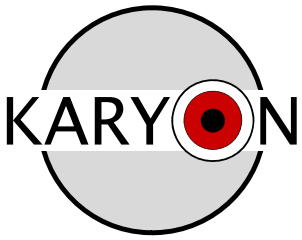

Kernel-based ARchitecture for safetY-critical cONtrol

KARYON

FP7-288195

D4.3 – First report on Cooperative Diagnostics

Work Package	WP4		
Due Date	M24	Submission Date	2013-12-03
Main Author(s)	Olaf Landsiedel (CTHA)		
Contributors	Salvo Tomaselli (CTHA), Elad M. Schiller (CTHA)		
Version	1.2	Status	Final
Dissemination Level	Public	Nature	Report
Keywords	Distributed Diagnostics, Debugging Distributed Systems, Tracing, Deterministic Replay		
Reviewers	Kenneth Östberg (SP)		

Part of the Seventh

Framework Programme

Funded by the EC – DG INFSO

Version history

Rev	Date	Author	Comments
V0.1	2013-08-29	Olaf Landsiedel (CTHA)	Initial Structure
V0.2	2013-09-10	Olaf Landsiedel (CTHA)	First draft
V0.3	2013-09-12	Olaf Landsiedel (CTHA)	Complete, needs polishing & clean-up
V0.4	2013-09-16	Olaf Landsiedel (CTHA)	Polishing, added cites
V0.4	2013-09-17	Olaf Landsiedel (CTHA)	Ready for internal review
V0.5	2013-09-23	Olaf Landsiedel (CTHA)	Integrated review from SP (Kenneth Ö.)
V1.0	2013-09-23	Olaf Landsiedel (CTHA)	Final
V1.1	2013-11-12	Olaf Landsiedel (CTHA)	Integrated comments from WP4.3 WebEx 2013-11-06
V1.2	2013-12-03	António Casimiro (FFCUL)	Final review and delivery

Glossary of Acronyms

AUTOSAR	AUTomotive Open System Architecture
CPS	Cyber Physical System
CPU	Central Processing Unit
ECU	Engine Control Unit
KARYON	Kernel-based ARchitecture for safetY-critical cONtrol
MCU	Microcontroller Unit
MILD	Minimal Intrusive Logging and Deterministic Replay
OS	Operating System
TinyOS	Tiny Operating System (for WSNs)
Tx.y	Task belonging to work package x, with serial number y
WP	Work Package
WPx	Work Package with serial number x
WSN	Wireless Sensor Network

Executive Summary

Collaborative vehicles demand for thorough testing and evaluation, as their operation is inherently safety critical. However, diagnosing and debugging such cooperative systems during deployment is challenging, due to the concurrent nature of distributed systems, the interaction between the different vehicles, and the limited insight that any deployed system offers.

In KARYON we address this challenge by designing MILD; providing Minimal Intrusive Logging and Deterministic replay. MILD enables logging of events on deployed Cyber-Physical Systems at minimal intrusion and their cycle accurate and deterministic replay in controlled environments such as system simulators. In this report we present the design and architecture of MILD, discuss the underlying motivations for its design, and present an initial prototype implementation. Additionally, we report insights into its flexibility and discuss performance results.

Table of Contents

List of Figures

1. Introduction

The KARYON project (Kernel-based ARchitecture for safetY-critical cONtrol) focuses on the predictable and safe coordination of smart vehicles that autonomously cooperate and interact in an open and inherently uncertain environment. For example, cooperating on common driving tasks such as cooperative road trains, lane change, and virtual traffic lights, KARYON aims to provide safer, greener, and more efficient transportation. Similarly, KARYON provides a platform allowing UAVs to interact and cooperate on common tasks such surveillance of natural disasters.

1.1 Motivation and Background

Vehicular systems, both in automotive and aviation, are inherently safety critical. Any of these safety critical components is required to be highly fault-tolerant in operation. A vehicle comprises numerous mechanical, hydraulic, software and hardware components as sub-systems. And throughout the recent years we saw an increasing amount of embedded software and hardware in vehicular systems. A modern vehicle contains up-to 100 embedded, microprocessor-based electronic control units (ECUs) and close to 100 million lines of software code [1] . The on-going development towards autonomous and, as a next step, cooperative vehicles (as focused on in this project), will increase the numbers of ECUs as well code complexity in the coming years.

All safety critical systems, ranging from advanced convenience to safety features, must be designed to be fault tolerant. Thus, they must be able tolerate faults in order to ensure the safety of the vehicle, its drivers and passengers, as well as the surroundings including other vehicles and pedestrians.

Commonly, a thorough safety analysis is conducted systematically during the design phase of a vehicle. Thus, well before the vehicle is put into operation, an offline analysis is conducted to evaluate and analyse the impact and likelihood of possible faults and their consequences. The goal is to identify faults that can lead to undesirable consequences and implement the required counter measures to ensure safety. Commonly, such counter measures are either fail-safe or fail-operational: Fail-safe denotes that a system shuts-down into a safe state after a failure is detected, i.e., it stops being available while not causing any harmful, undesired consequences. In contrast, fail-operational denotes that is system continues to provide a certain, often degraded, level of service in presence of failure while still providing the required safety.

Independent of the counter measures taken in the presence of failures, a key requirement is that a system must be able to detect that a failure is present in the first place. Moreover, this detection must deterministically happen within a specified amount of time (denoted as detection latency). This upper bound on the latency is required to ensure that the vehicle (or individual components) can safely transition from normal operation to a fail-safe or fail-operational stage. This is a key requirement, as vehicles need to maintain a safe behaviour also during this transitional phase. Additionally, not all faults can be detected by the system. The term detection coverage denotes the faults that can be detected by the system and for which fail-safe or fail-operational mitigation strategies are implemented.

In KARYON, the safety kernel supervises the failure states of the individual components. It controls the overall level of service that a vehicle can provide based on the status of its own components and the status of nearby vehicles. Thus, in this aspect, KARYON and its safety kernel strongly differ from the state of the art, where safety decisions merely depend on the status of local components. KARYON, in contrast, provides cooperative services and thus decisions and actions inherently require consensus on planned activities between vehicles. Thus,

the failure of a component on one vehicle directly impacts the status and quality of the information and strategies a vehicle receives from other vehicles.

1.2 Purpose and Scope

The cooperative nature of KARYON is a fundamental shift when compared to today's vehicles that each operate on their own. Due to its cooperative nature, KARYON demands for new approaches for diagnosing the cooperation and interaction of smart vehicles: As vehicles in KARYON interact and communicate to agree on actions to execute in consensus, any system to diagnose and debug the well being of the deployed system must inherently cover all participating units. Thus, we cannot employ traditional diagnostics systems that commonly track a number of components in one vehicle.

To debug distributed and cooperative functionalities we require new approaches to diagnostics. We address these challenge, with a new, distributed diagnostics system. Its key contribution is to provide a global, unified view on the individual components of the vehicles of interest even in presence of

- failure of components or parts of them, and
- failure of the wireless communication between vehicles.

Our architecture for distributed diagnostics shall be readily available as a tool to collect the global view from individual components. Collecting traces from each component of interest and streaming these out, it provides

- an online view on the components of interest across multiple vehicles, and
- the cycle-accurate replay of the traces after collecting them from multiple vehicles.

1.3 Relation to Other Work

Debugging large-scale distributed systems has received significant attention in the recent years with the raise of cloud computing and peer-to-peer networking. A common approach is to collect traces of events and to use their logical relationship in the system to build globally consistent snapshots and to enable replay [2] [3] [4] [5] . However, these mainly target Internet based applications and their resource requirements make them not suited for resource constrained, embedded systems such as ECUs in vehicles. Nonetheless, their design motivated our work and we carefully designed MILD to adapt them for the use in resource constrained, embedded systems and to ensure minimal intrusion. Others [6] [7] reflect the resource constraints of sensor networks but do not focus on trace-driven replay. In this context, some approaches [6] log each functional call and its parameters to provide the user with a complete view on the system. In this work, we argue that tracing all the function calls is costly and we show that it is not necessary to trace them entirely if the state of a node at a given time can be restored and replayed deterministically.

2. Design Overview and Challenges

Due to their safety critical nature, Cyber-Physical Systems such as collaborative cars or smart grids demand for thorough testing and evaluation. However, diagnosing such systems once deployed is challenging, due to (1) the interactive and concurrent nature of distributed systems and (2) the limited insight that any deployed system offers.

Addressing these challenges, we introduce MILD for Minimal Intrusive Logging and Deterministic replay of deployed Cyber-Physical Systems. MILD enables (1) logging of events with minimal intrusion of deployed systems, and (2) the deterministic, cycle accurate replay of events in controlled environments such as test-beds and system simulators. As a result, MILD offers deep insights into real-world deployments and allows diagnostics and testing in realistic settings.

2.1 Design Challenges

Diagnosing distributed systems is challenging due to two key reasons:

1. **Interactive and concurrent nature of distributed systems**: In KARYON, a set of cooperative vehicles interacts over wireless communication channels to achieve consensus on the next actions to execute. Each vehicle senses, processes information, communicates, and actuates driven by its computing infrastructure. In parallel, the other vehicles do the same. Thus, it prohibitively difficult to examine the state of one individual vehicle without impacting the other vehicles. Moreover, as a distributed system, it is mandatory to evaluate all vehicles participating in a cooperative activity, as only a global view on the distributed system can provide the required insights.
2. **Limited insight offered by deployed systems:** Any deployed system is difficult to diagnose, as diagnostics often require physical access to the system of interest. Moreover, diagnostics should not impact the normal operations of a vehicle. In distributed systems, such as cooperative vehicles, this is even more challenging: for diagnostics we require a global view on all participating units. Thus, we need to be able to extract information from all participating vehicles simultaneously, even in the presence of failed components and unreliable wireless communication.

After detailing on the key challenges, we next give an overview on MILD, our architecture addressing these challenges.

2.2 Design Overview

Addressing the above challenges, we develop MILD, an architecture providing Minimal Intrusive Logging and Deterministic replay. MILD enables debugging of cooperative vehicles by (1) logging of events on deployed Cyber-Physical Systems, such as cooperative vehicles, and (2) their deterministic and cycle-accurate replay in controlled environments such as test-beds and system simulators (see Figure 1).

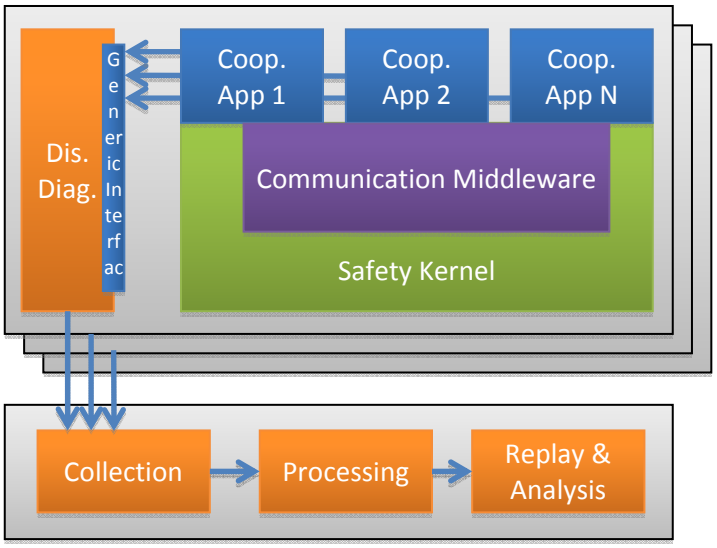

Figure 1: System Overview of Distributed Debugging with MILD. We depict MILD connected to three cooperative applications, which run on top of the safety kernel and communication middleware. Additionally, we depict the data handling pipeline of MILD (lower part) for collection, processing and replay / analysis of traces.

1. **Minimal intrusive tracing and logging with logical clocks:** We trace and log incoming and outgoing events on each component of interest. Amending each event with a logical timestamp (and local timestamp when required), we can later replay execution in a cycle accurate manner even in presence of timer failures or limited connectivity of the wireless coordination system. To avoid any impact of the logging on the safety of the system, the small run-time overhead of our logging can either be included in the safety and real-time analysis or the logging can be executed by dedicated hardware.
2. **Consistent Global View based on Traces:** Utilizing the logical timestamps of each recorded event, we next order the traces of all components into a consistent global view. We can either do this in real-time, by collecting a feed of the events from each components or off-line after traces from all components have been collected.
3. **Deterministic Replay of Traces:** Utilizing the ordered trace, we feed it into either a test-bed or a system simulator. Such a cycle-accurate replay in a controlled setting allows us to stop the replay were required and examine the state of individual components without impact on the overall execution. This strongly simplifies detecting the root cause of any failure or bug. Please note, that for two independent events that happen in parallel, for example, on two different nodes, MILD cannot distinguish which one happens first. However, as these events are independent and hence did not impact each other, the order of their replay does not impact the overall result.

To illustrate the feasibility and low overhead of our architecture, we present a prototype implementation of MILD and discuss initial evaluation results. We show our initial results on wireless sensor networks, as their embedded nature and wireless communication strongly mimics the requirements of cooperative vehicles. Also, for these have publicly accessible, large-scale real-world test-beds available, ranging up to 400 wireless nodes [8] [9] [10] . Nonetheless, the design and implementation of MILD is generic and can be readily integrated into other platforms including AUTOSAR [11] .

3. Architecture

After discussing challenges for distributed diagnostics and the underlying design idea of MILD in the previous section, we now detail on the its architecture. It consists of three building blocks (see Figure 2):

1. A logging element on each node: For each system component of interest, this logging element is responsible for the minimal intrusive tracing and logging with logical clocks.
2. A collection system that collects and combines the traces to a consistent, global view. This system can either be operated in real-time, assuming that a connection to each logging element is available through which the traces can be collected. Alternatively, the system can operate "off-line" once the traces from all nodes have been collected.
3. A replay environment, which feeds the traces to each node in a test-bed or in a system simulator.

In the following we discuss each of them.

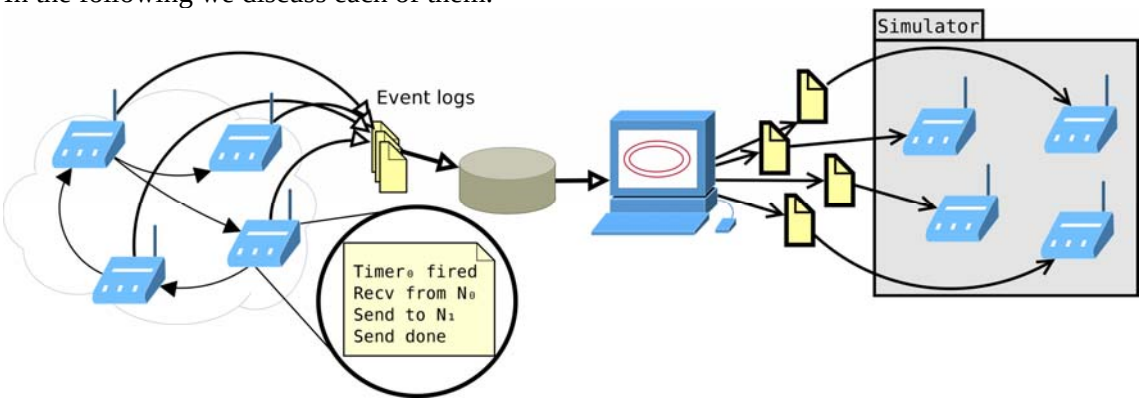

Figure 2: Architecture Overview: MILD consists of three key elements: (1) logging elements on the individual nodes (on the left); (2) data collection elements (in the middle); and (3) replay, e.g., with a system simulator (on the right).

3.1 Distributed Logging for Deterministic Replay

In MILD, all nodes are equipped with lightweight instrumentation to allow them to record events of interest. To limit the overhead, we only record the events that are of interest to a particular application. For example, when we are debugging a communication protocol, we log in- and outputs such as messages and function calls corresponding to communication. This information is sufficient to deterministically replay any code in a simulator or test-bed for debugging. Thus, we can recreate the exact program execution in a controlled environment, which allows for easy analysis for the program flow and detecting bugs: For example, we can step through the execution of a distributed system or evaluate the values of individual variables.

While MILD is designed for minimal intrusive logging, a certain overhead of the logging cannot be avoided. Thus, for hard real-time certain logging operations will be part of the time critical program execution. MILD addresses this with two options: (1) as the logging overhead is very limited, the additional CPU cycles of our logging can be directly included in any safety and deadline analysis and, as a result, become part of the verified design. (2) When requested, MILD can utilize dedicated hardware support for the logging to avoid any overhead. The KARYON project partner SP is working on such a hardware assisted monitoring and its integration into the safety kernel. Both design choices can also be combined on a per ECU level.

To realize our logging we rely on three key elements on a node. These can be either provided by hardware or in software and is transparent to the architecture of MILD.

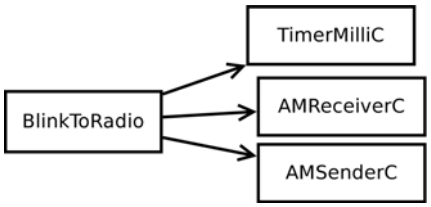

Figure 3: Sample application without logging elements. We depict a simple TinyOS application (named BlinkToRadio) that uses three resources: Timers, radio transmission, and radio receive modules.

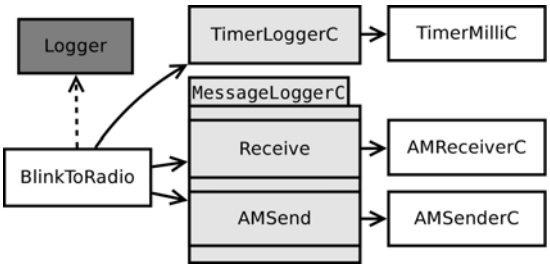

Figure 4: Sample application (same application as on the left) with logging enabled. We note that the software components of the application remain unmodified. MILD merely hooks into the points of interaction of the software components, e.g., function calls from one component to another.

- **Wrappers: Tracking In- and Output of Software Modules.**
  For each software module that shall be included in the logging and replay, we log all in- and output events such as messages or function calls (see Figure 4). Each event is coupled with a logical timestamp to ensure deterministic replay and to track interactions between nodes.
- **Central module: Minimal intrusive state collection.**
  All events are collected at in a central module. Acting as background task, it allows the collection of event traces via debugging ports at minimal intrusion. In our prototype implementation, we either log events out to the serial port where they are collected or we write them to flash storage. As a result, we do not utilize the main communication sub-system and thus avoid any side effects on it.
- **Initial State Collection.**
  To allow us to dynamically enable and disable logging, we store the initial states of a module, i.e., its variables, when enabling logging. Loading the initial states in the simulator and then replaying all input to one or more modules ensures deterministic replay.

3.2 Collection: Analysing and Sorting Logs

Once all events are collected from the individual nodes via their debugging ports, we utilize their logical timestamps to construct a globally ordered view on the system (see Figure 5). Since all the events carry a sequence number that is locally unique, sorting the local events of a node is immediate. Events such as radio events have (or can have) a received counterpart on the other nodes. These events are used to obtain a global order of events.

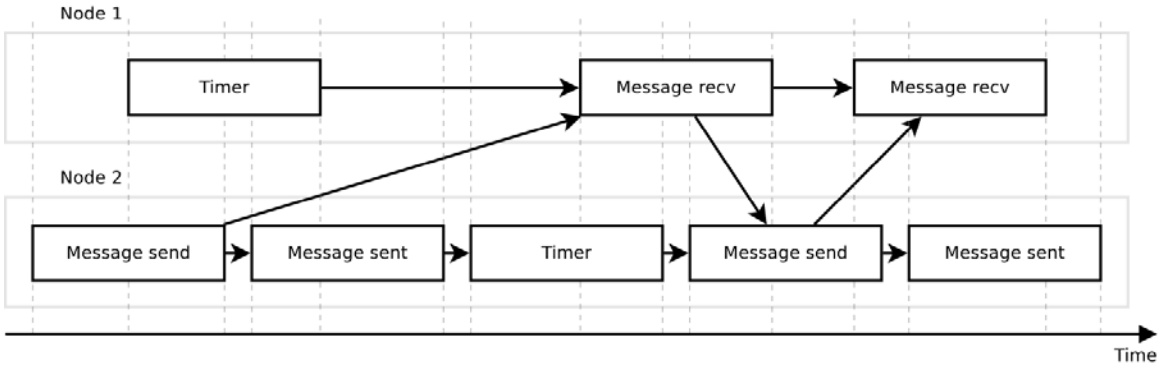

Figure 5: Dependency graph of the events traced on two nodes (based on logical clocks).

To obtain a partial global ordering of events, every event is treated as a node of a dependency graph. Events that are only local (e.g. timer events) depend on their predecessor in the event log of the node, while others can depend on events on nearby nodes. One event on a node can only be replayed if the events it depends on have been replayed as well. To generate the dependencies, events are scanned to find the matching events on other nodes. Lost sent messages don't have a corresponding receive event, so they are automatically considered as local events instead. Additionally, we use the recorded output to determine deviations from the replay, which indicate subtle system bugs such as buffer overflows, etc.

3.3 Deterministic Replay in System Simulation

The final element of MILD is the replay of logs in system simulators. In the replay, we use the wrappers (see Figure 4) in reverse operation: instead of logging all in- and out-going events of a software module, the wrappers now act as event sources. Thus, feeding from the global event log, the software module now replays the events in the same order as on the deployed system. Basing on modern system simulators, emulators, or testbeds, we provide cycle accurate replay of events. Thus, MILD can utilise the advanced debugging capabilities of modern system simulators and allows monitoring of individual variables and stepping through code fragments. Such tasks commonly cause prohibitively high overhead and side effects when performed on deployed systems directly.

4. Implementation and Evaluation

In this section we discuss implementation details, sketch on a first case study of how MILD can be used to detect common bugs in distributed systems, and present results from an initial performance evaluation.

4.1 Prototype Implementation

We implement our first prototype of MILD in TinyOS [12] , an operating system for Wireless Sensor Networks (WSNs). In our prototype we utilize the software driven solution for logging (see Section 3.1). This approach supports rapid prototyping and evaluation for two key reasons: (1) we do not rely on any dedicated hardware and can use off-shelf micro-controllers and (2) these off the shelf microcontroller are common in the testbeds we have access to.

As noted above, we utilize a Wireless Sensor Network for this initial evaluation, as its test-beds are readily available for large-scale testing. For example, we have access to multiple test-beds such as Twist [8] at TU Berlin, Germany, with 90 nodes, Indriya [9] at National University of Singapore, Singapore, with 140 nodes and KansaiGenie [10] at Ohio State University, Ohio, with about 400 nodes.

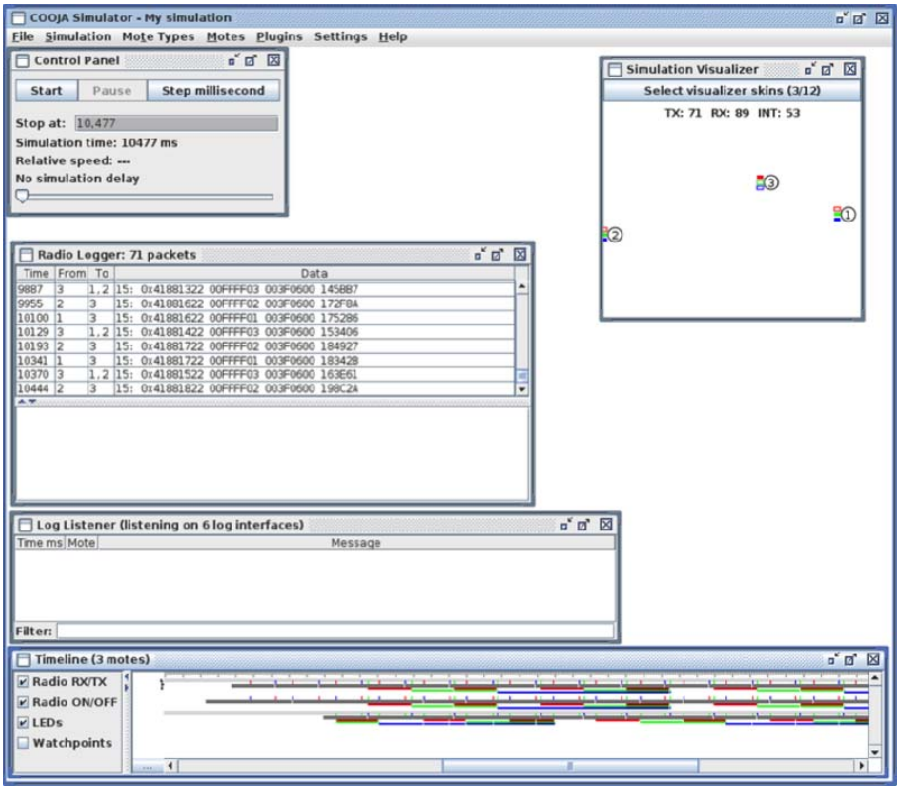

Figure 6: Example Screenshot of a sample replay environment: the full system simulator "Cooja"

4.2 First Case Study: Diagnosing Split-Phase Faults

Most long operations in TinyOS are implemented as split-phase, when, for example, a command to initialise a device is sent from the application to the lower layer, and then an event

is sent back to signal that the initialisation is complete. The authors of [6] use LEACH [13] 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 information 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.

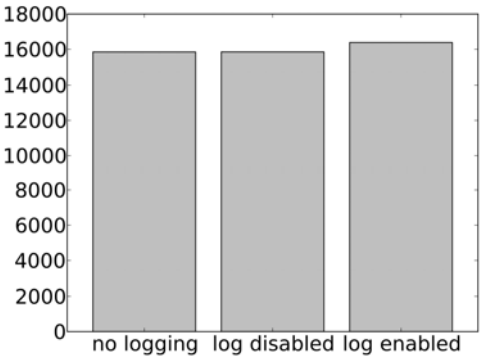

Figure 7: Initial performance evaluation: Comparison of the cycles needed by the application to send one message ("BlinkToRadio" application). Our results show a very limited logging overhead.

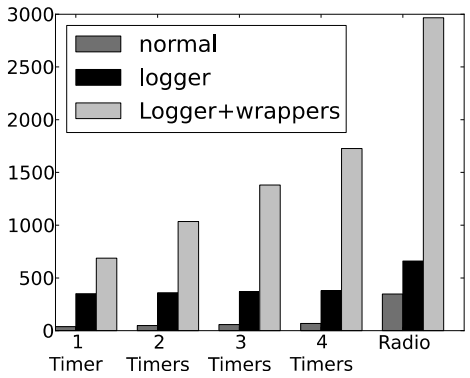

Figure 8: RAM usage as reported by the compiler for: the Blink application with 1,2,3,4 timers and for "RadioCountToLeds". Every wrapper can buffer 40 events.

4.3 Initial Evaluation: Performance and Memory Overhead

In this section we demonstrate that the extra cost added by our instrumentation is acceptable and brings a fixed cost to use the central module, which needs the serial stack, and a variable cost depending on the amount of wrappers used, that mostly depends on the size of the memory area for the log. About the speed, as shown in Figure 7 with logging disabled the overhead is almost zero (3 CPU cycles) and logging one event does not bring much overhead (around 500 cycles) comparing to the effort to generate the event itself. These 500 cycles includes both storing of the event and to schedule the logging message on the serial port. Please note that these results are initial results based on our preliminary prototype implementation. It is part of our on-going work to aim for optimizations here.

Concerning the memory, Figure 8 shows how allocating large space for logs in the wrappers brings a significant increase in memory usage, while the central module is not so heavy in itself.

5. Discussion

After introducing the design of MILD and evaluating its performance and overhead, we next reflect on our design choices and the performance results. Please note that the results discussed below are based on our initial prototype implementation. Our next step is to focus both on adding new functionally as well as improve the performance of the system design.

5.1 Benefits

The evaluation results underline five key benefits of MILD and its architecture:

- Lightweight Logging: Our design choice to buffer the traces and using a low-priority background task to collect them, results in a very small overhead in terms of MCU cycles during each event. This is key to unsure a minimal intrusive logging and tracing of events. As a result, MILD can record hundreds of events per second even on very small, embedded systems.
- Flexible Logging: The design of MILD allows developers to target their tracing to components of interest. Additionally, it allows to dynamically turn tracing on and off as well as to re-target tracing at run-time. Thus, this keeps both the logging overhead as well as its data stream limited and controllable. Moreover, it allows developers to flexibility adept tracing to new insights they learned during an on-going analysis without requiring physical access to any of the components.
- Deterministic Replay and Global View: Achieving a global, consistent view – as provided by MILD – onto a dynamic, distributed system such as cooperate vehicles is a key requirement for effective diagnostics and failure detection.
- Online and offline tracing: Depending on whether an online connection to each vehicle is available or not, tracing and collection with MILD can be done both online and offline. As this is transparent to the replay systems, a single system design provides both.
- Detecting timing faults: Basing on logical timestamps and not hardware timers, MILD can be also used to detect low-level failures such as of timer faults. Assume that a communication sub-system on one node suffers from a timing fault and, for example, sent out a message too late. In this case, this will trigger a timeout on another node, which is recorded by MILD just as the late message transmission (and its corresponding reception). Thus, when constructing the global view onto the system, we can trace and detect that the message was send too late, i.e., after the reception took place after the timeout.
- Detecting crashes: If a crash is caused by the code that we are logging, replaying the log that led to the crash will in most cases also trigger the crash in the test-bed or full-system simulation environment. As a result, we can track the execution that led to the bug in the simulation environment enabling us to detect its cause. If the bug is triggered from the outside, for example, by voltage fluctuation, or by code that we are not tracking, the replay in MILD detect that log and replay do not match and we signal this with an error message.
- Detecting value errors: In the replay of MILD, we can track any variable of interest. Thus, when enabling the logging in Mild we merely have to select which software components we are interested in. During the replay, we can then track any variable within these software components. This design as the following key benefit: The user does not have to select the variable during logging time. This is very practical, as an

error observed can often be connected to some software modules but not to individual variables. During the replay, the user flexibility selects variables of interests and then uses the powerful debugging utilities of modern system simulators to track values, define breakpoints, and assertions.

5.2 Limitations

Albeit MILD is designed to be minimal intrusive, any tracing inherently causes a certain overhead. Due its careful design, i.e., relying on buffering and a low-priority background process for collection, we efficiently limit this overhead to a couple of MCU cycles. Independent of any optimizations, when relying on a software-based solution, a certain overhead cannot be avoided, as we have to log the event. MILD addresses this with two options: (1) as the logging overhead is very limited, the additional CPU cycles of our logging can be directly included in any safety and deadline analysis and, as a result, become part of the verified design. (2) When requested, MILD can utilize dedicated hardware support for the logging to avoid any overhead. The KARYON project partner SP is working on such a hardware assisted monitoring.

6. Conclusions and Next Steps

In this report we introduced MILD, a lightweight architecture for minimal intrusive logging and deterministic replay of deployed Cyber-Physical Systems. MILD enables (1) logging of events with minimal intrusion of deployed systems, and (2) the deterministic and cycle-accurate replay of events in controlled environments such as system simulators and test-beds. As a result, MILD offers deep insights into real-world deployments and allows debugging and testing in realistic settings. We discuss the architecture of MILD and show initial results of our prototype implementation based on TinyOS, a widespread operating system for sensor networks. Future directions include optimization to further reduce logging overhead and the application of MILD in on-going deployments to gather real-world experience in utilizing MILD.

References

[1] R.N. Charette, "This Car Runs on Code", in IEEE Spectrum, Feb. 2009, http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code

[2] D. Geels, G, Altekar, S. Shenker, and I. Stoica: “Replay debugging for distributed applications”, in Proceedings of the Annual Conference on USENIX ’06 Annual Technical Conference. ATEC ’06 (2006)

[3] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, “Friday: global comprehension for distributed replay”, in Proceedings of the 4th USENIX conference on Networked systems design and implementation. NSDI’07 (2007)

[4] D. Dao, J. Albrecht, C. Killian, and A. 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)

[5] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M.F. Kaashoek, and Z. Zhang, “D3s: debugging deployed distributed systems”, in Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation. NSDI’08 (2008)

[6] V. Sundaram, P. Eugster, and X. Zhang, “Efficient diagnostic tracing for wireless sensor networks”, in Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems. SenSys ’10 (2010)

[7] 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 (200

[8] V. Handziski, A. Köpke, A. Willig, and A. Wolisz, “TWIST: a Scalable and Recongurable Testbed for Wireless Indoor Experiments with Sensor Networks”, in RealMAN: Proc. of the Int. Workshop on Multi-hop Ad Hoc Networks: from Theory to Reality, 2006.

[9] M. Doddavenkatappa, M. C. Chan, and A. Ananda. “Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed”, in TridentCom: Proc. of the ICST Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities, 2011.

[10] E. Ertin, A. Arora, R. Ramnath, V. Naik, S. Bapat, V. Kulathumani, M. Sridharan, H. Zhang, H. Cao, and M. Nesterenko. “Kansei: a testbed for sensing at scale”, in Proceedings of the 5th international conference on Information processing in sensor networks (IPSN '06), 2006

[11] H. Heinecke, K.P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.L. Maté, K. Nishikawa, T. Scharnhorst, “AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E Architectures”, in Convergence International Congress & Exposition On Transportation Electronics (2004)

[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, “Systemarchitecture directions for networked sensors”, ACM SIGOPS Operating Systems Rev. 34(5), 2000

[13] M.J. Handy, M. Haase, and D. Timmermann, "Low energy adaptive clustering hierarchy with deterministic cluster-head selection", 4th International Workshop on Mobile and Wireless Communications Networks, 2002

