Component-Based Model for Heterogeneous Nodes in Wireless Sensor Networks

Atif Naseer
Science and Technology Unit, Umm Al-Qura University, Makkah, Saudi Arabia
Email: anahmed@uqu.edu.sa

Basem Y. Alkazemi and Hossam I Aldoobi
College of Computer and Information Systems, Umm Al-Qura University, Makkah, Saudi Arabia
Email: {bykazemi, hidoobi}@uqu.edu.sa

***Abstract*—Data exchange between heterogeneous motes in a network requires careful understanding of the data model adopted by every mote type. This can be a very challenging concern, especially when dealing with a wide variety of motes at runtime where new nodes can join a network to provide certain services. This paper identifies the key architectural characteristics of two commonly recognized operating systems, namely TinyOS and Contiki, which behave differently when dealing with data exchange and state control. The aim is to map the architecture of TinyOS and Contiki on to a component-based software engineering model. This discussion leads to describing abstractly the design of a generic component-based middleware model to address interoperability issues when dealing with these two types of motes.**

***Index Terms*—wireless sensor network, middleware, operating system, interoperability, component-based model**

I. Introduction

Wireless sensor networks represent the backbone of various industrial and healthcare applications nowadays. They consist of a large number of sensor nodes (motes) that are able to sense and communicate data between the underlying network and the applications. These nodes can vary in their software and hardware architecture according to application needs. However, a generic architecture of motes is described in Fig. 1. Wireless sensor networks are composed of many sensor nodes capable of performing different types of action. A number of operating systems for motes are available; however, we limited our investigation in this work to only Contiki and TinyOS, as we believe these are the most commonly used nowadays. Both operating systems comply with specific architectural assumptions that are significantly different from one another. To communicate with these two types of nodes, different message types have to be sent to each one as per their predefined signatures. Therefore, this paper identifies the differences between these two operating systems mainly in terms of data exchanging models in order to pave the way for designing a generic middleware model that addresses these differences. This paper characterizes the architecture of TinyOS and Contiki into a component-based software engineering model.

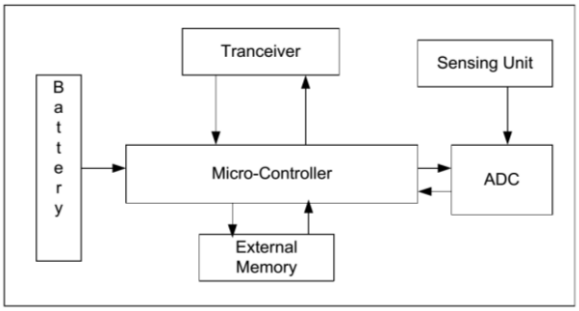

Figure 1. Architecture of a sensor node

The paper is further structured as follows. In Section II, we discuss component-based software engineering. In Section III, we present the TinyOS and Contiki programming models. We then characterize TinyOS and Contiki into a component-based model in Section IV. We discuss future research directions and conclude the paper in Section V.

II. Component-Based Software Engineering

"Component-Based Software Engineering (CBSE) is a branch of software engineering that stresses the separation of concerns in respect of the wide-ranging functionality available throughout a given software system" [1]. In component-based software engineering, each component is integrated with some other components to fulfill user requirements. Most of the components available in this model can be reusable. A software component can be a software package, a web resource, a web service or a module. All system processes are placed into separate components so that all of the data and functions inside each component are semantically related (just as with the contents of classes). Because of this principle, it is often said that components are modular and cohesive.

Components communicate with one another via interfaces, thus every component must have two types of interface: 'provides' and 'uses'. When a component

Manuscript received March 1, 2015; revised July 11, 2015.

doi: 10.18178/lnit.3.1.25-30

offers services to other components, it needs a 'provides' interface. This interface is the signature of that component. When a component needs to use another component, it adopts its 'uses' interface, which represents the services it needs. Fig. 2 shows a software-component-based model with two components (A and B) and their 'provides' and 'uses' interfaces. Component A provides some interfaces that can be used by component B. Also, it can use the interface of component B with its 'uses' interface. Hence:

- A component should have its own data and function inside each component so that each component is semantically related (just as with the contents of classes).
- In a complete system, a component should communicate with other system via their interfaces.
- A component may be composed of other components, or a component can instantiate some other components.

Every desktop and laptop computer must have an operating system running, but the operating systems designed for wireless sensor networks are quite different in their architecture. Some of these operating systems are static in nature, while others have dynamic functionality; also, some have a monolithic approach, while others have a modular approach.

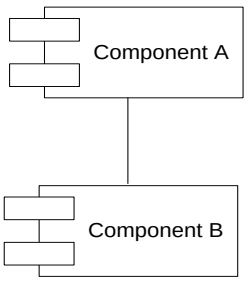

Figure 2. Component-based model

There are lots of well-known operating systems and programming models available for wireless sensor networks. Here, we consider TinyOS and Contiki operating systems and study their programming model architecture. The aim is to map the architecture of TinyOS and Contiki onto a component-based software engineering model. For this purpose, we consider some basic characteristics of a component model and map the architecture of TinyOS and Contiki according to these characteristics. These characteristics are:

- Initialization: In a component-based model, every component uses some methods to initialize themselves or other components.
- State control: There should be some method that will control the state of the components.
- Communication: The communication between components is done through some interfaces; hence, there should be some methods used for binding two components.
- Data exchange: To exchange data between two components, one should component use a method to exchange the data.

In this paper, we map the structure of TinyOS and Contiki onto a component-based model and try to identify the interfaces and methods used inside the components. This study will help us in building generic component-based middleware that will work as a mediator between applications and the underlying network and will resolve the architectural differences between two heterogeneous networks.

III. The TinyOS and Contiki Programming as Models

In this section, we briefly discuss the characteristics of TinyOS and Contiki in order to give some background information about their main building blocks. We map the structure of the TinyOS and Contiki programming models onto a software engineering component-based model.

TinyOS was developed by the University of California [2]. It follows the component-based development model, which is mainly based on the event-driven programming model [3]. Every component in this OS is organized and written in the nesC language [4]. nesC supports and reflects the design of TinyOS: TinyOS uses a component-based model, and nesC supports and provides component-based programming. nesC provides two types of components: module components and configuration components. Modules provide the implementations of one or more interfaces of components, while configurations are used to assemble other components together, connecting the interfaces used by some components to the interfaces provided by others. This is called wiring. The TinyOS interfaces include sets of function names, parameters and their directions (commands and events). Components communicate with one another through commands and events. Each component declares some commands it uses and some events that it signals. A component consists of a set of tasks, events and command handlers. Commands are used to invoke the requests, and events are the notification messages sent to the caller.

Fig. 3 shows the component-based model of nesC. Here, three components are connected together: component A and B are of the module type, while component Ac is of the configuration type and will wire the components. As we know every component should have some interfaces, the nesC language also defines these interfaces for components. Every component defines two scopes: one for their specification (which contains the names of their interfaces) and one for their implementation. In the above example, component A provides an interface called Ainterface and uses an interface called Binterface.

Once the interfaces have been defined, the next step is the actual implementation, which is mostly encapsulated and defined by three elements inside the components: commands, events and tasks [3]. These three elements are simple C functions that are used because specific interfaces are required by every node to communicate with one another. Commands are used to query a component to perform an operation (like data

exchanging), to start some computation or to query a sensor. The event will decide the completion of requests, and tasks will be executed later by the scheduler [5].

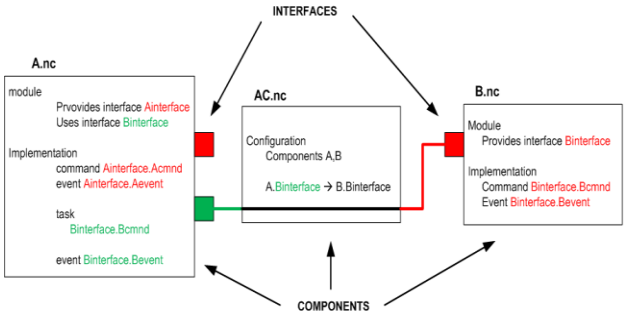

Figure 3. The TinyOS programming model

Contiki was developed by the Swedish Institute of Computer Sciences [6]. Contiki follows a modular architecture. It does not exactly follow the component-based model: it follows the hybrid model. Contiki programs can be written in a threaded fashion, where interprocess communication is enabled by passing messages through events. Contiki has an event-driven kernel, where pre-emptive multi-threading is implemented [7]. The Contiki architecture consists of the Contiki kernel, libraries, a program loader and processes [8]. As compared to a generic component-based model, we consider the Contiki kernel, libraries, program loader and processes as the components of this model. Fig. 4 shows the partitioning of core and loaded programs. All Contiki programs are processes. A process behaves like a component and should have some core functionalities and interfaces to interact with other components. A process is written as a simple C function with the variables and name of the process as parameters. Contiki supports event handling and executes processes based on some events. Contiki processes must have some interfaces that support these events. Processes in Contiki are started when the system boots or when a module that contains a process is loaded into the system. Processes run when something happens, such as a timer firing event or an external event occurring.

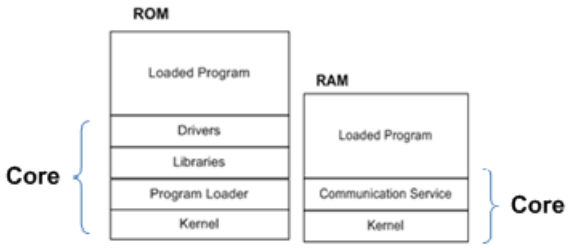

Figure 4. The TinyOS architecture [9]

A Contiki process provides the implementation that defines the information about the process, which is called the process control block. The process control block contains runtime information about the process and is stored in RAM. It contains information about each process, such as the state of the process, the name of the process and a pointer to the process thread. The process control block is only used by the kernel internally and cannot be directly accessed by processes. Fig. 5 shows a simple process control block.

```
struct process {
  struct process *next;
  const char *name;
  int (* thread)(struct pt *,
                 process_event_t,
                                 process_data_t);
  struct pt pt;
  unsigned char state, needspoll;
};
```

Figure 5. A process control block [10]

As there are lots of processes and they are scheduled according to events, each process in the process list has the six items of information shown in Table I. The process list is implemented using a simple single linked list.

TABLE I. THE CONTIKI PROCESS STRUCTURE

Field	Description
Next	Pointer to next process in process list.
name	Name of process.
thread	Pointer to protothread event handler.
Pt	Protothread corresponding to process.
State	Process state (None, Running, Called).
needspoll	Flag requesting that process check for data.

As we can see from the above table, the thread field contains a pointer to the event handler that is associated with the process. Every process has an event handler, as events are the main way through which Contiki processes communicate with one another. Here, the 'pt' contains the protothread state, and the 'state' field contains the current state of the process. When the process starts, the state of the process is in the RUNNING state. When a process is scheduled, it will change to the CALLED state; when the process is stopped, it will change to the EXIT state. Hence, a typical Contiki application consists of one or more processes and should contain the following blocks within the processes:

- Process declaration: Every process in the application should be defined by the PROCESS macro.
- Autostart list: This contains the list of processes that need to start automatically when the module is booted. This is managed by the AUTOSTART_PROCESS.
- Event handler: Each process in the module has a body, which functions as an event handler. The body, which starts with the PROCESS_THREAD macro, contains:
  - Initialization – Resources are allocated and variables are initialized.
  - Infinite loop – The waiting and processing of events take place.
  - Resource de-allocation – Resources are released at the end of the process thread (body).

```
#include "contiki.h"

PROCESS(example_process, "example process");

AUTOSTART_PROCESSES(&example_process);

PROCESS_THREAD(example_process, ev, data)
{
  static struct etimer timer;

  PROCESS_BEGIN();

  etimer_set(&timer, CLOCK_CONF_SECOND);

  while(1)
  {
    PROPROCESS_WAIT_EVENT(ev == PROCESS_EVENT_TIMER);
    printf("Hello, world\n");
  }

  PROCESS_END();
}
```

Figure 6. A simple contiki application

The first parameter is always the name of the process; the second parameter is 'ev', which is used to make the program respond to events that occur in the system. The third parameter is used to receive some data along with this event. The event parameter decides which process has to be executed. The rest of the instructions of the process will be written inside PROCESS_BEGIN ( ) and PROCESS_END ( ). There are two kinds of events in Contiki: synchronous and asynchronous. Synchronous events are like interprocedure calls, while asynchronous events are like the posting of tasks in TinyOS [5]. The process thread is a single protothread that is invoked from the process scheduler. The processes implement an event handler function and are only executed through these handlers [11]. Instead of using the simple event handlers, processes use protothreads. The advantage of using a protothread over an ordinary thread is that it does not require any additional stack. The protothread interfaces consists of four basic operations: initialization PT_INIT(), execution PT_BEGIN(), conditional blocking PT_WAIT_UNTIL() and exit PT_END(). These operations are simple C functions.

Fig. 6 shows a simple Contiki application that contains a single process. It waits for a timer event; when the timer expires, the message "Hello world" is displayed.

IV. Characterizing TinyOS and Contiki into a Component-Based Model

In this section, we try to map the programming models of Contiki and TinyOS onto a component-based model. As we discussed earlier, TinyOS and Contiki follow different architectures and programming models. In the previous section, we defined some of the characteristics of the component-based model; here, we identify the methods used inside TinyOS and Contiki and map them onto the component-based model's characteristics. We assume a simple scenario in which one node in a network tries to send some data to another node within the same network.

TABLE II. The Component-Based Model of Contiki

Characteristics	Method	Parameters
Initialization	PROCESS (example_process, "example process)" AUTOSTART_PROCESS(	*example_process* is the name of the process to be started.
State Control	SENSORS_ACTIVATE(button_sensor) ;	The sensor is activated after calling this function.
Communication	server_conn=udp_new(NULL, UIP_HTONS (UDP_CLIENT_PORT), NULL); udp_bind(server_conn,UIP_HTONS(UDP_SERVER_PORT));	The connection is created with *udp_new()*, it gets a local port number assigned automatically. *Udp_bind()* have following parameters. ***Conn*** *:* A pointer to the UDP connection that is to be bound. ***port:*** The port number in network byte order to which to bind the connection port
Data exchange	uip_udp_packet_send(c, data, len);	The message is delivered to other component using *uip_udp_packet_send* having following parameters: ***C:*** is the destination port i.e server port. ***Data*****:** is the actual data to send. ***Len*****:** is the length of the buffer (packet/data).

A. The Component-Based Model of Contiki

Here, we discuss how a Contiki process can be defined as a component of software engineering model. We map the methods used inside a Contiki process to the characteristics of the component-based model. The Contiki PROCESS_THREAD is the main function that is used inside every process. Table II shows the complete list of methods and signatures used by Contiki processes. A Contiki process always starts from the AUTOSTART_PROCESS, so whenever the hardware device that will be the destination of compiled code switches on, the process starts running. The sensor node uses the SENSORS_ACTIVATE method to change its current state. The communication is the binding of two components together so that they can communicate with each other. In a Contiki process, the binding is done through a binding method that will bind two components running on different nodes. The binding of components is necessary because a component cannot communicate with another component without any binding. If one node needs to exchange data with another, it needs to implement some method for sending data. Usually, the node generates an event based on a time stamp. After a certain time, the node sends data towards the other node using the following method:

*static void send_packet(void *ptr)*

The definition of this method invokes the server using

uip_udp_packet_sendto(client_conn,buf, strlen(buf),&server_ipaddr,UIP_HTONS (UDP_SERVER_PORT));

and delivers the message by using

uip_udp_packet_send(c, data, len);

where:

C: is the destination port (i.e. the server port).
Data: is the actual data to send.
Len: is the length of the buffer (packet/data).

B. The Component-Based Model of TinyOS

TinyOS has an event-driven, component-based model approach. In the TinyOS programming model, the programmer needs to write the specification of the components used and the components interactions among themselves. TinyOS uses 'configuration' and 'module' as components, thus we use them interchangeably to refer to software components. A module needs to define some interfaces at the beginning of the module file. A TinyOS component needs two types of interfaces: 'provides' and 'uses'. The configuration module connects the two interfaces and serves as a container of multiple modules. Hence, in general, every 'provides' interface of one module connects with a 'uses' interface of another module.

To implement a complete application, we must write three files. One is the header file, which contains declarations of the different variables to be used, the message structure to be used, etc. The second file should contain the module component code with its interfaces, events and tasks. The third file should be the configuration file, which binds the different module components together. Table III shows the complete list of methods and signatures used by TinyOS components.

TABLE III. THE COMPONENT-BASED MODEL OF TINYOS

Characteristics	Method	Parameters

Initialization	module ModuleName{
	uses interface Packet;
	uses interface AMSend;
	uses interface Receive;
	}
	event void Boot.booted()
		ModuleName is the name of the module to be started having different interfaces defined.
		Booted() function will be used to start the node.

State Control	event void AMControl.startDone(error_t err) event void AMControl.stopDone(error_t err)	If the radio is started successfully, *AMControl.startDone* will be called with the *error_t* parameter set to a value of SUCCESS. If, however, the radio does not start successfully, then it obviously cannot be used so we try again to start it.

Communication	implementation {
	...
	App.Packet -> AMSenderC;
	App.AMPacket -> AMSenderC;
	App.AMSend -> AMSenderC;
	App.AMControl -> ActiveMessageC;
	App.Receive -> AMReceiverC;
	}
		The binding between two components is established inside the configuration component. Every provided and used interface is bound together here so that they can exchange the data using these interfaces.

Data exchange	AMSend.send(AM_BROADCAST_ADDR, &pkt, sizeof(BlinkToRadioMsg)	The packet is sent to all nodes in radio range by specifying AM_BROADCAST_ADDR as the destination address. *pkt* is the actual packet and *sizeof* contains the size of packet

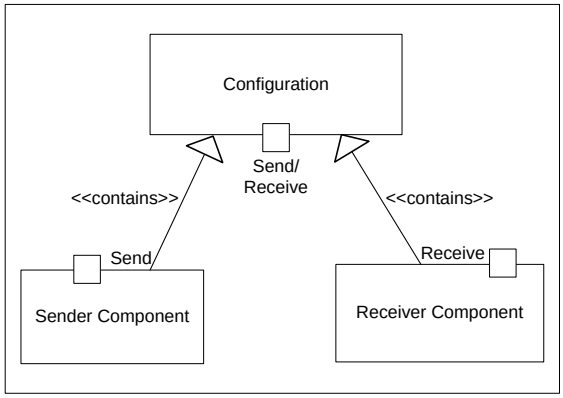

Figure 7. The TinyOS component-based model

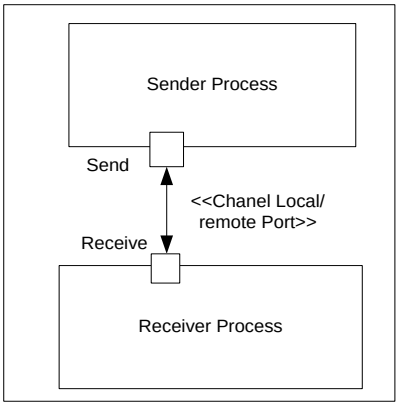

Figure 8. The contiki component-based model

The TinyOS module initializes with the module name and its interfaces; the initialization of the module should define the interfaces that it provides and uses. The node is started after calling the *booted ()* function. Once all the modules have initialized in different files, the state of the node needs to be checked. The *startdone ()* event performs this task and starts the radio successfully by calling this function. The binding between two components is established inside the configuration component. Every provided and used interface is bound together here so that they can exchange the data using these interfaces. The *send ()* method is used to send the packet to the other component. This method contains the sender address, the data and the size of the packet.

We have converted the actual TinyOS and Contiki models into a component-based software engineering model. Fig. 7 and Fig. 8 shows the general component-based model of TinyOS and Contiki. It is easy to communicate within a network that has the similar configuration. However, to communicate across networks, there should be some model that supports data interoperability between heterogeneous networks. The architecture of both nodes is different in terms of their software and hardware interfaces.

V. Conclusion and Future Work

In this paper, we have discussed the component-based model for software engineering. We have discussed the programming models of TinyOS and Contiki and mapped the structures of TinyOS and Contiki onto the component-based model. We have discussed in detail the different types of methods and signatures used in TinyOS and Contiki for component initialization, state control, communication and data exchange. In the next paper, we will discuss the challenges of component-based middleware and propose a complete architecture of this. Also, some experimentation results will be presented in the coming paper.

Acknowledgment

This work is funded by grant number 11-INF1674-10 from the Long-Term National Plan for Science, Technology and Innovation (LT-NPSTI), the King Abdul-Aziz City for Science and Technology (KACST), Kingdom of Saudi Arabia. We thank the Science and Technology Unit at Umm-Al-Qura University for their continued logistics support.

References

[1] (October 24, 2014). Component-Based software engineering. In Wikipedia, The Free Encyclopedia. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Component-based_software_engineering&oldid=632477662

[2] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System architecture directions for networked sensors," *Sigplan Not.*, vol. 35, no. 11, pp. 93–104, 2000.

[3] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, *et al*., "TinyOS: An operating system for sensor networks," in *Ambient Intelligence*, Berlin: Springer, 2005, pp. 115-148.

[4] D. Gay, P. Levis, R. V. Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

[5] Tobias Reusing Comparison of Operating Systems TinyOS and Contiki Sensor Nodes--Operation, Network and Application (SN), vol. 7, 2012.

[6] TinyOS Wiki - Platform Hardware. [Online]. Available: http://docs.tinyos.net /tinywiki/ index.php? title=Platform_Hardware&oldid=5648

[7] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - A lightweight and flexible operating system for tiny networked sensors," in *Proc. 29th Annual IEEE International Conference on Local Computer Networks*, 2004.

[8] Beginner's Guide to Crossbow Motes. [Online]. Available: http://www.pages.drexel.edu/~kws23/tutorials/motes/ motes.html

[9] M. O. Farooq and T. Kunz, "Operating systems for wireless sensor networks: A survey," *Sensors*, vol. 11, no. 6, pp. 5900-5930, 2011.

[10] Contiki-OS processes. [Online]. Available: https://github.com/contiki-os/contiki/wiki/Processes

[11] B. Dunkels and T. Gronvall, "Voigt contiki – a lightweight and flexible operating system for tiny networked sensors," in *Proc. First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, 2004.

Atif Naseer obtained his BS degree in Software Engineering from UET Taxila, Pakistan and MS degree in Software Engineering from National University of Sciences and Technology, Pakistan. He is acadamecian by profession and is currently serving as lecturer and researcher in Science and Technology Unit at Umm-al-Qura University, Makkah, Kingdom of Saudi Arabia. He has been attached with academia for over 8 years and has served as permanent faculty in many prestigious universities. His area of specialization are Software Engineering, Wireless Sensor Networks, Wireless Mesh Networks, Simulation and Modeling, Big Data Analysis, Geographic Information Systems. Contact him at the Science and Technology Unit, Umm-al-Qura University, Makkah Saudi Arabia; anahmed@uqu.edu.sa.

Basem Y. Alkazemi is an associate professor at Umm Al-Qura University (UQU) in Saudi Arabia under the school of computer science & Engineering. He obtained his PhD in 2009 from Newcastle University in U.K. His PhD topic was concerned with addressing the complexity of re-using open-source software components. Basem is currently holding the position of vice dean of Deanship of scientific research at UQU. He is a member in the IEEE, SIGSOFT-ACM, and SEI societies. His main research interests include software architectural patterns, software product lines, Aspect-oriented SE, SOA, and CBSE. Contact him at Dept. of Computer Engineering., Collage of Computer and Information Systems., Umm alqura University., Mecca., Saudi Arabia; bykazemi@uqu.edu.sa

Hossam Aldoobi is an MS student at Umm Alqura University. His research interest includes Geographic Information Systems, Wireless Sensor Networks, Cryptography and Information Security. He received a Bachelor degree in Computer Engineering from Umm Al-Qura University. Contact him at Dept. of Computer Engineering., Collage of Computer and Information Systems., Umm alqura University., Mecca., Saudi Arabia; Hidoobi@uqu.edu.sa