This paper was presented as part of the main technical program at IEEE INFOCOM 2011

# TinyLTS: Efficient Network-Wide Logging and Tracing System for TinyOS

Robert Sauter[1], Olga Saukh[2], Oliver Frietsch[3] and Pedro José Marrón[1]
[1]Universität Duisburg-Essen and Fraunhofer IAIS, Germany
[2]ETH Zürich, Switzerland
[3]Rheinische Friedrich-Wilhelms-Universität Bonn, Germany

***Abstract*—Logging and tracing are important methods to gain insight into the behavior of sensor network applications. Existing generic solutions are often limited to nodes with a direct serial connection and do not provide the required efficiency for network-wide logging. Instead, this is often realized by application-specific subsystems developed for custom logging statements. In this paper, we present TinyLTS – a generic and efficient Logging and Tracing System for TinyOS. TinyLTS consists of a compiler extension that separates dynamic from static information at compile time, a declarative solution for inserting logging statements, an extensible framework for flexible storing and transmitting of logging data and a frontend for recombining dynamic and static information. Our system provides concise yet expressive programming abstractions for the developer combined with efficiency comparable to custom solutions.**

## I. Introduction

"From a system implementation standpoint, the most important decision we made in CTP Noe's design was including a detailed logging layer.". This quotation from the SenSys'09 Collection Tree Protocol paper [1] is a good indicator for the importance of providing the seemingly simple functionality of logging. While significant research in traditional systems and in sensor networks has been dedicated to create intricate tools to help debugging and evaluating systems, logging and tracing are still among the primary means for developers to get insight into the behavior of single nodes and the network as a whole.

The CTP implementation is also a good indicator for what is missing in sensor networks. Using simple text based functions like `printf` or a variety of logging frameworks are a viable solution for traditional systems. However, this approach is not feasible for sensor network deployments due to the large overhead when transmitting the logging information over the network and still incurs a notable overhead when used with flash memory or a serial port. Instead, often custom logging layers are implemented where the developers create dedicated message formats for use at runtime and interpreters on the PC side to transform the data into human readable information.

We propose an alternative approach that combines the simplicity and convenience of printf with the runtime efficiency of custom logging implementations. In this paper, we present TinyLTS: a generic logging framework for TinyOS. We provide a compiler extension that allows using logging statements comparable to logging frameworks. The compiler extension separates static (e.g. the format string and the types of logged variables) from dynamic information and generates the necessary code to transmit only the dynamic data at runtime. We provide a frontend for the PC that recombines the logging data collected at runtime with the static data extracted during compilation to generate textual logging messages based on the specification of the developer or as a basis for custom analysis and visualization tools. Instead of developing a dedicated protocol for transmitting logging data, we opt for reusing the transmission protocol of the application in order to reduce RAM, program memory and network overhead (e.g. incurred by additional beacons). We provide an implementation using CTP and also investigate how small modifications to the protocol can improve the suitability for transmitting application messages and logging data over the same protocol. Our framework is easily extensible to support other routing protocols since most of the work is performed generically by the compiler extension and the frontend. Additionally, we provide implementations for logging to flash memory or over the serial port. In addition to the standard formatting options of printf, we introduce additional formatting directives and operators that allow for example logging variables of composite types such as structs and support the common use case of tracing functions with a dedicated construct. These additions are targeted at increasing the maintainability of the code by decoupling logging code from application changes and allow developers to write more concise logging statements. Additionally, we provide a declarative instrumentation language that allows inserting logging statements from outside of the application source code.

Therefore, our contribution in this paper is fourfold. We develop a logging and tracing framework that provides a compiler extension for runtime efficiency, logging implementations for common use cases and an extensible frontend described in Section IV. We enhance logging and tracing statements to enhance maintainability and usability discussed in Section II. We introduce a concise and expressive language for declarative instrumentation in Section III and we investigate optimizations of routing protocols to improve the suitability for transporting both application packets and logging packets in Section V. Our paper is completed by the evaluation in Section VI followed by a discussion of related work.

## II. Design and Usage

The goal of TinyLTS is to provide developers an easy-to-use framework efficient enough to use during deployments in

978-1-4244-9921-2/11/$26.00 ©2011 IEEE 

non-tethered environments. The following paragraphs describe the primary goals and approaches of our system.

### A. Familiarity

Ease-of-use is the foremost goal of TinyLTS and, thus, the primary mechanism is similar to printf in C programming. By additionally including a severity, the syntax closely resembles established logging solutions provided for traditional systems like log4j and ACE [2].

```
logNow((severity, formatString, ...));
```

Our system supports the common format and type specifier in the formatString, which can be followed by an arbitrary number of arguments that will be included in the logging message. Moreover, TinyLTS supports additional formatting directives and variable types as described below.

### B. Maintainability

Our approach provides several important advantages in the area of maintainability. First, the logging functionality can be completely disabled at compile time with a simple compiler switch. Second, when combined with a configuration framework, logging functionality can also be disabled at runtime. Third and most importantly, we provide a new operator and new as well as enhanced formatting directives to support reflection, i.e. name and type information, and arbitrary structs:

- `^` The new unary operator ^ provides 'complete' dereferencing of the subsequent expression, i.e. it is replaced by the correct number of $\star$ operators to obtain a non-pointer type.
- `%s` The formatting directive `%s` is enhanced to support arbitrary data types, therefore, freeing the programmer from having to explicitly specify the type of the corresponding positional argument. Additionally, this directive provides support for structs: when the data type of the corresponding parameter is a struct, the output equals the use of the `%s` directive recursively on any field. For each field the operator ^ is implicitly applied and, thus, the output contains the actual contents of the variable in human-readable form.
- `%r` The new formatting directive `%r` ('r' for reflection) builds upon the enhanced directive `%s` but extends the value of the corresponding variable by its name and data type. Additionally, the fields of structs are recursively described in the same way.
- `%R` The new formatting directive `%R` builds on the previous one. However, instead of printing the corresponding positional parameter, the directive instructs to use all the arguments of the function enclosing the `logNow()` command. This allows a convenient declaration of tracing without having to adapt to changes in the function declaration.

These extensions fit naturally in the usage patterns for the printf idiom without requiring developers to learn any new complicated constructs. More importantly these extensions allow significantly increasing the independence of logging statements from the rest of the source code: changes in function names, function arguments and variable types including struct members are automatically taken into account without requiring intervention of the developer.

### C. Brevity

We achieve the goal brevity to a large extent by the functionality discussed in the previous paragraphs, which automatically derives a large amount of information that has to be manually specified in pure C solutions. Moreover, we also declare a nesC attribute [3] which allows to enable the tracing of a function, i.e. logging the call and the arguments, in a compact way.

```
@trace((severity, formatString, ...))
@trace() // =trace((SEVERITY_DEBUG, "%R"))
```

The canonical version shown first provides exactly the same functionality as the `logNow()` command and is internally expanded to a call to `logNow()` at the beginning of the function. The second form is the tersest possibility to enable tracing of a function including all its arguments.

Additionally, TinyLTS automatically stores information about a logging statement's location in the source code. This includes the name of the nesC-module, the function name, the source file and the line number for all logging and tracing commands which removes the need for the use of C macros like `__LINE__`. This information does not incur any runtime overhead.

Optionally, at runtime the framework automatically records the current time of the invocation of the logging function (global time requires time synchronization) and/or a sequence number for logging invocations. However, since supporting this functionality incurs additional overhead at runtime, this functionality can be enabled and disabled for both sequence numbers and timestamps individually. Additionally, the number of bytes for storing this information and for the timestamp also the granularity can be specified to give fine-grained control over the overhead and make it possible to easily adapt to the requirements of the application.

### D. Runtime Efficiency

We achieve runtime efficiency primarily by distinguishing between static information known at compile-time and dynamic information that is only known at runtime. Only dynamic information is transmitted during runtime which allows developers to use informative texts without impact on the program. Additionally, compared to other solutions that provide a printf-like method, our solution reduces the demands for RAM, program memory and computation time on the node because no parsing of the format string is necessary. This also minimizes the impact on the behavior of the application and reduces the probability for causing a Heisenbug by adding logging statements. We discuss the details for achieving runtime efficiency in the next sections.

### E. Extensibility

While we are providing a complete system including logging implementations for serial connections, recording in flash

```
1  SineSensorC Read read (SEVERITY_INFO, "%r", counter)
2  MVizC {
3     .* startDone () // for all interfaces
4     report_.* () // for all report_ functions
5     Timer {
6        fired ("busy: %s, reading: %s", busy, reading)
7     }
8  }
9  #ifdef LOG_LEDS
10    #include "logleds.lts" /* log LED changes */
11 #endif
```

Fig. 1. TinyLTS Declarative Example

memory, a collection routing based version and an optimized protocol for network-wide logging and tracing, we designed our framework as an extensible solution. It is easy to implement alternative mechanisms, i.e., the use of alternative network protocols or even the use of special hardware characteristics (e.g. multiple radios). Although minimizing logging information may be not as crucial for every logging implementation, these implementations can still benefit from the ease-of-use provided by the functionalities described above. Additionally, the implementation itself is significantly simplified even compared to providing only printf-like functionality.

## III. Declarative Instrumentation

Besides the logging and tracing statements, we also provide the possibility to declaratively instrument source code with logging functionality. Our approach is similar to Aspect Oriented Programming [4]: logging statements specified external from the source code are woven into the application source. For the design of the declarative language we followed the same goals as for the use of the logging statements: brevity, familiarity for the programmer and ease-of-use. We show a sample script in Fig. 1 targeted at instrumenting the TinyOS example application MViz.

An instrumentation script consists of an arbitrary number of weaving statements each comprising a selector followed by a logging statement in parentheses (line 1 in Fig. 1 shows a canonical example). The selector specifies the function to be traced and, thus, consists of the nesC component, the interface and the function name (separated by spaces). The logging statement consists of the severity, the format string and the variables that should be logged. The severity is optional (defaulting to `SEVERITY_INFO`) and empty parentheses (e.g. line 3) are handled like an empty `@trace()` statement and log the function name and its arguments. In case of a component-local C function, the interface specification of the selector must be empty. In addition to instrumenting function declarations, we also allow inserting logging statements at arbitrary positions in the source code by specifying the component and a line number. Besides the concise form of the selector and the sane defaults for the logging statement, we provide three additional constructs to allow shortening the weaving statements and increasing expressiveness. First, component selectors and interface selectors can be followed by a block enclosed in curly braces {} where only the remaining selectors must be specified as shown in lines 2 and 5 of the example. This simplifies the common case of instrumenting more than one function of a component or an interface and also increases the readability of the instrumentation script. Second, we support C preprocessor statements, which provide the definition of macros, modularity (e.g. line 10) and conditional compilation (e.g. line 9). Conditional compilation also enables better integration in the compilation process and allows, e.g., platform dependent instrumentation or (de)activating the logging of certain components. Third, the most powerful technique is the support for regular expressions for selecting the component, interface and function. Lines 3 and 4 show examples for instrumenting multiple functions with one weaving statement. The support for regular expressions is also the reason for using space as a separator instead of a dot.

Declarative instrumentation does not only provide a non-intrusive alternative to adding logging statements to applications but is also especially suited for instrumenting system code. More information about the lower layers is especially helpful for evaluation purposes and declarative instrumentation renders modifying system and library code unnecessary.

## IV. Framework

In this section, we present the overall architecture of the framework and provide an overview over the compilation process on the one hand and the logging process at runtime on the other hand.

### A. Architecture and Compiler Extension

In Fig. 2 we show the process during compilation and runtime for the use of TinyLTS. The core of our framework is an extended version of the nesC compiler. This version accepts the additional constructs discussed in Section II and declarative instrumentation as discussed in Section III and is responsible for separating dynamic information (the variables to be logged) from static information known at compile time (e.g., the format string). It transforms the `logNow()` statements and the `@trace()` attributes to standard nesC calls using a predefined interface while generating a unique event ID for each statement. All modules that use logging statements are wired to the logging configuration provided by TinyLTS. If a source file does not contain any logging statements or the generation of logging instructions is disabled, no traces of the functionality are left in the resulting binary. Additionally, the compiler creates the so-called sensor node descriptor containing the necessary metadata (event IDs, format strings, types of the variables, locations of the logging statements, etc.) extracted from the source code and stores it in the TinyLTS Mote Repository. We also extended the TinyOS make system to automatically create a mapping in the repository between a node ID and associated sensor node descriptor when a binary is installed on a node. Our extended nesC-compiler is a drop-in replacement for the original and, therefore, the build process for TinyLTS enabled applications including logging statements does not differ from the default process for TinyOS from a developer's point of view.

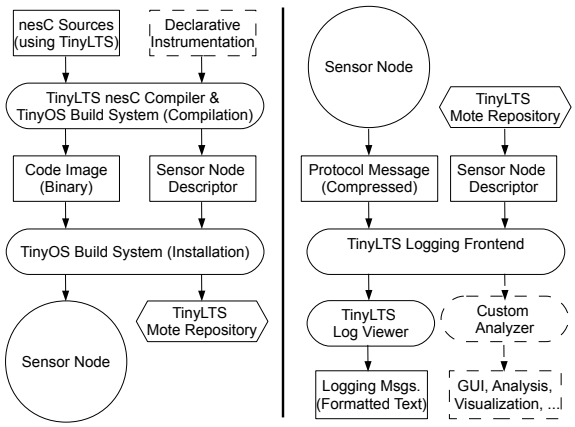



Fig. 2. TinyLTS Overview (compile time/runtime)

### B. Logging Implementations and Extension

We provide several logging implementations to cover the common use cases:

*Serial Connection:* This logging implementation uses the serial connection and is suitable for small tests or permanent testbeds where every node is connected to a host PC.

*Flash Memory Recording:* This implementation records all logging messages in flash memory for post-mortem analysis or for retrieval phases separate from the collection phase.

*Collection:* We provide an implementation using the TinyOS standard collection interface. Note that using multi-hop network protocols by nature incurs a greater impact on the application compared to the previous implementations.

*CTP with extensions (LTP):* We have extended CTP, the current standard implementations of the collection interface, with several functionalities to allow reducing the impact of logging messages and to increase the priority of application messages. These extensions are discussed in Section V.

Since there are cases where our logging implementations cannot be readily used (e.g., when the application does not use the collection interface for routing), one primary goal was to make the framework easily extensible. New logging implementations only have to implement the simple interface shown in Fig. 3. Since the majority of work is done at compile time by our compiler extension, the logging implementations on the nodes are comparably small and simple.

### C. TinyLTS Logging Frontend and Viewer

We provide two components for interpreting the logging data gathered at runtime. The Logging Frontend is a library that provides the necessary functions to combine logging messages received from the network with the correct static information stored in the sensor node descriptor. Since our goal is to reduce overhead on the nodes as far as possible, we send data in the native format of the target platform and the frontend interprets platform specific data representations (e.g., endianness). Currently, we provide the necessary support for msp430 (e.g., TelosB), avr (e.g. MicaZ) and PC (for TOSSIM live). The TinyLTS Log Viewer uses the collected information

```
interface Lts {
    command error_t start(lts_severity_t severity, lts_event_id_t eventId,
                          lts_data_length_t totalDataLength);
    command error_t appendData(const void* data, lts_data_length_t dataLength);
    command error_t finish();
}
```

Fig. 3. Interface for TinyLTS implementations

and generates log messages based on the format strings of the log statements. Additionally, it is possible to use the frontend to create custom applications for analyzing and visualizing the logging data without having to parse the textual output. This simplifies for example the use of logging statements for evaluation purposes.

## V. Logging Protocol

Since network communication has the biggest impact on the behavior of the application, we also investigated possibilities to reduce the impact of logging not readily found in general purpose WSN routing protocols. However, since the network capacity is limited and message losses occur frequently compared to traditional networks, decreasing the impact on application messages results in decreasing the number and, thus, the ratio of successfully transmitted logging packages. Therefore, we distinguish between two kinds of optimizations: first, we consider optimizations that decrease the overhead incurred by lower layers. Second, we provide several enhancements that decrease the priority of logging messages compared to application messages. These enhancements are for the case when the logging messages are of less priority than the application messages. Additionally, we provide the option to use dedicated sinks or additional sinks for logging messages. An example use case for this option is the testing phase at the beginning of a deployment where the goal is to evaluate the performance of the final software and, thus, only the same sinks as in the operational phase should be used. It is often important to get more detailed insight into the network operation and, therefore, logging is more valuable and often more comprehensive in this phase. Additional sinks dedicated to logging messages decrease the impact on the application performance while still providing necessary information. To some extent, this approach can replace specialized hardware with multiple communication interfaces.

The concepts described below are independent of the actual routing protocol used by the application. Our implementation uses the TinyOS collection interface and for more intrusive changes (e.g. the support for multiple sinks), we use the popular implementation CTP as the foundation for our tests.

### A. Reducing Overhead

Our framework achieves a significant reduction of overhead mostly due to the separation of dynamic data from static data. Moreover, TinyLTS can buffer several logging calls thereby increasing the message size but decreasing the number of packets and, therefore, the overhead incurred by the lower layers. Additionally, the framework offers several options for

fine-grained control of metadata recorded for logging calls, e.g. the granularity of timestamps and if timestamps or sequence numbers are included at all. Moreover, TinyLTS only stores timestamp differences and sequence number differences when aggregating multiple logging calls in one packet. We considered several possibilities for aggregating messages, e.g. allowing different event IDs, source nodes or a combination of these approaches. We opted for allowing only different event IDs, since supporting different origins would further increase the overhead and also require to add the size of the data, when the network is heterogeneous. TinyLTS uses the severities to prioritize among different logging calls. This does not reduce the overhead but favors the information deemed more important by the developer.

When using the following extensions implemented for CTP, we additionally exploit unused bits in the header of CTP packets (e.g., only 2 bits of the 1-byte 'option' field are used by CTP itself) for the metadata of TinyLTS to further decrease the overhead. We also implemented piggybacking, i.e. appending logging information to application packets when the maximum payload length is not used, by adding a notification from CTP to the upper layer just before a packet is transmitted.

### B. Privileging Application Messages

We use a dedicated buffer for logging messages to enable prioritization among logging messages based on their severity and to prevent application messages from getting lost due to full queues in the routing layer. TinyLTS uses the already existing 'intercept' hook provided by CTP to prevent CTP from storing logging messages in its own queue. However, we needed to modify CTP to allow reinserting forwarding messages in the appropriate queue since the path information (origin and time-has-lived) has to be preserved. Additionally, we introduce an 'idle' mode: the logging implementation passes messages to the routing layer only if the routing layer is idle, i.e. no messages are in the send queue or in the forwarding queue. Moreover, TinyLTS reduces the number of retries used by the routing layer for logging messages. We extended CTP with the necessary hooks to allow this control.

To create the necessary hooks and notifications enabling the functionalities described so far, we just had to add a few lines to CTP and about 20 more for the necessary interface, default implementations of commands and wiring.

### C. Dedicated Sinks for Logging

The biggest potential for gain lies in the use of additional sinks. As discussed above, certain scenarios merit the use of additional sinks just for logging. The logging implementation uses this feature straightforwardly if provided by the routing protocol. Since CTP supports multiple sinks, but there is no possibility to distinguish among multiple sinks for different message types, we added the necessary support. Since the number of sink types increases RAM and ROM overhead, our version only supports one more type dedicated to logging.

TABLE I
COMPARISON WITH CUSTOM LOGGING LAYER

| Property | CTP | LTS | Printf |
|---|---|---|---|
| **Avg. Payload/Bytes** | 8 | 7 | 27 |
| **Rel. Avg. Payload** | 1.00 | 0.85 | 3.18 |
| **Avg. Serial/Bytes** | 21 | 16 | 40 |
| **Rel. Avg. Serial** | 1.00 | 0.75 | 1.90 |
| **ROM/Bytes** | 36792 | 36742 | 38860 |
| **RAM/Bytes** | 2376 | 1948 | 2106 |

## VI. EVALUATION

Although we consider the increase in maintainability, flexibility and ease-of-use as discussed in Sections II and III to be one of the most important contributions of this work, these metrics are very hard to quantify. Therefore, we concentrate on three issues in the evaluation. First, we investigate replacing a custom logging layer of an existing system by our solution and by TinyOS Printf and compare the three implementations. Second, we analyze the ROM and RAM requirements imposed by our logging implementations. Third, we evaluate LTP - our logging-oriented extensions to CTP - with regard to decreasing the impact of logging messages on application behavior.

### A. Case Study: Replacing a Custom Logging Layer

We chose CTP as the candidate for this case study because it is publicly available in the TinyOS distribution, it is widely used and its logging layer has been identified as an important part by its developers. Due to space constraints, we present only a short overview of the logging layer and refer to the available source code for details. The logging layer consists of standard nesC components and is targeted at evaluating and debugging collection routing protocols. The interface consists of five functions where each function takes the event type as the first parameter. One function just logs the event, two generic functions take one respectively three 16-bit parameters and two functions have the appropriate arguments for specific events in the routing protocol. There is one message struct that comprises the type, a sequence number and a union of the possible argument sets. The available event types are defined as an `enum`. In the source tree, there is one logging implementation available, which uses the serial connection.

For our analysis, we replace the calls to the custom logging layer by `logNow()` calls and by calls to TinyOS Printf respectively. The format string consists of the event ID name and the required number of `%u` directives depending on the used logging function (in addition to the sequence number). For the application, we use the TinyOS sample application MViz that periodically senses data and uses CTP to forward them to a sink and, therefore, is a good representative for both a typical CTP client and a typical sensor network application. We use Cooja [5] to emulate a 25 TelosB node network for 5 minutes and show the results in Table I for the original implementation, TinyLTS and TinyOS Printf. We show the average size of a logging message without ('Payload') and with ('Serial') the overhead incurred by the serial port protocol as

well as the relative sizes compared to the logging layer of CTP. Additionally, we also present the program memory and RAM requirements. Printf generates a considerably larger amount of data that would further increase when using more informative format strings. The reduction in data size when using TinyLTS compared to the original is due to an implementation decision of the CTP developers: the implementation always transmits the generic logging message type regardless of the number of parameters necessary for actual logging function. We use exact byte sizes, which is also requires less buffer space and reduces the RAM requirements. Moreover, the use of TinyLTS replaces more than 250 lines of code for the CTP custom logging layer (not including the code for the PC based interpreter).

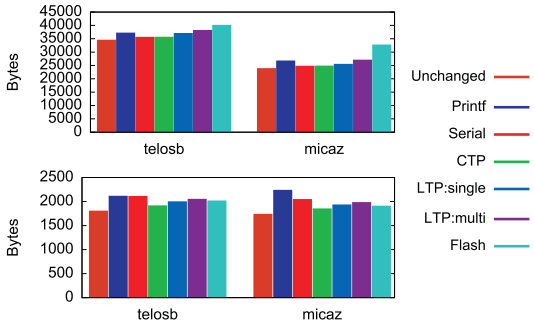



Fig. 4. Program memory and RAM requirements

### B. Memory Requirements and Compile Time

Regardless of the protocol or method used for transmitting or storing the logging information, the memory requirements are very important metrics for all logging implementations. Since TinyOS integrates application code, libraries and the operating system into one binary and the overall size depends heavily on the overlap of required components, it is not possible to determine the size of TinyLTS separately. Instead we use MViz as the base application to measure the overhead of our system and compare it with the overhead of the TinyOS Printf implementation. We added one logging statement each to two functions of MViz using our system and compare it with an implementation using TinyOS Printf. In Fig. 4, we show the program memory and RAM overhead for the unchanged MViz application, MViz with Printf and with our implementations. The overheads of our CTP-based implementations are relatively small, because MViz already uses CTP as the application protocol. However, we still consider this representative since one core idea of our system is to use the same protocol as the application for network wide logging and TinyLTS can be easily used with other protocols. The overhead of our flash implementation consists primarily of the TinyOS support for accessing flash, which is not included in the base MViz application. It is noteworthy that the overhead for using TinyOS Printf is larger than using our serial implementation or even our simple CTP based TinyLTS implementation. The RAM overhead of all implementations (including Printf) results primarily from the buffer allocated for the logging messages. When comparing the RAM overhead of Printf and our serial implementation, we opted for the same buffer size although in almost all cases our implementation requires less space, because no strings are generated on the node.

Adding more logging statements does not incur any RAM overhead when using TinyLTS. The program memory overhead depends on several factors, most importantly the number and complexity of the logging expressions, but can also vary due to compiler optimizations and platform peculiarities. For msp430 based platforms, the cost is usually around $20 + 12k$ bytes where $k$ is the number of variables to be logged.

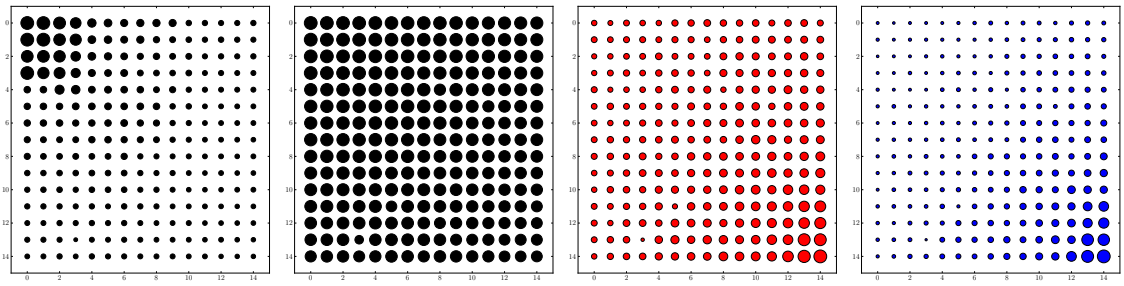
We also investigated the overhead on compilation time for building applications with and without TinyLTS. We tested a minimal application consisting of one logging statement included either explicitly by calling the functions from Fig. 3 or by the use of the TinyLTS compiler. For the overall build process transforming nesC source code to a binary for TelosB, TinyLTS incurs a small 3% overhead.

Additionally, we counted the number of chars required for modifying the MViz source code for use of Printf and TinyLTS (439 vs. 73 for this example). However, the overhead of Printf heavily depends on the variables logged and, thus, a systematic comparison without bias is impossible.

### C. LTP

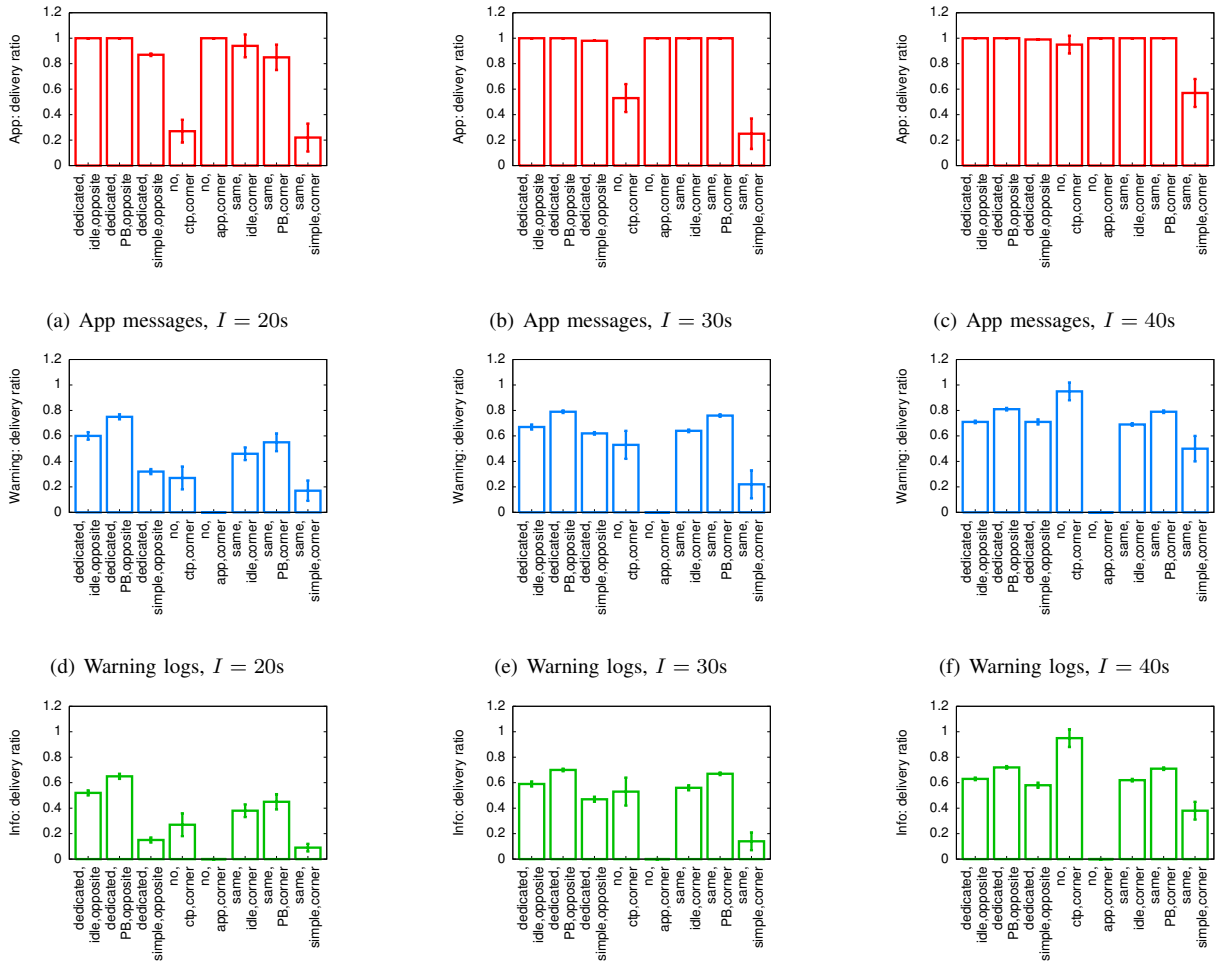
We evaluated LTP in both simulation and testbed using the same scenario: we assume a sensor network application uses CTP to periodically (interval $I$) send some data from each node to the sink. To investigate the impact of logging, we added two different logging messages: a low severity message ('Info log') produced with the period $I_i = I/2$ and a high severity message ('Warning log') produced with the period $I_w = 2I$. Thus, when logging is enabled, the overall message generation frequency is 3.5 times the application message frequency. After a startup phase we run the simulation for approximately 200 application messages. Our primary metric is the ratio between successfully received messages at the sink and the total number of generated messages. We test the following parameter combinations: first, we distinguish between the three major categories no logging messages ('no'), one sink for application and logging messages ('same') and dedicated sinks for logging messages ('dedicated'). For the logging strategies we distinguish between 'simple', i.e. prioritization just among logging messages, 'idle' and piggybacking ('PB') as discussed in Section V. We compare application messages only ('app') also with additional generated CTP messages equaling in size and frequency the logging messages ('CTP'). We use one corner node as the application sink and the opposite corner as the logging sink when evaluating the support for dedicated sinks.

We used TOSSIM of TinyOS 2.1 for simulation. For the topology, we use the '15-15-medium-mica2-grid.txt' topology of the TOSSIM distribution, i.e., a topology generated from empirical measurements in a 15×15 grid. Additionally, we use the 'meyer-heavy.txt' configuration file for noise simulation [6]. We ran each parameter combination discussed above with



(a) App msgs., same corner (b) App msgs., opposite corner (c) Warn. logs, opposite corner (d) Info logs, opposite corner

Fig. 5. Success rates, dedicated sinks for logging, interval: $I = 20$s




(a) App messages, $I = 20$s (b) App messages, $I = 30$s (c) App messages, $I = 40$s

(d) Warning logs, $I = 20$s (e) Warning logs, $I = 30$s (f) Warning logs, $I = 40$s

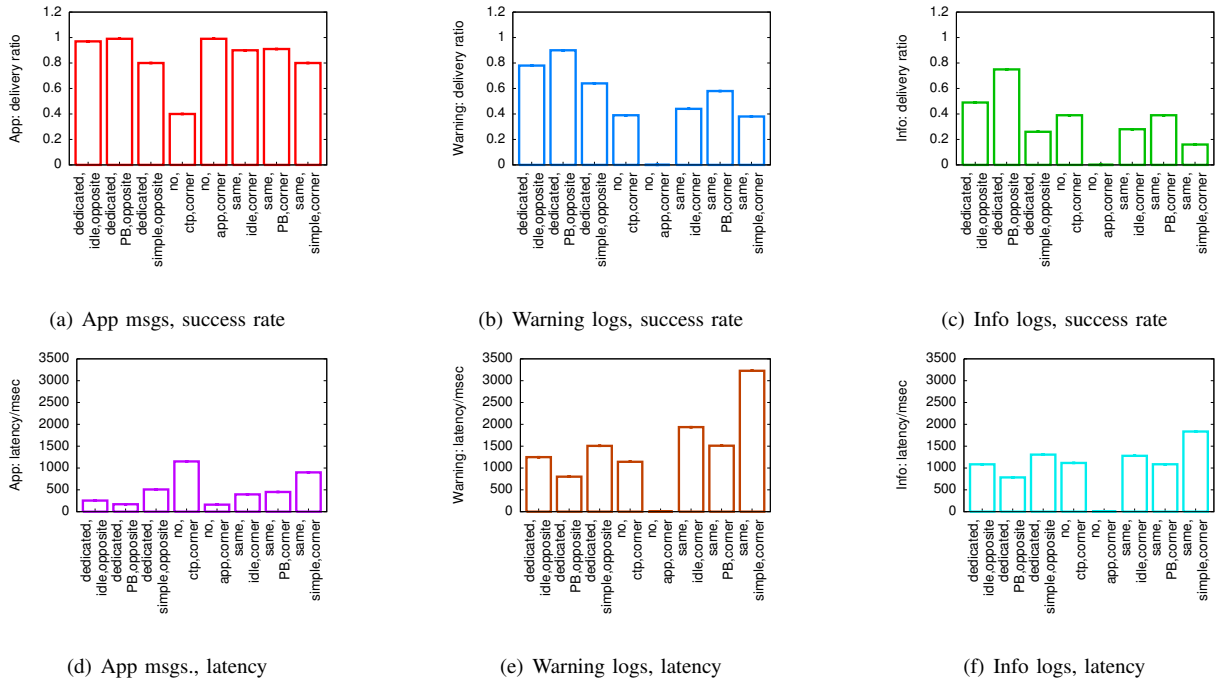
(g) Info logs, $I = 20$s (h) Info logs, $I = 30$s (i) Info logs, $I = 40$s

Fig. 6. Success rates for different strategies and intervals

10 different random seeds and use the mean of the runs.

In Fig. 5 we show a detailed view of the success rates for each node using different sink placements. The area of each circle is proportional to the packet success rate of the corresponding node. The advantage of using the opposite corner for the logging sink is highlighted when comparing a) and b) which show the success rate of application messages. Fig. 5c) and d) show the effect of prioritization among logging messages. The success rate of high severity messages is considerably higher than of low severity messages. Additionally, when comparing the success rates with the application messages success rates, the higher priority of the latter is clear.

In Fig. 6 we show the overall mean success rate and standard deviation for different strategies for the intervals 20s, 30s and




(a) App msgs, success rate

(b) Warning logs, success rate

(c) Info logs, success rate

(d) App msgs., latency

(e) Warning logs, latency

(f) Info logs, latency

Fig. 7. Success rates and latencies for different strategies with the interval $I = 2s$

40s. First, the success rates of application messages for different strategies show that the extensions discussed in Section V significantly reduce the impact on the number of successfully received application messages. The differences decrease when the interval increases as can be seen when comparing a)-c). Second, using a dedicated sink for the logging messages has by far the greatest impact on the success rate of application messages as well as logging messages. It is also noteworthy that with the interval 40s (c), f), i)) the use of plain CTP for application messages and logging messages has only a slight impact on application messages but provides the best performance for the logging messages. This is expected as all the extensions favor application messages over logging messages. The graphs also show that when comparing idle mode with piggybacking (which works on top of idle), the success rate for application messages is slightly lower for piggybacking but the rates for logging messages is higher. This results from the larger size of application packets and the related increase in loss rate combined with the increased amount of data used for logging.

In addition to simulation, we also evaluated our system on the TWIST testbed of the Telecommunication Networks Group at TU Berlin [7]. We use the 102 TelosB nodes deployed in a multi-story office building. Due to the long run-time we were only able to conduct one test for each parameter combination for the intervals $I = 2s$ and $I = 5s$. For the latter case the results for all strategies are almost the same because nearly all packets arrive at the sinks. For $I = 2s$, we show the results for different strategies in Fig. 7. The success rates presented in a)-c) show the same pattern as the results from simulation, only the difference between Warning logs and Info logs is greater. The latencies presented in d)-e) show clearly the effect of buffering of logging messages. The latency of application messages is significantly smaller. The higher latency of Warning logs compared to Info logs is directly related to the higher success rate and privileged treatment. For Info logs, only packets arrive at the sink that do not suffer from many retransmissions and from congestion on their paths and, therefore, the latency of the arriving packets is smaller. This behavior also shows when comparing the latencies of different strategies and in relation to their success rates.

In summary, both simulation results and the tests with real nodes show the effect of our extensions for reducing the impact of logging on the application behavior. However, for best results in case the network is not able to fully provide the necessary bandwidth, we still recommend using a flash based design for post-mortem analysis or for dedicated retrieval phases. Then again the message frequencies for the tested network sizes are relatively high and common workloads of messages every few minutes can be easily supported.

## VII. Related Work

Especially in the last few years, a growing body of work has been dedicated to simplify the development of sensor network applications and assist in debugging. From the viewpoint of the developer, our solution is most closely related to the TinyOS printf library [8] and similar functions of other WSN operating systems because both use the printf-idiom. However, TinyLTS surpasses printf in functionality, flexibility and runtime efficiency considerably. TOSSIM [9] also provides a simple printf-like statement for logging during simulation but cannot

be used during deployment and does not provide the extended capabilities for format strings introduced by TinyLTS.

With Declarative Tracepoints [10] the authors present the use of a declarative language to instrument LiteOS [11] applications with a number of predefined functions. Related to TinyLTS is the ability to record the call of a function (however, without its arguments only) to flash memory. The functionality for logging is very limited since the chosen approach does not allow accessing local variables or arguments of functions and does not always work in case of compiler optimizations. While the declarative language for specifying instrumentation serves a similar purpose in selecting insertion points, TinyLTS provides a more streamlined language for logging for which it is more expressive and concise. Our declarative language fulfills also the same goal as Aspect Oriented Programming [4]: separation of concerns. By specializing the language towards inserting logging statements, which is perhaps the most common example use case of AOP, we provide a more concise and simpler solution. EnviroLog [12] allows recording function calls to flash and replaying them later and the authors of [13] provide a tracing tool that captures the program flow of a function. LiveNet [14] and DustMiner [15] use network sniffers to eavesdrop on network messages to increase visibility for network operations without incurring overhead but cannot improve visibility of the behavior within nodes.

Solutions like Clairvoyant [16] provide source level debugging and [17][18] are examples for (partially) automated diagnosis for sensor networks. As in traditional systems, these approaches complement logging solutions like TinyLTS.

## VIII. Conclusions and Future Work

We presented TinyLTS, a generic and efficient system for logging and tracing with implementations for serial connection, flash memory and a collection protocol. We use a compiler extension to separate static and dynamic information of logging statements for runtime efficiency comparable to custom solutions. The extension automatically records commonly used metadata such as the location of the logging call in the source code and the names and types of the variables. Additionally, we increase decoupling of application code from logging statements by extending the printf-idiom with support for composite data types and a concise yet expressive language for declarative instrumentation. We showed that our logging implementations only require a moderate overhead when integrated in applications and simulation and tests with real sensor nodes show that our extensions to CTP are able to decrease the impact on the behavior of the application. We showed in a case study that substituting TinyLTS for the custom logging layer of a real system is feasible and can even decrease the overhead incurred by logging.

For the future, we are considering tighter integration with simulators. While TinyLTS can be used with TOSSIM Live [19] using the simulated serial port, a replacement of the `dbg` statement would further increase the convenience for the developer. Additionally, we are planning to investigate the support for other operating systems.

## Acknowledgments

We would like to thank the Telecommunication Networks Group at TU Berlin for providing us with the opportunity to evaluate our system on their testbed TWIST [7]. This work has been partially supported by CONET, the Cooperating Objects Network of Excellence, and NOBEL (www.ict-nobel.eu) funded by the European Commission under FP7 with contract numbers FP7-2007-2-224053 and FP7-247926-ICT-2009-4.

## References

[1] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection tree protocol," in *Proc of the 7th ACM Conf. on Embedded Networked Sensor Systems (SenSys'09)*. ACM, 2009.

[2] D. Schmidt and S. Huston, *C++ Network Programming: Systematic Reuse with ACE and Frameworks, Vol. 2*. Pearson Education, 2002.

[3] P. Levis and D. Gay, *TinyOS Programming*. Cambridge Univ. Press, 2009.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proc of the 11th European Conf. on Object-Oriented Programming*, 1997.

[5] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-Level Sensor Network Simulation with COOJA," in *Proceedings 2006 31st IEEE Conference on Local Computer Networks*, 2006, pp. 641–648.

[6] H. Lee, A. Cerpa, and P. Levis, "Improving wireless simulation through noise modeling," in *Proc of the 6th intl. conf. on Information processing in sensor networks (IPSN'07)*. ACM, 2007.

[7] V. Handziski, A. Köpke, A. Willig, and A. Wolisz, "Twist: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks," in *Proc of the 2nd intl. workshop on Multi-hop ad hoc networks: from theory to reality (REALMAN'06)*. ACM, 2006.

[8] The TinyOS printf Library. [Online]. Available: http://docs.tinyos.net/index.php/The_TinyOS_printf_Library

[9] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire TinyOS applications," in *Proc of the 1st intl. conf. on Embedded networked sensor systems (SenSys'03)*. ACM, 2003.

[10] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo, "Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks," in *Proc of the 6th ACM conf. on Embedded network sensor systems (SenSys'08)*. ACM, 2008.

[11] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He, "The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks," in *Proc of the 7th intl. conf. on Information processing in sensor networks (IPSN'08)*. IEEE Computer Society, 2008.

[12] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic, "Achieving repeatability of asynchronous events in wireless sensor networks with envirolog," in *Proc of the 25th IEEE Intl. Conf. on Computer Communications (INFOCOM'06)*, Apr. 2006.

[13] V. Sundaram, P. Eugster, and X. Zhang, "Lightweight tracing for wireless sensor networks debugging," in *Proc of the 4th Intl. Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks (MidSens'09)*. ACM, 2009.

[14] B.-R. Chen, G. Peterson, G. Mainland, and M. Welsh, "Livenet: Using passive monitoring to reconstruct sensor network dynamics," in *Proc of the 4th IEEE intl. conf. on Distributed Computing in Sensor Systems (DCOSS'08)*. Springer-Verlag, 2008.

[15] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han, "Dustminer: troubleshooting interactive complexity bugs in sensor networks," in *Proc of the 6th ACM conf. on Embedded network sensor systems (SenSys'08)*. ACM, 2008.

[16] J. Yang, M. L. Soffa, and K. Whitehouse, "Effective source-level debugging of wireless sensor networks," in *Proc of the 5th intl. conf. on Embedded networked sensor systems (SenSys'07)*. ACM, 2007.

[17] K. Liu, M. Li, Y. Liu, M. Li, Z. Guo, and F. Hong, "Passive diagnosis for wireless sensor networks," in *Proc of the 6th ACM conf. on Embedded network sensor systems (SenSys'08)*. ACM, 2008.

[18] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, "Sympathy for the sensor network debugger," in *Proc of the 3rd intl. conf. on Embedded networked sensor systems (SenSys'05)*. ACM, 2005.

[19] C. Metcalf, "Tossim live: Towards a testbed in a thread," Master's thesis, Colorado School of Mines, 2007.