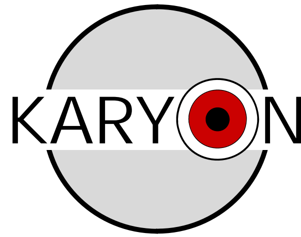

Kernel-based ARchitecture for safetY-critical cONtrol

KARYON

FP7-288195

D4.6 - Cooperative Diagnostics

Work Package	WP4		
Due Date	M37	Submission Date	2014-11-17
Main Author(s)	Olaf Landsiedel (CTHA)		
Contributors	Salvo Tomaselli (CTHA), Elad M. Schiller (CTHA)		
Version	1.0	Status	Final
Dissemination Level	Public	Nature	Report
Keywords	Distributed Diagnostics, Debugging Distributed Systems, Tracing, Deterministic Replay		
Reviewers	António Casimiro (FCUL), Elad M. Schiller (CTHA)		

Part of the Seventh

Framework Programme

Funded by the EC – DG INFSO

Version history

Rev	Date	Author	Comments
V0.1	2014-09-14	Olaf Landsiedel (CTHA)	Initial Structure
V0.2	2014-09-29	Olaf Landsiedel (CTHA)	First draft
V0.2	2014-10-14	Olaf Landsiedel (CTHA)	Text complete (missing: clean-up, figures)
V0.3	2014-10-16	Olaf Landsiedel (CTHA)	Clean-up: Figures, references, citations
V0.4	2014-10-17	Olaf Landsiedel (CTHA)	Review
V0.5	2014-10-30	Olaf Landsiedel (CTHA)	Review II
V1.0	2014-11-04	António Casimiro (FCUL)	Final review
V1.1	2014-11-17	António Casimiro (FCUL)	Small update and delivery.

© 2014 KARYON Project

Glossary of Acronyms

AUTOSAR	AUTomotive Open System Architecture
CPS	Cyber Physical System
CPU	Central Processing Unit
ECU	Engine Control Unit
IoT	Internet of Things
KARYON	Kernel-based ARchitecture for safetY-critical cONtrol
MCU	Microcontroller Unit
LibReplay	Library for Logging and Replay of distributed, embedded applications
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

© 2014 KARYON Project

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 LibReplay; providing lightweight, distributed logging and deterministic replay. LibReplay 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 LibReplay, discuss the underlying motivations for its design, and present our research-prototype implementation. Additionally, we report insights into its flexibility and discuss performance results.

Editor note: All marked text (in yellow) corresponds to text that has been left unchanged with respect to the preliminary version of this deliverable (D4.3 – First report on Cooperative Diagnostics).

© 2014 KARYON Project

Table of Contents

1. Introduction ... 8
1.1 Motivation and Background ... 8
1.2 Purpose and Scope ... 9
1.3 Relation to Other Work ... 10
2. LibReplay Design Overview and Challenges ... 11
2.1 Design Challenges ... 11
2.2 Design Overview ... 12
3. Architecture ... 14
3.1 Distributed Logging for Deterministic Replay ... 14
3.2 Collection: Analysing and Sorting Logs ... 16
3.3 Deterministic Replay in System Simulation ... 17
4. Implementation and Evaluation ... 18
4.1 Prototype Implementation ... 18
4.2 Example: Diagnosing Split-Phase Faults ... 19
4.3 MCU and Memory Efficiency of LibReplay ... 19
4.4 LibReplay and Traditional Approaches to Logging ... 20
5. Discussion ... 21
5.1 Benefits ... 21
5.2 Limitations ... 22
6. Conclusions and Next Steps ... 23
References ... 24

© 2014 KARYON Project

List of Figures

Figure 1: System Overview of Distributed Debugging with LibReplay. We depict LibReplay connected to three cooperative applications, which run on top of the safety kernel and communication middleware. Additionally, we depict the data handling pipeline of LibReplay (lower part) for collection, processing and replay / analysis of traces. 12
Figure 2: Architecture Overview: LibReplay consists of three key elements: (1) logging elements on the individual nodes (on the left); (2) processing (in the middle); and (3) replay, e.g., with a system simulator (on the right)... 14
Figure 3: We log function calls to and from the code of interest, such as a malfunctioning routing protocol. For replay, we feed the logs back to the code of interest. Replay in a full-system simulator provides us with well-established debugging tools such as stepping through code, breakpoints and watchpoints. 15
Figure 4: Sample application without logging elements. We depict a simple TinyOS application (named `BlinkToRadio`) that uses three resources: Timers, radio transmission, and radio receive modules. 16
Figure 5: Sample application (same application as on the left) with logging enabled. We note that the software components of the application remain unmodified. LibReplay merely hooks into the points of interaction of the software components, e.g., function calls from one component to another. 16
Figure 6: Dependency graph of the events traced on two nodes (based on logical clocks)...... 17
Figure 7: Example Screenshot of a sample replay environment: the full system simulator "Cooja". 18
Figure 8: Logging has only small impact on program execution. A low-priority background process transfers log to the storage, e.g., the serial. 19
Figure 9: The RAM footprint of LibReplay mainly depends on the size of the logging buffers. ROM remains constant independent of buffer size. 19
Figure 10: The overall memory footprint of LibReplay is small when compared to the application itself (default setting, 300 bytes buffer)... 20
Figure 11: The memory footprint of LibReplay is similar to traditional logging systems. The footprint of TinyLTS is taken from its publication [8] , as the source code is not available to us. 20
Figure 12: Average MCU duty-cycle in a CTP network of 25 nodes. We distinguish leaf nodes and forwarders. For LibReplay we also distinguish between logging and the background (BG) process. 20

List of Tables

Table 1: LibReplay logging example: Without (left) and with (right) logging of the Receive interface. Adding logging to applications requires merely few changes to the wiring of TinyOS applications. LibReplay provides common logging components, such as the ReceiveLogC component used in this example to log the Receive interface. 15

© 2014 KARYON Project

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.

Diagnostics in any distributed Cyber-Physical System is challenging: Failures are often prompted by a particular, complex concatenation of events. Moreover, dynamic interactions between individual nodes and with the environment make it time-consuming to track and reproduce a fault. We introduce LibReplay to ease diagnostics in distributed Cyber-Physical Systems; it provides:

1. lightweight and flexible logging and
2. deterministic replay.

LibReplay logs function calls to and from the application or another code of interest. It enables deterministic replay of execution traces a controlled environment such as a full-system simulator. This allows the user to benefit from well- established debugging tools such as stepping through code, breakpoints, or watchpoints.

In this report, we discuss the refined design and architecture of LibReplay. In our previous report "D4.3 – First report on Cooperative Diagnostics" we discussed our initial results. Thus, while motivations and goals are consistent with the previous report, we significantly deepen our discussion of the architectural design and evaluation.

1.1 Motivation and Background

Vehicular systems, both in automotive and aviation, are inherently safety critical. Any of their 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 significantly increase both the numbers of ECUs and the 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 a system continues to provide a certain, often degraded, level of service in presence of failure while still providing the required safety.

© 2014 KARYON Project

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 this 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, denoted LibReplay. 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.

Diagnostics in distributed CPSs is challenging:

1. CPS are distributed and deeply embedded into a non-deterministic environment.
2. The non-determinism of both the wireless network and the physical environment in which the nodes are embedded makes it time-consuming to track and reproduce faults and bugs.
3. These are often prompted by a particular, complex concatenation of events.

Source-level debugging capabilities as we are used to in sequential programming would significantly ease the debugging process. However, the distributed and embedded nature prevents us from pausing program execution on a node to examine its state. Large-scale distributed systems on the Internet solve this issue by employing logging and replay capabilities. These log all interaction between the code of interest and the system itself, e.g., function calls to and from a part of an application that is suspected to malfunction. Next, they replay the execution of the code of interest accordingly to the logged function calls and their parameters. As a result, the local replay can be debugged using well-established debugging tools such as GDB and allows for stepping through code, breakpoints, and watchpoints. While this technique is well known in large-scale distributed Systems, we see limited application in the area of Cyber Physical Systems due to the resource limitations of embedded and networked devices.

© 2014 KARYON Project

This work closes this gap and provides LibReplay: lightweight, distributed logging and deterministic, source-level replay. LibReplay allows diagnosing of distributed CPS applications and protocols with source-level debuggers. We achieve this by replaying execution traces in a full-system simulator or testbed.

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 LibReplay to adapt them for the use in resource constrained, embedded systems and to ensure minimal intrusion.

In the context of distributed applications in resource constrained, embedded systems logging and tracing are two common approaches for diagnostics. Logging tools [6] [7] [8] record execution details. Commonly, they store the log in the flash memory for off-line collection or feed them to a host system, for example, via the serial port. In practice, diagnostics and bug hunting with such logging tools often follows an iterative approach: (a) adding or refining logging statements, (b) re-executing the system until the bug is triggered, and (c) analysing the log and spotting bug appearances. The developers have to repeat these steps until they understand the bug causes, try to fix them and then check whether all bugs were removed by again repeating these steps. Moreover, the non-deterministic and dynamic nature of the wireless network and interactions with the physical environment make it time consuming to reproduce a bug sufficiently often for this repetitive approach. In contrast, LibReplay logs function calls and their parameters to and from the code of interest, such as a malfunctioning routing protocol. As a result, LibReplay collects sufficient information to replay program execution deterministically allowing one to employ source-level debuggers for bug hunting. In our experience, this limits the need for repeated testing, and in most cases a single logging run is sufficient to fix the bug in replay debugging, because much of log analyses and bug spotting is carried out off-line using an iterative debugger that replays the log.

Tracing tools [9] [10] [11] [12] [13] follow a different approach: They trace the program execution by logging function calls. For example, tracing logs each function and its parameters that a packet takes on its path through the protocol stack from the application to the radio driver. A key challenge is that tracing program execution leads to large traces when compared to traditional logging [14] . Some approaches [11] [13] address this challenge with additional hardware on the nodes. For example, Minerva [11] connects a dedicated debugging-board to the JTAG adapter of the sensor node. Controlling multiple debugging-boards over Ethernet, Minerva can examine network-wide state. LibReplay, in contrast, merely logs function calls and their parameters to and from the code of interest, limiting its intrusiveness while not requiring additional hardware. 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.

© 2014 KARYON Project

2. LibReplay 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 LibReplay for lightweight, distributed logging and deterministic replay of deployed Cyber-Physical Systems. LibReplay enables:

- **Lightweight, distributed logging and**
- **Deterministic, source-level replay**

As a key contribution, LibReplay enables debugging of deployed, distributed applications and protocols with source-level debuggers, such as GDB [15] . We achieve this by replaying execution traces cycle-accurate and deterministically in controlled environments such as test-beds and system simulators, e.g., Cooja [16] , MspSim [17] Avrora[18] , or QEMU [19] . As a result, LibReplay offers deep insights into real-world deployments and allows diagnostics and testing in realistic settings.

2.1 Design Challenges

Diagnosing distributed, cyber physical systems is challenging due to three key reasons:

1. They are **inherently distributed and deeply** embedded into a **non-deterministic environment**.
2. The **non-determinism of both the wireless network and the physical environment** in which the nodes are embedded makes it time-consuming to track and reproduce bugs.
3. Bugs are often prompted by a **particular, complex concatenation of events**.

For example, 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 is 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.

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.

Source-level debugging capabilities as we are used to in sequential programming, i.e., local and non-distributed applications, would significantly ease the debugging process. For example, stepping through code, breakpoints, and watchpoints are well-established tools to debug sequential code. However, the above-discussed distributed and embedded nature prevents us from pausing program execution on a node to examine its state. After detailing on the key challenges, we next give an overview on LibReplay, our architecture addressing these challenges.

© 2014 KARYON Project

2.2 Design Overview

Addressing the above challenges, we develop LibReplay, an architecture providing lightweight, distributed logging and deterministic, source-level replay. It (1) logs events on deployed Cyber-Physical Systems, such as cooperative vehicles, and (2) allows their deterministic and cycle-accurate replay in controlled environments such as test-beds and system simulators.

Thus, LibReplay allows debugging of deployed, distributed applications and protocols in Cyber Physical Systems with source-level debuggers, such as GDB. We achieve this by replaying execution traces in a full-system simulator.

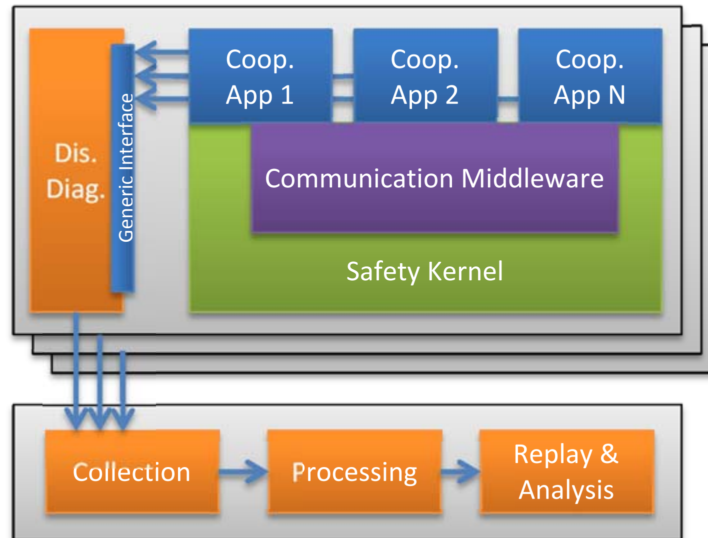

Figure 1: System Overview of Distributed Debugging with LibReplay. We depict LibReplay connected to three cooperative applications, which run on top of the safety kernel and communication middleware. Additionally, we depict the data handling pipeline of LibReplay (lower part) for collection, processing 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. Moreover, it employs a two-phase logging to minimize the side effects of logging on program execution. We log all calls to and from the code region of interest and their timestamps. LibReplay buffers log data in RAM to avoid the overhead during the logging process. Later a deferred background-process transfers the buffers to flash or the serial port for storage.
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 (near) 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 where 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, LibReplay 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 LibReplay and discuss evaluation results. We show our 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 [20] [21] [22] . Nonetheless, the design and implementation of LibReplay is generic and can be readily integrated into other platforms such as AUTOSAR [23] .

© 2014 KARYON Project

3. Architecture

After discussing challenges for distributed diagnostics and the underlying design idea of LibReplay in the previous section, we now detail on the its architecture. With LibReplay, we log function calls to and from a user-specified code-region of interest, such as a malfunctioning routing protocol. In the replay, we feed the calls back to the code of interest in the same order as they were logged on the real system (see Figure 2). Thus, in the replay every event happens in the same order as on the real system and with the same function parameters. Using cycle-accurate simulation of the entire system, each event will also take the same number of cycles as on the real system. Thus, a complete log that includes all functions to the code of interest generates a complete replay with all local states equaling to the ones of the real-system. Moreover, LibReplay inherently also intercepts calls from the code of interest to the internal clock system. Thus, any call to read the local clock results in us replaying the same value the code read in on the real-system (and not the local time of the simulation). We note that due to the run-to-completion semantics, e.g., tasks in TinyOS, of many OSs for CPS and IoT we do not have to log the OS scheduler itself. Moreover, LibReplay does not focus on low-level parts of the system that interact with interrupts such as device drivers.

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.

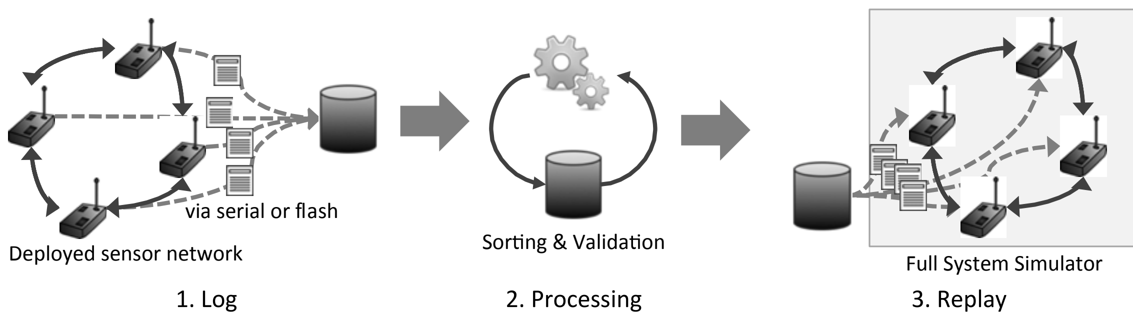

Figure 2: Architecture Overview: LibReplay consists of three key elements: (1) logging elements on the individual nodes (on the left); (2) processing (in the middle); and (3) replay, e.g., with a system simulator (on the right).

3.1 Distributed Logging for Deterministic Replay

The first building block of LibReplay is its lightweight, flexible logging-architecture. Thus, 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. Overall, it has three design goals:

© 2014 KARYON Project

- To reduce the overhead of logging to limit potential side effects on program execution.
- To provide distributed logging of events across multiple nodes.
- To ease integration into user-defined applications and components.

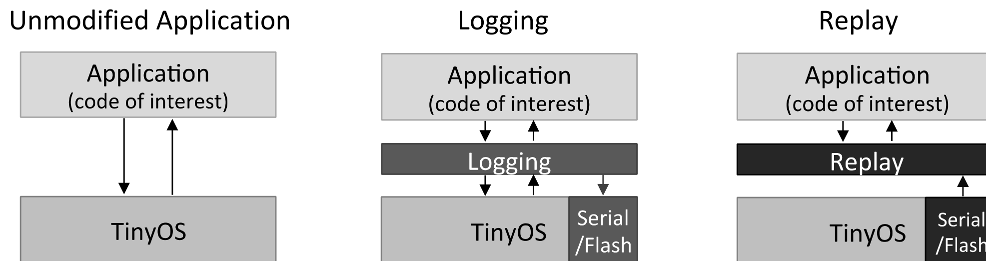

Figure 3: We log function calls to and from the code of interest, such as a malfunctioning routing protocol. For replay, we feed the logs back to the code of interest. Replay in a full-system simulator provides us with well-established debugging tools such as stepping through code, breakpoints and watchpoints.

Deferred logging to limit side effects on applications: Whenever a function to or from the code-region of interest is called, LibReplay logs the function, its parameters, the return value, and a logical timestamp, i.e., an event sequence-number. To limit run-time overhead, LibReplay employs a two-phase approach to logging: As a first step, any log data is merely buffered in RAM and the execution can continue with only minimal delay. As second step, a deferred, background process -- only scheduled if no other process is to be scheduled -- handles the storage itself: it moves the log buffers to flash or the serial port for storage.

Distributed logging of events across multiple nodes: When testing and debugging distributed systems, we experienced it as essential that we can trace events and messages across multiple nodes. For example, we often needed to trace how a single message travels through the network and which state changes it triggered along its path, such as timeouts and re-transmissions. To trace events across multiple nodes, LibReplay adds a logical timestamp to each outgoing radio message, which is send by the code of interest. Optionally, LibReplay can also re-use sequence numbers and source addresses that most protocols already provide to identify messages and their order uniquely. This avoids overhead, as no additional timestamps need to be added to messages.

Easy to integrate into user-defined applications and components: When designing LibReplay, we put a special focus on its ease and flexibility of use. For example, LibReplay can be easily integrated into own applications and tailored by adding own logging interfaces. LibReplay places a logging component between each interface of the code of interest and the OS (see Table 1). LibReplay provides logging components for common interfaces of TinyOS, such as `ReceiveLogC` (see Table 1). It logs the data flow to and from the `Receive` interface. Logging components are kept simple to reduce run-time complexity and to ease their adaption to user-defined interfaces.

Without Logging	With Logging

`[...]`
 	
`App.Receive -> AMReceiverC;`
 	
`[...]`
	`[...]`
	`components new ReceiveLogC() as Log;`
	`App.Receive -> Log;`
	`Log.Receive -> AMReceiverC;`
	`[...]`

Table 1: LibReplay logging example: Without (left) and with (right) logging of the `Receive` interface. Adding logging to applications requires merely few changes to the wiring of TinyOS applications. LibReplay provides common logging components, such as the `ReceiveLogC` component used in this example to log the `Receive` interface.

© 2014 KARYON Project

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 LibReplay 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. LibReplay 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, LibReplay 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 LibReplay.

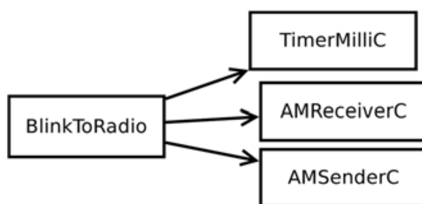

Figure 4: Sample application without logging elements. We depict a simple TinyOS application (named `BlinkToRadio`) that uses three resources: Timers, radio transmission, and radio receive modules.

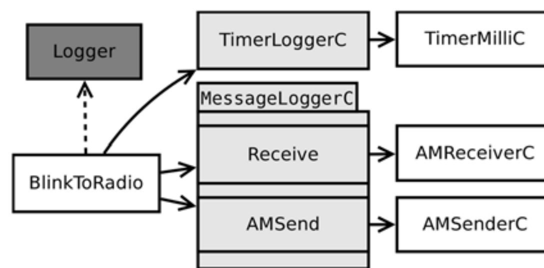

Figure 5: Sample application (same application as on the left) with logging enabled. We note that the software components of the application remain unmodified. LibReplay merely hooks into the points of interaction of the software components, e.g., function calls from one component to another.

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 6). 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.

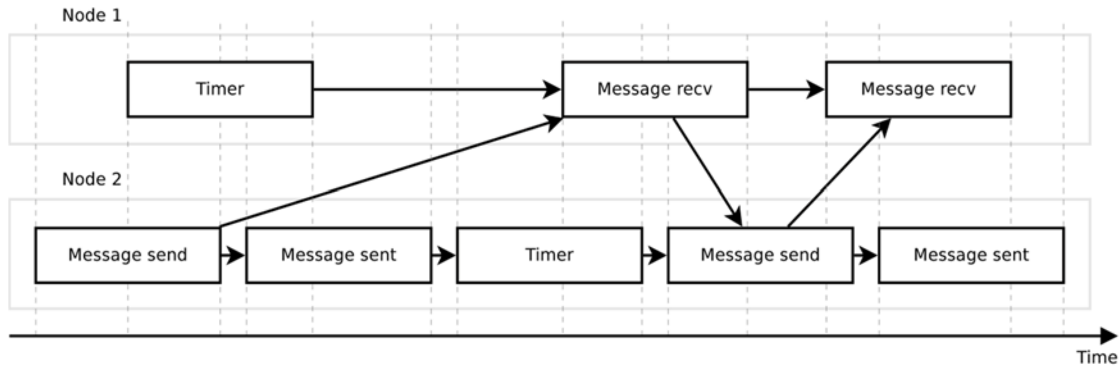

Figure 6: Dependency graph of the events traced on two nodes (based on logical clocks).

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 LibReplay is the replay of logs in system simulators. For replay, it replaces each logging component with its counterpart replay-component. Similar to the logging components, we have one replay component per interface and LibReplay provides these for the common interfaces in TinyOS. Compared to the logging components, the data flow is now reversed: Replay components feed events to the application (see Figure 3: We log function calls to and from the code of interest, such as a malfunctioning routing protocol. For replay, we feed the logs back to the code of interest. Replay in a full-system simulator provides us with well-established debugging tools such as stepping through code, breakpoints and watchpoints.

). Utilizing the advanced debugging capabilities of modern system simulators that allow monitoring of individual variables and stepping through code fragments. Note that when performing such tasks on the deployed systems directly, they cause high overhead and significant side effects. Additionally, we use the recorded output to detect deviations between the log and the replay, which can indicate subtle system bugs such as buffer overflows, etc. Note that the main replay-target of LibReplay are full-system simulators, as these can replay multiple nodes, and we can analyse their interaction. However, LibReplay can also replay the execution on a real node and we can connect and debug via JTAG, for example.

4. Implementation and Evaluation

In this section we discuss implementation details, sketch on a first case study of how LibReplay can be used to detect common bugs in distributed systems, and present results from our performance evaluation. We begin with a set of micro benchmarks to determine MCU and memory efficiency. Next, we compare LibReplay to the state of art and show that its overhead is similar to today's approaches to logging while these commonly do log sufficient information to provide replay capabilities.

4.1 Prototype Implementation

We implement LibReplay in TinyOS 2.1.2 [24] , an operating system for Wireless Sensor Networks (WSNs) and evaluate using TelosB sensor nodes. In our research 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 [11] at TU Berlin, Germany, with 90 nodes, Indriya [21] at National University of Singapore, Singapore, with 140 nodes and KansaiGenie [22] at Ohio State University, Ohio, with about 400 nodes.

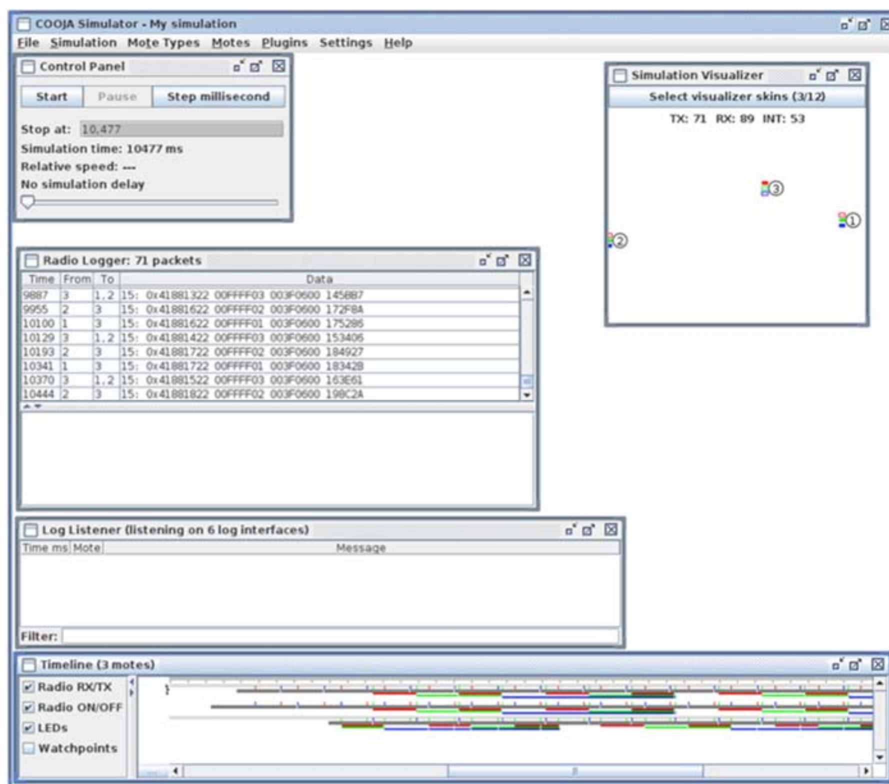

Figure 7: Example Screenshot of a sample replay environment: the full system simulator "Cooja".

© 2014 KARYON Project

4.2 Example: 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 [9] use LEACH [25] 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.

4.3 MCU and Memory Efficiency of LibReplay

In LibReplay, logging consists of two steps: the fast logging itself to an in-memory buffer and a second low-priority background process that handles the heavy lifting to external storage. As a result, the logging itself has only minimal impact on the program execution (see Figure 8). The RAM footprint of LibReplay strongly depends on the buffer size chosen (see Figure 9). ROM is stable independent of the buffer size chosen. For the following, we use the default value of 300 bytes for the buffer. Our experience shows that this is sufficient for most application scenarios, and it is similar to the default setting in the state of the art. Nonetheless, when compared to the overall memory footprint of the application, the footprint of LibReplay stays small (see Figure 10) leaving sufficient space for complex applications.

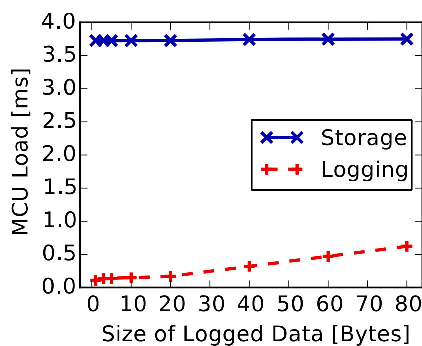

Figure 8: Logging has only small impact on program execution. A low-priority background process transfers log to the storage, e.g., the serial.

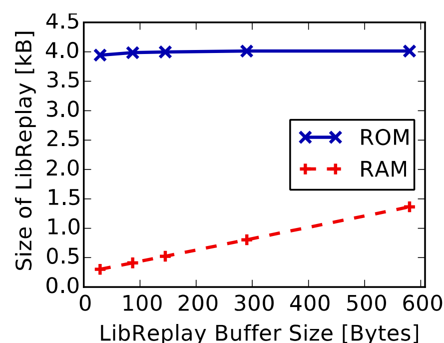

Figure 9: The RAM footprint of LibReplay mainly depends on the size of the logging buffers. ROM remains constant independent of buffer size.

© 2014 KARYON Project

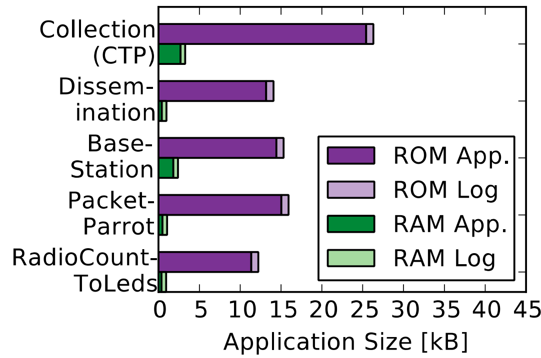

Figure 10: The overall memory footprint of LibReplay is small when compared to the application itself (default setting, 300 bytes buffer).

4.4 LibReplay and Traditional Approaches to Logging

We compare the efficiency of LibReplay to traditional logging approaches: *printf*, *TinyLTS [8]* , and the customized logging layer of the Collection Tree Protocol (CTP) [7] . To our best knowledge, the source code of TinyLTS has not been released. Thus, we rely, where appropriate, on the numbers published in the corresponding paper. Our results show that both the memory footprint and the MCU load of logging with LibReplay is comparable to these traditional approaches to logging (see Figure 11 and Figure 12). We note that these, in contrast to LibReplay commonly do not log sufficient information to enable replay debugging.

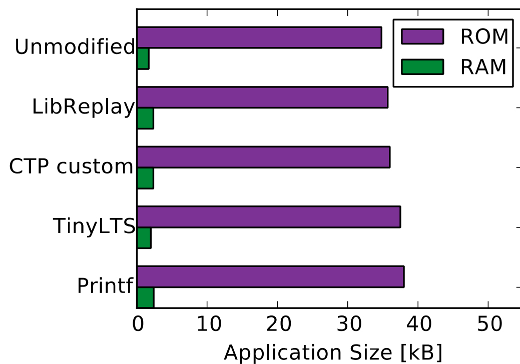

Figure 11: The memory footprint of LibReplay is similar to traditional logging systems. The footprint of TinyLTS is taken from its publication [8] , as the source code is not available to us.

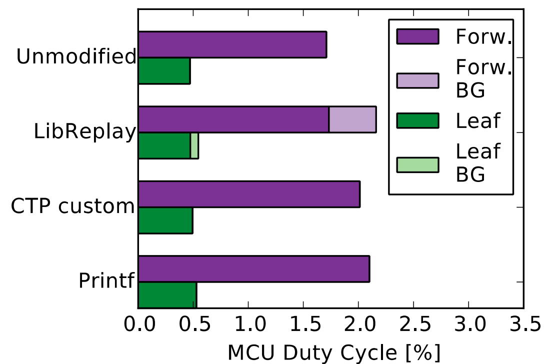

Figure 12: Average MCU duty-cycle in a CTP network of 25 nodes. We distinguish leaf nodes and forwarders. For LibReplay we also distinguish between logging and the background (BG) process.

5. Discussion

After introducing the design of LibReplay 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 key benefits of LibReplay 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, LibReplay can record hundreds of events per second even on very small, embedded systems.
- Flexible Logging: The design of LibReplay 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 LibReplay – 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 LibReplay 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, LibReplay 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 LibReplay 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 LibReplay detect that log and replay do not match and we signal this with an error message.
- Detecting value errors: In the replay of LibReplay, we can track any variable of interest. Thus, when enabling the logging in LibReplay 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.

- In TinyOS, modules are the natural integration points for logging. They encapsulate local state, and state changes are only triggered via their interfaces. Nonetheless, the design of LibReplay is generic and is not bound to TinyOS. For example, instead of interfaces we can log traditional function calls to and from a block of code. This, for example, matches the design of other common OS in CPS and IoT such as AUTOSAR [23] , Contiki [26] or FreeRTOS.

5.2 Limitations

Albeit LibReplay 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. LibReplay 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, LibReplay can utilize dedicated hardware support for the logging to avoid any overhead, similar to related approaches [11] [13] . Also, the KARYON project partner SP is working on such hardware assisted monitoring.

© 2014 KARYON Project

6. Conclusions and Next Steps

In this work package we designed and developed LibReplay, a lightweight architecture for distributed logging and deterministic replay in sensor networks. LibReplay enables (1) event logging with small intrusion of the system, and (2) deterministic event replay in controlled environments such as system simulators. As a result, we can exploit the debugging capabilities of modern system simulators.

LibReplay simplifies bug hunting in deployed sensor networks and provides a debugging experience similar to debugging (local and non-distributed) sequential programs. We discuss the architecture of LibReplay and our research-prototype implementation. Our performance evaluation show that the efficiency of LibReplay is similar to the state of the art, which commonly does not log sufficient information to provide replay capabilities. Overall, LibReplay offers deep insights into real-world deployments and allows debugging and testing in realistic settings.

Future directions include extending LibReplay with snapshot capabilities [27] [28] This will increase the flexibility when debugging long-running applications: We take a snapshot of the state of the code of interest and begin logging afterwards.

© 2014 KARYON Project

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] W. Dong, H. Chao, J. Wang, C. Chen, J. Bu and X. Xu, “Dynamic Logging with Dylog for Networked Embedded Systems”, in Proceedings of the IEEE International Conference on Sensing, Communication, and Networking (SECON). (2014)

[7] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss and P. Levis, “Collection Tree Protocol”, in Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems. SenSys ’09 (2009)

[8] R. Sauter, O. Saukh, O. Frietsch and P. J. Marron, “TinyLTS: Efficient Network-Wide Logging and Tracing System for TinyOS”, in Proceedings of the IEEE International Conference on Computer Communications (INFOCOM). (2011)

[9] 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)

[10] 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 (2009)

[11] P. Sommer and B. Kusy, “Minerva: Distributed Tracing and Debugging in Wireless Sensor Networks”, in Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems. SenSys ’13 (2013)

[12] L. Wan and Q. Cao, “Towards Instruction Level Record and Replay of Sensor Network Applications”, in Proceedings of the IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2013

[13] M. Tancreti, M. S. Hossain, S. Bagchi and V. Raghunathan, “Aveksha: A Hardware-software Approach for Non-intrusive Tracing and Profiling of Wireless Embedded Systems”, in Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems. SenSys ’11 (2011)

[14] V. Sundaram, P. Eugster and X. Zhang, “Prius: Generic Hybrid Trace Compression for Wireless Sensor Networks”, in Proceedings of the 10th ACM Conference on Embedded Networked Sensor Systems. SenSys ’12 (2012)

© 2014 KARYON Project

[15] M. R. Stallmanand C. Support, “Debugging with GDB: The GNU source-level debugger, GDB version 4.16”, Free Software Foundation. (1996)

[16] F. Österlind, A. Dunkels, J. Eriksson, N. Finne and T. Voigt, “Cross-Level Sensor Network Simulation with COOJA”, in Proceedings of the IEEE Conference on Local Computer Networks (LCN). (2006)

[17] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, and T. Voigt, R. Sauter and P. J. Marron, “Towards Interoperability Testing for Wireless Sensor Networks with COOJA/MSPSim”, in the Proceedings of the European Conference on Wireless Sensor Networks (EWSN). (2009)

[18] B. L. Titzer, D. K. Lee, and J. Palsberg, “Avrora: Scalable Sensor Network Simulation with Precise Timing”, in Proc.~of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN). (2005)

[19] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator”, in Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC). (2005)

[20] 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.

[21] 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.

[22] 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

[23] 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)

[24] 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

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

[26] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors”, In Proceedings of the IEEE Confernece on Local Computer Networks (LCN). (2004)

[27] F. Österlind, A. Dunkels, T. Voigt, N. Tsiftes, J. Eriksson and N. Finne, “Sensornet Checkpointing: Enabling Repeatability in Testbeds and Realism in Simulations”, in Proceedings of the European Conference on Wireless Sensor Networks (EWSN). (2009)

[28] A. Löscher, N. Tsiftes, T. Voigt and V. Handziski, “Efficient and Flexible Sensornet Checkpointing”, in Proceedings of the European Conference on Wireless Sensor Networks (EWSN). (2014)