ISSN 0105-8517

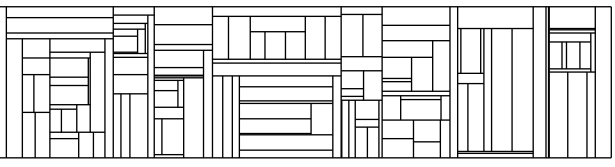
TinyDebug

Multi-Purpose Passive Debugging Framework for Embedded Wireless Systems

Morten Tranberg Hansen

DAIMI PB - 595
October 2011

DEPARTMENT OF COMPUTER SCIENCE
AARHUS UNIVERSITY
IT-parken, Aabogade 34
DK-8200 Aarhus N, Denmark



TinyDebug: Multi-Purpose Passive Debugging Framework for Embedded Wireless Systems

Morten Tranberg Hansen
Department of Computer Science
Aarhus University
mth@cs.au.dk

ABSTRACT

Debugging embedded wireless systems can be cumbersome due to low visibility. To ease the task of debugging this paper present TinyDebug which is a multi-purpose passive debugging framework for developing embedded wireless systems. TinyDebug is designed to be used throughout the entire system development process, ranging from simulation to actual deployment. TinyDebug provides out-of-the-box message oriented debugging and event logging mechanism while enabling more advanced debugging techniques to process the same debug events.

We present the TinyDebug framework with all its features from event logging to extraction and show how the framework improves upon existing message based and event logging debugging techniques while enabling distributed event processing. We also present a number of optional event analysis tools demonstrating the generality of the TinyDebug debug messages.

1. INTRODUCTION

An important aspect of any system development is debugging. Being able to debug a system means that the developer can validate that a system performs as intended. In embedded wireless systems debugging is especially hard due to low visibility and unpredictable environmental effects that might change over time. To cope with this, embedded wireless systems need to be extensively tested throughout its development process in different environments before an actual deployment.

An embedded wireless system development process will normally start out in a simulator, validating that the logic of the application works in a (to some degree) simplified environment where it's easy to reproduce event traces. The next step would be to validate the system behavior on real hardware. This often includes testing on a local testbed consisting of a small number of nodes wired to a PC. Such a setup does not enable one to do much network analysis, so the next step would be to validate the system on a larger spatially distributed testbed, possible with a wired or wireless back channel for increased visibility. This is often done indoor or in close proximity of an office building and hence might not reflect the actual target environment. Thus the final test should be done with a real deployment in the target environment where a wired or wireless back channels would not be possible.

Going through the development process debugging channels becomes unavailable and resource consumption becomes more critical, so the further the developer gets the more the cost of debugging needs to be considered. In simulation resources are not a concern and hence advanced debugging methods can be used without considering the costs. Taking the development a step further from simulation, a testbed (at any scale) is limited by the capacity of its back-channel, and a real deployment can be limited down to a few flags embedded in a data packet. These limitations can make it impossible to employ advanced user-driven interactive debug methods [12, 13, 14, 3] and hence one would have to do with some form of passive debugging mechanism which is the subject of this paper.

We argue that passive debugging mechanisms are a vital part of any sensor network system (even as a basic building block for more advanced user-driver interactive debug mechanism), and can be divided into three categories: message based debugging, event logging and analysis, and distributed event processing. Message based debugging refers to the printf like family of debug statements which outputs a human-readable message whenever an event occurs [6]. Event logging sacrifices the human readability of the message based debugging for efficiency and only outputs and identifier (ID) and its arguments whenever and event occurs [1, 2]. Finally, with distributed event processing the events are not directly outputted but instead processes internally and only sent whenever a change is detected [7]. Although these methods are not mutually exclusive, they are often used in separation: message based debugging is used in simulation and in a small local testbed when during initial development of an application, event logging and analysis is used at large scale experiments where the resource restrictions prevents text messages from being used, and finally distributed event processing is used in deployments where event logging is either too expensive or not feasible. Thus, developers tend to change debugging strategy throughout the development of an application, which does not only increase the complexity of application development but also makes it harder to reproduce bugs in different environments.

To ease debugging throughout the development process

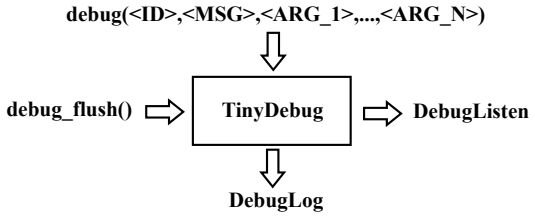


Figure 1: Overview of the TinyDebug embedded architecture and the syntax of its debug function. Debug events are internally processed in TinyDebug and provided to applications through a listen and a logging interface.

we present TinyDebug: a multi-purpose passive debugging framework for developing embedded wireless systems. TinyDebug enables message oriented debugging with the efficiency of event logging, event logging with variable number and types of arguments, and easy accessible hooks for distributed event processing—all based on the same user inserted debug statements.

The paper is organized as follows. Section 2 present the TinyDebug framework with emphasis on its features and related tools. Section 3 explores a number of TinyDebug use cases and evaluate how it performs compared to current practice, while Section 4 concludes the paper. The TinyDebug code presented in this paper is made public available at `https://github.com/mortenthansen/tinyos/tree/tinydebug-1.0`.

2. TINYDEBUG

TinyDebug is a multi-purpose passive debugging framework for developing embedded wireless systems. It is indeed not the first attempt to debug embedded wireless networks but are, to best of our knowledge, the first attempt to design a multi purpose debugging framework that support the entire application development process ranging from initial simulations to actual real-world deployments.

For efficiency the design of TinyDebug is kept simple. TinyDebug unifies event logging, collection and processing by providing an efficient passive debugging mechanism which can be used out-of-the-box or as a basic building block to implement advanced debugging techniques in different environments. This includes logging mechanisms such as code annotations [13] or automatic insertions [5, 11], event collection mechanisms [9, 7], event trace analyzers [5, 11, 8], or user interactive debugging methods [13, 14, 3] . Hence, TinyDebug does not enable any novel debugging methods but instead unifies existing once by providing an efficient general purpose event logging mechanism which can be used by the developer at all stages in the development process.

Systems can interact with TinyDebug through two system calls (`debug` and `debug_flush`) while software components can subscribe to TinyDebug events through a listening or a logging interface. Figure 1 shows an overview of TinyDebug embedded architecture including the syntax of its `debug` function.

2.1 Debug Function

A core component of TinyDebug is its generic `debug` function which is used to log an event. The syntax of the TinyDebug `debug` function is inspired by the TOSSIM `dbg` function [6] and takes two mandatory string parameters, an ID and a message, together with an optional number of arguments.

Format	Argument
%hhu	8bit unsigned integer
%hu	16bit unsigned integer
%lu	32bit unsigned integer
%llu	64bit unsigned integer
%hhi	8bit signed integer
%hi	16bit signed integer
%li	32bit signed integer
%lli	64bit signed integer
%hhx	8bit unsigned hexadecimal
%hhx	16bit unsigned hexadecimal
%hhx	32bit unsigned hexadecimal
%f	32bit float

Figure 2: TinyDebug supported arguments. The format of the arguments is embedded in order in the message string. This is similar to the well known printf family of debug statements. Note that TinyDebug support float arguments.

The ID string is used to identify a particular event. TinyDebug uses a hierarchical ID name-space where the hierarchical levels are separator by a ",". This makes filtering of events and the task of avoiding unwanted ID name clashes a lot easier. The message string is a human-readable text string describing the event and its arguments. Arguments are described in the familiar printf way by embedding their type into the message string using the special format strings shown in Figure 2.

The fact that the TinyDebug `debug` function makes use of two string argument is a tradeoff between program memory efficiency and usability. First, the use of an ID string was chosen so that it could have a meaningful name and that it did not have to be previously defined (which would have been the case if an enumeration constant was used). Secondly, the use of a message string was done to enable message based debugging and in order to support a variable number and type of arguments to the same unified `debug` function.

TinyDebug does not specify how calls to the `debug` functions are inserted into an application and assume that these are already there. In some systems they could be automatically inserted [3, 11], but in the most common case they would be manually inserted. One might consider this a cumbersome approach, but we argue that any developer would use some form of message oriented debug mechanisms (including printf, TOSSIM dbg, or assertions) during initial development of a module. Thus, if the developer used the TinyDebug `debug` function in that initial development, the statements would already be present and available in the later stages of the development process.

2.2 Message Format

Internally in TinyDebug all calls to the `debug` functions are processed and translated into the generic TinyDebug debug message format shown in Figure 3. This includes a length field specifying the length of the debug message, a unique UID identifying the specific call to the `debug` function (note that this is not the same as the ID string parameter from the `debug` function), a time-stamp, a sequence num-

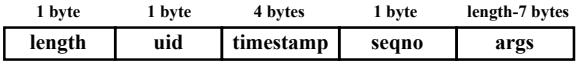


Figure 3: TinyDebug debug message format. All TinyDebug events are transformed into debug messages of this format before being provided to any embedded listeners or loggers.

```
interface DebugListen {
  async command void handle(const char* id,
                            debug_msg_t* debug);
}
```

Figure 4: Debug Listen interface which is used every time a TinyDebug debug event is processed. Once a call to the handle command returns the ownership of the allocated debug message returns to TinyDebug.

ber, and the variable number of arguments given. The UID is used as a reference to the specific `debug` call and hereby the human-readable message and the number and types of arguments which, for efficiency, is not stored with every debug message. The fact that the UID is only one byte long limit a system to 255 different debug statements. In systems where this is not enough the UID field can be extended to two bytes, enabling 65536 different debug statement, which should be enough for even the most avanced embedded system. The time-stamp is in local millisecond time but is transfered into a global time when the debug message is collected from the embedded device, and the sequence number can be used to detect any missing debug messages.

2.3 Event Distribution

TinyDebug provide two ways for other modules to receive debug messages: as a logger or a listener. A logger is a module meant for storing traces of debug messages for later inspection and a listener is a local module interested in the debug messages triggered by the running system.

TinyDebug internally keeps a buffer from which space for new debug messages are allocated. A listener is handed a debug message immediately upon generation and is expected to just do a fast inspection of the debug message and then return whereas logging of a debug message depends on peripherals and hence can be a more time consuming task. Therefore, in order to not interfere with the running system, logging is only done whenever instructed to by the application through a call to the `debug_flush` function or whenever the internal buffer is almost full. If the logging is triggered by the buffer almost being full, TinyDebug generates a special debug message informing the developer that this was the case and that logging might have interfered with the application. The fact that logging is deferred means that in the common case more than one debug message will be logged at a time which increases energy efficiency when logging to a device such as a flash disk. The TinyDebug listening and logging interfaces is shown in Figure 4 and Figure 5, respectively.

If the rate of logged events (calls to the `debug` function) is higher than the rate events can be flushed from the debug buffer, debug messages can be lost. If this happens, TinyDebug generates a special debug message informing the developer that debug messages has been lost. The developer

```
interface DebugLog {
  command void flush(uint8_t* buf, uint16_t len);
  event void flushDone();
}
```

Figure 5: Debug Log interface which is used whenever instructed to by a call to the debug_flush function or the internal TinyDebug buffer is almost full. The ownership of the allocated debug messages (stored in buf) does not return to TinyDebug until the callee signals flushDone.

can then refer to the debug messages sequence numbers in order to figure out how many were lost. Note that TinyDebug always leaves buffer space for its own special debug messages so that these can always be logged.

2.4 Event Logging

TinyDebug comes with two standard ways of logging debug messages: directly to the serial or the radio, or to flash for later extraction through the serial or radio.

In theory there is no restriction on the length of a debug message which depends on the number and types of the arguments, but in practice with a 1 byte length field, they are limited by a max length of 255. When logging a debug message to flash, this length is not an issue as most block sizes are larger than 255, but when logging a debug message over the serial or radio it could be. TinyDebug handles this with fragmentation. It adds a one byte fragment header to all debug messages sent over the serial or radio where the first four bits describe the number of fragments a debug message has been divided into and the last four bits describe the current fragment number. Fragmented debug messages can then be assembled at the receiver (see next subsection).

Every debug message contains a local time-stamp from the time of creation. When a debug message is transferred to another node the time-stamp is transferred into the receivers time. TinyDebug does this by modifying the time-stamp before transmission to be the difference between the event time and the transmission time. At the receiving side, this difference will then have to be subtracted the receiver's local time of reception, and hence the time-stamp will be in the receiver's local time. This is similar to the time-stamping approach taking in RITS [10] which reports an accuracy in the order of few tens of micro seconds.

2.5 TinyDebug Client

TinyDebug provides a PC side TinyDebug client which is able retrieve debug messages from a number of nodes simultaneously while handling possible fragmentation that might occur due to debug message being longer than the links payload size. The fact that the same TinyDebug client, running on the same host PC, receives data from all nodes enable it to transform the time since an event from the received debug messages into global POSIX time based on the same PC clock. Furthermore, if debug messages are stored in flash for later retrieval, the TinyDebug client will send a retrieval command to the nodes upon connection which will initiate the retrieval of all stored debug messages.

TinyDebug include a fetch script which automatically fetches an applications calls to the `debug` function and their respective UID, ID string and message string from the source code

```
POSIX time      integer     integer     string   integer     integer
<global timestamp>, <nodeid>, <seqno>, <ID>, <arg1>, ..., <argN>
```

Figure 6: TinyDebug client output CSV data format. Integers are represented by their base 10 value.

right after it has been compiled. This information is then fed into the TinyDebug client which processes the incoming debug messages according to their UID, and outputs an easy parsable comma separated value (CSV) string of events, or the corresponding human readable text.

The CSV format is shown in Figure 6 and consists of a global time-stamp, a node identifier, the node sequence number ranging from 0 to 255, the message ID string, and the arguments. TinyDebug does not enforce any method or tool for how this data is processed, but in the following section we will suggest a few approaches.

2.6 Implementation Details

TinyDebug is implemented for TinyOS where it is integrated into the TinyOS tool-chain as an extra make target. Thus TinyDebug can be easily enabled or ignored per application when compiling a program.

It is no secret that processing TinyDebug events takes resources and hence when compiling a program with TinyDebug one might not want to process all event embedded in the code. For efficiency, TinyDebug uses a filter or an ignore list to enable a developer to filter or ignore some debug messages. These lists are given as compile time constants in TinyDebug.

The syntax of the TinyDebug `debug` is similar to the TinyOS TOSSIM `dbg` function so when compiling TinyDebug for TOSSIM each call to the TinyDebug `debug` function is accompanied by a similar call to the TOSSIM `dbg` function. Thus one can use the TinyDebug `debug` function as a replacement for the TOSSIM `dbg` function which means that when developing with TOSSIM one would not need to make use of the TOSSIM serial connection when doing message based debugging.

The TinyDebug client is implemented similar to its TinyOS Printf client counterpart and takes a "-comm" parameter specifying where a sensor node is connected. Furthermore, it can also take a list of connection points as standard input which it will try and connect to. This enables the TinyDebug to collect debug data from multiple sensor nodes, simultaneously, and then write this data into one combined CSV log file. This comes especially handy when dealing with testbeds where mulitple nodes might be logging debug events simultanously to a wired or wireless back-channel.

3. USE CASES

In this section we show how TinyDebug can be used for simple message based debugging, event logging and analysis, and simple distributed event processing.

3.1 Message Based Debugging

The printf family of message based debugging functions is used when debugging various systems. They provide, even in embedded systems, a convenient way to debug an application. The fact that TinyDebug makes use of a human readable message string makes it compatible with printf.

In embedded systems printf is often implemented by compiling a character string on the embedded device and then

```
Hi I am writing to you from my TinyOS application!!
Here is a uint8: 123
Here is a uint16: 12345
Here is a uint32: 1234567890
```

Figure 7: Message based debugging application output of the default TinyOS printf test application. TinyDebug improves the communication overhead of this output compared to printf with 63%

sending the entire character string over the serial to a terminal on a PC. This is expensive and scales with the size of the message. TinyDebug improves upon this by only sending the UID and the arguments over the serial and then compiling the character string in the TinyDebug client on the PC. Thus instead of scaling with the message size TinyDebug scales with the arguments.

To illustrate the improvement of TinyDebug compared to printf we compare the communication overhead of the default TinyOS printf test application using the default printf implementation and our TinyDebug implementation. The application is very simple and only prints the four lines of text with embedded integers shown in Figure 7 when started. Using printf the total amount of data sent over the serial when the application is started is 180 bytes whereas with TinyDebug it is 67 bytes. This is an improvement of 63%.

In addition to this improvement TinyDebug, as opposed to print, handles floats as one would handle integers (see Figure 2). With the TinyOS printf implementation, floats has to be printed as two integers: one representing the integer value and one representing the rest with some granularity.

3.2 Event Logging

When debugging advanced distributed protocols at a larger scale, the number of events increases and the overhead of traditional message oriented debugging becomes unacceptable. In such cases, more space efficient events represented by an event UID and a number of arguments is used. A widely used example of this is the TinyOS collection debug messages [1] which is used to debug collection protocols such as the well known Collection Tree Protocol (CTP) [4]. The collection debug messages does not support variable numbers and types of arguments and consist of an UID, a sequence number, and three 16 bit unsigned integer argument. Hence whenever a debug message does not make use of all three argument space is wasted.

To illustrate the improvement of TinyDebug compared to the TinyOS collection debug messages we ran CTP with the two approaches on a 25 node grid network while logging 30000 events. Collection debug events are already embedded into the CTP code but TinyDebug events are not. Instead of embedding TinyDebug events into the CTP code we made a collection debug to TinyDebug adapter which translates the collection debug events to corresponding TinyDebug events with similar arguments. Thus we do not have to modify the existing CTP code in order to debug it with TinyDebug. Note that this adapter approach can be applied in general to existing code already including some form of debugging mechanism.

The 30000 CTP debugging events using the collection debug messages results in 510000 bytes of debug data whereas TinyDebug produces 525715 bytes of debug data. This is a 3% increase caused by the fact that TinyDebug debug mes-

```
>> nodes(2)
 ans =
                                id: 1
       Collection__FE_ARRIVED_MSG: []
    Collection__FE_DUPLICATE_CACHE: []
    Collection__FE_DUPLICATE_QUEUE: []
        Collection__FE_FORWARD_MSG: []
      Collection__FE_LOOP_DETECTED: []
       Collection__FE_RECEIVED_MSG: []
  Collection__FE_SENDDONE_WAITACK: [11x4 double]
  Collection__FE_SEND_QUEUE_EMPTY: [196x4 double]
          Collection__FE_SENT_MSG: [195x4 double]
      Collection__TREE_NEW_PARENT: [2x4 double]
 Collection__TREE_RECEIVED_BEACON: [313x4 double]
     Collection__TREE_SENT_BEACON: [15x4 double]
```

Figure 8: Matlab processed TinyDebug event data for a TinyOS CTP run. The event data is stored in a per node structure where the events are identified by a Matlab compatible message ID string.

sages includes an extra 4 byte time-stamp. If we leave out the time-stamp from TinyDebug debug messages it will only produce 405715 bytes of data which is a 20% improvement compared to the collection debug messages.

A related TinyOS debug mechanism is DiagMsg [2] which supports variable number and type of arguments by prepending every argument with a 4bit type field. DiagMsg will outperform collection debug messages if the event arguments vary a lot whereas if there are always three 16 bit integer arguments the type fields become redundant, and the collection debug message approach becomes the preferred one. The beauty of TinyDebug is that it will always have minimal overhead as the number and type of arguments are not embedded in the debug messages. However, one limitation of TinyDebug compared to DiagMsg is that it is currently limited to 255 different debug events, but as discussed in Section 2.2 this can be extended by increasing the size of the UID's.

3.3 Event Analysis

Event logging is tightly coupled to event analysis. Event analysis is application specific and cannot be completely generalized which is why TinyDebug does not enforce any method or tool for how the event data shown in Figure 6 is processed. However, TinyDebug provides a couple of optional tools which can ease the task of event analysis.

First, TinyDebug comes with a speed optimized `debug2mat` script which converts TinyDebug event data into easy accessible binary Matlab data. The Matlab data is a node array where each element is a structure containing the node ID and its event data identified by its message ID string (where "," is substituted with "__" for Matlab compatibility). Each event is represented by its time-stamp and its arguments. The Matlab data format for a the CTP TinyDebug run discussed in the previous subsection is shown in Figure 8. This format enables one to do easy per node event analysis with human readable event names. Furthermore, once the data is saved as a Matlab binary file it is a lot faster to load it into Matlab. For small experiments with limited event data this is not a concern, but for longer experiments, where the CSV data can be in the order of mega bytes, only having to process this once makes a big difference.

TinyDebug also comes with a network visualization tool called DVIZ which provides a graphical online or off-line representation of the network dynamics and statistics based on the TinyDebug debug messages. DVIZ processes per node events and based on their message ID string either ignores the event, count the number of event, accumulate an event argument, or show the latest value of an event argument. The action to be taken for each event is given as input to DVIZ but defaults to counting the number of events if nothing is specified for a certain message ID string. DVIZ also shows a graph of the network where the edges from a node to another is set upon reception of a certain argument of a specified parent event. The parent event and argument is specified by its message ID string and argument number, and is given as input to DVIZ. For the convenience of the user, DVIZ is able to automatically layout the network graph with a minimal number of overlapping edges but also enable the user to statically specifying the locations of nodes. The location of nodes is specified in a special topology file which is optionally given as input to DVIZ. The format of this topology file is similar to the once produced by the widely used TinyOS TOSSIM link generator based on a USC link model [15].

Figure 9 shows a screen-shot of DVIZ while running the 25 node CTP experiments from the previous subsections. In this example argument 2 of the "Collection__FE_SENT_MSG" event is used as the parent identifier and DVIZ is given the fixed grid topology as input. In the figure, the user has selected node 0, 1, 5, and 6, which means that the statistics of these nodes are shown in the table in the bottom of the figure. In this example all events are simply counted except for the "Collection__FE_ARRIVED_MSG" event which shows the source of the last arrived message.

3.4 Distributed Event Processing

When debugging a system in a resource restricted real world deployment logging entire event traces might be considered too expensive or not feasible. Hence one might want to process the events locally and then only inform the user whenever something important changes. To enable this, TinyDebug provides the debug listen interface which a customized distributed event processing solutions can hook into and then get immediate information about debug events.

On top of listening to debug events any such distributed event processing solution will need a way to inform the user of changes. If it is considered too expensive to do this by sending specialized messages over the serial, radio, or to the flash disk it would have to prepend any such information to already existing messages sent. To ease the task of doing this TinyDebug comes with a specialized debug header component which can be wired into any communication stack with the standard TinyOS send, receive and packet interfaces. This component provides the type parameterized interface shown in Figure 10 where a distributed event processing solution should take the appropriate actions to either read or set the header `hdr` from message `msg` in the send, intercepted, received, or snooped events.

An example of a distributed event processing solution is a simple network diagnostic tool reporting a nodes next hop during collection in a sensor network. Such a solution could embed a couple of bytes representing a node and its parent in all data packet using the debug header component. A node will only have to report this whenever a new parent is reported by the "Collection__TREE_NEW_PARENT" event which it would have to listen too through the debug listen

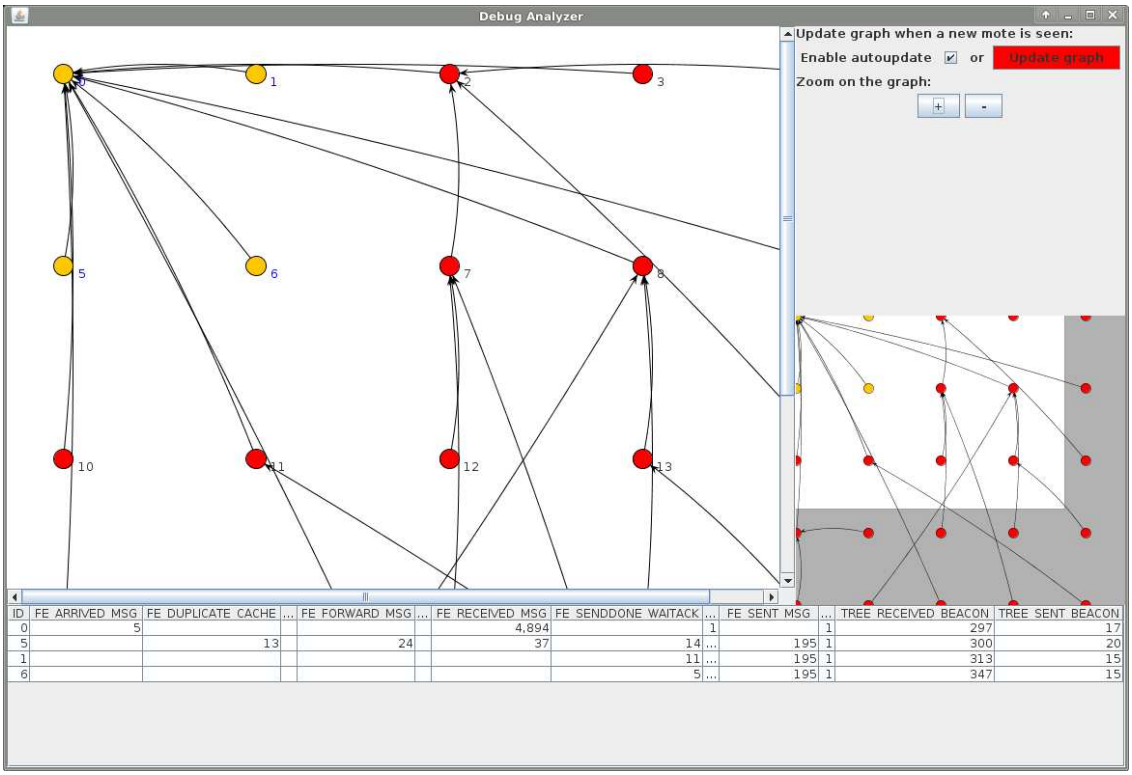


Figure 9: DVIZ Graphical Network Analyzer. DVIZ workspace is divided into three sections: a network graph showing the current network topology (left), a control panel (right), and a table showing statistics for the nodes selected in the network graph (bottom).

```
interface DebugHeader<t> {
 event void sendHeader(t* hdr, message_t* msg);
 event void interceptHeader(t* hdr, message_t* msg);
 event void receiveHeader(t* hdr, message_t* msg);
 event void snoopHeader(t* hdr, message_t* msg);
}
```

Figure 10: Debug header interface which can be used by any distributed event processing solution that want to embed debug information into existing data packets.

interface. At the origin the embedded parent information will be set on a data packet in the `sendHeader` event, if changed, and at any intermediate node in the path of the data packet, it would be done in the `interceptHeader` event, if it has changed and is not already set by any of its children. Assuming a relative stable network, the root of the collection will over time have a complete overview of the current network topology without receiving specialized debug messages and without the developer having to modify existing source code to report the change of parent.

It is important to note that prepending debug messages to existing messages can introduce new bugs in a system and hence should be done at all stages of the system development process if needed in the final deployment in order to eliminate Heisenbugs (which is a bug that disappear or alters its behavior when an attempt to isolate it is made).

4. CONCLUSIONS

This paper presented TinyDebug as a multi-purpose passive debugging framework usable in all levels of the embedded wireless system development process.

We presented TinyDebug's generic `debug` function and showed how calls to this is processed and converted into generic debug messages. We showed how TinyDebug enable these messages to either be buffered for later extraction through the radio or serial, or internally processed by some distributed event processing solution. TinyDebug is not only an embedded framework, it also comes with a TinyDebug client designed to extract debug messages from multiple embedded system while handling possible fragmentation.

By case studies we demonstrated how TinyDebug improves upon existing embedded message based and event logging debugging techniques, and how it can easily be used to implement more advanced distributed event processing debugging techniques. More specifically, we showed that TinyDebug decreases the amount of data sent over the serial with message based debugging with 63% for the standard TinyOS printf test application and with 20% for 30000 debug events with CTP running on a 25 grid network when the the overhead of the extra TinyDebug time-stamps are neglected.

The TinyDebug code presented in this paper is made public available at `https://github.com/mortenthansen/tinyos/tree/tinydebug-1.0`.

5. REFERENCES

[1] TinyOS Collection Debug Message. `http://code.google.com/p/tinyos-main/source/browse/\#svn\%2Ftrunk\%2Ftos\%2Flib\%2Fnet`, Oct. 2011.

[2] TinyOS Diagnostic Message. `http://code.google.com/p/tinyos-main/source/browse/\#svn\%2Ftrunk\%2Ftos\%2Flib\%2Fdiagmsg`, Oct. 2011.

[3] 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 *Proceedings of the 6th ACM conference on Embedded Network Sensor Systems*, SenSys '08, pages 85–98, 2008.

[4] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection Tree Protocol. In *Proceedings of the 7th ACM Conference on Embedded Network Sensor Systems*, SenSys '09, 2009.

[5] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 99–112, New York, NY, USA, 2008. ACM.

[6] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 126–137, 2003.

[7] K. Liu, M. Li, Y. Liu, M. Li, Z. Guo, and F. Hong. Passive diagnosis for wireless sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 113–126, 2008.

[8] H. Pham and J. Mazzola Paluska. PerViz: Painkillers for pervasive application debugging. pages 208–216, Mar. 2010.

[9] S. Rost and H. Balakrishnan. Memento: A Health Monitoring System for Wireless Sensor Networks. In *2006 3rd Annual IEEE Communications Society on Sensor and Ad Hoc Communications and Networks*, volume 2, pages 575–584. IEEE, Sept. 2006.

[10] J. Sallai, A. Lédeczi, and P. Dutta. On the Scalability of Routing Integrated Time Synchronization.

[11] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: global views of distributed program execution. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 141–154, New York, NY, USA, 2009. ACM.

[12] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks*, EWSN '05, pages 121–132, Feb. 2005.

[13] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using RPC for interactive development and debugging of wireless embedded networks. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks*, IPSN '06, pages 416–423, 2006.

[14] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 189–203, 2007.

[15] M. Zuniga and B. Krishnamachari. Analyzing the transitional region in low power wireless links. In *In First IEEE International Conference on Sensor and Ad hoc Communications and Networks (SECON*, pages 517–526, 2004.