works on the control script to get participants' information (node address). The node manager requires this information to create and manage serial communication agents as well as command execution agents. In addition, the script parser ensures that each active node in the *flow* section is declared in the *init* section to make sure that all nodes are properly configured before performing the experiment; otherwise, unexpected results may occur or the experiment may not successfully proceed.

Phase-II: This phase is responsible for command level preparation and interpretation. The first operation of phase-II is serialising the commands in time sequence. The initialisation command (initial parameters of each node defined in the *init* section) is inserted prior to all other commands which are specified in the *flow* section. All the remaining commands and their parameters are kept in a chain. The command interpreter maintains a dictionary, which maps the literal commands into their executable proxy and translates the parameters into binary format. After the interpretation, each command contains three fields: the address of target node, executable command ID, and parameters. The key role of the command distributor is to distribute commands to specific script execution agents of the target nodes. The command distributor is an event-based component which is controlled by a timer, so that commands are dispatched in the time sequence they are specified in their scripts.

Phase-III: This phase is responsible for command execution. The last stage of the control script interpreter is to execute the commands, which is carried out by script execution agents (SEA). An SEA is a logical representation of a physical node at the side of the server and it is managed by a node manager. The creation of SEA relies on both the control script and the existence of corresponding nodes which are physically connected to the server. A configuration file is used to map the addresses specified in the control script to a physical node. An SEA receives commands from the dispatcher, caches the commands locally, and forwards them sequentially to a communication agent to be transmitted to individual nodes. It also implements a feedback mechanism to ensure that commands are executed successfully or, if they fail to execute, to inform the dispatcher about it.

The physical communication of experiment procedures is carried out by serial and/or socket communication agents (SCA), which are themselves managed by the node manager residing in the server block. The agents are created during the initialisation stage of command execution. Each agent has a command handler and listener. The command handler maintains a transmission channel for an active node. All the command messages are encapsulated in concrete TFCP packets and passed to the command handler via which they are transmitted to the specific node. The listener, on the other hand, regularly polls the receiver channel and maintains a local message buffer to store incoming TFCP packets. A valid message frame is detected, validated, and unpacked by the TFCP stack and messages are passed to the message filter for further processing.

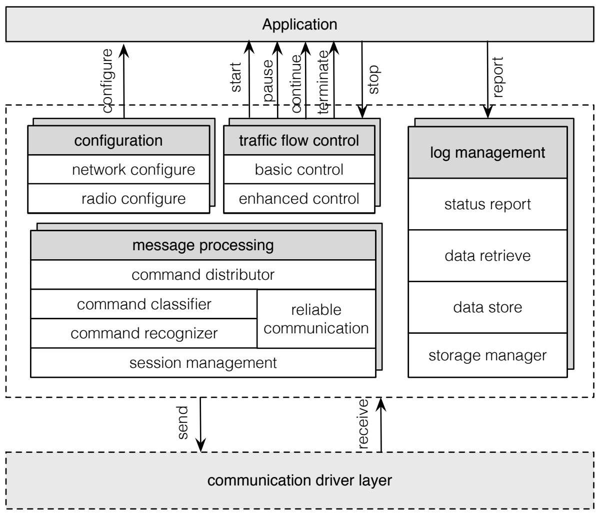


Fig. 5: The software architecture of the middleware component for TinyOS

In addition to the three command execution phases, the server-side block also implements a data acquisition component as displayed in Figure 4. Our system supports both push and pull modes. In the push mode, data are reported to the server automatically while experiments are being executed; in the pull mode, data are stored locally and retrieved by the server using data retrieve command after the experiment. In both modes, the data messages are received by the SCA and filtered by the message filter component. The message filter is a component that filters specific type of messages. For instance, in our implementation, only the data message can pass the filter and then pushed to the unified message queue. After processing, the data messages are stored in a database. In the present implementation, the message processing function is a stub which only stores the message in raw format into the database. Users can define their own procedure to process messages before storing.

B. Client-Side Block

The client-side block of our system receives commands, schedule and execute them locally, and provides feedback to the server-side block. Figure 5 illustrates the architecture of the client-side block for TinyOS environment using a USB interface for a backbone network. The TFCP layer sits on top of the driver layer (for our case, but it can also be implemented on top of any communication stack), where it can utilize either serial or network connection for data and/or command transfer between nodes and the server. The TFCP layer exposes 5 interfaces to application layer for control and status report purpose. These are (a) *message processing*, (b) *configuration*, (c) *traffic flow control*, (d) *log manager*, and (e) *status report* (not shown in the figure).

The message processing component is responsible for managing sessions; recognise, classify, and distribute commands, and provide reliable communication. Since hundred percent communication cannot be guaranteed in any communication link (not even for a serial communication), some of the sensor nodes may not have been properly configured before starting an experiment (for example, the transmission channel may not have been set). This may hinder an experiment from being executed successfully. To deal with this issue, we introduce session keys. A session key is updated only during a configuration phase and each command in the same session will hold the same key for consistency. Nodes which do not have the appropriate session key will not participate in the current experiment. The command recogniser, classifier, and distributor make up a command processing chain. The reliable communication component is responsible for providing feedback or to raise an error flag to the server after a command is locally executed.

The configuration component is responsible for setting up the experiment parameters. After proper validation, the parameters are distributed to two interfaces (radio and network). The radio configuration interface is used to set the radio-related parameters, such as communication channels and transmission power levels and the network configuration interface is used to initialise application-specific parameters, such as destination and next-hop addresses and inter-packet-interval (IPI).

The log management component provides four main functionalities: storage management, data storage, data retrieval, and status report. We divided the local flash into several blocks (the number of blocks and block size can be configured at compilation time). At booting, the storage manager checks the block list and finds available free blocks to use. If there is no available block, the system will halt and generate an error message to notify the user. The binary image of each log item is stored in blocks sequentially. For each test run, only one block is used. If the block is full, all the remaining items are withdrawn, for erasing the flash at runtime costs time (it takes more than a few seconds to erase the whole flash). During data retrieval, the whole block is read at one time by specifying the block ID to simplify the design and implementation complexity. Collecting performance indicator metrics in real time is one of the main tasks of TFCP. All the information is reported to the server as a *push* process. Unlike *printf()*, which can only send string messages, the status report function provides a common interface to transfer any data of any format as a binary stream. The entire TFCP stack implementation in a TinyOS environment for the TelosB platform has a footprint of 1058 Bytes of ROM and 84 bytes of RAM.

VI. Experiments Using TFCP

We used our system to conduct experiments in two different testbeds. In the first testbed, we investigated link quality fluctuation and the performance of two burst-transmission strategies which deal with link quality fluctuations. The deployment for this experiment setting took place in an outdoor environment and the network consisted of 14 TelosB nodes. We set up a backbone network using USB cables and active USB hubs for programming the nodes and managing experiments executions.

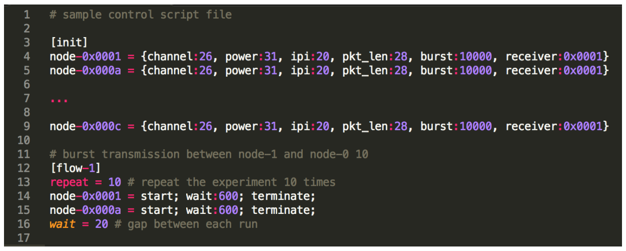
TABLE III: Parameters used for burst transmission experiments in outdoor environment.

parameter	value
environment	outdoor
distance between nodes	10 - 30 m
information to collect	timestamp, RSSI, noise, LQI, seqno.
channel	26
tx-power	0 dBm, -10 dBm
packet length	28 bytes
IPI	20 - 100 ms
burst size	500 - 10000
total number of packets	10,000

In the second testbed, we evaluated the performance of TFCP in a lab environment with variable number of nodes (physical and virtual nodes). We used hybrid communication channels (serial and Ethernet) as backbone networks to manage the experiments.

A. Link Quality Fluctuation

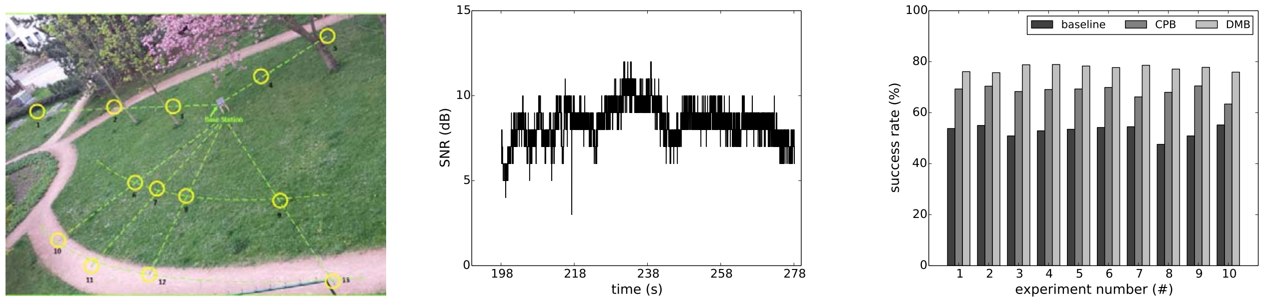
The quality of links in any network and particularly in wireless sensor networks considerably affects the reliability (expected packet loss) and lifetime of the networks (due to retransmission of lost packets). The effects of poor reliability can be observed in the quality of data that can be extracted from them (for example, in terms of expected end-to-end latency and jitter). Independent studies [2], [14] have shown that link quality fluctuation is frequently observed in wireless sensor networks even in static deployments. Most existing strategies dealing with link quality fluctuation rely on sufficient statistics collected from the networks to estimate stable and bursty durations. The statistics collection process requires repeated and complex experiments to insure that the statistics are representative.



```
# sample control script file

[init]
node-0x0001 = {channel:26, power:31, ipi:20, pkt_len:28, burst:10000, receiver:0x0001}
node-0x000a = {channel:26, power:31, ipi:20, pkt_len:28, burst:10000, receiver:0x0001}

...

node-0x000c = {channel:26, power:31, ipi:20, pkt_len:28, burst:10000, receiver:0x0001}

# burst transmission between node-1 and node-0 10
[flow-1]
repeat = 10 # repeat the experiment 10 times
node-0x0001 = start; wait:600; terminate;
node-0x000a = start; wait:600; terminate;
wait = 20 # gap between each run

```

Fig. 6: A snapshot of the control script for investigating link quality fluctuation.

For our testbed we used two burst transmission strategies that rely on the statistics of link quality fluctuations. Both strategies first transmit packets continuously in burst and gather link quality metrics from acknowledgement packets (RSSI, LQI, background noise, timestamp and sequence of received acknowledgement packets). These metrics are then used to establish statistics pertaining to link quality fluctuations and to compute expected stable durations (both good and bad). The expected stable durations are taken into account to determine the number of packets that can be transmitted in burst. The first strategy defines stable durations as random



(a) (b) (c)

Fig. 7: Experiment results for link quality fluctuation. (a) The topology of the textbed. (b) The snapshot of the SNR of acknowledgement packets gathered in real time. (c) The packet delivery rate of the two strategies during different experiment runs for a specific link.

variables and establishes the cumulative distribution function of stable durations for each link (we label this strategy as “CPB”) whereas the second strategy employs a two-stage Markov process to model link quality fluctuations as states (we label this strategy as "DMB"). This strategy also relies on statistics to determine the expected duration of states and state transitions.

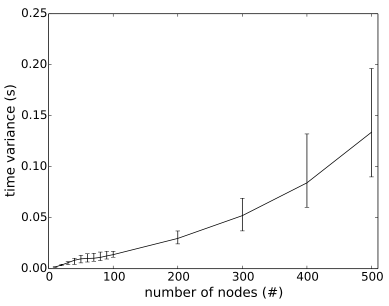
To collect sufficient statistics for both strategies, first a communication pair is identified and the physical parameters for communication are fixed (channel, transmission power level, and packet size). Then the number of packets which should be transmitted in burst and the inter-packet interval are determined. Then each experiment is repeated 10 times and the link quality indicators are locally stored. While the experiments are being conducted, the link quality indicators are gathered from the individual nodes using TFCP. Tables III displays the specifications and parameters we defined for the experiments. Figure 7 (a) displays the topology of one of the testbeds. Figure 6 displays the snapshot of our control script in which two links ([node 1, node 10] and [node 1, node 12]) are identified for the experiments. In the first link 10,000 packets are transmitted in burst with an inter-packet-interval of 20 ms. There is a 20 s intermission between experiment runs. Likewise, in the second experiment 5000 packets are transmitted in burst between node 1 and node 12 with an inter-packet-interval of 100 ms. The intermission is the same as the first experiment. Figure 7 (b) and (c) display a snapshot of the SNR fluctuation of a specific link as observed in real time, and a comparison of the packet delivery rate of the two strategies and a baseline in which no strategy was used to transmit packets.

B. Performance Evaluation

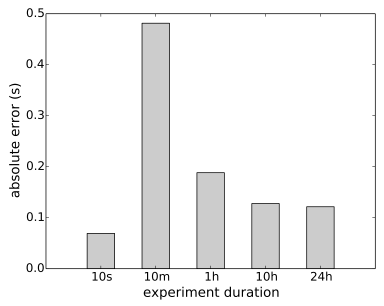
We evaluated the performance of TFCP in two ways: (1) The deviation of start time during the simultaneous execution of an experiment as the network size increases; and (2) the deviation of an experiment completion time as the duration of experiment increases. In the first case, we conducted a set of small experiments with variable number of physical nodes (from 5 to 20) as well as virtual nodes (up to 500). In each of the experiments all nodes should begin the experiment simultaneously. The experiments consist of nodes periodically transmitting a fixed amount of packets; each experiment is repeated 100 times. Fig. 8 (a) shows the average deviation of the starting time as a function of network size. As can be seen, even for a large network consisting of 500 nodes, the testbed guarantees a deviation of experiment start time which is below 200 ms. Likewise, Fig. 8 (b) displays the deviation of intended time of completion for different experiment durations. In this experiment we set up a testbed of 10 TelosB nodes which were connected to the controlling server via USB cables and hubs. We performed a series of experiments the duration of which varied from 10 seconds to 24 hours. The maximum deviation between the actual experiment duration and the intended duration was in the order of hundred milliseconds only. Lastly, we inserted arbitrary number of control commands (*pause* and *continue* commands) in the experiments lasting up to 600 seconds, and varied the number of nodes from 1 to 10. We did not observe significant increments of experiment completion times when the number of commands increased (shown in Fig. 8 (c))

VII. CONCLUSION

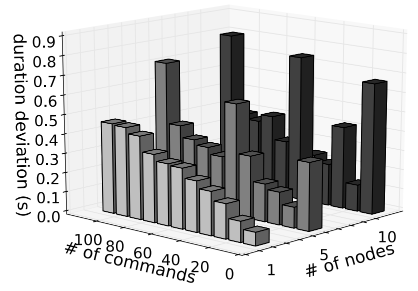
In this paper we proposed a system architecture for managing complex experiments in wireless sensor networks. The system architecture consists of a server-side block for specifying, managing, and dispatching inter- and intra-experiment activities and a client-side block for receiving and executing commands and for collecting experiment-related feedback and data. We also defined light-weight traffic flow controlling primitives and a communication protocol for exchanging commands and feedbacks. The system-side block is implemented in python whereas the TCFP stack at the client-side was implemented for TinyOS environment has a footprint of 1058 bytes of ROM and 84 bytes of RAM in a TelosB platform. Our approach does not require special hardware or backbone network infrastructure.



(a)



(b)



(c)

Fig. 8: Performance analysis of TFCP: (a) Deviation in experiment starting time for networks of different sizes. Ideally, all nodes should begin executing an experiment at the same time regardless of the network size; (b) Time deviations between intended and actual experiment completion time for different experiment durations; (c) The deviation in the duration of arbitrary control commands in a single experiment.

We tested our system with our local testbed having small network sizes. We investigated the performance of two burst transmission techniques dealing with link quality fluctuations. The network consisted of 14 nodes and pairs of nodes communicated with each other while we gathered useful metrics from the network to establish statistics pertaining to link quality fluctuations. We used these statistics to estimate expected stable durations. The platform simplified the execution of all experiments and provided us with great latitude to define and redefine experiment settings. In future we are aiming to extend our system architecture to include additional features such as mobility management and to interface our system with Matlab and the R-statistical platform, so that statistical data can be directly transferred from our system to these tools for statistical analysis. Work is also in progress to make the source code and the testbeds available online for the research community.

References

[1] experimentation scripts 0.8. https://github.com/wisebed/experimentation-scripts/wiki/Experimentation-Scripts-0.8, July 2012.

[2] M. H. Alizai, O. Landsiedel, J. Á. B. Link, S. Götz, and K. Wehrle. Bursty traffic over bursty links. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 71–84. ACM, 2009.

[3] N. Baccour, A. Koubaa, M. B. Jamâa, D. Do Rosario, H. Youssef, M. Alves, and L. B. Becker. Radiale: A framework for designing and assessing link quality estimators in wireless sensor networks. *Ad Hoc Networks*, 9(7):1165–1185, 2011.

[4] C. A. Boano, M. Zúñiga, J. Brown, U. Roedig, C. Keppitiyagama, and K. Römer. Templab: A testbed infrastructure to study the impact of temperature on wireless sensor networks. In *Proceedings of the 13th international symposium on Information processing in sensor networks*, pages 95–106. IEEE Press, 2014.

[5] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 307–320. ACM, 2006.

[6] I. Chatzigiannakis, S. Fischer, C. Koninis, G. Mylonas, and D. Pfisterer. Wisebed: an open large-scale wireless sensor network testbed. In *Sensor Applications, Experimentation, and Logistics*, pages 68–87. Springer, 2010.

[7] B. N. Chun, P. Buonadonna, A. AuYoung, C. Ng, D. C. Parkes, J. Shneidman, A. C. Snoeren, and A. Vahdat. Mirage: A microeconomic resource allocation system for sensornet testbeds. Institute of Electrical and Electronics Engineers, 2005.

[8] W. Dargie and J. Wen. A seamless handover for wsn using lms filter. In *Local Computer Networks (LCN), 2014 IEEE 39th Conference on*, pages 442–445. IEEE, 2014.

[9] C. B. des Roziers, G. Chelius, T. Ducrocq, E. Fleury, A. Fraboulet, A. Gallais, N. Mitton, T. Noël, and J. Vandaele. Using senslab as a first class scientific tool for large scale wireless sensor network experiments. In *NETWORKING 2011*, pages 147–159. Springer, 2011.

[10] E. Ertin, A. Arora, R. Ramnath, V. Naik, S. Bapat, V. Kulathumani, M. Sridharan, H. Zhang, H. Cao, and M. Nesterenko. Kansei: a testbed for sensing at scale. In *Proceedings of the 5th international conference on Information processing in sensor networks*, pages 399–406. ACM, 2006.

[11] V. Handziski, A. Köpke, A. Willig, and A. Wolisz. Twist: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *Proceedings of the 2nd international workshop on Multi-hop ad hoc networks: from theory to reality*, pages 63–70. ACM, 2006.

[12] X. Ju, H. Zhang, and D. Sakamuri. Neteye: a user-centered wireless sensor network testbed for high-fidelity, robust experimentation. *International Journal of Communication Systems*, 25(9):1213–1229, 2012.

[13] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel. Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *Information Processing in Sensor Networks (IPSN), 2013 ACM/IEEE International Conference on*, pages 153–165. IEEE, 2013.

[14] S. Munir, S. Lin, E. Hoque, S. M. S. Nirjon, J. A. Stankovic, and K. Whitehouse. Addressing burstiness for reliable communication and latency bound generation in wireless sensor networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 303–314, New York, NY, USA, 2010. ACM.

[15] T. Rakotoarivelo, M. Ott, G. Jourjon, and I. Seskar. Omf: a control and management framework for networking testbeds. *ACM SIGOPS Operating Systems Review*, 43(4):54–59, 2010.

[16] P. Sommer and B. Kusy. Minerva: Distributed tracing and debugging in wireless sensor networks. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, page 12. ACM, 2013.

[17] K. Srinivasan, P. Dutta, A. Tavakoli, and P. Levis. An empirical study of low-power wireless. *ACM Transactions on Sensor Networks (TOSN)*, 6(2):16, 2010.

[18] Texas Instruments. *MSP430 Flash Memory Characteristics*, September 2006. Revised April 2008.

[19] J. Wen, Z. Ansar, and W. Dargie. A link quality estimation model for energy-efficient wireless sensor networks. In *Communications (ICC), 2015 IEEE International Conference on*. IEEE, 2015.

[20] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, page 68. IEEE Press, 2005.