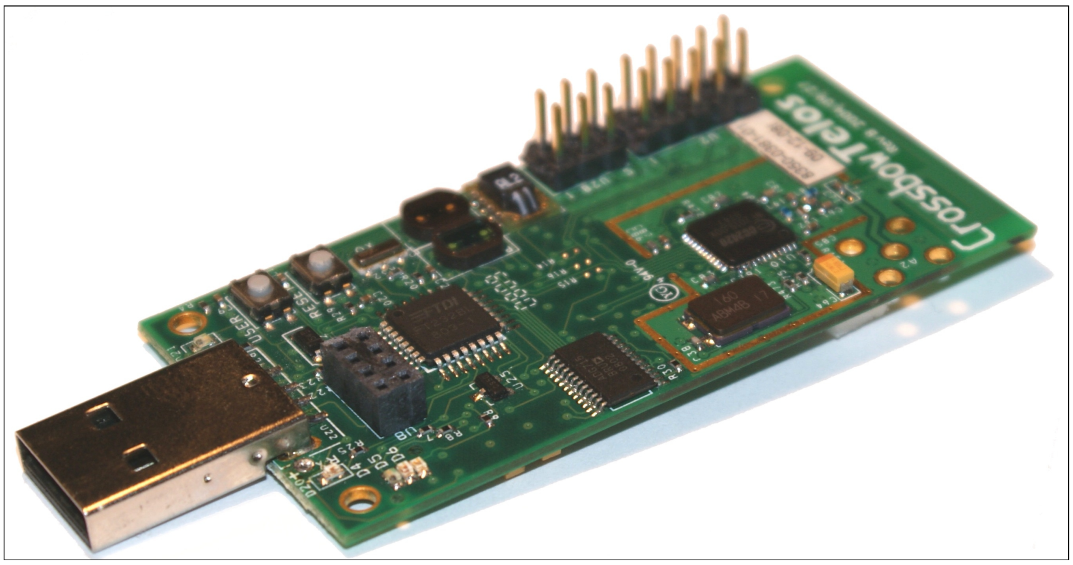
CHALMERS



Debugging Wireless Sensor Networks
with Incremental Snapshots

Master of Science Thesis in Computer Systems and Networks

SALVATORE TOMASELLI

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2013

Capturing events from WSNs
and using those events to replay and debug.

© Salvatore Tomaselli, June 2013.

Examiner: Olaf Landsiedel

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

[Cover: a Telos node.]

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

Abstract

Wireless sensor networks (WSNs) consist of cheap nodes equipped with sensors and radio, able to pass data and control devices.

Debugging WSNs is an open problem because they consist of several independent systems without advanced debugging capabilities, usually deployed in unprotected environments. Simulators are often used to reproduce real networks; with the extra capability of being able to accelerate or slow down the time and inspect the internal state of every node in the network.

Often an important condition for being able to fix a bug is the possibility of reproducing it consistently, identifying the causes and conditions that trigger the problem; however, it can happen that some conditions will present themselves only in the real network, and possibly in a way that appears totally random, making the implementation of a solution extremely time consuming.

In this thesis we present a tool able to collect events from a network and replay them in a simulator. We introduce the design of the tool, show real world applications and evaluate the performances.

Contents

Chapter 1

Introduction

Wireless sensor networks (WSNs) consist of cheap nodes equipped with sensors and radio, able to pass data and control devices. They can be used for diverse purposes, mostly to monitor the environment, which can be in a natural setting or inside domestic/industrial environments.

They can be effectively used, for example, to monitor volcanoes [25], or other kinds of geo-monitoring applications [30], or industrial applications [5].

The nodes (or motes) can have one or more sensors attached, to measure temperature, humidity, pressure, light, and so on. They can be used to log the information they gather and store it on a local flash memory or forward it to their neighbours using the integrated radio. The fact that they are battery powered and often deployed in places not readily accessible makes the energy requirements very stringent. Low power hardware and network protocols are used to maximise the battery life and also to reduce the cost of the devices.

The increased usage that this class of devices is experiencing, together with the vast number of possible uses make the topic interesting and yet not well explored.

Software for WSNs is commonly written using an event based model that requires less resources than the equivalent mode with blocking operations and multiple processes running. This is a more complex style than the one used on commodity machines because the program logic is implemented in many small functions and state machines are needed to implement the correct behaviour.

Different kinds of programming bugs can happen on a WSN, but their solution is made more complex both by the distributed nature of the applications and by the limited hardware used. The devices do not posses screens and common debugging techniques such as stepping are not available unless the program is executed inside a simulator instead of the actual device.

In debugging it is often crucial to understand the causes that trigger a bug, and being able to reproduce a bug can be very helpful because the flow of the program can be analysed until the problem is found. The purpose of the thesis is to help in debugging WSN by capturing the state of a network and moving it in a simulator, so that a situation can be reproduced and the program flow can be inspected using the tools provided by the simulator.

Pure functional programming languages like Haskell are based on the concept that a pure function should have some input and some output, and the output should depend solely on the input and not on internal state; however any internal state can be expressed as explicit input to a function and any variation to it can be expressed in the output of the function itself. Given input and initial internal state, a program will follow the same flow and obtain the same results all the time it is executed.

The central idea of this thesis is to design and implement a subsystem to use on WSNs able to capture the state of the entire network, by composing the states of the nodes together, so that it can be moved in a simulator to be analysed. An immediate observation is that nodes do not share memory or clocks and are not in any way in sync, so capturing a global situation in the same instant is not possible. We overcome the limitation by capturing an initial state and then the events occurring in the system, to be able to reproduce the state of a node in an instant t and throughout the time period when events are being captured. By capturing the events over a time span rather than on a single instant it is possible to obtain a more faithful copy of the situation inside the simulator, because the nodes that started the snapshot in an earlier moment can be moved forward in time to align with the nodes that took the snapshot at a later moment.

Rather than using costly messages to sync the clocks of the nodes in the network, our approach is to use the messages that the nodes send to be able to find a global order of the events captured on the various nodes. This lets us avoid the problem of clock skew, round trip time and does not require the nodes to exchange any extra message.

Our design uses instrumentation on every node and a central program on the sink that is able to collect, process and re-upload the captured data inside an identical WSN reproduced inside a simulator. In this thesis, we introduce a set of tools to (1) log events with minimal intrusion of deployed systems, and (2) deterministically replay of events in controlled environments such as system simulators.

Chapter 2

Background and related work

To design and implement the proposed tool, previous literature was reviewed to find limitations and similarities with the problem.

2.1 Synchronized clocks

The most immediate solution is to globally log events from a network and use a global clock to mark the events and be able to obtain an accurate sorting afterwards. However, as shown by other works[20] synchronizing clocks of motes is quite difficult and expensive, as it is not possible to have an exact figure of the time needed to send, receive, and process messages. Apart from that, clock drift is influenced by environmental temperature and changes over time. Clock synchronization also requires a number of messages being exchanged to reduce the delta of the clocks in the network under the desired value. All these limitations make synchronized clocks inapt for a debugging system that is lightweight and non-intrusive.

We believe that precise clocks are not needed for our application, as partial ordering is sufficient to produce consistent global snapshots.

2.2 Operating system

Operating systems meant to run on embedded systems tend to be more tied to their specific task and less similar to general purpose systems. Contiki [8] and TinyOS [2] are both systems for use in environments with constrained resources and make heavy use of event based programming.

TinyOS [2] is a multi-platform event based OS implemented in NesC [11]. It provides several modules at different level of abstraction to interact with the hardware and provides library functions. During the compilation process the application is compiled together with the OS in a single binary that can be run by the motes of the WSN. This has the advantage of using less memory, not needing run-time linking or virtual memory, but only one program can be loaded and run, and there is no separation between the OS and the application, which is then able to access any low level functionality with no restrictions. It has no concept of multiple users or processes, does not use virtual memory, has a limited support for threads and does not implement blocking I/O operations. Programs designed to run on TinyOS are deeply affected by those differences and need to be written in an appropriate style, that leaves space for unexpected interleaving of events or other errors, when not programmed carefully.

A TinyOS application (and the OS itself) is composed by a set of modules that can provide or use certain interfaces. For example all the modules pertaining the sensors can provide an interface to power the device on or off. An interface defines a group of commands and events; they both are functions but the direction in which they are called changes, so they can be used to provide a two way of communicating between the modules. Generally events are generated as consequence of hardware interrupts to signal the event to the application, while commands are used by the application to set the status of the lower layers.

The modules used need to declare which interfaces they use and/or provide, and then a configuration file is needed to *wire* the modules together, connecting the interface provided by a module to another module that uses it. The mechanism is quite generic and also allows multiple instances of some modules (if they specifically allow that) and multiple wirings. Modules can be easily replaced by other modules providing the same interfaces, and can be reused in multiple applications.

Besides commands and events defined in interfaces, modules can contain task functions. They are low priority functions that can be preempted by higher priority functions but not by other tasks, and are meant to perform long operations splitting them in many steps. Tasks can be posted and they will be queued for execution and run to completion. A task performing a very long operation should hence split it in phases and post itself to be scheduled in the next cycle until the operation is fully completed. The fact that a task cannot be preempted by other tasks generates a situation where tasks that run for too long and starve other tasks leading to bugs.

2.3 Debugging distributed systems

WSNs, like all distributed systems, base their operation not only on the internal state of a node, but on the global state of the entire network, which also includes messages that are in transit, along the internal states of all the nodes. The correctness of the operation depends on the aggregate state of the entire network. In non-distributed applications simulating all the possible states presents prohibitive costs that for any real application make the technique infeasible. In a distributed system the amount of possible states grows exponentially.

A number of tools and systems have been devised to help dealing with distributed systems. A common approach is to collect traces from the nodes of the network and use them to create global snapshots of the network to later enable replaying [7, 12, 13, 22]. These tools are designed to work on Internet based applications and do not share the same strict resource constraints that are present on WSNs, however basic ideas and approaches can be reused and adapted to embedded systems.

2.3.1 WSNs

Typical nodes used for WSNs provide RAM that is up to six orders of magnitude less than what can be found on a common PC. For example a TelosB (a mote architecture, based on an MSP430 CPU) mote comes with 10kB of RAM. Comparing the available resources on the different classes of systems, it is clear that the debugging on WSNs must present substantial differences to obey to the constraints.

A number of WSNs specific approaches have been proposed. For example [33] periodically collects the aggregated state of the nodes to see if faults occurred.

NodeMD [18] focuses on detecting faults at runtime to prevent the nodes from being completely disabled by them. Its fault detection component is capable of detecting overflows, locks, and other application-defined faults. It detects stack overflows by inserting instrumentation in the code to inspect the stack and registers at every function call, checks for locks using a software watchdog that is instructed at compile time with estimates of the time needed by a thread to complete a task. The notification component logs the fault in a circular buffer, and the diagnosis component can bring the node in a recovery state where it can be remotely operated to inspect the status and the log, for debugging. This system can be very helpful in dealing with deployed nodes preventing them to go offline, but it does not take networking into account.

Sympathy [27] is a metrics collector for data collection applications. It is based on the observation that insufficient amount of data collected in the sink is caused by faults in the network; so it uses data flow for the metrics. Then it uses the data to indicate failures their possible source. It is used to report failures but has a limited utility in investigating their original cause.

Dustminer [17] logs events in the network and then uses a probabilistic data mining approach to automatically scan the trace to try to expose the sequence that led to a fault.

MDB [28] uses the higher level abstraction of *macroprogramming* to abstract the complexity of using multiple machines and introduces a debugger a la GDB, allowing different views of the network, stepping and hypothetical changes, where changes are introduced without the need to re-run the entire program. The tool has a substantial energy overhead so it is unlikely to be used on deployed situations but the limitation does not affect testbeds.

Other works focus on the introduction of programming abstractions [6, 21, 23, 24] and do not focus on the debugging itself but in helping to create an environment where it is easier to write correct programs with less opportunity for faults to happen, while some tools [15] can use invariants and check their validity at runtime also across node boundaries. This introduces an overhead on the number of bytes that need to be sent and a small overhead on the memory used. Another drawback of the method is that the invariants across nodes are not tested in real time.

A closer approach to our method is used in [29] where each functional call and its parameters are logged to provide the user with a complete view on the system; but the trace can't be replayed. The authors use a combination of methods [3, 19] to trace the flow of the program inside the functions and across them, exploit the presence of paths that are much more used than the paths relative to exceptional conditions by compressing them. Since generic algorithms such as LZW are too heavy for the nodes, they identify at compile time the most common sequences and generate lookup tables to compress them. They use two buffers to be able to trace all the activity on one buffer while the other is being written on the flash memory, and then use the traces to be able to inspect the flow of the program at a later time.

In this thesis we agree that tracing all the input is costly but we argue and we show that it is not necessary to trace the input entirely if the state of a node at a given time can be restored and replayed deterministically.

Chapter 3

Design

To achieve the goal of recreating the snapshot of an entire WSN, the system must be capable of creating snapshots of the single nodes that compose the network. Composing together snapshots of the single parts is not a solution that would work, because their lack of shared clock and memory makes it impossible to instruct an entire network to capture a snapshot in a given moment. Various delays in the communication and in the scheduling would make the result unreliable.

The chosen approach to reach the goal is to capture and replay the events occurring on the nodes, replicating the input, to obtain a program that can re-perform the same calculations. We consider several types of events in all the nodes of the network, and as some of them are messages between the nodes, it will be possible to devise a partially ordered global sequence of events, where only the order of the events that include interaction between nodes are globally sorted, and the relative order between the local events on different nodes is unknown.

To make it possible, all nodes must be capable of logging some events during their normal operation, and store them. The developer will have to decide which event sources are interesting or necessary for the replay. As the events from the entire network need to be compared with each other to find the matching events, the nodes must be able to send the entire log to a sink node that must be able to collect and analyse all this data.

In this chapter we provide details on how the system and its various parts must be put in place and an overview of how they act and what tasks they carry out.

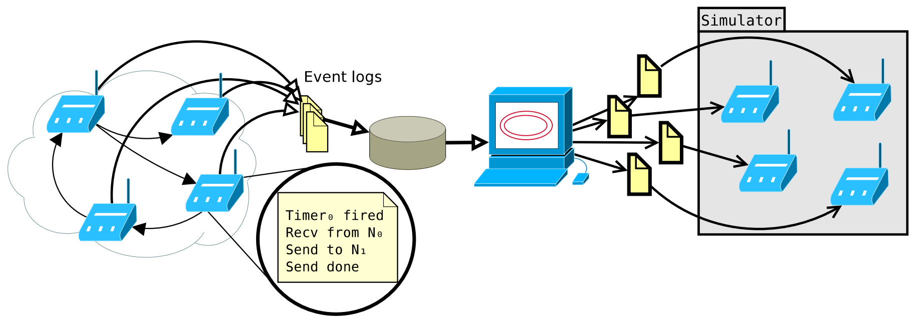


Figure 3.1: Global architecture of the system. The events from the WSN are collected, processed and sent to the simulator to be replayed.

3.1 Nodes

The TinyOS[16] event based operating system is used for the implementation, so all the nodes are equipped with this OS, which can be modified and expanded to fit the needs of the project. Since it is implemented using nesC[11], all the changes need to be implemented using the same language.

In addition to the normal program, the design is to add extra instrumentation to allow events to be captured, stored and replayed. This additional instrumentation can operate in three modes: normal, log, replay.

normal which is also the default state, lets the program run normally.

log in this mode the system works normally but all the events generated by the watched modules are being stored.

replay in this mode the events generated by the watched modules are ignored, and the node is able to replay events grabbing them from the event log.

3.1.1 Wrappers

To allow the architecture to be extendable, the concept of wrapper is introduced.

A wrapper is a piece of code that wraps around an event source, and performs some collateral actions (such as logging the event) before making the right calls and resuming the normal flow of the program. This component is in charge of both logging and replaying the events related to the wrapped event source.

Applications use different modules, that might or might not need to be debugged, the developer of the application is in charge of deciding which components of the application need to be wrapped. As this design is meant to be extendable a wrapper is only required to implement a certain minimal interface and is left free on the internal implementation details. However all wrappers perform a similar function and it is normal to expect similarities in their design.

A common behavior for a wrapper upon receiving an event is illustrated in Algorithm 1. Showing how a wrapper is expected to behave in the different states.

Algorithm 1 S is the state in which the logging module is. P are the parameters to the event

```
1: procedure EventRaised($P$)
2:     if $S = NORMAL$ then
3:         $event(P)$    ▷ Calls event, which is the normal event handling procedure in the application
4:     else if $S = LOG$ then
5:         $LogEvent(P)$
6:         $event(P)$    ▷ stores the event in the log, and passes the event to the application.
7:     else if $S = REPLAY$ then    ▷ Simply ignores the event
8:     end if
9: end procedure
```

As an event source could in truth generate more than one type of event, wrappers should repeat that pattern for all of the events. But in many cases it should be possible to avoid having large amounts of duplicated code.

Often components not only generate events, but accept calls to query or change their status. For example a timer can generate events but can also be queried by the application, for the clock value. As a consequence wrappers around such components must be able to deal with the fact that the component can both call and be called by the application; in case of a call coming from the application, the wrapper is expected to behave as shown in Algorithm 2 and log the call with its result, to later be able to replay it.

To deal with this situation it can be convenient to store the logs in a queue structure, extended to allow to also read and write at specific positions. This allows to replay events and function calls in a simple way, as shown in Algorithms 3 and 4. On replay, the wrapper will scan the queue, find the first entry that has a type corresponding to an event, trigger the event, and change the type of the entry to Nil. When the application calls a command, the wrapper will intercept the call and scan the logs to find the result that was yielded by that call in the original run.

Periodically the events that have been replayed, are removed from the log, to speed up the other operations, as shown in Algorithm 5. This can usually be done after having replayed an event.

Algorithm 2 S is the state in which the logging module is. P are the parameters to the call

```
procedure WrapCall($P$)
    if $S = NORMAL$ then
        return $Call(P)$                ▷ Calls the function in the component
    else if $S = LOG$ then
        $r \leftarrow Call(P)$
        $LogCall(P, r)$
        return $r$
    else if $S = REPLAY$ then
        $r \leftarrow CommandCall(Q, t)$   ▷ looks in the log for the result of the command t
        return $r$      ▷ the application gets the same return value that it received when running the first time
    end if
end procedure
```

Algorithm 3 Replay an event

```
procedure ReplayEvent( $Q$ )
    for $i \leftarrow 0$ to $len(Q)$ do
        if $Q[i].type \in E$ then           ▷ E is the set of all the types that correspond to events
            $wrappedEvent(Q[i])$     ▷ calls the function in the application
            $Q[i].type \leftarrow Nil$              ▷ Marks the event as invalid

            return $Nil$
        end if
    end for
end procedure
```

Algorithm 4 Returns to the application the same value that the command call returned during the original run

```
procedure CommandCall($Q, t$)   ▷ Q is the log queue and t the type that identifies the command
    for $i \leftarrow 0$ to $len(Q)$ do
        if $Q[i].type = t$ then
            $Q[i].type \leftarrow Nil$              ▷ Marks the event as invalid

            return $Q[i].retval$ ▷ returns the original return value, stored inside the log entry, to the application
        end if
    end for
end procedure
```

Algorithm 5 Removes from the log the events that have been already replayed.

```
1: procedure ClearLog($Q$)
2:     while $Q.head.type = Nil$ do
3:         $Q.dequeue$
4:     end while
5: end procedure
```

The shown algorithms are quite generic to provide an idea of how a wrapper should be written, but still leave room to improvement or changes if the needs for a specific wrapper should be different.

Log format

As different kinds of wrappers must be allowed without changes to the other modules, only the specific wrapper has to be aware of the data format used for the log, to let the other components remain completely unmodified when wrappers are added or modified. The interface between the wrapper and the central module only requires that all log entries must have a fixed size, and that every type of wrapper must be identified by a unique integer.

The limitation on the constant size of the log entries can cause some waste of memory when a wrapper has events with different sizes. However a wrapper is free on the internal format it uses and is just required to have a fixed length when dumping or loading the log, so a wrapper implementation is free to use compression internally.

3.1.2 Central module

A central module is also needed. Its task is to coordinate the wrappers and communicate with the external world.

In our design, the central module is the entity responsible for changing the state and start and stop the logging, as well as replaying the events. It must be possible to use this module from inside the application running in the node, as well as externally, from the testbed. The module is designed to operate over the serial port and be capable of dumping and loading logs using it.

When changing state: all the wrappers should be notified of the new state, possibly in an atomic way, so that no events can occur during the transition.

To provide local ordering of the events the central module provides a centralized counter, and all the wrappers are expected to atomically read the counter and increment it before assigning a sequence number to a log entry. Figure 3.2 shows different wrappers with stored events, each with different incremental sequence number, so that they can be sorted.

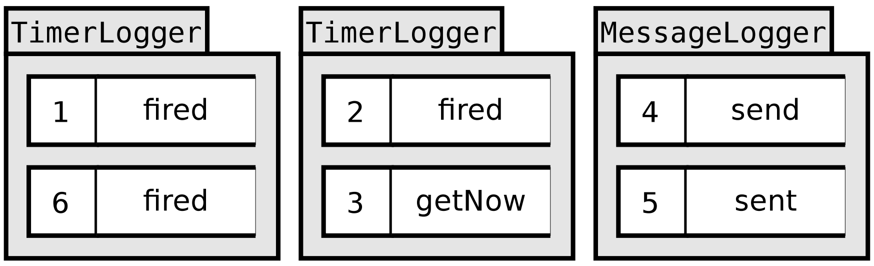
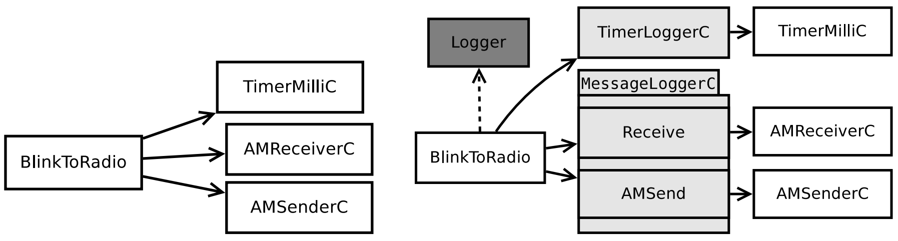


Figure 3.2: Wrappers store events, keeping a sequence number that is unique on the node.



(a) A generic application (b) The same application with the new wiring

Figure 3.3: Wiring to add wrappers. The original modules are indicated in white, the light gray ones are the wrappers and the dark one is an optional interface to the central module.

During the replay stage the central module must determine which of the wrappers has the event with the lowest sequence number, then has to instruct that particular wrapper to replay one event to be sure that the events are replayed in the correct order.

3.1.3 Wiring

The TinyOS concept of wiring modules together is exploited in this design. It is used to support the insertion of wrappers without the need of extensive changes to the code. Figure 3.3 shows a simple application, originally wired to TimerMilliC, AMReceiverC and AMSenderC. With logging in place the wiring is modified to interpose wrappers between the original module. The figure also shows an optional wiring to Logger, which can be used by the application to control the logging but is not a necessary component.

Thanks to the use of wiring and wrappers that provide the same interfaces as the original components, only the configuration file with the wiring requires modifications while the application itself is left completely unchanged.

Dump and loading logs

At compile time or at boot time, each wrapper must be connected with the central module, and identified with an unique id that must not change across reboots.

Allowing generic wrappers means that no assumptions can be made on the size of the entries of their logs. There can exist wrappers that have entries larger than the MTU, so to transfer the logs over the serial port a protocol has been designed. Every event is preceded by a packet indicating the unique wrapper id, the wrapper type and the entry size as shown in Figure 3.4, and then followed by a variable number of packets containing the raw binary data for the entry, as provided by the wrapper, and unprocessed by the central logger module.

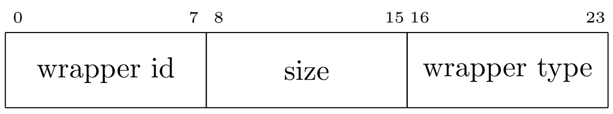


Figure 3.4: Packet format for the packet preceding a series of messages describing one single event. The wrapper type field is needed by the sink, to provide indication on how to process the following raw data.

The central module will dump the log querying all the wrappers multiple times until all their internal logs are empty. At that point a special message to indicate that the dump is complete will be sent.

The unique id per wrapper is necessary because when loading the logs it will be used by the central module to determine to which wrapper the event should be routed.

The events can be loaded back into the motes by sending the same sequence of header packet, plus raw binary data. Wrapper modules are expected to handle their own raw data.

The central module must be able to dump and load event logs while the mote is still operating, but it must be granted that while doing these operations the mote is not in log or replay states, which means that no mutual exclusion support must be implemented to access the logs.

3.1.4 Initial state

To generate the snapshot of a node, generating the same events is not enough to faithfully reproduce a program flow. Also its initial internal state must be reproduced. Several approaches have been considered for this problem.

The idea of logging from boot and only allow replay from the boot configuration has been considered but deemed to bring too many limitations on the system. Resources in general and the logging space in particular are quite limited on WSNs, so the log could only span a few seconds or minutes, depending on the activity of the node. Also, it would have been impossible for the application to turn on the logging only when a specific event is triggered.

Our approach is to let the developer manually indicate which variables and data structures need to be preserved. When the mote's state is set to log it will copy all the indicated variables to a separate memory area, which will be dumped to the host system together with the events.

Collecting the initial state is also responsibility of the central module and it should be done in the same atomic section as informing all the wrappers that the state was changed. When switching to replay mode, the initial state which is stored in a separate memory area needs to be restored before events can be replayed.

The memory area for the initial state must obviously be large enough to contain all the initial state, and hopefully no larger than that. In this model pointers pose a problem because they must be handled manually by the developer which must add both the pointer and the pointed area to the set of variables that constitute the initial state. The creation of a tool to automatically deal with pointers was considered but abandoned since pointers can have multiple levels, loops, and similar complexities.

3.2 Sink

TinyOS offers different interfaces to access the serial port. On the lowest level the interface allows sending one byte at a time and generates an event when the byte was sent[14]. The implementation of this level is also platform specific and not portable, so instead an higher level portable implementation was chosen, where frames and packets are used on the serial port; so the code on the sink must implement the same protocol to be able to communicate with the nodes.

As specified by TEP113[14], communications PC-to-mote support acknowledgments that indicate the sequence number of the message, so the software on the sink can implement a timeout and can retry if an ACK is not received within a certain time.

The implementation on the sink must be able to provide a reasonably easy to use library to work with the events, abstracting from packets as much as possible. It is very important that the events are not treated as obscure binary data because the events that are logged on multiple nodes need to be used to provide a global ordering of the events on the WSN. To allow future expansion the implementation needs to be flexible enough to easily add new event types.

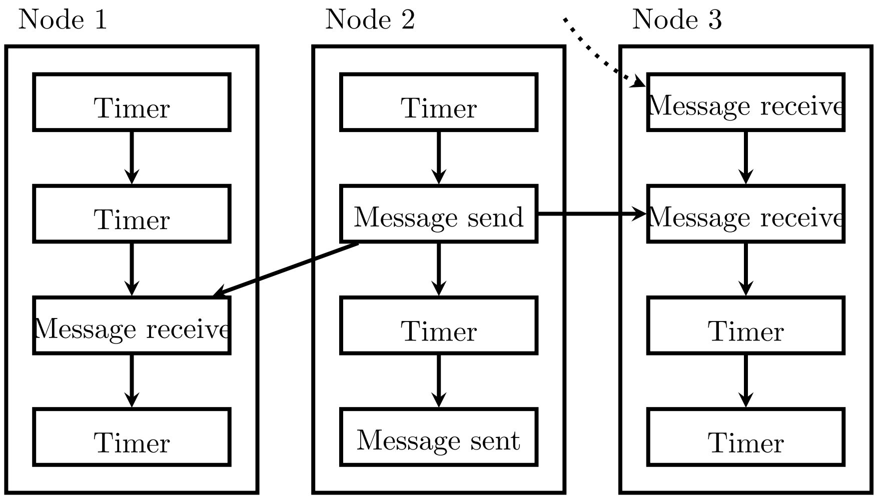


Figure 3.5: The figure shows a dependency graph for the events of three nodes. The first receive event logged does not have a counterpart so it will not depend on anything.

3.2.1 Event sorting

Since all the events carry a sequence number that is unique on every node, sorting the local events on a node is immediate. Events such as radio events or dissemination send have (or can have) a received counterpart on the other nodes. These events are used to obtain a global order of events.

To obtain a partial global ordering of events, on the sink every event is treated as a node of a dependency graph. Events that are only locals (e.g. timers) depend on their predecessor in the event log of the node, while in general events can depend on events on other nodes.

One event on a node can only be replayed if the events it depends on have been replayed.

In a sequence of events only containing local events, it is not possible to obtain a global sequence of events for the WSN, but it does not matter because local events are not affected by nor affect the other nodes in the network.

In Figure 3.5 is shown how the dependencies work. The event *message sent* does not depend on a message received for the reason that a message could be lost, so it would be incorrect to do such an assumption. Also the processing times of the sending and receiving nodes are unknown so it might very well be that the real time when the events occur does not have a precise order.

The drawback of this scheme is that in case of a network with a number of nodes that are just sending data, all the events from a sender node could occur before all the events from the other nodes (which is very unlikely to

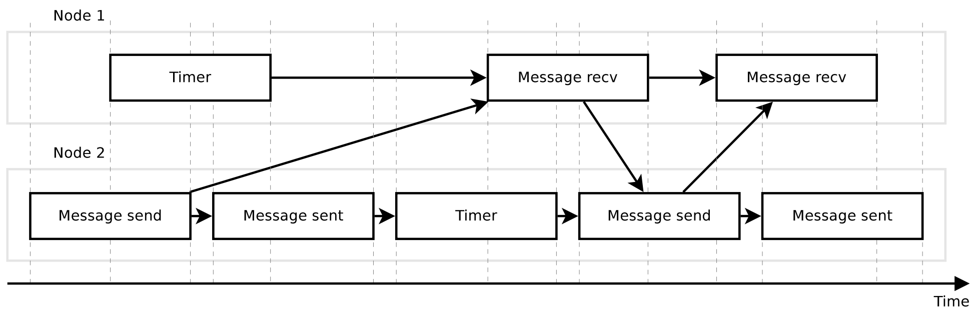


Figure 3.6: Events ordering based on dependencies, including the observation that if a node receives two messages, the second one must have been sent after the first was received.

happen in reality). To address the problem, rather than adding timestamps or unrealistic dependencies, the taken approach was based on the observation that if node B receives two messages from node A, then the second message must have been sent after the first one was received, or it would have been lost instead. This leads to the dependency scheme shown in Figure 3.6 that does not have the limitation.

Clearly lost sent messages don't have a corresponding receive event so they will automatically considered as local events instead.

The approach of using local sequence numbers and then locating dependencies on the other nodes in fact generates the same outcome as if vector clocks had been used. Letting the computation and comparison to be performed at a later time on the sink node has the advantage of not requiring the nodes to keep track of the clocks of the other nodes thus saving memory, and exchanging messages containing the local clocks is not necessary. The nodes will then not generate extra messages that might influence the run of the program with unseen side effects and interleavings, and large networks will not cause problems requiring proportionate memory overheads in the nodes.

Message matching

Matching a send event with a receive event correctly is central to obtain correct results. For TinyOS radio messages the procedure involves using the length byte, to compare only the meaningful part of the buffer and discard the garbage at the end. Sequence numbers are also zeroed before the comparison.

When the sequence number resets there is a risk of matching the wrong messages, if they happen to contain the exact same content. However generally the log space for the message wrapper is much smaller than the 256 messages needed to overflow the sequence number so the situation is not taken into account.

Chapter 4

Implementation

In this chapter we describe the specific implementation choices that were made.

4.1 Nodes

The major task of logging events is made simpler by the fact that event based programming is common on WSNs, so events logged are directly mapped on events provided by TinyOS.

The functional tests were carried out using the Cooja[9] simulator. In all the tests the Serial Socket plug-in was used to provide an interface between the motes inside the simulation and the sink.

During the work difficulties arose to synchronize the simulation with the code running on the host (for example it was necessary to insert some sleeps in the host code to give time to manually decrease the simulation speed before initiating the replay). To mitigate the problem a Cooja plug-in[31] was implemented to provide some functions to control the simulation over the D-Bus[32] IPC system, allowing code running on the host outside Cooja to control the simulation.

4.1.1 Central module

The central module provides an interface that allows other modules to control the state of the logging.

It also provides a parametrized interface meant to be wired with the wrappers. This interface is the sole way of communication between the wrappers and the central module.

It allows the central module to change the state in the wrappers, and allows the wrappers to require unique event IDs or to notify the central module to stop the logging because the memory for the log is full.

The event

```
event error_t enqueue_partial
    (nx_uint8_t *buf, uint8_t start, uint8_t len);
```

is used to load events from the sink. One event can be loaded to the wrapper in several partial chunks, so it is up to the wrapper to decide when the event is complete. The need for this arises from the fact that events have variable lenght, and the central module is not aware of the lenghts of the events from the various wrappers. Being able to transmit one event in several packets allows events of any size. All the chunks are always delivered in order, so it is possible for the wrapper to know when one event is complete and more data belongs to the next event, as shown in Algorithm 6.

Algorithm 6 buf is the pointer to the start of the event chunk. start refers to where the chunk is located in the global event and len is the size of the current chunk.

```
1: procedure EnqueuePartial($buf, start, len$)
2:     static  partial[logEntrySize]                ▷ Accumulator for the events
3:     $memcpy(partial + start, buf, len)$          ▷ appends the chunk to the accumulator
4:     if start+len = logEntrySize then             ▷ When the event is complete, is added to the log
5:         $Q.enqueue(partial)$
6:     end if
7: end procedure
```

To replay an event, first the wrapper with the event with the lowest ID must be located, and then it can be instructed to replay the event as shown in Algorithm 7.

Since wrappers log events and commands, but only events are replayed, a wrapper must execute the procedure shown in Algorithm 8. As it is possible to see, the procedure of replaying one event is not very efficient and despite a list of wrapper ID ordered according to the event priority could have been generated on the sink and uploaded to the nodes, it was decided not to do so, because this procedure is only called at debug time in the simulator and we don't care about the performances in this scenario, as long as they are reasonable.

Dump task The task of dumping the logs is implemented using the common TinyOS event-based model, so when one packet is sent, an event is raised and some internal status is checked to determine which packet will be sent after. As a consequence, other tasks posted by other modules can run while a log dump is being performed, so the node can continue to work normally while dump is in progress. Clearly the more task posting themselves are running, the longer they will all take to complete their operation.

Algorithm 7 The central module queries all the wrappers to know which one has the event with the lowest ID, and then replayes that event.

```
1:  procedure ReplayOneEvent
2:      $min \leftarrow MAXINT$
3:      $w \leftarrow Nil$
4:      for $i \in wrappers$ do
5:          if $i.headId < min$ then
6:              $min \leftarrow i.headId$
7:              $w \leftarrow i$
8:          end if
9:      end for
10:     if $w \neq Nil$ then
11:         $w.replay$                              ▷ Replayes one event.
12:     end if
13: end procedure
```

Algorithm 8 E is the set of the event types, Q is the log queue, used as a list in this case.

```
1: procedure headId($Q$)
2:     for $i \in Q$ do
3:         if $i.type \in E$ then
4:             return $i.id$
5:         end if
6:     end for
7:     return $MAXINT$
8: end procedure
```

So performances will be degraded, but the node will work. Symmetrically when loading a log, the messages are handled as they reach the node and nothing blocking occurs so the node can operate normally.

4.1.2 Queue

The queue provided by TinyOS has been extended to allow read/write access of all the stored items, to permit the modification of items, and allowing certain procedures without the need of a linked-list. All the following algorithms use such extended queue structure.

The following commands were added to the standard queue.

```
command t element(uint8_t idx);
command t* elementptr(uint8_t idx);
command void clear();
```

element allows to retrieve arbitrary elements from the queue, where 0 is always the head of the queue.

elementptr returns the pointer to the nth element, in fact allowing the modification of elements that are already in the queue. This feature is used to implement Algorithm 3.

clear allows the central module to have a way to quickly wipe out all the logs, if necessary.

4.1.3 Wrappers

All wrappers provide/implement two interfaces called In and Out (or similar names), that must be manually wired by the developer around the component that she wishes to wrap.

Internally the wrappers must implement and provide both interfaces and connect one to another, to make the application work as if the wrapper was not there, during normal mode, and to intercept and generate different events during the other modes.

All the wrappers wire to the central logger using an *unique* parameter, generated at compile time by the *ncc* compiler, ensuring that every wrapper gets a different ID. This lets the developer create as many wrapper instances as they want, since more IDs will be generated at compile time. And by using *uniqueCount* the central module can know how many wrappers are wired to it.

Timer wrapper

A generic module was implemented to wrap the TinyOS timers.

It has the precision tag and is generic so it can wrap around any TinyOS timer. Its TimerIn and TimerOut interfaces are to be wired with the timer and with the application.

Despite the message is meant to be sent over the serial port, the internal log format is a normal struct, so that in case padding is needed for the platform the compiler is free to insert it, to make the logging of the event as fast as possible.

When dumping or loading log, the events in the internal log are converted in a network format, using an *nx_struct*, as shown in Fig 4.1.

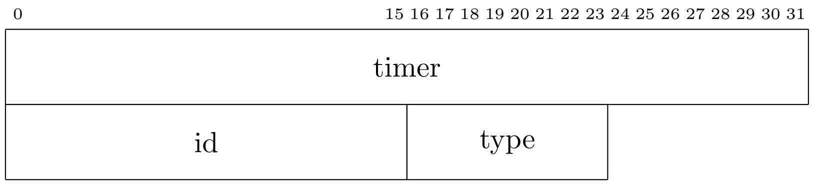


Figure 4.1: Data structure to log an entry for a timer event. The timer field stores the value of the timer, the ID field is the global ID obtained from the central logger module, and the type indicates which kind of event or command the entry is referring to.

Timer interfaces in TinyOS provide ten commands. In the wrapper to avoid massive code duplication, since all the commands are handled in almost the same way, a multiplexer function was created, that acts as shown in Algorithm 9, which is an implementation of what shown in Algorithm 2.

Message wrapper

This wrapper logs radio messages, it provides and uses Send and Receive interfaces, so only one wrapper is required to use send and receive.

Internally the structure to handle events and commands is very similar as the structure described in Algorithms 1 and 2 so it is not described here.

In the internal log format the radio message is stored with a *message_t* type, so it can just be copied by logging.

However, problems arise because the size of the *message_t* type can be changed at compile time, so an event entry sent over the network would have a different size depending on that; and that would cause the tools running on the sink to be unable to recognize the format.

To avoid random failures and indicate a problem the compilation of the program for the node will fail if the size of the *message_t* struct is not the expected one. This will let the programmer know that she will need to modify the tools on the sink to work with the new size.

A very neat macro from [26] was defined to recreate the behavior of the *static_assert* present in C++11 compilers[1]. As NesC code is compiled into C, the macro needs to work in C

Algorithm 9 The generic command multiplex procedure can log and replay any command. All the commands call to it and pass a different type, that uniquely identifies the specific command. The function Command1 is an example of how a command would use the multiplexer.

```
procedure $command\_multiplex(\ result, type\ )$
    $allocate\ ev$                              ▷ creates a variable for a log entry
    if $S = NORMAL$ then
        return $result$
    else if $S = LOG$ then
        $ev.id \leftarrow getUniqueId()$                ▷ prepares the log entry
        $ev.timer \leftarrow result$
        $ev.type \leftarrow type$
        $Q.enqueue(ev)$                                  ▷ logs the entry
        return $result$
    else if $S = REPLAY$ then
        for $i \in Q$ do
            if $i.type \in C$ then    ▷ C is the set of types corresponding to commands
                $i.type = Nil$
                return $i.timer$
            end if
        end for
    end if
end procedure
procedure $Command1$    ▷ this shows how the multiplexer is called by the various commands
    $r \leftarrow RealTimer.Command1()$
    return $command\_multiplex(r, 1)$
end procedure
```

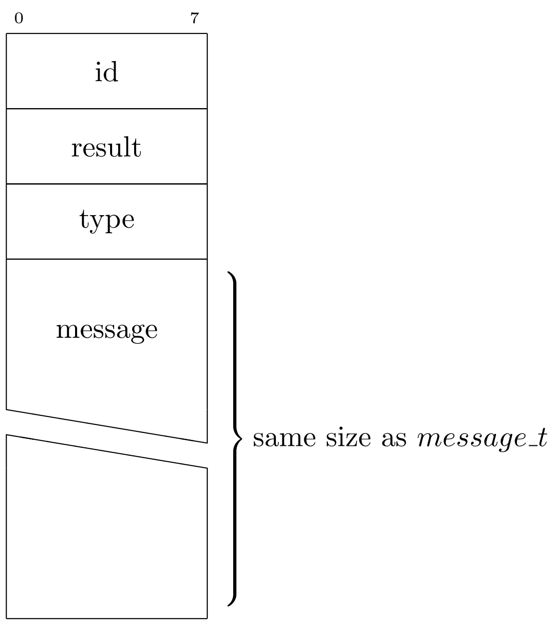


Figure 4.2: Packet format for the dump of a message event.

```
#define STATIC_ASSERT(COND,MSG) typedef char
    static_assertion_##MSG[(COND)?1:-1]
```

On failure the assert will display an error message, that will contain also the string indicated in the macro, giving some information to the developer to understand why the compilation failed.

Dissemination wrapper

TinyOS provides a Dissemination interface. Applications can use it to spread values across the network and receive them. The interface does not explain how the values are to be transmitted, so the interface needs to be wired with one implementation to be used.

In case the dissemination implementation is to be considered correct, rather than storing all the radio messages, it is possible to only log the dissemination events, to save log space.

The biggest challenge to implement a dissemination wrapper is caused by the fact that the TinyOS interface for dissemination allows the type of the variable to be disseminated to be set at compile time. And there can be many instances of dissemination modules on the same node, each with a different type.

As a result the required size to store one event can have significant variations and is known at compile time. Because of that it is possible to let the wrapper just use the needed space rather than allocate extra space.

Internally two queues are used instead of one; one holds the extra informations added by the wrapper and the other one holds the dissemination value, when dumping or loading the values from the two queues are concatenated in a single buffer.

This is possible because the central module queries every wrapper about the size of their log entries one by one regardless of the type of the wrapper (timer, radio, and so on), thus allowing separate wrappers of the same type to have log entries with different size.

On the side of the sink, matching the events is just a matter of comparing the variable field to find the ones that match. However performing more complex operations and read the content of the variable can be complex if multiple variables are being disseminated, because the type and wrapperId association is established at compile time.

Since the dissemination update interface has no events, during replay the wrapper just dequeues events without doing anything. The updater needs a wrapper anyway so that the events can be matched on the sink with the received updates on the other nodes.

The set command, used to set default values is left untouched by the wrapper, since it is meant to be used just to set the initial default. On replay if there are no events recorded, and hence the value never changed, that default will be returned.

4.1.4 Initial state

When starting the log, the initial state needs to be copied to a separate memory area.

The implementation is to have an array containing pointers and sizes of all the needed variables, and a function iterate and copy them using *memcpy* one immediately after another in the destination memory. No padding is used, making the copy potentially slower than necessary.

Post-processor tool

The procedure to copy the initial state involves an array of tuples describing the memory location and the size of all the variables to preserve.

This information is available at compile time but current compilers do not implement optimizations to move arrays to the constant pool if they are totally constant and the values they hold are known at compile time.

To keep the list of pointers and sizes on the constant pool, a Python tool was written. It parses the C program generated by the ncc compiler before it is compiled by gcc and finds all the variables that are marked to be part of the initial state to keep in storage when logging. Then it will rewrite the C code inserting instructions to generate the array in a static way so that they can be placed on the constant pool rather than on RAM.

This tool is also capable of counting the exact amount of space needed to store the initial space, thus avoiding the need to use memory areas with a static size larger than the expected amount of data they need to contain.

To inform the post-processor that a variable must be kept in the set of variables needed to be stored for the initial state, the developer must call a special function, passing the pointer to the variable and its size. The function is defined inside one of the header files and is just an empty function; its only purpose is to let the ncc compiler mangle the names of the parameters so that the post-processor can look for calls to that function in the resulting C file, collect them and use them to generate the array of pointers without needing to know the module where they had been originally declared.

Dump message format

The entire initial state will be stored in one array, and the offset of the variables in this array will depend on the compilation stage, so in general this memory area has to be considered as an obscure blob with unspecified format. Its exact format can be retrieved by inspecting the C code that is passed to gcc.

The format of this memory area is highly volatile and changes depending on which variables are being preserved and where are they declared in the code, no tools are provided to manipulate it. The host tools will simply collect the information and send it back to the mote to be restored.

If needed, the post-processor tool can be modified to generate code able to access this memory. This code would be of limited use anyway, because in case of native structs being stored, the tool only knows their size and not their internal format, so without any major re-factory, the code would only be able to produce a raw binary field.

Since this memory area can be larger than the MTU, to dump it, additional information is needed to be able to restore it at a later time. The message format used, shown in Figure 4.3, uses packets as large as the MTU, and describes chunks of the memory area containing the initial state.

These messages need to be stored and can be send back from the sink at any time.

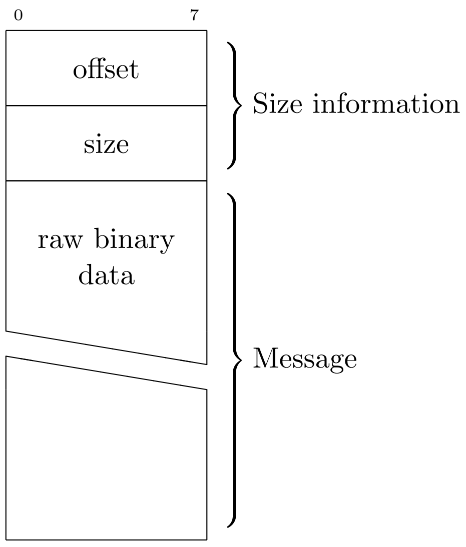


Figure 4.3: Packet format for the dump of the initial state. The offset and size fields are used to copy the message in the right part of the memory area that contains the initial state.

4.2 Sink

The TinyOS serial protocol[14] was implemented on the sink, using the Python language. The *mig* tool provided by TinyOS to implement mote-PC communication can generate Python code amongst the other targets, but the Python generated code doesn't have as much features as the other targets, so an implementation from scratch was preferred.

4.2.1 Enums

When dealing with constant values it is useful to be able to define enums to group related constants together. The Enum class was added in Python 3.4[4]. Unfortunately the implementation is made to run on Python 2.7 so the support is not available. An *Enum* class was implemented to have similar functionalities.

The enum must extend the enum class and directly define the constants in the body of the class. The abstract class provides a function to get the name of the constant given the value.

4.2.2 Communication

Frames

A *Frame* object has been implemented to map in software the concept of frames in the transmission. The frames use two special bytes to delimit the frames and to escape if those bytes appear in the payload. So upon receiving the frame is first decoded using the format shown in Figure 4.4 and then the CRC is checked to make sure it is correct. An exception is raised if the check fails

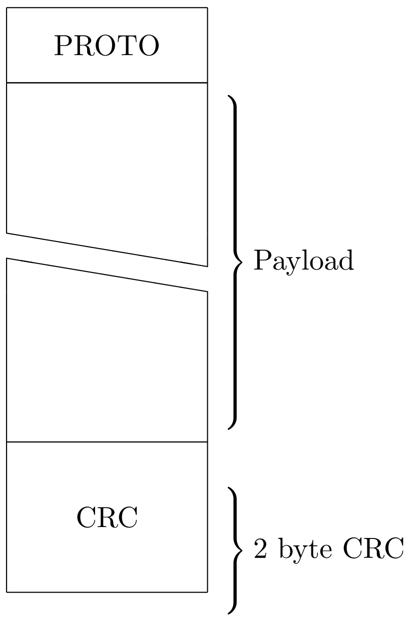


Figure 4.4: TinyOS serial frame after having been decoded.

The class can also do the reverse and generate valid frames given a payload.

Constants for the various protocols are defined. One of such protocols is used for the ACK messages, the frame payload will contain the sequence number of the packet being acknowledged. The frame class doesn't handle ACKs itself, because it has no knowledge of sequence numbers as well.

Packet

Frames that do not contain ACKs, contain packets. Another class has been implemented to deal with this layer.

A packet structure (shown in figure 4.5) contains many useless fields (such as source and destination) because it is the same structure that TinyOS uses for the radio stack, where the medium is not point to point.

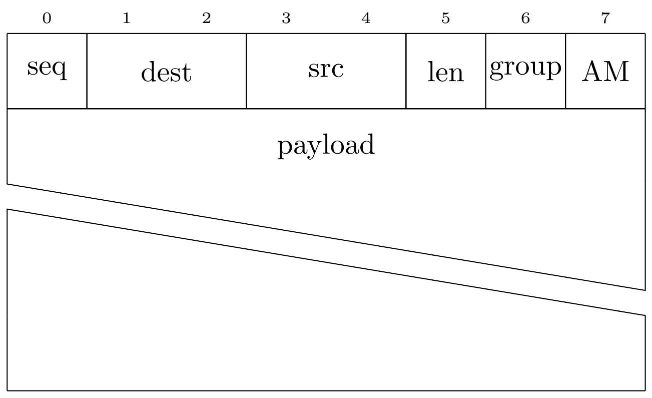


Figure 4.5: TinyOS packet format.

In the implementation, the class ignores the sequence number, because the same packet could be transmitted several times but every of these times it would need to have a different sequence number to be accepted.

The *AM* field is quite important. It is used by TinyOS to route the messages to the various modules, so the field determines which function of the mote will receive the message. An enum is defined with the values used in the AM field by the logger module. It is important to note that should these values collide with those used by some specific application on the mote and hence be changed, the sink code should be adapted as well to reflect the change.

Handler

The *Handler* module takes care of the connection with one mote. It can send and receive messages.

When sending the handler can accept a payload and an AM or directly a packet. To make things more straightforward the handler can accept lists of packets and will send them all.

The handler keeps track of the sequence number and will construct the correct data-stream to send to the node. The send method is implemented in a blocking fashion and in case of a missing ACK the handler will do a few attempts of retransmission before giving up and raising an exception.

Reading data from an handler is a bit more complex. An handler has a blocking file descriptor pointing to its mote, and exposes a read() method. The application must select() or poll() on all the file descriptors from the handlers to know when it is possible to call their read() method.

Handlers don't make use of locks so if they are used in a threaded application instead, the application should make sure that no concurrent reads or writes are called on the same handler.

Handlers have a *packet* attribute, that lists all the packets received by the handler. This is done because a read can often get partial frames that cannot be parsed by the upper layers, so partial data is kept in a buffer and decoded into packets only when the frames are complete.

4.2.3 Event handling

A generic *nx_struct* class to deal with packed structs was written, it accepts the binary data and a tuple describing the struct and initializes the object creating the necessary fields. The class is also able to pack the struct again, after modifications; and to print a nice representation of itself.

A class to represent events *nx_event*, subclasses nx_struct, and its main purpose is to offer a function to convert the event into a list of packets that the nodes could understand.

Then more subclasses are provided for each event type, and they are associated with the *wrapper type* field shown in Fig 3.4. Those subclasses are very minimal: they just provide an indication of the internal format used by the wrapper to store the log.

As a result of this implementation, letting the sink be able to work with a new type of wrapper is not very hard and only requires indicating the internal binary format and associating the class with the constant number used to indicate that wrapper.

4.2.4 Sink tool

A general sink tool is provided; it is able to connect to several motes using sockets, and can instruct them to start/stop the logging, gather the log dump and load it, to tell the nodes to replay it.

Parallel map

Most of the operations that need to be performed on all the nodes are implemented using *map* on the list of the connections, to have all the nodes do the same operation. For long operations (namely dumping a log and loading it on the node) it is convenient to have a parallel map that can execute all the operations in parallel and still return the same result as the sequential map. In this way, even if the nodes are slow when compared to a common PC, the time of the operation can be decreased because the time waiting for a node to reply does not block the communication with the other nodes.

Python provides a parallel map implementation, but unfortunately that implementation does not allow file descriptors to be transfered from the parent process to the subprocesses. So a new parallel map was implemented in the python code. It runs several child processes using the POSIX fork(), uses pipes to get their return value and serializes/de-serializes data to the pipes to make sure python objects are correctly returned.

The speedup obtained by this improvement is quite impressive, (see section 5.1.1) and allows the sink tool to be more interactive.

4.2.5 Event sorting

Conceptually every event is a node in a dependency graph, that can depend on multiple events. However a common pattern exists in having a group of local events intermixed by events that cross the node boundaries. As a result for the implementation it was chosen the approach of group events into chunks, with the knowledge that every event in the chunk depends solely by its predecessor, and a chunk can depend on multiple other chunks.

To implement this, the events in every node are grouped using send/receive events as separator, every group is a chunk and will depend on the previous chunk. During this chunking pass, two lists pointing to send and receive events are kept and updated to speed up the 2nd pass.

On the 2nd pass, the list containing the receive events is scanned, and for every item in it, the send list is scanned to find the matching event. If the event is found then is added as predecessor of the receive event.

During this pass the predecessors from the same node are scanned to search a receive event, which, if found, is added as dependency to the send event.

Chapter 5

Evaluation

5.1 Sink benchmarks

5.1.1 Parallel or serialized dump and load

The sink can need to manage networks with a certain amount of nodes. In the original implementation when dumping the log of events, the 1st node would be queried and the sink would have waited until the dump was complete, to proceed with the 2nd node and so on. The same implementation was used to load logs on the nodes. The improved version can communicate with each node in parallel.

To perform this test the BlinkToRadio test application was used, and the events were collected for 10s, on a network composed by 5 nodes. The results are shown in Figure 5.1.

5.2 Node benchmarks

In this section we perform and show the results of some performance tests on the nodes, to show the overhead introduced by the logging in the system.

All the tests were performed compiling the application with *telosb* target, and then loading the binary file in Cooja, as a *Sky mote.*

MSPsim[10] was used in conjunction with Cooja for the tests. The simulator offers the possibility of counting the cycles of the mote's CPU, and Cooja has the capability of pausing the simulation based on various events. MSPsim's cycle counter was modified to output the current count on the standard output when the update button is pressed or when the simulation is started/paused. This allows for shell tools such as *grep* or *tr* to be used to filter and collect the relevant data for further analysis.

In general counting CPU cycles has proven to be easier and more precise than counting milliseconds, allowing us to ignore the effects of other processes running on the host machine, so they were preferred in most of the

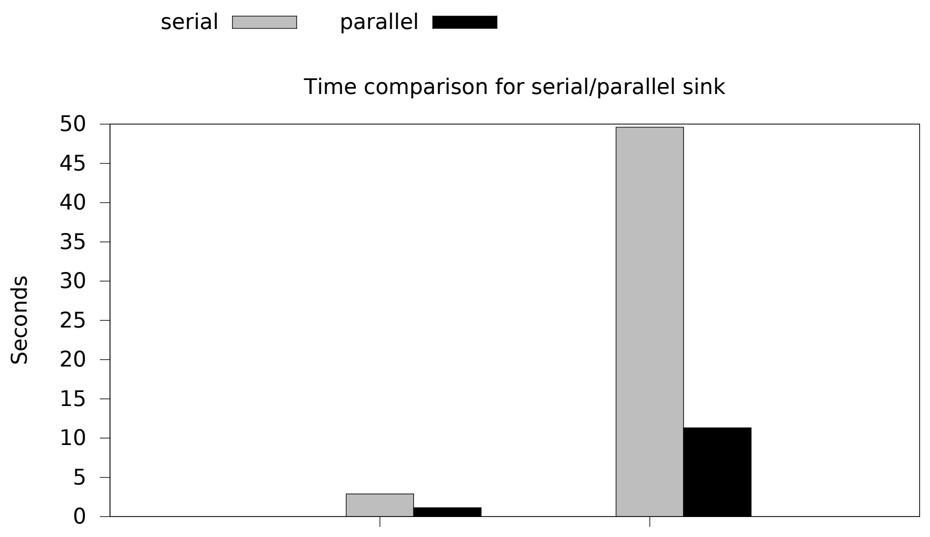


Figure 5.1: Time needed respectively to dump the event log from the WSN and to load it on the nodes.

tests.

5.2.1 Aggressive blinking

To perform this test, an application that blinks the LEDs of the node was written.

The application uses one timer and has one task that reads the value from the timer, sets the LEDs on that value and posts itself. The task is posted in the boot section, and never stops to post itself.

The timer is just used to read the time-stamp, but not to trigger events.

Cooja was configured to interrupt the simulation on LEDs events, and the amount of cycles performed by the CPU until that moment were counted and stored in a file.

A shell script was used to unpause the simulation at regular intervals, which will automatically stop again almost immediately, on the next LED event.

Three versions of the application were tested.

logging in this version the application uses the log for the timer and the log is enabled at boot, before the task is posted.

normal in this version the timer is wrapped by the logger component, which is not used to log the messages.

nolog in this version, there is no logging

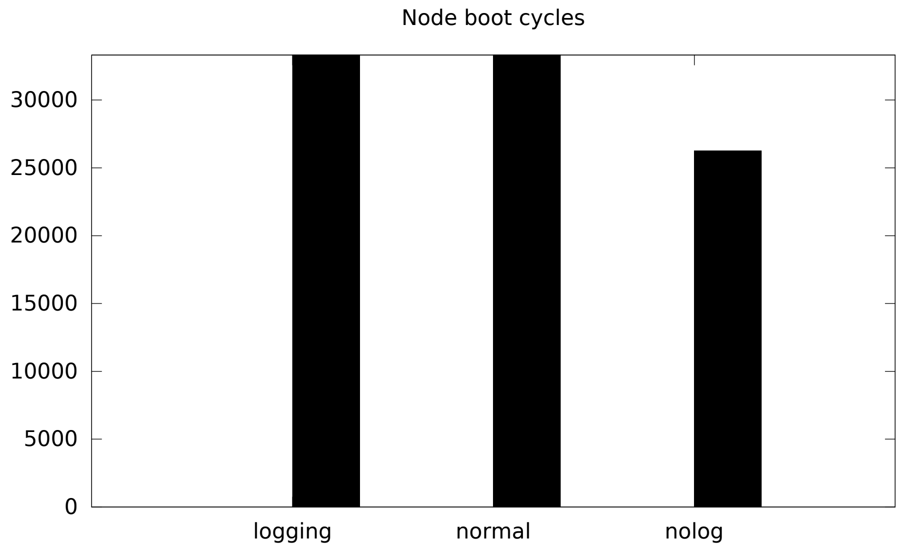


Figure 5.2: Cycles needed by the CPU to set a led for the first time

This application is very CPU intensive. It never stops waiting for events, it always has a task pending.

As shown in Figure 5.2, the initialization time for a node is a higher if the logging facility is included in the program, regardless of it being active or not. The compiler can't aggressively remove the code because the logger listens to the serial port and can start logging if such a command is received, so this shows that even if they seem not to be used, the logging capabilities are loaded anyway.

In Figure 5.3 is shown how many CPU cycles occur between one LED event and the next. And it appears evident that the logging has a certain cost on the CPU. However common WSN applications aren't CPU bound and this test was performed specifically to show the drawbacks. The values shown are an average calculated on the same program running on a long period of time. The period used in the case of logging was shorter than the other cases, to make sure that all the events could fit in the log.

5.2.2 More wrappers

A typical application will need to log more modules than a single timer. This test focuses on exposing the effects of the increasing number of wrappers within an application.

A blink application was written, to use from one to four timers, with and without the logging modules compiled in. Then after all the timers are instructed to fire periodically at different intervals, a task is scheduled to increment a counter and set the LEDs with the value of the counter. The

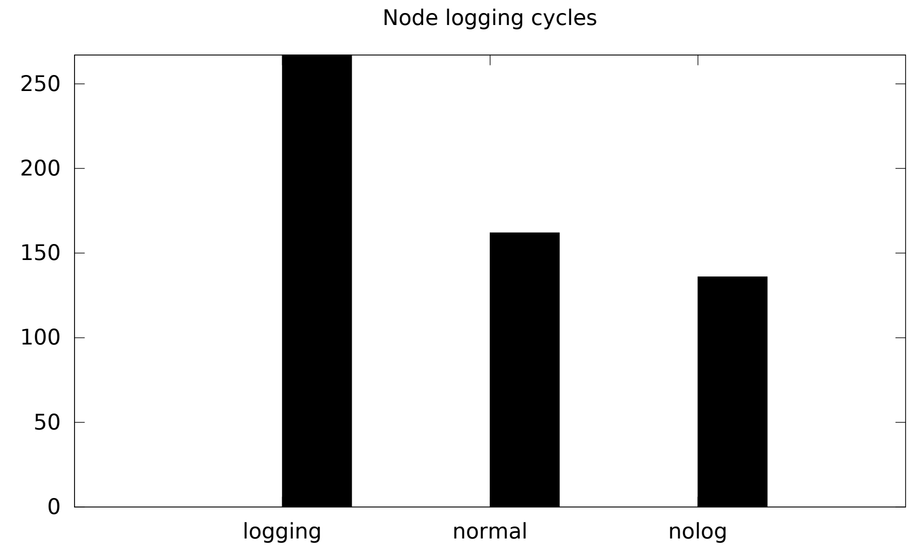


Figure 5.3: Average number of CPU cycles between successive led operations

task is also scheduled when any of the timers fires, but this is not important for this test since just booting times were taken into account.

Figure 5.4 shows that adding more timers does not increase the time required to start the program, but having more wrappers around those timers seem to generate a linear overhead instead.

The overhead added by each wrapper is quite small when compared with the general overhead of using the logging, which comes in whatever number of wrappers is used.

The specific values are likely to be different when different type of wrappers are being used, but the test clearly shows that every wrapper will impose a cost at boot time.

5.2.3 Fast radio

In this test the message wrapper is tested. The application is a variation of the example *RadioCountToLeds* from TinyOS. All the nodes have the same program, but the node with ID 1 sends messages and the other receive those messages and set their leds accordingly. In the original example, messages are sent periodically using a timer, but in this modified version a task is used to send the messages, and the task is scheduled every time a *sendDone* event is raised, to send messages as fast as possible.

It was observed that one radio message is sent every two radio medium events signaled by Cooja, so half of the partial cycle counts had to be discarded.

The results of the test are shown in Table 5.5 and in Figure 5.6. As it is

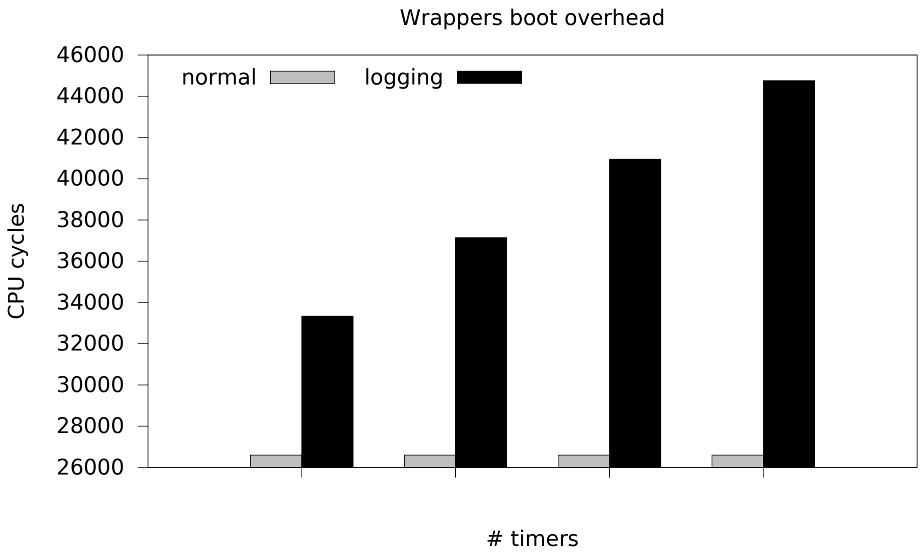


Figure 5.4: Cycles needed to turn on one LED, using 1,2,3 and 4 timers, wrapped or without logging.

	normal	logger
boot ms	4482	4493
boot cycles	83947	116488
messages ms	4605	4627
messages cycles	604388	681954

Figure 5.5: Cycles needed by the two versions to boot and to send 35 messages.

possible to see, for more complex applications the overhead is quite minimal both because the application in itself is bigger and generates more load, and because the impact on boot time is reduced by the fact that many modules are used anyway by the application itself and don't need to be loaded just for the logger.

When logging, the messages needs to be copied entirely into the log, so this imposes a limit on the maximum throughput achievable when logging is in use.

The results of this test are aggregate results over 17 messages sent, so the difference between the normal version and the logging version are emphasized.

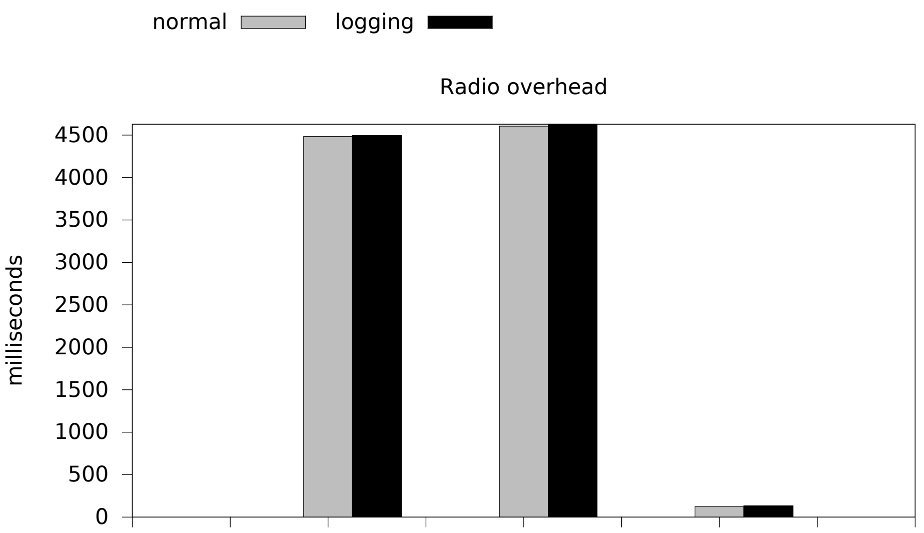


Figure 5.6: Comparison in milliseconds to: boot, boot and send 35 messages, send 35 messages.

5.2.4 Cycles per radio message

In this test the same application used in the *Fast radio* test was used, but instead of gathering data on the time needed, the focus is shifted on the extra effort needed by the CPU to log a single radio message.

The simulation was configured to have just one node and to stop when events occur on the radio medium rather than on the LEDs. The cycles counter of the CPU was used to know the needed time between one message and the other. The initial counter results were removed to avoid taking into account the boot time, already considered in other tests.

From the sequences of values, new sequences were generating by calculating the difference of every value with the previous one, to know how many cycles elapsed between one message and the next. Then the average was calculated on those values.

The number of transmissions done with the logging enabled was kept to a minimum value because the logging automatically stops if the log space is full and that would have yielded incorrect results on the average. So the number of transmissions were enough so that all the messages could fit in the log.

Figure 5.7 shows how the logger module brings no substantial overhead when disabled, and when the overhead on the single message is comparatively very little. Table 5.8 shows the numerical averages between one packet and the next in the different cases. It is possible to see that a disabled message logger only causes a slight increase on the average.

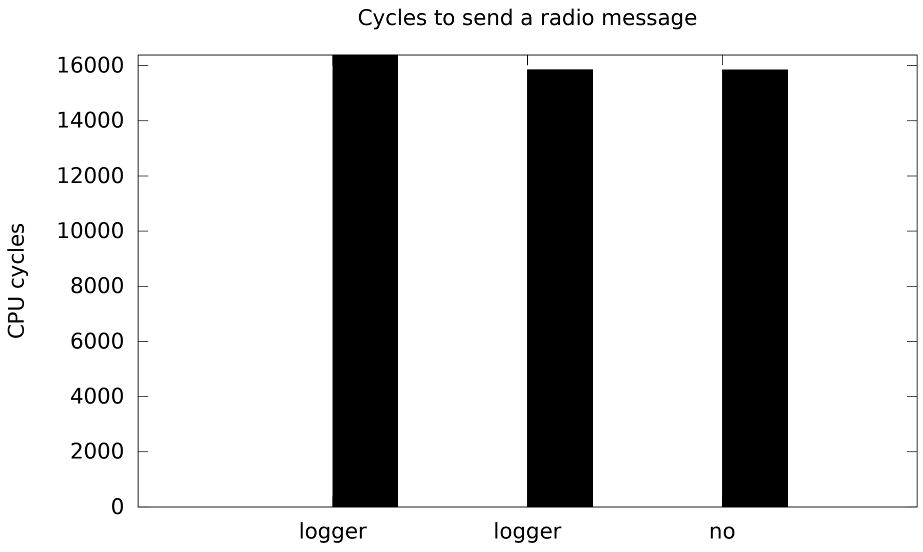


Figure 5.7: Comparison of the cycles needed by the application to send one message, in a modified BlinkToRadio application.

logger enabled	logger disabled	no logger
16382	15848	15845

Figure 5.8: Average amount of cycles needed to send one message in a modified BlinkToRadio application.

5.2.5 Memory usage

Using the logging module inside an application will have a cost in terms of memory used.

The effects are due to the use of the serial stack, and by the need of every wrapper to have a memory area to use for the log.

Wrappers have a default log size (defined inside an header file) so the larger effect of adding a new wrapper in the application is that an array of size n, with elements large enough for the event data structure will be placed on memory.

Developers can (and should) tweak the default queue sizes according to the specific need of the application, keeping in mind that the event structure for different wrappers have different sizes.

In addition to that, a wrapper usually need to keep some more internal state, that doesn't depend on the size of the log.

In table 5.9 the memory usage is reported, the wrappers are used just for the timers, and each of them has a size of 40 entries. As it is possible to see, adding a wrapper requires an extra 346 B. The struct containing a

# timers		ROM	RAM
1 timer	normal	2490 B	38 B
	logging	7650 B	688 B
2 timers	normal	2644 B	48 B
	logging	8556 B	1034 B
3 timers	normal	2698 B	58 B
	logging	9318 B	1380 B
4 timers	normal	2714 B	68 B
	logging	10012 B	1726 B

Figure 5.9: Memory usage as reported by the compiler for the many timers application compiled with different amount of timers and adding/removing log support

	ROM	RAM
message wrapper	18454 B	3654 B
normal message	16932 B	1350 B

Figure 5.10: Memory usage increase to add a message wrapper over an already logged application.

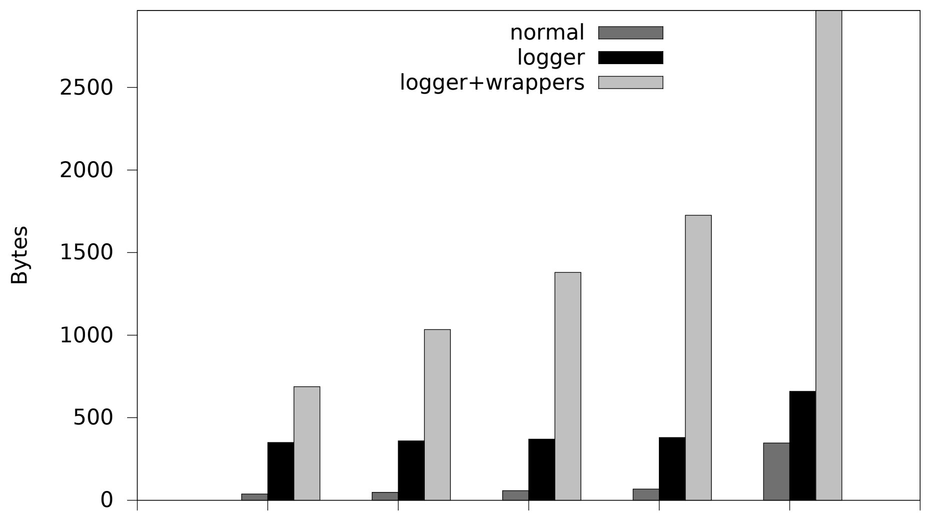


Figure 5.11: RAM usage as reported by the compiler for: blink application with 1,2,3,4 timers and for RadioCountToLeds. Every wrapper can store 40 events.

timer event has fields for 7 B, plus padding for 40 items it gives 320 B; that account for most of the memory overhead caused by an extra timer wrapper. 10 B are actually being caused by having an extra timer. So this shows that the impact on memory is strictly related by the size of the log.

Another interesting observation from the data is that also the size on the ROM is increased, despite the wrappers are identical and share the same code. This is caused by the NesC implementation of generic modules that duplicates the code for every instance.

In table 5.10 is shown how large is the overhead to add a message wrapper over an application that uses the radio and already has logging in place. Normally a *message_t* struct takes 52B, the wrapper adds 4B to that for extra information, so with a log of 40 entries, this requires 2240 B, which is not a small amount on a node. This poses a limit on how many messages can be logged in a real application, considering that more structures will be needed by other modules and stack space is also needed to operate correctly.

Figure5.11 shows how on non-trivial applications the overhead of the logger instrumentation is not this big, while the space necessary for the wrapper can be substantial, if the size for the log is not modified.

5.3 Split-Phase Faults

Most long operations in TinyOS are implemented as split-phase, when for example a command to initialize a device is sent from the application to the lower layer, and then an event is sent back to signal that the initialization is complete.

The authors of [29] use LEACH as a case study to explain how their tracer helped in finding an implementation error.

LEACH is a TDMA-based dynamic clustering protocol. In the example the problem was caused on the cluster head by a timer event trying to send a debug message while another component was sending the informations about the cluster to a node requesting access. The bug was caused by the fact that in the timer event, the type of the message was set, although the send itself would fail, the message itself was sent with a different type (because there is only one buffer for the messages, and the original content had been modified by an interleaving event) and acknowledged and ignored by the receiver, which had no function associated with that type of message.

With our implementation the log on the non-head node would show no activity, since the wrapper is placed at high level and the message would be discarded before reaching it, and the head node would show an interleaving of a timer between send and sendDone, and also carry enough information to show that the buffer's content was altered.

Chapter 6

Conclusion

6.1 Limitations and future work

A number of possible future improvements are explored in this section. They are meant to highlight the limitations of the present work and suggest ways for further improvement.

As we have shown in the evaluation section, the memory expensive message wrapper would need some improvements to be able to store more messages. For example, if no bugs occur the content of the message buffer after it has been sent is identical, so there is no need to store it twice. Also, the packet could present some redundancy and remain largely identical, so storing a delta rather than an entire copy could improve the memory usage at the expense of an increased CPU usage. The trade-off would need to be evaluated carefully. A number of other specific wrappers can be devised and implemented expanding the current architecture. They could also be used to wrap around the lower level components, closer to the hardware layer instead, if needed. They would have to overcome the current design limitation forcing wrappers to ignore events while in replay mode.

The current implementation does not take scheduling into account but assumes that in the interval between the command to enter replay mode, and the first command to replay an event, enough time will pass to exhaust all the queued tasks, and the replay will proceed undisturbed without influences. This might not always be the case, and including the scheduler into the logging model would help in this direction.

6.2 Conclusion

In this thesis we introduced a tool to log events from the nodes of a WSNs and to replay them in a controlled environment to facilitate debugging.

We have shown how we exploit the concept of wiring in nesC to easily introduce wrappers around the modules, that allow us to intercept and log all the interaction between two modules. The same design was used to perform the reverse operation and simulate an interaction between modules.

We also illustrate how practically the system is implemented and made to work, and the methods used to extend the concept of logging and replaying from a single node to an entire network.

The investigated results show that the used approach has a tolerable impact on performances. This, with the possibility of easily integrating our system within pre-existing networks and distributed applications, makes the use of the present work a viable solution to investigate and solve bugs in WSNs applications.

Bibliography

[1] Information technology – Programming languages – C++, 2011.

[2] The TinyOS 2.x Working Group. Tinyos 2.0. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 320–320, New York, NY, USA, 2005. ACM. ISBN 1-59593-054-X. doi: 10.1145/1098918.1098985. URL `http://doi.acm.org/10.1145/1098918.1098985`.

[3] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7641-8. URL `http://dl.acm.org/citation.cfm?id=243846.243857`.

[4] Ethan Furman Barry Warsaw, Eli Bendersky. Adding an enum type to the python standard library, 2013. URL `http://www.python.org/dev/peps/pep-0435/#status-of-discussions`.

[5] Ningxu Cai and Robert W Brennan. Distributed sensing and control architecture for automotive factory automation. In *Holonic and Multi-agent Systems for Manufacturing*, pages 165–174. Springer, 2009.

[6] Elaine Cheong, Judy Liebman, Jie Liu, and Feng Zhao. Tinygals: a programming model for event-driven embedded systems. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, pages 698–704, New York, NY, USA, 2003. ACM. ISBN 1-58113-624-2. doi: 10.1145/952532.952668. URL `http://doi.acm.org/10.1145/952532.952668`.

[7] Darren Dao, Jeannie Albrecht, Charles Killian, and Amin Vahdat. Live debugging of distributed systems. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, 2009.

[8] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors.

In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[9] Joakim Eriksson. Detailed simulation of heterogeneous wireless sensor networks, 2009.

[10] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik österlind, and Thiemo Voigt. Poster abstract: Mspsim an extensible simulator for msp430-equipped sensor boards.

[11] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, 38(5), may 2003.

[12] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, 2006.

[13] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: global comprehension for distributed replay. In *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, NSDI'07, 2007.

[14] Ben Greenstein and Philip Levis. TinyOS Extension Proposal (TEP) 113: Serial Communication. `http://www.tinyos.net/tinyos-2.x/doc/html/tep113.html`, 2006.

[15] Douglas Herbert, Vinaitheerthan Sundaram, Yung-Hsiang Lu, Saurabh Bagchi, and Zhiyuan Li. Adaptive correctness monitoring for wireless sensor networks using hierarchical distributed run-time invariant checking. *ACM Trans. Auton. Adapt. Syst.*, 2(3), September 2007. ISSN 1556-4665. doi: 10.1145/1278460.1278462. URL `http://doi.acm.org/10.1145/1278460.1278462`.

[16] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev.*, 34(5), nov 2000.

[17] Mohammad Maifi Hasan Khan, Hieu Khac Le, Hossein Ahmadi, Tarek F. Abdelzaher, and Jiawei Han. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 99–112, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-990-6. doi: 10.1145/1460412.1460423. URL `http://doi.acm.org/10.1145/1460412.1460423`.

[18] Veljko Krunic, Eric Trumpler, and Richard Han. Nodemd: diagnosing node-level faults in remote wireless sensor systems. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, MobiSys '07, pages 43–56, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-614-1. doi: 10.1145/1247660.1247669. URL `http://doi.acm.org/10.1145/1247660.1247669`.

[19] James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 259–269, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5. doi: 10.1145/301618.301678. URL `http://doi.acm.org/10.1145/301618.301678`.

[20] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Optimal clock synchronization in networks. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 225–238, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-519-2. doi: 10.1145/1644038.1644061. URL `http://doi.acm.org/10.1145/1644038.1644061`.

[21] Philip Levis and David Culler. Mate: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(5):85–95, 2002. ISSN 0163-5980. doi: 10.1145/635508.605407. URL `http://doi.acm.org/10.1145/635508.605407`.

[22] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3s: debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, 2008.

[23] Liqian Luo, Tarek F. Abdelzaher, Tian He, and John A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Trans. Embed. Comput. Syst.*, 5(3):543–576, aug 2006. ISSN 1539-9087. doi: 10.1145/1165780.1165782. URL `http://doi.acm.org/10.1145/1165780.1165782`.

[24] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, mar 2005. ISSN 0362-5915. doi: 10.1145/1061318.1061322. URL `http://doi.acm.org/10.1145/1061318.1061322`.

[25] Yang Peng, Richard Lahusen, Behrooz Shirazi, and WenZhan Song. Design of smart sensing component for volcano monitoring. In *Intelligent Environments, 2008 IET 4th International Conference on*, pages 1–7. IET, 2008.

[26] Zoltán Porkoláb, József Mihalicza, and Ádám Sipos. Debugging c++ template metaprograms. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 255–264, New York, NY, USA, 2006. ACM. ISBN 1-59593-237-2. doi: 10.1145/1173706.1173746. URL `http://doi.acm.org/10.1145/1173706.1173746`.

[27] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 255–267, New York, NY, USA, 2005. ACM. ISBN 1-59593-054-X. doi: 10.1145/1098918.1098946. URL `http://doi.acm.org/10.1145/1098918.1098946`.

[28] Tamim Sookoor, Timothy Hnat, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. Macrodebugging: global views of distributed program execution. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, 2009.

[29] Vinaitheerthan Sundaram, Patrick Eugster, and Xiangyu Zhang. Efficient diagnostic tracing for wireless sensor networks. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, 2010.

[30] Igor Talzi, Andreas Hasler, Stephan Gruber, and Christian Tschudin. Permasense: investigating permafrost with a wsn in the swiss alps. In *Proceedings of the 4th workshop on Embedded networked sensors*, EmNets '07, pages 8–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-694-3. doi: 10.1145/1278972.1278974. URL `http://doi.acm.org/10.1145/1278972.1278974`.

[31] Salvatore Tomaselli. cooja-dbus. `https://github.com/ltworf/cooja-dbus`, 2013.

[32] Koen Vervloesem. Control your linux desktop with d-bus. *Linux J.*, 2010(199), nov 2010. ISSN 1075-3583. URL `http://dl.acm.org/citation.cfm?id=1883556.1883559`.

[33] Zhao, Y.J.; Govindan, R. ; Estrin, D. Residual energy scan for monitoring sensor networks, 2002.

#

Appendices

Appendix A

Add logging

In this chapter we show how the logging feature can be added into an existing application. Some code is shown in *diff* format, to highlight the needed changes into an already existing application.

A.1 Initial state

Variables part of the initial state need to be module-global, they are indicated as part of the initial state by this function call:

```
  bool variable;

  task void do_task() {
+     LOG_STATE_VAR(&variable,sizeof(variable));
  }
```

The function call is just a placeholder, it will not perform any operation at runtime, but being it a function, that code needs to be placed inside a function. It is not important inside which function it is located.

A.2 Timer wrapper

The app has this line:

```
uses interface Timer<TMilli> as MilliTimer;
```

In the wiring, that interface is wired to a *TimerMilliC*. We declare a timer wrapper with the same precision and wire it to both the timer and the application as shown in the following diff-style configuration.

```
  components new TimerMilliC();

- App.MilliTimer -> TimerMilliC;

+ components new TimerLoggerC(TMilli) as TL;
+ TL.TimerIn -> TimerMilliC;
+ App.MilliTimer -> TL;
```

A.3 Radio wrapper

The application declares uses as follows:

```
uses {
    interface Packet;
    interface Receive;
    interface AMSend;
}
```

And the configuration is changed to add the wrapper:

```
  components new AMSenderC(AM_TYPE);
  components new AMReceiverC(AM_TYPE);

+ components new MessageLoggerC() as ML;

+ ML.AMSendIn -> AMSenderC;
+ ML.ReceiveIn -> AMReceiverC;
+ ML.Packet -> AMSenderC;

- App.Receive -> AMReceiverC;
- App.AMSend -> AMSenderC;
+ App.Receive -> ML.ReceiveOut;
+ App.AMSend -> ML.AMSendOut;

  App.Packet -> AMSenderC;
```