# TinySDM: Software Defined Measurement in Wireless Sensor Networks

Chenhong Cao, Luyao Luo, Yi Gao, Wei Dong*, and Chun Chen
Zhejiang Provincial Key Laboratory of Service Robot,
College of Computer Science, Zhejiang University, China
Emails: {caoch, luoly, gaoy, dongw}@emnets.org, chenc@zju.edu.cn

***Abstract*—Network measurement, which provides detailed information about the behaviors of operational networks, is essential for network management in wireless sensor networks. In the literature, there have been many approaches focusing on measuring *individual* aspect of the network, e.g., per-packet routing path and per-hop delay. However, there lacks a general support for conducting different measurement tasks. When managing an operational network, a network operator often needs to switch the current measurement task to a different one, in order to diagnose the observed symptoms. In this paper, we propose TinySDM, a *software-defined measurement architecture* for WSNs. TinySDM provides a general support for conducting different measurement tasks. TinySDM defines a set of carefully selected hooks that allow the users to easily execute their own measurement tasks. In addition, TinySDM provides a C-like language called TinyCode Language (TCL) to enable easy customization of measurement tasks. By only transmitting the binary code of the measurement task, TinySDM significantly reduces the size of the disseminated data compared with existing reprogramming approaches. We implement TinySDM on the TinyOS/TelosB platform and evaluate its performance extensively in a testbed with 60 nodes. We also use TCL to implement four specific measurement tasks. Results show that TinySDM is flexible, efficient and easily programmable.**

## I. Introduction

Wireless sensor networks (WSNs) have been widely employed for enabling various applications such as environment monitoring [1], ecosystem management [2], and infrastructure protection [3]. Managing WSNs becomes increasingly difficult due to the rapid growth of their network scale and system complexity. Network measurement plays an important role in the management of WSNs. Many network tasks such as topology control, routing improvement, load balance and performance diagnosis [4–7] rely on accurate and timely measurements of various system metrics, e.g., per-packet routing path, per-link loss ratio and per-hop delay. For example, PAD [5] relies on the dynamic routing topology for online diagnosis of an operational sensor network.

Previous efforts on WSN measurement [8–11] focused on each *individual* aspect of the network, e.g., the routing path [8], packet loss ratio [11], etc. However, there lacks a *general* support for conducting *different* measurement tasks.

Consider a large scale sensor network consisting of thousands of sensor nodes. Usually, some *lightweight* measurement tasks, e.g., PAD [5], are already included in the application code in order to facilitate the management of such a network. However, there may exist *unexpected* symptoms, e.g., large transmission delays [9], which cannot be well explained by these lightweight measurement tasks. As a network manager, you face the practical requirement of *easily customizing* a more suitable measurement task, e.g., delay monitoring [9], and *quickly deploying* this new measurement task.

Performing this routine task is very difficult. First, it is difficult to customize a new measurement task. A programmer often needs to find the right places from a large codebase before writing customized measurement functions. This requires a deep understanding of the underlying implementation details. For example, in order to conduct delay measurement using Domo [9], a programmer needs to perform source-level modifications in the MAC layer, network layer and application layer: In the MAC layer, packet timestamps should be recorded; In the network layer, the transmission delay should be accumulated and updated before it is piggybacked in the forwarded data packet; In the application layer, the packet structure should be modified to accommodate additional measurement data.

Second, it is costly to deploy a new measurement task. While wireless reprogramming approaches [12, 13] offer general approaches for deploying WSNs software, they often introduce unnecessarily large overhead for deploying a measurement task. For example, wireless reprogramming using the standard approach in TinyOS, i.e., Deluge [12], requires transferring the entire program image, introducing a large dissemination overhead. Recent incremental reprogramming approaches significantly reduce the dissemination cost [13]. However, they usually require a hardware reboot, incurring a large network initialization overhead for recovering the network status.

To address these challenges, we propose TinySDM, a *software-defined measurement architecture* for WSNs. TinyS-

*We sincerely thank our shepherd Prof. Jack Stankovic and the IPSN reviewers for their valuable comments and feedback. This work is supported by the National Science Foundation of China (No. 61472360 and No. 61502417), Zhejiang Commonwealth Project (2015C33077), Zhejiang Provincial Natural Science Foundation of China (No. LY16F020006), Fundamental Research Funds for the Central Universities, China Ministry of Education—China Mobile Joint Project (No. MCM20150401), and Zhejiang Provincial Platform of IoT Technology (No. 2013E60005). Wei Dong is the corresponding author.

DM provides a *general* support for conducting *different* measurement task.

**First, it allows easy customization of different measurement tasks.** TinySDM defines a set of carefully selected hooks that allow users to easily execute their own measurement tasks. In addition, TinySDM allows users to easily add global variables for the measurement task. To facilitate programming, TinySDM provides a C-like language called TinyCode Language (TCL) which is *expressive* enough for implementing a variety of measurement tasks. TCL provides a good *separation* of the measurement logic and remaining code (including the OS, protocol, and application) so that users can focus on the measurement task at hand.

**Second, it allows fast and efficient deployment of a new measurement task.** The measurement tasks expressed in TCL are compiled into separate binary modules which can be dynamically loaded on the sensor nodes. To deploy a new measurement task, it only needs to disseminate the binary code relevant to the measurement task. Compared with traditional reprogramming approaches, the dissemination of a measurement task in TinySDM requires much less transmission overhead, which is important for energy-constrained WSNs [14–16]. Moreover, TinySDM does not require a hardware reboot to execute or stop a new measurement task, incurring no initialization overhead for collecting network status. In addition, high level interfaces for adding or deleting a measurement task at runtime are also provided in TinySDM.

We implement TinySDM on the TinyOS 2.1.2/TelosB platform. We use TinySDM to implement four existing measurement tasks to demonstrate that its expressiveness allows easy customization of different measurement tasks. We also carefully evaluate its implementation overhead. Results show that TinySDM introduces small CPU and memory overhead. More importantly, TinySDM significantly reduces the size of the disseminated data for deploying new measurement tasks compared with using reprogramming protocols to update measurement tasks.

The contributions of this paper are summarized as follows:

- **Software-defined measurement architecture for WSNs.** To the best of our knowledge, we are the first to propose a software-defined measurement architecture which provides a general support for conducting different measurement tasks.
- **Expressive language design.** We design a C-like language called TCL. Four case studies demonstrate its expressiveness for easily customizing different measurement tasks.
- **Efficient implementation.** We describe an implementation of TinySDM on the TinyOS 2.1.2/TelosB platform. We show that TinySDM allows fast and efficient deployment of new measurement tasks.

The rest of this paper is structured as follows. Section II introduces the related work. Section III gives an overview of TinySDM. Section IV presents the base program generation in details. Section V describes the TinyCode Language in details, illustrating its core features using examples. Section VI shows the expressiveness of TCL using existing measurement tasks. Section VII describes the implementation of TinySDM. Section VIII evaluates the performance using real testbed experiments. Section IX presents a discussion on TinySDM. Finally, Section X concludes this paper and our future work.

## II. Related Work

The related works of TinySDM can be divided into the following categories: measurement in sensor networks, dynamic instrumentation framework in sensor networks, software defined architecture in sensor networks, and software defined measurement in the Internet.

**Measurement in sensor networks.** In order to obtain the internal behaviors of sensor networks, many measurement approaches [5, 6, 8–11, 17–19] focusing on various aspects have been proposed in the past years. In this paper, we use TinySDM to implement four typical measurement tasks, including, an end-to-end delay measurement approach E2EDM [10], a path tracking approach PAD [5], a per-hop delay measurement approach Domo [9], and a measurement-based diagnosis approach Sympathy [6].

Traditional delay measurement relies on network synchronization which may not be available in large scale sensor networks. To tackle this problem, Wang et al. proposed an end-to-end delay measurement method [10] based on the MAC layer timestamping [20]. Through MAC layer timestamping, the sojourn time of a packet at each node can be captured accurately. By accumulating the packet sojourn times of each node along its packet routing path, the end-to-end delay can be obtained. Domo [9] is a more fine-grained delay measurement method which achieves lightweight and accurate per-hop per-packet delay measurement.

Besides the delay measurement approaches, there are also approaches aiming at measuring packet routing path in sensor networks. PAD [5] employs a packet marking scheme to record a static routing path. Each passing packet carries one hop information of the static path and the whole path can be recovered after collecting multiple such packets.

Sympathy [6] is a typical debugging tool that actively collects runtime network status from each node and analyzes them at the sink. It collects information such as routing table, neighbor list, traffic flow, and etc. The insight is that the failure can be detected and localized by carefully selecting an optimal set of metrics.

Different from these approaches, TinySDM is not a measurement approach for a specific network metric, but is a software defined measurement architecture which supports fast and efficient task deployment, as well as easy customization of different measurement tasks.

**Dynamic instrumentation framework in sensor networks.** Quanto [21] instruments OS system calls for tracing power consumption of logical activities. Though Quanto measures energy usage in a fine-grained manner, it fails to support the measurement tasks in a user-defined manner. In addition, TinySDM provides more flexibility for measurement tasks, which allows users to define the hook positions. Declarative Tracepoints (DT) [22], which is a declarative framework for debugging sensor networks, introduces TraceSQL to program debugging actions as tracepoints and allows addition and deletion of tracepoints at runtime. TinySDM has similar goals to DT such as achieving application independence and easy programming. However, TinySDM and DT are different in the following ways. First, unlike DT, TinySDM mainly focuses on sensor network measurement. TinySDM provides a good separation of the measurement logic and the remaining code (including the OS, protocol, and application) by defining a set of carefully selected hooks. Therefore, users can focus on the measurement task at hand, without having a deep understanding of the underlying implementation details. Second, TinySDM supports measurement tasks which involve multiple sensor nodes, while DT focuses on node debugging. For example, in order to measure end-to-end delay of a packet, all nodes in the routing path of that packet should update the packet to accumulate the packet sojourn times at these nodes [10].

**Software defined architecture in sensor networks.** Existing proposals about software defined architecture in WSNs aim at controlling the routing behavior. Sensor OpenFlow [23] extends the OpenFlow approach — the most popular instance of SDN, and offers a more flexible specification of the rules to configure the routing policy. Unlike Sensor OpenFlow, SDN-WISE [24] is a stateful SDN solution for WSNs. Different from these approaches which focus on routing, TinySDM focuses on providing a flexible measurement architecture for sensor networks.

**Software defined measurement in the Internet.** OpenSketch [25] separates the measurement data plane from the control plane. In the data plane, OpenSketch provides a simple three-stage pipeline (hashing, filtering, and counting), which can be implemented with commodity switch components and supports many measurement tasks. However, OpenSketch focuses on estimating aggregate statistics (flow level) while TinySDM is specifically designed for WSN measurements which focus on both node level aggregate statistics and packet level network behaviors.

TPP [26] is a simple, programmable interface that enables endhosts to attach instructions into the packets to query the switch memory directly in the data plane. Specifically, these instructions are carried in the header of a subset of packets. The operations include read, write, or perform simple, protocol-agnostic computation using switch memory. In TPP, both the instructions and the execution results are encapsulated in the header of a packet. In TinySDM, only measurement results are encapsulated in a packet. The code for a measurement task is disseminated to each node in a separate phase. Hence, TinySDM is able to perform more complicated tasks than TPP.

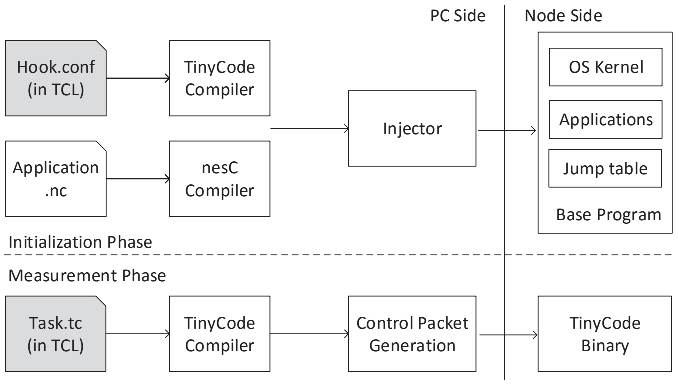


**Fig. 1: Overview of TinySDM.**

## III. Overview

The TinySDM system aims to provide a general support for conducting different measurement tasks. Figure 1 shows its overall architecture. The procedure of TinySDM is separated into two phases: initialization phase and measurement phase.

In the initialization phase, a modified version of the program image is generated and programmed into all nodes in the network. The modifications include the injection of a number of hooks specified by a configuration file `Hook.conf` written in TCL. In this paper, we refer to this modified version of program image as the *base program*. Details about the base program generation are given in Section IV.

In the measurement phase, a network operator can write a customized measurement task in TCL and use the TinyCode compiler to generate the measurement task binary. Then the measurement task binary is embedded to a number of control packets and sent to all nodes in the network. Details about the TCL and the control packet generation are given in Section V.

## IV. Base Program Generation

In the initialization phase, the base program is generated and programmed into all nodes in the network. Figure 2 illustrates the generation of the base program. The application and kernel components are written in nesC (the language used in TinyOS) and are compiled into C code (.c) by the nesC compiler. The hook positions are specified in a configuration file (.conf). The preprocessor then analyzes the configuration file to generate a system jump table in assembly (.s) and inserts hooks into the specified positions. The GCC toolchain further compiles these files (.c and .s) using the compiler (cc) and assembler (as). Finally, an executable ELF file is generated through a two-phase linking process.

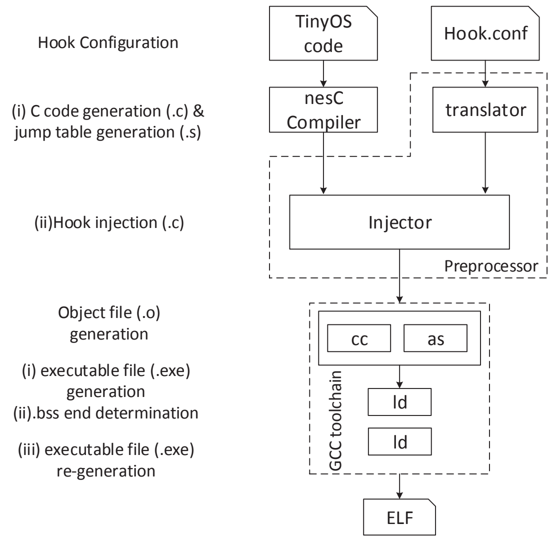

Hook Configuration
TinyOS code
Hook.conf
(i) C code generation (.c) & jump table generation (.s)
nesC Compiler
translator
(ii)Hook injection (.c)
Injector
Preprocessor
Object file (.o) generation
cc
as
(i) executable file (.exe) generation
(ii).bss end determination
ld
(iii) executable file (.exe) re-generation
ld
GCC toolchain
ELF

**Fig. 2: The generation of base program.**

## A. Hook Configuration File

The configuration file consists of two types of statements: hook statements and reserved packet memory statements. The *hook statements* are used to specify the possible positions where the users want to perform the measurement tasks. The *reserved packet memory statements* are used to reserve memory in the data packets for collecting the measurement data. Specifically, a hook is located by a pair consisting of a function name and a file name. By default, the hook locates at the beginning of the function. Additional keywords, **START** and **END**, can be used to specify the exact address of where to perform the measurement task. The default name of a hook is unique and follows the rule: `HOOK_$FileName_$FunctionName`. `$FileName` (resp. `$FunctionName`) represents the value of the variable `FileName` (resp. `FunctionName`). For special symbol in the function name (e.g., '.' in the function name under TinyOS), it will be replaced by an underscore (i.e., symbol '_').

With the keyword **AS**, the hook can be given an alias. To reserve packet memory, the packet type and the reserved size should be specified with the keyword **RESERVE**. As an example, the following code shows how to configure a hook that performs the measurement task every time the packet is forwarded and how to reserve 20 bytes in the packet type `TestSDMMsg` to record the measurement data. In this example, the default name of the hook is `HOOK_CtpForwardingEngineP_SubReceive_receive`. This hook is positioned at the beginning of function `SubReceive.receive()` from the file `CtpForwardingEngineP.nc`. Starting with keyword **HOOKPOS**, the following example shows how to set the hook configuration and the reserved space size in a specific packet type.

```
HOOKPOS SubReceive.receive() START
FROM CtpForwardingEngineP.nc
AS HookCtpReceive;
RESERVE TestSDMMsg packetsize=20;
```

We develop a tool, `gbc`, to integrate the operations of preprocessor and GCC toolchain in the initialization phase.

## B. Hook Positions

We reserve default hook positions in the configuration file (`Hook.conf`). The user can also customize other hook positions according to measurement requirements. In order to make the reserved hook positions be expressive enough to support most measurement tasks, we carefully analyze typical measurement tasks in sensor networks. Existing measurement tasks can be divided into the following two categories.

**(1) Tasks that query runtime network status:** These measurement tasks actively query runtime network status on the node side. For example, Sympathy [6] measures various network metrics including connectivity metrics, flow metrics and node metrics. We reserve the hook positions where we can compute and retrieve these metrics.

**(2) Tasks that require per-packet computation:** WSNs are usually multi-hop networks. In order to obtain packet level information, many existing measurement tasks in sensor networks need to modify the packet at multiple forwarding nodes, e.g., end-to-end delay measurement and routing path measurement. This information can be obtained at the forwarding time. Thus, we also reserve hooks in the routing/forwarding services provided by the node operating system.

## V. TCL Programs

In the measurement phase, a network operator can write a customized measurement task in TCL and use the TinyCode compiler to compile it. Then the measurement task binary is embedded to a number of control packets and sent to all nodes in the network. In this section, we will first describe the whole measurement phase briefly, and then focus on describing the TCL program in details.

## A. Measurement Phase Overview

In the measurement phase, user customizes the measurement tasks and deploys them through *control packets*. Figure 3 shows the generation of the control packet. The programs of measurement tasks (.tc) are written in TCL. The TinyCode compiler translates the user programs from TCL into C with equivalent semantics. The C file is further processed by the GCC compiler to generate the object file (.o). Using the tool (relocate) we developed, the binary file (.ihex) of the measurement task is generated. We develop a tool SendPkt (.java) to generate the control packets and disseminate them

**Table 1: Overview of TinyCode Language Keywords**

| TinyCode Language Overview | | | |
|---|---|---|---|
| **File types** | **Statement types** | **Keywords** | **Description** |
| *Configuration file* | *Configuration statements* | HOOKPOS, START, END, RESERVE | Configure the hook positions and reserve packet memory. |
| *Measurement task file* | *Paket declarations statements* | ALLOC | Allocate space to additional fields from the packet space reserved. |
| | | AS | Assign an alias. |
| | *Variable declarations statements* | INTEGER_8,INTEGER_16,INTEGER_32, STRUCT,EXTERN | Declare TCL variables. |
| | *HOOK declarations statements* | HOOK | The basic format is: `HOOK HookName{}`. |
| | | IF, ELSE, ELSE IF | General condition statements. |
| | | EXIT | Exit the current executing task. |
| | | TERMINATE | Terminate the specified task. |
| | *Built-in functions* | GetTimeNow() | Obtain the current execution time. |
| | | Find(arrayName, fieldName, value) | Search for the item in arrayName whose fieldName equals to value. |

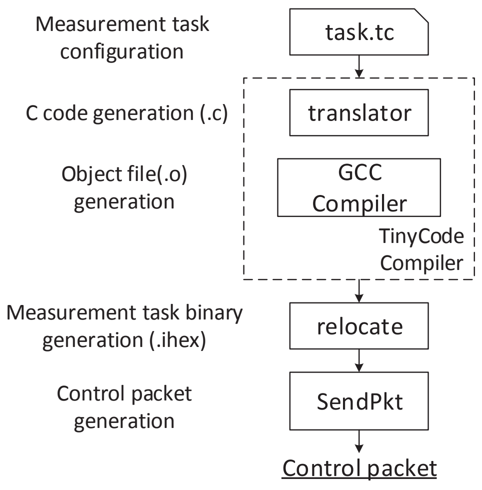

Measurement task configuration
task.tc
C code generation (.c)
translator
Object file(.o) generation
GCC Compiler
TinyCode Compiler
Measurement task binary generation (.ihex)
relocate
Control packet generation
SendPkt
Control packet

**Fig. 3: The generation of the control packet.**

```
ALLOC TestSDMMsg{
 INTEGER_32 delaysum;
 INTEGER_32 timestamp;
} AS T;

 HOOK HOOK_CC2420TansmitP_receive{
  INTEGER_32 receiveTime;
  //The receiving time of the packet.
  receiveTime = getTimeNow();
  T:timestamp = receiveTime;
 }

 HOOK HOOK_CC2420TransmitP_send{
  INTEGER_32 sendTime;
  INTEGER_32 hopdelay;
  //The sending or forwarding time of the packet.
  sendTime = getTimeNow();
  hopdelay = sendTime - T:timestamp;
  T:delaysum = T:delaysum + hopdelay;
 }
```

**Fig. 4: TCL example for end-to-end delay measurement**

to each node. To facilitate the usage, we develop a tool, `gtc`, to integrate these operations.

### *B. TCL Program Example*

In the rest of this section, we focus on describing TCL in details. Table 1 shows the TCL keywords, including the ones used in both the configuration file and the measurement task file. Then we use an example to describe the TCL.

In this example, we describe the implementation of the delay measurement approach [10]. It measures the end-to-end packet delay based on the MAC layer time stamping [20]. A packet is timestamped when it is being transmitted and received. With an additional field in packet for recording, the delay can be calculated by accumulating sojourn time (i.e. node delay) at each node.

In the initialization phase, we implement an application `TestSDM`. We define the packet type as a structure named `TestSDMMsg`. In the measurement phase, the TCL program of our implementation is illustrated in Figure 4. This example demonstrates the use of packet configurations, hook declarations and associated operations. The measurement task starts with the declaration of two new fields in the packet structure for delay measurement. The variable `delaysum` is the sum of hop delays along the routing path, while `timestamp` is the packet reception time at the current hop. The task body consists of two simple processing functions at different hook positions. The first hook is located at the function where packets are received in the MAC layer. When a packet is received, the associated function of this hook records the receiving time into the packet. Here, `GetTimeNow()` is a built-in function which is used to obtain the current execution time. The second hook is located at the function where the packet is forwarded. At this hook position, the associated function obtains the forwarding time and computes the sojourn time at the current node. Then, this sojourn time is added to the sum of delays `delaysum` in the packet. Finally at the sink, the end-to-end delay of each packet can be obtained by reading the delay field from the packet, which records the sum of the delays before the last hop.

## C. Program Structure

The TCL programs consist of three types of statements: message configuration statements, variable declaration statements, and hook declaration statements. The example in Table 2 shows these three types of statements.

*1) Packet configuration statements:* These statements configure the structure of the payload field in a packet. As shown in line 1 to line 4 of the end-to-end delay example, two fields `delaysum` and `timestamp` are allocated to the message type `TestSDMMsg` from the reserved packet space using the keyword **ALLOC**. The keyword **AS** is used to give the message type an alias to facilitate its reference in the following program. For example, in line 9, `T:timetamp` is used to address the packet memory instead of `TestSDMMsg:timestamp`.

*2) Variable declaration statements:* Variables are declared by specifying the variable types and names. As shown in line 6, 12 and 13, three integer variables are declared. In order to provide more flexible manipulation to both the packet memory and the node memory, we support three types of integer with different length, as well as the **STRUCT** to support the structure variable.

The local variable is declared within the hook, while the global variable is declared beyond all hooks. The scope of the global variable is the entire program. If the global variable used in the TCL programs is referenced to the exiting global variables on node, the keyword **EXTERN** is used. For example, in `TestSDM`, each node is assigned a node ID whose value is stored in the global variable `TOS_NODE_ID`. If the TCL program needs to obtain the node ID, the following declaration should be added at the beginning:

```
EXTERN INTEGER_16 TOS_NODE_ID;
```

*3) Hook declaration statements:* Hook declaration statements are the key functional components of a program starting with a `HOOK` keyword. Each statement consists of two parts, hook name and TCL operations. The hook name indicates the position of this hook as described in Section IV-A. The TCL operation indicates the operation to be conducted at the specified hook position. One or more hooks can be declared to complete one measurement task cooperatively. Each measurement task is described by a single file, which is first compiled as binary codes and then loaded and executed by nodes.

## D. TCL Operations

TCL operations indicate the operations to be conducted at the specified hook positions. According to the objects being operated, the TCL operations can be divided into three categories: operations on packet memory, operations on the global variables declared in the measurement task, and operations on the existing global variables declared in the base program.

First, a TCL operation can access the packet memory. Many measurement tasks need to record or encode information of the current hop into the data packet and pass it to the next hop. To perform these operations, TCL supports the access to the packet memory through the structured address described in Section V-C. As shown in line 17 of the example shown in Figure 4, a TCL operation can read and write the field `delaysum` in the packet memory.

Second, a TCL operation can operate on the global variables declared in TCL programs. For example, in order to measure the number of packets transmitted by each node, we can write the TCL program as follows. A global variable `counterPktTrans` is declared to count the number of packets transmitted. The task consists of two hooks. The first one in the packet forwarding function is responsible for increasing the counter each time a packet is forwarded. The second hook in the packet generation function writes the counter into the packet.

```
ALLOC TestSDMMsg{
  INTEGER_16 counter;
} AS T;
INTEGER_16 counterPktTrans;

HOOK HOOK_CtpForwardingEngineP_forward{
 counterPktTrans++;
}

HOOK HOOK_TestSDMMsgC_sendMessage{
 T:counter = counterPktTrans;
}
```

Finally, TCL operations can directly read existing global variables/constants on nodes. For example, the `TOS_NODE_ID` is an existing constant storing the current node ID. A task for path measurement could directly access it and write its value into the packet.

To further facilitate programming using TCL, TCL also provides two built-in functions, `Find` and `GetTimeNow`. The method `Find(arrayName, fieldName, value)` has three parameters, `arrayName` representing the name of an array of structures, `fieldName` representing the name of the field of interest, and `value` representing the value to find. The function returns a pointer which points to the matched structures. For example, the function call `Find(routingTable, "neighbor", 5)` returns a pointer which points to the first structure with its field "neighbor" equals to 5, in the array `routingTable`. The `Find` function is preprocessed since the C language cannot perform field name matching directly.

## VI. Case Studies

We demonstrate the expressiveness of TinySDM and TCL by complementing several recent measurement tasks using TCL. Note that the potential expressiveness of TCL is far from exhaustively covered. The analysis shows that the overhead is acceptably small. We will discuss the following measurement tasks: (i) the path measurement method described in PAD,

(ii) the delay measurement method, Domo, (iii) the metric collecting process in Sympathy.

### A. PAD

PAD is a passive diagnosis approach that explores the root causes of exceptions in a running sensor network. On the node side, it employs a packet marking scheme to dynamically reconstruct the network topology. On the sink side, it employs an inference model to infer the possible root causes. We implement the packet marking scheme with the application `TestSDM` implemented in TinyOS 2.1.2 and reconstruct the network topology.

During the packet delivery, only one selected sensor node marks its ID and updates the hop count field based on a set of rules. We first configure the packet structure. As the following code illustrated, two additional fields *pass node ID* and *hop count* are added for recording measurement data. When the source node generates a new data packet, it leaves the *pass node ID* field empty and the *hop count* initially to be zero.

```
ALLOC TestSDMMsg{
  INTEGER_16 passNodeID;
  INTEGER_16 hopCount;
} AS T;
```

Each node maintains a buffer for its down-stream source nodes. The buffer entry is defined as follows. It consists of the *source node ID* and the *sequence number* of the recently received packet from the source.

```
STRUCT BUFFER_Entry {
  INTEGER_16 src;
  INTEGER_16 seqNo;
} BUFFER_Entry;
BUFFER_Entry  pkt_table [MAX_BUFFER_ENTRY];
```

The marking associated operations are put in the hook located at the receiving function.

```
HOOK HOOK_CtpForwardEngineP_SUBRECEIVE_RECEIVE{
  BUFFER_Entry item;
  IF(T:passNodeID != NULL){
    EXIT;
  }
  item=Find(pkt_table, "src", T:source);
  IF(item == NULL){
    T:passNodeID = TOS_NODE_ID;
        item=Find(pkt_table, "src", NULL);
        item->src = T:source;
        item->seqNo =  T:seqno;
  }
  ELSE{
        IF(item->seqNo+1 == T:seqno){
        item->seqNo = item->seqNo+1;
        T:hopCount = T:hopCount+1;
    }
    ELSE{
         T:passNodeID = TOS_NODE_ID;
         item->seqNo = T:seqno;
    }
  }
}
```

When receiving a packet, the node checks whether it is marked. If yes, it exits the task. Otherwise, it checks its buffer to see whether there is a packet marked with the same source node. If there is no entry, the packet is marked and the source node ID is recorded into the buffer, as well as the sequence number. Otherwise, if the sequence number is consecutive, node updates the corresponding sequence number. If the sequence number is not consecutive, the packet is marked and the buffer entry is updated.

**Overheads:** We use the lines of code written by user as the metric to show the simplicity of TCL. In this particular case, the lines of code written in TCL is 32 while a total of 57 lines code is needed in C code. We also evaluate the transmission overhead which consists of the task dissemination overhead and the measurement data collection overhead. The control packets actually being disseminated are composed of the hook ID, load address and executable binary codes. In the above implementation, the control packets have a total size of 286 bytes. To collect the measurement data, TinySDM adds only 4 bytes to each data packet.

### B. Domo

Domo is a delay measurement method that achieves lightweight and accurate per-hop per-packet delay reconstruction in WSNs. In Domo, each packet carries a sum-of-delay value which is the sum of sojourn times of a set of packets. These packets are sent/forwarded by the source node of $p$ and are between packet $p$ and its previous packet with the same source node. We establish an inequation for this constraint by finding packets that must be included in the calculation of sum-of-delay. Thus the implementation needs to modify the packet fields as well as computing the sum-of-delay on the node side.

As the following TCL program illustrated, the global variable `sum_of_delay` is used as a buffer to store the sum of node delays.

```
ALLOC TestSDMMsg{
        INTEGER_32 sum_of_delay;
        INTEGER_32 timestamp;
} AS T;
//The buffer to store the sum of node delays.
INTEGER_32 sum_of_delay = 0;
EXTERN INTEGER_16 TOS_NODE_ID;
HOOK HOOK_CC2420ReceiveP_receive{
        INTEGER_32 receiveTime;
        receiveTime = getTimeNow();
        T:timestamp = receiveTime;
}
HOOK HOOK_CC2420TransmitP_send{
        INTEGER_32 sendTime;
        INTEGER_32 hopdelay;
        sendTime = getTimeNow();
        hopdelay = sendTime - T:timestamp
        sum_of_delay = sum_of_delay + hopdelay ;
        IF (T:sourcenode == TOS_NODE_ID){
                T:sum_of_delay = sum_of_delay;
                sum_of_delay = 0;
        }
}
```

**Table 2: Expression by TCL.**

| Metric | Expression |
|---|---|
| Routing table | Hook position: In the packet generation function at the source node. Hook operations: Read the specific table data structure and put into the reserved space in each packet. |
| Neighbor list | Similar to the previous one. Read the neighbor list into the packet at the source node. |
| Packets transmitted | Hook positions: The first one is in the packet forward function, the second one is in the packet generation function. Hook operations: The first hook is responsible for declaring a counter and increasing the counter every time the hook is reached. The second hook is responsible for reading the counter into packet. |
| Sink packets received | Only need to implement at the sink side with a counter to count the received packets. |
| Sink last timestamp | Only need to implement at the sink side. |
| Node uptime | Obtain the time at the packet generation function using the built-in function `GetTimeNow()` and record it into the packet. |
| Bad packets received | Insert the hook into the CRC check function. Declare a counter and increase each time the CRC failed. |

Similar to end-to-end delay measurement described in the previous section, we use two cooperated hooks to compute the packet sojourn time on a node. The first hook locates at the packet receive function in the MAC layer, while the second locates at the packet send function. The buffer is updated by adding the sojourn time of the new packet in the second hook. If the packet being transmitted is a local one, the sum of delays in the buffer will be written into the packet and then be cleaned.

**Overheads:** To implement Domo, a total of 22 lines of code is needed in TCL, while 66 lines of code in C. In this case, the size of the measurement task binary to be disseminated is 652 bytes and the additional overhead for collecting measurement data is 8 bytes in each data packet.

### C. *Sympathy*

Sympathy is a detecting and debugging tool designed for data collection applications in sensor networks. The insight is that the failure can be detected and localized by collecting and analyzing a minimal set of metrics. It localizes the failure to one of the three possible causes: the node itself, the transmitting path, or the sink. Sympathy consists of metric collection on the node side and failure diagnosis on the sink side. We implement the metric collection subsystem on the node side. We summarize typical metrics collected by Sympathy and how they could be expressed by TCL in Table 2.

## VII. Implementation

We implement TinySDM on the TinyOS 2.1.2/TelosB platform. In TinySDM, we implement a measurement component

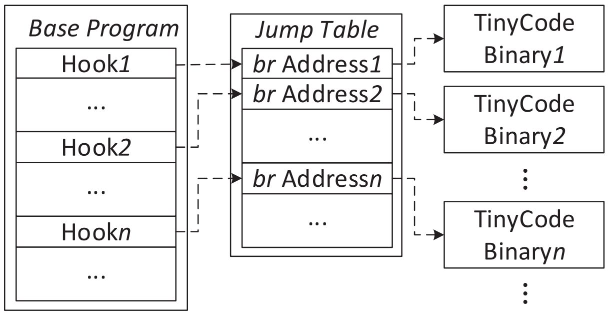


**Fig. 5: The redirection scheme of the jump table.**

to provide the control of measurement tasks at each node. The measurement component consists of an agent, a jump table and the measurement task binary. The agent is responsible for analyzing the control packets and performing corresponding operations, such as load measurement code to RAM or terminate an existing task. Both of these operations will result in a modification to the jump table which is also performed by the agent. In this section, we first describe the measurement component implementation in detail. Then we will describe how TinySDM disseminates the measurement task binaries to all nodes in a reliable and consistent manner.

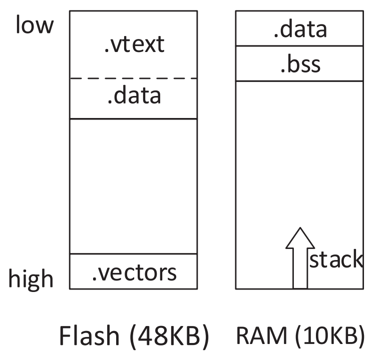
### A. *Jump Table*

The jump table is placed at a fixed location in the RAM and can be updated dynamically. For each hook, we allocate an entry in the jump table. Each entry contains a branch instruction which can direct the instruction flow to the corresponding measurement task binary. The jump table is automatically generated according to the hook configuration file in the initialization phase. Initially, each entry contains only a return instruction without any other operation. The redirect scheme is illustrated in Figure 5. When a hook is reached, the instruction flow is directed from the base program to the corresponding entry in the jump table. Then the branch instruction in that entry redirects the instruction flow to the memory address of the associated task. Once the execution is finished, the instruction flow returns to the base program and resumes execution. This process can be viewed that the program reached a mini context switch at each hook.

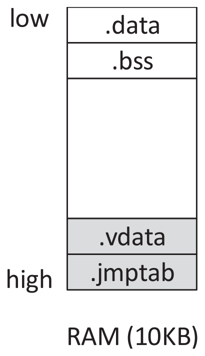
### B. *Memory Layout*

In the current implementation, we focus on the TelosB sensor node. In the discussion section, we will discuss how to port TinySDM to other platforms. A TelosB node provides 48KB program memory (0x4000-0xFFFF) and 10KB RAM (0x1100-0x38FF).

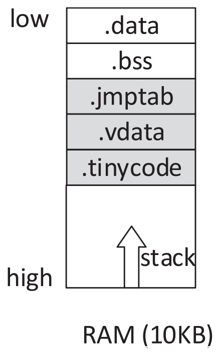
In TinyOS, the nesC code is compiled into C code by the nesC compiler. The C code is then compiled and linked into an executable file (ELF) by the GCC toolchain. Since TinyOS uses a simple binary file format (.ihex) for transferring and loading, the ELF file is further transformed into a simplified



(a)



(b)



(c)

**Fig. 6: Memory layout in RAM and the program flash. (a) Memory layout for TinyOS. (b) Memory layout in RAM after the first phase of linking. (c) Memory layout in RAM after the second phase of linking.**

format. This loadable binary file is stored in the external flash of sensor nodes and can be loaded onto the specified program memory address, including the `.text` section, `.data` section and `.vector` section. When the code execution starts, the initializer (usually located at 0x4000) initializes the `.data` section and the `.bss` section in RAM. Figure 6a illustrates the memory layout on the TelosB platform after this process.

In TinySDM, three additional sections are added in the RAM, the `.vdata` section for global variables, the `.jmptab` section for the jump table and `.tinycode` section for measurement task binaries. We put the `.jmptab` section and the `.tinycode` section in the RAM instead of the program flash since they are changing frequently.

Having the efficient utilization of memory in mind, we carefully organize these sections as shown in Figure 6c. First, the `.jmptab` section is placed directly after the .bss section and the address is fixed during the runtime. The size of jump table is determined by the configuration file which is also fixed. Second, the `.vdata` section is placed after the `.jmptab` section. Note that the `.vdata` section has a fixed size which is reserved for possible use of global variable in the future. Third, the `.tinycode` section directly follows the `.vdata` section.

In TinySDM, we use a two-phase linking process to determine the base address of each section so that these sections can be placed compactly and memory can be more efficiently used. In the first phase, we specify non-conflict base addresses to the linker as follows:

$$\text{START}_{.\text{jmptab}} = \text{END}_{\text{RAM}} - \text{SIZE}_{.\text{jmptab}}$$

$$\text{START}_{.\text{vdata}} = \text{START}_{.\text{jmptab}} - \text{SIZE}_{.\text{vdata}}$$

where $\text{SIZE}_{.\text{jmptab}}$ and $\text{SIZE}_{.\text{vdata}}$ are known after compilation. Figure 6b illustrates the memory layout in RAM after this linking phase. Then we use them to determine the final base addresses:

$$\text{START}_{.\text{jmptab}} = \text{END}_{.\text{bss}}$$

$$\text{START}_{.\text{vdata}} = \text{START}_{.\text{jmptab}} + \text{SIZE}_{.\text{jmptab}}$$

$$\text{START}_{.\text{tinycode}} = \text{START}_{.\text{vdata}} + \text{SIZE}_{.\text{vdata}}$$

Figure 6c illustrates the final memory layout after the two-phase linking process. Note that it is also possible to use a linker script to place these sections in a compact manner, which is considered as future work.

## C. *Tiny Code Dissemination*

In TinySDM, measurement tasks are programmed in TCL at the PC side by the user. The TCL programs are compiled into binaries by the TinyCode compiler. Then the measurement task binaries are disseminated to all nodes in the network. We employ the dissemination protocol described in R3 [13] which is based on Deluge [12]. In Deluge, it transmits the entire page (1,104 bytes) even when the data size is far smaller than the page size. Since the size of the measurement task binary is usually small, we use the optimized version of Deluge described in R3 for reliable data dissemination.

In TinySDM, multiple nodes may be involved in the same measurement task, e.g., end-to-end delay measurement. Therefore, it is important that every node in the network is conducting the same version of measurement task. In TinySDM, we employ a version control strategy through keeping a version number both at the sink side and the node side. Each time a new task is deployed or an old one is terminated, the version number increases by 1. At the node side, the version number is piggybacked every time when a new packet is generated. At the sink side, when a packet is received, the sink checks whether the version number in the packet is consistent with the one the sink keeps. If a mismatch is detected by the sink, the sink will retransmit the control packets carrying the measurement task binaries.

## VIII. Evaluation

In Section VI, we have shown the expressiveness of TCL. In this section, we focus on evaluating the deployment efficiency and the overhead of TinySDM. For deployment efficiency, the main metric is the completion time of disseminating control packets to update a measurement task. Since the size of disseminated data is a key impact factor of the completion time, we also evaluate the sizes of the disseminated data using TinySDM as well as two reprogramming approaches. For overhead, we focus on the CPU and memory overhead of TinySDM. The CPU overhead represents the runtime overhead of TinySDM, and the memory overhead includes the ROM and RAM overhead of TinySDM.

As described in the previous section, we implement four typical measurement approaches on our TinySDM system. The four approaches are End-to-End delay measurement, PAD, Domo and Sympathy. For comparison, we also implement them directly without TinySDM. Reprogramming approaches are employed under this circumstance for task deployment.

## A. Experiment Setup

The experiments are conducted in a testbed of 60 TelosB nodes. The 60 nodes are arranged in a grid with 0.8m separation between adjacent nodes. The sink node is located at the center. The nodes are running operating system TinyOS 2.1.2. The PC platform of the base station is Ubuntu 13.10 with 2.3GHz quad-core CPU and 4GB memory. For the base program, we implement `TestSDM` based on the `TestNetwork` application which includes many TinyOS services such as the collection tree protocol CTP [27] and the optimized version of Deluge described in R3 [13].

## B. Deployment Efficiency

To show the efficiency of task updating in TinySDM, we evaluate the completion time of the control packets dissemination. Figure 7 shows the result of disseminating the binary codes of PAD. Since in WSNs, the transmission power level is closely related to the dissemination performance. We evaluate the impact of transmission power on the completion time of control packets in TinySDM. We set three different transmit power levels: “Low” represents $-23$dBm, “Medium” represents $-20$dBm, and “High” represents $-10$dBm. As shown in Figure 7, the completion time decreases as the power increases, and 90% of the nodes successfully received the control packets in less than 30s.

Since the size of the disseminated data is the key impact factor to the completion time, we also reports the size of the disseminated data. In TinySDM, the disseminated data is the control packets including the new measurement task binary. We also used two reprogramming protocols, Deluge [12] and R3 [13], to compare their performance to TinySDM. Deluge is the standard reprogramming protocol, while R3 is the most recent incremental reprogramming approach in WSNs. Table 3 shows the size of the disseminated data for deploying four different measurement tasks. From the table, we can see that TinySDM significantly reduces the size of the disseminated data for a task deployment.

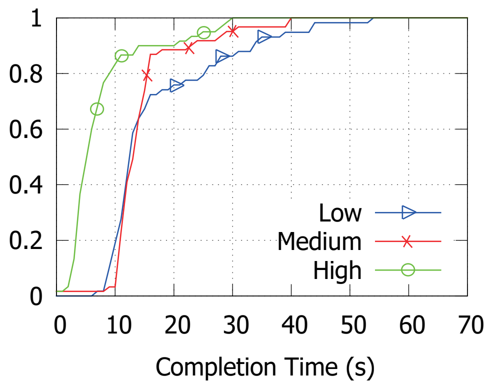


**Fig. 7: The completion time of control packets in TinySDM with different level of transmission power.**

**Table 3: The sizes of disseminated data using three approaches (in bytes).**

| | Deluge | R3 | TinySDM |
|---|---|---|---|
| E2E | 41232 | 8884 | 612 |
| PAD | 39024 | 2794 | 280 |
| Domo | 40128 | 8886 | 646 |
| Sympathy | 41232 | 5409 | 552 |

## C. TinySDM Overhead

**CPU overhead.** We use CPU cycle consumption to measure the runtime overhead. In TinySDM, hooks are inserted into the system codes or application codes which are further compiled into a single image. All hooks are initially empty as there is no task attaching to them. When a new measurement task is deployed, it is attached to corresponding hooks. When the hook is reached, the corresponding binary code snippets are executed, i.e., the corresponding hook operations are performed.

We evaluate the CPU cycle consumption of both the empty hooks and the hook operations. We identify that each empty hook consumes 22 CPU cycles. The item in the jump table corresponding to an empty hook contains a return instruction. This will cause the instruction flow to return back to the base program without doing anything. Thus, the overhead of adding multiple hooks is usually tolerable.

The cost of hook operations is dominated by the memory access latency. To evaluate the CPU consumption when operations are performed, we measured three kind of operations: when the hook operation is to read a 16-bit memory variable, when it is to write a 16-bit memory variable, when it is to write a 16-bit variable into the packet memory. On average, the read operation on node memory variable costs 10 CPU cycles, the write operation on node memory variable costs 13 CPU cycles, and the write operation on packet memory costs 15 CPU cycles.

**Memory Overhead.** The memory overhead includes the program memory overhead and the RAM overhead.

The program binary size is the size of base program

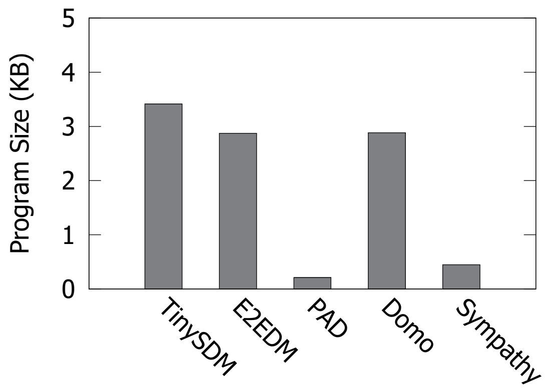


**Fig. 8: The program size of base program in TinySDM and four cases implemented without TinySDM.**

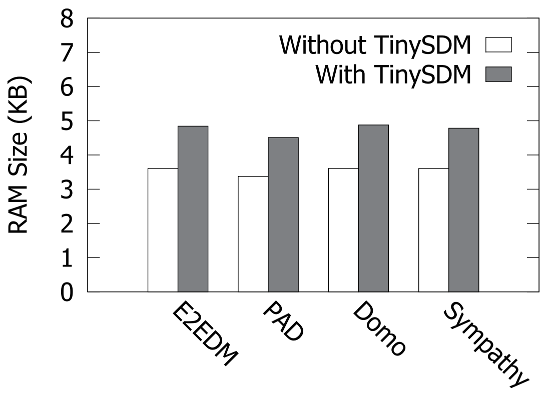


**Fig. 9: The RAM consumption of four cases implement with TinySDM and without TinySDM.**

in TinySDM and remains the same during the runtime. In TinySDM, the program binary size increases due to the agent code, the jump table and the inserted hooks. We evaluate the program binary size after implementing TinySDM, i.e., the size of the base program. For comparison, we also evaluate the program binary size after implementing four example cases without TinySDM. As mentioned in the evaluation setup subsection, all the five approaches are implemented based on the `TestNetwork` application in TinyOS. Therefore, we report the overhead by subtracting the program memory consumptions of the five approaches by the program memory consumption of `TestNetwork` itself.

Figure 8 shows the results. E2EDM in the figure represents the end-to-end delay measurement case. The program memory overhead of TinySDM is about 3.4KB. The program memory overheads of four measurement tasks are smaller than TinySDM. However, the base program in TinySDM is able to support each of these four measurement tasks, as well as many other measurement tasks. Considering the benefit of TinySDM in term of easy customization and efficient deployment, the program memory overhead is acceptable.

The RAM consumption is shown in Figure 9. We show the statically allocated RAM of the four cases both with and without TinySDM. We can see that using TinySDM needs about 1KB more RAM compared with the direct implementation without TinySDM. This mainly due to the jump table and binary codes of the measurement task.

## IX. Discussion

In Section VII we present the implementation details of TinySDM on TinyOS/TelosB platform. In this section, we further discuss how to extend TinySDM to other platforms, as well as how to achieve a good balance of easy programming and measurement ability.

**Extension to other platforms.** We currently implement TinySDM in the TinyOS/Tmote platform. However, its principles can also be extended to other platforms. For example, in order to port TinySDM to Contiki OS [28], there are two main modifications which should be conducted. First, the default hook positions in the initialization phase should be reconfigured according to the implementation of Contiki OS, as well as the base program generation. Second, the dissemination procedure of the binary code of measurement tasks should be implemented based on the dissemination protocol in Contiki. TinySDM can also be ported to Mica nodes. Since Mica nodes use Atmega128L controller which is based on the Harvard architecture, TinySDM can be implemented by placing the binary code of measurement tasks into the flash instead of RAM. The main goal of TinySDM is to provide the general measurement architecture to support many measurement tasks. When TinySDM is implemented in different platforms, the network operator can focus on the measurement task design, without a deep understanding of the platform-dependent implementation details.

**Tradeoff between easy programming and measurement ability.** In TinySDM, we provide a set of default hook positions based on a careful analysis about typical measurement tasks. A network operator can also customize its own hook positions to support more measurement tasks. By this design, a measurement task can be easily programmed since the network operator does not need a deep understanding about the application details. However, the measurement ability is limited by the hook positions.

Dynamic instrumentation [29] allows a network operator to add new hooks after network deployment, which provides a higher measurement ability. By this design, a measurement task can be easily programmed since the network operator does not need to understand the application in detail. Therefore, there is a tradeoff between easy programming and measurement ability. In TinySDM, we focus on easy programming. We show that the default hook positions are sufficient to implement many measurement tasks in sensor networks. If a network operator wants to add a measurement task which is not supported by the existing hooks, the operator can still use dynamic instrumentation techniques to add new hooks, which is considered as future work.

## X. Conclusion

This paper proposes a software defined measurement architecture called TinySDM, which allows easy programming and supports multiple measurement tasks. TinySDM facilitates the development of measurement task by providing a C-like language called TinyCode Language (TCL). It reduces the task deployment time significantly since it only needs to transfer the binary code of measurement tasks. It also avoids the cost of node reboot since it enables adding and deleting measurement tasks at runtime. We describe the design and implementation of TinySDM on top of the TinyOS/TelosB platform. We demonstrate the expressiveness of TCL by describing the implementation of four measurement tasks in the literature, including end-to-end delay measurement, PAD, Domo, and Sympathy. We evaluate its performance in real testbed and the results show that TinySDM is general, efficient and easily programmable.

There are multiple directions of future work. For example, we would like to port TinySDM to other embedded platforms and OS. We would also like to implement more measurement tasks using TinySDM.

## References

[1] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and Yield in a Volcano Monitoring Sensor Networks," in *Proc. of USENIX OSDI*, 2006.

[2] L. Mo, Y. He, Y. Liu, J. Zhao, S.-J. Tang, X.-Y. Li, and G. Dai, "Canopy closure estimates with GreenOrbs: sustainable sensing in the forest," in *Proc. of ACM SenSys*, 2009.

[3] M. Ceriotti, L. Mottola, G. P. Picco, A. L. Murphy, S. Guna, M. Corra, M. Pozzi, D. Zonta, and P. Zanon, "Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment," in *Proc. of ACM/IEEE IPSN*, 2009.

[4] Y. Liang and R. Liu, "Routing topology inference for wireless sensor networks," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 2, pp. 21–28, 2013.

[5] Y. Liu, K. Liu, and M. Li, "Passive diagnosis for wireless sensor networks," *IEEE/ACM Transactions on Networking*, vol. 18, no. 4, pp. 1132–1144, 2010.

[6] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, "Sympathy for the sensor network debugger," in *Proc. of ACM SenSys*, 2005.

[7] W. Dong, Y. Liu, Y. He, T. Zhu, and C. Chen, "Measurement and analysis on the packet delivery performance in a large-scale sensor network," *IEEE/ACM Transactions on Networking*, vol. 22, no. 6, pp. 1952–1963, 2014.

[8] Y. Gao, W. Dong, C. Chen, J. Bu, W. Wu, and X. Liu, "iPath: Path inference in wireless sensor networks," *IEEE/ACM Transactions on Networking*, vol. 24, no. 1, pp. 517–528, 2016.

[9] Y. Gao, W. Dong, C. Chen, J. Bu, T. Chen, M. Xia, X. Liu, and X. Xu, "Domo: passive per-packet delay tomography in wireless ad-hoc networks," in *Proc. of IEEE ICDCS*, 2014.

[10] J. Wang, W. Dong, Z. Cao, and Y. Liu, "On the Delay Performance in a Large-scale Wireless Sensor Network: Measurement, Analysis, and Implications," *IEEE/ACM Transactions on Networking*, vol. 23, no. 1, pp. 186–197, 2015.

[11] Y. Yang, Y. Xu, X. Li, and C. Chen, "A loss inference algorithm for wireless sensor networks to improve data reliability of digital ecosystems," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 6, pp. 2126–2137, 2011.

[12] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proc. of ACM SenSys*, 2004.

[13] W. Dong, C. Chen, J. Bu, and W. Liu, "Optimizing Relocatable Code for Efficient Software Update in Networked Embedded Systems," *ACM Transactions on Sensor Networks*, vol. 11, no. 2, pp. 22:1–22:34, 2014.

[14] X. Zheng, Z. Cao, J. Wang, Y. He, and Y. Liu, "ZiSense: Towards Interference Resilient Duty Cycling in Wireless Sensor Networks," in *Proc. of ACM SenSys*, 2014.

[15] Y. Zhang, S. He, and J. Chen, "Data Gathering Optimization by Dynamic Sensing and Routing in Rechargeable Sensor Networks," *IEEE/ACM Transactions on Networking*, vol. PP, no. 99, pp. 1–15, 2015.

[16] Z. Li, M. Li, and Y. Liu, "Towards Energy-Fairness in Asynchronous Duty-Cycling Sensor Networks," *ACM Transactions on Sensor Networks*, vol. 10, no. 3, pp. 38:1–38:26, 2014.

[17] X. Lu, D. Dong, X. Liao, S. Li, and X. Liu, "PathZip: A lightweight scheme for tracing packet path in wireless sensor networks," *Computer Networks*, vol. 73, no. C, pp. 1–14, 2014.

[18] M. Keller, J. Beutel, and L. Thiele, "How was your journey?: uncovering routing dynamics in deployed sensor networks with multi-hop network tomography," in *Proc. of ACM SenSys*, 2012, pp. 15–28.

[19] G. Hartl and B. Li, "Loss inference in wireless sensor networks based on data aggregation," in *Proc. of ACM/IEEE IPSN*, 2004.

[20] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi, "The flooding time synchronization protocol," in *Proc. of ACM SenSys*, 2004.

[21] R. Fonseca, P. Dutta, P. Levis, and I. Stoica, "Quanto: Tracking Energy in Networked Embedded Systems." in *Proc. of USENIX OSDI*, 2008.

[22] Q. Cao, T. Abdelzaher, J. Stankovic, and L. Luo, "Declarative Tracepoints: A Programmable and Application Independent Debugging System for Wireless Sensor Networks," in *Proc. of ACM SenSys*, 2008.

[23] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[24] L. Galluccio, S. Milardo, G. Morabito, and S. Palazzo, "SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for WIreless SEnsor networks," in *Proc. of IEEE INFOCOM*, 2015.

[25] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch." in *Proc. of USENIX NSDI*, 2013.

[26] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of little minions: using packets for low latency network programming and visibility," in *Proc. of ACM SIGCOMM*, 2014.

[27] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection Tree Protocol," in *Proc. of ACM SenSys*, 2009.

[28] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *Proc. of IEEE LCN*, 2004.

[29] W. Dong, L. Luo, and C. Huang, "Dynamic Logging with Dylog in Networked Embedded Systems," *ACM Transactions on Embedded Computing Systems*, vol. 15, no. 1, pp. 5:1–5:25, 2015.