Enabling Flexible Medium Access Design for Wireless Sensor Networks

Junaid Ansari, Xi Zhang, Obaid Salikeen and Petri Mähönen
Institute for Networked Systems, RWTH Aachen University, Kackertstrasse 9, D-52072, Aachen, Germany
Email: {jan, xzh, osa, pma}@inets.rwth-aachen.de

Abstract—Wireless sensor networks are integrated into various daily life applications. These diversified applications impose different requirements on MAC protocols. Therefore, it has become increasingly important to be able to prototype new MAC designs efficiently. In this paper, we describe a component based framework, which enables flexible and rapid MAC designing for sensor networks. We have identified and implemented a basic component library, which serves as the basis for realizing a wide range of protocol designs. We provide an interactive graphical user interface for designing MAC protocols and a tool-chain for auto-code generation. This takes off the burden of learning programming language and platform details from the designers. Our framework facilitates fast prototyping of MAC solutions and allows reusability of existing design sessions through an interactive ‘drag n drop’ based user friendly environment. We have carried out detailed performance comparison of widely known MAC protocols developed through our framework to their monolithic counterparts on real sensor network testbed. Our results indicate that the framework allows high degree of component reusability across different MAC implementations with only a slight memory overhead and without compromising performance characteristics.

I. INTRODUCTION

Wireless Sensor Networks (WSNs) are deployed in a wide range of daily life applications with different sensing and communication requirements. As a consequence, various MAC solutions have been proposed Federal to different application characteristics [1], [2]. Designing and implementing MAC protocols is a tedious task and typically requires domain knowledge of the underlying hardware platform and the operating system. Furthermore, MAC protocols are generally implemented in a monolithic fashion with tight coupling to the underlying platform which limits the room for experimentation and makes modifications a cumbersome task. This also hinders the possibilities of code reuse and portability to other platforms. Bachir et al. [3] have shown that none of the popularly used protocols give a universally optimal behavior, which advocates multi-mode and multi-protocol operation depending upon the application requirements. This affirms the importance of designing different suitable protocols and the code reusability.

We have an on-going research towards flexible MAC realizations for wireless networks with a focus on designing efficient hardware architectures and MAC meta-language. We have proposed a decomposable MAC framework [4] which partitions the MAC component implementation in software and hardware appropriately in order to simultaneously allow flexibility and execution efficiency. A particular MAC protocol can be composed at runtime using the wiring engine and a host meta-compiler. Although such a MAC realization is not feasible for resource constrained sensor nodes, here we have applied the concept of composing MAC protocols based on common functional components in order to allow fast prototyping, high degree of code reuse and minimizing porting efforts. Additionally, we have developed a ‘drag and drop’ based user-friendly environment for designing WSN MAC protocols. We have analyzed a wide range of MAC protocols in order to distil the common features into a set of reusable basic components. We have also identified more complex common patterns among protocols and defined higher level components based on the basic components. The goal is to realize a particular MAC implementation by connecting these reusable unit components together. The design complexity is lowered through the use of higher level components. The idea is based on the LEGO(TM) philosophy, where complex structures can be constructed using basic building blocks.

We have designed an interactive GUI based environment, which we refer to as MAC Protocol Designer (MACPD). MACPD allows designers to express MAC protocols through flowcharts composed of various MAC subcomponents. MACPD auto-generates TinyOS 2.x source code [5] and executable scripts from the designed flowchart which are directly compiled and downloaded onto sensor nodes using the same GUI based development environment. Currently, the framework supports TelosA, TelosB, Micaz, BSN-node and IntelMote2 sensor node platforms. XML model files generated from the flowcharts are used for storing the re-loadable design developed by a user. The user can thereby simply reuse and modify the previously saved sessions. The user can also add a particular customized design to the library as a reusable component, which reduces the future designing efforts. Our framework provides a tool-chain from a MAC design to the execution of the code on the selected sensor node platform. An introductory video of the tool-chain can be found at [6]. We will demonstrate the user-friendly aspects of MACPD at ACM SenSys’10 [7]. There is a common belief that auto-generated embedded code results in large memory footprint and lacks performance characteristics. However, our work shows that careful component designing approach based on the functional commonalities among protocols gives similar performance metrics on COTS sensor nodes as the monolithic MAC implementations and also allows fast prototyping.
The rest of the paper is organized as follows: In Section II we review the related research work in the field. Section III presents the design rationale of the framework, while Section IV describes the implementation details. Section V presents the empirical performance evaluation of the framework and its comparison to existing approaches. Finally, Section VI concludes the article with future work directions.

II. RELATED WORK

In related research work, efforts have been made to achieve a unified protocol structure for easing implementation burden and maximizing code reuse. Polastre et al. propose a unified link layer abstraction, which helps the implementation of a broad range of networking and data link technologies without significant loss of efficiency [8]. ULLA [9] focuses on the design of flexible link layer APIs. It offers a common interface to fetch link layer information independent of the underlying radio technology. In Unified Power Management Architecture (UPMA) [10], link-layer interfaces are defined to separate core radio functionality from power management features so that power managers can be plugged into different hardware specific MAC implementations. In a continuation work on UPMA, focus has been laid on the optimization of all the radio states [11] for achieving energy efficiency. The component-based MAC Layer Architecture (MLA) [12] [13] further extends the UPMA idea by identifying and modularizing platform-independent components. The implementation shows significant code reuse across different protocols with a low memory overhead without significant loss of performance metrics. However, MLA decomposes only the platform independent part of MAC protocols. The hardware and protocol specific code still contributes heavily to the MAC implementation effort, which we have addressed in our work. Since we focus on identifying the basic MAC functional components, unlike MLA, our framework is not limited to the existing protocol genre and is extendible to non-classical designs. This certainly widens the experimental room for research activities using our approach.

III. FRAMEWORK DESIGN

We have analyzed a wide range of MAC protocols based on the CSMA, TDMA and hybrid principles in order to identify the building blocks [4]. Our goal is to design a framework that allows easy and fast implementation of MAC protocols in a user friendly manner with a high degree of code reuse.

Our primitive level components are defined to be the low-level hardware-dependent functionalities providing platform independent interfaces, e.g. timer, random number generator, carrier sensing, etc. In this work, we have implemented the basic components for the popularly used Texas Instrument’s CC2420 packet radio. Besides the basic components, we also define higher level components which consist of the basic components. The higher level components depict the similarity patterns among different protocols and can flexibly be defined by a MAC designer using our framework.

TABLE I

<table>
<thead>
<tr>
<th>Block</th>
<th>Usage and composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFF</td>
<td>Binary exponential backoff mechanism, Random Number Generator, Carrier Sensing</td>
</tr>
<tr>
<td>LplCoordinator</td>
<td>Controlling the Low Power Listening (LPL) operation, Radio Control, Carrier Sensing, Timer</td>
</tr>
<tr>
<td>Expect Frame</td>
<td>Used when a node awaits for a packet, ReceiveFrame, Timer</td>
</tr>
<tr>
<td>Send Packet</td>
<td>Used for sending a packet, SendFrame, Radio Control</td>
</tr>
<tr>
<td>Send Preamble</td>
<td>Used in Preamble Samping MAC protocols, SendFrame, Timer</td>
</tr>
<tr>
<td>RTS/CTS/DS/ACK</td>
<td>Four-way handshake mechanism, Send Packet, Expect Frame, Timer, Carrier Sensing</td>
</tr>
</tbody>
</table>

A. Basic Components

Basic components represent the fundamental functionalities that MAC protocols would require, such as timer, backoff counter, carrier sensing, frame formation, sending a frame, receiving a frame, etc. These components are not only common to different protocols but are also repeated within a particular protocol. Radio control parameters such as radio state switching, setting the frequency, transit power levels, receiver sensitivity thresholds, etc. are also inevitably used in all the MAC protocols.

B. Secondary Components

Across many different MAC protocols, the pattern and order in which primary blocks are connected is often identical. Therefore, we also define secondary level MAC components as building blocks. We have implemented a number of secondary level MAC components in the framework based on the provided functionality and their degree of re-use across different protocols. The most commonly used components are listed in Table I. Besides only the basic components, secondary components can also be used as sub-blocks in realizing other more complicated secondary components.

C. Arithmetic, Binary and Logical Expressions

Apart from the MAC functional components, arithmetic, binary and logical expressions are necessary to link the components, declare and process parameters and variables, etc. while realizing the complete state-machine of a MAC protocol. These expressions are exposed in the GUI to the end-users.

IV. IMPLEMENTATION DETAILS

Fig. 1 shows the layered architecture of the framework, where each layer performs its tasks in a top-down work flow. The layers are decoupled through well-defined interfaces.

A. User Interaction

The top most layer provides a flexible and user-friendly interface for the MAC design. It is based on the Java-Swing GUI (c.f. Fig. 3) and allows interactive ‘drag-and-drop’ feature for MAC development. All the basic and custom defined MAC
components are made available to the user. A designed can simply select a component and drag it to the provided drawing grid. The User Interface layer gathers the information of the state-machine and logic flow of the MAC from the user.

Fig. 2 represents the graph data structure used for maintaining a particular block design. There are two types of edges: ‘Contains Edge’ and ‘Follow Edge’. Contains Edge reflects the implementation of a method, whereas the Follow Edge is used for showing the connection among the nodes. Four different types of nodes are used to maintain the states of different TinyOS 2.x constructs. ‘BlockNode’ maintains all the information related to method calls of the TinyOS 2.x specific constructs while ‘DeclarationNode’ keeps track of the state information of different declarations. The nodes in the graph represent TinyOS 2.x command calls, event signals, function calls, task postings and the corresponding declarations, event-handlers and implementations. Furthermore, the program logic with If-Else statements and arithmetic, boolean and logical expressions are represented through the ‘IfElseNode’ and ‘ExpressionNode’, respectively. A user sets the states of the nodes through dialog screens and defines connections among the nodes to complete the MAC design. The edges in the graph represent the flow of a protocol. While parsing, the top layer processes the information of the graph and reports the missing or invalid data. It also validates the basic design and reports illogical connections.

B. XML Modeling

This layer is responsible for maintaining an XML model of the user design. XML tags are generated for each node, which preserves the state of a node and its relationship with other nodes in the graph. XML model is used to store TinyOS 2.x constructs in a tree structure. All the user designs are preserved in XML so that these can be utilized and updated without having the need to redesign everything from the scratch. For instance, we implemented MFP-MAC [14] simply by few modifications on the flowchart design of B-MAC [15] in the GUI. Fig. 4 shows the XML definition of the Radio Control component as an example. It contains the nesC module, configuration and interface file listings and the definitions of all the declarations associated with the interface. The figure shows only the startRadio command. All the five possible declarations in nesC, commands, tasks, event definitions, event handlers and normal functions [5], are modeled in XML in a similar manner.

C. Code Loader

The ‘Loader Layer’ interacts directly with XML model layer to load the XML in RAM in the form of Java objects. These objects are populated with the state information of the nodes from XML and are used in the auto code generation process.
D. TinyOS 2.x Code Generation

The ‘Code Generator Layer’ parses and translates the Java objects into auto generated TinyOS2.x source code, which is compiled and directly deployed onto the targeted platform. We have specially considered that the auto-generated code is clean and systematic so that advanced users may easily make modifications and build custom applications on top of it. We have implemented a number of MAC protocols using the framework including S-MAC [16], B-MAC [15], MFP-MAC [14] and X-MAC [17]. The implementations provide a well-defined MAC API to allow interaction with the MAC to the application and to control the MAC parameters. For instance, the interface of the MFP-MAC below contains the common functions to send/receive packets and to control the LPL operation of the MAC protocol.

```java
interface MFP_MAC {
    command void setCheckInterval(float interval);
    command void setDutyCycle(float dutyCycleValue);
    command uint8_t getCheckInterval();
    command float getDutyCycle();
    command void init(bool ENABLE_ACR);
    command error_t send(message_t* msg, uint8_t preambleType, uint8_t len, an_addr_t destAddress);
    event void sendDone(message_t* msg, error_t error);
    event message_t* receive(message_t* msg, uint8_t len);
}
```

Fig. 5 shows a simplified component diagram of the auto-generated code for MFP-MAC. The MAC implementation contains the low-level platform drivers upon which the basic components have been implemented. The basic components are exposed as a user library (in addition to other higher level components) in the GUI for developing a particular MAC or designing other higher level components.

![Simplified component diagram of the auto-generated code for MFP-MAC protocol.](image)

V. PERFORMANCE EVALUATION

We have conducted the analysis for component reuse, proportion of the total number of lines of code to the reusable lines of code, memory footprint and the achieved throughput for well-known MAC protocols implemented using MACPD. We have also carried out comparative empirical studies on these metrics for different MAC characteristics to their monolithic counterparts and those developed through MLA [18].

A. Component Re-usage

MACPD allows realizing a MAC protocol by simply connecting the functional components together. The idea is to minimize the protocol specific code to a minimal by providing flexible and reusable MAC components. Table II lists the components which are used in the implemented protocol realizations. The shaded component list represents the basic components. It may be noted that radio control (functionality: turning on/off the radio, switch state TX/RX), sending and receiving frames are fundamental to all the protocols. Timer instances and carrier sensing functionalities are also common to all the contention based protocols. Having a rich set of reusable components reduces the implementation efforts on part of the designer. This is indicated through a high proportion of the reusable lines of code as described in Section V-B.

<table>
<thead>
<tr>
<th>TABLE II</th>
<th>COMPONENT REUSE FOR MAC REALIZATIONS.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>B-MAC</td>
</tr>
<tr>
<td>Sent Preamble</td>
<td>1</td>
</tr>
<tr>
<td>Block</td>
<td>Expect Frame</td>
</tr>
<tr>
<td>Block</td>
<td>Send Packet</td>
</tr>
<tr>
<td>Block</td>
<td>BEB</td>
</tr>
<tr>
<td>Block</td>
<td>LplCoordinator</td>
</tr>
<tr>
<td>Block</td>
<td>SendFrame</td>
</tr>
<tr>
<td>Block</td>
<td>ReceiveFrame</td>
</tr>
<tr>
<td>Block</td>
<td>RadioControl</td>
</tr>
<tr>
<td>Block</td>
<td>Timer/Alarm</td>
</tr>
<tr>
<td>Block</td>
<td>Rand. No. Gen.</td>
</tr>
<tr>
<td>Block</td>
<td>Carrier Sensing</td>
</tr>
</tbody>
</table>

B. Code Line Reuse

Fig. 6 shows the total number of lines of nesC code and the number of lines of reusable code for different MAC protocols.

![Overall number of lines of nesC code and the percentage of the reusable code for different protocols implemented using MACPD.](image)
implemented through our framework. It is remarkable that the overall proportion of the code reusability across different MAC implementations is approximately 80%. This indicates that our framework gives a high degree of code reusability, thereby providing a fast means for prototyping different MAC solutions.

Fig. 7 compares the proportion of the nesC code reuse for our framework to the MLA [12] approach. It is evident that our approach allows a higher proportion of code reuse as compared to the MLA. For the three protocols considered, the code reuse for MACPD is above 80% while it is approximately 55% for MLA. The total number of lines of code for the MAC implementations in MLA is approximately 75% of the auto-generated code through our framework. MACPD gives a bigger total number of lines of code because of the higher component modularization offered by it.

C. Memory Footprint

MACPD generates the nesC code for TinyOS 2.x operating system for the target platform. The MAC code is embedded with the rest of the components and the TinyOS 2.x scheduler as a single executable binary. Table III lists the memory footprint in terms of RAM and ROM consumption for the binary. The table shows the effect of CC2420 driver stack implementation in TinyOS 2.0.1 and TinyOS 2.1.1 for TelosB platform. It can be observed that there is a slight decrease in the code memory for the in the later version because of more modularized stack implementation. Table IV compares the memory footprint on TelosA, MicaZ and Imote2. It is worth noting that the TinyOS 2.x binary along with the MAC protocols occupies no more than approx. 5% of the available RAM and ROM on these platforms.

Table V shows the comparison of the memory footprint for different MAC implementation among the monolithic implementation, MLA based implementation and the implementation using our framework. In order to ensure fairness in our analysis, we used the same CC2420 driver stack (TinyOS 2.0.1) and the same test application for the three design approaches. The table shows that the MAC implementations using MACPD are clearly more memory efficient than MLA and despite the auto-generated code, their memory footprint closely resembles to those of the monolithic implementations.

D. Goodput Analysis

In order to observe the performance characteristics of MAC protocols developed through MACPD, we replicated the experiment.
Fig. 9. Aggregate goodput comparison of B-MAC developed using MACPD and MLA at different Packet Generation Rates (PGRs).

In particular, we have defined and implemented a rich set of reusable flexible functional components upon which a MAC protocol can be built with minimal efforts. We have released the source code of MACPD at [19].

ACKNOWLEDGMENT

We thank the financial support from Deutsche Forschungsgemeinschaft through UMIC Research Centre, EU through 2PARMA and WASP projects and RWTH Aachen University.

REFERENCES