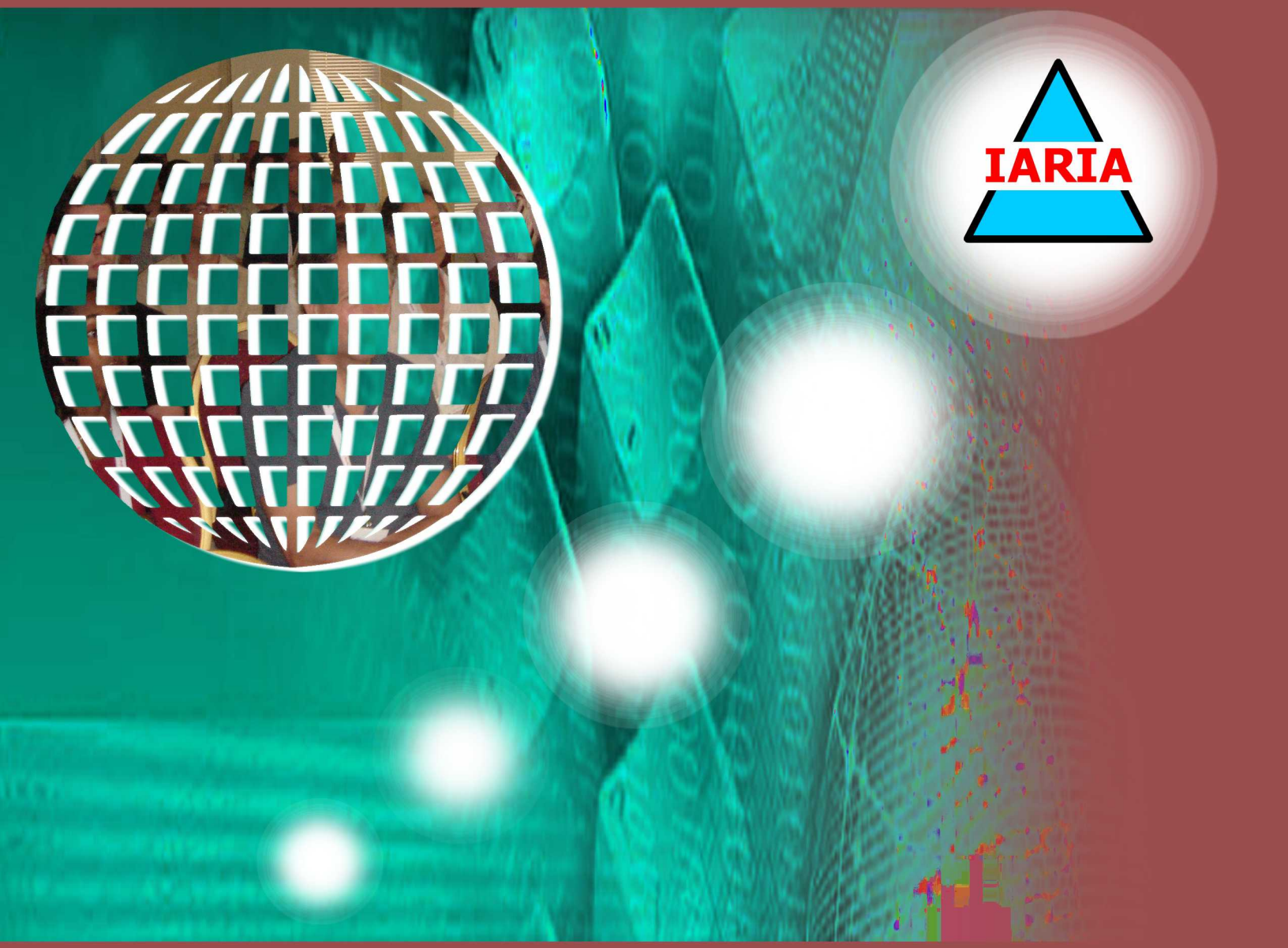
International Journal on Advances in Software



2009 vol. 2 nr. 4

The *International Journal on Advances in Software* is published by IARIA.
ISSN: 1942-2628
journals site: http://www.iariajournals.org
contact: petre@iaria.org

Responsibility for the contents rests upon the authors and not upon IARIA, nor on IARIA volunteers, staff, or contractors.

IARIA is the owner of the publication and of editorial aspects. IARIA reserves the right to update the content for quality improvements.

Abstracting is permitted with credit to the source. Libraries are permitted to photocopy or print, providing the reference is mentioned and that the resulting material is made available at no cost.

Reference should mention:

International Journal on Advances in Software, issn 1942-2628
vol. 2, no.4, year 2009, http://www.iariajournals.org/software/

The copyright for each included paper belongs to the authors. Republishing of same material, by authors or persons or organizations, is not allowed. Reprint rights can be granted by IARIA or by the authors, and must include proper reference.

Reference to an article in the journal is as follows:

<Author list>, "<Article title>"
International Journal on Advances in Software, issn 1942-2628
vol. 2, no. 4, year 2009,<start page>:<end page> , http://www.iariajournals.org/software/

IARIA journals are made available for free, proving the appropriate references are made when their content is used.

Sponsored by IARIA
www.iaria.org

Copyright © 2009 IARIA

International Journal on Advances in Software
Volume 2, Number 4, 2009

Editor-in-Chief

Jon G. Hall, The Open University - Milton Keynes, UK

Editorial Advisory Board

- Meikel Poess, Oracle, USA
- Hermann Kaindl, TU-Wien, Austria
- Herwig Mannaert, University of Antwerp, Belgium

Software Engineering

- Marc Aiguier, Ecole Centrale Paris, France
- Sven Apel, University of Passau, Germany
- Kenneth Boness, University of Reading, UK
- Hongyu Pei Breivold, ABB Corporate Research, Sweden
- Georg Buchgeher, SCCH, Austria
- Dumitru Dan Burdescu, University of Craiova, Romania
- Angelo Gargantini, Universita di Bergamo, Italy
- Holger Giese, Hasso-Plattner-Institut-Potsdam, Germany
- Jon G. Hall, The Open University - Milton Keynes, UK
- Herman Hartmann, NXP Semiconductors- Eindhoven, The Netherlands
- Hermann Kaindl, TU-Wien, Austria
- Markus Kirchberg, Institute for Infocomm Research, A*STAR, Singapore
- Herwig Mannaert, University of Antwerp, Belgium
- Roy Oberhauser, Aalen University, Germany
- Flavio Oquendo, European University of Brittany - UBS/VALORIA, France
- Eric Pardede, La Trobe University, Australia
- Aljosa Pasic, ATOS Research/Spain, NESSI/Europe
- Robert J. Pooley, Heriot-Watt University, UK
- Vladimir Stantchev, Berlin Institute of Technology, Germany
- Osamu Takaki, Center for Service Research (CfSR)/National Institute of Advanced Industrial Science and Technology (AIST), Japan
- Michal Zemlicka, Charles University, Czech Republic

Advanced Information Processing Technologies

- Mirela Danubianu, "Stefan cel Mare" University of Suceava, Romania
- Michael Grottke, University of Erlangen-Nuremberg, Germany
- Josef Noll, UiO/UNIK, Sweden

- Olga Ormandjieva, Concordia University-Montreal, Canada
- Constantin Paleologu, University 'Politehnica' of Bucharest, Romania
- Liviu Panait, Google Inc., USA
- Kenji Saito, Keio University, Japan
- Ashok Sharma, Satyam Computer Services Ltd – Hyderabad, India
- Marcin Solarski, IBM-Software Labs, Germany

Advanced Computing

- Matthieu Geist, Supelec / ArcelorMittal, France
- Jameleddine Hassine, Cisco Systems, Inc., Canada
- Sascha Opletal, Universitat Stuttgart, Germany
- Flavio Oquendo, European University of Brittany - UBS/VALORIA, France
- Meikel Poess, Oracle, USA
- Kurt Rohloff, BBN Technologies, USA
- Said Tazi, LAAS-CNRS, Universite de Toulouse / Universite Toulouse1, France
- Simon Tsang, Telcordia Technologies, Inc. - Piscataway, USA

Geographic Information Systems

- Christophe Claramunt, Naval Academy Research Institute, France
- Dumitru Roman, Semantic Technology Institute Innsbruck, Austria
- Emmanuel Stefanakis, Harokopio University, Greece

Databases and Data

- Peter Baumann, Jacobs University Bremen / Rasdaman GmbH Bremen, Germany
- Qiming Chen, HP Labs – Palo Alto, USA
- Ela Hunt, University of Strathclyde - Glasgow, UK
- Claudia Roncancio INPG / ENSIMAG - Grenoble, France

Intensive Applications

- Fernando Boronat, Integrated Management Coastal Research Institute, Spain
- Chih-Cheng Hung, Southern Polytechnic State University, USA
- Jianhua Ma, Hosei University, Japan
- Milena Radenkovic, University of Nottingham, UK
- DJamel H. Sadok, Universidade Federal de Pernambuco, Brazil
- Marius Slavescu, IBM Toronto Lab, Canada
- Cristian Ungureanu, NEC Labs America - Princeton, USA

Testing and Validation

- Michael Browne, IBM, USA
- Cecilia Metra, DEIS-ARCES-University of Bologna, Italy
- Krzysztof Rogoz, Motorola, USA
- Sergio Soares, Federal University of Pernambuco, Brazil

- Alin Stefanescu, SAP Research, Germany
- Massimo Tivoli, Universita degli Studi dell'Aquila, Italy

Simulations

- Robert de Souza, The Logistics Institute - Asia Pacific, Singapore
- Ann Dunkin, Hewlett-Packard, USA
- Tejas R. Gandhi, Virtua Health-Marlton, USA
- Lars Moench, University of Hagen, Germany
- Michael J. North, Argonne National Laboratory, USA
- Michal Pioro, Warsaw University of Technology, Poland and Lund University, Sweden
- Edward Williams, PMC-Dearborn, USA

International Journal on Advances in Software
Volume 2, Number 4, 2009

CONTENTS

Design of a Decoupled Sensor Network Architecture Based on Information Exchanges

Eli De Poorter Ingrid Moerman Piet Demeester
Ghent University - IBBT, Department of Information Technology (INTEC)
Gaston Crommenlaan 8, Bus 201, 9050 Ghent, Belgium
Email: eli.depoorter@intec.ugent.be

Abstract

Sensor networks are used for simple monitoring applications, but also for complex applications, such as wireless building automation or medical assistance. Current layered architectures do not support the dynamic and heterogeneous nature of these networks. Therefore, we present an alternative architecture that decouples protocol logic and packet representation. Using this system, multiple information exchanges are automatically combined in a single packet. In addition, the system dynamically selects the most optimal network protocols and supports system-wide quality-of-service. Thus, our architecture is better suited for next-generation applications. We illustrate our architecture with several code examples, and prove that our architecture is much more scalable, in terms of memory requirements, energy requirements and processing overhead, than tradition system architectures.

Keywords: wireless sensor networks, sensornet architecture; system architecture; QoS; energy efficiency; heterogeneity; protocol selection

1 Introduction

Sensor nodes are small and cheap devices that can monitor their environment. They are equipped with a simple radio to communicate with other sensor nodes. Due to their low cost, many sensor nodes can be distributed over an area. Over the last years, many wireless sensor networks (WSNs) have been deployed to monitor nature and office environments [2].

This article extends the paper '*An Information Driven Sensornet Architecture*' [1]. It adds illustrative code examples, it elaborates on several design choices and it provides additional evaluation results.

Due to these successes, other application domains have expressed a strong interest in using WSNs for more complex tasks. More sophisticated applications, such as process monitoring and control, wireless building automation, medical monitoring, disaster intervention or asset tracking, also benefit greatly from the use of many cheap sensor devices. However, the number of successful deployments of these applications is far less [3].

These next-generation applications impose many network requirements which are not found in traditional WSNs.

- To provide sufficient end-user support, a WSN must be easy to update and maintain. *Run-time addition* of new services and network protocols should be supported.
- WSNs will become *heterogeneous* [4], containing both simple nodes (such as light switches) and more complex nodes (such as heating controllers).
- Additionally, *QoS requirements* can no longer be ignored [5]. Medical, security and surveillance applications require that each application has its own set of specific QoS requirements.
- Since future applications will require even smaller nodes, new ways have to be found to ensure that network protocols have a very *small memory footprint*.
- Since most sensor nodes are battery powered, *energy efficiency* remains very important.

At present, there is no architecture that supports all of these challenges. As stated by Culler et. al: "*the primary factor currently limiting progress in sensornets is not any specific technical challenge but is instead the lack of an overall sensor network architecture*" [6].

Therefore, in this paper, we present an information driven architecture ('IDRA') for wireless sensor networks. This framework is specifically designed to support next-generation WSN applications. It takes into account the heterogeneity of the sensor nodes and supports energy-efficiency and QoS at an architectural level. What's more,

the proposed architecture can be made fully compatible with existing legacy sensor networks.

The remainder of this paper is organized as follows. In Section 2, we discuss the design philosophy behind our information driven system architecture. Section 3 clarifies how such an information driven architecture can be implemented. Next, in Section 4, we demonstrate how this architecture can be used to support energy efficiency, QoS, heterogeneity and legacy networks. In Section 5, we discusses possible shortcomings of the architecture. We describe our implementation experiences in Section 6 and evaluate the performance of our system in Section 7. Next, in Section 8, we compare the architecture with existing sensornet architectures. Finally, Section 9 concludes this paper.

2 What is an information driven architecture?

The main responsibility of a network protocol is to ensure that information is relayed to the correct destination. However, in practice, a network protocol has many responsibilities that are not directly related with its main function. Each protocol layer must *(i)* define a message format (including header and trailer fields), *(ii)* provide buffers to temporarily store packets and *(iii)* gather information from other nodes.

We argue that this approach is very inefficient. It makes no sense that every individual protocol layer has to bear the burden of gathering information, providing buffers and implementing header manipulations. Common sense indicates that such functions, which are implemented multiple times in the system, should be implemented in a shared library. Furthermore, the same information is often gathered by multiple protocols, each time resulting in an additional information exchange.

Therefore, we propose a system in which protocol design is based on 'information exchanges'. The role of a network protocol is simplified to its 2 main tasks: sending information and interacting with the relayed information. Packet creation and buffer provisioning are delegated to the architecture. This way, network protocols are simpler and require less memory.

3 Design of an information driven architecture

In this section, we describe how such an information driven architecture can be implemented. A conceptual representation is given in Figure 1.

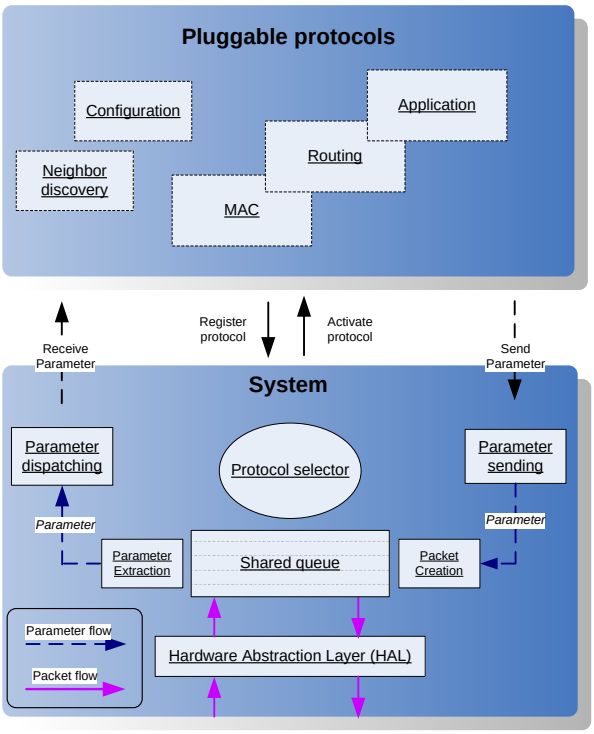


Figure 1. In an information driven architecture, protocols exchange information with the system, and rely on the system to create and send packets.

3.1 Information exchanges

Network protocols often exchange information with a remote node. Typical examples of exchanged information are:

- measured data values, such as the local temperature;
- status updates, such as the remaining battery capacity;
- or control information such as a route-request.

Using our information driven approach, network protocols do not create a new packet to send these types of information to a remote node. Instead, they rely on the system to send and receive information.

To send information to a remote node, the protocol hands over an information parameter to the system, together with the required destination (Fig. 1, 'Parameter sending'). The system will create a new packet and ensures that the parameter is encapsulated into this packet.

To receive information from other nodes, network protocols indicate to the system what type of parameters they are interested in. Whenever a packet arrives at its final destination, the system decapsulates the information parameters and distributes them to the interested protocols (Fig. 1, 'Parameter dispatching').

The main advantages of transferring the creation of packets to the system are: *(i)* the system can ensure that redundant control information is sent only once; *(ii)* protocols are simpler since they do not need to implement memory operations for manipulating header fields; and *(iii)* by combining multiple information parameters into a single packet, the number of required packets decreases drastically (Section 4.1).

3.2 Shared queue

Traditional networks use a 'store-and-process' approach, where each network protocol stores the packets in an internal queue before processing. Each protocol layer requires these queues because packets may be passed up or down faster than they can be processed. Thus, the total amount of buffer memory increases linearly with the number of protocol layers.

In contrast, in the information driven architecture, incoming packets are stored in a system-wide queue (Fig. 1, 'Shared queue'). The system selects which packets are ready for processing by a network protocol so that one-after-another the protocols can process the packets.

The use of a shared, system-managed queue has several advantages: *(i)* protocols are simpler and smaller since they do not have to allocate queue memory; *(ii)* packets do not need to be copied between protocols, resulting in less processing overhead; *(iii)* since the queue occupation from all protocols is averaged, less total queue memory is required; and *(iv)* monitoring and managing the total number of packets in the system is simpler.

3.3 Packet façade

Traditional network protocols encapsulate packets in header fields. These header fields contain information which can only be read by the protocol that created the packet header. In our information driven architecture, a packet façade (Fig. 2) is responsible for packet creation.

The system uses the packet façade to create new, standardized packets. Information parameters are encapsulated in the payload of the created packet and stored in the shared queue.

Network protocols use the packet façade to interact with relayed packets. Protocols add or retrieve packet attributes, such as 'source', 'destination', 'QoS ID' or 'time-to-live'. These packet attributes fulfill the same role as traditional header fields, but are more dynamic: they can be omitted or added freely without redefining the packet structure. *Moreover, packet attributes have a system-wide significance: they can be inspected by the system or any other protocol.*

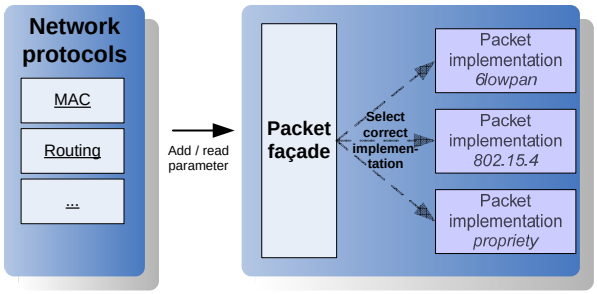


Figure 2. Through a packet façade, protocols interact with packets. Protocols do not require any knowledge about the actual packet construction.

The packet façade uses a separate 'packet implementation' module to convert the attributes of a packet into an actual radio packet. Thus, the system and the protocols do not need to worry about the actual storage of the control information associated with a packet. Developers can choose to use one of the existing 'packet implementation' modules, or provide their own (propriety) packet implementation. This way, the packet type can be changed without any changes to the system or the protocols: protocol logic and packet representation are decoupled.

Using a separate packet façade has the following advantages: *(i)* protocol development is simplified since there is no need to define headers; *(ii)* packet attributes have a system-wide significance and can be inspected by any protocol or architecture; *(iii)* since protocols are not tied to a specific packet implementation, the encompassing packet type can easily be changed or optimized (e.g.: 6lowpan, IEEE802.15.4 or a custom packet).

3.4 Pluggable protocols

The traditional OSI reference model [7] uses a non-flexible layered architecture: packets are sent to a predetermined protocol layer. In contrast, the information driven architecture does not statically wire packets to a specific protocol. The system decides at run-time which protocols should be selected to process incoming packets (Fig. 1, 'Protocol selector').

To be selected for packet processing, a protocol must register itself by adding *filters* to the system. These filters indicate for which packet types the protocol is optimized. Through the packet façade, the system checks if the characteristics of the arriving packets match any of the registered filters, and selects the appropriate network protocols to process the packet. When no filters match, a default routing or MAC protocol is chosen.

This approach has the advantages that multiple, specialized network protocols can be combined in the same sys-

tem. For example, a routing protocol implements an efficient broadcast algorithm. It registers itself for all packets that are broadcasted over the network. Another routing protocol delivers high QoS guarantees. It can add a filter to indicate that it is optimized for routing voice packets.

'Plugging' in protocols, rather than statically wiring them, has several advantages. *(i)* Since many applications have diverse network requirements, the architecture is able to dynamically change between different routing or MAC protocols at run-time. The optimal protocol is selected by the system based on the network context or the packet type [8]. *(ii)* Run-time insertion of protocols is supported. *(iii)* Legacy systems can be supported (Section 4.4). More system-specific implementation details can be found in Section 6.

4 Exploiting the concept

Current system architectures are not designed to support energy-efficiency, QoS, heterogeneity or legacy networks. Traditionally, support for these advanced features is very complex and requires major changes to the underlying system architecture. The lack of architectural support for QoS and heterogeneity is a major obstacle that hampers the deployment of many next-generation applications for WSNs.

In the next sections, we demonstrate how these features can easily be implemented at an architectural level in our information driven system.

4.1 Energy efficiency

Since energy is scarce in sensor nodes, wireless sensor networks aim to transmit as few packets as possible. To this end, 'measured data' from different nodes can be combined into a single packet by data-aggregation protocols [9]. This data-centric approach has several limitations: *(i)* data aggregation is application dependent; *(ii)* aggregation protocols are often coupled to a specific routing protocol; *(iii)* data aggregation is limited to the measured '*data*', it does not include other exchanged control messages; *(iv)* data that originates from different applications can not be aggregated.

We claim that this approach should be broadened so that *all* types of information exchanges are aggregated. Many information exchanges between nodes are not very time-sensitive, such as status information, remaining energy information, or low-priority routing information. As such, it is reasonable to assume that some of these packets can be delayed for a short amount of time before being sent. When a protocol requests the sending of a parameter, the protocol should also give an indication of the time-sensitivity of the parameter.

The system collects the information parameters in the waiting space of a central repository (Figure 3). Delay-tolerant parameters can remain in the waiting space for up to a per-parameter predefined period of time. Whenever a packet is relayed through the node, all information parameters to the same 'next hop' or 'destination' address are added to the packet. If no data has been relayed within the allowed waiting time, the system generates a new packet which combines all parameters that are destined for the same node. Finally, when a packet reaches its final destination, the system will distribute the encapsulated information parameters to the interested protocols.

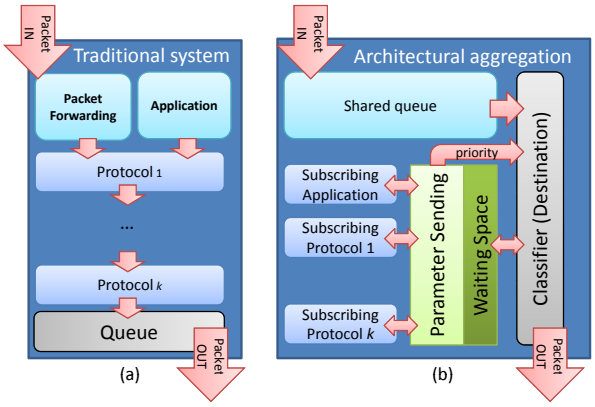


Figure 3. Extending the data aggregation concept. (a) Traditional architecture. (b) Architecture with support for global aggregation.

For a more in-depth analysis, we refer to [10], where it is shown that, ideally, the number of transmissions can be lowered by a factor, equal to the number of information parameters which can be combined in a single packet. In contrast with traditional aggregation schemes [9, 11], this approach is part of the architectural design. As a result, the network developer can combine this approach with any type of routing protocol, and information from all layers can be aggregated, rather than just application data.

4.2 Architectural QoS

QoS guarantees are required by many medical, security, critical monitoring and control applications. However, current QoS research focuses mainly on one of the network layers and solves only a few of the application requirements. As stated by Troubleyn et al. in [5], QoS should be supported in both the protocols and the architecture. Only then can system-wide QoS be guaranteed.

The information driven architecture is very suited to support architectural QoS. Through the packet façade, the system can read the QoS attributes of any relayed packet. Since all packets are stored in a shared packet queue, the system can monitor all available packets. This gives the system a

clear view on the expected delay of each packet. QoS can be supported by giving precedence to packets with a higher QoS level, or by intelligently dropping non-priority packets. To fulfill QoS requirements, a QoS protocol can put the processing of low-priority packets on hold, even when those packets are currently being processed by a protocol. Finally, the architecture supports multiple protocols, so that high-priority packets which require strict QoS guarantees can be processed by specialized protocols.

4.3 Heterogeneity support

Applications such as process and asset monitoring, disaster intervention and wireless building automation require special devices ('actuators'), which can interact with the environment [2, 4]. Thus, future sensor networks will consists of nodes with strongly diverging capabilities.

In a layered architecture, every sensor node needs to support the same protocol stack. Since no protocols can be omitted or added, the layered approach has limited support for heterogeneity. When using an information driven architecture, packet attributes remain associated with a packet, whether or not the protocol that added them is executed. Thus, the system can choose to omit non-essential protocols from nodes with little capabilities (Figure 4). The system can also choose to execute different, more simple protocols on lightweight nodes. These protocols can add their own packet attributes or can reuse the packet attributes that were added by previous protocols. This flexibility ensures that our architecture is suitable for both high capacity and low capacity nodes.

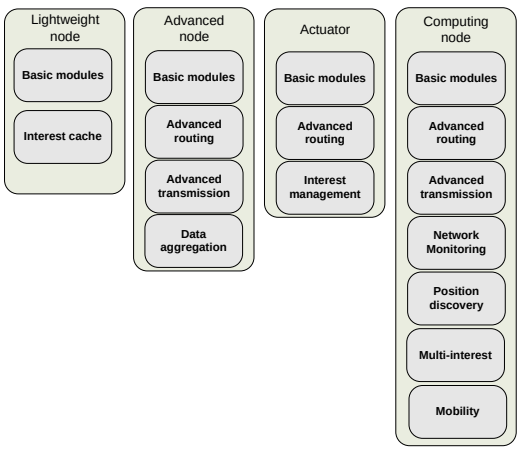


Figure 4. Depending on their capabilities, the number of protocols can be varied

4.4 Legacy support

One of the main problems new architectures face, is that they are not backwards compatible with existing infrastructure. Traditionally, the only solution is the installation of a translation node through which all communication passes, which results in a very inefficient network use.

The information driven architecture allows legacy nodes and IDRA nodes to communicate directly by supporting *(i)* legacy packet types, and *(ii)* a legacy MAC protocol.

Legacy packets can be supported transparently for the new protocols by providing the corresponding 'packet implementation'. This implementation should store the relevant control parameters at the expected header locations of the legacy packets.

The *legacy MAC protocol* can be ported to the new architecture. By registering it as the optimal MAC protocol for neighboring legacy nodes, the 'protocol selector' will always select the correct MAC protocol to send packets to the legacy nodes.

5 Disadvantages

In the next section, we discuss some possible disadvantages of using an information driven architecture.

5.1 Protocol-defined packets

In an information driven architecture, the system defines how packets are constructed. In addition, messages defined by the radio, such as ACKs, are supported through the transmission settings library.

However, some MAC protocols use custom-defined packet types (such as 'strobes') for their operation. Currently, the architecture does not allow network protocols to send self-defined packets. We feel that allowing this would take away many advantages of the packet façade approach. Asynchronous MAC protocols can still be implemented by sending the same packet multiple times.

If further research indicates that support for protocol-defined packets is absolutely required, a library will be added that allows a protocol to send and receive a self-defined packet. Of course, these self-defined packets can not profit from many of the advantages of the information driven approach (such as reuse of packet parameters, efficient combination of exchanged information, and the decoupling of protocol logic and packet representation).

5.2 Standardization

An information driven architecture strongly benefits from an approach that standardizes both the information parameters and the attributes that are associated with a packet.

An *information parameter* represents information that is exchanged between protocols or applications on different nodes. Information parameters are often of interest to several protocols (for example: the remaining energy of neighboring nodes). When an already existing, standardized information parameters is added to the waiting space of the system, the corresponding value is updated. The information parameter is sent only once, rather than once for every interested protocol.

A *packet attribute* represents packet-associated information that is required to route a packet to its destination. Typical examples are a 'next hop' or a 'time-to-live' attribute. Standardizing control parameters ensures that they can be read by all network protocols, and that the system is aware of the properties of each packet.

6 Implementation

The presented information driven architecture ('IDRA') has been implemented using the TinyOS [12] operating system. Run-time addition of protocols is currently not supported, since TinyOS does not support dynamic code updates.

6.1 Internal workflow

Figure 5 shows the internal workings of the architecture. The shared queue is the central component of the architecture. Each packet in the shared queue has an associated *packet status*. Depending on the status of a packet, the following actions can be taken.

Pre-processing is executed once for each arriving packet. During pre-processing, duplicate packets and packets with a different next hop address are dropped. Also, when the packet reaches its final destination, the encapsulated information parameters are extracted and distributed to the interested protocols.

Processing is executed next. The 'Protocol selector' module analyzes the packet and selects the most optimal network protocol based on the packet characteristics (see Section 6.2).

Post-processing Packets that have been processed by the network protocols are prepared for sending. Relevant packet attributes (such as the sender address) are updated. Also, if parameters to the same next hop address or destination address are available, they are aggregated to the packet.

Ready For Sending After post-processing, the packets remain in the shared queue until the MAC protocol orders the system to send the packet.

6.2 Protocol sequence

When using a traditional layered architecture, arriving packets must first be stripped of their headers before they can be processed. Therefore, packets go through the protocol stack twice: once from the bottom to the top to remove the packet headers, and a second time from the top to bottom to actually process the packets. In the IDRA system, network protocols can access any packet information through the packet façade. Thus, there is no need to execute each protocol layer twice.

There is no fixed call sequence for the protocols since multiple protocols can co-exist on the same node. Depending on the protocol filters (see Section 3.4), the most optimal network protocol is selected. The algorithm for protocol selection is very simple so that it can be implemented even on lightweight nodes.

1. First, all interested monitoring protocols are executed. A monitoring protocol is considered interested if at least one of its filters match the current packet.
2. Afterwards, the most optimal routing protocol is executed. This is the network protocol with most matching filters (or, in the case of a tie, the first protocol that registered itself).
3. Finally, the most optimal MAC protocol is executed. This is the network protocol with most matching filters (or, in the case of a tie, the first protocol that registered itself).

The network protocols are always executed in the same order. Once a protocol finishes processing a packet, it signals one of the following return values to the system:

SUCCESS The network protocol finished successfully; the next protocol can be executed.

FAIL The network protocol can not process the packet; the packet should be dropped from the shared queue.

EBUSY The network protocol is not yet ready to process the packet; the protocol will be called again at a later time.

6.3 System libraries

A vertical component is available to the architecture (Fig. 5, 'module interactions'). Through this component, the following system provided libraries are available:

Information Exchange: through this interface, protocols can distribute information parameters to other nodes and receive information parameters from other nodes. The system converts the information parameters into a packet.

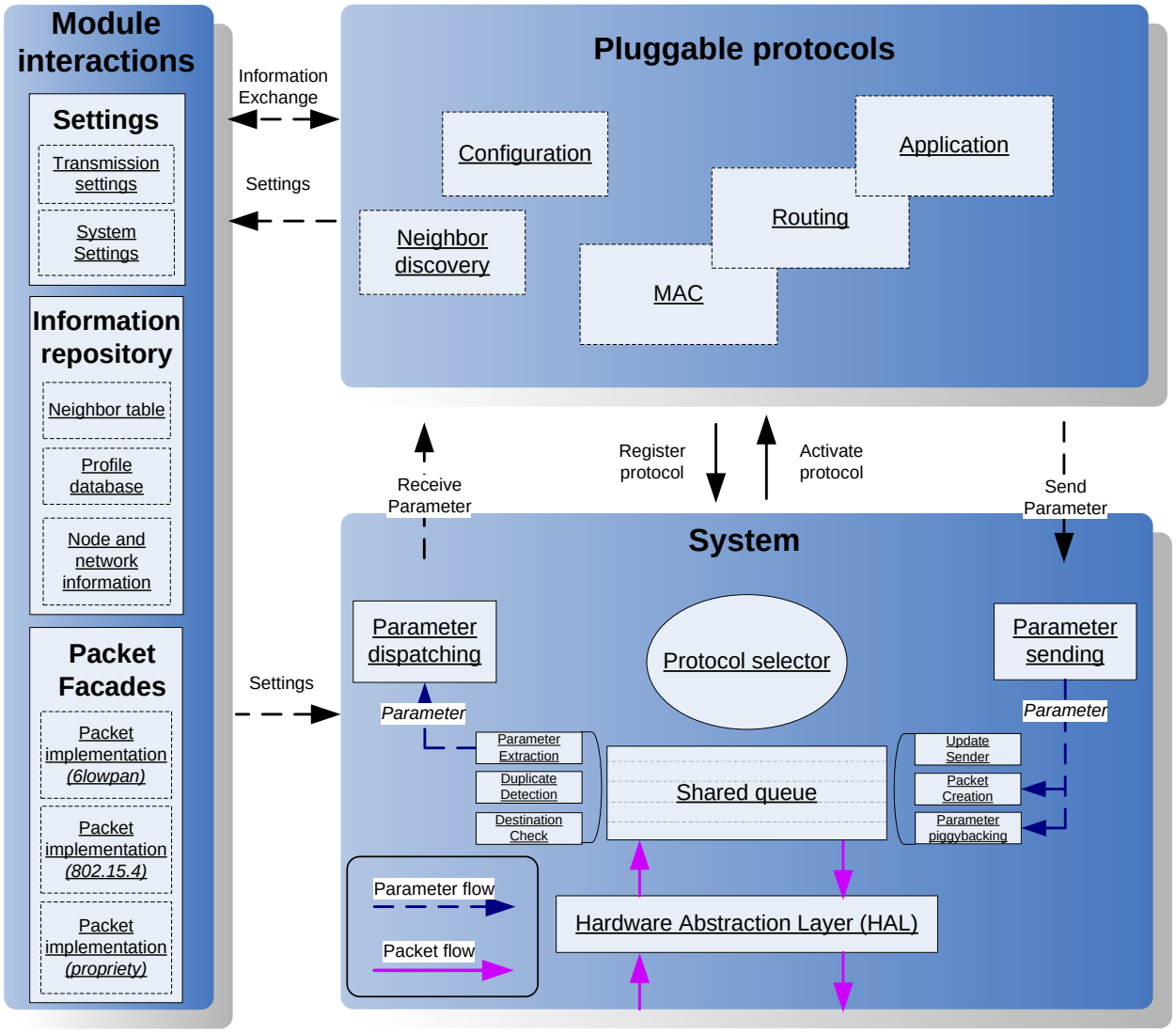


Figure 5. The information driven architecture ('IDRA').

Packet Façade: the packet façade is used to interact with system-created packets. Packet attributes can be added or read.

Protocol Selection: protocols can add filters to this component to indicate for which packet types the protocol is optimized.

System Settings: through this interface, modules change or read system settings, such as the node ID or the current battery voltage.

Transmission Settings: through this interface, protocols manage the sending of packets. It has provisions for *(i)* requesting how many packets from the shared queue are ready to be send, *(ii)* ordering the system to send a specific packet, and *(iii)* changing the radio settings.

6.4 Example code

In this section, we show how these libraries can be used to create a new network protocol. A simple routing protocol for broadcast packets is shown in Table 1. First, the network protocol registers itself as a routing protocol that is optimized for broadcast packets. Whenever the system selects this routing protocol for processing a packet, the network protocol uses the packet façade to update the packet attributes.

A second example is shown in Table 2 and demonstrates the implementation of a simple MAC protocol. Through the 'Transmission Settings' library , radio settings such as the transmission power and channel can be changed. Even though packets are send by the system, the MAC protocol

```
Module BroadcastRouting{

    // 1. Declaration of variables
    uint8_t moduleID=unique("protocol");
    uint8_t ttl=0;

    // 2. The protocol registers itself as a routing protocol suitable for broadcast packets
    command error_t Module.init(){
        call ProtocolRegistration.addFilter(moduleID, ROUTING_PROTOCOL,
                    DESTINATION, 2, EQUALS, BROADCAST_ADDR);
        return SUCCESS;
    }

    // 3. Executed when the protocol is selected for routing a broadcast packet
    command void Module.processPacket(uint8_t moduleId, void* packet){
        if(moduleId == moduleID){

            // 3.1. Get the time-to-live of the packet
            result=call PacketFacade.getPacketAttribute(packet, TIME_TO_LIVE, sizeof(ttl), (void*) &ttl);

            // 3.2. If the time-to-live equals zero: drop the received packet.
            if(ttl==0) signal Module.processPacketDone(moduleId, packet, FAIL);

            // 3.3. Otherwise, update the time-to-live and broadcast the packet again[a].
            else{
                ttl=ttl-1;
                call PacketFacade.setPacketAttribute(packet, TIME_TO_LIVE, sizeof(ttl), (void*) &ttl);
                signal Module.processPacketDone(moduleId, packet, SUCCESS);
            }
        }
    }
}
```

[a] A more advanced implementation should also check for duplicate packets to prevent broadcast storms.

Table 1. Example code: a simple broadcast routing protocol.

remains in control of sleeping schemes and accurate timing of the sending. The MAC protocol can request at any time which packets are ready for sending and order the system to send a specific packet.

7 Evaluation

In this section, we evaluate three key criteria of our architecture: the memory footprint, the energy efficiency and the processing overhead.

7.1 Memory footprint

The memory footprint of the different components of the architecture is shown in Table 3. The entry 'Other system components' refers to the implementation of preprocessing and postprocessing functions, QoS provisions and a shared neighbor table. The full architecture requires about 20kb

Component	ROM	RAM
Information exchanges (Section 3.1)	800	585
Shared queue (Section 3.2)	1330	2560
Packet façade (Section 3.3)	1402	6
Protocol selection (Section 3.4)	206	234
Radio support (HAL)	8892	322
Other system components	7606	1637
Total	20236	5344

Table 3. Memory footprint (in bytes) of the different architectural components of the system.

```
Module SimpleMAC{

    // 1. Get a unique protocol ID
    uint8_t moduleID=unique("protocol");

    // 2. The protocol registers itself as a general purpose MAC protocol
    command error_t Module.init(){
        call ProtocolRegistration.addFilter(moduleID, MAC_PROTOCOL,
                    0, 0, 0, 0); // default protocol
        call alarmClock.startPeriodic(200); // Every 200 msec, the protocol checks if packets are ready to be sent
        return SUCCESS;

    // 3. Regularly check if a packet is ready for sending
    event void alarmClock.fired(){
        if(call TransmissionSettings.checkNumberOfPackets() > 0) {
            call TransmissionSettings.sendPacket();
        }
    }

    // 4. Inform the system that the packet can be removed from the shared queue
    event void TransmissionSettings.sendPacketDone(uint32_t timestamp, error_t result){
        if(result==SUCCESS) call TransmissionSettings.removeLastSentPacket();
    }
}
```

Table 2. Example code: a simple MAC protocol.

Protocol name	ROM	RAM
TOS2.1 MAC [12]	11528	320
SCP-MAC [13]	21372	1056
X-MAC [14]	19854	876
IDRA LPL MAC [15]	822	176
IDRA S-MAC [16]	1126	184

Table 4. Comparison of the memory requirements (in bytes) of TinyOS MAC protocols with IDRA MAC protocols.

Protocol name	ROM	RAM
CTP [17]	7234	1198
DYMO [18]	11404	482(+60 per route)
Lunar [19]	5000	1518
IDRA CTP [17]	712	130
IDRA DYMO [20]	5008	312(+18 per route)

Table 5. Comparison of the memory requirements (in bytes) of TinyOS routing protocols with IDRA routing protocols.

ROM and 5 kB RAM memory, well under the memory limit of most sensor nodes.

To demonstrate the feasibility of the IDRA architecture, we implemented 2 MAC protocols (S-MAC [16] and LPL [15]) and 2 routing protocols (Collection Tree Protocol [17] and DYMO [20]).

When using a layered architecture, the memory consumption increases linearly with the number of network protocols. In contrast, using the IDRA architecture requires a significant initial investment in terms of memory, even without any network protocols. However, adding protocols to the system requires significantly less memory than when using a more traditional approach (Table 4 and 5). Due to the system provided libraries for most typical operations, the memory requirements of network protocols are reduced by a factor 2 to 10. Thus, the greater initial memory cost of our system is quickly offset.

7.2 Aggregation efficiency

An important feature of IDRA is the automatic combination and aggregation of information exchanges. In [10] an ILP formulation is given for calculating required number of packet transmissions for different protocols when using our

aggregation method.

The result is illustrated in Figure 6, where the number of packet transmissions is shown for a variable number of protocols. All protocols regularly send information to a neighboring node. Information exchanges for protocol i are sent every (ΔT_i) time units. The first protocol has a protocol cycle (ΔT_1) of five time units, each additional protocol has its ΔT increased by one.

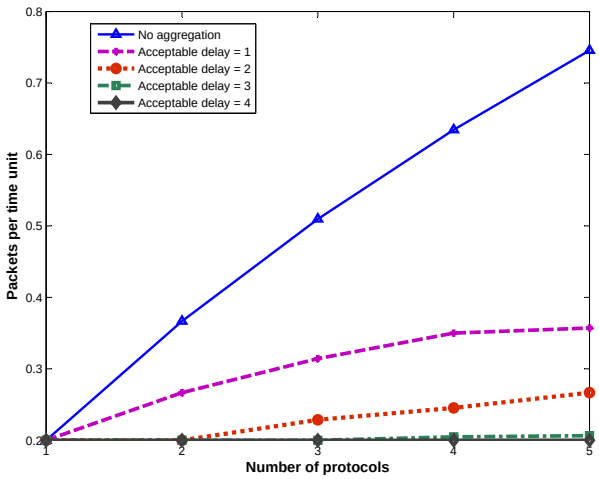


Figure 6. Required number of packets per time unit when using global aggregation with multiple protocols ($\Delta T_i = i + 4$).

Figure 6 shows that, when using global aggregation, adding new protocols does not significantly increase the required number of packets. When the acceptable delay increases, the number of packet required decreases. The minimal number of packets per time unit that needs to be sent is one packet every ΔT_1 time units, with ΔT_1 the lowest information interval. Thus, when the acceptable delays increases, the average number of packets per time unit decrease, provided that the packet size is big enough to contain the information from all the protocols. The maximum reduction of transmissions is equal to the number of control packet types. *To summarize, global-aggregation results in more profit when more parameters need to be exchanged and when these parameters have no strict deadlines.*

7.3 Processing overhead

A final performance criteria is the processing overhead of the system. System overhead is mainly caused by the following operations:

Aggregation overhead Searching for parameters to combine with routed packets results in additional processing delay. This causes problems when high-priority traffic is delayed. Therefore, the QoS module has the option to disable aggregation for high-priority packets. Thus, additional delay will only be introduced for low-priority traffic.

Packet façade overhead Using a packet façade to associate control parameters with a packet causes additional processing delay. If all types of packet attributes are not known in advance, they must be stored sequentially in a header byte array. This type of packet implementation is not very efficient, but can be used for any possible combination of packet attributes. When all packet attributes are known in advance, they can be associated with a specific header location. In this case, the packet façade can be implemented very efficiently using a simple switch statement.

Overhead for storing packets IDRA requires very few copy actions for processing packets: arriving packets are stored once in the shared queue, and remain there until processing is finished. This is much more efficient than copying a packet once for each protocol layer.

The total packet delay depends strongly on the complexity of the used packet façade and the complexity of the network protocols. However, even when using relatively network protocols, such as the combination of an AODV routing protocol, an LPL-like MAC protocol and 6lowpan packets, the processing delay is less than 20 milliseconds on a telosb [21] sensor node. This delay is well below the duty cycle of most typical WSN MAC protocols.

8 Related work

In this section, we will compare our architecture with other proposed architectures for WSNs.

8.1 A Sensor Network Architecture (SNA)

The sensor network architecture (SNA) [22] is based on 'functionality': the authors analyzed thoroughly which 'functions' or 'components' are often executed by protocols. They provided a modular MAC layer (called 'SP') [23] and a modular routing layer (called 'NLA') [6, 24]. A protocol designer can use these modules to 'build' a custom network protocol. Additionally, a cross-layer database is provided that shares components such as a message pool (similar to the 'shared queue'), a link estimation and an extensible neighbor table.

The SNA has several similar goals as IDRA, but differs in the following ways. *(i)* Rather than delegating tasks to a central system, their goal is to enable the quick development of protocol layers, using the provided components for each

layer. *(ii)* Protocols need to define their own headers, and must encapsulate packets from higher layers. *(iii)* Dynamic selection between protocols is not supported, and protocols can not view or reuse each others packet attributes. *(iv)* Limited support for energy-efficiency. Since their system can not extract meaningful parameters from packets, they combine full packets rather than only the relevant information. Additionally, they can not aggregate information to non-neighboring nodes. *(v)* Provisions for QoS or heterogeneity are not supported.

A similar component based MAC Layer Architecture (MLA) [25] also includes power management. The memory footprints of SCP-MAC and X-MAC from table 4 are calculated using the MLA architecture.

8.2 A declarative sensornet architecture

The declarative sensor network architecture [26, 27] (DSN) aims to facilitate the programming of sensor nodes, using a declarative language (called Snlog). This language provides a high level of abstraction: protocols describe what the code is doing but not how it is doing it. Algorithms are implemented using predicates, tuples, facts and rules.

The compiler represents all this information as tables. Rules are converted to dataflow plans using database operations (Join, Select, Aggregate and Project). Execution of the dataflow plans is triggered by the associated predicates. Finally, the intermediary operators are compiled into a nesC program.

DSN is especially suited for recursive protocols, such as tree construction (which requires only 7 lines of code). Additionally, protocol interoperability can be supported using database scheme matching techniques on the packets. However, the architecture currently has several disadvantages: *(i)* complex data structures are not supported, *(ii)* total memory size increases (up to a factor 3) and *(iii)* no fine grained radio control is supported (which makes the language unsuited for low-level MAC protocols).

8.3 Modular architectures

One of the limitations of a layered architecture is that it is difficult to incorporate new cross-layer services since interfaces are explicitly embedded in each layer. An alternative is to completely discard the layered structure.

Instead of using protocol layers, all responsibilities of a protocol layer are divided over separate modules [28] with a well-defined function. For example, a complex MAC layer can be divided into a neighbor management module, a sleep management module, a channel monitoring module and a retransmission module.

To prevent a large number of dependencies between the different modules, modules do not interact with each other directly. Instead, communications between modules go through a cross-layer database repository [29].

The use of a modular architecture has several advantages:

- Duplication of functionality is prevented;
- When developing a new network protocol, existing modules can easily be reused;
- cross-layer information can be exchanged, supporting the development of energy-efficient protocols;
- Depending on the node capabilities or network conditions, it is easy to add or adapt a single module;

In the design of our information driven architecture, no assumptions are made on the use of layering, modular or hybrid approaches. For modular approaches, our filter based protocol registration (Section 3.4) can be expanded to support dynamic call sequences [30]. Registering modules can indicate in which sequence the modules should be executed. Depending on the situation and packet type, the appropriate call sequence can be initiated. Thus, our proposed information driven architecture is easily adaptable to be compatible with both the layered and the modular approach.

9 Summary

Wireless sensor networks are used for increasingly complex applications, such as wireless building automation and process and asset monitoring. These applications demand more and more functionalities from the underlying sensor network. They are often deployed on strongly heterogeneous nodes and require adaptive and reliable end-to-end services. Such requirements cannot be supported at a protocol level but should be part of the overall sensornet architecture. However, no architecture currently exists that supports heterogeneity, easy protocol-integration and QoS as part of its architectural design.

Therefore, in this paper, we proposed an information driven sensornet architecture. This architecture is based on the notion that protocols should be simplified to their two main tasks: exchanging information and interacting with the relayed information. By intelligently manipulating this information, the system can support advanced network requirements for next-generation sensornet applications.

More specifically, the information driven approach has the following key advantages:

- by using a packet façade for packet interactions, protocol logic is decoupled from packet representation;
- rather than statically wiring protocols, protocols are dynamically selected (based on protocol-provided filters);

- by providing a shared, system-wide packet queue the overall memory footprint is reduced and system-wide QoS can be enforced;
- heterogeneity is promoted since protocols can be added to a node according to its capabilities;
- and finally, by efficiently combining the information exchanges, the number of transmitted packets can be strongly reduced.

To demonstrate the feasibility of these concepts, we implemented all techniques in a single architecture. We demonstrated that in our system, network protocols require significantly less memory (up to a factor 10), at the price of a larger initial memory cost. In addition, the number of packets per time unit can decrease up to a minimum of $\frac{1}{\Delta T_x}$, with ΔT_x the lowest information interval. Finally, we demonstrated that the additional processing time of our system is far less than the sleeping delay in typical wireless sensor networks.

To conclude this paper, we are convinced that future applications for WSNs will be very demanding on the network in terms of flexibility, reliability and adaptivity. In this paper, we claimed that network requirements, such as support for QoS, heterogeneity and energy-efficiency, should be part of the architectural design, rather than being added as an afterthought. As such, innovative architectural techniques that support these requirements, like the ones proposed in this paper, will be of great importance to the successful development of next-generation sensornet architectures.

Acknowledgments

This research is funded by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen) through a PhD. grant for E. De Poorter. The author wishes to acknowledge the valuable feedback of colleagues Pieter De Mil, Bart Jooris, Benoît Latré, Evy Troubleyn, Lieven Tytgat and partners from the IBBT-DEUS project.

References

[1] Eli De Poorter, Ingrid Moerman, and Piet Demeester. An information driven sensornet architecture (best paper award). In *The Third International Conference on Sensor Technologies and Applications (sensorcomm 2009)*, Athens/Glyfada, Greece, June June 18-23, 2009.

[2] I. Akyildiz and I. Kasimoglu. Wireless sensor and actor networks: Research challenges. *Ad Hoc Networks Journal (Elsevier)*, 2(4):351–367, October 2004.

[3] Carlos F. García-Hernández, Pablo H. Ibargüengoytia-González, Joaquín García-Hernández, and Jesús A. Pérez-Díaz. Wireless sensor networks and applications: a survey. *IJCSNS International Journal of Computer Science and Network Security, VOL.7 No.3,*, March 2007.

[4] M. Yarvis, N. Kushalnagar, H. Singh, A. Rangarajan, Y. Liu, and S. Singh. Exploiting heterogeneity in sensor networks. *in Proceedings of the IEEE Infocom*, 2005.

[5] Evy Troubleyn, Eli De Poorter, Ingrid Moerman, and Piet Demeester. AMoQoSA: Adaptive Modular QoS Architecture for Wireless Sensor Networks. *SENSORCOMM 2008, Cap Esterel, France*, August 25-31, 2008.

[6] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, , and I. Stoica. A unifying link abstraction for wireless sensor networks. *SenSys '05, San Diego, CA, USA,*, pages pp. 76–89, Nov. 2005.

[7] Hubert Zimmermann. OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28, no. 4:425 – 432, April 1980.

[8] J. Hoebeke, I. Moerman, B. Dhoedt, and P. Demeester. Towards adaptive ad hoc network routing. *International Journal of Wireless and Mobile Computing*, vol. 1:pp. 1–8, 2005.

[9] R. Rajagopalan and P.K. Varshney. Data-aggregation techniques in sensor networks: a survey. *Communications Surveys & Tutorials, IEEE*, 8(4):48–63, Fourth Quarter 2006.

[10] Eli De Poorter, Stefan Bouckaert, Ingrid Moerman, and Piet Demeester. Broadening the concept of aggregation in wireless sensor networks. *SENSORCOMM 2008, Cap Esterel, France*, August 25-31, 2008.

[11] E. Fasolo, M. Rossi, and M. Widmer, J.and Zorzi. In-network aggregation techniques for wireless sensor networks: a survey. *Wireless Communications, IEEE [see also IEEE Personal Communications]*, 14(2):70–87, April 2007.

[12] Tinyos operating system. http://www.tinyos.net/.

[13] W. Ye, F. Silva, and J. Heidemann. Ultra-low duty cycle mac with scheduled channel polling. pages 321–334, Boulder, CO, November 2006.

[14] M. Buettner, G. Yee, E. Anderson, and R. Han. X-MAC: A short preamble mac protocol for duty-cycledwireless networks. pages 307–320, Boulder, CO, November 2006.

[15] J. Hill and D. Culler. Mica: a wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November 2002.

[16] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *21st Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 1567–1576, June 2002.

[17] Collection Tree Protocol (CTP) for tinyOS 2.x., december 2008. http://www.tinyos.net/tinyos-2.x/doc/html/tep123.html.

[18] Tymo source code repository. tymo: Dymo implementation for tinyos, december 2008. http://tymo.sourceforge.net.

[19] LUNAR - Lightweight Underlay Network Ad hoc Routing, december 2008. http://cn.cs.unibas.ch/projects/lunar/.

[20] Ian D. Chakeres and Charles E. Perkins. Dynamic manet on-demand routing protocol (dymo). *IETF Internet Draft, draft-ietf-manet-dymo-12.txt, http://ianchak.com/dymo/draft-ietf-manet-dymo-12.html*, February 2008 (Work in Progress).

[21] Telosb reference datasheet. *http://www.xbow.com/Products/-productdetails.aspx?sid=252*.

[22] Arsalan Tavakoli, Prabal Dutta, Jaein Jeong, Sukun Kim, Jorge Ortiz, David Culler, Phillip Levis, and Scott Shenker. A modular sensornet architecture: past, present, and future directions. *SIGBED Rev.*, 4(3):49–54, 2007.

[23] D. Culler, P. Dutta, C.T. Eee, R. Fonseca, J. Hui, P. Levis, J. Polastre, S. Shenker, I. Stoica, G. Tolle, and J. Zhao. Towards a sensor network architecture: Lowering the waistline. In *In Proceedings of the Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.

[24] C.T. Ee, R. Fonseca, S. Kim, D. Moon, A. Tavakoli, D. Culler, S. Shenker, and I. Stoica. A modular network layer for sensornets. *In the Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006), Seattle, WA*, November 2006.

[25] Kevin Klues, Gregory Hackmann, Octav Chipara, and Chenyang Lu. A component-based architecture for power-efficient media access control in wireless sensor networks. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 59–72, New York, NY, USA, 2007. ACM.

[26] Arsalan Tavakoli, David Chu, Joseph M. Hellerstein, Phillip Levis, and Scott Shenker. A declarative sensornet architecture. *SIGBED Rev.*, 4(3):55–60, 2007.

[27] David Chu, Joseph M. Hellerstein, and Tsung te Lai. Optimizing declarative sensornets. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 403–404, New York, NY, USA, 2008. ACM.

[28] Robert Braden, Ted Faber, and Mark Handley. From protocol stack to protocol heap: role-based architecture. *SIGCOMM Comput. Commun. Rev.*, 33(1):17–22, 2003.

[29] Tommaso Melodia, Mehmet C. Vuran, and Dario Pompili. The state of the art in cross-layer design for wireless sensor networks. *Wireless Syst./Network Architect., LNCS 3883*, page 78–92, 2006.

[30] Eli De Poorter, Benoît Latré, Ingrid Moerman, and Piet Demeester. Universal modular framework for sensor networks. In *International Workshop on Theoretical and Algorithmic Aspects of Sensor and Ad-hoc Networks (WTASA'07)*, pages pp 88 – 96, Miami, USA, June June 28-29, 2007.

Extending a Time-Triggered System by Event-Triggered Activities

Josef Templ, Johannes Pletzer, Wolfgang Pree, Peter Hintenaus, Andreas Naderlinger
C. Doppler Laboratory Embedded Software Systems
University of Salzburg
Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

***Abstract*—Time-triggered execution of periodic tasks provides the cornerstone of dependable real-time systems. In addition, there is often a need for executing event-triggered activities while the system would be otherwise idle. We first present the foundation of a time-triggered system based on the Timing Definition Language (TDL). Then we introduce event-triggered activities as an extension of the purely time-triggered programming model. If time-triggered and event-triggered activities exchange information among each other, the data flow must be synchronized such that reading unfinished output data is avoided. The paper describes a lock-free solution for these synchronization issues that is based exclusively on memory load and store operations and can be implemented efficiently on embedded systems without any operating system support. We also discuss the implications of our synchronization approach for the semantics of combined time-triggered and event-triggered execution in a dependable real-time system. A case study of an Inertial Navigation System (INS) illustrates these extensions.**

***Keywords*—Lock-free; Synchronization; Time-triggered; Event-triggered; Synchronous; Asynchronous; Activity; TDL**

I. INTRODUCTION

A dependable real-time system performs safety critical tasks by periodic execution of statically scheduled activities [18]. The pre-computed schedule guarantees that the timing requirements of the system will be met in any case by taking the worst case execution time into account. Such operations are called time-triggered (alias synchronous) activities. The timing requirements of such activities are typically in the range of milliseconds or sometimes even below.

In addition, many dependable real-time systems execute event-triggered (alias asynchronous) activities that are, for example, triggered by the occurrence of an external hardware interrupt or any other kind of trigger. In the context of a dependable real-time system such asynchronous activities are considered to be not as time critical as synchronous tasks are, and can therefore be executed in a background thread while the CPU is otherwise idle.

Adding asynchronous activities to a time-triggered system could be done in a platform specific way by directly programming at the level of the operating system or task monitor. However, this approach has two drawbacks: (1) it is highly platform dependent and (2) it does not support proper synchronization of data exchanged between synchronous and asynchronous activities.

In order to tackle both problems we extended a tool chain [2, 3] for time-triggered systems by asynchronous activities. This tool chain supports the Timing Definition Language (TDL), which allows one to specify the timing behavior of a real-time system in a platform independent way. TDL separates the specification of the timing behavior from the implementation of the tasks. We extended TDL by a notation for asynchronous activities and provided a runtime system for this extended TDL on a number of target platforms [4].

The resulting lock-free approach for data flow synchronization [1] is not specific for TDL but—we believe—can be applied to other time-triggered systems that need to be extended with asynchronous activities. Our synchronization approach can be implemented efficiently without any operating system support such as monitors [5] or semaphores [6] and it avoids the need for dynamic memory allocation and the danger of deadlocks and priority inversions. It also keeps the impact of event-triggered activities on the timing of time-triggered activities as low as possible. For more information on non-blocking synchronization techniques please refer to [7, 8].

Note: This paper represents an extended version of [1]. It presents (1) the lock-free synchronization approach for data flow between event-triggered and time-triggered activities in the context of the TDL project, where the approach had been developed, and it adds (2) a non-trivial example that shows the integration of event-triggered and time-triggered activities. We shall start with an explanation of TDL's time-triggered programming model and language features that are relevant for understanding the proposed extensions and the example.

II. TIME-TRIGGERED ACTIVITIES IN TDL

A particularly promising approach towards a high-level component model for real-time systems has been laid out by the introduction of the so-called Logical Execution Time (LET [12]), which abstracts from the physical execution time on a particular platform and thereby abstracts from both the underlying execution platform and the communication topology. Thus, it becomes possible to change the underlying platform and even to distribute components between different nodes without affecting the

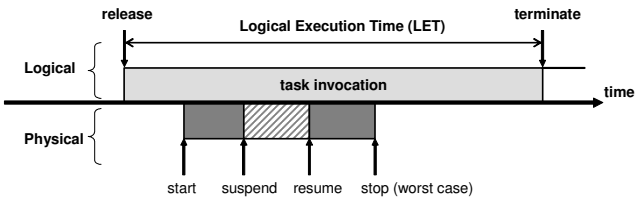


Figure 1. Logical Execution Time abstraction

overall system behavior. LET means that the observable temporal behavior of a task is independent from its physical execution. It is only assumed that physical task execution is fast enough to fit somewhere within the logical start and end points.

Figure 1 shows the relation between logical and physical task execution. The inputs of a task are read at the release time and the newly calculated outputs are available at the terminate time. Between these, the outputs have the value of the previous execution. LET provides the cornerstone to deterministic behavior, platform abstraction and well-defined interaction semantics between parallel activities. It is always defined which value is in use at which time instant and there are no race conditions or priority inversions involved.

TDL is a LET-based language. The basic construct that represents an executable entity is called a task. Several tasks can be executed in parallel and each task invocation may have its specific LET and execution rate. As real-time systems typically exhibit various modes of operations, TDL allows the specification of such modes. A TDL mode consists of a set of periodically executed activities. In addition to task invocations, an activity can also be an actuator update or a mode switch. The LET of a task is always greater than zero, whereas actuator updates and mode switches are executed in logical zero time (LZT). As the top-level structuring concept, TDL provides the notion of a module. Figure 2 sketches a sample module with two modes containing two cooperating tasks each.

TDL modules support an export/import mechanism

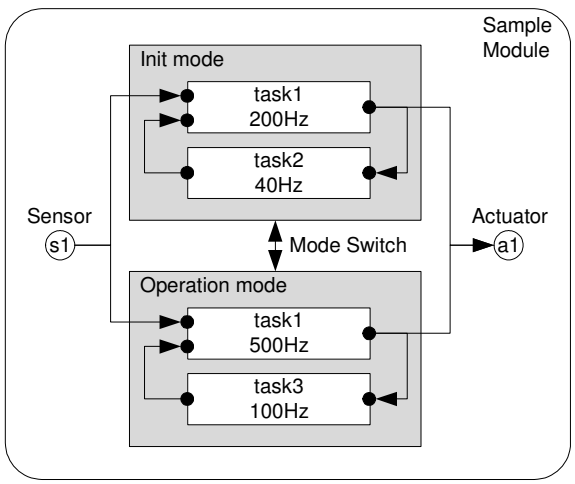


Figure 2. Schematic representation of a TDL module

similar to modern general purpose programming languages such as Java or C#. A service provider module may export e.g. a task's outputs, which in turn may be imported by a client module and used as inputs for the client's computations. Every module provides its own distinguished start mode. Thus, all modules execute in parallel or in other words, a TDL application can be seen as the parallel composition of a set of TDL modules. It is important to note that LET is always preserved, that is, adding a new module will never affect the observable temporal behavior of other modules. It is the responsibility of internal scheduling mechanisms to guarantee conformance to LET, given that the worst-case execution times (WCET) and the execution rates are known for all tasks.

Parallel tasks may depend on each other, i.e. the output of one task may be used as the input of another task. All tasks are logically executed in sync and the dataflow semantics is defined by LET. There is always a distinguished time base which drives all time-triggered activities and that is why they are also called synchronous activities.

The following TDL source code corresponds with the schematic representation of the module in Figure 2.

```
module Sample {

  sensor int s1 uses getS1;
  actuator int a1 uses setA1;

  task task1 {
    input int i1; int i2;
    output int o1;
    uses t1Impl(i1,i2,o1);
  }

  task task2 {
    input int i1;
    output int o1;
    uses t2Impl(i1,o1);
  }

  task task3 {
    input int i1;
    output int o1;
    uses t3Impl(i1,o1);
  }

  start mode Init [period=25 ms] {
    task
      [freq=5] task1(task2.o1,s1);
      [freq=1] task2(task1.o1);
    actuator
      [freq=5] a1 := task1.o1;
    mode
      [freq=1] if toOperation(s1)
               then Operation;
  }

  mode Operation [period=10 ms] {
    task
      [freq=5] task1(task3.o1,s1);
      [freq=1] task3(task1.o1);
    actuator
      [freq=5] a1 := task1.o1;
    mode
      [freq=1] if toInit(s1)
               then Init;
  }
}
```

The module is named Sample. It declares a sensor s1 and an actuator a1, both of type integer. The sensor value is provided by the external getter function getS1 and the actuator value is written by the external setter function setA1. Functionality code such as getters, setters, or task implementation functions are not implemented in TDL but must be provided in a conventional programming language such as C. The module declares three tasks task1, task2, and task3. The module further declares two modes Init and Operation. The mode Init has a period of 25ms and it executes task1 with a frequency of 5 (=200Hz) and task2 with a frequency of 1 (=40Hz). The actuator a1 is also updated 5 times per mode period with the new value of output port o1 of task task1. Once every 25ms the external boolean function toOperation decides whether to resume with the initialization

or to switch to the second mode Operation. The mode Operation executes the tasks task1 and task3 and also updates the actuator a1.

III. ADDING EVENT-TRIGGERED ACTIVITIES

We assume that time-triggered activities have the highest priority in a dependable real-time system. The runtime system executes a pre-computed schedule and reads inputs and writes outputs at well-defined time instants, which are synchronized with a global time base such as the clock of a time-triggered bus system.

Asynchronous activities must not interfere with the timing properties of synchronous activities. This is achieved by running asynchronous activities in a thread with lower priority than synchronous activities. However, things get more complicated when synchronization of the data flow is involved, as will be described below.

A. Asynchronous activities in TDL

TDL supports three kinds of synchronous activities. Task invocations and actuator updates also give sense when triggered asynchronously and should therefore be supported. Mode switches affect the time-triggered operation of a module and are therefore not supported as asynchronous activities.

An asynchronous task invocation consists of (1) reading input data (also called input ports), (2) execution of the task's body, and (3) writing of output data (also called output ports). With respect to synchronization issues, actuator updates do not introduce new problems because they can be seen as a special case of a task invocation. Figure 3 shows the task model that we assume.

The execution of a task's body is independent of the environment if input reading and output writing are separated from the implementation. Therefore we assume that internal copies of all input and output ports are maintained by the system. The task's body operates exclusively on these internal port copies.

Reading of input data may involve a sequence of memory copy operations that could be preempted by a hardware interrupt or by a time-triggered operation, which has higher priority. Therefore we need to synchronize input data reading with the rest of the system such that all input ports are read atomically.

Like input data reading, writing of output data is a sequence of memory copy operations that could be preempted by a hardware interrupt or by a time-triggered operation. It needs to be synchronized with the rest of the system such that all output ports are updated atomically.

B. Triggers for asynchronous activities

Asynchronous activities may be triggered by different events. We have identified the following three kinds of trigger events, which are consequently supported in our extension of TDL:

1) Hardware interrupt

A (non-maskable) hardware interrupt has the highest priority in the system. It may even interrupt synchronous activities. We must therefore take care that the impact of hardware interrupts on the timing of synchronous activities is minimized. Hardware interrupts may be used e.g. for connecting the system with asynchronous input devices.

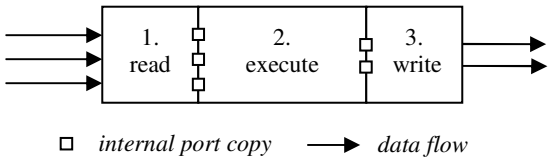


Figure 3. Assumed task model

2) Asynchronous timer

A periodic or a single-shot asynchronous timer may be used as a trigger. Such a timer is independent from the timer that drives the synchronous activities because it introduces its own time base. An asynchronous timer may for example be used as a watchdog for monitoring the execution of the time-triggered operations.

3) Port update

Updating an output port may be considered an event that triggers an asynchronous activity. We assume that both a synchronous and an asynchronous port update may be used as a trigger event. In case of a synchronous port update, i.e. a port update performed in a time-triggered activity, we must take care that the impact on the timing of the synchronous activities is minimized. Port update events may e.g. be used for limit monitoring or for change notifications.

C. Semantics of asynchronous activities

Obviously, the triggering of an asynchronous activity must be decoupled from its execution. In addition, reading input ports for an asynchronous activity must be done at the time of execution, not at the time of triggering. Thereby we move as much work as possible into the asynchronous part and minimize the impact of trigger events on the timing of synchronous activities, which is particularly important for hardware interrupts and synchronous port updates.

If multiple different asynchronous activities are triggered, the question arises whether they should be executed in parallel or sequentially in a single thread. We opted for the sequential case because (1) on some embedded systems there is no support for preemptive task scheduling and (2) because data flow synchronization is simplified as will be shown later. In practice, we expect this not to be a severe restriction because time critical tasks will be placed in the synchronous part anyway.

We assume that asynchronous activities that are registered for execution may have different priorities assigned. The set of registered events thus forms a priority queue where the next activity to be processed is the one with the highest priority.

If one and the same asynchronous activity is triggered multiple times before its execution, the question arises if it should be executed only once or multiple times, i.e. once per trigger event. We opted for executing it only once because this avoids the danger of creating an arbitrary large backlog of pending activities at runtime if the CPU cannot handle the workload. In addition this decision also simplifies the

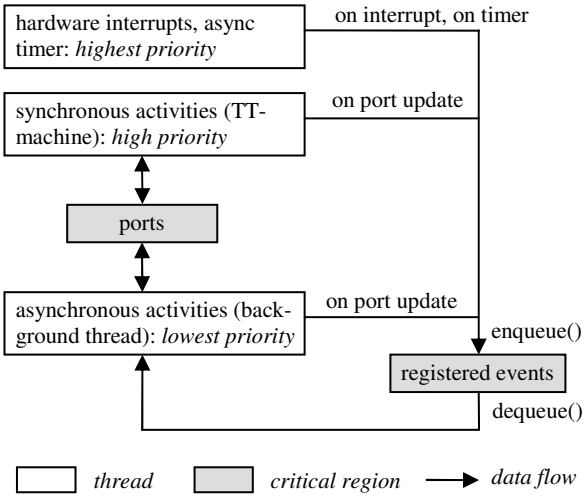


Figure 4. Threads and critical regions

mechanism for registering trigger events as will be shown later.

The following list summarizes our design decisions:

- Triggering of an asynchronous activity is decoupled from its execution.
- Reading input ports for an asynchronous activity is done at the time of execution, not at the time of triggering.
- Asynchronous activities are executed sequentially.
- The execution order of asynchronous activities is based on priorities.
- If one and the same asynchronous activity is triggered multiple times before its execution, it is executed only once.

IV. THREADING AND SYNCHRONIZATION

Figure 4 outlines the threads involved including their priority and the critical regions. The time-triggered activities are represented by a thread named TT-machine. This thread may need further internal threads but we assume that all synchronization issues are concentrated in a single thread that coordinates the time-triggered activities. It should also be noted that an asynchronous timer thread could also run at a lower priority as long as it is higher than the priority of the asynchronous activities.

The following situations that need synchronization can be identified and will be described below in more details: (1) Access to the priority queue of registered events. (2) Reading the input ports for an asynchronous activity. This must not be interrupted by the TT-machine. (3) Updating the output ports of an asynchronous activity. This must be finished before the TT-machine uses the ports.

A. *The priority queue of registered events*

As mentioned before, asynchronous events are not executed immediately when the associated trigger fires but need to be queued for later execution by the background thread. Since asynchronous events may be associated with a priority, we need a data structure that allows us to register an event and to remove the event with the highest priority. Such a data structure is commonly referred to as a priority queue. It provides two operations enqueue and dequeue, which insert and remove an entry with the property that the element being removed has the highest priority. A number of algorithms exists for implementing priority queues with logarithmic behavior of the enqueue and dequeue operation. However, in our case it is more important to minimize the run time of enqueue in order to minimize its impact on the timing of synchronous activities.

	priority	pending
event 0	0	true
event 1	2	false
event 2	2	false
event 3	1	true

Figure 5. Array representation of trigger events

Elements are enqueued when an asynchronous event occurs and the event is not yet in the queue. As mentioned earlier, an event can be a hardware interrupt, an asynchronous timer event, or a port update event. Port updates may origin from an asynchronous task or from a synchronous task that is executed by the TT-machine. enqueue will never be preempted by dequeue, however, enqueue may be preempted by another enqueue operation.

Elements are dequeued by the single background thread that executes asynchronous activities. This thread may be preempted by interrupts and by the TT-machine. Thus, dequeue may be preempted by enqueue operations.

As shown by the example in Figure 5 we chose an array representation of the triggerable events because this is both thread safe and provides for a fast and constant time enqueue operation. We use a Boolean flag per event that signals that an event is pending. The flag is cleared when an event is dequeued. From that time on it may be set again when the associated trigger fires. The flag remains set when the same trigger fires again while the flag is already set. The thread-safe enqueue operation boils down to a single assignment statement and the dequeue operation becomes a linear search for the event with the highest priority over all pending events. Registering an event from a non-maskable interrupt or from a synchronous port update thereby has only a negligible effect on the timing behavior of synchronous activities. The linear search in the background thread is expected to be acceptable for small to medium numbers of asynchronous events (< 100), which should cover all situations that appear in practice.

It should be noted that the array representation of the priority queue does not impose any restriction on the number of events the system can handle. There is one array element for every trigger and the number of triggers is known statically. Thus, the array can always be defined with the appropriate size.

The background thread for executing asynchronous operations is a simple infinite loop that runs with lower priority than the TT-machine thread. For a particular target platform there may be some refinements with respect to the CPU load, which is increased to 100% by permanently polling the event queue.

```
static Thread asyncThread = new Thread() {
  public void run() {
    for (;;) {
      int next = dequeue();
      if (next >= 0) {
        executeEvent(next);
      }
    }
  }
};
```

The procedure executeEvent is supposed to execute the asynchronous activity identified by next. Within its implementation there will be synchronization issues with respect to reading input ports and writing output ports as described below. Instead of showing the complete implementation, which depends on the particular environment, we will focus on the synchronization issues only.

B. *Reading the input ports for an asynchronous task*

While performing asynchronous reading of input ports the following situation may arise: An asynchronous input port reading involving multiple input ports (or at least multiple memory load operations) has been started. The first port has been copied. The second port has not yet been copied but the TT-machine preempts the background thread and updates the source ports. When the background thread continues it would read the next port, which has a newer value than the first port. Moreover, this situation may in principle occur multiple times when the TT-machine preempts the background thread after the second port has been read, etc. We have to make sure that reading all of the input ports is not preempted by the TT-machine. Since asynchronous activities don't preempt each other, we know that there can only be one such asynchronous input port reading that is being preempted. Therefore we can introduce a global flag that is set by the TT-machine in order to indicate to the background thread that it has been preempted. The background thread then has to repeat its reading until all of the ports are read without any preemption. The following Java code fragments outline a possible implementation.

Asynchronous port reading within executeEvent uses a loop in order to wait for a situation where input port reading is not preempted by the TT-machine. Therefore, our solution does not qualify as a wait-free nonblocking algorithm [7]. It should be noted, however, that (1) starvation cannot occur in the TT-machine and (2) in practice it does also not occur in the background thread because even in the unlikely case that the TT-machine's schedule reserves 100% of the CPU, this refers to the worst case execution time, which typically will not always be required.

```
do {
  ttmachineExecuted = false;
  //copy input ports
  ...
} while (ttmachineExecuted);
```

The relevant TT-machine code, which is assumed to be placed in a central procedure of the TT-machine named ttmachineStep may look like this:

```
void ttmachineStep() {
  ttmachineExecuted = true;
  //perform operations for this time instant
  ...
}
```

C. *Updating the output ports of an asynchronous task*

In the case of asynchronous output port updates the following situation may arise: An asynchronous output port update involving multiple output ports (or at least multiple memory store operations) has been started. The first port has been copied. The second port is not yet copied but the TT-machine preempts the background thread and reads both output ports. Now one port is updated but the second is not. Since this interruption cannot be avoided, we must find a way for proper synchronization.

Since we assumed earlier that updating the output ports is separated from the implementation of a task, we can encapsulate the output port update operations of a task in a helper procedure that we call the task's *termination driver*. Since asynchronous activities don't preempt each other, we know that there can only be one such termination driver being preempted and it suffices to make that very instance available to the TT-machine by means of a global variable. Whenever the TT-machine performs its next step, it checks first if a termination driver has been interrupted. If so, it simply re-executes this driver! This means that the driver may be executed twice, once by the background thread and once by the TT-machine. This is only possible if the driver is *idempotent* and *reentrant*, i.e. its preemption and repeated execution does not change its result. Fortunately, termination drivers have exactly this property because they do nothing but memory copies and the source values are not modified between the repeated driver executions. The source values are the internally available results of the most recent invocation of this asynchronous task and only a new task invocation can change them. Such a task invocation, however, will not happen because the background thread executes all asynchronous activities sequentially.

It should be noted that the property of idempotency does not hold for copying input ports as discussed in the previous subsection because a preemption by the TT-machine may alter the value of a source port that has already been copied. This means that we really need two ways of synchronization for the two cases.

It should also be noted that setting the driver identity must be an atomic memory store operation. If storing e.g. a 32 bit integer is not atomic on a 16-bit CPU, an additional Boolean flag can be used for indicating to the TT-machine that a driver has been assigned. This flag must of course be

set after the assignment of the driver's identity. If this initial sequence of assignments is preempted, the TT-machine will not re-execute the driver and that is correct because the driver has not yet started any memory copy operations.

The following Java code outlines the implementation of asynchronous task termination drivers and the corresponding code in the TT-machine. Setting, testing and clearing the driver identity is kept abstract because the details may vary between target platforms. Since Java lacks function pointers we use an integer id and a switch statement instead. Variations, e.g. using C function pointers or Java singleton classes, are of course possible.

```
void callDriver(int id) {
  switch (id) {
    ...
    case X: //termination driver for async task X
      assignAsyncTerminateDriverID(X);
      //perform memory copy operations
      ...
      clearAsyncTerminateDriverID();
      break;
    ...
  }
}
```

The relevant TT-machine code including the code introduced in the previous subsections looks like this:

```
void ttmachineStep() {
  ttmachineExecuted = true;
  if (asyncTerminateDriverIDassigned()) {
    callDriver(asyncTerminateDriverID);
  }
  //perform operations for this time instant
  ...
}
```

It suffices to clear the registered termination driver at the end of the termination driver itself. There is no need to do it after callDriver() in ttmachineStep because the driver's re-execution will clear it anyway.

The resulting runtime overhead for supporting asynchronous operations in the TT-machine is the assignment of the ttmachineExecuted flag and the test for the existence of a preempted asynchronous task termination driver, which is acceptable because this happens only once per TT-machine step. In case of preempting such a driver the time for re-execution must be added. When a port update trigger is used, then the enqueue operation is also a small constant time overhead that affects the TT-machine. There is no other runtime overhead for integration of event-triggered activities in the TT-machine.

V. MEASUREMENT RESULTS

Table I shows the time needed for various operations on different platforms. The platform named *MicroAutoBox* uses a PowerPC 750FX CPU running at 800 MHz and the Microtec C compiler version 3.2 with optimization level 5. The platform runs the dSPACE Real-Time Kernel as its operating system. The platform named *ARM* uses an ARM7 TDMI CPU running at 80 MHz and the GNU C compiler with optimization level 2 and runs without an operating system. The platform named *RENESAS* uses a Renesas M32C/85 CPU running at 24 MHz and the GNU C compiler version 4.1 with optimization level 3. The platform runs the Application Execution System (AES) provided by DECOMSYS and executes the programs from read-only memory, which slows down the execution. This system does not support external interrupts for user level programs. The platform named *SHARC* uses an Analog Devices SHARC ADSP-21262 CPU running at 200 MHz and the VisualDSP++ C compiler version 5.0 with maximum optimization level.

TABLE I. MEASUREMENT RESULTS [NANOSECONDS]

Platform (MHz)	Interrupt	Port Update	dequeue N
MicroAutoBox (800)	420	8	11 * N + 60
SHARC (200)	1030	72	30 * N + 110
ARM (80)	700	200	287 * N + 500
RENESAS (24)	N.A.	1200	790 * N + 2500

The column *Interrupt* shows the time needed for an external hardware interrupt trigger, which includes the interrupt handling overhead and the enqueue operation. The column *Port Update* shows the time needed for a synchronous port update trigger, which consists only of the enqueue operation. The column *dequeue N* shows the time needed for the search for the next event to be processed as a linear function of the array size *N*. All timings are given in nanoseconds.

The values shown in the columns *Interrupt* and *Port Update* are critical for the timely execution of synchronous operations as they impose an overhead that may affect the TT-machine. Even on the slowest platform the required time is only slightly above one microsecond. In comparison with the *ARM* platform, the *Interrupt* time for *MicroAutoBox* shows that the operating system introduces a significant overhead.

The values in the column *dequeue N* only affect the background thread and are not visible to the TT-machine. On the slowest platform a time of 81.5 microseconds results for $N = 100$, which means that response times in the range of milliseconds can easily be achieved for asynchronous operations.

VI. IMPLEMENTATION

We have implemented the proposed solution in the context of the TDL tool chain. Currently we support two networked target platforms, (1) the dSPACE MicroAutoBox, which is a widely used prototyping platform for embedded systems in the automotive industries, and (2) the NODE RENESAS platform provided by DECOMSYS (now Elektrobit). Furthermore, we are experimenting with standalone platforms including bare hardware based on an ARM7 and a SHARC processor.

Both networked systems are programmed in C and support a FlexRay [19] bus interface and the time-triggered activities are synchronized with FlexRay's global time base. The availability of a high-level description language (TDL)

for timing properties as well as for asynchronous activities allowed us to generate the required glue code such as the event table, the termination drivers and all the code needed for the background thread and for data flow synchronization automatically. Even when we added support for distributing the data flow across multiple nodes we relied on the data flow synchronization approach presented in this paper.

In analogy to handling the execution of asynchronous activities in a background thread, network frames that communicate the outputs of asynchronous task invocations must be sent in a way such that they do not interfere with time-triggered frames, i.e. data sent by synchronous activities. Depending on the communication protocol being used, this can be done by configuring such asynchronous frames as low priority frames (if the bus protocol supports priorities) or by assigning them a designated section in the communication cycle (typically done when using time-triggered protocols such as FlexRay or TTEthernet).

Our implementations use the FlexRay communication protocol. FlexRay is a time-triggered protocol targeted at the automotive industry. It has a significantly higher bandwidth than other field bus protocols and is designed to handle safety critical applications such as steer-by-wire systems. A FlexRay communication cycle constantly repeats itself and it consists of a mandatory static part and an optional dynamic part. The static part is divided into equally sized slots which are statically assigned to specific nodes, thereby guaranteeing uninterrupted transmission. The dynamic segment also has a static size, but it is dynamically allocated to nodes upon runtime. We use the static segment for synchronous frames and the dynamic segment for asynchronous frames.

VII. EXAMPLE

As an example for a real-world TDL application, we present an augmented strap down inertial navigation system (INS) [13] designed for computing the position, velocity, and attitude of a sailing vessel at sea. The example is split into several modules and uses asynchronous activities for connecting asynchronous I/O with the time-triggered navigation system core.

An INS determines the position of a vehicle with respect to some (inertial) reference system by measuring the three accelerations along and the three angular velocities around the vehicle's axes with respect to the reference system, using three accelerometers and three gyroscopes which are firmly attached to the vehicle's body. By solving the equations of motion the INS computes the position, velocity, and attitude of the vehicle. An augmented INS uses additional inputs, such as position information from a GPS receiver and compass headings, to correct the drift of the inertial sensors.

A. Hardware

The hardware (see Figure 6) for the augmented INS consists of an Analog Devices ADSP-21262 Signal Processor [14], a LAN interface with TCP/IP functionality in firmware, an ADIS family micromechanical inertial sensor [15] and a two axis fluxgate compass [16]. Besides a floating point signal processing core with a peak SIMD performance of 1.2 GFlops, the ADSP-21262 contains an I/O processor that is capable of managing several block transfers between memory and periphery simultaneously. The inertial sensor is connected to the signal processor using an SPI bus [17]. It samples the rotations around the three axes of the vehicle and the accelerations along these axes 819.7 times per second. The excitation coil of the fluxgate compass is attached to the ADSP-21262 using a sampling DA converter. The two sense coils of the compass are connected to two sampling AD converters. All three converters operate at 48K samples per second. For determining the heading of the vehicle the compass has to be excited periodically via the DA converter and its response measured via the two AD converters.

B. TDL definitions

A TDL module starts with its name and the list of imported modules. When importing a module it is possible to define an abbreviation for it:

```
module INS {
  import Kalman as K;
  ... //constants, types, ports, tasks, modes, asyncs
}
```

Next, constants and types can be declared. Besides the basic types as in Java, TDL supports structures and arrays of constant size. By denoting a name public any importing module is allowed to refer to this name:

```
public const NavPeriod = 1220us;
public type Vector = struct {
  float x, y, z;
};
type FluxBuffer = int[120];
```

The sensor and actuator declarations that follow define the hardware inputs and outputs used by the module. With the uses clause one specifies the name of the external getter or setter function to access the hardware:

```
public sensor InSens in uses getInertial;
```

The global output ports come next. Global output ports are not dedicated to an individual task but may be used by all

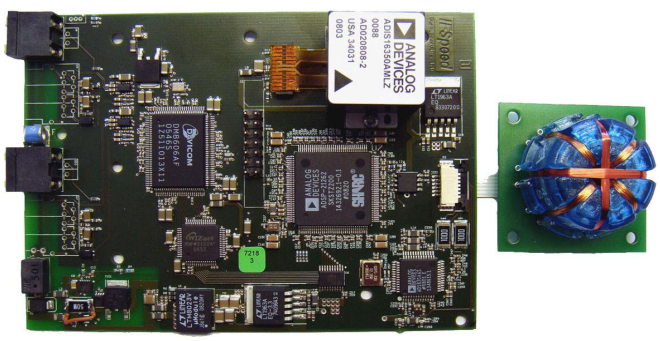


Figure 6. INS Hardware

tasks in the module. A port is updated at the end of the LET of the task that writes it:

```
public output Vector pos;
```

Next, the tasks with their input, output, and state ports are declared. In the uses clause the name of the external function providing the task's functionality is specified. The last four parameters in the example below refer to global output ports:

```
task solveMotion {
  input InSens in; Vector cPos; Vector cVel; Quaternion cAtt;
  uses deadReconing(in, cPos, cVel, cAtt, pos, vel, att, time);
}
```

A mode is a set of activities, i.e. task invocations, actuator updates and mode switches, which are executed periodically with the mode period *p*. For each activity a frequency *f* and, optionally, a guard can be specified. For a task invocation the LET of this invocation is p / f. In the following mode declaration, the period is set to NavPeriod. Both the solveMotion and acquireMagHandling tasks are invoked once per period so that the LET of both tasks is NavPeriod. The mode Navigation is declared as start mode which means that the execution of the module starts with this mode.

The names of entities imported from some other module are qualified either by the name of the imported module or by its abbreviation (e.g. K.pos):

```
start mode Navigation [period = NavPeriod] {
  task [freq = 1] solveMotion(in, K.pos, K.vel, K.att);
  task [freq = 1] acquireMagHeading();
}
```

Finally, asynchronous activities can be specified as in the following code fragment. Once the interrupt named iGPS occurs, the task receiveGPS is enqueued for later processing and executed by a background thread. The mapping of the logical interrupt name iGPS to a particular interrupt line is platform dependent and must be specified outside the TDL source code.

```
asynchronous {
  [interrupt = iGPS, priority = 2] receiveGPS(INS.time);
}
```

C. Complete TDL modules

In our hardware three independent asynchronous timing sources are visible to the software: the processor clock, the sampling events of the inertial sensor, and the sampling events of the DA and AD converters. Choosing the sampling events of the inertial sensor as the time base for the synchronous activities allows us to solve the equations of motion and to consider other sensor inputs using Kalman filters [16] synchronously with the inertial data stream.

The module INS processes the inputs of the inertial sensor and of the fluxgate compass. For each new inertial measurement the task solveMotion advances the estimates for the position, the velocity, and the attitude of the vehicle. Quaternions are used for the representation of attitudes.

The excitation of the fluxgate compass is supplied with a continuous data stream by the I/O processor of the ADSP-21262. The data streams from the two sense coils are captured and transferred to buffers in memory by I/O processor. The size of the array type FluxBuffer is made large enough to hold the data acquired during one period of the mode Navigation for both sense coils. A state port (essentially a private static variable) containing two buffers, one for capturing and one for processing, is introduced for avoiding any array copy operations. Task acquireMagHeading is associated with *two* external functions (TDL task splitting), (1) a long running function integrateFluxGate, and (2) an LZT function exciteFluxGate indicated by the attribute release. The basic idea is that the LZT function is called first at the LET start and provides the new output values in a very short time, closely approximating LZT. The long running function is executed during the LET. The LZT function exciteFluxGate restarts the data stream to the fluxgate compass and switches between the two buffers at the start of the LET of task acquireMagHeading. By invoking acquireMagHeading in mode Navigation with the same frequency as solveMotion the compass is synchronized to the inertial sensor.

The module INS counts the sampling events in the task solveMotion to provide a time base for the other modules. The period of 1220 microseconds for the mode Navigation is the time that passes between two consecutive samples of the inertial sensor.

```
module INS {
  import Kalman as K;
  public const NavPeriod = 1220 us;
  public type Vector = struct {float x, y, z;};
  public type Quaternion = struct {float x0, x1, x2, x3;};
  public type InSens = struct {
    float aX, aY, aZ;
    float omegaX, omegaY, omegaZ;
  };
  type FluxBuffer = int[120];
  type FluxDoubleBuffer = struct {
    byte bufState; FluxBuffer flux1, flux2;
  }
  public sensor InSens in uses getInertial;
  public output Vector pos; Vector vel; Quaternion att;
  public output long time; Vector mHead;
  task solveMotion {
    input InSens in; Vector cPos; Vector cVel; Quaternion cAtt;
    uses deadReconing(in, cPos, cVel, cAtt, pos, vel, att, time);
  }
  task acquireMagHeading {
    state FluxDoubleBuffer flux;
    uses [release] exciteFluxGate(flux);
    uses integrateFluxGate(flux, mHead);
  }
  start mode Navigation [period = NavPeriod] {
    task [freq = 1] solveMotion(in, K.pos, K.vel, K.att);
    task [freq = 1] acquireMagHeading();
  }
}
```

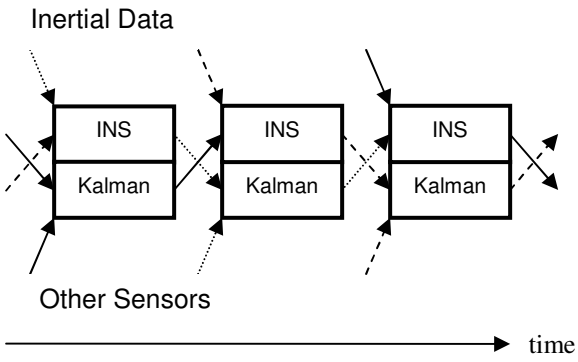


Figure 7. Data Flow

The module GPS receives position and velocity information from a GPS receiver via the LAN interface typically once per second. The LAN interface chip has an internal memory buffer. It activates interrupt iGPS of the signal processor to demand service.

To maintain a timing relationship with the inertial data each dataset from the GPS receiver is time stamped as soon as it is received.

```
module GPS {
  import INS;
  public output INS.Vector pos; INS.Vector vel; long timeStamp;
  public task receiveGPS {
    input long time;
    uses getGPSData(time, pos, vel, timeStamp);
  }
  asynchronous {
    [interrupt = iGPS, priority = 2] receiveGPS(INS.time);
  }
}
```

On power on, the module Kalman aligns the estimates for the vehicle's position, velocity, and attitude. Once a good initial fix has been achieved, it switches to Filter mode. It then combines the inertial measurement, the GPS position and velocity, and the compass heading into an estimate of the vehicle's position, velocity, and attitude.

```
module Kalman {
  import INS; GPS;
  public output INS.Vector pos; INS.Vector vel;
  public output INS.Quaternion att; long stamp;
  public task align {
    input INS.InSens in; INS.Vector mHead; long time;
    uses doAlign(in, mHead, time, pos, vel, att, stamp);
  }
  public task filter {
    input INS.Vector nPos; INS.Vector nVel; INS.Quaternion nAtt;
    input INS.Vector mHead; long time;
    input INS.Vector gpsPos; INS.Vector gpsVel; long gpsStamp;
    uses doKalmanFilter(nPos, nVel, nAtt, mHead, time, gpsPos,
                        gpsVel, gpsStamp, pos, vel, att, stamp);
  }
  start mode Align [period = INS.NavPeriod] {
    task [freq = 1] align(INS.in, INS.mHead, INS.time);
    mode [freq = 1] if isAligned() then Filter;
  }
  mode Filter [period = INS.NavPeriod] {
    task [freq = 1] filter(INS.pos, INS.vel, INS.att, INS.mHead,
                           INS.time, GPS.pos, GPS.vel, GPS.timeStamp);
  }
}
```

The module NavReporter finally communicates the navigational solutions to the outside world. Whenever a new measurement is available, indicated by a port update on the port Kalman.stamp, it makes it available on the LAN. The asynchronous operation uses the default priority, which is the lowest priority (0). Reading the input ports (K.pos, K.vel etc.) is an atomic operation.

```
module NavReporter {
  import Kalman as K; INS;
  public task reportNav {
    input INS.Vector pos; INS.Vector vel; INS.Quaternion att;
      long stamp;
    uses doReporting(pos, vel, att, stamp);
  }
  asynchronous {
    [update = K.stamp] reportNav(K.pos, K.vel, K.att, K.stamp);
  }
}
```

Figure 7 depicts the dataflow between the modules INS and Kalman. Arrows of the same style indicate measurements that are combined by the Kalman filter into one navigation solution. Note that it takes two sampling periods of the inertial sensor until the data arrives at the output ports of the Kalman filter. For slow moving vehicles like sailing vessels this deems satisfactory. For faster moving vehicles one would combine the two functions solveMotion and doKalmanFilter in one task.

VIII. Related Work

The xGiotto language [9] also aims at the integration of time-triggered and event-triggered activities. xGiotto's compiler is supposed to perform a static check for the absence of race conditions. Due to the specific design of xGiotto, a precise check is possible but not in polynomial time. Therefore, only a conservative check is done in the compiler. We do not need such a check at all as we defined appropriate semantics for event-triggered activities and use the proposed synchronization mechanisms for their integration into a time-triggered system.

RT-Linux [10] is an extension of the Linux operation system which adds a high priority real-time kernel task and runs a conventional Linux kernel as a low priority task. Its interrupt handling mechanism is similar to what we propose for the event queue as all interrupts are initially handled by the real-time kernel and are passed to a Linux task only when there are no real-time tasks to be run. In our approach, the only immediate reaction to an interrupt is its registration in

the priority queue so that it can be processed later when no time-triggered activity is executed.

In [11] a non-blocking write (NBW) protocol is presented. The writer is executed by a separate processor and is not blocked. It updates a concurrency control field (CCF) which indicates whether it currently writes data to a shared variable. The reader uses the CCF to loop until no write operation is executed while it reads from the shared data structure. This relates closely to our synchronization strategy for reading input ports for an asynchronous activity. In our case the writer would be the TT-machine which is not blocked.

A comprehensive overview of the field of non-blocking synchronization can be found in [8]. Among other techniques, it also describes a so-called roll-forward synchronization approach by means of a helper function, which looks similar to the one we used for synchronizing output port writing.

IX. CONCLUSIONS

We have presented the extension of a time-triggered system by event-triggered activities. Data flow between time- and event triggered activities must be carefully synchronized in order to avoid race conditions. We have shown that a non-blocking lock-free solution for data flow synchronization is indeed possible. Our solution does not need any operating system support such as monitors or semaphores and thereby avoids dynamic memory operations and the danger of deadlocks and priority inversions. There is also no need for switching off interrupts and the solution also works in a shared-memory multiprocessor system where the time-triggered and event-triggered activities are performed on separate CPUs. Our approach relies exclusively on atomic memory load and store operations, which are provided by every CPU in hardware. An appropriate semantics for asynchronous activities helped us to keep the solution simple and efficient.

ACKNOWLEDGMENT

We want to thank Gernot Turner for providing us with the hardware for the INS.

REFERENCES

[1] J. Templ, J. Pletzer, W. Pree, “Lock-Free Synchronization of Data Flow Between Time-Triggered and Event-Triggered Activities in a Dependable Real-Time System,” In Proceedings of the 2nd International Conference on Dependability (DEPEND 2009), Athens, Greece, 2009.

[2] W. Pree and J. Templ, “Modeling with the Timing Definition Language (TDL),” Proceedings ASWSD 2006, LNCS 4922, 133-144, Springer, 2008.

[3] J. Templ, “Timing Definition Language (TDL) Specification 1.5,” Technical Report, University of Salzburg, 2008, http://softwareresearch.net/pub/T024.pdf.

[4] J. Templ, J. Pletzer, and A. Naderlinger, “Extending TDL with Asychronous Activities,” Technical Report, University of Salzburg, 2008, http://softwareresearch.net/pub/T022.pdf.

[5] C. A. R. Hoare, “Monitors: An Operating System Structuring Concept,” Comm. ACM 17 (10), 549–557, 1974.

[6] E. W. Dijkstra, “Cooperating sequential processes,” in “Programming Languages,” Academic Press, New York, 1968.

[7] M. P. Herlihy, “A Methodology For Implementing Highly Concurrent Data Structures,” Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming, ACM, New York, 1990.

[8] M. B. Greenwald, “Non-Blocking Synchronization and System Design,” PhD Thesis, CS-TR-99-1624, Stanford U., 1999.

[9] A. Ghosal, T. A. Henzinger, C. M. Kirsch, and M. A. A. Sanvido, “Event-driven programming with logical execution times,” in “Hybrid Systems Computation and Control,” Lecture Notes in Computer Science 2993, Springer, 2004.

[10] V. Yodaiken and M. Barabanov, “A Real-Time Linux,” Proceedings of the Linux Applications Development and Deployment Conference (USELINUX), Anaheim, CA, 1997.

[11] H. Kopetz and J. Reisinger, “The non-blocking write protocol NBW,” Proceedings of the 14th IEEE Symposium on Real-Time Systems, 131-137, IEEE, New York, 1993.

[12] T. Henzinger, B. Horowitz, C. Kirsch, “Giotto: A time-triggered language for embedded programming,” In Proc. of EMSOFT, LNCS 2211, pages 166–184. Springer, 2001

[13] D. H. Titterton and J. L. Weston, “Strapdown inertial navigation technology,” 2nd Ed. IEEE radar, sonar, navigation and avionics series 17, 1996. ISBN 978-0863413582

[14] Analog Devices, “SHARC Embedded Processor ADSP-21261/ADSP-21262/ADSP-21266, Data Sheet, Rev. E. 2008,“ Analog Devices, USA.

[15] Analog Devices, “Six Degrees of Freedom Inertial Sensor ADIS16364, Data Sheet, Rev PrA. 2008,” Analog Devices, USA.

[16] Autonnic, “AR45 Two Axis Magnetometer Component with Floating Core, Data Sheet, 2008,” Autonic Research, Great Britain.

[17] Freescale Semiconductor, “Serial Peripheral Interface (SPIV3) Block Description”.

[18] H. Kopetz, “Real-Time Systems - Design Principles for Distributed Embedded Applications,” ISBN 0792398947, Springer, 2007.

[19] R. Makowitz, C. Temple, “FlexRay - A Communication Network for Automotive Control Systems,” Proc. WFCS 2006, pp. 207–212.

Enabling Effective Dependability Evaluation of Complex Systems via a Rule-Based Logging Framework

Marcello Cinque*, Domenico Cotroneo*, Antonio Pecchia*
*Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II
Via Claudio 21, 80125, Naples, Italy
Email: {macinque, cotroneo, antonio.pecchia}@unina.it

***Abstract*—Field Failure Data Analysis (FFDA) is a widely adopted methodology to characterize the dependability behavior of a computing system. It is often based on the analysis of logs available in the system under study. However, current logs do no seem to be actually conceived to perform FFDA, since their production usually lacks a systematic approach and relies on developers' experience and attitude. As a result, collected logs may be heterogeneous, inaccurate and redundant. This, in turn, increases analysis efforts and reduces the quality of FFDA results.**

This paper proposes a rule-base logging framework, which aims to improve the quality of logged data and to make the analysis phase more effective. Our proposal is compared to traditional log analysis in the context of a real-world case study in the field of Air Traffic Control. We demonstrate that the adoption of a rule-based strategy makes it possible to significantly improve dependability evaluation by both reducing the amount of information actually needed to perform the analysis and without affecting system performance.

***Keywords*-Field Failure Data Analysis; Dependability Evaluation; Logging Rules; Automated Log Analysis.**

I. INTRODUCTION

Field Failure Data Analysis (FFDA) embraces several techniques aiming to characterize the dependability behavior of a computing system during the operational phase. Dependability of a computing system is the "ability to deliver service that can justifiably be trusted" [9]. FFDA-based dependability analysis relies on *natural*, i.e., not forced errors and failures, and it commonly exploits logs available in the system under study. Logs are files where applications and system modules register events related to their normal and/or anomalous activities. For this reason, logs "are one of the few mechanisms for gaining visibility of the behavior of the system" [2].

FFDA has shown its benefits over a wide range of systems even though logs are often an under-utilized resource [3], since their production is known to be a developer-dependent [4] and error-prone [5] task. Log production lacks a systematic approach and relies on developers' experience and attitude. In fact, crucial decisions about logging are left to the last phases of the software development cycle (e.g., coding). As a result, it is reasonable to state that current logs do not seem to be actually conceived to perform dependability evaluation.

Logged information can be heterogeneous and inaccurate [5], [6]. Heterogeneity may affect both *format* and *content*, and usually increases as the system complexity increases. Many current FFDA tools address format heterogeneity. Content heterogeneity is more challenging since the *meaning* of a logged event depends on what the developer actually intended to log. Inaccuracy is related to the presence of duplicate or useless entries as well as to the absence of relevant failure data. Error propagation phenomena, which result in multiple and apparently uncorrelated events [7], [8], represent a further threat for logs effectiveness. A widely adopted strategy to address this phenomena is to use an *one-fits-all* timing window to coalesce related events. However, this is usually performed without any awareness of the actual correlation among log messages [2]. The risk is to classify correlated failures as uncorrelated, and vice versa, thus leading to unrealistic and wrong results.

The mentioned issues make the analysis of failure data a very hard task. As a matter of fact it requires significant manual efforts and ad-hoc algorithms and techniques to remove useless data, to disambiguate events, and to coalesce correlated ones. These efforts are exacerbated in case of complex, networked systems composed by several software items, each of them with its own logging mechanisms. As a result, the quality of FFDA-based dependability evaluation may significantly reduce.

We believe that a promising solution to overcome this limitation is to re-think the way in which logs are produced and analyzed. A viable strategy is to provide software developers with a comprehensive *logging framework*, inspired by a high-level system model, which specifies rules to produce log events, and tools to automate their collection and analysis. Our proposal aims to improve the quality and the effectiveness of logged events with respect to traditional logging, to achieve accurate and homogeneous logs, which are ready to be analyzed with no further processing, and to make it possible to extract, even on-line, value-added information based on log events produced by individual system components.

This paper presents the concepts underlying our proposal by focusing on dependability evaluation of complex systems. More in details, we describe logging rules and algorithms

aiming (i) to unambiguously detect the occurrence and the location of a failure (in particular, in this paper the focus is on *timing* failures [9]), (ii) to trace error propagation phenomena induced by interactions within the system, and (iii) to enable effective dependability measurements. We also describe currently available automated log collection and analysis tools. We demonstrate the effectiveness of the proposed strategy, compared to a real-world logging subsystem, in the context of a case study in the field of the Air Traffic Control (ATC). The adoption of systematic logging rules significantly increases the quality of FFDA-based dependability evaluation (e.g., availability and time to failure) and makes it possible to achieve valuable insights about the behavior of the system in hand. We experience that the proposed framework allows reducing the amount of information actually needed to perform the analysis without affecting performance. In particular, log size decreases by more 94.3% and system performance is improved by more 12.1% when compared to the initial logging subsystem.

The rest of the paper is organized as follows. We describe related work in the area of FFDA in Section II while Section III presents the system model underlying the design of our framework. Rules to produce log events and algorithms enabling their on-line processing are presented in Sections IV and V, respectively, while Section VI describes the ongoing implementation of the proposed log collection and analysis infrastructure. We describe the reference case study and the experimental campaign in Section VII and results achieved with traditional logging techniques and the proposed framework in Sections VIII and IX, respectively. Section X provides the estimation of the overhead introduced by the proposed framework on the system in-hand while Section XI concludes the work.

The paper improves and extends the proposal presented in [1]. In particular (i) we describe an additional set of logging rules aiming to figure out the operational state of an entity, (ii) we provide a comprehensive framework to collect and analyze proposed rule-based logs, and (iii) we significantly improve the experimental campaign by performing in-depth availability and failure analyses of collected logs.

II. Related Work

FFDA studies commonly adopt log files as source of failure data. Logs are usually conceived as human-readable text files for developers and administrators to gain visibility in the system behavior, and to take actions in the face of failures. A programming interface usually allows applications to write events, i.e., lines of text in the log, according to developers' needs. Well-known examples of event logging systems are UNIX syslog [10] and Microsoft's event logger.

FFDA has shown its benefits over a wide range of systems during the last three decades. A non-exhaustive list includes, for example, operating systems [5], [4], control systems and mobile devices [11], [12], supercomputers [2], [13], and large-scale applications [14], [15], [16]. These studies contributed to gain a significant understanding on the failure modes of these systems, and made it possible to improve their successive generations [17].

Log analysis is usually done manually, by means of ad-hoc algorithms and techniques to remove useless data (e.g., housekeeping events [8], which report non-error conditions) to disambiguate events, and to coalesce correlated events. In particular, with respect to dependability evaluation, significant efforts are needed to identify system reboots and failure occurrences, which are used to estimate, for example, the system availability and Time To Failure. A commonly used approach to figure out a reboot signal from logs is to locate specific event patterns (e.g., [5]). On the other hand, the identification of failure-related log events is more challenging. This task usually requires a preliminary log inspection (e.g., to figure out events severity and error-specific keywords within the logged text) as well as procedures to cluster a set of related alerts to a single alert per failure [2].

It thus emerged the need for software packages which integrate a wide range of the state-of-the-art FFDA techniques, such as tools easing, if not automating, the data collection, coalescing, and modeling tasks. An example is MEADEP [18], which consists of four software modules, i.e., a data preprocessor for converting data in various formats to the MEADEP format, a data analyzer for graphical data-presentation and parameter estimation, a graphical modeling interface for building block diagrams, e.g., Weibull and k-out-of-n block, and Markov reward chains, and a model-solution module for availability/reliability estimation with graphical parametric analysis. Analyze NOW [19] is a set of tools tailored for networks of workstations. It embodies tools for the automated data collection from all the workstations, and tools for automating the data analysis task. In [20], [21] a tool for on-line log analysis is presented. It defines a set of rules to model and to correlate log events at runtime, leading to a faster recognition of problems. The definition of rules, however, strongly relies on log content and analysts' skills.

Despite these efforts, several works have pointed out the inadequacy of event logs to perform dependability evaluation. A study on Unix workstations [5] recognizes that logs may be incomplete or imperfect, and it describes an approach for combining different data sources to improve system availability estimation. In [4], a study on a networked Windows NT system shows that many reboots, i.e., about 50%, do not show any specific reason, thus enforcing the need for better logging techniques. A study on supercomputers [2] shows that logs may lack useful information for enabling effective failure detection and diagnosis. Recent studies (e.g., [22], [23], [24]) highlight logs inadequacy at providing evidence of software faults, which can be activated on the field by complex environmental conditions [25]. For example, bad pointer manipulations may originate a crash before any information is logged in C/C++ programs.

Recent contributions address inefficiency issues of log files. A proposal for a new generation of log files is provided in [26], where recommendations are introduced to improve log expressiveness by enriching their format. A metric is also proposed to measure information entropy of log files, in order to compare different solutions. Another proposal is the IBM Common Event Infrastructure [27], introduced mainly to save the time needed for root cause analysis. It offers a consistent, unified set of APIs and infrastructure for the creation, transmission, persistence and distribution of log events, according to a well-defined format.

All these studies represent an important step forward for log-based dependability evaluation of computer systems. However, they mainly address *format heterogeneity* issues, i.e., they focus on *what* has to be logged. Logs incompleteness and ambiguity cannot be solved acting solely on format. Developers may miss to log significant failure events, and they may produce events with ambiguous descriptions. At the same time, tools for automated log analysis may coalesce uncorrelated events. In this paper logging rules are introduced to define the points in the source code *where*, other than *what*, events should be logged. We aim to achieve homogenous, i.e., both in format and semantics, logs, even if produced by software components coming from different developers. This allows improving failures detection and related coalescence, hence increasing the overall quality of FFDA results.

III. System Model

We use a high-level model to describe the main components of a system and the interactions among them. This makes it possible to design the proposed logging framework without the need for focusing on a specific real-world technology. More in details, we use the model (i) to figure out where to place effective logging mechanisms within the source code of an application, and (ii) to design general-purpose algorithms and tools to automate log collection and analysis. Following this objective we classify system components in two categories, according to the following definitions:

- **entity**: *active* system component. It provides *services* that can be invoked by other entities. An entity executes local computations, it starts interactions involving other entities or resources of the system and it can be the object of an interaction started by another entity.
- **resource**: *passive* system component. At most it is the object of an interaction started by another entity of the system.

Proposed definitions provide very general concepts, which have to be specialized according to designer's needs. For example, entities may model processes or threads, i.e., active elaboration components, while resources may model files and/or databases. Furthermore, entities may represent logical components, e.g., the executable code belonging to a library or package of code, independently of the process/thread executing it.

As stated, entities **interact**, e.g., by means of function calls or method invocations, with other system components, i.e., entities or resources, to provide complex services. We do not consider a specific real-world interaction mechanism. Our focus is on the properties of an interaction, i.e., (i) it is always started by an entity (ii) its object can be another entity or a resource of the system (iii) it possibly originates further computation if the object of the interaction is an entity.

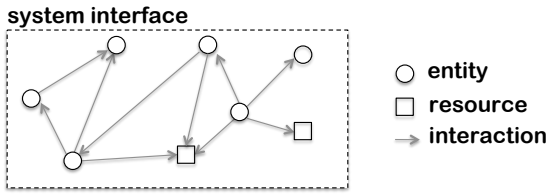


Figure 1: System overview.

We adopt a graphic formalism to represent the proposed concepts. More in details, entities and resources are represented as circles and squares, respectively, while an interaction as a direct edge from the caller entity to the called. Figure 1 is provided for example.

IV. Logging Rules

Taking into account the proposed system model, we investigate how to place log events within the source code of an entity to enable effective dependability measurements. To this aim, we identify two types of events, i.e., *interaction* and *life-cycle* events, respectively. The former provides failure-related information, the latter allows figuring out the operational state of an entity. Jointly with the event definition, we present a **logging rule**, which formalizes its use during the coding phase. Each rule defines *what* to log, i.e., the event that has to be logged by the entity, and *where* to log, i.e., the point in the source code where entities have to log the event.

A. Interaction Events

Interaction events aim to make it possible (i) to detect entity failures, (ii) to discriminate if they are related to a *local* computation or to an interaction towards a failed entity or resource. We describe the principle underlying interaction events in the following. Section V-A provides an in-depth discussion about the hypothesis and failure mode assumptions underlying their analysis during system operations. We start focusing on services provided by system entities, addressed by the following rules:

- **R1**, *Service Start - SST*: the rule forces the SST event to be logged before the first instruction of each service

provided by an entity. It provides the evidence that the entity, when invoked, starts serving the requested interaction.

- **R2**, *Service End - SEN*: the rule forces the SEN event to be logged after the last instruction of each service. It provides the evidence that the entity, when invoked, completely serves the requested interaction.

Figure 2 clarifies the aim of R1 and R2 by means of an example in the field of object oriented programming. Let A be an object providing the service named `serviceA()` and `log()` be a facility to log the described events. When the service is invoked, A logs the SST event. If a fault is triggered during the service execution (e.g., due to a bad pointer value used by `I1` in Figure 2) SEN will miss in the log of the entity.

```
void A::serviceA(int* ptr){
      log(SST);           //R1
      cout << *ptr;       //I1
      b.service();        //I2
      log(SEN);           //R2
}
```

Figure 2: Logging rules (R1,R2)

SST and SEN events alone are not enough to figure out if an entity failure is due to a *local* error or to an interaction with a failed entity or resource. As depicted in Figure 2, if A does not log the SEN event, we are not be able to figure out if the outage is due to `I1`, i.e., the local computation, or to `I2`, i.e., the interaction involving another entity. For this reason we introduce interactions-related events:

- **R3**, *Entity (Resource) Interaction Start - EIS (RIS)*: the rule forces the EIS (RIS) event to be logged before the invocation of each service. It provides the evidence that the interaction involving the entity (resource) is actually started by the calling entity.
- **R4**, *Entity (Resource) Interaction End - EIE (RIE)*: the rule forces the EIE (RIE) event to be logged after the invocation of each service. It provides the evidence that the interaction involving the entity (resource) ends.

No other instructions are allowed between the events EIS (RIS)-EIE (RIE). By using R3 and R4 the example code shown in Figure 2 turns in Figure 3. In this case, if the interaction `b.serviceB()` fails (e.g., by never ending, as in case of a hang in the called entity), we are able to find it out, since the event EIE is missing.

An entity usually provides more than one service or start more than one interaction. In this case multiple SSTs (EISs) are produced by the entity and it is not possible to figure out the service (interaction) they are actually related to. To overcome this limitation, the *start* and *end* events, related to each service or interaction within the same entity, are logged jointly with a unique key.

```
void A::serviceA(int *ptr){
      log(SST);           //R1
      cout << *ptr;       //I1
      log(EIS);           //R3
      b.service();        //I2
      log(EIE);           //R4
      log(SEN);           //R2
}
```

Figure 3: Logging rules (R3,R4)

We recognize that the extended use of logging rules may compromise code readability. However, by taking advantage of their simplicity, it is possible to design ad-hoc supports to automatically insert them just before the compilation stage. Such an approach makes rules-writing transparent to developers and does not require the direct modification of the source code. This issue is not currently a priority, however we aim to address it in the future.

B. Life-cycle Events

Interaction events do not allow understanding if an entity is currently down, or if it restarted after a failure. To overcome this limitation, we introduce life-cycle events, which aim to make it possible to figure out the operational state of an entity by providing evidence that it actually started its execution or it has terminated properly. Two rules fit this aim:

- **R5**, *Start up - SUP*: the rule forces the SUP event to be logged as the first instruction executed by an entity, at its startup.
- **R6**, *Shut down - SDW*: the rule forces the SDW event to be logged as the last instruction executed by an entity, when it is properly terminated.

These events are useful to evaluate dependability figures, even on-line, such as uptime and downtime for each system entity. Furthermore, SUP and SDW sequences allow identifying clean and dirty shutdowns, e.g., two consecutive SUP events are an evidence of a dirty shutdown. This type of events has been already used or proposed by past studies to identify clean and dirty reboots of operating systems [28], [4]. Our idea is to exploit this concept at a finer grain, according to the system model and applied to all entities.

V. Events Processing

The joint use of both the proposed model and logging rules makes a system to be perceived, from the analyst point of view, as a set of entities, each producing an *event flow*. These flows can be used to extract, during system operations, useful insights about the current execution state of the entities as well as to detect failure occurrences. To this aim we design algorithms to identify and to correlate alerts during system operations and to perform effective dependability measurements.

A. Alerts Identification

Analyzing log files to isolate entries, which provide evidence that a failure occurred in the system, is a time-consuming task of FFDA. We refer to these entries as to *alerts*. As discussed in Section II, alerts identification is usually preformed by looking at the severity level or the type of the entry (if available in the logging mechanism) and by analyzing the free text contained in the entry (e.g., to understand if it contains specific error-related keywords, such as `error`, `halt`, `unable`, etc.). As discussed, logs inaccuracy may compromise this kind of analysis. Entries with the semantics, but containing a different text, can be erroneously classified as different and vice versa. In addition, some failures, e.g., hangs, are unlikely to produce entries useful for their identification.

Interaction events are designed to make it possible to automate alerts identification, and to discriminate between alerts due to *local* or *external* causes, as explained in the following. By construction, logging code interleaves the source code of the entity. Hence we assume as possible errors the ones that result in modification, suspension or termination of the entity control flow, thus leading to delayed or missing log events. This assumption indirectly provides possible failure modes covered by the proposed logging mechanism. Let clarify the concept by examples. The assumption covers crashes or hangs (both active and passive) failures of the system entities. When an entity crashes (or hangs) while serving a request, SEN is missing in the related flow. At the same time the calling entity may not be able to correctly log its EIE. The assumption, on the other hand, does not fit value failures, but, at the state of art, it is known that they seem to be not detectable solely via logs.

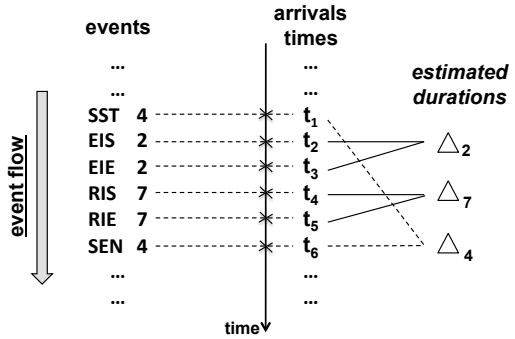


Figure 4: Alert identification.

Since our focus is on anomalies that ultimately result in delayed or missing events, we design external detectors based on timeouts. As a reminder, log events are provided in *start-end* pairs. A SST has to be followed by the related SEN, and an EIS has to be followed by the related EIE (in a similar way for a resource). We measure the time between two related events (i.e., the *start* and *end* events belonging to the same service or interaction) during each fault-free operation of the target system, in order to keep constantly updated the expected duration (e.g., Δ_2, Δ_7, and Δ_4, in Figure 4) of each pair of events. A proper timeout is then tuned for the alerts identification process. Figure 4 clarifies the concept. Proposed detector generates an alert whenever an *end* event is missing. We define three types of alerts:

- *entity interaction alert - EIA*: it is generated when EIS is not followed by the related EIE within the currently estimated timeout;
- *resource interaction alert - RIA*: it is generated when RIS is not followed by the related RIE within the currently estimated timeout;
- *computation alert - CoA*: it is generated when SST is not followed by the related SEN within the expected timeout and neither an entity interaction alert, i.e., EIA, nor a resource interaction alert, i.e., RIA, has been generated.

As discussed in Section IV a computation alert represents a problem that is *local* with respect to the entity that generated it. On the other hand, an interaction alert reports a misbehavior due to an *external* cause.

B. Alerts Coalescence

Error propagation phenomena, due to interactions among system components, usually result in multiple alerts. Coalescence makes it possible to reduce the amount of actually useful information to perform the analysis, by putting together distinct alerts into a *clustered* one. As discussed in Section I, traditional log-analysis commonly faces alert redundancy by means of time-based approaches, but without any awareness of the actual correlation among log messages.

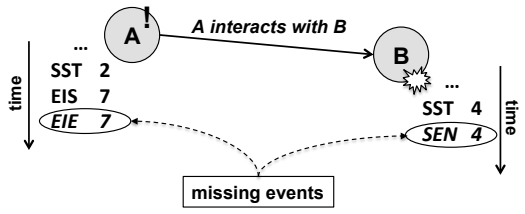


Figure 5: Example.

The use of precise logging rules significantly reduces analysis efforts and increases the effectiveness of the coalescence phase. As a matter of fact, interaction events make it possible to discriminate different types of alerts, each of them with its own specific meaning: CoA and RIA allow to identify a failure source, EIA to trace error propagation phenomena.

Figure 5 clarifies the concept. Let A and B be two system entities. A starts an interaction with B. If a latent fault is triggered during the service execution, B does not log SEN. At the same time A, that is still working, is not able to

properly log the EIE because of the outage of B. An *entity interaction alert* and a *computation alert* are raised for A and B, respectively.

We generalize the example according to the proposed model. A system is composed by several entities, which interact among them in order to provide complex services. An interaction chain is ultimately composed by simpler one-to-one interactions between (i) two entities or (ii) an entity and a resource. When a fault is triggered within an entity, we experience one CoA or RIA, related to the component ultimately responsible of the problem, possibly followed by multiple EIAs coming from the entities involved in the interaction chain.

The described principle underlies the following coalescence strategy. When a CoA or RIA is observed in the system, a new tuple, i.e., a clustered alert, is created. Each experienced EIA is stored until the previous and successive tuples have been created and it is subsequently coalesced with the one that is closer in time. Each tuple allows achieving useful insights about the failure occurred in the system. As a matter of fact it provides information about the *source* of a failure and all the entities involved due to propagation phenomena. Tuples produced with this approach can be used to evaluate the failure behavior of each system entity, e.g., in terms of the time-to-failure statistical distribution.

C. Identification of Execution States

We design a state-machine (Figure 6) to figure out the execution state of an entity during system operations. To this aim we use both life-cycle events and described alerts. We identify three possible states detailed in the following:

UP - the entity is up and properly running;
BAD - the entity may be in a corrupted state;
DOWN - the entity is stopped.

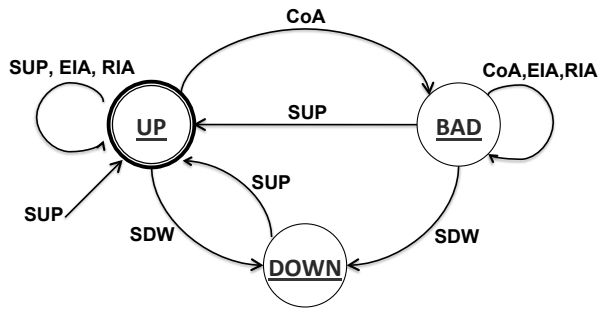


Figure 6: Entity execution states

When an entity starts, a SUP event is received, and the UP state is assumed. If an interaction alert, i.e., EIA or RIA, is received, the entity is considered to be still UP, but it is likely involved in a failure caused by another system entity or resource. In some cases, e.g., due to scheduled maintenance or to the persistence of interaction problems, an entity may be restarted. In this case, a SDW event will be observed, which makes the entity transit into the DOWN state. If a CoA is received, the entity transits into the BAD state. As discussed, a CoA is the result of a *local* problem within the entity. We thus assume that the internal state of the entity may be corrupted. An entity in a BAD state may be (i) still able to perform normal operations (e.g., the CoA is the result of a transitory problem) or (ii) actually failed (e.g., crashed). If an alert, i.e., CoA, EIA or RIA, is received, the entity is considered to be still BAD. When the entity is resumed, we may observe either a clean or dirty restart. In the former case the couple SDW, SUP is observed, which means that the entity was still active and able to handle a shutdown command. In the latter case, only the SUP event is observed (transition BAD to UP), i.e., the entity is restarted abruptly.

The evolving of the state machine over time makes it possible to achieve useful insights about the dependability behavior of the entity. As for example, (i) the time the entity persists in the UP state contribute to the estimation of the uptime of the entity, (ii) the time between a SUP and a CoA event is an estimate of the Time To Failure.

VI. Logging Framework

We design a comprehensive framework to automate online collection and analysis of described events. Figure 7 depicts the proposed infrastructure and highlights its main components, i.e., (i) the operational system producing log events according to the rules (ii) a transport layer named LogBus (iii) a set of *pluggable* components, which perform several types of analyses.

Events are sent over the **LogBus**, which is the adopted transport layer among the machines of the system under analysis and the processing components. LogBus keeps logically separated event flows coming from distinct entities of the system by means of labeling each flow with a unique key. We implement a C++ object-based LogBus prototype using standard TCP sockets to transmit events. An API exposing simple methods to access the infrastructure hides internal event management mechanisms.

LogBus forwards events towards an extensible set of ***pluggable* components**. Each component connect the LogBus during system operations and subscribe only the class of events (i.e., interaction or life-cycle) it is interested in, by using a filtering mechanism provided by the LogBus API. This makes it possible to design specific tools, just doing a part of the whole FFDA analysis but doing it in an effective way. Furthermore, the adoption of a model-based approach makes it possible to reuse a designed tool. In fact the analysis is performed on events with well-defined semantics, despite of the system producing them. Output coming from different components is combined to produce value-added information.

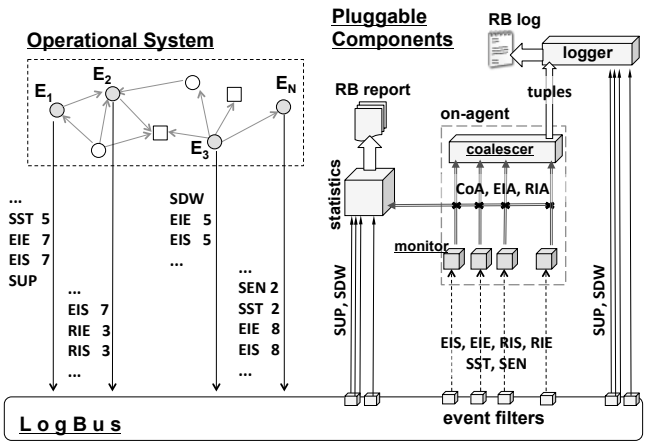


Figure 7: Logging and Analysis Infrastructure.

We describe currently available tools in the following. It should be noted that they only provide a possible set of analysis tools. In fact, LogBus is an *open* platform, which makes it possible to connect novel components provided by third-party developers.

The **on-agent** component includes a set of *monitors* and a *coalescer*. Each **monitor** subscribes interactional events for a unique entity and implements the alert identification strategy described in Section V-A. Generated alerts are supplied to the **coalescer** during operations. This component, in turn, implements the coalescence approach described in Section V-B. Produced tuples are not immediately stored on a log file, but they are forwarded to the *logger*.

The **logger** component is the one in charge of maintaining a log file, i.e., the Rule-Based (RB) log, specifically conceived to perform dependability analyses. It jointly stores both tuples supplied by on-agent and life-cycle events coming from the LogBus. Figure 8 shows the content of the RB log. This enables detailed analyses without preprocessing effort. As a matter of fact, tuples provide clustered failure data, thus avoiding the need for coalescence procedures. This information is combined with life-cycle events, which avoid the need for manual efforts to figure out reboot occurrences. Section IX shows how this log has been used to characterize the dependability behavior of the reference case study.

The **statistics** component keeps the state machine described in Section V-C for each entity of the system. More in details, it manages a set of variables, which are updated upon state transitions during system operations. These are used, for example, to estimate, for each entity (i) uptime, downtime (i.e., the time between a SDW and a SUP), and failtime (i.e., the time between a CoA and a SUP when no SDW event has been experienced between them) (ii) SUP, SDW and alert counts and, (iii) availability. This information is used to provide an on-line snapshot for the overall system, i.e., the Rule-Based (RB) report.

```
Timestamp                Type      Source    Affected
-------------------------------------------------------------------
...
2009/05/03  16:42:55     SUP       [E1]
2009/05/03  16:50:40     SUP       [E4]
...
2009/05/06  09:45:05     CoA       [E3]      [E4,E1,E2]
...
2009/05/10  08:15:07     CoA       [E1]      [E5,E3]
...
2009/05/10  10:50:40     SDW       [E4]
2009/05/10  10:50:43     SDW       [E5]
2009/05/10  10:51:20     SDW       [E3]
...
```

Figure 8: Content of RB log.

VII. Case Study

We evaluate the effectiveness of both *traditional* and *rule-based* logging approaches at characterizing the dependability of an operational system. To this aim we deliberately emulate a known failure behavior into a real-world software system producing both its own logs and instrumented to use the described framework. Field data collected with both the mechanisms during a 32 days long-running experiment are analyzed.

A. Air Traffic Control (ATC) Application

The reference application consists of a real-world software system in the field of ATC. In particular we consider a **Flight Data Plan (FPL) Processor**. FPLs provide information such as a flight expected route, its current trajectory, vehicle-related information, and meteorological data.

The FPL Processor is developed on the top of an open-source middleware platform named CARDAMOM[1]. This platform provides services intended to ease the development of critical software systems. For example, these include Load Balancer (LB), Replication (R), and Trace Logging (TL) services, used by the application in hand. The FPL Processor integrates the OMG-compliant[2] Data Distribution System (DDS) [29]. DDS allows applicative components to share FPLs in our case study. This is done by means of the `read` and `write` facilities provided by the DDS API, which allow to *retrieve* and to *publish* a FPL instance, respectively.

Figure 9 depicts the FPL Processor. It is a CORBA-based distributed object system. It is composed by a replicated **Facade** object and a set of processing **Servers** managed by the LB. Facade accepts FPL processing requests (i.e., insert, delete, update) supplied by an external Tester and guarantees data consistency by means of mutual exclusion among requests accessing the same FPL instance. Facade subsequently redirects each allowed request to 1 out of the 3 processing Server, according to the *round robin* service policy. The selected server (i) retrieves the specified FPL

[1] http://forge.objectweb.org/projects/cardamom

[2] OMG specification for the Data Distribution Service, `http://www.omg.org`

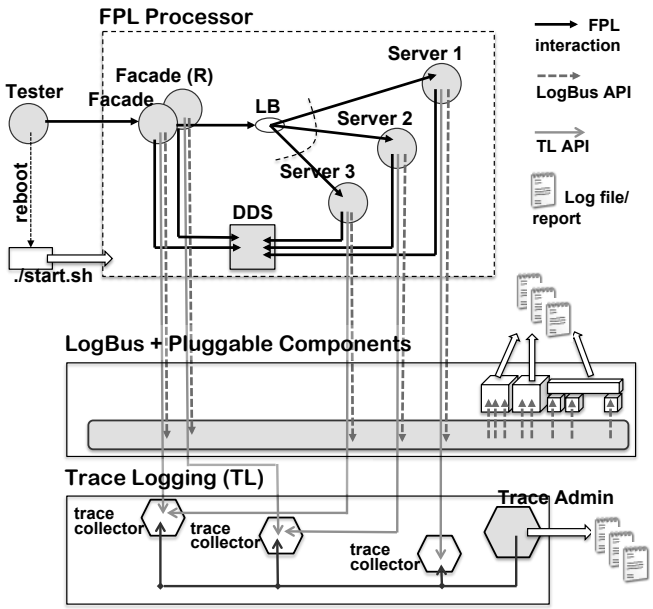


Figure 9: Case study.

instance from the DDS middleware (ii) executes request-related computation, and (iii) returns the updated FPL instance to the Facade object. Facade publishes the updated FPL instance and finalizes the request by acknowledging the Tester.

Tester object invokes Facade services with a frequency of 1 request per second. Under this workload condition a request takes about 10 ms to be completed, as shown in Section X. We instrument the Tester object in order to detect request failures. A timeout-based approach is adopted to this aim. We assume 15 ms to be an upper bound for a request to be completed. Consequently, if a request is not acknowledged within a 50 ms timeout, it is considered as failed. Due to the replicated nature of both Facade and Server objects, one request failure does not imply that the mission of the FPL Processor is definitively compromised. The system may be in a degraded state, but still able to satisfy further requests. For this reason we assume the mission of the system to be definitively compromised only if 3 consequent requests fail. In this case the Tester object triggers the FPL Processor reboot via the `start.sh` bash script. We experience that the application reboot time varies between 300 s and 400 s.

Machines composing the application testbed (Intel Pentium 4 3.2 GHz, 4 GB RAM, 1,000 Mb/s Network Interface equipped) run a RedHat Linux Enterprise 4. A dedicated Ethernet LAN interconnects these machines. About 4,000 FPLs instances, each of them of 77,812 bytes, are shared with the DDS.

Table I: Time To Failure (TTF) distributions

Object	Distribution
Facade	$F(t) = 1 - e^{-0.000001t^{0.92}}$
Server	$S(t) = 1 - e^{-0.000005t^{0.92}}$

B. Logging Subsystems

FPL Processor uses the **TL service** to collect log messages produced by applicative components (Figure 9). TL provides a hierarchical mechanism to collect data. A `trace collector` daemon is responsible to store messages coming form processes deployed on the same node. A `trace admin` process collects per-node log entries and store these data in a file. Each log entry contains information such as a timestamp, 1 out of 5 severity levels (i.e., `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`), source-related data (e.g., process/thread id), and a free text message. We assume data collected via the TL service to be an example of traditional logs.

We instrument the application code to produce rule-based events and to integrate the **LogBus** infrastructure (Figure 9). To this aim we assume a high-level system model. In particular each FPL object (i.e., Facade and Serves) is an *entity* and the DDS, as a whole, is modeled as a *resource*. Interactions consist of both CORBA-based remote methods invocations and DDS read/write facilities.

C. Experiments

We leave the FPL Processor running for about 32 days, from Aug-07-2009 to Sep-07-2009. During this period we do not wait for *natural* occurring errors but we deliberately emulate a known failure behavior in the system. Our aim is to evaluate if/how traditional and rule-based logs allow to reconstruct this known dependability behavior. More in details we perform availability and failure analyses by using both logs.

We instrument FPL Processor objects (i.e., Facade and Servers) to trigger failures according to the Time To Failure (TTF) distributions shown in Table I (time measured in centiseconds). We find the Weibull distribution a proper choice since it has shown to be one of the most used distribution in failure analysis [30]. However, any other reliability function clearly fits the aim of the experiment. Different scale parameters assure that Facade and Servers fail with different rates. When an object failure has to be triggered according to the current TTF estimate we inject either a crash or a hang with the same probability. A faulty piece of code, i.e., a bad pointer manipulation and an infinite wait on a locked semaphore, is executed to emulate, crashes and hangs failures, respectively. Jointly with the execution of the faulty code we record the type of the emulated failure, i.e., crash or hang, as well as the component executing it. An object failure always results in a system failure in our case study, as the current FPL request does not correctly succeed.

Table II: Failures breakup by object

Object	Failures
Facade	260
Server 1	732
Server 2	772
Server 3	738
Total	*2,502*

Furthermore an object is not immediately resumed after a crash failure. This is the reason why subsequent crashes lead progressively to the reboot signal. In this case the FPL Processor *as a whole* is restarted. We experience that during the 32 days period the FPL Processor is rebooted 400 times and 2,502 object failures are triggered. Table II reports the failures breakup by object. We collect logs and/or reports produced both by the TL service and pluggable components to perform the analysis.

VIII. Analysis of Trace Logging (TL) Log

TL log collected during the long-running experiment is about 2.2 MB and contains 24,126 lines.

A. Availability Analysis

We perform FPL Processor availability analysis by estimating system *uptimes* and *downtimes*, as described in previous works in the area of FFDA (e.g., [4], [28]). To this aim, for each reboot occurred during the experiment, we identify the timestamp of (i) the event notifying the end of the reboot, and (ii) the event immediately preceding the reboot. A *downtime estimate* is the difference between the timestamps of the two events. An *uptime estimate* is the time interval between two successive downtimes. Uptime and downtime estimates are used to evaluate system availability by means of Equation 1.

$$A = \frac{\sum_i uptime_i}{\sum_i uptime_i + \sum_i downtime_i} \cdot 100 \qquad (1)$$

The described approach requires the identification of application reboots from logs. To this aim we directly inspect TL log in order to identify sequences of log events triggered by application reboots. Figure 10 depicts a simplified version of such a reboot sequence. The "`Startup complete`" event identifies the end of the reboot. We assume the event preceding the "`CDMW Finalize`" event to be the one preceding the reboot.

We develop an ad-hoc algorithm to automatically extract (i) reboot events, and (ii) uptime and downtime estimates from TL log. Table III provides statistics characterizing the estimates. Downtime estimates are close to the expected reboot time. We estimate FPL Processor availability according

```
2009/26/08 14:41:05 INFO CDMW Finalize
2009/26/08 14:41:20 INFO Parsing XML Finalize FDPSystem
2009/26/08 14:41:47 INFO FDP Server
2009/26/08 14:43:54 INFO Finalize APP1/Server process
...
[omissis]
...
2009/26/08 14:43:13 INFO CDMW Init
2009/26/08 14:43:23 INFO Parsing XML Init file FDPSystem
2009/26/08 14:43:27 INFO FDP Server
2009/26/08 14:43:30 INFO Initialize APP1/Server process
   with XML File
2009/26/08 14:43:40 INFO CDMW init ongoing for APP1/Server
2009/26/08 14:44:10 INFO Acknowledge creation of process
   APPL1/Server
...
[omissis]
...
2009/26/08 14:46:44 INFO Acknowledge creation of process
   APPL4/Facade
2009/26/08 14:46:48 INFO Startup complete
```

Figure 10: FPL Processor reboot sequence (TL log).

Table III: Downtime and uptime estimates: statistics (TL log)

	Downtime	Uptime
Value	350.2 ($\pm$23.6) s	6,740.6 ($\pm$4,399.6) s
Minimum	300.1 s	843.4 s
Maximum	400.2 s	32,518.6 s

to Equation 1. Equation 2 provides A_T, i.e., the availability estimate resulting from TL log.

$$A_T = \frac{2,689,487.9\ s}{2,689,487.9\ s + 143,736.5\ s} \cdot 100 \approx 94.9\% \qquad (2)$$

A_T is about 94.9%. The overall downtime is 143,736.5 s. It should be noted that this is a realistic finding. As a matter of fact a reboot of the FPL Processor takes about 350.2 s (Table III). During the long-running experiment 400 reboots occur. An overall downtime estimate is thus $400 \cdot 350.2s = 140,080s$, which is close to the actual one.

B. Failure Analysis

As discussed in Section II, we investigate TL log to characterize failure related data. The analysis reveals that anomalous conditions and error propagations phenomena usually result in the higher severity levels, i.e., `WARN`, `ERROR`, and `FATAL`, provided by the TL logging mechanism. We develop an algorithm to automatically extract failure related entries from TL log by means of the severity information. This procedure filters 20,637 out of the collected 24,126 events. In other words 3,489 events, i.e., about 14% of the amount of the collected information, are used to perform failure analysis.

It should be noted that a component failure might lead to multiple log entries due to propagation phenomena within the system. We filter out redundant entries by applying the *tuple heuristic* [8]. Its aim is to put together distinct entries in

a clustered one, i.e., the tuple, with respect to a coalescence timing window. The objective is to build one tuple for each actually occurred failure. We implement LogFilter as described in [2] to analyze the collected TL log.

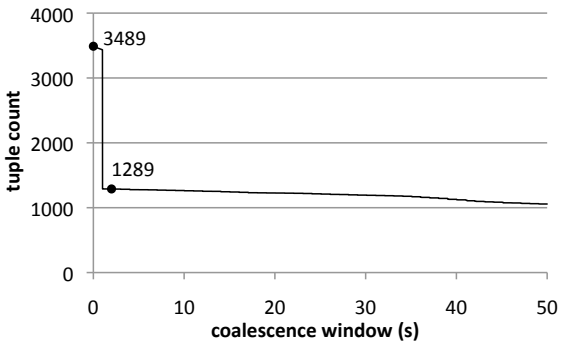


Figure 11: Time effect on tuple count

We perform a sensitivity analysis to choose a suitable coalescence window for the proposed case study. Figure 11 shows the analysis results. Tuple count suddenly decreases from the 3,489 initial value, since log entries related to the same failure are very close in time. As suggested in [8] the vertex of the "L" shaped curve represents the internal clustering time of the system and the coalescence window should be greater that this value. We thus assume **2 s**, i.e. 1,289 tuples, to be a suitable coalescence window for our case study. It should be noted that only 1,289 out of the 2,502 actually emulated failures result from the analysis. An in depth analysis reveals that only crashes are logged while hangs do not leave any trace in TL log.

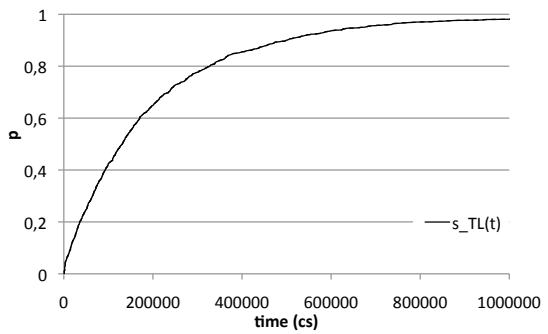


Figure 12: FPL Processor estimated TTF (TL log).

We estimate the TTF distribution for the FPL Processor, named `s_TL(t)`, by using the timestamp information of both the tuples and the events notifying the end of a reboot. Figure 12 depicts the analysis finding. Resulting Mean Time To Failure (MTTF) is approximately 34 minutes. This is greater than the expected since only 1,289 out of the 2,502 actual emulated failures result from the analysis.

Table IV: Downtime and uptime estimates: statistics (RB log)

	Downtime	**Uptime**
Value	350.2 ($\pm$23.6) s	6,740.6 ($\pm$4,399.6) s
Minimum	300.1 s	843.4 s
Maximum	400.2 s	32,518.6 s

Regardless of the quality of the achieved finding, `s_TL(t)` provides a characterization of the failure behavior of the system under study. Anyway, it is not clear how this finding could be actually exploited by developers, e.g., to drive specific dependability improvements where needed. In the proposed case study, among multiple notifications reported by the TL log, we are not able to figure out the object that first signaled a problem, thus preventing and in-depth characterization of the system in hand.

IX. Analysis of Rule-Based (RB) Log

During the 32 days long-running experiment about 30 millions of rule-based events are sent over the LogBus. Resulting RB log, provided by the *logger* pluggable component, is about 128 KB and contains 4,500 lines. It should be noted that the size of RB log is about 5.7% when compared to TL log. The amount of information actually needed for the analysis phase has been significantly reduced with the proposed strategy.

A. Availability analysis

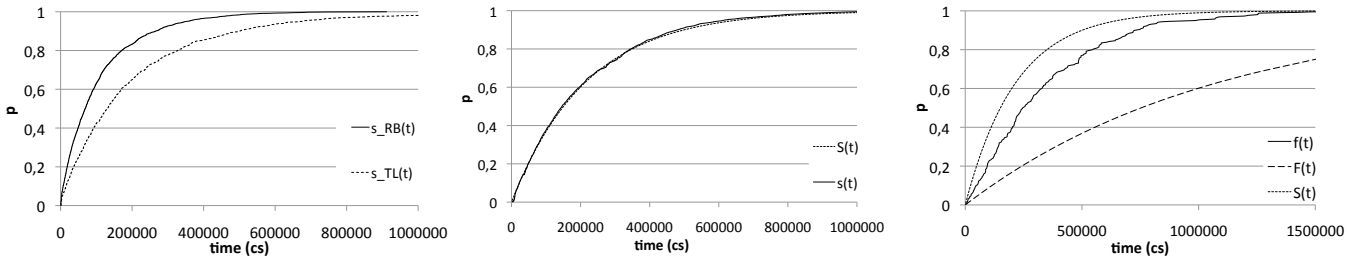
We perform FPL Processor availability analysis by tailoring the approach described in Section VIII-A to RB log. In this case, application reboots are identified by SDWs-SUPs sequences. Figure 13 is provided as an example.

```
2009/26/08  14:41:05      SDW       [Facade]
2009/26/08  14:42:55      SUP       [Server1]
2009/26/08  14:43:40      SUP       [Server2]
2009/26/08  14:45:05      SUP       [Server3]
2009/26/08  14:46:40      SUP       [Facade]
```

Figure 13: FPL Processor reboot sequence (RB log).

Facade SUP identifies the end of a reboot. We assume the event preceding the first SDW of a reboot sequence to be the one preceding the reboot itself. Table IV provides statistics characterizing uptime and downtime estimates. We estimate FPL Processor availability according to Equation 1. Equation 3 provides A_{RB}, i.e., the availability estimate resulting from RB log.

$$A_{RB} = \frac{2,689,488.1\ s}{2,689,488.1\ s + 143,736.3\ s} \cdot 100 \approx 94.9\% = A_T \quad (3)$$



(a) FPL Processor estimated TTF. (b) Server 2 estimated TTF. (c) Facade estimated TTF.

Figure 14: Estimated TTF distributions (RB log).

Table V: Kolmogorov-Smirnov test

Process	Samples	D	L
Server 1	732	0.0410	0.80<L<0.90
Server 2	772	0.0325	L<0.80
Server 3	738	0.0253	L<0.80

A_{RB} is about A_T. The proposed framework allows to estimate system availability as well as a traditional logging approach, however the introduction of SUP and SDW events significantly reduces analysis efforts. In addition, as shown in Section IX-C we are allowed to perform a detailed availability analysis for each system entity.

B. Failure analysis

We exploit the RB log to gain insights of the FPL Processor dependability behavior. The tuple heuristic is not needed anymore. By construction, RB log already contains a clustered entry for each occurred failure, and the presence of life-cycle events allows to easily extract TTF estimates.

RB log contains 2,502 tuples. It should be noted that this is the amount of the actually emulated failures as shown in Table II. We perform a TTF analysis for the FPL Processor *as a whole*, i.e., by jointly considering tuples from all system entities. Figure 14a shows both `s_TL(t)` and `s_RB(t)`, i.e., the application TTF estimated by analyzing RB log. Resulting MTTF is approximately 17 minutes, thus shorter if compared to `s_TL(t)`. This finding highlights deficiency of TL log at providing evidence of all the occurred failures in the reference case study.

Information provided by RB log makes it possible to achieve further insights about the dependability behavior of the proposed case study. As a matter of fact, we use the *source* field supplied by the logger component for each tuple, to figure out TTF distributions for each entity of the system. Figure 14b depicts the estimated TTF, named `s(t)`, when compared to `S(t)`, for Server 2 (a similar finding comes out for the two remaining Servers). The experienced distribution is close to the one emulated during the long running experiment. We perform the Kolmogorov-Smirnov test to evaluate if `s(t)` is a *statistically* good `S(t)` estimate. Let (i) D be the maximum distance between the analytical and the estimated distributions and (ii) L be the resulting significance level of the test. Table V reports results obtained for the all the Servers. The low value of L assures that the collected samples are consistent with the actual failure distributions.

We perform a similar analysis for the Facade object. Figure 14c shows `f(t)`, i.e., the TTF estimate. It is immediate to figure out that `f(t)` is different form the emulated `F(t)`, but lower than `S(t)`. This is a realistic finding, which depends on the *recovery* strategy adopted in the case study. In our long-running experiment Servers exhibit a failure rate higher than the Facade (Table I). This makes it very likely that all Servers have crashed while the Facade is still properly working. In this case the Tester object triggers the FPL Processor reboot, thus preventing the Facade object from exhibiting its actual behavior.

By concluding, the proposed strategy enables an in-depth characterization of the FPL Processor dependability behavior. The comparison between the estimated TTF distributions, i.e., `f(t)` and `s(t)`, makes it possible to identify the actual most failure-prone entity within the system. This information can be used, for example, to reduce the Mean Time To Repair [31] or to apply proper recovery actions only when needed [32].

C. On-line Report

The *statistics* component provides a snapshot, i.e., the RB report, of the current states of the system entities during the operational phase. This information is not available with the TL logging subsystem and it is the result of the proposed strategy. Table VI shows the RB report at the end of the long running experiment. In the following, we discuss the resulting findings in order to evaluate if they are realistic with respect to the emulated failure behavior.

Facade availability is 95%, thus close to the one estimated for the system as a whole (Equation 3). As a matter of fact when the Facade is unavailable, the FPL Processor is rebooted, since FPL requests cannot be satisfied anymore.

Table VI: RB report at the end of the long-running experiment

	Uptime	Downtime	Failtime	SUP	SDW	CoA	EIA	**Avalability**
Facade	2,700,740 s	129,111 s	4,863 s	400	385	260	2,242	95.0%
Server 1	1,479,900 s	770 s	1,354,250 s	400	7	732	0	52.2%
Server 2	1,591,580 s	1,470 s	1,240,200 s	400	7	772	0	56.1%
Server 3	1,522,890 s	1,860 s	1,308,500 s	400	6	738	0	53.8%

Consequently, the Facade object is not allowed to remain in a failed state for a long time (i.e., a low failtime). On the other hand, Servers availability is around 54%. Due to the adoption of the LB policy, even if a Server crashes, the two remaining ones make it possible to execute subsequent FPL request. It may take a long time before the application is rebooted and a crashed Server is resumed.

Facade SDWs are mainly *clear*, i.e. the SUP count is close the SDW one. This is a realistic finding, since the Facade object has a failure rate lower than the Servers. As discussed, it is very likely that it is still able to correctly handle FPL requests when the reboot signal is triggered. Not the same for the Server objects. In this case most of the reboots are dirty.

Adopted logging rules, make it possible to understand if a problem with an entity is caused by a propagating error and thus to prevent erroneous findings. Table VI reports CoA and EIA counts, which allow to break the total amount of outages for each system entity by local, i.e. CoA, and interaction, i.e., EIA. Servers exhibit only CoAs, as they do not start interactions with any other entity within the system. It should be noted that the CoA count is equal to the actual emulated failure count for each Server (Table II). This finding demonstrates the effectiveness of the proposed alerts identification strategy with respect to the proposed case study. On the other hand, alerts experienced by the Facade object are mainly due to interaction causes.

X. Overhead Estimation

We evaluate how both TL and LogBus subsystems affect application performance. It should be noted that system *response time* and *resource usage* are widely recognized to be effective metrics for performance analysis in computer systems [33]. This is the reason why we choose the Round Trip Time (RTT) of FPL requests, measured at the Tester node, to be the reference metric for our case study. RTT makes it possible to achieve insights about the FLP Processor performability.

We analyze performance by taking into account two parameters, i.e., the specific logging subsystem and the FPL request invocation period. The logging subsystem assumes a value in $LS = \{T, RB, TL\}$ with T, RB, TL denoting, no logging subsystem, Rule-Base framework and Trace Logging, respectively. Invocation period varies in the range $I = \{300, 400, 600, 800, 1,000\}$ ms. These values take into account real world traces coming from the Air Traffic Control domain where CARDAMOM is commonly used as support middleware.

We design a full-factorial experimental campaign by executing a stress test for each combination of parameters in $LS \times I$. In particular, for each test we execute 3,000 FPL Processor requests, i.e, 1,000 requests per type (i.e., insert, delete, update) and we subsequently estimate the mean RTT after filtering outliers out. Figure 15 depicts experienced RTTs.

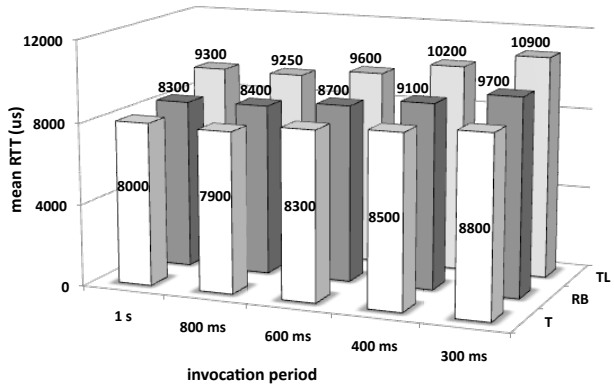


Figure 15: FPL requests RTT.

For each value of the invocation period in I we estimate the overhead of TL with respect to RB, i.e., $O_{TL,RB}$, when compared to T. Equation 6 shows how we estimate $O_{TL,RB}$. $O_{TL,T}$ and $O_{RB,T}$ denote the overhead of TL and RB with respect to T, respectively.

$$O_{TL,T} = \frac{RTT_{TL} - RTT_T}{RTT_T} \cdot 100 \quad (4)$$

$$O_{RB,T} = \frac{RTT_{RB} - RTT_T}{RTT_T} \cdot 100 \quad (5)$$

$$O_{TL,RB} = O_{TL,T} - O_{RB,T} = \frac{RTT_{TL} - RTT_{RB}}{RTT_T} \cdot 100 \quad (6)$$

Table VII summarizes overhead estimates. For each value of the invocation period in I, $O_{TL,RB}$ is a positive value. In other words, according to Equation 6, overhead introduced by RB is lower than TL when compared to T. Mean $O_{TL,RB}$

Table VII: Overhead estimates

	1s	800ms	600ms	400ms	300ms
$O_{TL,T}$	16.3%	17.1%	15.7%	20.0%	23.8%
$O_{RB,T}$	3.8%	6.3%	4.8%	7.1%	10.2%
$O_{TL,RB}$	12.5%	10.8%	10.9%	12.9%	13.6%

is approximately 12.1%. This value roughly estimate the expected overhead of TL when compared to RB. By concluding, the proposed logging framework does not affect, if not improve, application performance in the proposed case study.

XI. Conclusion

The paper described a framework to overcome the well-known limitations of traditional logging with respect to the dependability evaluation of complex systems. After presenting the principles underlying our proposal, we describe logging rules and algorithms enabling effective dependability evaluation of complex systems. We provide an in-depth comparison between the proposed framework and a real-world logging subsystem in the context of the Air Traffic Control domain. Results show that the proposed rule-base strategy:

- *Eliminates preprocessing effort to analyze data*. We show that the analysis of a traditional log, such as the one collected via the Trace Logging service, requires significant manual effort and ad-hoc procedures to identify and extract log events (e.g., reboot and failure occurrences) relevant for the analysis phase.
- *Preserves and improves findings of traditional log analysis*. Rule-based log makes it possible to estimate system availability as well as traditional logging. Furthermore it increases the quality of TTF analysis by means of an exhaustive coverage of timing failures in our case study.
- *Provides value-added information*. The proposed framework makes it possible to gain in-depth visibility of the dependability behavior of a system by means of a finer analysis grain. In particular, it enables valuable results (e.g., TTF distributions, on-line statistics) for each system entity, which cannot be achieved with traditional logging techniques.
- *Reduces the amount of information actually needed to perform the analysis*. Log size is reduced by more than 94.3% with respect to an example of traditional log. This improvement does not introduce information loss.
- *Does not affect application performance*. The overhead introduced by the proposed framework on the FPL Processor is 12.1% *lower* than the one introduced by the Trace Logging service.

Future work will encompass the definition of novel logging rules, aiming to support a wider range of FFDA analyses. LogBus and pluggable components will be consequently enhanced in order to provide additional features and capabilities. We also intend to explore the use of standard languages, e.g., XML, to represent the rule-based log.

Additionally, it is needed to deal with existing components that do not adopt the proposed logging rules. In this case, we claim the need for component-specific wrappers to produce events according to the described strategy. Following this direction, future work will be also devoted to research for model-driven techniques to automate the logging-code writing process. This is a need to significantly reduce, if not completely eliminate, manual instrumentation efforts.

Acknowledgment

This work has been partially supported by the "Consorzio Interuniversitario Nazionale per l'Informatica" (CINI) and by the Italian Ministry for Education, University, and Research (MIUR) within the frameworks of the "Centro di ricerca sui sistemi Open Source per la applicazioni ed i Servizi MIssion Critical" (COSMIC) Project (`www.cosmiclab.it`) and the "CRITICAL Software Technology for an Evolutionary Partnership" (CRITICAL-STEP) Project (`http://www.critical-step.eu`), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 230672, in the context of the Seventh Framework Programme (FP7).

References

[1] M. Cinque, D. Cotroneo, and A. Pecchia. A logging approach for effective dependability evaluation of complex systems. In *Proceedings of the 2nd International Conference on Dependability (DEPEND 2009)*, pages 105–110, Athens, Greece, June 18-23, 2009.

[2] A. J. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *International Conference on Dependable Systems and Networks (DSN 2007)*, pages 575–584. IEEE Computer Society, 2007.

[3] C. Lim, N. Singh, and S. Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *International Conference on Dependable Systems and Networks (DSN 2008)*, Anchorage, Alaska, June 2008.

[4] M. Kalyanakrishnam, Z. Kalbarczyk, and R. K. Iyer. Failure data analysis of a LAN of windows NT based computers. In *Proceedings of the Eighteenth Symposium on Reliable Distributed Systems (18th SRDS'99)*, pages 178–187, Lausanne, Switzerland, October 1999. IEEE Computer Society.

[5] C. Simache and M. Kaâniche. Availability assessment of sunOS/solaris unix systems based on syslogd and wtmpx log files: A case study. In *PRDC*, pages 49–56. IEEE Computer Society, 2005.

[6] M. F. Buckley and D. P. Siewiorek. VAX/VMS event monitoring and analysis. In *FTCS*, pages 414–423, 1995.

[7] Michael M. Tsao and Daniel. P. Siewiorek. Trend analysis on system error files. In *Thirteenth Annual International Symposium on Fault Tolerant Computing, IEEE Computer Society*, pages 116–119, 1983.

[8] J. P. Hansen and D. P. Siewiorek. Models for time coalescence in event logs. In *FTCS*, pages 221–227, 1992.

[9] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan.-March 2004.

[10] C. Lonvick. The bsd syslog protocol. *Request for Comments 3164, The Internet Society, Network Working Group, RFC3164*, August 2001.

[11] J.-C. Laplace and M. Brun. Critical software for nuclear reactors: 11 years of fieldexperience analysis. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, pages 364–368, Paderborn, Germany, November 1999. IEEE Computer Society.

[12] M. Cinque, D. Cotroneo, and S. Russo. Collecting and analyzing failure data of bluetooth personal area networks. In *Proceedings 2006 International Conference on Dependable Systems and Networks (DSN 2006)*, pages 313–322, Philadelphia, Pennsylvania, USA, June 2006. IEEE Computer Society.

[13] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. K. Sahoo. Bluegene/L failure analysis and prediction models. In *Proceedings 2006 International Conference on Dependable Systems and Networks (DSN 2006)*, pages 425–434, Philadelphia, Pennsylvania, USA, June 2006. IEEE Computer Society.

[14] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Proceedings 2004 International Conference on Dependable Systems and Networks (DSN 2004)*, pages 772–, Florence, Italy, June-July 2004. IEEE Computer Society.

[15] D. L. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[16] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *DSN*, pages 249–258. IEEE Computer Society, 2006.

[17] B. Murphy and B. Levidow. Windows 2000 Dependability. MSR-TR-2000-56, Microsoft Research, Microsoft Corporation, Redmond, WA, June 2000.

[18] D. Tang, M. Hecht, J. Miller, and J. Handal. Meadep: A dependability evaluation tool for engineers. *IEEE Transactions on Reliability*, pages vol. 47, no. 4 (December), pp. 443–450, 1998.

[19] A. Thakur and R. K. Iyer. Analyze-now - an environment for collection and analysis of failures in a networked of workstations. *IEEE Transactions on Reliability*, pages Vol. 45, no. 4,560–570, 1996.

[20] R. Vaarandi. Sec - a lightweight event correlation tool. In *IEEE IPOM'02 Proceedings*, 2002.

[21] J. P. Rouillard. Real-time log file analysis using the simple event correlator (sec). *USENIX Systems Administration (LISA XVIII) Conference Proceedings*, Nov. 2004.

[22] D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore. Investigation of failure causes in workload-driven reliability testing. In *Foundations of Software Engineering*, pages 78–85. ACM Press New York, 2007.

[23] D. Cotroneo, S. Orlando, and S. Russo. Failure classification and analysis of the java virtual machine. In *Proc. of 26th Intl. Conf. on Distributed Computing Systems*, 2006.

[24] L.M. Silva. Comparing error detection techniques for web applications: An experimental study. *7th IEEE Intl. Symp. on Network Computing and Applications*, pages 144–151, 2008.

[25] J. Gray. Why do computers stop and what can be done about it. In *Proc. of Symp. on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[26] F. Salfner, S. Tschirpke, and M. Malek. Comprehensive logfiles for autonomic systems. *Proc. of the IEEE Parallel and Distributed Processing Symposium, 2004*, April 2004.

[27] IBM. Common event infrastructure. http://www-01.ibm.com/software/ tivoli/features/cei.

[28] C. Simache and M. Kaâniche. Measurement-based availability analysis of unix systems in a distributed environment. In *Proc. of the 12th IEEE International Symposium on Software Reliability Engineering*, 2001.

[29] Gerardo Pardo-Castellote. OMG data-distribution service: Architectural overview. In *ICDCS Workshops*, pages 200–206. IEEE Computer Society, 2003.

[30] T.-T.Y. Lin and D.P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *IEEE Transactions on Reliability*, pages 419–432, 1990.

[31] G. Khanna, I. Laguna, F.A. Arshad, and S. Bagchi. Distributed diagnosis of failures in a three tier e-commerce system. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS07)*, pages 185–198, Oct 10-12, 2007.

[32] I. Rouvellou and G. W. Hart. Automatic alarm correlation for fault identification. In *INFOCOM '95: Proceedings of the Fourteenth Annual Joint Conference of the IEEE Computer and Communication Societies (Vol. 2)-Volume*, page 553, Washington, DC, USA, 1995. IEEE Computer Society.

[33] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons New York, 1991.

The Impact of Source and Channel Coding in the Communication Efficiency of Wireless Body Area Networks

Richard Mc Sweeney, Christian Spagnol, and Emanuel Popovici
Department of Microelectronics Engineering
University College Cork
Cork, Ireland
richardmcs@ue.ucc.ie, c.spagnol@ue.ucc.ie, e.popovici@ucc.ie

Luigi Giancardi
Department of Biophysical and Electronic Engineering
University of Genoa
Genoa, Italy
luigi.giancardi@ingegneria.studenti.unige.it

***Abstract*—This paper examines the system level energy performance of Wireless Sensor Motes for Electroencephalography (EEG) patient monitoring application by the use of concatenated source and channel coding. The addition of coding in a power constraint system has its advantages by reducing the energy per bit, but it also has its drawback in the cost of the power consumption in the encoding and decoding processes. In this work Huffman code is implemented as the source coding, and a shortened Reed-Solomon code is used for channel coding. The reliability and energy savings of the scheme is presented and the impact of the coding procedure on the communications performance is analyzed. The results show that it is possible to have Bit Error Rate (BER) and compression gains in the system, and that the computational time of purely software oriented implementations are not optimal. Also, the possibility for future extensions of this coding scheme, which would introduce better efficiency and accuracy, are shown. The error patterns that occur in the channel are investigated, and a design space for a possible Hybrid Automatic Repeat reQuest (HARQ) scheme that would minimize the power consumption of this implementation is proposed.**

***Keywords*-Medical application; power reduction; WBAN; Huffman source coding; Reed-Solomon channel coding**

I. INTRODUCTION

The growing interests and developments in the area of wireless sensor networks have opened up many avenues for the applications of such systems in remote monitoring, whether they may be in medical, environment, security, surveillance, or industrial. Patient monitoring and personal healthcare are the focus of this paper, and power optimization in the wireless communication by the use of source and channel coding are presented [1].

Wireless Body Area Network (WBAN) is one type of network that is considered for patient monitoring, where the sensors are distributed around the body and their communications range is limited to the immediate vicinity. They monitor the body and process the acquired data on the battery operated sensor node and then wirelessly transmit them to a monitoring station for further analysis or alarm. The main constraints of such systems include reliability, area, timing, and efficiency, however the main bottleneck that has been generally accepted is the issue of power consumption [2].

The power consumption of the wireless sensor node is distributed among several different areas of the device, where the most common configurations include one or several sensors, a microprocessor/microcontroller/DSP referred as software component, a Custom Digital Signal Processing unit referred as digital hardware component (FPGA/ASIC), memory, and a transceiver. Since one of the most power-consuming devices is the RF module, in order to achieve minimum system power consumption the most effective solution is to buffer the data and operate the transmitter at the highest possible data rate at low duty cycle, thus minimizing the time in which the communication occurs. This can lead to design constraints such as data rate and packet size, and can have a significant effect on the communication efficiency.

Since WBAN in patient monitoring is employed in an indoor environment with communication occurring through the patients body, it is expected that the channel quality would vary significantly. This would require frequent retransmissions, which is not energy efficient if a number of these errors in the corrupted packets can be corrected, thereby mitigating repeat transmissions. This is done by the use of Forward Error Correction (FEC), which introduces systematic redundancies allowing transmission of data at a reduced energy per bit, achieving the same bit error rate. The cost of FEC is additional decoder power consumption at the receiver. To further increase the throughput of the communication, source coding is used to remove redundancies that are inherent in the data. Compression reduces the amount of energy required per bit-of-information in transmission through the channel. Both methods work on the assumption that power savings in the wireless data transmission can be achieved at the expense of power consumption in the encoding/decoding stages of either the processor or the dedicated hardware.

The work proposed in this paper focuses on a software implementation of a source and channel coding scheme on an 8-bit micro-controller. Huffman code is used for com-

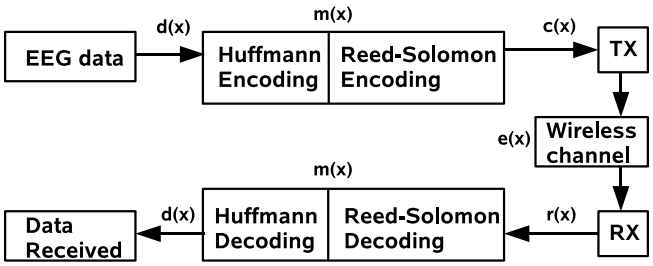


Figure 1. The proposed system flow, illustrating the steps taken from source to sink

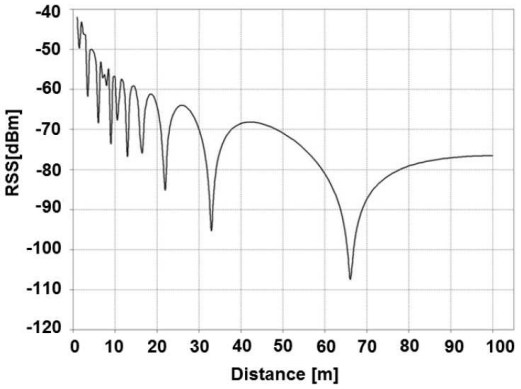


Figure 2. Path loss model for free space with ground reflection at a power level of 0dBm. [5]

pression, and a shortened Reed-Solomon RS(28,24) code over Galois Field GF(2^8) is used for FEC. The system is designed with the applications in the Electroencephalography (EEG) patient monitoring in WBAN architecture. The system itself is implemented on the Tyndall 25mm mote [3] that has an Atmel mega128L processor, and a Nordic nrf2401 transceiver that operate on the 2.4 GHz Industrial, Scientific and Medical (ISM) band. By using the sample EEG data [4], the power measurements are performed in a laboratory environment. The system logical architecture is illustrated in Fig. 1.

Section II discusses the constraints of WBAN and the type of channel that is encountered. Section III of this paper presents the theory and background of source coding and illustrates the factors that affect the quantization and Huffman compression scheme. The theory and methods used in Reed-Solomon channel coding is presented in Section IV. Section V of the paper gives the results of the power reduction and gains in terms of compression and error correcting code. Section VI presents the power savings of using RS to mitigate retransmissions. The conclusion of the findings and the future works are presented in Section VII.

II. WBAN CONSTRAINTS

It is well known that wireless communication has variable channel characteristics that are determined by many factors such as transmitter/receiver power, communication frequency, modulation scheme, reflection, scattering, obstacles, and interferences from other radiating sources. Indoor wireless sensor networks are more susceptible to these factors since the motes operate at lower communication power levels and in environments that contains more obstacles and reflecting sources. The result is that the Receiver Signal Strength (RSS) profile shows pockets of low sensitivity determined by the wave reflections and scattering effects or other factors such as interference and direct obstructions. This in turn affects the Bit Error Rate (BER), and can result in poor communication even at small distances.

Fig. 2 is a path loss model from [5] that illustrates the impact of RSS considering only free space with ground reflections at a power level of 0dBm. It is observed that there is quite a large degradation in the RSS even at small distances.

When implementing wireless communication in WBAN, which has communication distances that are even smaller and that often place the persons body in between the sensor nodes, the problem of reliability in the communication becomes even more pronounced. Therefore it becomes inevitable that severely varying BER would be encountered and the solution becomes that either the transmitter power needs to be increased or some form of error correction needs to be used to alleviate this increase in signal power.

The structure of a WBAN system with a gateway is shown in Fig. 3. The links between the sensors (S) and the master nodes (MN) are the WBAN, and the link with the MN to the monitoring station (MS) is the WLAN. These two types of links have channel characteristic that are different and demand physical (PHY) and media access control (MAC) layers that differ from one another. In the case of WLAN for example, the network can adopt either singlehop or multi-hop schemes, and can have RSS profile that are affected significantly by reflections and scattering from various sources. They may also have different data rate and may use a different frequency for communications than that of WBAN. The WBAN on the other has very small communications range, is usually single-hop architecture and pose a different challenge to the WLAN by often placing direct obstruction (the body) in the line of sight.

One of the works in channel performance and the effect of path loss in the body is presented in [6], where an investigation of the path loss in flat biological tissue at 2.4GHz ISM band is performed. The research draws conclusions on that among the tissue types investigated, the thickness of skin and fat layers have the most variable influence on the path loss, and that proper sizing of the antenna is an important factor. The Research reported in [7] investigates the path loss for the human arm and torso, and path loss parameters were derived from experimental measurements

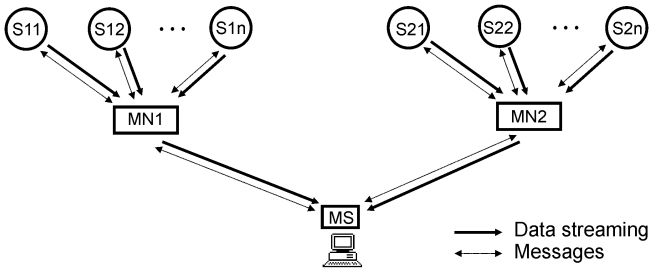


Figure 3. Possible WBAN system connected to WLAN gateway [8]

and are then compared with the model stated in [6]. The study shows that the path loss along the arm and the torso follow the same course but the magnitude of the loss along the torso is higher due to increased absorption, and suggests that the path loss model for a flat homogeneous tissues may underestimate the effect near a human body. These studies show that there is quite a significant amount of degradation in communications efficiency near the human body, and that the designs of WBAN needs to have a high communications power or an appropriate FEC to keep the desired BER.

III. Source Coding

Data compression algorithms can be divided into two families, namely lossless compression and lossy compression techniques [9]. The lossless compression scheme allows perfect reconstruction of the original data, while lossy compression returns an approximation that can achieve better compression rates. The choice of compression scheme used depends heavily on the application and the performance required by the system.

Due to lack of approved standards, medical data is often required by clinical practice physicians to be lossless, and is believed to be an essential requirement for a correct diagnosis. However as it is noted in [10], the diagnosis of 8-bit resolution EEG data gives enough precision to ensure a correct interpretation of the signal by a physician. In this work the EEG data used was sampled with a 12-bit resolution ADC, and an 8-bit micro-controller is used to perform the compression. Basing our system upon the observations made in [10], the ADC data is uniformly quantized from 12-bits to 8-bits. The operation affects the quality of the signal, but also has its advantages by providing easier computations for the 8-bit microcontroller, and by providing less memory requirements for the Huffman codeword LUT.

Huffman code has been chosen due to its simplicity. Given a known probability function, the huffman tree can be built and stored in the SRAM or the flash in the micro-controller. The encoding is performed on the sensor node immediately after data acquisition, using just memory accesses to the codeword LUT. However decoding has higher complexity since it has to assess individual bits to find the corresponding codeword in the LUT. But this is performed on the master node, which is a more powerful machine, hence is not considered critical.

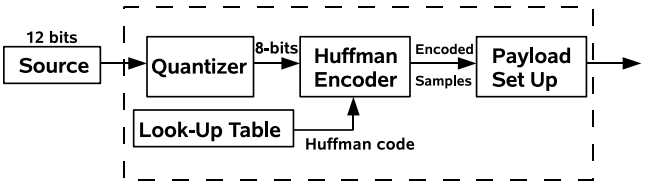


Figure 4. The procedure for source encoding

Therefore for this work a trade-off solution is proposed, where some error (loss) is introduced before the lossless compression stage. The steps of this technique are illustrated in Fig. 4.

The initial step quantizes the 12-bit EEG data to 8-bits, which results in resolution-loss of the amplitudes that is uniformly distributed over the whole dynamic range of the data set. The quantized EEG data is then compressed using the static Huffman encoder, and is subsequently passed into a payload setup, where bit packing is performed. The packet is then either encoded by the channel-encoding scheme, or wirelessly transmitted directly.

A. Quantization

The procedure for the quantization is as follows. The EEG data set is uniformly quantized over the entire dynamic range. The error introduced by the quantizer varies with the maximum amplitude that has to be represented after the quantization, where the peak of the Gaussian distribution plays an integral role in the choice of the quantization interval. This peak amplitude parameter VM is chosen so as to minimize the negative effect of the quantization without significantly affecting the compression performance. The cost-function in (1) is used to select VM.

$$min(VM)\left[J = \alpha Err(VM) + \frac{(1-\alpha)}{Gain(VM)}\right], \quad (1)$$

where the term $Gain(VM)$ is an expression of the source coding gain (explained in details in Section V.A), and the term $Err(VM)$ is an error-function defined by:

$$Err(VM) = \sqrt{\frac{1}{S}\sum_{i=1}^{S}|q_i - d_i|^2 w_i}, \quad (2)$$

where S is the number of EEG samples in the entire training set, $|q_i - d_i|$ is the difference between the quantized value and the original sample, and w_i is the weight-coefficient. i.e. the frequency of occurrence of each value in the database. The α value is a term used to vary the weight between the error and the coding gain.

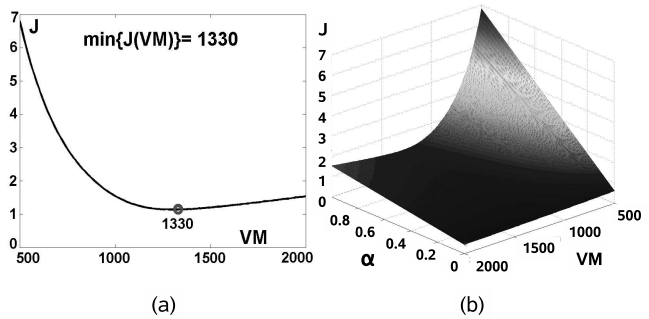


Figure 5. (a) Illustration of the local minima for the 2D Cost function without α (b) 3D Cost function

The plot of the cost function $J(VM, \alpha)$ is shown in Fig. 5(b). It is seen from the figure that this is not a convex function, there are no local minima, and the global minima occurs at α = 0. The conclusion is that α does not contribute to the cost function and is trivial when trying to determine the optimum value of VM. Therefore the cost function is analysed without α. This is illustrated in Fig. 5(a). The minima of this new function $J(VM)$ is found to be at VM = 1330, which is an integer representation of the voltage value that was measured on the sensor.

After the quantization, the data will assume only 256 different values. This is the base-requirement for the next step of the source coding algorithm.

B. Huffmann Coding

Given a discrete source of symbols, the Huffman coding procedure builds for that source a variable-length minimum-redundancy prefix code. The minimum redundancy code is often referred as optimum in the sense that it leads to the lowest possible average codeword length, given N symbols and M digits. It is important to note that for every source there are several optimum codes. We considered only prefix codes, in which a codeword cannot be a prefix of some other codeword, because there is no loss of generality in considering only the prefix ones. In fact it can be shown that given a general optimum code, there exists a prefix code that achieves the same results [10].

The Huffman code tree [11] has a number of leaves proportional to the number of different symbols coming from the discrete source: in this case they are 256, as the Discrete Source is the Quantizer from the previous step. Each leaf contains a string of bits, here called codeword that corresponds to an input symbol. Going from the root deep through the code tree, the symbols become step-by-step less likely, while the related codewords much longer. The average length of the code is defined as,

$$L_{av} = \sum_{i=0}^{N-1} b_i v_i, \qquad (3)$$

where b_i is the codeword, and v_i is the corresponding error probability. The Huffman compression scheme is designed so that the average length of the code is smaller than the uncoded version.

An issue related to this code is that it is impossible to put a limit on the maximum length of a codeword. This is due to fact that the maximum depth of the Huffman tree results from the code-design procedure without any possibility of control. This is a problem since a fixed maximum codeword length is required when working with an 8-bit microprocessor that cannot efficiently handle variables exceeding the size of 8-bits.

To solve this problem a hybrid technique based on the Collapsed Huffman Tree (CHT) is used [10]. Each Huffman codeword larger than length 8-bits are appended a CHT codeword that flag such a case. The resulting final codeword has 16-bits. It has been observed that this does not have a significant negative impact on the average length of the code, because the CHT leaf collects the most unlikely symbols of a given source.

C. Multiple Huffman codes

A possible variant of both classic Huffman and CHT coding is the use of Multiple Huffman codes, i.e. a family of codes that allow the encoder and the decoder to switch between them, by following a certain rule.

By definition, Huffman codes are built (and optimized) to best represent the source of symbols they have been constructed from. This means that such a code has good performances when the training sequence used to build the code is truly representative of the source of symbols that are to be encoded. These considerations imply that a Huffman code is optimum when the source that has been compressed emits the symbols with a homogeneous time-independent distribution. Even if this is true over a large time scale (which is the assumption that the Huffman coding takes), it could be not true over a small time window. In the case of EEG signal processing, it is observed that the distribution of amplitudes differs from normal activity to seizure activity as shown in Fig. 6.

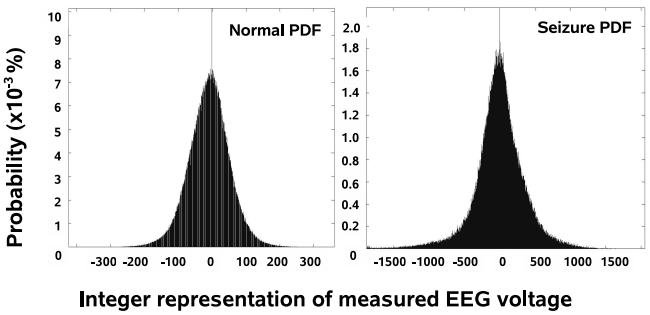


Figure 6. Distribution of amplitudes for a large number of samples taken from normal activity (left) and abnormal activity (right)

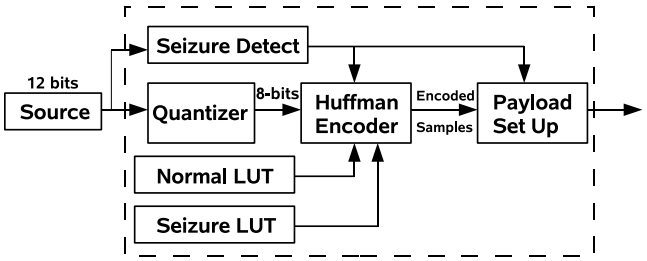


Figure 7. Modified compression scheme for seizure-aware coding

The knowledge of when abnormal activity occurs during a long-term EEG examination would permit to introduce further optimizations to this scheme by the use of multiple Huffman codes. A possible scheme to realize this idea is shown in Fig. 7.

In such a case where the sensor node is aware of the current brain state by means of either an internal procedure or an external indication, the seizure detection block chooses the appropriate Huffman tree to perform the encoding. In this implementation, two Huffman trees are used; one optimized for normal activity, and another one for seizures activity. The codewords are packed in the payload setup as before.

IV. Reed Solomon Channel Coding

The channel code that was chosen for this particular implementation of FEC is the shortened Reed- Solomon (28,24,4) over the Galois Field size of GF(2^8). The RS code [12] is traditionally popular and has been a longtime industry-standard that has found uses in various applications such as satellite communications, Digital Video Broadcasting (DVB), Compact Disks (CD), Digital Versatile Disks (DVD), mass storage, and in wireless communications. The main reasons for such wide-ranging popularity stems from the fact that the RS code has efficient encoding and decoding algorithms, and targets (multiple) burst errors.

A. Reasons for code choice

The RS(28,24,4) over the GF(2^8) is implemented in this scheme due to a variety of reasons. One of the requirements for a power-constrained wireless sensor mote is to operate the most power consuming devices as seldom as possible. The transceiver is then needed to work at as small time frame as possible and at high data rate. These constraints lead to the need of the packet to be small but also at the same time to keep the power consumption of the FEC as low as possible. Therefore the FEC should keep the size small and be computationally easy. This is especially the case for the power-constrained sensor nodes, but is also true for the master node in terms of time taken to decode, which becomes critical in networks that utilize Time-Division Multiple Access (TDMA) protocols.

Various research groups have performed works in the field of feasibility of FEC in Wireless Sensor Networks (WSN). The research done in [13] explores the power estimation of Hamming codes, Convolutional codes (CC), and RS codes and has proposed a framework for the design space of FEC for WSN. It was established that RS codes perform the best in terms of total energy consumed by the motes, and it was seen that packet size of 31 bytes consumes the least amount of power at varying node distances. Through the BER analysis it is shown that the computational power of RS(31,29,3) is the lowest at BER of 10^{-4} and RS(31,27,5) is the lowest at path loss exponent of 4, which equates to a dense noisy environment. The authors of [14] have performed works on power estimation of various BCH, RS, and CC cores on tsmc180nm ASIC process and Xilinx Spartan III FPGA platform. It is shown that the power consumption of linear block codes are much less than that of CC, and that BCH and RS codes are useful in WSN applications. In [15], empirical research has been performed on the BER of the motes in short range ($<$ 1.5 m) WSN, and investigated the packet loss and packet reception of the sensors. The packets are sized to 64 bytes, and through their analysis the authors found that the average number of bit errors in a packet that passed synchronization is 16.68 bits.

The maximum packet size of the burst mode in the nrf2041 is 32 bytes, of which 4 bytes are set for synchronization and address. The maximum size of the packet payload is 28 bytes, and thus the code is built with n=28. The reason for choosing the field size of GF(2^8) is due to the 8-bit micro-controller, where all operations are performed in 8-bit blocks, therefore for ease of operation, and to reduce computational power, the code is built over this field size.

The combination of these reasons not only shows that the choice of RS(28,24) is acceptable for short range WSN, but also justify them from an implementation point of view and that the performance should be similar to the stated figures.

B. Code Architecture

Following the encoding of the EEG data by the Huffman compression scheme on the sensor node, the resulting message $m(x)$ is systematically encoded by the concatenated code and is sent wirelessly through a highly noisy indoor environment. The systematic encoding ensures that the data symbols appear in the codeword, and is expressed, using polynomial notation as follows,

$$c(x) = p(x) + x^{n-k}m(x), \tag{4}$$

where the parity symbols $p(x)$ is chosen such that the codeword $c(x)$ is divisible by the generator polynomial.

Due to the complexity of the Galois field multiplication, and the limitations of the processor to perform only one operation per clock cycle, a bit serial multiplier is used for multiplications in GF(2^8), and a look-up-table is made for inversion to facilitate maximum speed of the operation [16].

Once the packet is received on the master node, the word may be corrupted and this is expressed as $r(x) =$

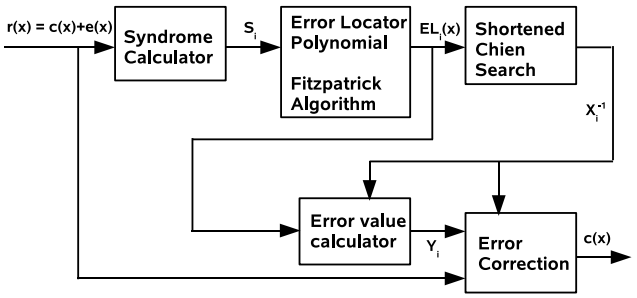


Figure 8. Reed-Solomon Decoder architecture

$c(x) + e(x)$, where $r(x)$ is the received data, $c(x)$ is the codeword, and $e(x)$ is the added channel noise. The RS decoder attempts to correct the errors by means of polynomial operations. The decoding flow can be seen in Fig. 8.

The $(n - k = 2t)$ syndromes S_i are computed and these are used to calculate the Error Locator Polynomial $EL_i(x)$ in an iterative fashion. The commonly used Berlekamp-Massey algorithm [17] for the calculation of the error locator polynomial is here replaced with the Fitzpatrick algorithm [18] due to its faster performance.

Once the Fitzpatrick algorithm has calculated the error locator polynomial, the roots of the polynomial, X_i^{-1} are then found by the shortened Chien search algorithm, which only cycle through the last 28 elements of the GF(2^8) since the code has been shortened to RS (28,24). This observation leads to a considerably shortened computational time for the search calculation.

The resulting roots X_i^{-1} are the inverses of the error locations in the received word and are used for the calculation of the error values Y_i. These error values are calculated without the use of the error evaluator polynomial, as was proposed in [19]. The equation used for the error value calculations is shown in (5).

$$Y_i = \frac{X_i^{-2_t+1}}{EL_{i'}(X_i^{-1})EL_i'(X_i^{-1})}, \quad (5)$$

where $EL_{i'}(x)$ is the update polynomial obtained through the Fitzpatrick algorithm, and $EL_i'(x)$ is the formal derivative of $EL_i(x)$. This algorithm has the advantage over the traditional Forney algorithm [20] in that it does not have to calculate the error evaluator polynomial, thereby saving significant computational effort.

Finally the error is corrected with the use of the error values Y_i and their corresponding error positions X_i^{-1}.

V. Power Measurements and Results

Several implementations are considered for the analysis of the power consumption and the performances of the source and channel codes. These are summarized in Table I.

Table I
Communication Versions Used

Version	Description
Orig.	12-bit EEG data that is transmitted as a 16-bit word
Quant.	Quantized 8-bit EEG data
QH	Quantized 8-bit EEG data with Huffman coding
QRS	Quantized 8-bit EEG data with RS coding
QHRS	Quantized 8-bit EEG data with Huffman and RS coding

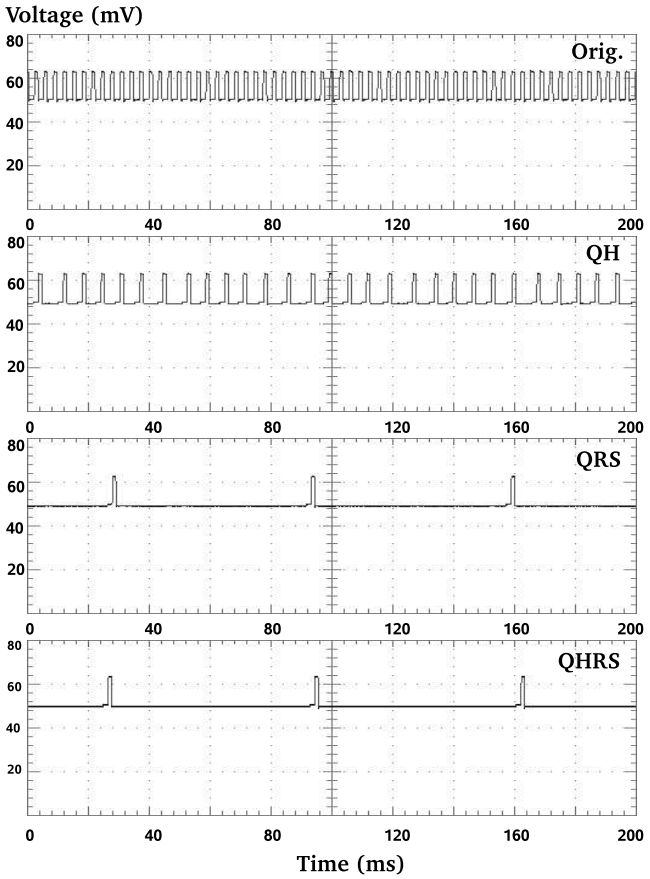


Figure 9. TX system current consumption vs. time

The data is sent wirelessly in burst mode of the nrf2041 at data rate of 250 kb/s and at TX power level of -20dBm. The transmission packets are sized to 32 bytes per packet, with 28 bytes of information payload.

The result current consumption waveforms at the transmitter for the versions 1 to 4 are shown in Fig. 9, where each peak corresponds to a packet being sent. The waveforms are captured via voltage measurement across a 1.7 Ω external resistor with the supply voltage for the Tyndall mote set at 5V. The micro-controller operates on 3.3V, and the transceiver operates on 2.5V. The power consumption of the voltage regulators and the surrounding circuitry are assumed to be negligible. It should also be noted that

Table II
TX POWER AND ENERGY CONSUMPTION

Version	Pac rate (s^{-1})	Pwr TX (mW)	En /Pac TX (mJ)	Pwr RX (mW)	En /Pac RX (mJ)
Orig.	320.5	65.334	0.2038	88.2	0.2752
QH	159.2	59.168	0.3716	88.2	0.5540
QRS	15.21	54.274	3.5683	88.2	5.7988
QHRS	14.79	54.247	3.6678	88.2	5.9635

the development board on which the measurements are performed adds 20.38mV shown in Fig. 9, and needs to be subtracted in order to calculate the power consumption of the mote itself.

Table II summarizes the communication performance and the energy usage of the various schemes at the transceiver power level of -20dBm.

Moving down the column for the different versions, it is observed that the power consumption of the system drops due to a decrease in the packet rate, but the overall energy consumption rises due to the heavier processing demands of the codes. This is due to the limitations of the 8-bit microprocessor to carry out tasks such as efficient bit-wise Galois-Field multiplication, hence high amounts of computational time were introduced that led to reduced packet rate.

In regards to *Orig.* and *QH*, it is observed that *Orig.* has twice the packet rate of *QH* due to the necessary computational effort required by the Huffman coding operation in *QH*. However as seen in Table V, the compression gain works out to be around 60% for most of the time, and therefore the overall throughput of the Huffman encoded data is superior.

In terms of channel coding, it is seen from Table II that communication has higher throughput and hence better performance when the code is not used. However these results do not take into account of the coding gain provided by the FEC nor the channel characteristics. Investigation and results on these aspects are provided in Section V.C and Section VI respectively. It is shown that there are advantages to be had in using FEC in terms of coding gain and in mitigating retransmissions as well as the system power.

A. Compression Performance

The Compression rate is strictly dependent on the average length of the source code, L_{av} as defined in (3). We consider the case of a Huffman tree built using the entire EEG database as the training set with VM=1330 and 8-bit quantization. The compression performance of huffman and collapsed huffman trees are shown in Table III.

It is seen from the table that the efficiency of the CHT technique η_{cht} is lower than the original one η , and the average length of the CHT $L_{av_{cht}}$ is higher than the original L_{av}. Although this is a slight trade-off in the performance

Table III
COMPRESSION FIGURES

Term	Term Description
H = 5.345	Entropy of the signal from quantizer
L_{av}=5.366	Avg. length for the standard Huffman tree
$L_{av_{cht}}$ =5.96	Avg. length for the Huffman tree with the CHT leaf
CHT = 38	Position of the CHT leaf in Huffman tree
η = 0.996	Efficiency of Huffman coding
η_{cht} = 0.897	Efficiency of Huffman coding with the CHT leaf
Mem=507	Memory usage of original LUT in Bytes
Mem_{cht}=38	Memory usage of CHT LUT in Bytes

of the code, it is also seen from the comparison of Mem and Mem_{cht} that there are large memory savings of approximately 92.5% to be had from using CHT.

Although the figures of Table III give the performace of the code that is optimized for the given statistics, it should be noted that the EEG sample used to build the tree contains 20% seizure activity. Since it is rare to see 20% seizure activity in the real case, a test that was devised to explore the compression scheme on various EEG data types. i.e. Seizure and Non-seizure activity. The test consists of transmitting the EEG data in 200 packets through the wireless link, and counting how many information bits are communicated. The average packet length is sized to 24 bytes, where for *QH* one byte is reserved for information on the number of significant bits in the data payload. Fragment 1 holds EEG data activity of seizure prone patients, Fragment 2 holds the data of the patients when in seizure, and Fragment 3 is the normal EEG data of healthy adult subjects. The results are summarized in Table IV.

Table IV
COMMUNICATION PERFORMANCE WITH VARIOUS TYPES OF DATA

FRAGMENT 1	Orig.	Quant.	QH
Total number of packets	200	200	200
Total EEG samples sent	2400	4800	7725
Avg. data payload length (bits)	192/192	192/192	179/184
FRAGMENT 2	Orig.	Quant.	QH
Total number of packets	200	200	200
Total EEG samples sent	2400	4800	3605
Avg. data payload length (bits)	192/192	192/192	177/184
FRAGMENT 3	Orig.	Quant.	QH
Total number of packets	200	200	200
Total EEG samples sent	2400	4800	8690
Avg. data payload length (bits)	192/192	192/192	181/184

The table shows that the quantized values, *Quant.* sends twice as many samples than that of original values *Orig.* and that *QH* achieves even better performance for Fragments 1 and 3. The average data payload length varies for *QH* due to the variable length nature of Huffman coding. To

Table V
HUFFMAN PERFORMANCE WITH VARIOUS EEG

	FRAGMENT 1	FRAGMENT 2	FRAGMENT 3
Avg. length	4.682	9.878	4.1988
$Gain_{cht}$	2.560	1.210	2.850
Gain (%)	60.91	17.35	64.91

summarize the compression performance, the experimental average code length and the corresponding compression gain of the Huffman encoded data for each Fragment can be seen in Table V, where only *QH* is considered. The estimation of the overall compression gain is made using a simple relationship in (6), where 12 is the original resolution of the sample.

$$Gain = \frac{12}{L_{av}}, \qquad Gain_{cht} = \frac{12}{L_{av_cht}}, \tag{6}$$

The results show that coding gain is strictly dependent on the nature of the signal, and if considering that segments such as fragments 1 and 3 occur for more than 99% of the time, the little increment from 8 to 9.878 bits per sample (during the seizure activity) does not affect the compression capabilities of the system over a macro-scale.

B. Multiple Huffman codes

Previous considerations on EEG signal compression with CHT Huffman code show that the coding performances in terms of accuracy and coding gain are strictly related to the composition of the signal entering the encoder. This is inherent to the Huffman-code building algorithm, since seizures (and abnormal activity in general) are less likely than normal activity over a large time scale, than a general activity Huffman code. Even if the encoding scheme here presented can represent any sample with similar level of quality (due to uniform quantization), the performance of the encoder decreases when abnormal activity is present.

In this section, a mixed-activity signal with known occurrences of seizure activity has been encoded using both single and multiple codes. Using the same criterion as (1), the best value for VM (which is a parameter related to quantization intervals size, i.e. to accuracy) are chosen, and two more Huffman codes are built. Based on these, a mixed code is implemented that takes advantage of seizure awareness (available by construction, in this particular case) to introduce further optimization.

The results of the comparison are shown in Table VI, where f is the expected frequency of occurrence of seizures. It is seen that the seizure-aware (multiple Huffman) coding shown in column 4 introduces a double advantage.

First, it encodes the mixed-activity samples with an increased efficiency in comparison to the general code (column 3). This can be observed by looking at the larger number of samples-per-packet sent. Secondly, since the value VM is targeted on the particular class of brain activity, the effects of quantization on signal quality are lower, i.e. the reconstructed (decoded) samples are more similar to the original ones. For example if f = 0.1 (which implies 10% seizure activity), the average error would result to be 6 times lower than the case when the general code is used. This means that the reconstructed signal would have 6 times more accuracy than before. This shows that the multiple-tree scheme brings improvements with respect to the single-tree implementation in terms of compression and the accuracy of the data due to the change in the quantization intervals.

Table VI
A PERFORMANCE COMPARISON BETWEEN DIFFERENT HUFFMAN CODES

Signal	Normal activity	Seizure activity	Mixed activity	
Code	Normal act. code	Seizure act. code	General code	Multiple code
VM	250	1680	1330	Norm, Seiz
ERR (VM)	0.07	0.46	0.64	(1-f)*0.07 + f*0.46
EEG / Pac	25.06	20.03	20.15	24.29

C. Reed Solomon Coding Gain

The coding gain of the RS(28,24) code is usually calculated by setting the desired bit error rate of the uncoded BPSK and the coded BPSK and then measuring the difference between the Signal-to-Noise Ratio (SNR) required to reach such BER. This is achieved by varying the transmitter power levels. However due to the limitations of the transceiver of the Tyndall mote in setting the TX power levels, the SNR was modified by varying the distance between the TX and RX at a constant transceiver power level of -20dBm. It is also noted that the modulation scheme used by the nrf2041 is Gaussian Frequency Shift Keying (GFSK). The results of the measurement are shown in Fig. 10. It should be noted that the distance is inversely proportional to the SNR, and also that the BER decreases with rising SNR.

It is observed visually that as the BER decreases, the gap between the GFSK(uncoded) and that of RS(28,24) become larger. It is interesting to note that there are certain points when the communication becomes worse even at small distances where one would expect lower BER, such as the case at the distance 0.15m. At other distances such as 0.95m to 1.3m, the effect is more prominent. This is due to path loss and reflections from the ground and walls, which interact with the original signal to produce low receiver signal strength. To measure the coding gain of the RS code, the following model is used [21],

$$P_{TX,U}[W] = \eta_U \frac{E_b}{N_0} N \left(\frac{4\pi}{\lambda}\right)^2 d^n, \tag{7}$$

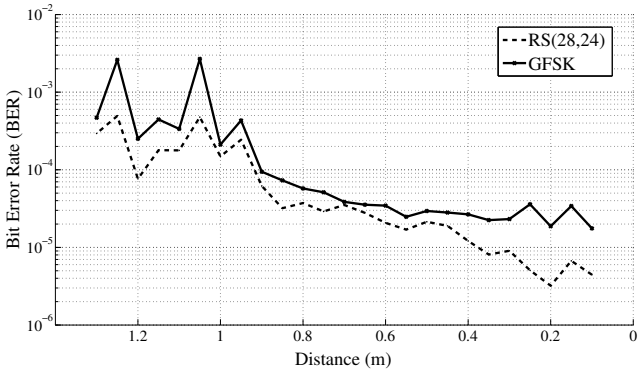


Figure 10. Plot of TX-RX distance vs. Bit Error Rate

where:
η_U = Spectral efficiency (=1 for BPSK)
E_b/N_0 = SNR (energy ratio)
N = Signal noise (Thermal noise*Bandwidth)
λ = Transmitted wavelength
d = Distance between TX and RX
n = Path loss exponent

By placing the coded and uncoded case in proportion, the following relationship is obtained.

$$\frac{P_{TX,RS}[W]}{P_{TX,U}[W]} = \frac{\eta_{MAX}\left(\frac{E_b}{N_0}\right)_{RS} N\left(\frac{4\pi}{\lambda}\right)^2 d^n}{\eta_{MAX}\left(\frac{E_b}{N_0}\right)_{U} N\left(\frac{4\pi}{\lambda}\right)^2 d^n}. \quad (8)$$

Since the same channel and transmitter are used, the above equation can be simplified to,

$$\frac{\left(\frac{E_b}{N_0}\right)_U}{\left(\frac{E_b}{N_0}\right)_{RS}} = \left(\frac{d_{RS}}{d_U}\right)^n. \quad (9)$$

To account for experimental error, the BER values shown in Table VII were chosen to calculate the coding gain. The table illustrates the values of the FEC gain in relation to various environmental conditions expressed as the path loss exponent, where n=2 is for free space, n=3 is for indoor environment, and n=4 is for indoor environment with many obstructions.

It is observed from Table VII that the coding gain rises as the BER is reduced, and that it starts to make larger gains at BER less than 10^{-4}. As mentioned previously, due to the geometrically constrained antenna, the BER is expected to be lower than these values for a system with a better matched antenna and that perhaps these gains actually start at even lower BER. Considering that the frequency of the transceiver used is in a busy communication band (2.4GHz), and the testing is performed in a laboratory setting with plenty of reflections, it is reasonable to assume that the value of the path loss exponent n=3 for this scenario.

Table VII
FEC GAIN FOR VARIOUS ENVIRONMENTAL MODELS

BER	2x10^{-5}	3x10^{-5}	9x10^{-5}	2x10^{-4}	3x10^{-4}
Gain dB (n=2)	5.0263	2.8394	0.5097	0.3855	0.4706
Gain dB (n=3)	7.5394	4.2591	0.7646	0.5782	0.7059
Gain dB (n=4)	10.0526	5.6788	1.0194	0.7709	0.9412

VI. ANALYSIS OF THE CHANNEL

The BER was measured by tallying the number of byte errors that occurred in the communcation at varying distance values of 10cm to 130cm. They are then normalized and presented in terms of overall percentage of errors that occurred in the experiments. These are shown in Fig. 11. To account for the variation in the channel characteristic, the following analysis was performed on four different types of communication based on their bit error rates. They are summarized in Table VIII, where the BER and distance values correspond to the GFSK plot shown in Fig. 10.

Table VIII
CHANNEL SEGMENTS

Segment	Description
10_to_50	Between BER of 2x10^{-5} and 3x10^{-5} (10→50cm)
55_to_90	Between BER of 3x10^{-5} and 1x10^{-4} (60→90cm)
95_to_130	Between BER of 2x10^{-4} and 4x10^{-4} (95→130cm)
105_&_125	BER of 3x10^{-3} (105cm and 125cm)

A. Error Distribution

The plots in Fig. 11 show that a vast majority of the errors that occur are short bursts, where for segments *10_to_50*, *55_to_90*, and *95_to_130*, single byte errors occur at around 60-70% of the time. As the BER rises beyond 10^{-3} (segment *105_&_125*), the error distribution flattens out and is seen that longer burst type errors occur (up to 6 bytes) more frequently. Single byte errors happen less often (40%), and the energy of the noise looks to be spread more around a few bytes. It can also be observed that there is a slight rise in the 28-byte error type in all the segments in this figure. This is considered as packet loss, where an entire packet becomes corrupt and needs to be retransmitted.

From the error distribution, it is also possible to calculate the retransmission rate required for error free communication, and how FEC can be used to reduce this value, hence save energy. Equation (10) is used to calculate the required packet rate.

$$ARQ_m = \frac{\sum_{i=m}^{28}\sum_{j=i}^{28} e_j}{T_{Pac}}, \quad (10)$$

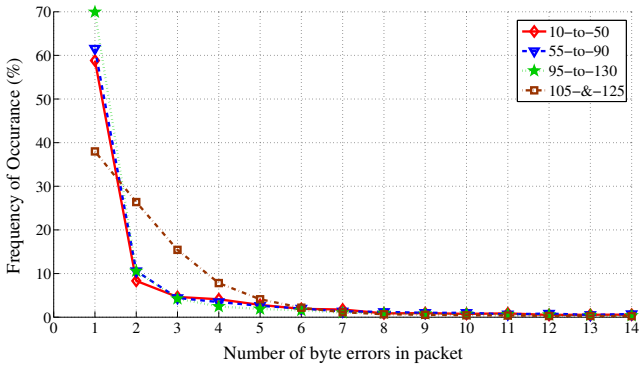


Figure 11. Types of errors occuring for various channel characteristics

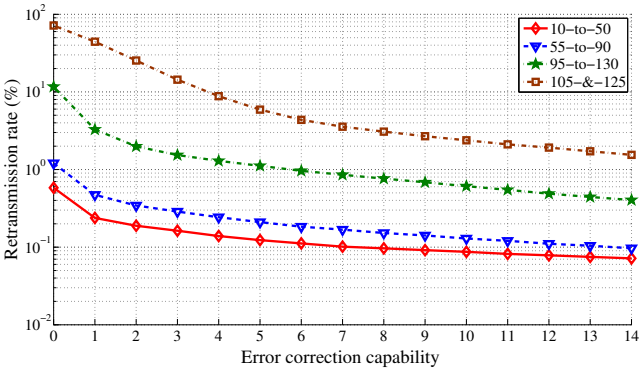


Figure 12. Required retransmission rate for error free communication

where ARQ_m is the retransmission rate, e_j is the error types shown in Fig. 10, T_{Pac} is the total number of packets sent over the channel, and m is the range of error correction capability. The calculation of the retransmission rate is performed and the results are shown in Fig. 12.

The plot shows that with a larger BER such as segment *105_&_125*, the overall retransmission rate is higher of around a factor of 10^2 than that of almost error free communication of segment *10_to_50*. Also it can be observed that the addition of FEC alleviates the need for retransmissions, and this value goes down as the error correcting capability rises.

B. Power savings

Considering that the communication occurs at the same packet rate for the uncoded and the coded versions in the link, it is possible to calculate the power savings of the system. Such assumption can be achieved by using a code thats implemented in hardware such as an FPGA or an ASIC. Therefore one can derive an optimal operating point for FEC that best utilizes the energy when working at varying BER levels at this packet size and data rate. Although the experiment did not directly place any obstructions between the TX and RX, it did however reach a distance in which the receiver sensitivity failed. The experiment was also performed in a closed laboratory environment that provided strong reflections of the transmission. Therefore it is assumed that at higher transmit power (e.g. -10dBm) on a real patient would provide a similar type of scenario, and that the results could be used to design a system specific HARQ.

The power and energy savings are calculated by comparing the original values without any coding with that of increasing error correction capability. Equation (11) is used to calculate the new packet rate Pac_{new_m} from the original packet rate Pac_{orig} and the retransmission requirements RT_m that is in terms of percentage.

$$Pac_{new_m} = \frac{100 Pac_{orig}}{Pac_{orig} + \left(\frac{Pac_{orig} RT_m}{100}\right)}. \quad (11)$$

The total transmission time, T_{tot} is calculated using (12), where T_{TX} is the time taken to transmit a packet.

$$T_{tot_m} = T_{TX} Pac_{new_m}. \quad (12)$$

The average power is found using the information about the power consumption of the micro-controller $P_{\mu C}$, and the power consumption of the transceiver P_{TX}. By knowing the time it takes to transmit the packets in a given window, the product of the respective times and the known powers give the average power consumption of the system by following the relationship shown in (13).

$$\begin{aligned} P_{Avg_m} &= T_{tot_m}(P_{\mu C} + P_{TX}) + (1 - T_{tot_m}) P_{\mu C} \quad (13) \\ &= P_{\mu C} + P_{TX}(T_{tot_m}). \end{aligned}$$

P_{Avg_m} is used to find the average energy E_{Avg_m} that is the average energy consumed by the system to transmit one packet. This is achieved using (14).

$$E_{Avg_m} = \frac{P_{Avg_m}}{Pac_{new_m}}, \quad (14)$$

The energy savings is calculated by taking the difference in the average energy consumption per packet for transmission with no coding, $E_{Avg_{No\_FEC}}$ and that of energy consumption of coded transmission E_{Avg_m}, as shown in (15).

$$E_{Savings_m} = E_{Avg_{No\_FEC}} - E_{Avg_m}, \quad (15)$$

From the energy savings, the average power savings or the difference in terms of power between the uncoded and the coded transmission is calculated at the respective required packet rates as shown in (16).

$$P_{Savings_m} = E_{Savings_m} Pac_{new_m} \quad (16)$$

Fig. 13 illustrates the amount of power saved for the various segment types plotted against the error correcting

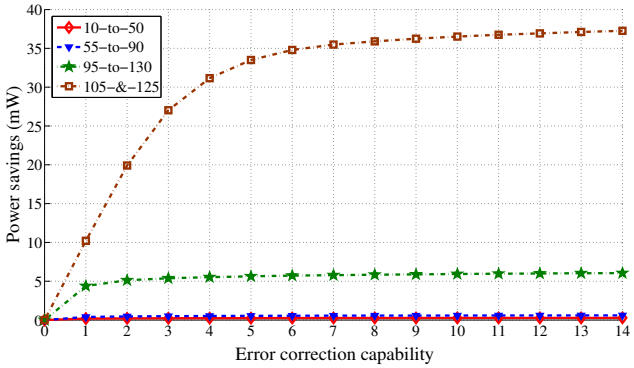


Figure 13. Power savings using shortened Reed-Solomon (28,k,t) with various error correcting capabilities

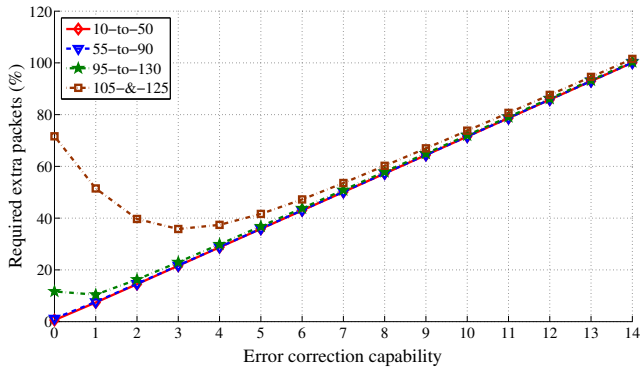


Figure 14. Error correction capability vs. Extra transmissions needed (%)

capability. The zero point on the horizontal axis refers to the case when no FEC is used. It is observed that the power savings approach a certain threshold where the addition of more FEC does not give significant savings. It is also seen that worse the channel, the more benefit in terms of power savings of the wireless sensor mote is achieved by the addition of FEC, and that hardly any power savings are to be had in segments *10_to_50* and *55_to_90*.

Although the addition of FEC can mitigate retransmissions, the cost of this operation is the additional parity bits in the packets, which reduces the overall throughput of the data. Therefore it is also important to consider the cost of total throughput versus the gains achieved by the utilization of FEC. This is illustrated in Fig. 14, where the additional packets needed for a complete transmission of a data set in terms of percentage is shown versus the error correction capability. The point where no error correction is available refers to a link that utilizes just retransmissions.

The figure shows that for segments *10_to_50* and *55_to_90*, the addition of FEC does not result in better throughput. However at a higher BER of segment *95_to_130*, it is observed that there is a slight improvement at m=1. Moving up to segment *105_&_125* shows that there

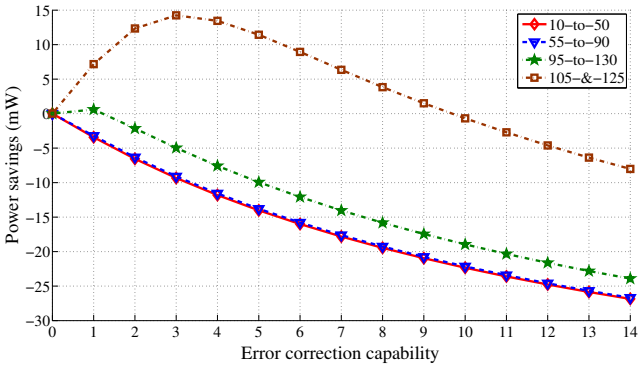


Figure 15. Total TX system power savings using FEC considering throughput

is significant savings as the error correction capability rises, and that the optimum for this BER is at m=3.

The total power savings is calculated by applying (11) to (16), to the results shown in Fig. 13 and Fig. 14. The retransmission requirement RT_m in (11) is modified to incorporate the effect of the throughput. The calculated total power savings are plotted in Fig. 15.

The total power savings shown in the figure strongly reflects the effect that the throughput has on the effectiveness of the FEC. Again it is seen that there are hardly any savings by the FEC at low BER, but it does start to become energy efficient at above BER of 10^{-4}. In segment *95_to_130* it is observed that there is small savings of 0.6mW at m=1, and at segment *105_&_125* the optimum error correction value is at m=3 with savings of 14mW.

VII. Conclusion and Future Work

The possibility of energy savings using a software implementation of a serially concatenated Huffman-RS code was presented. The analysis of the Huffman compression show that EEG compressions gains are achieved in a general case, and that the use of multiple trees for normal and seizure activity present even more gains.

In terms of the mote system performance however, the implementation presented long computational time that made the coding seem less practical. The main reason is due to the limitations of using the 8-bit microprocessor, where the computationally difficult Galois Field operation for the Reed-Solomon code presented significant addition to the processing time. Thus the packet rate was reduced and the energy consumption per packet was increased. This would result in faster depletion of the battery on the real system, which would have a negative effect on the maintenance of the patient monitoring wireless sensor node.

The error distribution for the channel at various BER levels was also performed, and the power savings by the mitigation of retransmissions was calculated. The findings show that FEC can save power at BER levels of 10^{-4} or

higher, and approach optimum values at error correction capabilities of up to 3 bytes for this implementation.

Future work will explore the hardware-software codesign of the Reed-Solomon code along with a HARQ scheme, and also investigate the advantages of joint source-channel coding for medical applications WBAN.

ACKNOWLEDGMENT

The authors wish to thank the Tyndall National Institute for their support in the provision of hardware through the SFI-NAP scheme and for the facilitation of the testing process. This work is funded by SFI-EEDSP for Mobile Digital Health, grant number: 07/SRC/I1169.

REFERENCES

[1] R. Mc Sweeney, L. Giancardi, C. Spagnol, and E. Popovici, "Implementation of source and channel coding for power reduction in medical application wireless sensor network," in *Third International Conference on Sensor Technologies Applications (SENSORCOMM'09)*, Athens, Greece, Jun. 2009, pp. 271–276.

[2] S. Drude, "Requirement and application scenarios for body area networks," in *Mobile and Wireless Communications Summit 2007: 16^th IST*, Budapest, Hungary, Jul. 2007, pp. 1–5.

[3] Tyndall National Institute, Available at: http://www.tyndall.ie/mai/25mm.htm.

[4] R. G. Andrezjak, K. Lehnertz, F. Mormann, C. Rieke, P. David, and C. Elger, "Indications of nonlinear deterministic and finite dimensional structures in time series of brain electrical activity: Dependence on recording region and brain state," *Physical Review E*, vol. 64, no. 6, p. 061907, Nov. 2001.

[5] T. Stoyanova, F. Kerasiotis, A. Prayati, and G. Papadopoulos, "A practical rf propagation model for wireless network sensors," in *Third International Conference on Sensor Technologies Applications (SENSORCOMM'09)*, Athens, Greece, Jun. 2009, pp. 194–199.

[6] L. Roelens, W. Joseph, and L. Martens, "Characterization of the path loss near flat and layered biological tissue for narrowband wireless body area networks," in *International Workshop on Wearable and Implantable Body Sensor Networks, (BSN'06)*, Cambridge, Massachusetts, U.S.A, Apr. 2006, pp. 50–56.

[7] E. Reusens, W. Joseph, G. Vermeeren, and L. Martens, "On-body measurements and characterization of wireless communication channel for arm and torso of human," in *International Workshop on Wearable and Implantable Body Sensor Networks, (BSN'07)*, Aachen University, Germany, Mar. 2007, pp. 264–269.

[8] S. Marinkovic, C. Spagnol, and E. Popovici, "Energy-efficient tdmabased mac protocol for wireless body area networks," in *Third International Conference on Sensor Technologies Applications (SENSORCOMM'09)*, Athens, Greece, Jun. 2009, pp. 604–609.

[9] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*. Norwell, MA: Kluwer Academic Publishers, 1992.

[10] G. Antoniol and P. Tonella, "Eeg data compression techniques," *IEEE Transactions on Biomedical Engineering*, vol. 44, 2, pp. 105–114, Feb. 1997.

[11] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the I.R.E.*, vol. 40, 9, pp. 1098–1101, Sep. 1952.

[12] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, 2, pp. 300–304, Jun. 1960.

[13] S. Chouhan, R. Bose, and M. Balakrishnan, "A framework for energy-consumption-based design space exploration for wireless sensor nodes," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, 7, pp. 1017–1024, Jul. 2009.

[14] G. Balakrishnan, M. Yang, Y. Jiang, and Y. Kim, "Performance analysis of error control codes for wireless sensor networks," in *International Conference on Information Technology, (ITNG'07)*, Las Vegas, Nevada, USA, Apr. 2007, pp. 876–879.

[15] A. Willig and R. Mitschke, "results of bit error measurements with sensor nodes and casuistic consequences for design of energy-efficient error control schemes," in *Proc. 3rd European Workshop on Wireless Sensor Networks*, Zrich, Switzerland, Jan. 2006, pp. 310–325.

[16] S. Lin and D. Costello, *Error Control Coding: Fundamentals and Applications*. New Jersey, USA: Prentice Hall, 1983.

[17] J. L. Massey, "Shift register synthesis and bch decoding," *IEEE Transactions on Information Theory*, vol. 15, pp. 122–127, Jan. 1969.

[18] P. Fitzpatrick and S. Jenning, "Comparison of two algorithms for decoding alternant codes," *Applicable Algebra In Engineering, Communication and Computing*, vol. 9, 3, pp. 211–220, 1998.

[19] E. Popovici, "Algorithms and architectures for decoding reed-solomon and hermitian codes," Ph.D. dissertation, University College Cork, University College Cork, Cork, Ireland, 2002.

[20] G. D. Forney, "On decoding bch codes," *IEEE Transactions on Information Theory*, vol. 11, pp. 393–403, Oct. 1965.

[21] S. L. Howard, C. Schlegel, and K. Iniewski, "Error control coding in low-power wireless sensor networks: When is ecc energy-efficient?" *EURASIP Journal on Wireless Communications and Networking*, vol. 2006, 2, pp. 1–14, 2006.

Multi-level Security in Wireless Sensor Networks

Faruk Bagci and Theo Ungerer
Institute of Computer Science
University of Augsburg
Augsburg, Germany
{*Bagci, Ungerer*}*@Informatik.Uni-Augsburg.DE*

Nader Bagherzadeh
Department of Electrical Engineering and Computer Science
University of California
Irvine, USA
nader@uci.edu

***Abstract*—As the potential range of applications for sensor networks expands, the need for security mechanisms grows. Security relevant problems are limited mostly to areas such as key distribution and cryptographic algorithms, due to severe resource constraints in wireless sensor networks. Even if it is not possible to cover all threats, a security architecture for sensor networks should provide mechanisms on several levels, in order to maximize the obstacles for attackers. This paper presents *SecSens*, an architecture that provides basic security components for wireless sensor networks on multiple system levels. Since robust and strong security features require powerful nodes, SecSens uses a heterogeneous sensor network. In addition to a large number of simple (cheap) sensor nodes providing the actual sensor tasks, there are a few powerful nodes (cluster nodes) that implement the required security features. The basic component of SecSens offers authenticated broadcasts to allow recipients to authenticate the sender of a message. On the basis of this basic component, SecSens provides a key management, used for exchange of secret keys among nodes. In order to bind nodes to their neighborhood, keys and their owners are linked together. This ensures a certain grade of location relationship. To protect the Sensor network against routing attacks, SecSens includes a probabilistic multi-path routing protocol, which supports the key management and the authenticated broadcasts. SecSens also provides functions to detect forged sensor data by verifying data reports en-route. In order to evaluate the efficiency of our security architecture, we simulated different sizes of sensor networks. Furthermore, SecSens is successfully evaluated in a real test environment with two different kinds of sensor boards.**

***Keywords*-wireless sensor network; security architecture; key management; energy efficiency; multi-path routing; en-route filtering**

I. INTRODUCTION

Wireless Sensor Networks (WSN) have emerged as a new information-gathering paradigm based on the collaborative efforts of a large number of self-organized sensing nodes. These networks form the basis for many types of smart environments such as smart hospitals, intelligent battlefields, earthquake response systems, and learning environments. A set of applications, such as biomedicine, hazardous environment exploration, environmental monitoring, military tracking and reconnaissance surveillance, are the key motivations for the recent research efforts in this area [7] [9].

Different from traditional networks, sensor networks do impose a set of new limitations for the protocols designed for this type of networks [10]. Devices in sensor networks have a much smaller memory, constrained energy supply, less process and communication bandwidth. Topologies of the sensor networks are constantly changing due to a high node failure rate, occasional shutdown and abrupt communication interferences. Because of the nature of the applications supported, sensor networks need to be densely deployed and have anywhere from hundred to thousands of sensing devices, which are orders of magnitude larger than traditional ad hoc mobile networks. In addition, energy conservation becomes the center of focus because of the limited battery capacity and the difficulty of recharge in the hostile environment. With fundamental difference between traditional networks and sensor networks, it is not appropriate and probably inefficient to port previous solutions for ad hoc networks into sensor networks with only incremental modifications. For instance, the sheer number of sensor nodes makes flooding-based standard routing schemes for ad hoc networks undesirable [4].

Because of the steady increase in applications, security requirements for sensor networks have received more attention. Areas such as health or safety critical industrial facilities offer very good use for sensor networks, on the other hand, they also demand high safety standards to be observed. A security architecture can never cover all types of threats simultaneously. The application determines, which attack vectors are probable in current scenarios, and how attractive collected and processed data could be for a potential intruder. In particular, denial-of-service attacks at network level, require special and expensive countermeasures. A comprehensive security architecture can increase protection and number of blocked attacks, but on the other hand, hardware costs and thus cost per sensor board increases, which is not always desirable. It also may increase energy requirements of sensor nodes significantly due to several successive protocols. This extra effort can, however, reduce considerably the life-time of individual sensor boards.

Not every sensor network pays an attack with enormous resources required to access its data, or to block it. In most cases, a combination of multiple protocols can confine

a wide range of threats. Therefore a good compromise between cost and protection is often not a full defense against all attacks, instead it is preferable to maximize the obstacles for attackers. Many attackers resile if they have to increase extremely costs to break a security architecture.

Regarding sensor networks, following basic requirements exist for the security concept:

- **Confidentiality:** Confidentiality means that information remain secret to unauthorized parties. Therefore sensor nodes must protect transferred data against illegal access.
- **Authenticity:** A node must always verify the authenticity of received messages. In particular, authenticity ensures that messages are really sent by the stated source.
- **Integrity:** Received data can be changed during transmission by failures or intent. Integrity should recognize these manipulations.
- **Timeliness:** The timeliness of data ensures that received information is up-to-date. An attacker should not be able to send old data (repeatedly).
- **Scalability:** Especially key management in large sensor networks can cause significant burden. Therefore the security architecture should consider also scalability.
- **Availability:** The sensor network should be robust and fault-tolerant, *i.e.* compromising individual node should not affect security of the entire network. On the other hand, the effort for security must not impair actual tasks of the sensor network, *e.g.*, by long delays.
- **Compromise:** A complete protection can not be ensured by any security architecture. Hence it must take the worst-case scenario into account, in which parts of the network are compromised. The aim should be a verification of data in order to prevent insider attacks, or at least to limit them locally.
- **Self-organization:** Characteristic for sensor networks is their capacity for self-organization. This should be considered by the security concept, especially in relation to key management and join/loss of nodes in the network.
- **Accessibility:** In order to decrease transfer costs, intermediate nodes on a path to the base station should aggregate and process received sensor data (*in-network-processing/ aggregation*). This implies, however, suitable access to secured data. By contrast, access must be minimized as far as possible to limit possible impact of compromised nodes.
- **Energy-efficiency:** To provide a certain life-time and to prevent attacks on depletion of energy resources, energy-efficiency plays an important role in security architecture that should be taken into account.

This paper describes *SecSens* a security architecture for wireless sensor networks that fulfills above mentioned requirements [1]. SecSens focuses mainly on a robust and secure routing protocol and protection against data manipulation. The next section describes related security approaches for sensor networks. Section III introduces the SecSens architecture and its security features. SecSens is successfully evaluated in a simulator and a real test environment with two different kinds of sensor boards. Section IV describes the evaluation results. The paper ends with the conclusion.

II. Related Work

SecSens focuses basically on three security aspects: key management, secure routing, and verification of sensor data. This section describes related approaches that are relevant for each aspect and can be used in wireless sensor networks.

Critical factor in key management is secure and efficient distribution of keys to sensor nodes. Because of limited resources in sensor networks usually symmetric keys are used. Symmetric encryption requires that both communicating nodes know the same secret key.

Key Management: [2] presents secure key distribution techniques for sensor networks. In particular, two approaches are described: single network-wide key and pair-wise shared key. The simplest method of key distribution is to pre-load a single network-wide key onto all nodes before deployment. Storage cost is minimal because each node has to store only one key. Unfortunately this approach provides sufficient security only if all nodes are protected against physical force. But this usually does not apply to low cost sensor nodes. The pair-wise shared key approach requires that every node in the sensor network shares a unique symmetric key with every other node. Hence, in a network of n nodes there are a total of $\binom{n}{2}$ unique keys, whereas each node has to store $n-1$ keys. The storage cost is proportional to the total number of nodes in the network. Therefore, the pairwise key scheme does not scale well for large sensor networks.

In [8] a security protocol for sensor networks called *SPINS* was presented for hierarchical sensor networks with one or more trustworthy base stations. SPINS consists of two parts: a secure network encryption protocol (SNEP) and authenticated broadcasts (μTESLA). Each sensor node receives on a secure channel an individual, symmetric master key, which is only known by the base station and the node. Using this master key the sensor node is able to generate all keys. The disadvantage of SNEP is that secure communication can be built only between a base station and nodes, and it is not possible to protect the communication in or between clusters. The second part of SPINS is μTESLA that provides sending of authenticated broadcasts. For symmetric encryption, sender and receiver must share the same secret. Consequently, a compromised receiver is able to act as a designated sender by transferring forged messages to all receivers. μTESLA uses delayed disclosure of symmetric keys for generating an asymmetry between sender and receiver. This approach requires weak

time synchronization of sender and receiver in order to achieve time shifted key disclosure. Storage cost increases because each node has to buffer packets which it can only verify after receiving the key in the future time-slots. Also, this causes new possibilities for DoS-attacks. An attacker can force a buffer overflow by sending planned broadcasts. Furthermore μTESLA leads to scalability problems, which are described in [6].

An important aspect for sensor networks is that different communication patterns exist requiring different security steps. [13] suggests an adjusted key distribution for different security requirements. For this reason, four different kinds of keys are used. The individual key is similar to the master key of SPINS. The second kind of key is a pair-wise shared key, which is generated in the initial phase for each known neighbor. Furthermore, the nodes have a cluster key for secure communication between cluster members. The last key is the group key that is used for secure broadcasting. This approach provides more flexibility but contains a security risk during the initial key distribution phase.

Secure Routing: Compromised sensor nodes can influence a sensor network, especially by manipulating of routing information. In order to minimize the impact, [3] suggests intrusion tolerant routing in wireless sensor networks called *INSENS*. The goal is to provide a working network even if parts of it are infiltrated. INSENS contains two phases: route discovery and data forwarding. In the first phase, the base station sends out a broadcast to build routes to each node. After receiving the route request, the nodes send a list of all known neighbors back to the base station. For the last step, the base station generates several disjoint paths to each node and sends this routing information back to all network members. Based on this routing table, the nodes can forward data to the base station (data forwarding phase). INSENS prevents the network against most outsider attacks and even insider attacks remain locally. But the dependance on the base station suggests a single point of failure. Furthermore, the route discovery phase is extremely energy inefficient.

Another approach for secure routing called *ARRIVE* is presented in [5]. The routing algorithm tries to send packets over different paths based on probability. Also, nodes in ARRIVE listen passively to communication among neighbors. In case of detected failures, other nodes can forward the packet on behalf of its neighbor. ARRIVE works with smaller routing tables, but the chosen path is not always optimal. Furthermore, ARRIVE does not provide authenticated broadcasts, that provides a mechanism for manipulation of routing information.

Verification of Sensor Data: Each individual sensor node is potential target for attackers. Using compromised nodes, an attacker can directly influence the sensor network by infiltrating false reports of network sensor data. This kind of attack is called fabrication report attack. These fake reports can reach the base station, if they remain undetected, where they can trigger off false alarms. Also this causes high consumption of bandwidth and energy. En-route filtering attempts to verify reports on the way from sending node to the base station. The goal is to detect and discard false reports earlier. [14] describes an interleaved hop-by-hop authentication scheme for filtering of injected false data. This algorithm recognizes infiltrated reports by a deterministic process, as long as no more than t nodes are compromised. The sensors build clusters with a minimum of $t+1$ nodes, where each group chooses a cluster head C. Only the cluster heads can send collected events in the form of reports to the base station. These reports include additionally to the actual event $t + 1$ independent confirmations of the respective parties, in order to verify the authenticity of the report. Each intermediate node checks the report on the way to the base station (*En-route filtering*). Unfortunately, this approach uses single path routing to the base station providing several security risks. A statistical en-route filtering is presented in [12] that enhances the approach above by using probabilistic algorithms. Each node chooses randomly a number of keys from a partition of a global key pool. Because of the probabilistic distribution of keys, any node can verify with a certain probability a report before it is forwarded. In this manner, [12] supports multi-path routing. But in both approaches, an attacker can create any report, once it has compromised at least t nodes. [11] attempts to solve this problem by binding keys to the location of nodes. The sensor area is divided into cells of width c, whereas each cell contains several keys. The nodes receive a location-bound key for each *sensing cell*. This approach bounds reports to their original location.

III. SecSens - Security Architecture for Wireless Sensor Networks

The sensor network in SecSens consists of clusters, each containing simple sensor nodes u_i and one powerful sensor node v that acts as a cluster-head. Sensor nodes u_i connect directly to the cluster-head, because routing in clusters is not necessary. Sensor nodes can be a member of several clusters. Cluster-heads again build together an inter-cluster network, that is used to transfer messages to base stations. It is assumed that sensor nodes have a fixed position, once they are attached to a location. SecSens works with multiple base stations to avoid the risk of single-point-of-failure (see Figure 1).

The security architecture of SecSens combines several security approaches in order to provide high protection. Basically SecSens contains four components, which interact with each other: authenticated broadcasts, key management, routing, and en-route filtering.

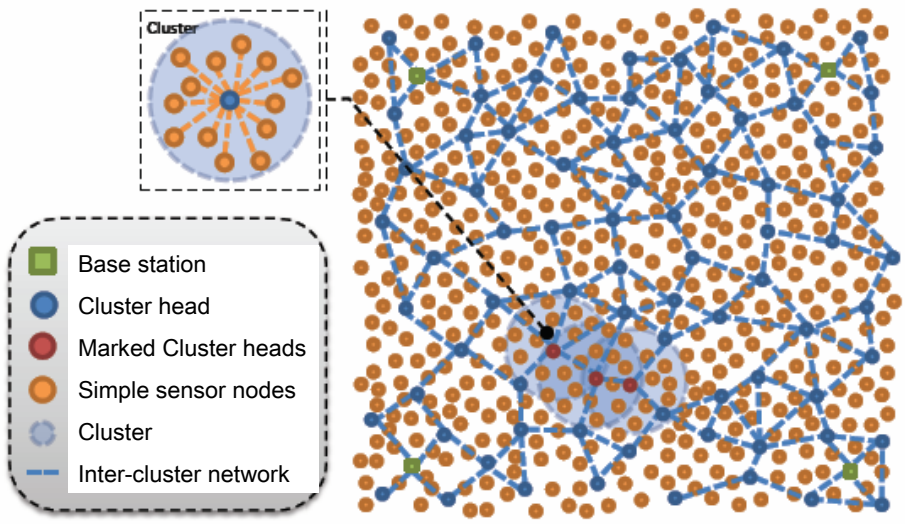


Figure 1. Basic sensor network architecture

A. Authenticated Broadcasts

Authenticated broadcasts ensure that the stated sender is identified as the true sender. Symmetric approaches use a shared key to generate a *message authentication code (MAC)*. In case of only one receiver, the sender is clearly identified. But if there are multiple receivers with the same shared key, this approach for authentication is not applicable. Potentially each receiver could be the sender. To solve this problem, there must exist an asymmetry between sender and receiver.

SecSens provides two authenticated broadcasts: broadcasts from base stations, and broadcasts in clusters. It is assumed that base stations are trustworthy and can not be infiltrated. In order to generate an asymmetry, SecSens uses key chains. Each packet contains a key. To decrypt a previous packet, a node has to wait for the key of next packet (Figure 2).

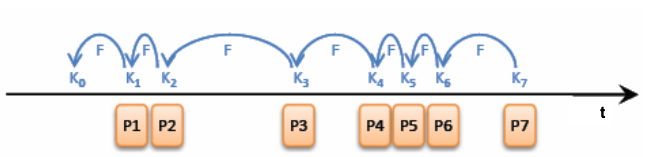


Figure 2. Key chain approach

The base station generates a key chain $K_0^b \dots K_n^b$ with sufficient number of keys using a publicly known one-way function F, so that for $i \in \{0, \dots, n-1\}$ is:

$$K_i^b = F(K_{i+1}^b) \tag{1}$$

Each node knows that the first key in the chain is $\langle i, K_i^b \rangle$ with $i = 0$. Therefore, K_0^b is public and K_1^b is the first non-disclosed key. Keys can be used only once. In order to broadcast a message M, the base station calculates the corresponding MAC using the next non-disclosed key K_i^b and sends it together with used key index to all its neighbors:

$$BS \rightarrow *: \qquad i, MAC_{K_i^b}(M) \tag{2}$$

A receiving node u checks, if it has already received a MAC for the stated index, before storing the MAC and index i into the MAC buffer. Consequently node u accepts only one MAC per key index. If i is a new index and u is a clusterhead, it forwards the message to all its neighbors. In this way, it is efficiently distributed over the inter-cluster network to all sensor nodes. After a maximum time T all sensor nodes know the MAC together with its key index. Sensor nodes can not manipulate the MAC, because the base station has not yet disclosed K_i^b at this time. Time T_p describes a dynamically adaptable system parameter. The base station can set T_p depending on the network size, whereas $T < T_p$. After expiration of T_p, the base station sends the actual message M besides the disclosed key data $\langle i, K_i^b \rangle$ to all neighbors:

$$BS \rightarrow *: \qquad M, \langle i, K_i^b \rangle \tag{3}$$

Sensor node u can now verify K_i^b using a previous disclosed key K_{i-1}^b. If it does not have K_{i-1}^b, it can verify the current key through recursively performing Equation 1 with previous keys. To prevent DoS-attacks, the new key must not be older than G_{max} generations. If K_i^b is finally

verified, node u makes new key data $\langle i, K_i^b \rangle$ effective. Subsequently, u can check message M using $MAC_{K_i^b}(M)$ and index i. It is important that the index for MAC and key are the same.

The concept of authenticated broadcasts for base stations has a disadvantage, if it is used for local authenticated broadcasts between clusters, because of the delayed disclosure of keys. However, all receivers of a message in a cluster can be reached after a single hop. This fact can be used to simplify the concept, in order to avoid time delays. The cluster-head generates a key chain $K_0^b \dots K_n^b$ using a publicly known function F (see Figure 2). All cluster members receive the first key K_0^b over a secure connection using pair-wise shared keys known only by cluster-head and respective node. Cluster-head v uses for each local authenticated broadcast a key K_i^b, which is not yet disclosed. It sends the message M together with the key data:

$$v \rightarrow * : \qquad M, K_i^b \tag{4}$$

Cluster nodes can verify K_i^b and the message M using previously disclosed keys. With this the sender is authenticated as cluster-head, because only the cluster-head can know K_i^b, that was not public until current message. On the other hand, no intermediate node can manipulate the message M, because all potential receivers of M are reachable with only one hop.

B. Key Management

Sensor nodes basically distrust each other. To build trust between two or more nodes, a shared secret in the form of keys is needed. However, neighborhood and relations between sensor nodes are not known before. Therefore, nodes must build trust during life-time, more strictly in the initial phase. For security reasons, a sensor node should join a network only once. In this critical phase, it can establish shared keys with its neighbors. Since this procedure is performed only once, the node is binding itself to the location. Furthermore, sensor nodes can communicate at several levels, that is cluster- or network-wide. Consequently, SecSens uses several kinds of keys to fulfill different security requirements.

Activation: In the initial phase, a sensor node needs an initial key K_I. This key is stored only on a specific activation node, which does not take part for usual networking tasks. The basic idea is that sensor nodes can not install themselves independently. Instead a trustworthy employee, who own the activation node, establishes sensors. In order to add a set of sensor nodes, the base station stores a randomly generated master key K_A, timer T_A, initial key K_I, and current group key K^g onto the activation node A. Using master key K_A, the base station can generate for each sensor node u a personal activation key K_u^a based on the node ID.

$$K_u^a = F_{K_A}(u) \tag{5}$$

All key material and other security critical data is stored only in RAM of A. T_A determines period of validity for the master key. After expiration of this time, the activation node deletes K_A and all critical data. Each sensor node u has a unique ID and a personal activation key K_u^a. In order to activate sensor node u, the activation node A has to be in the communication range. For security reasons, the radio power of A is kept low, to ensure physical proximity. After turning on for the first time, sensor node u broadcasts periodically its plain ID and the ID encrypted with the personal key with the same low radio power. Activation node A can easily verify the ID, because it knows the master key K_A. As the next step, A encrypts with personal key of u the initial key K_I, group key K^g, and data X that was given by base station. Finally, A sends encrypted message to sensor node u:

$$u \rightarrow A : \qquad u, MAC_{K_u^a}(u) \tag{6}$$

$$A \rightarrow u : \qquad \{K_I, K^g, X\}_{K_u^a} \tag{7}$$

After receiving all key material, sensor node u is activated and it deletes the personal activation key.

Group keys: The group key K^g is used by the base station to secure network-wide communication. An attacker, who compromises a node, can also access the group key. In order to update K^g, the base station broadcasts first a list of known compromised nodes $\{x_1, ..., x_m\}$ to all sensor nodes. Additionally, it sends a verification key $F_{K^{g\prime}}(0)$, whereas $K^{g\prime}$ is a new randomly generated group key, and F a publicly known one-way function. $F_{K^{g\prime}}(0)$ is used later to verify the new group key $K^{g\prime}$.

$$BS \rightarrow * : \qquad i, MAC_{K_i^b}(\{x_1, \dots, x_m\} \,||F_{K^{g\prime}}(0)) \tag{8}$$

$$BS \rightarrow * : \qquad \{x_1, \dots, x_m\}, F_{K^{g\prime}}(0), \langle i, K_i^b \rangle \tag{9}$$

The base station uses an authenticated broadcast with key K_i^b and index i that is not disclosed yet. After receiving key K_i^b and successfully verifying the above message (see section III-A), sensor nodes delete all pair-wise shared keys or cluster-keys with compromised node x_i. Cluster-heads additionally update their cluster-keys and inform other non-compromised cluster-heads about new cluster-key. Afterwards, all sensor nodes store verification key $F_{K^{g\prime}}(0)$. As a second step, the base station publishes new group key $K^{g\prime}$. Therefore, it encrypts the group key using its cluster-key K_{BS}^c and transfers the message to all direct neighbors. The neighbors can verify $K^{g\prime}$ using verification key $F_{K^{g\prime}}(0)$ and store afterwards the new group key.

$$BS \rightarrow * : \qquad \{K^{g\prime}\}_{K_{BS}^c} \tag{10}$$

If receiver u is a cluster-head, it forwards new group key $K^{g\prime}$ encrypted by its own cluster key K_u^c. Consequently, the new group key $K^{g\prime}$ is forwarded over the inter-cluster network to all sensor nodes. Since cluster-heads updated their cluster keys before, the compromised nodes do not receive the new group key. This procedure is periodically repeated by base stations to prevent the network against attacks. If there are no new known compromised nodes, the transferred list is empty.

Pair-wise shared keys: For secure communication between sensor nodes, pair-wise shared keys are used. A new cluster-head exchanges a pair-wise key with all neighbors. Simple sensor nodes communicate only over cluster-head, therefore, they need only a shared key with their cluster-head. Using the initial key K_I each node u generates a personal master key K_u^p based on its ID. In order to establish a pair-wise shared key with its neighbor v, node u needs the ID of v. For this reason u broadcasts a HELLO-message containing its ID. If v decides to establish secure connection with new node u, it answers with an acknowledgment containing its own ID.

$$u \rightarrow * : \quad u \tag{11}$$

$$v \rightarrow u : \quad v, MAC_{K_v^p}(u||v) \tag{12}$$

The additional MAC authenticates the acknowledgement of v, because u can calculate master key K_v^p of v using initial key K_I. Node u does not need to authenticate itself, because the succeeding message exchange verifies the identity of u. The pair-wise shared key K_{uv}^p can be calculated by both nodes without new message exchange:

$$K_{uv}^p = F_{K_v^p}(u) \tag{13}$$

After expiration of time T_I the nodes delete initial key K_I and all personal master keys of its neighbors received during initialization. Only own master key is stored for future pair-wise keys with new sensor nodes. The annulment of compromised pair-wise keys is efficiently realized by deletion of corresponding keys.

Cluster keys: Sensor nodes transfer information to all other cluster members using the cluster key without encrypting the message for each receiver separately. This approach allows in-network-processing and passive participation of sensor nodes within a cluster. Cluster-head u generates randomly cluster key K_u^c, if u joins a network or if u updates cluster key because of compromised nodes. Each cluster member $v_1, \ldots, v_m$ receives new cluster key K_u^c, whereas u encrypts cluster key using pair-wise shared keys $K_{uv_i}^p$ for $i \in \{1, \ldots, m\}$:

$$u \rightarrow v_i : \quad \{K_u^c\}_{K_{uv_i}^p} \tag{14}$$

Only sensor node v_i can decrypt cluster key and store it. If an additional sensor nodes v joins the network, it establishes a new pair-wise shared key with cluster-head. In this case it also gets the current cluster key. If a cluster member is compromised, cluster-head annuls cluster key K_u^c and distributes new key $K_u^{c\prime}$ as described above, without sending it to the compromised nodes.

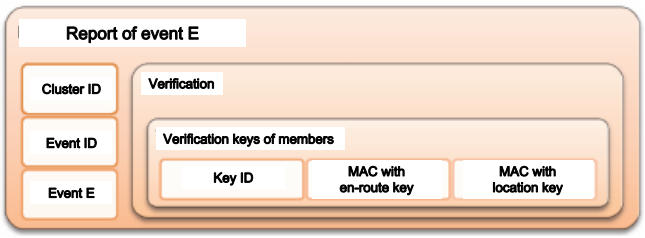


Figure 3. Report content

C. Routing

To prevent attacks on routing level or restrict them locally, SecSens provides a secure routing protocol. Simple sensors in SecSens do not need routing capability, because they exclusively communicate with the cluster-head. Therefore, routing is used only within the inter-cluster network built by cluster-heads. The routing algorithm has two phases: initialization and the actual routing. In the initialization phase each node gets a level using breadth first search that determines the distance to base station in hops. Since base station has level 0, its direct neighbors have level 1. The base station uses an authenticated broadcast including its ID BS_u, a non-disclosed key K_i^b, and key index i to authenticate an initialization.

$$BS \rightarrow * : \quad i, MAC_{K_i^b}(BS_u) \tag{15}$$

$$BS \rightarrow * : \quad BS_u, \langle i, K_i^b \rangle \tag{16}$$

Cluster-heads can identify from authenticated messages, which base station wants to update routing information. After reception of initialization, cluster-heads have time T to modify their routing tables. After expiration of T, further changes are not allowed. Cluster-heads set their level on $L = \infty$ after reception of the message (see Equation 16). The breadth first search can now begin. Starting from the base station, the level values are locally broadcasted by cluster-heads. To prevent outsider-attacks each cluster-head u uses key K_i^b, that will be published later, and its cluster key K_u^c to generate an encrypted message containing ID and level value L_u:

$$u \rightarrow * : \quad \{u||L_u\}_{K_u^c}, K_i^b \tag{17}$$

A cluster-head v updates its level to $L_v = L_u + 1$, if $L_v > L_u + 1$. It also stores level of u. After level update, v forwards its level value to its neighbors in the same way. Each base station triggers its own initialization without disturbing ongoing updates of others. The cluster-heads manage a routing table for each base station.

SecSens uses probabilistic multi-path routing based on the level values to forward messages from cluster-heads on the way to the corresponding base station. Cluster-heads build up a trust matrix, where each transmission to its neighbors is recorded. Based on this trust information and current level, cluster-heads calculate a probability value and write it into the packet header. This value is used to decide in which direction the packet has to be send. Each cluster-head modifies the probability value and sends the message over the most trustworthy route. Since this could lead to the problem that a packet stays at the same level while making a round-trip, the weight of upper level increases with each hop. This ensures that packet transfer goes in the direction of the base station.

Furthermore, SecSens provides passive participation, *i.e.* sensor nodes listen to packet transmissions of their neighbors. If cluster-head u detects a packet addressed to its neighbor v, and recognizes that v is not forwarding the message, u takes responsibility with a certain (low) probability. Also, if u assumes that v forwards the message to a non-existent node, u takes care of transferring.

D. En-route Filtering

Attacks like *report fabrication* or *false data injection* threaten the network by manipulating and infiltrating sensor data. SecSens prevents such threats using en-route filtering extending approach of [12]. Cluster-heads generate data reports containing sensor information of cluster members for sending them to base stations. These reports are verified during transfer through the inter-cluster network (see Figure 3).

En-route filtering consists of three phases: key generation, report generation, and verification.

Key generation: SecSens provides a global pool containing N en-route keys $\left\{K^e_0, \ldots, K^e_{N-1}\right\}$. The keys K^e_i are subdivided in n partitions with each m keys. Each sensor node generates all en-route keys in the initial phase using one-way function F and chooses randomly a partition j where it finally draws $k < m$ keys from set j:

$$\forall i \in \{0, \ldots, N-1\} : K^e_i = F_{K^e_M}(i) \qquad (18)$$

At least base stations can detect all fault reports, because they have global view of the key pool. Based on the same partition j, each node calculates in a similar way location key $K^l_{C,j}$ for all its clusters that it senses as a member. Sensor nodes bind themselves locally to actual cluster by the location key. After the initial phase nodes delete all remaining unused keys.

Report generation: If a cluster-head wants to generate a report, it collects sensor data from all its cluster members. Sensor nodes belonging to same cluster report events (sensor data) collectively by generating MACs based on their en-route keys, whereas keys must be chosen from different partitions. These multiple MACs collectively act as the proof that a report is legitimate. Finally, the cluster-head forwards the report to the base station over the inter-cluster network.

Verification of reports: A cluster-head receiving a report checks, if it has one of the keys, that were used to generate the MACs in the report. With a certain probability, it will be able to verify the correctness of MACs. A report with an insufficient number of MACs will not be forwarded. A compromised node has keys from one partition and can generate MACs of one category only. Since keys and indices of distinct partitions must be present in a legitimate report, the compromised node has to forge the other key indices and corresponding MACs. This can be easily detected by cluster-heads possessing these keys. If cluster-head has none of the keys and number of MACs is correct, it forwards report to the next cluster-head. Even if a forged report receives a base station, it can be detected, because base stations know all used keys.

IV. Evaluation

To evaluate the efficiency of our security architecture we implemented a simulation tool where it is possible to establish different sizes of sensor networks. Figure 4 shows the GUI of the SecSens simulator. The aim was to measure energy consumption and throughput of security mechanisms. SecSens simulator provides the possibility to change parameters like network size, node density, and number of cluster heads or cluster members. After initializing the network we performed several attacks and examined the network stability.

Denial-of-Service attacks are the most common threats in a network. SecSens limits the effect of DoS attacks to a local area compensating high node failures. When a node failure is detected, SecSens tries to send the message over an alternative route. Figure 5 shows the delivery rate of messages in dependance to node failure varying the distance between failure location and receiver. We simulated here a sensor network with 10.000 nodes. If the center of failure is near to message receiver (5 hops), SecSens finds enough alternative routes to achieve %50 delivery rate considering number of 100 failed nodes. Increasing the distance means that the failure center comes near to the message sender blocking most alternative routes.

Insider attacks aim to manipulate behavior of the sensor network by infiltrating false sensor data. SecSens uses data reports to secure the delivery. In order to generate a new report, a node needs five verification keys from five different neighboring nodes. If an attacker wants to send a false report, he has to compromise several nodes to access the

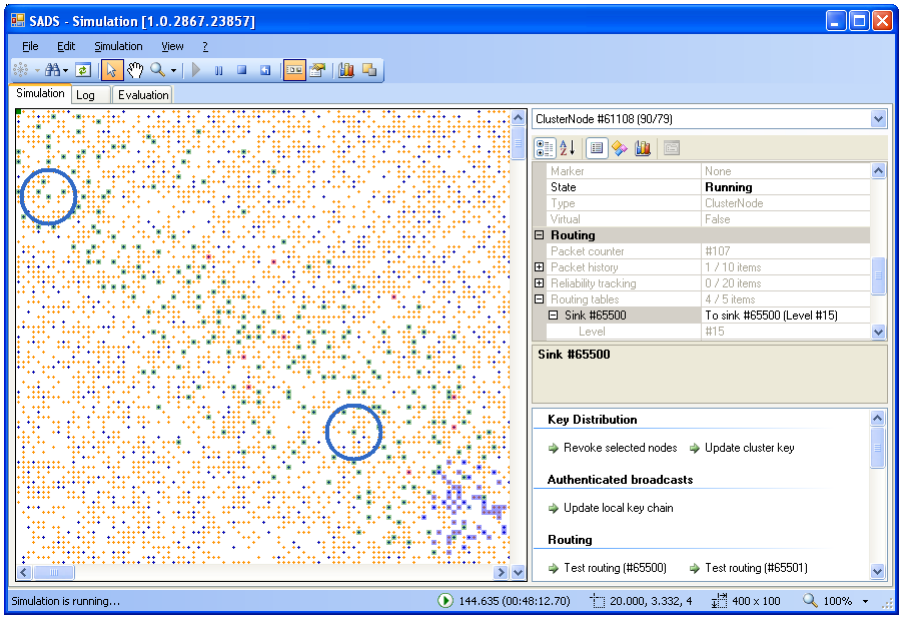


Figure 4. SecSens Simulation Tool

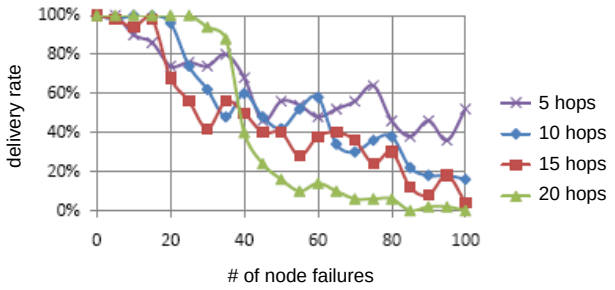


Figure 5. Message delivery rate after DoS attacks

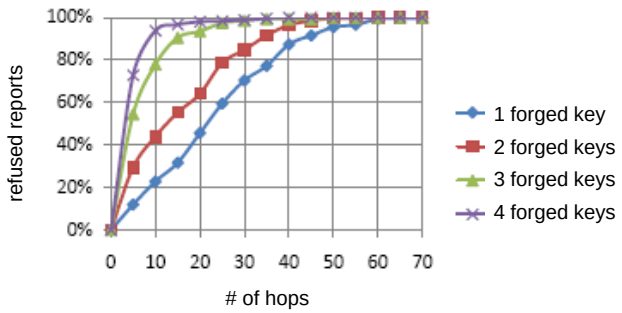


Figure 6. Rate of refused reports in reference to distance

verification keys (see previous section). Alternatively, he can forge missing keys. Figure 6 illustrates the rate of detected false reports in dependance of the distance between sender and receiver. As you can see in the results, the detection rate increases with more forged keys. Already after ten hops, SecSens can refuse %90 of false reports containing four forged keys.

The detection rate on intermediate nodes is directly dependent on the size N of the global key pool and number k of locally stored en-route keys (see previous section). With increasing number of en-route keys, the probability of an intermediate node to hold the same en-route key rises. Figure 7 shows the simulation results for using different numbers of en-route keys. As expected, the detection rate for false reports rises with increasing number of locally stored keys. On the other hand, the ratio k/N should not be too high, since a compromised node would irrevocably disclose a part of the global key pool.

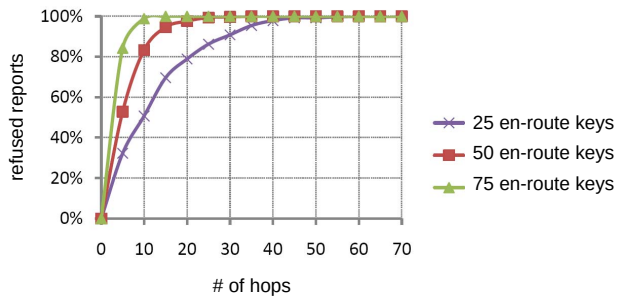
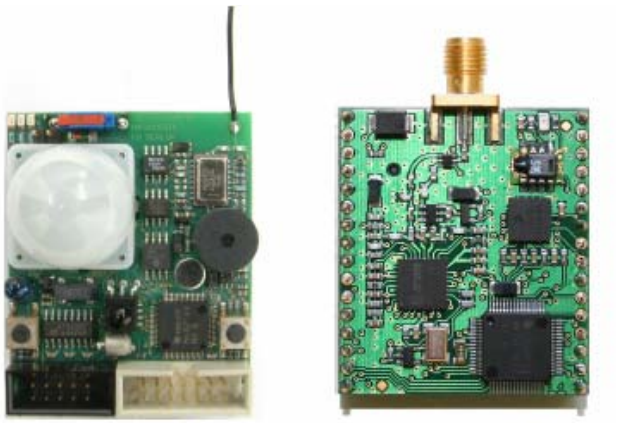


Figure 7. Influence of local stored en-route keys

In order to show the feasibility of SecSens on real environ-

ments, we established a testbed with different kinds of sensor nodes: ESB 430/1 and MSB-430 of Freie University Berlin (see Figure 8). Both sensor boards have the TI MSP430 microcontroller.



(a) ESB 430/1 (b) MSB-430

Figure 8. Sensor boards: ESB 430/1 and MSB-430

ESB 430/1 contains 60 KB flash memory and 2 KB RAM, whereas MSB-430 has 55 KB flash memory and 5 KB RAM. Because of the larger RAM, MSB-430 was used as cluster-head. We used four cluster-heads managing each four sensor nodes that were all ESB boards, making 20 nodes alltogether. Two PCs act as base stations. We could successfully perform all stages of SecSens which were described before.

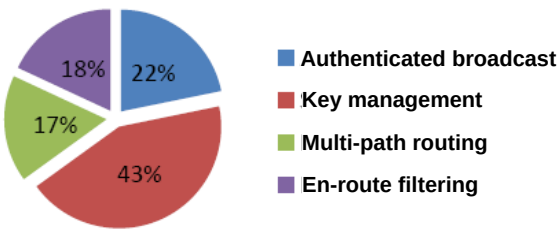


Figure 9. Memory map of SecSens security components

In order to use memory effectively, SecSens stores frequently changing data in RAM and relatively static data in EEPROM. Table I describes memory consumption of cluster-heads and sensor nodes. The most memory space is used for key management. Figure 9 shows the memory map of SecSens security components.

In the initial phase, energy consumption is comparatively high. For establishing the network and distribution of keys, cluster-heads consume in average $2.6Ws$ energy, whereas sensor nodes need only $0.2Ws$ (see Table II).

Energy consumption for sending reports depends on the

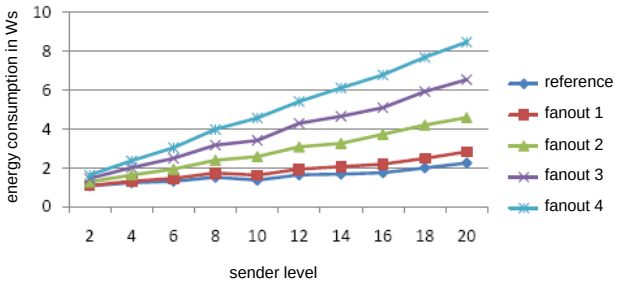


Figure 10. Consumed energy for reporting

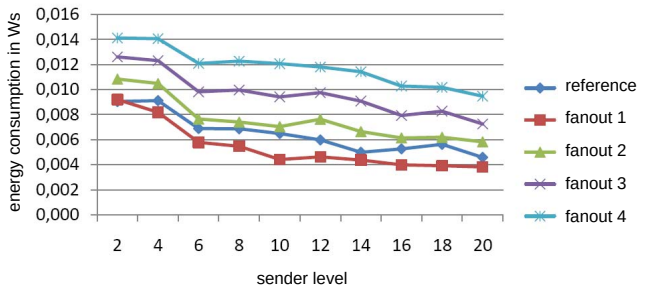


Figure 11. Energy consumption of single cluster head in reference to sender level

distance between cluster-head and the base station. We established in a second test a network with up to 20 cluster-heads placed in a line side by side. The last cluster-head received level 20, which means that it needs 20 hops to reach the base station. We measured the energy consumption for sending reports from different levels. The multi-path routing ensures a robust transmission, but sending duplicated

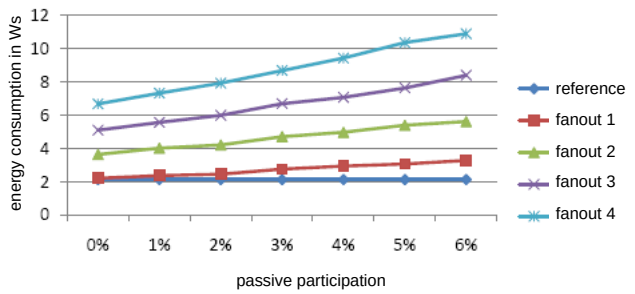


Figure 12. Energy consumption with passive participation

	packets	data	energy
sensor node	15	0,2 KB	0,2 Ws
cluster-head	142	4,1 KB	2,6 Ws

Table II
COSTS IN INITIAL PHASE

	AB	KM	MPR	EF	total
cluster-head: RAM	434 B	9 B	710 B	41 B	1194 B
cluster-head: EEPROM	800 B	2440 B	252 B	981 B	4473 B
sensor node: RAM	54 B	9 B	0 B	0 B	63 B
sensor node: EEPROM	0 B	92 B	0 B	537 B	629 B

Table I
SECSENS MEMORY REQUIREMENTS, (AB) - AUTHENTICATED BROADCASTS, (KM) - KEY MANAGEMENT, (MPR) - MULTI-PATH ROUTING, (EF) - EN-ROUTE FILTERING

packets from several routes (fanout) increases the energy consumption. Figure 10 shows consumed total energy in the sensor network for reports using no fanout (reference) or multiple fanouts. On figure 11 you can see the average energy consumption of a single cluster head in reference to the sender level. It is interesting to mention that energy consumption for sending four packets increases by only %200 instead of %400 as would be expected. The reason lies in the good balance of load in sending multi-path messages.

As mentioned above, nodes can listen to packet transmission of neighbors and can take the responsibility for forwarding with a certain probability in case of detected failures. Figure 12 describes the energy results for passive participation with different probabilities. Increasing passive participation naturally leads to higher energy consumption, because of the additional message delivery.

V. CONCLUSION

This paper presented *SecSens*, a multi-level security architecture for wireless sensor networks. SecSens combines several security approaches on different system levels in order to provide high protection. SecSens contains four components, which interact with each other: authenticated broadcasts, key management, routing, and en-route filtering. We implemented a simulation tool, where huge network sizes can be established. Evaluation results show that SecSens can resist DoS and insider attacks limiting failures to a local area. Infiltrated false sensor data can be refused with a high probability. We also demonstrated the feasibility of SecSens by building a real sensor network environment with two different kinds of sensor boards.

REFERENCES

[1] Faruk Bagci, Theo Ungerer, and Nader Bagherzadeh. SecSens - Security Architecture for Wireless Sensor Networks. In *The Third International Conference on Sensor Technologies and Applications (SENSORCOMM '09)*, Athens, Greece, June 2009.

[2] Haowen Chan, Adrian Perrig, and Dawn Song. Key distribution techniques for sensor networks. pages 277–303, 2004.

[3] Jing Deng, Richard Han, and Shivakant Mishra. Insens: Intrusion-tolerant routing for wireless sensor networks. *Computer Communications*, 29(2):216–230, 2006.

[4] P. Downey and R. Cardell-Oliver. Evaluating the Impact of Limited Resource on the Performance of Flooding in Wireless Sensor Networks. In *Proceedings of the 2004 international Conference on Dependable Systems and Networks*, Washington, DC, USA, June 2004.

[5] Chris Karlof, Yaping Li, and Joe Polastre. Arrive: Algorithm for robust routing in volatile environments. Technical Report UCB/CSD-03-1233, University of California at Berkeley, May 2002.

[6] Donggang Liu and Peng Ning. Multilevel μtesla: Broadcast authentication for distributed sensor networks. *Trans. on Embedded Computing Sys.*, 3(4):800–836, 2004.

[7] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, USA, September 2002.

[8] Adrian Perrig, Robert Szewczyk, J. D. Tygar, Victor Wen, and David E. Culler. Spins: Security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, 2002.

[9] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, 2000.

[10] R. Verdone and C. Buratti. Modelling for Wireless Sensor Network Protocol Design. In *International Workshop on Wireless Ad-hoc Networks (IWWAN 2005)*, London, United Kingdom, May 2005.

[11] Hao Yang, Fan Ye, Yuan Yuan, Songwu Lu, and William Arbaugh. Toward resilient security in wireless sensor networks. In *MobiHoc '05: Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing*, pages 34–45, New York, NY, USA, 2005. ACM Press.

[12] Fan Ye, Haiyun Luo, Songwu Lu, and Lixia Zhang. Statistical en-route filtering of injected false data in sensor networks. *IEEE Jounal on Selected Areas in Communications, Special Issue on Self-organizing Distributed Collaborative Sensor Networks*, 23(4):839–850, April 2005.

[13] Sencun Zhu, Sanjeev Setia, and Sushil Jajodia. Leap: efficient security mechanisms for large-scale distributed sensor networks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 62–72, New York, NY, USA, 2003. ACM Press.

[14] Sencun Zhu, Sanjeev Setia, Sushil Jajodia, and Peng Ning. An interleaved hop-by-hop authentication scheme for filtering of injected false data in sensor networks. *2004 IEEE Symposium on Security and Privacy*, 00:259–271, 2004.

Temporal Robustness of Real-Time Architectures Specified by Estimated WCETs

Lamine Bougueroua
LRIT
ESIGETEL
Avon, France
lamine.bougueroua@esigetel.fr

Laurent George
LACSC
ECE
Paris, France
lgeorge@ieee.org

Serge Midonnet
LIGM, UMR CNRS 8049
Université Paris-Est
Champs sur Marne, France
Serge.Midonnet@univ-paris-est.f

***Abstract*— Real-time dimensioning depends on the Worst Case Execution Time (WCET) of its tasks. Using estimated WCETs for the dimensioning is less conservative but execution overruns are more likely to happen. Fault tolerant mechanisms must be implemented to preserve the real-time system from timing failures, associated to late task termination deadlines misses, in the case of WCETs overruns. We show in this article how to compute the extra duration (allowance) on the WCETs that can be given to faulty tasks while still preserving all the deadline constraints of the tasks. This allowance is used on-line to tolerate WCET overruns. We present a mechanism called the Latest Execution Time (LET) using the allowance of the tasks for the temporal robustness of real-time systems. This mechanism only requires classical timers. Its benefits are presented in the context of a java virtual machine meeting the Real-Time Specification for Java (RTSJ) with estimated WCETs.**

Real-time System; fault tolerance; estimated WCET; allowance; slack time; temporal robustness

I. INTRODUCTION

Real-time scheduling theory can be used at the design stage for checking the timing constraints of a real-time system, whenever a model of its software architecture is available. In specific cases, standard real-time scheduling analysis techniques can significantly shorten the development cycle and reduce the time to market. The correct dimensioning of a real-time system depends on the determination of the Worst Case Execution Time (WCET) of the tasks.

Based on the WCET, a feasibility condition (e.g. [2][3][4]) can be established to ensure that the deadlines of all the tasks are always met, whatever their release times. The computation of the WCET can be performed either by analyzing the code on a given architecture or by measurement of the execution duration [5]. In both cases, the correctness of the WCET is hard to guarantee. The WCET depends on the condition of execution; the type of architecture, memory or cache. This can lead to an imprecise WCET. Also a very complex analysis may be necessary to obtain it. The obtained WCET can therefore be either pessimistic or optimistic compared to the exact execution duration obtained at run time. In the first case, we have reserved CPU resources that may not be used, leading to a pessimistic dimensioning of the system but the deadlines can always be guaranteed. In the second case we might have execution overruns, i.e. task durations exceeding their WCET but more CPU resources left to deal with such a situation. We are interested in this paper in the second case observing that an execution overrun does not necessarily lead to a deadline miss. With enough free CPU resources, a system can self stabilize and still meet the deadlines of all the tasks. So the problem consists of determining how much time may be allocated to the execution overrun.

In this paper, we consider that the WCETs are estimated by benchmark on a given architecture. The lack of precision on the WCETs is taken into account in this paper with the concept of allowance introduced in section 3.

We consider two types of faults captured by two errors: the *ExecutionOverrun* error and the *DeadlineMiss* error. In the first case, the task execution duration exceeds its WCET while in the second case the task does not meet its deadline. Yet, among all possible execution overruns, only those leading to a deadline miss, if nothing is done to correct the system, should be considered. We therefore need to determine how long we can let a faulty task proceed with its execution without compromising the deadlines of all other tasks. We call this duration the allowance of the task. Based on the allowance, we can determine when an execution overrun error should be raised.

In this paper, we consider a set $\tau = \{\tau_1, \tau_2,..., \tau_n\}$ of n sporadic or periodic tasks with timeliness constraints.

A task τ_i, is defined by:

- C_i, the estimated Worst Case Execution Time
- T_i, the inter-arrival time also called the period
- D_i, the relative deadline (a task released at time t must be executed by its absolute deadline $t + D_i$. We consider in this paper, that for any task τ_i , $D_i \leq T_i$
- P_i, the priority

In this paper, we first present a short description of related work (see Section 2). We then focus on the allowance of the WCETs in Section 3 when tasks are scheduled with Fixed Priority/highest priority first (FP/HPF) scheduling (e.g. [3][4][6]). We then show how to use the allowance on-line, with a mechanism called the Latest Execution Time (LET), in Section 4. The LET analysis was first introduced by Bougueroua and al. in [7]. Moreover [1], extends this analysis further by considering the EDF scheduling. In Section 5 we give results of some simulations, obtained from a tool we developed, so as to compare the different execution overrun management mechanisms. We do this firstly for a task deadline miss, and secondly for a task execution overrun, showing the benefits of the LET mechanism. Finally, we give some conclusions.

II. RELATED WORK

Most fault tolerant real-time systems present solutions to deal with deadline miss by stopping the execution of the tasks that miss their deadlines. In the case of overloaded systems, the idea is to stop some tasks so as to prevent the others from missing their deadlines and to come back to a normal load condition. Tasks are scheduled according to their importance: Locke's Best Effort Scheduling Algorithm in [8], D-Over in [9] and Robust EDF in [10]. The problem with this solution is that a task missing its deadline might already have had cascading effects on the other tasks of the system. The reaction might be too late.

In fact, a task which does not respect its deadlines might result in unacceptable delays on other lower priority tasks.

Several approaches have been considered in the state of the art: either in the dimensioning phase or on-line.

A. In a dimensioning phase:

In a dimensioning phase, a sensitivity analysis can be used to compute the maximum acceptable deviations on the WCETs, Period on Deadlines preserving the schedulability of the tasks (e.g. [11][12][13]). Most of the existing solutions to the sensitivity problem consider only one parameter can change. We place ourselves in this context for this article by considering the WCETs overrun. The reader interested by multidimensional analyses of sensitivity will be able to refer to work of Racu in [14] and [15], where a stochastic approach is proposed to deal with the variations of several parameters.

Bini in [13] shows how to calculate the maximum value of the multiplicative factor λ, applied to the set of tasks. In this case when scheduling FP of *n* periodic tasks, if for any task τ_i, $D_i \leq T_i$, then for any task τ_i, its WCET becomes $C_i + \lambda C_i$.

λ is the maximum value so that the set of tasks is schedulable. If for a set of tasks given, $\lambda < 0$ then initial set of tasks is not schedulable and it is necessary to reduce the WCET of λC_i so that the tasks set become schedulable.

The computation complexity of λ is pseudo-polynomial. The authors in [13] Show also how to calculate λ for a subset of tasks.

However, the use of a multiplicative factor λC_i for any task τ_i, gives more allowance to the task τ_i when its C_i is large, a choice which inevitably does not reflect the importance of the task.

In [16], the authors show how to determine for a scheduling FP, in the case $D_i \leq T_i$, the feasibility domain to *n* dimensions of WCETs (called C-space). They show that when the number of tasks is reasonable, it is possible to express the parametric equation of the WCETs feasibility domain in the form of a set of inequations to be tested. This approach is useful in a phase of dimensioning but from its polynomial pseudo complexity (the number of inequation is pseudo polynomial), is not applicable on line to determine if it is possible to continue the execution of a task exceeding its WCET.

Slack time analysis has been extensively investigated in real-time systems in which aperiodic (or sporadic) tasks are jointly scheduled with periodic tasks (e.g. [17][18][19]). In these systems, the purpose of slack time analysis is to improve the response time of aperiodic tasks or to increase their acceptance ratio. Yet, those approaches require high time overheads to determine the available slack time at a given time. Some research has been proposed to approximate the slack time (e.g. [19] and [20]). Nevertheless, the slack is computed for a single aperiodic task occurrence whereas the allowance is valid for all the requests of a periodic task. Computing the slack time for every periodic request would be time consuming.

B. On-line:

Some solutions consist of adapting the task parameters to the context of execution (e.g. approaches based on the variation of the periods (e.g. [21][22][23]) to obtain more or less precision). In the context of a network transmission, some transmission failures have been considered with the (m,k)-firm model [24] to tolerate m transmission failures among k.

This algorithm is classified as a best effort algorithm. In [25], the authors propose an extension of the model (m,k)-firm called Weakly-hard to consider non-consecutive failures.

The allowance on the WCETs proposed in Section 3 is computed with a sensitivity analysis with a sliding windows fault model. Yet, applying an identical scaling factor to a subset of tasks leads to providing more allowance to the tasks having higher WCETs. We remove this constraint by analyzing several allowance sharing strategies in Section 4. We then propose a fault prevention mechanism to prevent an execution overrun error from leading to a deadline miss error. We present the concept of Latest Execution Time a task can proceed with its execution without compromising the real-time constraints of all the tasks. The use of estimated WCET for the dimensioning of a real-time system enables us to be less conservative. However, it is necessary to take into account the faults that result in WCETs overruns and to guarantee the temporal robustness of the system in the case of execution overrun faults. The temporal robustness in our context is defined as follows: a faulty task should not have any influence on the other correct tasks. In our context, this means that an execution overrun of a task should not lead to a deadline miss of any other correct task.

III. ALLOWANCE ON WCETS – PRINCIPLES

The *processor utilization* corresponding to a task set τ is defined in [2]: $U = \sum_{n=1}^{n} \frac{C_i}{T_i}$. From this definition, $U \times 100$ represents the percentage of processor utilization. A necessary condition for the feasibility of a task set is: $U \leq 1$.

We are considering a real-time system based on the preemptive Fixed Priority, highest priority first (FP) scheduling algorithm with an arbitrary priority assignment. Preemptive scheduling means that the processing of any task can be interrupted by a higher priority task.

We define for any task τ_i scheduled with FP:

- hp(i) - the set of tasks except τ_i having a priority higher than or equal to τ_i except τ_i;

- lp(i) - the set of tasks having a lower priority than τ_i;
- hpR(i) and lpR(i) which denotes at any time the tasks released, respectively in hp(i) and lp(i).

The allowance consists of computing the allowance in the WCET tolerated by the system. It represents the maximum execution time that can be added to the WCET of a task without compromising the deadlines of all the tasks. We now give a more formal definition of the task allowance.

Definition 1:

A temporal fault (WCET overrun) of a task τ_i is said to be isolated if it does not result in any deadline miss of the other tasks.

Definition 2:

The task allowance is the maximum available CPU resources allotted to a faulty task when it exceeds its WCET. In addition, it represents the maximum duration that can be added for the execution of a task without compromising the execution of other tasks.

A. *Identification of the available CPU Resources*

The identification of CPU resources consists in computing from the available system resources the maximum extra execution duration that can be given to faulty tasks without missing the deadline of any task. The computation of available resources must be carried out during the worst case scenario thus minimizing the allowance of the tasks. Considering timeliness constraints, the worst-case scenario for the respect of the task deadlines occurs for FP scheduling in the synchronous scenario, i.e. all tasks are at their maximum rate and released synchronously at time t=0 (e.g. [4]).

We now show that the available CPU resources are also minimized in the synchronous scenario.

Let $t_i^0 \geq 0$ be the first activation request time of the task τ_i in an arbitrary processor busy period. The duration of free CPU resources left available by the system at any time $t \geq 0$ is given by:

$$t - \sum_{i=1}^{n} \max\left(0, \left\lceil \frac{t - t_i^0}{T_i} \right\rceil\right) \times C_i$$

This duration is minimum for $\forall i \in [1, n], t_i^0 = 0$, corresponding to the synchronous scenario. In this paper, we are therefore interested only in the synchronous scenario for the computation of the allowance on WCETs as the synchronous scenario is a possible scenario. According to figure 1, representing the execution of the three tasks (the task parameters are defined in table 1). We would draw your attention to the availability of many free CPU intervals resulting from processor idle times.

Let us suppose that none of the three tasks have exceeded its execution duration. In this case, the average processor utilization of the available resources U_{free} is equal to:

$$U_{free} = 1 - \sum_{n=1}^{n} \frac{C_i}{T_i}$$

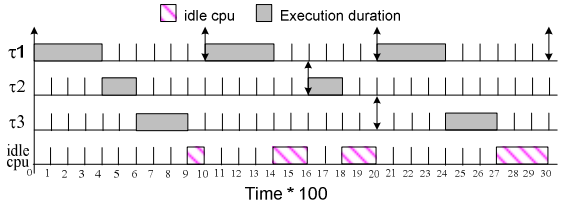


Figure 1. Available resources in the system.

TABLE 1. Average duration of free CPU resources

Task	C_i	D_i	T_i	Q_i	P_i
τ_1	400	1000	1000	325	High
τ_2	200	1600	1600	520	Medium
τ_3	300	2000	2000	650	Low

Those idle times could be used by faulty tasks. However, the amount of free resources, for each task, is not constant for each activation of the task. The average duration of free resource for a task τ_i is given by the following equation:

$$Q_i = U_{free} \times T_i$$

Table 1 gives the value of Q_i for all the three tasks τ_1, τ_2 and τ_3.

B. *Discussion - margin analyzes*

The use of the average duration of free CPU resources does not guarantee the respect of task deadlines. For example, in figure 2(synchronous scenario for the set of tasks given in table 1), during the first activation of task τ_2, the CPU resources which are really available for this task, in the event of a fault, are 300 units of time. This is less than the average quantity of available resource during future executions. (Q_2 = 520). The use of Q_2 leads to a temporal failure for task τ_3 activated at t=0.

We can use the total free resources during the lcm (least common multiple) of the periods of the tasks (also called the hyper period). The amount of available free resource during the hyper period is represented by the following equation:

$$Q_{hp} = U_{free} \times lcm(T_1, ..., T_n)$$

We obtain Q_{hp} = 2600 units of time in our example. However, this solution cannot guarantee the respect of the timeliness constraints of the tasks. In fact, task τ_1 as it has the greatest priority can use more CPU resources and this can result in deadline misses for lower priority tasks. Our goal is to use from amongst the available resources those which will not lead to deadline miss failures for all the tasks.

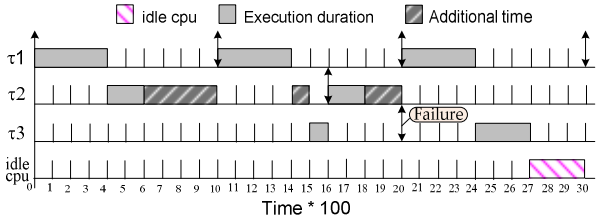


Figure 2. Available resources sharing

C. Allowance computation

The allowance on WCETs is related to the scheduling policy used by the system. In this article we are interested in FP scheduling with an arbitrary priority assignment. A classical feasibility condition for FP scheduling is obtained by calculating the worst-case response time R_i for any task τ_i. R_i is defined as being the longest duration time between release time and termination time [3]. A Necessary and Sufficient Condition (NSC) for the feasibility of Task Set Scheduled FP is then:

$$\forall i = 1..n,\ R_i \le D_i \text{ and } U \le 1$$

Many results showing the computation of the worst-case response time in a preemptive context are available [3][4]. In order to optimize the computation of the worst-case response time, [3] has proven that in the case of FP scheduling, when the deadlines are lower than or equal to the periods, the worst-case response time is obtained in the first activation of each task when all the tasks are released in the synchronous scenario.

Theorem 1: [3]

The worst-case response time R_i of a task τ_i of a non-concrete periodic, or sporadic, task set (with $D_i \le T_i$, $\forall i \in [1, n]$) is found in a scenario in which all tasks are at their maximum rate and released synchronously at a critical instant t=0. R_i is computed by the following recursive equation (where hp(i) denotes the set of tasks with higher priority than τ_i):

$$R_i^{m+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^m}{T_j} \right\rceil C_j$$

The recursion ends when $R_i^{m+1} = R_i^m = R_i$ and can be solved by successive iterations starting from $R_i^0 = C_i$.

We can easily show that R_i^m is not decreasing. Consequently, the series converges or exceeds D_i. In the latter case, task τ_i is not schedulable.

Remark:

A task is said to be non-concrete if its request time is not known in advance of its execution. In this paper, we only consider only non-concrete request times.

The computation of the allowance on WCETs can be carried out:

- In the dimensioning phase or on-line. Subsequently the tasks are activated by admission control, as long as the deadlines of the tasks can always be met. This is called the static approach.
- At each execution overrun detection. This is called the Dynamic approach.
- Once the static allowance is consumed then the dynamic computation is activated. This is called the Hybrid.

The computation of the allowance has a pseudo-polynomial time complexity (see property 1). Because of this we have adopted an approach based on a static computation of the allowance.

We consider in this paper a temporal fault model *[m/n]* corresponding to a fault model where there can be at most m≤n faulty tasks exceeding their WCET on a sliding window $W = min(T_1,\ldots, T_n)$. The allowance on the WCET of a task is related to the number of faulty tasks.

Firstly let us look at the case where m=1. The general case will be given in a later section. The allowance of a periodic or sporadic task τ_i, when only one task is faulty, called $A_{i,1}$, is obtained from the Necessary and Sufficient Condition (NSC) established in theorem 1.

Lemma 1:

The maximum allowance that can be given to the periodic or sporadic task τ_i, under the fault model [1/n], is equal to the maximum duration that can be added to the WCET of the tasks τ_i without missing the deadline of any task.

$$(1)\quad R_i^{n+1} = C_i + A_{i,1} + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \le D_i$$

$$\forall \tau_j \in lp(i):$$

$$(2)\quad R_j^{n+1} = C_j + \sum_{\tau_k \in hp(j), k \ne i} \left\lceil \frac{R_j^n}{T_k} \right\rceil C_k + \left\lceil \frac{R_j^n}{T_i} \right\rceil (C_i + A_{i,1}) \le D_j$$

$$(3)\quad U + \frac{A_{i,1}}{T_i} \le 1$$

Proof:

Let τ_i be the faulty task. By assumption, the maximum execution duration of τ_i is then $C_i + A_i$. Applying theorem 1 with the new execution duration of τ_i we find the formula (1) of lemma 1. Formula (2) is similar to formula (1) but relates to the tasks with lower priority than τ_i. It consists of checking that their response time is always lower than their relative deadline when τ_i uses its allowance. Formula (3) of the lemma is the Necessary Condition for the feasibility of the task set taking into account the allowance of the faulty task.

Example:

The allowance computation is done for all the tasks given in the previous example (see table 1). The following table gives the allowance values for each task:

TABLE 2. Margin $A_{i,1}$ - temporal fault model [1/n]

Task	τ_1	τ_2	τ_3
$A_{i,1}$	250	300	500

For example, when a fault occurs, task τ_2 will be able to use an allowance equals to 300 time units. Figure 3 illustrates the synchronous scenario.

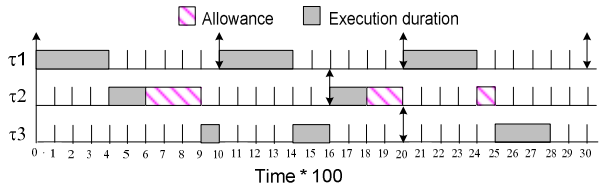


Figure 3. Task τ_2 Margin – example

These 300 time units guarantee that the other non faulty tasks will not be penalized by the faulty task and will respect their temporal constraints, as long as the faulty task does not consume more than its proper allowance.

IV. THE ALLOWANCE SHARING POLICY

Giving all the available resources to a faulty task is not an optimal solution in itself, as this can use up the totality of the resources without even correcting the fault. Worse than that, when consuming its allowance, the faulty task will reduce the allowance of more important tasks to zero. In that case, the recommended solution is the share of the available resources between the tasks. The resource sharing can be done in the following ways:

- Fair sharing allowance: this consists of allotting identical additional execution time to each faulty task, without taking into consideration the task parameters like: priority, importance and response time.
- Balanced allowance: this is based on tasks preference, i.e. when a task causes a fault, the available resources will be attributed according to importance of this task. In this case, a new parameter will be used: the weight, which will be attributed to each task according to its importance. The total amount of assigned weight must be equal to 1.

A. Fair sharing allowance

Let us look at the faulty tasks model [m/n]. In this section, we suggest an equal share of the static allowance between the m faulty tasks. For a task τ_i, we identify the set of *(m-1)* faulty tasks minimizing the allowance allotted to τ_i.

Lemma 2:

The maximum allowance that can be given to the faulty task τ_i, when using [m/n] fault model, is equal to the maximum duration that can be added to the execution of the faulty tasks. There are two conditions which must be fulfilled; firstly, (m-1) other faulty tasks must be taken into account and secondly, all deadlines must be respected.

$$(4)\quad R_i^{n+1} = C_i + A_{i,m} + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j + \sum_{\tau_j \in dis(i,m)} \left\lceil \frac{R_i^n}{T_j} \right\rceil A_{i,m} \le D_i$$

$\forall \tau_j \in lp(i)$:

$$(5)\quad R_j^{n+1} = C_j + \sum_{\tau_k \in hp(j), k \neq i} \left\lceil \frac{R_j^n}{T_k} \right\rceil C_k + \sum_{\tau_k \in dis(i,m) \cup \tau_i} \left\lceil \frac{R_j^n}{T_k} \right\rceil A_{i,m} \le D_j$$

$$(6)\quad \mathrm{U} + \sum_{\tau_j \in dis(i,m) \cup \tau_i} \frac{A_{i,m}}{T_j} \le 1$$

Where *dis(i,m)* = set of *(m-1)* which comprises the most unfavorable task execution time for the task τ_i in the event of faults. The most unfavorable task execution time is that which occurs when tasks are liable to consume most of the available resources, i.e. tasks which maximize the quantity of work $\left\lceil \frac{R_i}{T_j} \right\rceil$).

Proof:

Let τ_i be the faulty task. Let us suppose that there exists, at the most, *m* faulty tasks. In the case of equal shares, for a faulty task τ_i, each faulty task τ_j has an allowance $A_{i,m}$ added to its WCET. The tasks set with priority higher than the task τ_i will be divided into two sub-groups according to fault model of the task (faulty or not faulty). Applying theorem 1 with the new execution duration of τ_i we find the formula (4) of lemma 2.

The formula (5) consists of checking that the response time, of tasks with low priority than τ_i, is always lower than their relative deadline when τ_i consumes its allowance. The formula (6) of the lemma is the Necessary Condition for the feasibility of the task set taking into account the allowance of the faulty tasks.

Example:

We consider the parameters of the previous example (see table 1):

- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously at time t=0.
- The tasks τ_1, τ_2 and τ_3 have decreasing priorities.
- Any task can be faulty.

The following table gives the allowance values for each task:

TABLE 3. Margin value - temporal fault model [m/n]

Task	$A_{i,m}$		
	m=1	m=2	m=3
τ_1	250	125	100
τ_2	300	125	100
τ_3	500	166	100

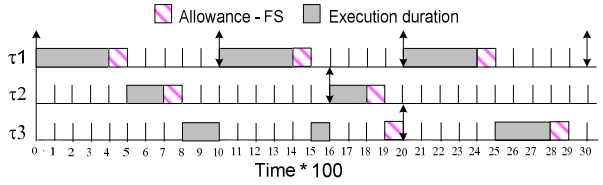


Figure 4. Margin example - fair sharing allowance

Figure 4 illustrates the scenario in which all the tasks are faulty, in this case no task misses its deadline if the faulty tasks do not consume more than their allowance.

Property 1:

The computation of the allowance, using FP and [m/n] fault model, has a pseudo-polynomial time complexity.

Proof:

The formula (6) of lemma 2 limits the maximum allowance for a task τ_i:

$$A_{i,m} \le \frac{1-U}{\sum_{\tau_j \in dis(i,m) \cup \tau_i} \frac{1}{T_j}}$$

Furthermore, we know that the deadline D_i is considered as an upper bound on the iteration number for the computation of the task response time ($R_i \le D_i$). This computation is carried out for a task τ_i and for all the tasks with lower priority than τ_i.

Thus we have $D_i + \sum_{\tau_j \in hp(i)} D_j$ iterations to execute. It follows that:

$$\left(D_i + \sum_{\tau_j \in hp(i)} D_j\right) \times \frac{1-U}{\sum_{\tau_j \in dis(i,m) \cup \tau_i} \frac{1}{T_j}} \le n \times (1-U) \times \max_{i=1..n}(D_i) \times \max_{i=1..n}(T_i)$$

The complexity is polynomial for each task, it takes $O(n \max_{i=1..n}(D_i) \max_{i=1..n}(T_i))$ time, thus the whole computation for n tasks takes $O(n^2 \max_{i=1..n}(D_i) \max_{i=1..n}(T_i))$, thus the complexity becomes pseudo-polynomial.

B. Balanced allowance

The share of the available free resources between faulty tasks should not be always being fair. A balanced approach takes into account the importance of a task. In that case, an additional parameter is used to balance the allowance among faulty tasks: a weight (φ_i) associated to task τ_i. The higher weight, the greater the importance.

Lemma 3:

The maximum allowance that can be given to the faulty task τ_i, when using [m/n] fault model, is equal to the maximum duration that can be added to the duration of the faulty tasks according to their weight.

$$(7)\quad R_i^{n+1} = C_i + A_{i,m} + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j + \sum_{\tau_j \in dis(i,m)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \left\lfloor A_{i,m} \times \frac{\varphi_j}{\varphi_i} \right\rfloor \le D_i$$

$$\forall \tau_j \in lp(i):$$

$$(8)\quad R_j^{n+1} = C_j + \sum_{\tau_k \in hp(j)} \left\lceil \frac{R_j^n}{T_k} \right\rceil C_k + \sum_{\tau_k \in dis(i,m) \cup \tau_i} \left\lceil \frac{R_j^n}{T_k} \right\rceil \left\lfloor A_{i,m} \times \frac{\varphi_k}{\varphi_i} \right\rfloor \le D_j$$

$$(9)\quad \forall i \in [1,n]: U + \frac{A_{i,m}}{T_i} + \sum_{\tau_j \in dis(i,m)} \left\lceil A_{i,m} \times \frac{\varphi_j}{\varphi_i} \right\rceil \frac{1}{T_j} \le 1$$

Where dis(i,m) = set of the *(m-1)* minimizing the allowance of task τ_i in the case of execution overrun faults. This set comprises tasks which maximize the quantity $\left\lceil \frac{R_i}{Tj} \right\rceil$.

Proof:

Let τ_i be the faulty task. Let us suppose that there exist at most *m* faulty tasks. In the case of balanced allowance sharing, for a faulty task τ_i, each *(m-1)* faulty tasks τ_j with higher priority than τ_i has an allowance equal to $\left\lfloor A_{i,m} \times \frac{\varphi_j}{\varphi_i} \right\rfloor$ added to its WCET. If we apply the modifications to the theorem 1 replacing the values of C_i by the C_i plus allowance, we will find equations (7) and (8) of lemma 3. Equation (9) of the lemma is the Necessary Condition for the feasibility of the task set, taking into account the allowance of the faulty tasks.

Example:

We consider the parameters of the previous example (see table 1):

- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously time t=0.
- The tasks τ_1, τ_2 and τ_3 have decreasing priorities.
- Any task can be faulty *(m=n)*.
- The WCET is the degree of importance of the task: $\varphi_i = \left\lfloor \frac{C_i}{\sum_{j=1}^{n} C_j} \right\rfloor$.

The following table gives the allowance values for each task:

TABLE 4. Margin value - Balanced allowance

Task	ϕ_i	$A_{i,3}$
τ_1	44	133
τ_2	22	66
τ_3	33	100

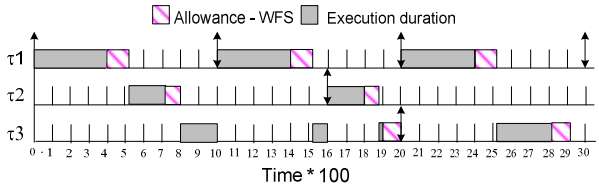


Figure 5. Margin example - balanced allowance

Figure 5 illustrates the scenario in which all the tasks are faulty. We observe that each task meets its deadline as long as the tasks do not consume more than their allowance.

V. LATEST EXECUTION TIME - LET

In order for static allowance to work properly execution overruns must be detected when they occur. This requires the presence of a detector in the system. This does not necessarily mean that available resource will be immediately consumed by a faulty task as a task with a higher priority may be running.

From this detector (WCET overrun), an approach using a budget manager (as in the approaches for management of the aperiodic tasks) is possible. [26] studies various strategies for budget management (differed server, polling server) for the allowance use with WCETs.

These solutions impose on each task a strict use of their allowance and do not allow tasks to recover the unused allowance of the other tasks. Moreover, when a task has an execution time lower than its WCET, it is impossible to recover the used duration to allocate it to other faulty tasks. Concerning the real-time java environment, the specification (RTSJ: Real-Time Specification for Java) proposes the use of handlers for the detection of the execution overrun (*CostOverRunHandler*) and for the detection of the deadline miss (*DeadlineMissHandler*) [27].

However, the execution overrun handler is rarely implemented in real-time systems (for example on *JbedRTOS* of the *Esmertec* Company [28]). In the current minimum implementation of RTSJ, the *CostOverrunHandler* is ignored, but a handler which detects an absolute deadline miss is provided. Thus we propose a new approach with an implicit recovery of resources not used by a task τ_i for all the tasks with lower priority than τ_i. Moreover, this approach solves the problem of the lack of overrun handlers.

It consists of using classical timers which are initialized with their Latest Execution Time (LET), computed according to the assumption that all the allowances have been consumed as well as the WCETs. The LET principles are described in the following subsections. We introduce the static and the dynamic LET. The first can be used for soft real-time systems; the second provides hard real-time guarantees.

A. Static Latest Execution Time

The static LET (called: $LET_{i,m}$) of a task τ_i corresponds to the worst-case response time of τ_i by supposing that all the m faulty tasks consume their allowance.

Lemma 4:

The static LET of a faulty task τ_i can be seen as a relative deadline of execution (a task released at time t must be executed by its absolute LET: t + $LET_{i,m}$), beyond which the risk of deadline miss is very high. Notice that with this approach, we cannot guarantee deadline respect (see figure 7). The static LET should therefore be used in a soft real-time context. It is equal to the worst-case response time of the task when all the faulty tasks consume their allowance. It is calculated as follows:

$$(10)\quad LET_i = C_i + A_{i,m} + \sum_{\tau_j \in hp(i)} \left\lceil \frac{LET_{i,m}}{T_j} \right\rceil C_j + \sum_{\tau_k \in disLet(i,m)} \left\lceil \frac{LET_{i,m}}{T_k} \right\rceil A_{k,m}$$

$$(11)\quad \forall \tau_k \in lp(i) \cup \tau_i : LET_k \le D_k$$

$$(12)\quad \forall i \in [1,n]: U + \sum_{\tau_j \in disLet(i,m) \cup \tau_i} \frac{A_{i,m}}{T_j} \le 1$$

Where disLet (i,m) = set of the *(m-1)* tasks minimizing the allowance of task τ_i in the event of faults (tasks suspected to consume the most free resources, i.e. tasks which maximize the quantity of work $\left\lceil \frac{LET_{i,m}}{T_k} \right\rceil A_{k,m}$).

Proof:

Let us suppose that there exist m faulty tasks. In this case, the WCET of each task is increased with a value equal to the allowance which is computed according to resource sharing mode (fair or balanced). Equation (10) corresponds to the formula of the worst-case response time computation. According to lemma 3, the new response time values must be lower than the tasks deadlines: $R_i \le LET_{i,m} \le D_i$, which respects the formula condition (11). The formula (12) of the lemma is the Necessary Condition for the feasibility of the task set.

Example:

We consider three tasks τ_1, τ_2 and τ_3 scheduled with FP, having decreasing priorities and we suppose at most *m=n=3* faulty tasks. We attribute to each task τi, i=1..3, the allowance values of $A_{i,3}$ and $LET_{i,3}$ (see table 5).

- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously at time t=0.
- Any task can be faulty.

The following table gives the allowance values for each task according to an equal share of the allowances between the *m* faulty tasks:

TABLE 5. Margin value - Static LET

Task	C_i	D_i	T_i	$A_{i,3}$	$LET_{i,3}$
τ_1	1	7	7	1	2
τ_2	2	11	11	1	5
τ_3	4	17	17	1	17

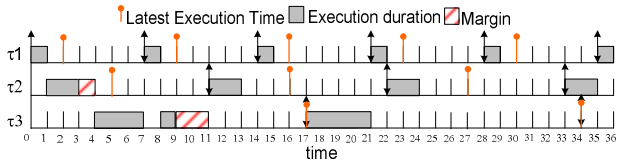


Figure 6. Execution example - Static LET

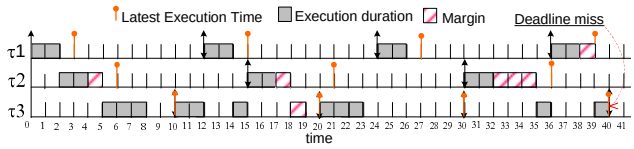


Figure 7. Execution example - Static LET limit

In figure 6, we can see that the first activation of task τ_3 may use several time units when it exceeds its WCET instead of the one granted by the allowance. With the LET mechanism, the unused allowance can be recovered by the faulty tasks.

Advantages of this solution:

- This mechanism improves system performance by the use of the allowance without requiring a cost overrun handler.
- The unused allowance of non faulty tasks can be recovered. Indeed, when a task does not use its allowance, all these saved resources (time CPU) can be used by all the other tasks.

In spite of its good performance, the LET static has a drawback; it does not guarantee the isolation of temporal faults. The computation of the static LET for task τ_i supposes that the tasks with higher priority are present in the system. This represents the worst-case scenario with the use of fixed priority driven schedulers. However, during execution, it is possible that a task with an intermediate priority can consume the allowance of task, with higher priority, not present in the system at that time. This has a knock-on effect on the execution of the task with lower priority (see figure 7). We identify this problem in the following example.

Example:

We consider three tasks τ_1, τ_2 and τ_3 scheduled with FP, having decreasing priorities and we suppose at most $m=n=3$ faulty tasks. We attribute to each task τ_i, i=1..3, the allowance values of $A_{i,3}$ and $LET_{i,3}$ (see table 6).

- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously at time t=0.
- Any task can be faulty.

The following table gives the allowance values for each task according to a fair share of the allowances between the *m* faulty tasks:

TABLE 6. Margin value - Static LET - example 2

Task	C_i	D_i	T_i	$A_{i,3}$	$LET_{i,3}$
τ_1	2	12	12	1	3
τ_2	2	15	15	1	6
τ_3	3	10	10	1	10

Figure 7 shows that there are possible cases where non faulty tasks (task τ_3 in figure 7) exceed their deadlines following faults generated by tasks with higher priority (tasks τ_1 and τ_2 in figure 7).

For example at $t_1 = 30$, the task τ_2 is running and has a LET deadline at $t_2 = 36$, (calculated with the presence of task τ_1) whereas τ_1 will not be activated until t = 36. The fact that τ_2 has the greatest priority in the interval $[t_1, t_2]$ authorizes the task to be executed up to t = 35 (see figure 7). It then takes all available CPU resources, and delays the execution of the task τ_3. At t_2, τ_1 starts its execution and leaves insufficient resources for task τ_3 (only one unit in this example). The failure of the task τ_3 at $t_3 = 40$ is due to an overconsumption of CPU resources by task τ_2, which by consuming its allowance, plus a part of that of τ_1 has indirectly used a part τ_3's allowance.

The problem with the static LET is thus that it is not possible to guarantee for periodic or sporadic tasks the isolation of temporal faults. However, it enables us to anticipate a deadline miss. The majority of real-time systems propose only one detector in the event of a deadline miss.

The problem with this solution is that a task which has missed its deadline may already have had a cascading effect on the other tasks of the system. The correction may come too late. Static LET, with its preemptive correction, is a possible solution for fault prevention. Here is the dynamic LET, which solves the problems brought to our attention by the use of the static LET.

B. *Dynamic Latest Execution Time*

Definition:

Let task τi be a new task released at time t_i. The dynamic Latest Execution Time of all the tasks in $\tau_i \bigcup lp(i)$ released at time t_j and FP scheduled is computed as follows:

(13) $$LET_{i,n}(t_i) = C_i + A_{i,n} + \max(t_i, \max_{\tau_j \in hp^R(i)} (LET_{j,n}(t_j))$$

(14) $$\forall \tau_j \in lp^R(i): \quad LET_{j,n}(t_j) = LET_{j,n}(t_j) + C_i + A_{i,n}$$

Where:

- t_j: is the last request time of task t_j
- $lp^R(i)$: denotes the set of tasks released and still in the system with priority lower than τ_i.
- $hp^R(i)$: denotes the set of tasks released and still in the system with priority higher than τ_i.

Lemma 5:

The dynamic LET guarantees the isolation of temporal faults in the event of WCETs overrun, as long as the faulty tasks do not exceed their dynamic LET.

Proof:

The dynamic LET is updated for all tasks with priority lower than or equal to task τ_i. It takes into account the released tasks with higher priority than τ_i. The dynamic LET

of the task τ_i corresponds to its completion time, after the execution of all the released tasks in *hp(i)*. For the task τ_i, the maximum number of requests for tasks activations with higher priority than τ_i is the maximum possible for the first activation of τ_i in the synchronous scenario. The response time of the first activation of τ_i in the synchronous scenario is the maximum limit of the response time of any instances of the task τ_i in any scenario. Thus we have:

$$\forall t_i \geq 0, LET_{i,n}(t_i) - t_i \leq R_i = C_i + A_{i,n} + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \left(C_j + A_{i,n}\right)$$

Where R_i is the response time, computed according to lemma 2, extended to the case $m=n$, including the allowances on WCETs, and respecting the condition: $R_i \leq D_i$.

Thus we find: $\forall t_i \geq 0, LET_{i,n}(t_i) - t_i \leq D_i$

Consequently all the tasks will respect their deadlines provided that they are not run after their dynamic LET. An execution overrun is then isolated as long as the tasks do not exceed their dynamic LET.

Example:

We consider three tasks τ_1, τ_2 and τ_3 with FP scheduling, having decreasing priorities and we suppose at most $m=n=3$ faulty tasks. Each task τ_i, i=1..3, has allowance values of $A_{i,3}$ (see table 7).

- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously at time t=0.
- Any task can be faulty.

The following table gives the allowance values for each task according to fair allowance sharing between the $m=n$ faulty tasks:

TABLE 7. Fair Allowance - Dynamic LET

Task	C_i	D_i	T_i	$A_{i,3}$
τ_1	4	10	10	1
τ_2	2	16	16	1
τ_3	3	20	20	1

At t=0, the dynamic LET of each task is equal to $C_i + A_i$ (values equal to: 5, 3 and 4 respectively). During the activation of τ_2, its LET is updated when a higher priority task is released into the system (task τ_1 in this example). When t=10, τ_1 activates and recalculates the LET of task τ_3, the LET of τ_3 changes from 12 to 17. At t=16, task τ_2 modifies the LET of task τ_3 up to the value of its deadline.

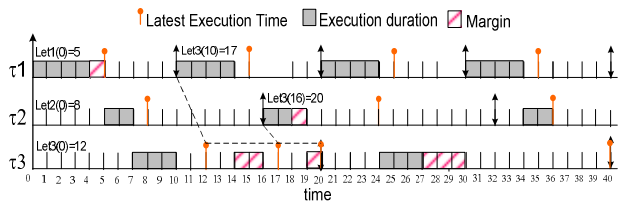


Figure 8. Execution example - Dynamic LET

The CPU resources not used by higher priority tasks can be exploited by task τ_3.

The advantages of the dynamic LET are shown in the following points:

- Error prevention: this is based on the determination of the maximum execution time that can be added to the EWCET of a faulty task without compromising the real-time constraints of all the tasks in the system.
- Failure prevention: our solution makes it possible to anticipate the failures. This leaves us a precious time to make a decision before the deadline miss and to guarantee the isolation of the fault if it occurs.
- Efficient resource management: unused resources can be recovered by faulty tasks automatically without needs of additional calculation (implicitly recovered).
- Independence of platform type: the dynamic LET is a solution, based on the use of timers' handlers, which is easily implemented on all real-times systems.
- Isolation of the faulty task: the dynamic LET guarantees the isolation of temporal faults in the event of EWCETs overrun, as long as the Dynamic LET is not exceeded by the faulty tasks.

C. Latest Execution Time implementation

We show in this section how to implement the allowance use in a real-time system. The following diagram (see figure 9-a) shows the possible execution states for a task.

- *Admission state*: ready to accept a new task τ_i. This state is only used for the first activation of the task. In this state, admission control, based on theorem 1, must be requested to check feasibility. If the new task set τ of n sporadic tasks, including τ_i is declared feasible; all the task deadlines can be met. The system then determines the allowance sets A_j, j=1..n based on lemma 2. Task τ_i can be started and then the scheduler will place the task into a *Ready state*. This means the task is ready to run. If task τ_i cannot be admitted, its execution state changes to *Halted*. The system will be notified of the admission failure. We will not deal in this paper with the treatment of this exception.
- *Running state*: based upon the behavior of the other tasks and threads in the system, a task is scheduled to begin executing, at which point it enters it's *Running state*.
- *Finished state*: while the task is running, the scheduler may preempt it and switch execution to another task. If this occurs, the Running task returns to a *Ready state*. Alternatively, the scheduler may decide to continue to execute the task. When the task has finally completed executing its run method, it enters a *finished state*.
- *Ready state*: in addition to the normal execution states, a task can enter a *Waiting state* if it needs resources which are not currently available. The scheduler may reschedule another task to begin execution. When the resources become available, the original task is return to a *Ready state* where it will be rescheduled.

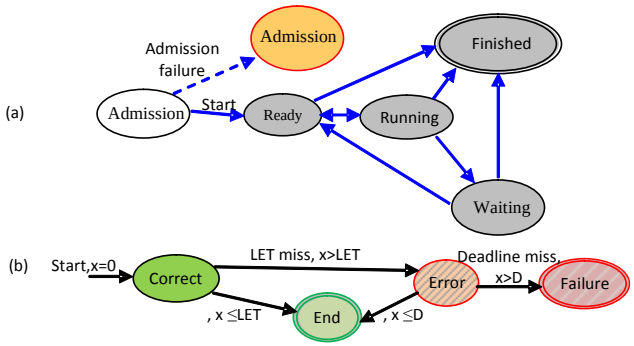


Figure 9. State diagram - Dynamic LET

The diagram b (see figure 9) shows the possible execution states of tasks:

- *Correct state*: if the task completes normally before the LET, its state changes to *end state*. Alternatively, the task becomes faulty and then the scheduler changes the task state to *error state*.
- *Error state*: different strategies can be used to deal with such a situation: stop the faulty task or execute it in the background (when no others tasks require execution). In the experiments carried out in the following section, we have chosen to stop the faulty task to prevent cascading effects on the other tasks. The state of the faulty task is set to *failure state* and an exception is used to inform the system of the failure.

VI. TESTS AND RESULTS

A. *Tests per simulation*

In order to make a comparative study on the robustness of the various solutions, it should be noted that in the following tests, we consider that the faulty tasks have real execution times which can exceed their execution time plus their allowance. We then obtain temporal failures and we compare the capacity of the various algorithms to contain these failures. Indeed, if all the tasks did not consume more than their allowance, we would not observe a temporal failure with the dynamic LET. In these simulations, we consider the following conditions:

- Given a set of 10 periodic tasks: τ_1, τ_2... τ_{10} scheduled with FP.
- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously at time t=0.
- The task τ_1 will have the greatest priority with very large execution duration.
- The tasks τ_1... τ_{10} will have very short execution duration.
- The tasks τ_1, τ_2... τ_{10} have decreasing priorities.
- The margin value for each task is calculated according to equal resource sharing between tasks.

We assume that every task can cause faults; we simulate this by modifying the task execution time according to the Normal law during each period. We carried out the test 100 times in order to study the failure tendency. In this experiment, we consider the tasks set τ with a processor utilization U=0.848. The goal of these simulations is to compare the performances between the various techniques of allowance management and a mechanism being satisfied to detect the deadline (NTD: nothing to be done in the case of faults).

The values represented on the x-axes indicate the number of simulations carried out (see figures 10, 11, 12 and 13). Each simulation corresponds to an execution scenario. The duration of the test is higher than the *lcm* of the periods of the tasks (14000 units of time).

The allowance value $A_{i,10}$, $\forall i \in [1,10]$, is equal to 22 time units. The following table (see table 8) gives the static LET values for each task according to an equal share of the allowances between tasks:

TABLE 8. Simulation example - 10 tasks

Task	C_i	D_i	T_i	P_i	$LET_{i,10}$
τ_1	120	200	200	10	122
τ_2	20	300	300	9	144
τ_3	20	300	300	8	166
τ_4	20	300	300	7	188
τ_5	5	500	500	6	195
τ_6	5	500	500	5	390
τ_7	5	500	500	4	397
τ_8	5	800	800	3	547
τ_9	5	800	800	2	554
τ_{10}	5	800	800	1	561

A task has an *indirect failure* when it does not exceed its WCET but misses its deadline. The deadline miss is then due when other faulty tasks exceed their WCET. This can only happen when no allowance mechanism is used. Without the allowance mechanism, we observe the multiplication of indirect failures.

Figure 10 shows that in the case of task fault, nothing to be done i.e. let the faulty task continue its execution, does not guarantee the task against a temporal failure but exposes the other tasks in the system to successive cascading failures. Only one task fault can cause several failures on the level of the other tasks. The use of allowance preserves the system from cascading effects. We remark also that the static allowance is less powerful than the LET techniques.

When the faulty tasks consume their entire allowance or if a task reaches it LET deadline, we put the execution of these tasks in background. The results obtained in this case (see figure 11) show the benefits of the solutions using the allowance. The performances (reduction of failures) of the solutions are always in the same order: the static LET gives the best results, followed by the dynamic LET and finally the equitable allowance.

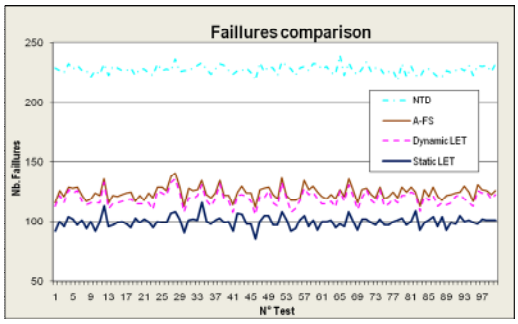


Figure 10. Failures comparison - stop after margin exceeds

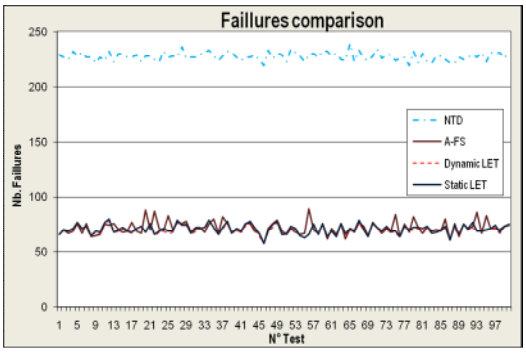


Figure 11. Failures comparison - background after margin exceeds

In the following simulation test, we increase the number of tasks in the system. We consider a set of 20 periodic tasks τ_1,.., τ_{20}. The first task will have the greatest priority. The 19 remaining tasks will have very short execution duration with decreasing priorities.

We assume that every task can cause faults, which is made possible by modifying the task execution time according to the Normal law during each period. We remake the test 100 times in order to study the failure tendency. In this experimentation, the processor utilization is equal to U=0.7766. The goal of this simulation is to study the effect of the increase on the number of tasks in the failure rate. The most important remark on this simulation (see figure 12) resides in the fact that the failure rate is considerably reduced if we use the allowance compared to the solution nothing to be done (NTD).

The increase in the number of tasks in the system amplifies the risk of the indirect failures. The results of the tests on the simulator show that the use of the allowance, whatever the adopted solution, makes it possible to reduce the number of failures in the system. We focused our tests on the LET mechanism which can be installed on a real-time platform. The results of simulations confirmed that the use of the static LET, in spite of some indirect failures, makes it possible to reduce the number of failures.

B. Practical Tests

The static LET mechanism has been tested on RedHat Linux 8 with kernel 2.4.18 and on the TimeSys RT/Linux 4 [29].

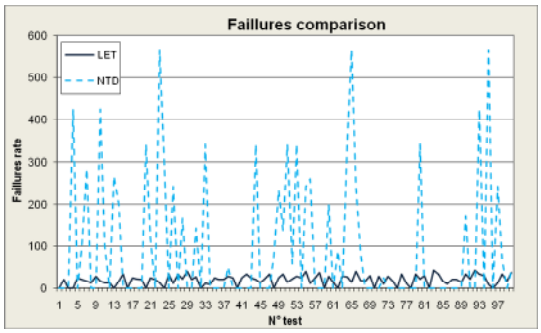


Figure 12. Failures comparison - LET and NTD

In this test, we compare the correction rates obtained by the static LET mechanism by simulation and on real platforms.

With the objective of to comparing like with like, we used the same example of tasks tested on the simulator (see table 8). The test was carried out on the platform described in the preceding point. In our test (see figure 13), we obtained on average, a 17% less good results of fault correction by using practical LET mechanism compared to the results obtained by simulation. In fact, we obtained a correction rate of faults, equal to 84% during the tests on the simulator and 67% during the tests on a real-time platform. We suspect the garbage collector of Java to be partly responsible for this degradation. The practical tests confirmed the benefits of the use of the allowance concept.

In the last experiment, we consider the same set of tasks; the difference resides in the execution duration of the task τ_1 which varies between 900 and 1500. This corresponds to a utilization ratio of the processor which varies between 0.533 and 0.883.

The tests were carried out in a synchronous scenario for a length of time equal to 10 times the *lcm* of the tasks (120000 time units). For each test, we take the average of 10 executions, which corresponds approximately to twenty minutes per test.

The test objective is to analyze, on a real-time platform *Jtime (TimeSys)*, the influence of the processor utilization rate on the performances of the static LET mechanism (see figure 14).

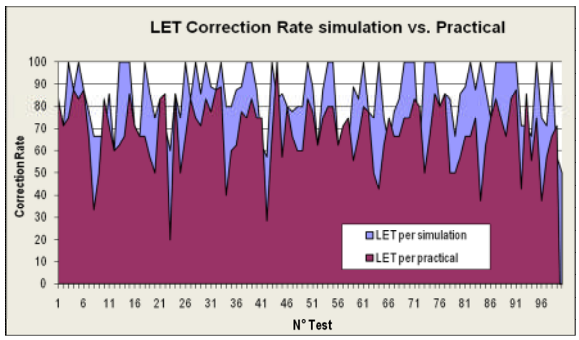


Figure 13. Correction rate - LET simulation vs. LET practice

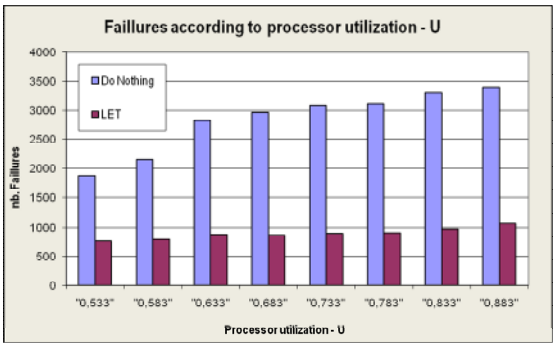


Figure 14. Failures - utilization rate of the processor

We can notice that the number of failures is more important when the static LET is used, which is not the case with solution *NTD*. The tests of the solutions on the simulator show the improvements made by the use of the allowance. The decrease of the performances in practice is explained by the system cost of the allowance mechanism.

VII. CONCLUSION

In this article we have considered the problem of fault prevention in a real-time system. The faults correspond to WCETs overruns resulting from the use of estimated WCETs. This enables us to consider an open architecture, independent of the operating system. The use of a virtual machine such as Java allows this independence. In this context, the worst case execution times are not easily determinable by a static analysis due to portability issues. The design of real-time system then requires techniques to tolerate an uncertainty on the WCET. The solution that we propose is based on the determination of the acceptable deviations on the WCETs which is called the allowance. The allowance enables us to reduce the failure rate of the real-time tasks and to make the system more robust in the event of temporal fault. This mechanism permits the recovery of free CPU resources in the event of temporal fault. We propose equations to compute the allowance in the case of a fixed priority scheduling. We proposed an implementation based on the computation of the Latest Execution Time (LET) preserving the timeliness constraints of the tasks. Simulations carried out and the studies on a real platform enabled us to conclude that this mechanism is interesting to limit the impact of temporal faults of the WCETs overruns by increasing the rate of their correction. Furthermore, the LET provides an implicit recovery of the unused allowance.

ACKNOWLEDGMENT

The authors gratefully acknowledge Siân Cronin at ESIGETEL for providing linguistic help.

REFERENCES

[1] L. Bougueroua, L. George, S. Midonnet, Dealing with execution-overruns to improve the temporal robustness of real-time systems scheduled FP and EDF. The second International Conference on Systems – ICONS 2007.Sainte-Luce, Martinique, 22-28 April 2007.

[2] Liu, C.L., Layland, J.W., Scheduling Algorithms for multiprogramming in a Hard Real Time Environment; Journal of Association for Computing Machinery, Vol. 20, N° 1, (Jan 1973).

[3] Joseph, M., Pandya, P., Finding reponse times in a real-time system ; BCS Computer Journal, 29(5), (1986) 390-395.

[4] Lehoczky, J.P., Fixed priority scheduling of periodic task sets with arbitrary deadlines; Proc. 11th IEEE Real-Time Systems Symposium, Lake Buena Vista, Floride, USA, (Dec 1990) 201-209.

[5] Puaut, I., Méthodes de calcul de WCET (Worst-Case Execution Time) Etat de l'art; 4ème édition Ecole d'été temps-réel(ECR), Nancy, (Sep 2005) 165-175.

[6] Tindell, K., Burns, A., Wellings, A. J., An extendible approach for analyzing fixed priority hard real time tasks ; Real-Time Systems, Vol. 6, N°2, (Mar 1994)133-151.

[7] L. Bougueroua, L. George and S. Midonnet, An execution overrun management mechanism for the temporal robustness of java real-time systems. The 4th workshop on Java technologies for Real-Time and Embedded Systems (JTRES) 11-13 October 2006, Paris.

[8] Locke, C.J., Best effort decision making for real-time scheduling; PhD thesis, Computer science department, Carnegie-Mellon university, (1986).

[9] Koren, G., Shasha, D., D-over: An Optimal On-line Scheduling Algorithm for over loaded real-time system; technique report 138, INRIA, (Feb 1992).

[10] Buttazo, G., Stankovic, J.A., RED: A Robust Earliest Deadline Scheduling; 3rd international Workshop on responsive Computing, (Sept 1993).

[11] Buttazzo, G., Lipari, G., Abeni, L., Elastic Task Model for Adaptive Rate Control; Proc. IEEE Real-Time Systems Symposium, Madrid, Spain, (Dec 1998) 286-295.

[12] Buttazzo, G., Lipari, G., Caccamo, M., Abeni, L., Elastic Scheduling for Flexible Workload Management; IEEE Transactions on Computers, Vol. 51, No. 3, (Mar 2002) 289-302.

[13] Bini, E., Di Natale, M., Buttazzo, G., Sensitivity Analysis for Fixed-Priority Real-Time Systems; Proc. 18th Euromicro Conference on Real-Time Systems, ECRTS'06, (2006).

[14] Racu, R., Jersak, M., Ernst, R., Applying sensitivity analysis in real-time distributed systems; Proc. 11th Real-Time and Embedded Technology and Applications - RTAS'05, (2005).

[15] Racu, R., Hamann, A., Ernst, R., A formal approach to multi-dimensional sensitivity analysis of embedded real-time systems; Proc. 18th Euromicro conference on realtime systems - ECRTS'06, (2006).

[16] Bini, E., Buttazzo, G., Schedulability Analysis of Periodic Fixed Priority Systems; IEEE Transactions On Computers, Vol. 53, No. 11, (Nov 2004).

[17] Lehoczky, J. P., Ramos-Thuel, S., An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks Fixed-Priority Preemptive systems; Proc. Real-Time System Symposium, (Dec 1992) 110-123.

[18] Davis, R.I. Scheduling Slack Time in Fixed Priority Pre-emptive Systems; Proc. 14th Real-Time Systems Symposium, (1993) 222-231.

[19] Spuri, M., Buttazzo, G., Scheduling aperiodic tasks in dynamic priority systems; Journal of real time systems, vol. 10, (1996) 179-210.

[20] Caccamo, M., Lipari, G., Buttazzo, G., Sharing resources among periodic and aperiodic taskc with dynamic deadlines; Proc. 20th IEEE Real Time System Symposium, (1999).

[21] Kuo, T.W., Mok, A.K., Load Adjustment in Adaptive Real-time Systems; Proc. 12th IEEE Real-Time Systems Symposium, (Dec 1991).

[22] Nakajima, T., Tezuka, H., A Continuous Media Application Supporting Dynamic QoS Control on Real-Time Mach; Proc. ACM Multimedia, (1994).

[23] Seto, D., Lehoczky, J.P., Sha, L., Shin, K.G, On Task Schedulability in Real-Time Control Systems; Proc. IEEE Real-Time Systems Symposium, (Dec 1997).

[24] Hamdaoui, M., Ramanathan, P., A dynamic priority assignment technique for streams with (m,k)-firm deadlines; IEEE Transactions on Computers, vol. 44(12), (1995) 1443-1451.

[25] Bernat, G., Burns, A.A.L., Weakly Hard Real-Time Systems; IEEE Transactions on Computers, (2001).

[26] Bougueroua L., Conception de systèmes temps réel déterministe en environnement incertain; PhD thesis, vol 1, n° 2007PA120004, SI (Mar 2007).

[27] Bollela, G., Gosling, Brosgol, Dibble, Furr, Hardin and Trunbull, The Real-Time Specification for Java; Addison Wesley, 1st edition, (2000).

[28] Esmertec, jbedRTOS: Java Bulding Embedded Operating System; http://www.esigetel.fr/images/stories/Recherche/SITR/spec-jbed-rtos.pdf.

[29] TimeSys, TimeSys' real-time Java Virtual Machine (JVM); http://www.timesys.com.

www.iariajournals.org

International Journal On Advances in Intelligent Systems
ICAS, ACHI, ICCGI, UBICOMM, ADVCOMP, CENTRIC, GEOProcessing, SEMAPRO, BIOSYSCOM, BIOINFO, BIOTECHNO, FUTURE COMPUTING, SERVICE COMPUTATION, COGNITIVE, ADAPTIVE, CONTENT, PATTERNS
issn: 1942-2679

International Journal On Advances in Internet Technology
ICDS, ICIW, CTRQ, UBICOMM, ICSNC, AFIN, INTERNET, AP2PS, EMERGING
issn: 1942-2652

International Journal On Advances in Life Sciences
eTELEMED, eKNOW, eL&mL, BIODIV, BIOENVIRONMENT, BIOGREEN, BIOSYSCOM, BIOINFO, BIOTECHNO
issn: 1942-2660

International Journal On Advances in Networks and Services
ICN, ICNS, ICIW, ICWMC, SENSORCOMM, MESH, CENTRIC, MMEDIA, SERVICE COMPUTATION
issn: 1942-2644

International Journal On Advances in Security
ICQNM, SECURWARE, MESH, DEPEND, INTERNET, CYBERLAWS
issn: 1942-2636

International Journal On Advances in Software
ICSEA, ICCGI, ADVCOMP, GEOProcessing, DBKDA, INTENSIVE, VALID, SIMUL, FUTURE COMPUTING, SERVICE COMPUTATION, COGNITIVE, ADAPTIVE, CONTENT, PATTERNS
issn: 1942-2628

International Journal On Advances in Systems and Measurements
ICQNM, ICONS, ICIMP, SENSORCOMM, CENICS, VALID, SIMUL
issn: 1942-261x

International Journal On Advances in Telecommunications
AICT, ICDT, ICWMC, ICSNC, CTRQ, SPACOMM, MMEDIA
issn: 1942-2601