Department of Automation and Systems Technology

A-Stack

A TDMA Framework for Reliable, Real-time and High Data-rate Wireless Sensor Networks

Emre Ilke Cosar, Aamir Mahmood, Mikael Björkbom (editor)

Aalto University

SCIENCE + TECHNOLOGY

TECHNICAL REPORT

Aalto University publication series
SCIENCE + TECHNOLOGY 2/2012

A-Stack: A TDMA Framework for Reliable, Real-time and High Data-rate Wireless Sensor Networks

Emre Ilke Cosar, Aamir Mahmood, Mikael Björkbom (editor)

Aalto University
School of Electrical Engineering
Department of Automation and Systems Technology
Wireless Sensor Systems Group

Aalto University publication series
SCIENCE + **TECHNOLOGY** 2/2012

© Emre Ilke Cosar, Aamir Mahmood, Mikael Björkbom (editor)

ISBN 978-952-60-4467-5 (printed)
ISBN 978-952-60-4468-2 (pdf)
ISSN-L 1799-4896
ISSN 1799-4896 (printed)
ISSN 1799-490X (pdf)

Unigrafia Oy
Helsinki 2012

Finland

Prof. Heikki Koivo (supervisor)

NORDIC ECOLABEL
441 697
Printed matter

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Emre Ilke Cosar, Aamir Mahmood, Mikael Björkbom (editor)

Name of the publication
A-Stack: A TDMA Framework for Reliable, Real-time and High Data-rate Wireless Sensor Networks

Publisher Aalto University

Unit Department of Automation and Systems Technology

Series Aalto University publication series SCIENCE + TECHNOLOGY 2/2012

Field of research Wireless Sensor Networks

Abstract
The reduced size, power consumption and cost of wireless sensors make them an exciting technology for many monitoring and control applications. However, developing reliable, real-time and high data-rate applications is challenging due to time-variations and interference in wireless channels and the medium access delays. For high data-rate and real-time applications, time division multiple access (TDMA) based medium access approach performs better as compared to carrier sense multiple access (CSMA) based approach. On the other hand, implementation of TDMA in resource constrained wireless nodes requires difficult design decisions.

This document presents A-Stack, a real-time protocol stack for time synchronized, multi-channel and slotted communication in multi-hop wireless networks. The stack is developed to meet the reliability and accuracy requirements of real-time applications such as wireless automation and wireless structural health monitoring. A-Stack provides a flexible development environment for such applications by ensuring deterministic reliability and latency. It includes MAC, routing and time-synchronization protocols as well as a node-joining algorithm. The stack is further supplemented with PC tools for optimizing the network as per the target application for easy prototyping. This document explains the design and operational aspects of A-Stack. Various deployment scenarios as well as long term system and communication reliability tests are presented in the document.

Keywords wireless sensor networks, real-time, protocol stack, time-sycnhronization, TDMA, SHM,

ISBN (printed) 978-952-60-4467-5 **ISBN (pdf)** 978-952-60-4468-2

ISSN-L 1799-4896 **ISSN (printed)** 1799-4896 **ISSN (pdf)** 1799-490X

Location of publisher Espoo **Location of printing** Helsinki **Year** 2012

Pages 73

Preface

This report presents A-Stack, a communication protocol stack for reliable, real-time and high data-rate wireless sensor networks. The work is done in Aalto University School of Electrical Engineering under Intelligent Structural Health Monitoring (ISMO) and Reliable and Real-Time Wireless Automation (RELA) projects. A-Stack development started in 2010 on top of an IEEE 802.15.4 stack. The main reason for developing A-Stack was the lack of software that would support long term real-time WSN deployments which require accurate time-synchronization and high data-rates. In order to be able to use the same software platform in a multitude of scenarios, special emphasis was put on the flexibility and ease-of-use.

Aamir Mahmood and Emre Ilke Cosar developed time synchronized event generation and handling on top of a MAC layer synchronization protocol (μ-Sync). Then, Emre extended this work to cover initialization, routing, MAC, multi-hop and multi-channel networks, scheduling and power control. Erkka Mutanen modified a tool that creates network schedules for A-Stack. Development work is supervised by Prof. Heikki Koivo. This report gives a detailed description of the stack, operation, internal issues, and usage as well as test cases and example configurations. The report is written by Emre and Aamir and edited by Mikael Björkbom.

Espoo, December 28, 2011,

Emre Ilke Cosar, Aamir Mahmood and Mikael Björkbom

Contents

1. Introduction

1.1 Background

A-Stack is a framework for developing reliable real-time Wireless Sensor Network (WSN) applications and networking algorithms. A-Stack incorporates multi-channel Time Division Multiple Access (TDMA), dynamic channel hopping, global time-synchronization, routing, network configuration and system reliability tools. These are implemented on a real-time operating system, which enables easy development of application and other layers.

This report presents the technical details and design decisions of A-stack. It is written to make the user understand the mechanisms in A-Stack. A widely used term in the document, also in the source code, is WCP, which refers to Wireless Communication Protocol. A-Stack started as a communication protocol development project but then extended to a stack through which several algorithms and applications can be implemented. In order to have full understanding of A-stack and its operation, it is advised to follow the source code together with this document.

Researchers on the WSN field need an open source development environment that will enable easy access to network and sensor data. Furthermore, an open and reliable software platform with good documentation has the potential to make development on network layer, data link layer, other services or applications affordable.

A-stack is developed considering the requirements of wireless automation and structural health monitoring applications. In the wireless automation field, the target is to collect data with high reliability and low latency in real-time without comprising the low power operation. Wireless channels are non-deterministic and time-varying and suffer from the interference caused by coexisting wireless networks. These characteristics can easily result in packet losses and increased packet delays and energy consumption. Therefore, it is very hard to ensure high communication reliability and deterministic operation by using wireless nodes. Carrier sense multiple access (CSMA) mechanisms have shown to be inefficient in comparison with time division multiple access (TDMA) based approaches when communication reliability and latencies are concerned [1]. Furthermore, TDMA approaches perform better in terms of energy efficiency for real-time control applications by duty-cycling the nodes [2].

Structural health monitoring (SHM) applications require collection of high fidelity data. Time-synchronized sampling is thus crucial for wireless SHM. In many cases, SHM applications require periodic data collection from the network to a common sink. Long data collection times result in higher power consumption and lower monitoring frequencies and make the WSN unfeasible. Time-slotted networks allow better accuracy for allocating the limited bandwidth among the nodes in the network and thus increase throughput. TDMA networks also enable energy savings since the nodes can sleep when there is nothing to send or receive. Reliability of the monitoring system is another critical element, i.e. the system should be functioning without problems for months and even years, in a real-world scenario.

In general, requirements set for wireless monitoring applications are diverse in their nature and it is not possible to provide generic solutions that will fit in every situation and sensor network [3]. Instead, an environment that enables easy development, prototyping and deployment is needed. A-Stack aims at providing such an open source environment for developing time-synchronized, slotted and multi-channel communication that does not compromise reliability, low power operation and usability.

1.2 Related Work

Several communication protocol stack designs has in the past been presented and evaluated. These designs consider OS, MAC and tailored applications for specific optimization goals. In this section, we revise the latest relevant approaches. Further review of previous communication protocols can be found in [2] and [4].

There are several operating systems designed especially for WSN applications. Notable of these are TinyOS [5] and Contiki [6]. Even though it is possible to implement time-critical tasks by using these operating systems, they are mainly designed for event-based operation. We use a real-time operating system, FreeRTOS [7], which allows pre-emptive scheduling and prioritization between tasks. Such an operating system allows implementation of time-critical and best effort tasks separately, which reduces code development effort.

SlotOS [9] is a programming approach that uses slotted programming for temporal decoupling of the different tasks of a sensor node such that at any time at most one task is active. With this approach, tasks can be implemented as independent software modules which simplify coding and enable code reuse.

The WirelessHART protocol [8] is an industrial standard for process and automation applications. It utilizes Time Synchronized Mesh Protocol (TSMP) [1] which combines TDMA with 10 $msec$ slots, channel hopping, channel blacklisting, and industry standard AES-128 ciphers and keys. A centralized network manager is responsible for making the routes and communication schedules. Commercial products that employ this standard are on the market. However, the network managers developed by companies are closed and the operation is generic which does not allow optimization for the task in hand.

GinMAC uses TDMA, off-line dimensioning, reliability control mechanisms and topology control mechanisms to ensure timely and reliable data delivery [10]. PIP is a connection-oriented, TDMA-based, multi-channel and centralized bulk transfer protocol [11]. In [2], transmission pipelining and multiple transceivers at the controller are used for achieving low latency requirements of hard-real time discrete control applications. TREnD protocol [4], aims at dynamic adaptation of protocol parameters for optimizing energy efficiency given a set of reliability and latency constraints. TREnD includes a hybrid TDMA/CSMA MAC, routing, data aggregation and duty cycling for single channel scenarios.

1.3 Original Contributions

The original contribution of our work is a development framework for reliable real-time WSN applications and protocols. A-Stack incorporates multi-channel TDMA, global time-synchronization, routing, network configuration, and system reliability tools. These are implemented on a real-time operating system, which enables easy development of application and other layers. In our approach emphasis is put on modularity and providing tools for realizing a multitude of real-time and reliable operation scenarios in practice, instead of creating an optimized protocol for one scenario.

An important characteristic is that the networks and schedules can be optimized for an application or protocol and can be tested in practice over long periods of time without need for extra effort. Channels can be independently assigned to the time-slots and can be changed in order to adapt to the varying channel conditions. Time-slot length of a full length packet is 8 $msec$ which can be reduced for smaller packets. In the stack, IEEE 802.15.4 addressees are used, and unicast packets are acknowledged within one time-slot. Nodes are time-synchronized throughout the operation. They can join and re-join the network at any time. Another important advantage of the stack is that the code is open, which enables conducting research on network, data link, service and application levels. The capabilities of A-Stack are validated through tests and deployments presented in this document. A brief description of the stack and test results can be found in [12].

1.4 Structure

This document is organized as follows. Chapter 2 presents the general architecture of the stack. System design is described in Chapter 3. Time synchronization is given in Chapter 4. Operation topics are given in Chapter 5. System set-up, example configurations as well as tests and deployments are explained in Chapter 6. Finally, conclusions are given in Chapter 7 along with the known issues and future development.

2. General Architecture

In this chapter, we discuss the main building blocks of A-Stack architecture. The radio transceiver, microcontroller unit (MCU), FreeRTOS real time kernel, radio and timer interrupts, three stack tasks (Medium Access Control (MAC), Packet Manager, and Service Manager) and Application task(s) running on the operating system constitute the main building blocks in A-Stack. Figure 2.1 shows these blocks in A-Stack.

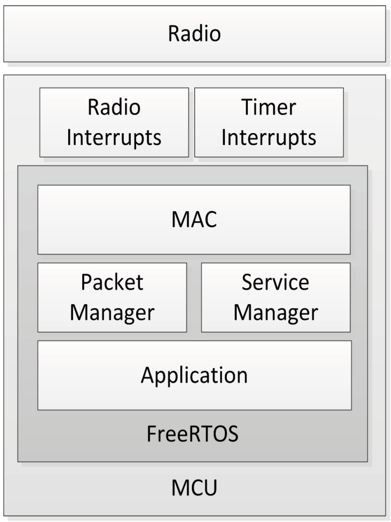


Figure 2.1. Building blocks of A-Stack

Hardware drivers, data structures and some of the communication functions used in the stack are based on NanoStack^{TM} , a 6LoWPAN protocol stack for wireless sensing and control using low power devices [13]. NanoStack started as an open-source development but as of January 2009 the source code was closed and stopped to be actively developed and maintained as an open-source project.

2.1 Hardware Components

The stack is implemented on Sensinode Micro.2420 nodes comprising of a TI MSP430 core and a Chipcon CC2420 transceiver, operating on the 2.4 GHz band, having 250 kbps data-rate. The Microcontroller unit (MCU) has 10 kB RAM and 256 kB flash memory. The nodes are running FreeRTOS, a real-time embedded operating system that supports both preemptive and cooperative operation. Even though the current implementation of A-Stack is on this hardware, it can be easily extended to other hardware platforms in the future.

2.2 Operating System

A-Stack runs on FreeRTOS real-time kernel. FreeRTOS is an open-source, real-time operating system designed for embedded systems. Preemptive multitasking in FreeRTOS enables real-time operation by task switching that occur when a higher priority task is triggered. This characteristic of the operating system (OS) ensures the time-critical tasks such as handling of time-slots are done correctly within their critical time span. FreeRTOS employs queues and semaphores for inter-task communication.

2.3 A-Stack Scheduler

It is useful to define here what is meant by a schedule and a scheduler. A schedule in this context refers to a set of timer events within a network-wide super frame. These events indicate the start or stop of a specific task (e.g. transmission and reception slots) within the super frame. Every node has a specific schedule. Timer interrupts, or events, in this schedule are the heart of the operation in A-Stack. Their deterministic generation ensures the continuity of the operation, and keeps the network schedule running with transmission and reception slots. Here, "scheduler" refers to the process of generating timer interrupts. When the scheduler is said to be running, it means that the timer interrupts are generated periodically. When a timer interrupt occurs, the scheduler creates an event to be processed and sets a new interrupt for the next occurrence based on the timer event duration. The scheduler is activated in a node after all the configuration packets are received and the node is time-synchronized with the network.

2.4 Tasks in A-Stack

Table 2.1 shows the tasks, their priorities and a brief description of their responsibilities in A-Stack. The stack is designed to support time-slotted communication, which brings the necessity to handle time-critical events within deterministic times. This is achieved by prioritizing the stack tasks.

Table 2.1. Tasks in A-Stack

Task	Priority	Responsibility
MAC	Highest	Communication and timer events handling.
Packet Manager	High	Incoming and outgoing packets handling, routing.
Service Manager	Medium	Initialization, node joining and reliability.
Application	Low-High (App. dependent)	Send/Receive packets, data acquisition, control etc.

2.4.1 MAC Task

When A-Stack scheduler is active, the MAC task is responsible for handling radio and timer interrupts. Once a packet is received in the radio, an interrupt is triggered in the MCU. The handler function of this interrupt creates an event to be handled in the MAC task and initiates a task switch in the operating system. MAC task handles the packet right after it is received since it is the highest priority task. Similarly, when a timer event occurs, a timer interrupt is generated and the MAC task is activated. The MAC task then handles the timer event by changing channels, turning on/off the radio or transmitting a packet from a queue, depending on the timer event parameters.

2.4.2 Packet Manager

The packet manager is responsible for managing the packets within the node. When a transmission is successful, it erases the packet from the transmission queue, and when a packet is received it handles the packet according to the packet type. Packet routing operations are also done in this task.

2.4.3 Service Manager

The service manager is responsible for providing additional services that might be required in the node operation. In the current implementation, it is responsible for node joining and checking whether the node is in the network and hence increasing the network reliability. If a node restarts or loses synchronization, the service manager helps the node to recover and re-join the network.

2.4.4 Application

The application task is the place where abstraction from the stack is obtained. A developer can use one or more application tasks for the code required to run the application. Data packets are formed and added to the transmission queue. Additionally, the incoming application data packets are handled in this task after being processed in the packet manager.

2.5 Interrupts

There are two sources of interrupts in A-Stack: radio interrupts and timer interrupts. Radio interrupts are generated whenever a packet is received in the radio module and this interrupt notifies the main microcontroller to fetch the data from the radio module. Timer interrupts are generated at the timeslot boundaries. The MAC task is responsible of handling these two types of interrupts. The interrupts are handled in their respective interrupt service routines (isr's). Once an interrupt occurs, the operating system tasks are halted and these isr's are invoked. These isr's should be a brief portions of code in order not to interfere with the operating system tasks. That is why isr's are designed to do the minimum requirements of the interrupt.

2.6 Supplementary Tools

A-Stack is supplemented with tools for easy prototyping and reliable operation. A sink node to PC communication tool interfaces the network and the user/network manager. A simple node discovery tool records the node-ids and MAC addresses. This information, together with the routing information, is then used to create the schedules. A schedule creation tool creates the schedule of the network. This tool can use several algorithms for scheduling based on the application requirements. We have implemented a default scheduling algorithm which assigns dissemination and convergecast slots to all the nodes in the network. As an alternative, we have integrated the WirelessTools software, developed by KTH [14] [15], into A-Stack schedule generation process. Finally, node-joining and re-joining tool ensures the network is reliable throughout the operation by handling node-join requests, re-joining requests, changing channels, changing duty cycles of the network etc.

3. A-Stack Design

In this chapter, design decisions taken for A-Stack are introduced in detail. Data stuctures, system states, timer events and packet handling are explained.

3.1 Stack Structures

This section presents the structures used in A-Stack.

3.1.1 Main Data Structure

A-Stack uses the same data structure as NanoStack, *buffer_t*. The main advantage of this structure is that it allows the packets to be passed within tasks without actually moving the data, but by moving only the pointer to this data. OS queues have limited storage and filling this storage with the actual data to be transferred would consume large amounts of space, instead NanoStack's solution provides an efficient structure to pass the data within resource limited embedded systems. A detailed explanation of the main data structure can be found in [16].

A pool of *buffer_t* instances are created when the stack is initialized. Allocation of this data structure is done through a ring buffer acting as a pool for the *buffer_t* instances. Front end functions *stack_buffer_get()* and *stack_buffer_free()* are used to take and free the buffers in the pool. For efficient use of the stack, it is important to understand the mechanism of the stack buffers, this is why reading related sections in [16] is strongly encouraged. Figure 3.1 shows the ring buffer for *buffer_t* instances.

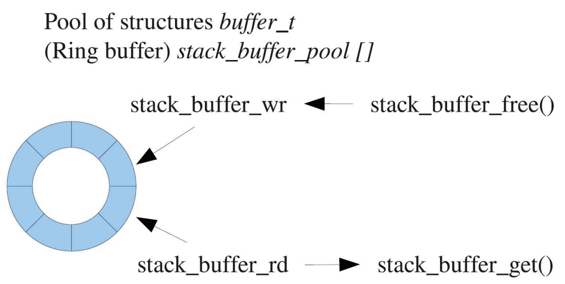


Figure 3.1. Ring buffer for *buffer_t* instances [16]

3.1.2 Communication Pair Structure

A communication pair is a neighbor of a node, to which it may send and receive packets. Every node in operation has communication pairs (cp) and data to send to these communication pairs. The *wcp_cp_t* structure (Fig. 3.2) is used to store wireless communication information needed for data exchange between the nodes.

```
/** communication pair structure, Includes the circular buffer info*/
typedef struct tagWCPcp
{
uint8_t cp_id; /*id of the comm pair*/
uint8_t tx_fail; /*failed transmissions to the compair*/
uint8_t read_pointer; /*indice number of last read buffer*/
uint8_t write_pointer; /*indice number of last written buffer*/
uint8_t no_of_buffers; /*number of buffers in the circular buffer*/
sockaddr_t dst; /*destination address*/
buffer_t *buffer_index[MAX_CP_TX_BUFFERS]; /*index of buffer pointers*/
}wcp_cp_t;
typedef wcp_cp_t* wcp_cpHANDLE;
```

Figure 3.2. Communication Pair Structure

cp_id represents the id of the corresponding communication pair. *tx_fail* is a parameter used for tracking how many consecutive transmissions have failed of the corresponding communication pair. The *tx_fail* parameter is used when managing buffers during operation.

Data transfer in the stack is done by using the *buffer_t* instances as explained in the previous section. Every *wcp_cp_t* object has a circular buffer of *buffer_t* instances. This circular buffer is used to store the buffers to be transmitted when the right communication slot occurs. *read_pointer*, *write_pointer*, *no_of_buffers* are the parameters used to manage the *buffer_index[MAX_CP_TX_BUFFERS]*.

dst is the destination address of the communication pair and its stored in a *sockaddr_t* structure, which is a native structure of NanoStack.

3.1.3 Timer Event Structure

In A-Stack, the *wcp_timer_event_t* structure (see Fig. 3.3) is used for storing the information regarding the timer events in the schedule.

```
/** definition of wcp timer event types , handled in mac */
typedef struct wcp_timer_event_t
{
mac_event_id_t type; /*Tx slot start/stop etc.*/
uint8_t duration;
unsigned portCHAR radio_channel;
uint8_t cp;
}wcp_timer_event_t;
typedef wcp_timer_event_t* wcp_timer_eventHANDLE;
```

Figure 3.3. Timer Event Structure

Timer interrupts are used to trigger timer events and hence time slots in the network schedule. Whenever a timer interrupt occurs, an event is sent to the MAC layer based on the *type* parameter. The *duration* parameter is used to set up the next timer event, right after one timer event occurs. The *radio_channel* parameter shows which *radio channel* will be used during the time slot. The *cp* parameter indicates the communication pair targeted within the time slot.

Note that the *duration* parameter is defined to be unsigned 8 bit. It can take values from 1 to 255, which then refers to slot lengths of 1 to 255 $msec$. However, in reality, length of a timeslot should not be larger than 65535 $msec$. This number is the maximum value a 16 bit counter can count up to when clock is running at 1 MHz. Capture and compare register of the MCU timer is also 16 bits, which means that longest interval between two timer interrupts is 65535 $msec$. For regular communication slots, 10 $msec$ is enough for any transmission and reception, however for idle slots (slots during which radio is in an idle state) slot length can be larger than 65 $msec$ depending on the schedule. Thus a separate mechanism is needed for them. In an idle slot, radio is idle this is why *cp* parameter is not needed. This parameter is used to store how many times a 65 $msec$ is needed for the idle slot. Idle slot handler in MAC task triggers 65 $msec$ events as many as the *cp* parameter. Maximum idle slot length is 65x256 $msec$ or 16.64 sec. When forming the schedules, consecutive idle slots are combined in order to decrease the maximum number of events thus reducing the memory usage.

3.1.4 Main Operation Structure

The main structure of the stack is *wcp_scheduler_t* (see Fig. 3.4). It is responsible for keeping all information regarding the operation, such as communication pairs and timer events information.

```c
/** MAIN STRUCTURE of the STACK*/
typedef struct tagWCPobj
{
uint8_t my_id; /*!< //MY ID*/
uint8_t current_timer_event; /*!< //index number of the current event*/
uint16_t frame_counter; /*!< //counts the number of frames*/
uint8_t idle_counter; /*!< //counts the number of idle frames*/
uint8_t idle_original; /*!< //stores the original number of idle frames*/
uint8_t idle_first; /*!< //first occurrence of idle counter*/
uint8_t adv_freq; /*!< //number of frames after an advertisement beacon is sent*/
uint8_t adv_check_freq; /*!< //parameter for checking the advertisement reception*/
uint8_t frame_freq; /*!< //number of frames after TX or RX slots will work (1 in X frames will work)*/
uint8_t frame_active; /*!< //indicates whether the frame is active or not */
uint8_t adv_resp_time; /*!< //ms after which a node will reply to an advertisement*/
uint8_t adv_rx; /*!< //indicates that an advertisement is received, so no need to run node join
sequence*/
uint8_t service_buffers_ready; /*!< //check whether service buffers (sync and advertise buffers) are
ready*/
uint8_t number_of_timer_events; /*!< //total number of events generated so far (to prevent unnecessary
malloc*/
uint8_t no_of_current_timer_events; /*!< //total number of current events that is running now*/
uint8_t advertised_when_off; /*!< //shows that advertisements received when scheduler was off, then
start node join*/
uint8_t route_back[MAX_HOPS-1]; /*!< //shows the route back to the sink from the node, 0 for sink.*/
uint8_t safe_operation; /*!< //indicates that scheduler is running and the frame counter is taken
//0 in the beginning, 1 if wcp is running(not off and not join) */
wcp_timer_states_t state; /*!< timer state */
unsigned portCHAR current_channel; /*!< current channel */
uint8_t change_channels; /*!< change channel activate */
uint8_t new_channel; /*!< new channel*/
uint8_t re_tx; /*!< this variable is used to manage not sent buffers, how many times to try in
different slots*/
uint8_t number_of_cp; /*!< number of communication pairs in use */
uint8_t change_frame_freq;
uint8_t new_frame_freq;
uint16_t frame_to_change_frame_freq;
uint8_t start_sim_active;
uint16_t frame_to_start;
xSemaphoreHandle start_sim_smphr; /*!<used for starting simultaneous action in network */
xQueueHandle app_queue; /*!< //this queue is for handling packets in
application layer */
xQueueHandle service_queue; /*!< */
wcp_cpHANDLE cp_index[MAX_CP]; /*!< */
wcp_timer_eventHANDLE timer_event_index[MAX_TIMER_EVENTS]; /*!< //array of pointers */
}wcp_scheduler_t;
typedef volatile wcp_scheduler_t* wcp_schedularHANDLE;
```

Figure 3.4. Main Operation Structure of A-Stack

my_id is the ID number of the node. *current_timer_event* is the index number of the current event. This number is used to point at the corresponding event in *timer_event_index[MAX_TIMER_EVENTS]* array.

idle_counter, *idle_original*, *idle_first* are used to manage idle time-slots as explained in Section 3.1.3.

adv_freq indicates the frequency of advertisement beacons. When it is set to 10 for example, an advertisement beacon is sent in every 10 superframes. A "superframe" in this case refers to a period that is equal to the duration of all the timer events.

adv_check_freq determines when a node disconnects if it stops receiving advertisements. It is calculated in PC and transferred to the nodes during initialization. As an example, if this variable is 2, a node will go to the node joining state after it sees that an advertisement is not received for 2 times consecutively . This check is done in service manager task and the period of this check depends also on *SERVICE_TASK_DELAY* defined in Section 5.4.3.

frame_freq determines how often a frame will be active. When it is 1, all the frames are active frames and reception and transmission can be done in that frame, but when this is for example 5, 1 every 5 frames is active. The change in frame frequency is used to decrease the power consumption.

frame_active parameter is used in MAC layer to check whether the node is in an active frame.

adv_rx variable is used to track whether advertisement beacons are received within their corresponding time slot. It is modified in MAC task, *MAC_RX_MES* event. It is controlled in Service Task in order to determine whether a node is within the network, if service task notices that a node is not receiving advertisements regularly, it will change the operation state to *NODE_JOIN*.

Every node creates *wcp_advertise_buf* once. This is the buffer used when transmitting advertisement beacons. Note that synchronization information is also added to this buffer. In order to prevent creating the same buffer over and over, *wcp_advertise_buf* is created once and used throughout the operation. After these are created for the first time, *service_buffers_ready* variable is set to 1 to ensure they are not created again in case of re-joining to the network.

number_of_timer_events variable is used to keep track of the maximum number of timer events generated in the node. Every time a timer event is generated, *malloc()* function is called and a portion of the memory is allocated for that event. This variable is controlled in order to prevent unnecessary memory allocation, for example a schedule update.

no_of_current_timer_events variable stores the number of timer events in the scheduler. This variable is used within the timer event interrupt service routine.

advertised_when_off parameter is used to change the state of the scheduler from *WCP_OFF* to *WCP_JOIN*. When a node re-starts, the initial state is *WCP_OFF* and NanoStack is running. Once an advertisement targeted to the node is received, this variable is set to 1 in MAC task. Actual state change occurs in the service manager task.

route_back[MAX_HOPS-1] array stores the route to be used when the node wants to send a packet back to the sink.

safe_operation indicates that the scheduler is running and frame numbers are in sync with the rest of the network. It does not refer to a safe mode, but it represents that the node is working fine and any operation can be done safely in upper layers.

state variable shows the state of the scheduler. These can be *WCP_OFF*, *WCP_JOIN* or other timer states. They will be explained in Section 3.2.1.

current_channel parameter shows the current channel used by the node. This variable is updated in every time-slot. The variables *change_channels* and *new_channel* are used to change the channel for transmission (*WCP_ TX_SLOT_START*) and reception (*WCP_RX_ SLOT_START*) slots in the node. Note that every time-slot can have a different channel and this change occurs in the beginning of that particular slot. *change_channels* in this case refers to what happens when the channels of the time-slots are updated. The actual change takes place in the MAC task is in the *WCP_FRAME_SLOT_START* event.

re_tx parameter indicates how many times a packet, which does not receive an ACK, is going to be transmitted before it is dropped. These retransmissions take place when the next time a transmission slot occurs for the same communication pair. For example, if *re_tx* is 1, a packet is dropped even though an ACK is not received after the first trial, but if it is 2, it would be transmitted another time.

number_of_cp defines the number of active communication pairs.

change_frame_freq indicates a request for frame frequency change is requested. This variable is checked in MAC task.

new_frame_freq indicates the new frequency to be set, whereas the variable *frame_to_change_frame_freq* indicates at which frame the change is going to take place. This is required in order to synchronize the entire network when a major change is to occur.

start_sim_active a simultaneous start of an operation (sampling, actuation, etc.) in the network. MAC layer handles this start by checking *frame_to_start* variable and eventually giving the semaphore, *start_sim_smphr*, when the time comes. This semaphore can be used in the application to trigger a certain operation.

app_queue is the queue used for *WCP_DATA* packets that are sent to the application layer by packet manager.

service_queue is the queue used for handling service messages. These are the messages received when the node is in *WCP_JOIN* state. This queue is handled in *service_manager* task.

Besides the two queues explained above, there is another queue that is actively used during the operation. This is *wcp_queue*, and it is handled in the *packet_manager* task. When the scheduler is running, all the received messages come here, and action is taken based on the packet type.

3.2 A-Stack States and Events

This section presents the states and events used in the A-Stack.

3.2.1 Timer States

Timer states are the main indicator of the scheduler state and they have a major role on the communication settings at different phases of the operation.

```
/** schedular states, updated in mac when changing events,
OFF means schedular is not running, JOIN means
node tries to join*/
typedef enum
{
RX_SLOT = 0, //no cca
TX_SLOT = 1, //no cca
RX_TX_SLOT = 2, // cca : in service slots and in frame slots
WCP_WAIT_FOR_ACK = 3,
WAIT_FOR_SLOT = 4,
WCP_OFF = 5,
WCP_JOIN = 6,
}wcp_timer_states_t
```

Figure 3.5. Timer States

Once a node is powered up, its initial timer state is *WCP_OFF*. In this state, NanoStack is active and packets are handled as explained in [16]. In this state the communication can be either broadcast or unicast. Note also that the default MAC implementation of NanoStack is IEEE.802.15.4 MAC, which utilizes clear channel assessment (CCA) when transmitting a packet. In this MAC implementation, a node may back-off and try re-transmitting a packet after a pseudo-random time interval depending on the channel conditions. In default NanoStack implementation, several packet headers and footers are formed after a packet is sent to the stack modules and these are memory operations including moving a certain amount of data. Furthermore packets move between several modules. Because of above mentioned characteristics, it is hard to estimate at which instant a packet is transmitted through the radio when NanoStack is active.

If a node starts to receive advertisements targeted to it when it is in *WCP_OFF* state, it will change its state to *WCP_JOIN*. In this state packets that are received in the MAC layer (or MAC task) are forwarded to the *service_queue*, or *service_task* (note that this task is indicated as *WCP_Service_task* in the source code). When a node is in this state, it receives and transmits only broadcast messages. These messages employ CCA and include IEEE802.15.4 headers, however, in case of a no clear channel, there are no re-transmissions.

The rest of the timer states do not employ back-off after a not clear channel. Time synchronized operation ensures that all the nodes receive and transmit at their corresponding time-slots. When state is not *WCP_OFF* and *WCP_JOIN*, packets, as instances of *buffer_t*, are formed and added to the corresponding communication pair buffer queue prior to their transmission. Packet formation includes adding the addresses and MAC headers. Once a transmission slot comes, these ready packets are transmitted without any further delay. This is an important characteristic of the A-Stack: packets are not modified any more in the MAC layer. This ensures that the transmission takes place within the potentially short time slots.

An important issue in forming communication packets is MAC sequence update. Normally, when the timer state is *WCP_OFF*, MAC sequence is added to the frame and the sequence is updated when the function *rf_802_15_4_create_mac_frame()*, is called. In other states, this function creates the frames but does not update the sequence. The correct sequence is added to the frame and then updated when *wcp_mac_tx_buf()* function is called. The reason for this difference in implementation is that, when the scheduler is running, the packets are transmitted not based on the time they are created, but based on the time slots. This is why, the MAC sequence number is not added until the last moment.

RX_SLOT and *TX_SLOT* states indicate that the communication is unicast and no clear channel assessment is done. In *RX_SLOTs*, radio is on, in *TX_SLOTs* radio is enabled if there is a packet to be transmitted. In *RX_TX_SLOTs* CCA is done, and if channel is busy no transmission takes place. Radio is always on in this timer state.

WAIT_FOR_ACK state indicates that a unicast transmission was done, and a timer is set in MAC. Within this time if an ACK packet is received, timer is stopped, the corresponding transmitted packet is removed from the buffer queue of the corresponding communication pair, and the timer is stopped. If an ACK is received when the timer is in another state, it is not considered to be valid.

WAIT_FOR_SLOT state indicates that the timer is in an idle state waiting for a communication slot to occur. In this state, the radio is off. Table 3.1 shows the Timer states and their functionalities.

Table 3.1. Timer States and their Functionalities.

Timer State	**Radio**	**CCA check**	**Transmission functions**	**Mac sequence update**	**Re-tx if channel busy**	**Active Stack**
WCP_OFF	ON by default	Yes	mac_tx_buf(), rf_write()	rf_802_15_4_create_mac_frame()	Yes	NanoStack
WCP_JOIN	ON	Yes	wcp_mac_tx_buf(), rf_write_no_cca()	wcp_mac_tx_buf()	No	A-Stack (service manager)
RX_SLOT	ON	-	-	-	-	A-Stack
TX_SLOT	ON if tx	No	wcp_mac_tx_buf(), rf_write_no_cca()	wcp_mac_tx_buf()	Packet manager decides	A-Stack
WAIT_FOR_ACK	-	-	-	-	-	A-Stack
RX_SLOT	OFF	-	-	-	-	A-Stack
RX_TX_SLOT	ON	Yes	wcp_mac_tx_buf(), rf_write()	wcp_mac_tx_buf()	Packet manager decides	A-Stack

3.2.2 Event Types and Their Handling

Table 3.2 shows the MAC events and their functionalities. Note that these events start with "WCP". This is done in order to distinguish them from native event types in NanoStack.

MAC_RX_MES is the event generated by the radio interrupt service routine when a packet is received. Packet reception in MAC is further explained in Section 3.3.3.

WCP_TX_SLOT_START event occurs when a transmission slot starts. Timer state of the scheduler is changed to *TX_SLOT*. Radio channel is changed if necessary. Buffer queue of the communication pair, which is assigned to the slot, is checked. If a packet is ready, it is transmitted. Otherwise radio is turned off.

WCP_RX_SLOT_START event occurs when a reception slot starts. Timer state of the scheduler is changed to *RX_SLOT*. Radio channel is changed if necessary.

Table 3.2. MAC Events and Their Functionalities

Event type	Timer State	Source	Functionality
MAC_RX_MES	RX_SLOT	Radio ISR	A radio packet is received.
WCP_TX_SLOT_START (TX)	TX_SLOT	Timer ISR	Change radio channel. Transmit a packet to the communication pair, if there is any.
WCP_RX_SLOT_START (RX)	RX_SLOT	Timer ISR	Change radio channel, wait for a packet.
WCP_FRAME_SLOT_START (FR)	RX_TX_SLOT	Timer ISR	Indicates a global frame starts. Update frame counter. Update schedule if there is any changes.
WCP_RE_TX	TX_SLOT	Packet Mngr.	Re-transmission within one time slot.
WCP_IDLE_SLOT (IDLE)	WAIT_FOR_SLOT	Timer ISR	Wait for next event.
WCP_SERVICE_TX_SLOT (S_T)	TX_SLOT	Timer ISR	Change radio channel, broadcast service messages or advertisements.
WCP_SERVICE_RX_SLOT (S_R)	RX_TX_SLOT	Timer ISR	Change radio channel.
WCP_SERVICE_IDLE (S_I)	WAIT_FOR_SLOT	Timer ISR	Wait for next event.

WCP_FRAME_SLOT_START event occurs when a frame starts. The variable *frame_counter* is incremented at the beginning of this event. All updates related to the schedule should be done within this slot. In current implementation the channels and frame frequency are updated, and simultaneous start command is given in this slot. Timer state of the scheduler is *RX_TX_SLOT*.

WCP_RE_TX event is generated by packet manager when a retransmission is needed within one time slot. This should be used with care and only when the time slots are long enough.

MAC_SYNC_MODULE is actually not an event, but it is an indicator for *buffer_t* in *from* field. If a buffer has this in its *from* field, the packet is transmitted by using *rf_write()* function instead of *rf_write_no_cca()* function.

WCP_IDLE_SLOT brings scheduler timer state to *WAIT_FOR_SLOT*, which indicates that the radio is off. These slots include a mechanism to handle time-slots longer than 65 $msec$ as described in Section 3.1.3.

WCP_SERVICE_TX_SLOT event brings the scheduler timer state to the *RX_TX_SLOT*. In this slot, radio channel is set and packets are broadcast. Advertisement beacons are generated at an interval determined by *adv_freq*. Note that these beacons are used with the nodes that have children. The end nodes, as known as leaf nodes, do not need to broadcast service messages. When there is no advertisement beacon to transmit, node checks whether there is any other service messages in the broadcast queue. The broadcast queue is simply defined as a communication pair with broadcast address and its index number in every node is *MAX_CP-1*.

MAX_CP parameter is defined in app_config.c and it indicates maximum number of communication pairs one node can have. The broadcast communication pair, with index number *MAX_CP-1*, stores the service messages. As it will be explained in node joining Section 5.3, network configuration packets are transmitted broadcast within these slots. If a configuration packet with packet type *WCP_JOIN_PERMISSION* is to be transmitted, the node adds the time of the next timer event. This is done to enable the child to join the network at the right time. Details of this operation can be found in Section 5.3.

WCP_SERVICE_RX_SLOT slot brings the timer state of the scheduler timer to *RX_TX_SLOT*. In this slot, radio channel is set and node waits for an incoming packet to be received.

WCP_SERVICE_IDLE event has the same implementation as that of a *WCP_IDLE_SLOT*. This slot is defined separately for optimization purposes during the node joining phase. Normally, all the idle slots are combined together when schedules are being created. This is done to decrease the number of timer events in one node. However, when a node wants to join it should have a *WCP_SERVICE_IDLE* slot in its schedule, which indicates the first event it will generate when it is joining the network.

3.3 Packet Handling and Routing

3.3.1 Packet Structure and Routing in Packet Manager

The packet structure is as shown in Fig. 3.6. Length of the "packet route", depends on the *MAX_HOPS* variable defined in app_config.c file.

MAC Address (16 bytes)	Packet Type	Source ID	Destination ID	Packet Route (MAX-HOPS-1 byte)	Payload

Figure 3.6. Packet Structure in Packet Manager

ID numbers are 8 bit fields and their assignment in A-Stack is shown in Table 3.3.

A simple source routing is implemented in A-Stack for testing and validating purposes. Routes are formed in MATLAB and distributed when a node wants to join the network. Simply, every node knows their route back to the sink node, and the sink node knows the routes to all the nodes.

Once a packet is received at a node, the node checks whether the packet

Table 3.3. ID Assignment in A-Stack

ID Number	Description
1	Sink
2-252	Network nodes
253	Network broadcast
254	Advertisement beacon
255	PC

is intended for it, if not, it has to forward the packet to the next destination. This is done in *wcp_find_next_hop_index()* function as described below.

If packet type is *WCP_SERVICE*, and the node receiving the packet is on the route, it will forward the packet using a service slot by inserting the packet to the broadcast queue (*MAX_CP-1*). Else if the packet destination is one-hop neighbor of the node, it will assign the packet to the corresponding communication pair. Otherwise, the node will check the "packet route" and find its own ID in the array, and then it would take the next element in the array as the next hop ID and assign the packet to that communication pair. If next hop id is not valid, the packet is dropped.

3.3.2 Packet Types and Their Handling

The packet types given in Fig. 3.7 are used in C code as well as the MATLAB code.

WCP_DISCOVER packets are sent while the NanoStack is active. Nodes receiving a broadcasted discover message reply with *WCP_RESPONSE* type of packet. When sink node receives a response from a node, it prints the node address, which is available in the packet, to the serial port and MATLAB would store the node address as well as the node id.

WCP_ASSIGN_CP packets are used to assign communication pairs to the nodes. *WCP_SCHEDULAR_SET* packets are used for setting the schedules. These packets originate from PC, and they will first reach sink. If the packet is intended for the sink, it updates the settings. Otherwise, the sink forwards the packet. Note that network nodes first ask for their communication pair, and only then the PC generates this packet as a response to the node request.

WCP_SERVICE packets are used to broadcast settings during the A-Stack operation such as updating channels used in the schedule, frame

```
/*PACKET TYPES*/
//if one decides to modify here, he should make sure to do the same in PC
/** Packet types used in the stack */
typedef enum
{
WCP_DISCOVER = 0,
WCP_ASSIGN_CP = 1,
WCP_SCHEDULAR_SET = 2,
WCP_SERVICE = 3,
WCP_DISCOVER_RESPONSE = 4,
WCP_STATUS = 5,
WCP_DATA = 6,
WCP_RESPONSE = 7,
WCP_START_SYNC =10, //indicates that sync beacons will come so change state.
WCP_ADVERTISE =11,
WCP_ASK_CP =12,
WCP_ASK_SCHEDULE =13,
WCP_ASK_JOIN =14, //ask for joining
WCP_JOIN_PERMISSION =15, //join request given
WCP_SINK_START =16, //sink starts its schedule, other nodes join one by one
WCP_SERVICE_DATA =17, //data to be sent within service slots. only broadcast.
}wcp_packet_types_t;
```

Figure 3.7. Packet Types

frequency, re-transmission parameter and start simultaneous command.

WCP_DATA packets are used to indicate data packets for the application layer.

WCP_ADVERTISE packets indicate advertisement beacons. These beacons are at the same time time-synchronization beacons.

WCP_ASK_CP, *WCP_ASK_SCHEDULE* and *WCP_ASK_JOIN* packets are generated when a node tries to join the network. These packets are generated in the service manager task.

WCP_JOIN_PERMISSION packet gives the join permission to a node. The packet is generated by the PC and has the settings for node joining.

WCP_SINK_START packet gives an indication to the sink node to start the scheduled operation.

WCP_SERVICE_DATA packets are used for transmitting application layer data within service slots.

The packet types *WCP_DISCOVER_RESPONSE*, *WCP_START_SYNC* and *WCP_STATUS* are not used in the current implementation.

3.3.3 Packet Reception in MAC

Packets are handled in MAC based on the timer state, packet type, and frame type. Figure 3.8 shows the flowchart of packet handling in MAC.

The MAC decides to which module to forward the incoming packet. It can be packet manager, service manager, modules of NanoStack, or the packet can be handled and cleared in MAC, as is done for advertisement beacons. Note that synchronization information is located in advertisement packets, and it is handled in rf.c before the packet arrives to MAC.

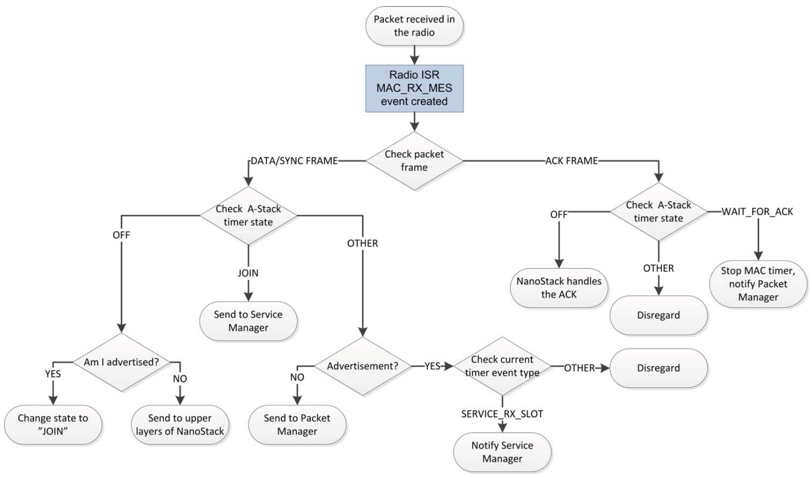

Packet received in the radio
Radio ISR MAC_RX_MES event created
Check packet frame
DATA/SYNC FRAME
ACK FRAME
Check A-Stack timer state
OFF
JOIN
OTHER
Send to Service Manager
Am I advertised?
YES
NO
Change state to "JOIN"
Send to upper layers of NanoStack
Advertisement?
Send to Packet Manager
Check current timer event type
SERVICE_RX_SLOT
Notify Service Manager
Disregard
Check A-Stack timer state
WAIT_FOR_ACK
NanoStack handles the ACK
Disregard
Stop MAC timer, notify Packet Manager

Figure 3.8. Flowchart of packet handling in MAC.

MAC handles a packet based on the output of *mac_header_analyze()* function. If the packet has a *FC_SYNCHRONIZATION_FRAME* (defined in time_sync.c) or a *FC_DATA_FRAME*, MAC decides where to forward the packet: NanoStack, Service Manager or A-Stack. When the scheduler timer state is *WCP_OFF*, packets are forwarded to NanoStack. However in this state, if the first byte of the data packet is *WCP_ADVERTISE* and if the nodes own id is included in the advertisement, the scheduler timer state is changed to *WCP_JOIN*. If the state is *WCP_JOIN*, packet is sent to *service_queue*. In other states, the packet is sent to *wcp_queue*. When the state is neither *WCP_JOIN* nor *WCP_OFF*, the packet type is checked and if it is an advertisement, directed to the node and if the current timer event type is *WCP_SERVICE_RX_SLOT*, *adv_rx* parameter is set to 1. This parameter is used to track whether the node is in the network by checking whether the advertisements are received within dedicated time-slots. This is done in MAC to ensure that there is no delay between packet reception and checking the time slot.

ACK Handling in A-Stack is designed for a smooth operation. Since there are multiple transmission time-slots with transmission functionality, there are more than one packet in the MAC at the same time. Thus it is important to determine correct ACKs for every packet. If a packet has a *FC_ACK_FRAME*, and timer state is *WCP_OFF*, ACK handling algorithm of NanoStack is followed. The *ACK_OK* indication is sent to the *wcp_queue* if the state is *WCP_WAIT_FOR_ACK* and the last sent buffer from the current timer event's buffer queue. Handling the state *WCP_WAIT_FOR_ACK* is extremely important for reliability in terms of data transfer. Whenever, a packet that requires an ACK is to be transmitted, a timer is launched by using the function *timer_rf_launch()* (in *wcp_mac_tx_buf()*, or in *mac_tx_buf()*).

If an ACK is not received within a pre-defined time interval, an event is generated from the mac timer interrupt service routine. This event is dedicated to *MAC_TIMER_INT_CB*. In *MAC_TIMER_INT_CB*, if active state is *WCP_OFF* NanoStack routine is followed. If active state is *WCP_JOIN*, an *ACK_TIMEOUT* event is sent to *wcp_queue*. When the state is *WCP_OFF*, *timer_rf_launch()* function is used as
timer_rf_launch(8000/PLATFORM_TIMER_DIV)
which indicates a waiting interval of 8 $msec$. Otherwise it is used as
*timer_rf_launch(WCP_ACK_WAIT*1000/PLATFORM_TIMER_DIV*).
Parameter *WCP_ACK_WAIT* is set in app_config.h file. It is advised to set this variable to 7 (which refers to 7 $msec$) for a 10 $msec$ slot.

3.3.4 Packet Handling in Packet Manager

Packets arriving to the Packet Manager are handled based on the packet type and destination. If the packet destination matches the node id, the node handles the packet internally. If the packet is aimed at another node, the node finds the next hop id from the routing part and forwards the packet. If the packet has a broadcast message id, the required information is fetched and the packet is broadcast when the *SERVICE_TX_SLOT* occurs next time.

Broadcast messages are used when a packet, such as a service message, is to be transmitted to all the nodes in the network. Direction of these messages is from the sink to the network, and not the other way around. Every node receiving these messages checks whether it has a *SERVICE_TX_SLOT*. Absence of this slot means that the node is an end node, and it has no other node to forward this service message. Thus end nodes drop the broadcast messages after they fetch the required information.

If the packet destination is different than that of the node id or the broadcast id, next hop id is taken from the routing array included in the packet and the packet is added to the transmission queue of the communication pair corresponding to the next hop id. If the packet destination is the same as the id of the node, it is handled based on the packet type as explained below.

Service Packets

The service packets have the packet type *WCP_SERVICE*. These packets are transmitted always in *WCP_SERVICE_TX_SLOTs*. In the current implementation of the stack, there are four types of service messages.

```
/*SERVICE TYPES*/
typedef enum
{
   WCP_CHANGE_RE_TX   = 0,
   WCP_CHANGE_CHANNEL= 1,
   WCP_CHANGE_FRAME_FREQ     = 2,
   WCP_START_SIMULTANEOUS    = 3,
}wcp_service_types_t;
```

Figure 3.9. Service message types

WCP_CHANGE_RE_TX service packet type is used to change *re_tx* parameter. *re_tx* parameter indicates how many times a packet, which does not receive an ACK, is going to be transmitted before it is dropped.

WCP_CHANGE_CHANNEL service packet type is used to change radio channel used for transmission and reception slots. Note that, in current implementation, only the channels used in transmission and reception slots are changed but channels used in service slots are not changed. The reason is to preserve a link for service and configuration in case a problem occurs during the update. Note that, a multi-channel schedule will change the channels when switching between time-slots. For example a node can have a RX slot at channel 15 and a TX slot at channel 14. MAC layer of the node will change the channels between 15 and 14 at every superframe. If the network manager decides to change the schedule of the node so that the node has a TX slot at channel 11 and RX slot at channel 13, it will use *WCP_CHANGE_CHANNEL*.

WCP_CHANGE_FRAME_FREQ service packet type is used to change the frequency of the superframes. By changing the frequency, user can choose to disable RX and TX slots when the operation does not require high data-rates.

WCP_START_SIMULTANEOUS service packet type is used to start a network wide operation, such as simultaneous sampling.

Data Packets

The data packets types are *WCP_DATA* and *WCP_SERVICE_DATA*. The *WCP_DATA* packets are sent within *WCP_TX_SLOT* whereas the packets of type *WCP_SERVICE_DATA* are sent within *WCP_SERVICE_TX_SLOT*. The *WCP_SERVICE_TX_SLOT* is available only for one direction: sink to network. Thus *WCP_SERVICE_DATA* packets should not be used for sending messages from the nodes to the sink. Data packets are forwarded to the application layer, application queue, for further handling.

Configuration Request Messages

The configuration request packets have the packet types *WCP_ASK_CP*, *WCP_ASK_SCHEDULE*, and *WCP_ASK_JOIN*. A joining node uses these packets to ask for configuration packets. These packets are handled separately in the sink node. If the sink node receives either one of these packets, it will print the request to the serial port, and MATLAB will take the necessary action. If a node other than the sink receives any one of these packets, it has to handle the request by preparing an appropriate packet to be sent to the sink and putting this packet into the corresponding transmission queue. Configuration messages corresponding to these requests are handled in service manager once they are created in PC. A more detailed explanation of the events is given in Section 5.2.

3.3.5 Packet Handling in Service Manager

When timer state is *WCP_JOIN*, packets received in MAC layer are forwarded to the service manager. Packet types which are forwarded by sink and handled in corresponding node's service manager are; *WCP_ASSIGN_CP*, *WCP_SCHEDULER_SET*, and *WCP_JOIN_PERMISSION*. The detailed functionality of these packets can be found in Section 3.3.2.

4. Time Synchronization

In order to enable time-slotted communication, all the nodes in a sensor network must be synchronized to a common time reference. To achieve this, A-stack integrates a time synchronization service. The time synchronization in wireless sensor network has been widely addressed in the literature. The readers interested in the details on other synchronization protocols can refer to [17][18] and the references therein. The time synchronization service in A-stack is built as an extension of our MAC layer synchronization protocol (μ-Sync) [17]. The original implementation of the μ-Sync protocol is for Sensinode Nano.2430 platform [17]. The μ-Sync protocol is ported to the Sensinode Micro.2420 platform in a separate effort [18] since Micro.2420 platform offers better stability of the clock i.e. oscillator. The underlying time-synchronization procedure along with the differences from the original implementation of μ-Sync is outlined in this report.

4.1 Implementation of Synchronization Clock

The MCU of Micro.2420 platform (TI MSP430F1611) provides two 16 bits timers, namely Timer-A and Timer-B. Timer-B runs as a local clock of a node for task scheduling and other MAC layer operations, e.g. random back-off, acknowledgments expiry time, etc. Timer-A is free and can be used for performing time-synchronization. This timer has one 16 bits counter (TAR) and three 16 bits configurable compare/control registers (TACCRx). The source of Timer-A is an 8 MHz clock derived from an external 16 MHz crystal oscillator which has an accuracy of ± 40 PPM. The 8 MHz source is divided by 8 before being passed to Timer-A, resulting in a tick resolution of 1 μsec. The value of TAR is constantly increasing with 1 μsec tick resolution, generating an interrupt at each overflow, i.e. every

65.535 $msec$. The interrupt is acknowledged in an ISR and a low resolution logical clock counter keeps a track of the number of overflows. The wrap-around time of each node can vary due to the accuracy, temperature and aging of the oscillator. Therefore, even if two clocks are synchronized, a time error is caused depending on the relative time error parameters of the clocks which will be further discussed in Section 4.3.

4.2 Synchronization Method

The μ-Sync is a broadcast based synchronization protocol where a synchronization beacon is generated at MAC layer by a reference node with a predefined period. The synchronization beacon carries a frame control field 0x05. The reference node timestamps the beacon just before the microcontroller is signaled to transmit the beacon. The one-hop recipient nodes timestamp the received beacon as the microcontroller signals the reception of the valid packet with an interrupt. By this method most of the uncertainties in the sender-receiver path are eliminated except for the delay in the packet reception interrupt time and the propagation time. However, their impact is minimal as compared to the desired accuracy of the time-synchronization service. Therefore, by carefully performing the clock offset budget analysis in the sender-receiver path within the time-stamping procedure of synchronization beacon, the nodes can be synchronized to a common reference accurately.

4.2.1 Clock Offset Budget Analysis

The time offset between the reference node and a child node under one-way sync beacons needs to be critically scrutinized. Figure 4.1 shows the critical path from sender at level (n-1) time-stamping the sync beacon at t_1 till the receiver at level n time-stamping the received beacon at t_2. The time length of the critical path Δt as given below

$$\Delta t = t_{\text{tx\_ts}} + t_{\text{encoding}} + t_{\text{prop}} + t_{\text{decoding}} + t_{\text{int\_handling}} + t_{\text{rx\_ts}} \tag{4.1}$$

should be added in t_1 in order to synchronize the receiver clock with that of the sender of sync beacon.

The factors contributing to Δt are elaborated below.

- Timestamp read/insert time: at sender ($t_{\text{tx\_ts}}$), the time between the in-

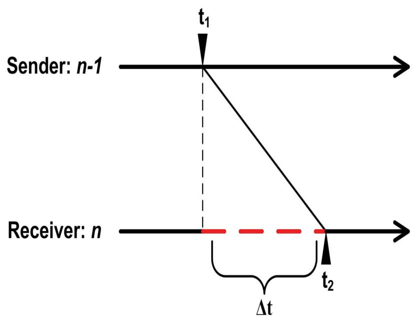


Figure 4.1. Critical path in one-way time synchronization

sertion of timestamp into a sync beacon and microcontroller signaling to radio chip to transmit. At receiver ($t_{\text{rx\_ts}}$), the time between the sync beacon reception signaled by the microcontroller and timestamp insertion.

- Interrupt handling time ($t_{\text{int\_handling}}$): the time between when an interrupt is raised and pending for the CPU time and when the microcontroller vectors through the interrupt. This time is less than a few microseconds. However, the delay can grow longer if the interrupts are temporarily disabled or a higher priority interrupt is in service.

- Encoding time/Decoding time ($t_{\text{encoding}}/t_{\text{decoding}}$): the time taken to encode the frame and transform into radio waves. It starts when the radio chip is signaled to transmit the frame. This time is deterministic and it is in the order of hundreds of microseconds. Decoding time is the time taken by the receiver to transform the radio waves and decode back into the binary message, and the message reception interrupt to be signaled to the microcontroller. This time is also deterministic and it is in the order of hundreds of microseconds.

- Propagation time (t_{prop}): the time taken by the radio waves from the transmitter antenna to reach the receiver antenna. The propagation time is less than one msec for distances smaller than 300 m [19].

We measured by configuring a reference node to send the sync beacons periodically. The reference node toggles an I/O pin after time-stamping the sync beacon while a child node toggles the I/O pin on receiving the beacon before it is being time-stamped. The time difference between the two toggle events is measured with an oscilloscope. The mean value of Δt is 825 $msec$, as tested for multiple nodes, with negligible variations as compared to the tick resolution.

4.3 Clock Skew

The instability of the crystal oscillators causes relative clock skew. A time-synchronization protocol only adjusts the current clock offset among the clocks and with time they start deviating from each other. This necessitates the periodic synchronization of the clock unless a clock skew estimation mechanism is employed on the nodes. The upper bound on the synchronization period depends on the clock skew rate. It must also take care of the nodes experiencing worst skew as well as application requirements.

In the sequel of characterizing clocks inherited instability and skew, we conducted similar experiments as described in [17] for Micro-nodes. To determine the relative clock skew of the nodes, first, we loosely synchronize a set of nodes with a reference node and later we let the clocks run undisciplined. The nodes are configured to toggle an I/O pin every 65.535 $msec$ and the clock skew between the reference node and the nodes is measured with a digital oscilloscope. The clock skew for five clocks is shown in the Fig. 4.2.

It shows that the clocks deviate both in positive and negative direction with respect to the reference clock. The skew in clock C is the lowest whereas node A has the highest skew. We tested five nodes and found skew in clock A to be worst, that is 3.83 $\mu sec/sec$. It also shows that the relative skew in the clocks is linear but with small fluctuations across the linear line.

For maintaining the synchronized operations in the network, the worst-case synchronization error among the nodes, $\tau_{\text{worst-case}}$, must be less than the guard time, T_g, in each time-slot. This condition places an upper bound on the synchronization period, T_{sync}.

$$T_{\text{sync}} < \frac{T_g - \tau_{\text{worst-case}}}{\varepsilon} \tag{4.2}$$

where ε is the clock accuracy normally specified in parts per million (PPM). In our platform, $\varepsilon = \pm 40$ PPM, $T_g = 3\ msec$ and $\tau_{\text{worst-case}} = 5\ \mu sec$, $T_{\text{sync}} < 2.995\, msec/80\,\text{PPM}$ = 37 sec, that is the clocks must be synchronized every 37 sec. Note this value depends on the selection of the guard

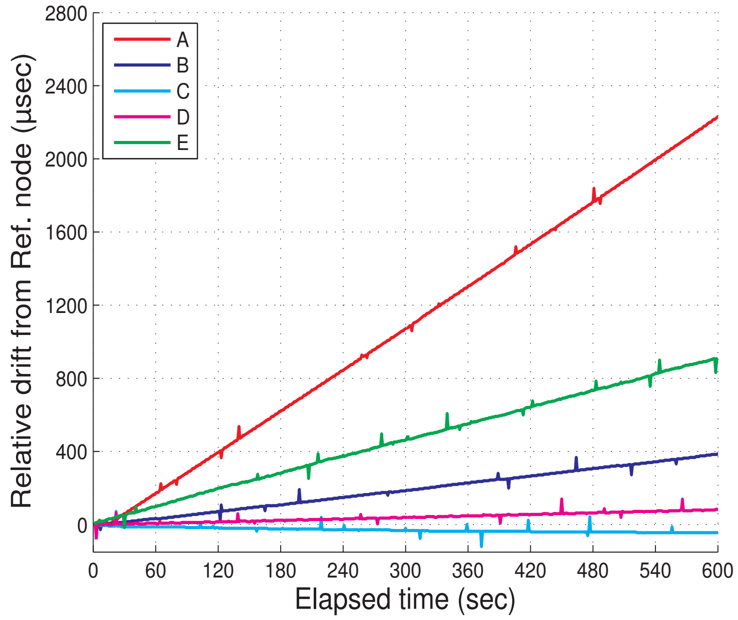


Figure 4.2. The effect of clock skew on time-synchronization

time, T_g.

4.4 Time Synchronization Accuracy of MICRO.2420 Platform

We tested the accuracy time-synchronization at first hop. In the tests, the sink node transmits a time synchronization beacon every 1 sec. At the reception, the nodes adjust their local time as mentioned in Section 4.2. The synchronization error is measured by toggling an I/O pin every 65.535 msec. The results are depicted in Fig.4.3. The absolute average synchronization error is 1.74 μsec. The absolute error is larger than the average value in the 24.73 % of times however the absolute error remains below 5 μsec in the 98.57 % of times. Figure 6 shows also the presence of few points in which the absolute synchronization error is remarkably higher (e.g. 12, 16, 18, 31, and 37 μsec). This fact is caused by the inherent instability of the crystal oscillator found in the Micro.2420 platform.

4.5 Time Synchronization Service

The time-synchronization service provides the following interfaces to control the core functionalities of time-synchronization and generation of

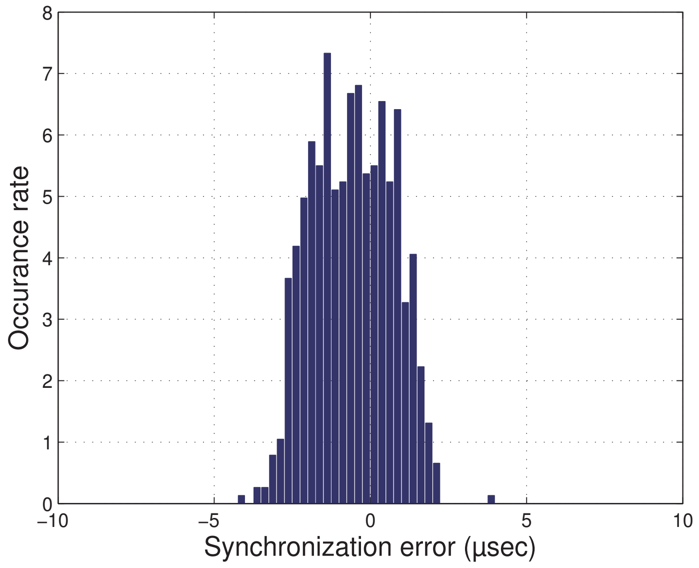


Figure 4.3. The 1-hop time synchronization accuracy of Micro.2420 nodes

scheduling events.

4.5.1 Time Synchronization Interface

The system clock as well as the time synchronization (if configured) is initialized with *xPortStartScheduler()* by using the functions *prvSetupTimerInterrupt()* and *prvSetupLogicalClockInterrupt()*. The time synchronization clock is setup such that the overflow of 16-bit TAR counter generates an interrupt which is handled in *ISR prvTickISRSCLK()* by reading the TAIV interrupt vector. The TAR overflow appears as case 10 in this ISR. At each overflow a low resolution tick counter, *xTickCountSCLK*, is incremented. This low resolution tick counter is used as common counter in the network for initializing the scheduler as explained in Section 4.5.3 . The synchronization information is added to the end of the advertisement beacons. The advertisement beacon once created follows the routine transmission procedure and it is time-stamped as mention in Section 4.2.

4.5.2 Time Synchronization and Advertisement Beacons

The distribution of time synchronization information is performed within *SERVICE_TX* slots using advertisement beacons. The time synchronization information includes 2 bytes representation of the source clock, i.e. TAR counter. A node receiving this information updates its clock according to the procedure given in Section 4.2.

4.5.3 Scheduler Event Generation from Timer

The generation and indication of the time-slots to the user of the time synchronization service is based on the TACCR0 compare/capture register of Timer-A. *TACCR0* is initially configured with the desired time-slot length in *prvTickISRSCLK()* as the scheduler is started. Later on, TACCR0 related interrupt is handled in *prvSLOT()* ISR. *prvSLOT()* takes care of the future time-slot generations by setting up *TACCR0* according to the duration variable defined in Section 3.1.3.

4.6 Known Problems and Enhancements

The current implementation of the time-synchronization procedure has the following open issues. The enhancements are also suggested along with these issues.

4.6.1 Clock Skew Estimation

An exchange of sync beacon achieves only instantaneous synchronization as the clocks soon start deviating from each other. However, if the relative clock drift is estimated from the past synchronization points it will reduce the required communication overhead for a given clock synchronization accuracy and hence save energy. Typically, the energy required to transmit one bit is equal to that required to execute three million instructions [20]. Hence, energy constraint sensor motes require as less frequent as possible exchange of synchronization beacons in exchange of clock drift estimation based on local computations. We plan to add the clock skew estimation module to A-Stack based on our proposal [21].

4.6.2 Time-Stamping Error at Receiver Side

The time-synchronization procedure relies on the time-stamping at the sender and receiver. The time-stamping method at sender is as good as it can be, however, the time-stamping at receiver has a problem, described next.

The time-stamping of a synchronization-beacon at the receiver is performed at *rf_isr*. The *rf_isr* is a maskable interrupt and it can be disabled at the reception time of a synchronization beacon. When *rf_isr* is disabled, it introduces inaccuracy in clock offset computation with respect to the synchronization beacon sender. The time-synchronization procedure compensates for the deterministic delay Δt at receiver side and adjusts the clock. However, if *rf_isr* is not serviced as soon as a synchronization beacon is received, it introduces inaccuracies. The examples in Table 4.1 and Table 4.2 shows the possible error in clock offset calculation.

Assuming that the reference and child clocks are synchronized, we consider some cases demonstrating how the delay in service time in *rf_isr* can affect the clock offset calculations. The clock is represented as **X** y, where bold face letter (**X**) represents the low resolution tick count and small letter (y) is the high resolution tick count. We also assume that the clocks are synchronized every 1 second and the clock skew is 3 $\mu sec/sec$. Also, the high resolution tick counter is 16 bit and low resolution tick count is 32 bit. The high resolution tick count runs at a speed of 1 MHz.

Table 4.1. Case-1: Assuming the child clocks runs faster than the root clock

Root	Child	Offset	Comments
2 15	**2** 15	**0** 0	-
3 500	**2** 503	**1** -3	rf_isr is serviced without delay
3 500	**2** 515	**1** -15	rf_isr is delayed

Table 4.2. Case-2: Assuming the child clocks runs slower than the root clock

Root	Child	Offset	Comments
2 15	**2** 15	**0** 0	-
3 500	**2** 497	**1** 3	rf_isr is serviced without delay
3 500	**2** 509	**1** -9	rf_isr is delayed

Suggested Enhancement; The above-mentioned problem can be avoided by time-stamping the synchronization beacon on SFD (start frame delimiter) detection. CC2420 generates a signal when a valid SFD is detected and Timer-A can be configured to capture TAR ticks count at SFD detection of synchronization beacon. This method would improve the accuracy of time-stamping on receiver side and hence synchronization.

4.6.3 Time Synchronization Service Reliability

The recent attempt in the integration of time-synchronization with TDMA MAC has revealed the following problems in time-synchronization implementation. The revivification of the procedure is important to ensure the long term reliability of the synchronization service in a network.

- An unnecessary interrupt (overflow interrupt) is generated in addition to compare register *CCR0* interrupt. This is due to the wrong configuration of time-synchronization timer initialization function *prvSetupLogicalClockInterrupt()*. An interrupt occurs when TAR is equal to *CCR0* whereas an unnecessary interrupt occurs as *TAR* wraps around (becomes zero). In order to avoid that *prvSetupLogicalClockInterrupt()* must be initialized as *TS_TIMER_CTL = TASSEL_2 | MC_1 | ID_3*.

- The naked function modifier attribute in ISR definitions must not be used as well as the *asm volatile("reti \ n\ t"\)*. The naked attribute prevents the compiler from generating prologue and epilogue code for an ISR. However, we are responsible for saving any registers that may need to be preserved, selecting the proper register bank, generating the reti instruction at the end, etc. Since, we do not possess the sufficient knowledge on these details there is a possibly of mistakes and it is recommended that we stick to inline assembler.

Open Issue: The time-synchronization related ISRs must be optimized such that only the necessary registers are saved.

5. Operation Topics: Network Configuration and System Setup

This chapter explains the network configuration and A-Stack system set-up. PC and sink node implementations are given in Section 5.1. Network configuration and node joining is explained in Section 5.2 and 5.3. Finally, system parameters to be set are explained in Section 5.4.

5.1 PC to Sink Communication

Reliable data exchange between A-Stack sink node, or gateway, and PC is established using PC to sink node communication protocol. In this protocol, the sink node employs a state machine to handle incoming data bytes. MATLAB employs a callback function to handle incoming data from the sink. The packet format used is given in Fig. 5.1. The ID assignments sender and receiver are given as in Table 3.3.

Start byte 7	Packet length	Packet type	Sender ID	Receiver ID	Payload

Figure 5.1. Packet Format

5.1.1 Sink Node Implementation

Sink node handles the data coming from the PC in a task called "serial task". This task is not implemented in source code of A-Stack, i.e. it is not in *wcp_functions.c*, since it is used only by the Sink node. Furthermore, it is easier to handle application specific commands at the application layer by having the task at Sink node.

The sink node receives the packets coming from the PC byte by byte. First it detects a Start Byte, which in this case is 7. Then it will receive the packet length. Then it reads the data bytes to an array. Once all the bytes are received, this array is handled according to the packet type.

Packet types are the same as the ones presented in Section 3.3.2. However, additional packet types can be assigned just for the communication between the sink node and the PC depending on the application.

A critical issue in the task reading the serial port is that it should not block. Reading bytes from the USART and handling the data once a packet is received is done one after another. If the data handling part blocks, new bytes might be lost.

5.1.2 PC Implementation using MATLAB

MATLAB program is used for interfacing the sink node and therefore the network. MATLAB initializes PC's serial port using a function *logmain(PORT_NUMBER)*. This function assigns a *serial_receive_callback* function for the messages received through the serial port. In the callback function, incoming data is received as strings. Incoming data is either a request from the nodes in the network or the data to be collected. MATLAB handles these data within the callback function.

wcp_send_array function is used in MATLAB for sending a packet to the network nodes or to the sink node. Packets sent from MATLAB includes packet type, source and destination ids.

5.2 Network Configuration

A series of MATLAB functions are implemented to configure networks when A-Stack is used. The actions taken for configuration will be presented step-by-step below. These functions present the default configuration of the network and the user can change the configuration to meet the requirements.

5.2.1 PC Initialization

The main structure used in MATLAB code for storing data regarding A-stack is named "WCP_APP". The *initialize_wcp_object(WCP_APP)* function is used to initialize the parameters used in the operation. These parameters include event types and packet types. An important characteristic of this function is that it includes the application specific definitions given in app_config.c file. If these parameters are changed in the source code, they have to be changed also in here for correct operation.

5.2.2 Operational Settings

The following fields in WCP_APP structure, defined in MATLAB, affect the operation of A-stack.

WCP_APP.superframe_length: super frame length in milliseconds. Note that length of the generated schedule can be greater than the length specified in here. In such case, the length of the generated schedule is used. If this length is shorter than the length defined in here, idle slots are added at the end of the schedule.

WCP_APP.superframe_freq: defines the superframe frequency as explained in Section 3.1.4.

WCP_APP.advertise_freq: this parameter indicates period of advertisement beacons in terms of superframes.

WCP_APP.rx_tx_slot_length: length of rx_tx slots in milliseconds

WCP_APP.sync_slot_length: length of synchronization slots in milliseconds

WCP_APP.frame_slot_length: length of frame slots in milliseconds

WCP_APP.tx_offset: the offset (guard time) to be added in a TX slot in order to make sure the receiver is listening, for reliable communication.

WCP_APP.service_slot_length: length of service slots in milliseconds

WCP_APP.pause_param: pause parameter used in MATLAB code between two consecutive message transmissions over serial port.

WCP_APP.re_tx: this parameter indicates number of retransmissions of a buffer, in the network nodes, in case of ACK failure.

WCP_APP.use_wireless_tools: this parameter activates the wireless tools schedule generation in create_schedule.m. If it is set to zero, a single channel schedule that connects all the nodes to the sink is generated.

WCP_APP.wireless_tools_option: this option is used to tune the generated schedule by using wireless tools. [1] will add transmission slots to children and reception slots to the parents, [2] will do vice versa, and when [1 2] is selected, both transmission and reception slots are added to the parents and children. Detailed information on wireless tools and its implementation on A-Stack can be found from [14].

5.2.3 Serial Port

Serial port is opened by using *logmain()* function. The created serial object is assigned to the *WCP_APP* parameter.

5.2.4 Node Discovery

Nodes are discovered by sending a discover command to the sink node. The sink node then broadcasts a discover message to the network and the nodes reply. The sink node prints the addresses of these nodes to the serial port together with node IDs. This information is then used when configuring communication pairs and schedules. Currently, all the nodes need to be close to the sink so that they can hear the discovery messages, however in the future a more advanced discovery and initialization scheme is to be implemented.

5.2.5 Routes

Routes are organized in the *compair_table* matrix that is stored in a parameter, *WCP_APP*. Currently, this matrix is organized semi automatically; i.e. the user chooses the number of hops, and the nodes are arranged to meet the targeted hop count. However, any combination of discovered nodes can be given to the system by using the *compair_table*. Section 6.2 includes examples showing how *compair_table* is used to define hops and routes.

5.2.6 Communication Pairs

Parent nodes and their children are set using *organize_routes()* function. This function organizes the routes using the *compair_table* defined earlier. Using the *compair_table*, parent nodes and their children are assigned using the function *organize_routes()*.

5.2.7 Schedule Creation

Schedules are created using the *create_schedule()* function. At first, a *FRAME_SLOT_START* slot is added to all the nodes in the network. Then optional *TX_SLOT_START* and *RX_SLOT_START* slots are generated for all the links. If a node has no transmission and reception for a period, *IDLE_SLOTs* are assigned to it. In the end, the consecutive idle slots are combined together to decrease the number of events. Finally *SERVICE_TX_SLOTS*, *SERVICE_RX_SLOTS*, and *SERVICE_IDLE* slots are added. If the user defined superframe length is not reached, *IDLE_SLOTS* are added to the end of the schedule.

5.2.8 Configuration Packets

The number of communication pairs or the events might be greater than that can be stored in a single packet. To deal with these, A-Stack supports multi-part configuration packets and these packets are formed before the scheduled operation starts. When the nodes try to join, these packets will be forwarded to them.

At this stage *WCP_APP*.tx_offset parameter is used to update the schedules, i.e. a guard time is added to the beginning of transmission slots so that the receiving nodes are guaranteed to be ON when a node is transmitting a packet.

5.2.9 Sink and Network Initialization

Once all the configuration packets are ready, the sink node is configured. After sink node receives its configuration, it is given a start permission command. Then the sink node starts advertising to its one hop neighbors. Nodes that hear advertisements send requests to receive their configuration packets if they hear advertisements targeted to them. Once a node has all its configuration parameters, it will ask for the join permission, and once that is received it will join the network. Nodes joining the network will start to advertise down in the tree topology, and their children will eventually join the network.

5.3 Node Joining

The configuration phase is critical when the size of the network is large. In A-Stack, end to end reliability during configuration is obtained by using a state machine in the network nodes. When a node intends to join, it will go through 3 states: 1. take communication pairs information, 2. take schedule information, and 3. take route, final settings and join permission. A node moves from one state to another only if all the configuration packets in its current state have been received.

Configuration packets are requested by the nodes based on their joining state. Note that the scheduled transmission slots are not employing a CCA mechanism, thus it is important for all the nodes in the network to transmit a packet only when they have a slot reserved for them. When a node wants to join, it doesn't know its transmission slots. In order to prevent joining nodes to interfere with the rest of the network, advertisements are used as an indicator. A joining node transmits a packet only after it hears an advertisement. Advertisements are transmitted by parent nodes in *SERVICE_TX_SLOTs* at intervals defined by the user. These slots are followed by *SERVICE_RX_SLOTs* for the joining nodes to reply on. During the joining phase, nodes employ CCA and they do not transmit a request if the channel is busy. This is critical as there might be multiple children of a parent that want to join at the same time. If a node cannot transmit a configuration request packet, it will wait for the next advertisement.

Joining nodes receive synchronization information as they receive advertisement beacons during node joining phase, and adjust their clocks accordingly. A critical issue is when exactly a node will join. Configuration packets are transmitted as broadcast messages at the last hop. This ensures that they are transmitted when a *SERVICE_TX_SLOT* occurs in the parent. When there is a *SERVICE_TX_SLOT* at a parent node, it's children nodes have *SERVICE_RX_SLOT* in their schedule and this slot is followed by a SERVICE_IDLE slot. A node will join at the end of this *SERVICE_IDLE* slot.

5.3.1 Re-Joining

If a node stops receiving advertisements within its *SERVICE_RX_SLOT*, it will try to re-join the network. The service manager task periodically checks whether the advertisements are received in right slots. If not, it will change the state of the node to JOIN. It will reset the scheduler but keep the configuration information. It will also drop all the packets from the previous state. In this case the node already has the configurations, thus only the join request will be sent to the PC through the parent node.

5.3.2 Service Manager when Node Joining

The initial state of the nodes is *WCP_OFF*. When a node hears an advertisement beacon that has the node's ID, the MAC layer will notify the service manager by changing the variable *advertised_when_off* to one. The service manager task monitors this parameter and if this parameter is one, it will start the node joining process by changing the state to *WCP_JOIN*. In this state, the packets are forwarded to the service manager from MAC.

Normally the service manager task is implemented to run in relatively longer intervals, in order not to consume MCU resources. In the current implementation, it executes in a polling fashion, i.e. it wakes up after an interval of 30 seconds. However, if the state is *WCP_JOIN*, it will enter a loop, and continuously process the incoming packets in a callback fashion. It will remain in the loop as long as the configuration phase is ongoing.

The last configuration packet is the join permission, which includes information on when exactly to start the schedule. Configuration packets are transmitted in *SERVICE_TX* slots from the parent, and all the children of a parent have *SERVICE_RX slot* corresponding to parent's *SERVICE_TX* slot in their schedule. Every *SERVICE_RX* slot is followed by a *SERVICE_IDLE* slot. A node will join the network at the end of this slot. In order to join at the right time, a node should know the index number of this event in its superframe. When preparing the join permission packet, MATLAB adds this index number into the packet and service manager uses this information for joining.

5.3.3 Time Required for Node Joining

Depending on the network configuration, time required for node joining can be different, but it is in the scale of seconds for one node in the network provided that the super frame is shorter than a second. Below is a formula for calculating how long it would take for a node in the network to join.

Time Required for Node Joining= A + B+ C+ D

A denotes the *SERVICE_TASK_DELAY* which defines how often the service task is triggered. After the first time an advertisement is heard, MAC notifies the service task for changing the state to JOIN. However service task is triggered only when *SERVICE_TASK_DELAY* ends, which causes the initial delay. *SERVICE_TASK_DELAY* parameter is used in service task, which is mostly idle when A-Stack is running, thus this parameter is chosen to be large, currently 30 seconds, in order not to consume MCU resources.

B denotes the time required for taking the communication pairs and it is a function of the superframe length. It can be found by using the formula below.

B= (2 + number_of_hops) x number_of_cp_packets x superframe_length

C denotes the time required for taking the schedule information.

C= (2 + number_of_hops) x number_of_sch_packets x superframe_length *number_of_hops* denotes the rank of the node in the hierarchy. Number of schedule packets and communication pair packets varies depending on the node and its configuration. If these packets are multi-part, *B* and *C* are scaled with *number_of_cp_packets* and *number_of_sch_packets*.

Finally, *D* is the time required for taking the last packet, which is the *JOIN_PERMISSION*.

D= (2 + number_of_hops) x superframe_length

Above formulas give the time required for node joining for one node in the network. Network setup time can be estimated by using these formulas. Note that nodes at different levels of the hierarchy require different times for node joining, thus the network initialization time varies depending on the topology. However this initialization time is less than 10 minutes when there are less than 15 nodes.

5.4 Setting Up A-Stack and Source Code

A-Stack is implemented for Sensinode Micro.2420 nodes on top of NanoStack-v1.0.3. This section presents the files modified or created for A-Stack.

5.4.1 Platform Folder

Platform/micro: time_sync.c
rf.c (replace)
port.c (replace)
platform.rules file, modified as below:
#Platform driver section
$(PLATFORM_DIR)/time_sync.c
#NanoStack section
$(COMMON_DIR)/wcp_functions.c
Platform/micro/include:
time_sync.h
rf.h (replace)

5.4.2 Common Folder

wcp_functions.c
common/include:
wcp_functions.h
wcp_defines.h

rf_802_15_4.h (replace)

common/modules

rf_802_15_4.c (replace)

5.4.3 Application Configuration

Every application includes an *app_config.h* file which includes variables needed for initialization of A-Stack. These variables are explained below.

HAVE_WCP variable is used in compile time to differentiate between the default NanoStack code and the modified one. When set to one, this variable enables changes related to A-Stack. If it is set to zero, default NanoStack is active.

Variables *MAX_CP*, *MAX_CP_TX_BUFFERS*, *MAX_TIMER_EVENTS*, *WCP_QUEUE_SIZE*, and *MAX_HOPS* are used for static memory allocation during initialization phase.

WCP_MAX_RE_TX variable indicates the maximum re-transmissions within one slot. Note that slot lengths should be longer if multiple retransmissions are to be used. This is different than the *re_tx* variable defined in Section 3.1.3.

PACKET_MANAGER_TASK_PRIORITY variable is used to set the priority of the packet manager task. This priority can be changed since it is not time critical. Currently, it has a higher priority than the application task, but for example if accurate sampling in application task is more important than handling the packets, application task priority can be set higher and packet manager task priority can be set lower.

SERVICE_TASK_DELAY variable indicates the waiting time before the service task executes.

6. Examples and Tests

6.1 Example Schedules

Schedules comprising of various network topologies can be realized by modifying a network table, WCP_APP.compair_table. After this table is formed, schedules are automatically generated by MATLAB tools of A-Stack. A-Stack also provides an option to use Wireless tools software [14] for schedule generation. Abbreviations used in figures can be found in Table 3.2. Numbers below transmission (TX) and (RX) slots refer to the channels used in these tests.

6.1.1 Single Hop Schedules

Figure 6.1 shows a single hop topology with six network nodes and a gateway. In this single hop case, only a single channel is used as all the nodes are connected to the gateway which can handle only one channel at a time. Table 6.1 shows the network table, or *compair_table*, used for creating the single hop schedule. This schedule assigns *TX_SLOT (TX)* only to the network nodes and not to the sink node since only network nodes are expected to deliver data. Commands and data from the sink node can be disseminated within *SERVICE_TX (S_T)* slots.

Table 6.1. Single hop schedule, network table.

Parent	**Children**					
1	2	3	4	5	6	7

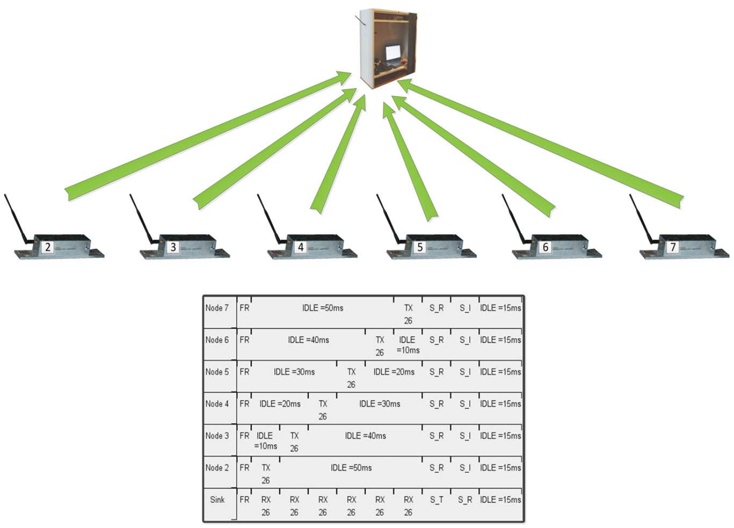

2
3
4
5
6
7
Node 7 FR IDLE =50ms TX 26 S_R S_I IDLE =15ms
Node 6 FR IDLE =40ms TX 26 IDLE =10ms S_R S_I IDLE =15ms
Node 5 FR IDLE =30ms TX 26 IDLE =20ms S_R S_I IDLE =15ms
Node 4 FR IDLE =20ms TX 26 IDLE =30ms S_R S_I IDLE =15ms
Node 3 FR IDLE =10ms TX 26 IDLE =40ms S_R S_I IDLE =15ms
Node 2 FR TX 26 IDLE =50ms S_R S_I IDLE =15ms
Sink FR RX 26 RX 26 RX 26 RX 26 RX 26 RX 26 S_T S_R IDLE =15ms

Figure 6.1. Single hop network and its schedule. Schedule length is 100 ms.

6.1.2 3-Hop Schedule for Data Collection (Convergecast)

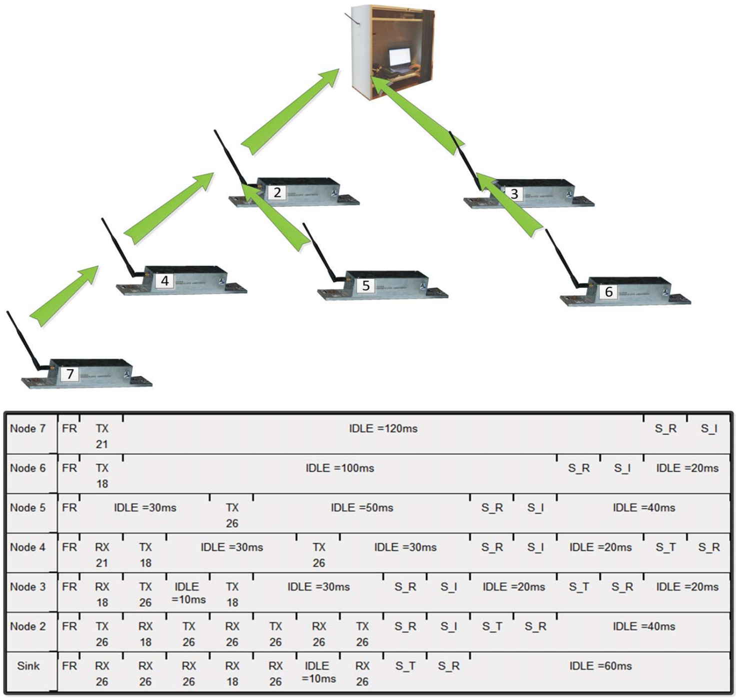
Figure 6.2 shows a 3 hop network topology and the schedule generated for this network. There are 6 network nodes and a gateway. Note that all the network nodes are assigned a link to the Sink node within one superframe. This means that within the superframe length, which is 155 ms in this case, every node in the network can transmit a packet to the sink. Table 6.2 shows the convergecast schedule network table. Note that every event in the schedule is labeled according to Table 3.2. Channels used in these slots are shown as numbers below the event type in the schedule representation. Multi-channel operation can be seen in Fig. 6.2 by looking at different channels used in the simultaneous timeslots.

Table 6.2. 3 hop convergecast schedule network table

Parent	Children	
1	2	3
2	4	5
3	6	
4	7	

6.1.3 3-Hop Schedule for Data Collection and Dissemination

Figure 6.3 shows the network topology for both way data transport, i.e. sink-to-network and network-to-sink. In this case every node in the net-



Node 7	FR	TX 21	IDLE =120ms											S_R	S_I
Node 6	FR	TX 18	IDLE =100ms									S_R	S_I	IDLE =20ms	
Node 5	FR	IDLE =30ms			TX 26	IDLE =50ms					S_R	S_I	IDLE =40ms			
Node 4	FR	RX 21	TX 18	IDLE =30ms			TX 26	IDLE =30ms			S_R	S_I	IDLE =20ms		S_T	S_R
Node 3	FR	RX 18	TX 26	IDLE =10ms	TX 18	IDLE =30ms			S_R	S_I	IDLE =20ms		S_T	S_R	IDLE =20ms	
Node 2	FR	TX 26	RX 18	TX 26	RX 26	TX 26	RX 26	TX 26	S_R	S_I	S_T	S_R	IDLE =40ms			
Sink	FR	RX 26	RX 26	RX 26	RX 18	RX 26	IDLE =10ms	RX 26	S_T	S_R	IDLE =60ms					

Figure 6.2. 3 hop convergecast topology and schedule. Schedule length is 155ms.

work can transmit one packet to the sink and receive one packet from the sink within one superframe, which is 245 ms in this case. The network table for this case is the same as in Table 6.2.

6.1.4 Notes on A-Stack Schedule

There are many possibilities when forming A-Stack schedules. The required time-slots for scheduler to run are *SERVICE_TX_SLOT*, *SERVICE_RX_SLOT*, *SERVICE_IDLE_SLOT*, *FRAME_SLOT_START*. The advertisements used for configuration, time-synchronization beacons and configuration packets are transmitted within *SERVIVCE_TX_SLOTs* and received within *SERVICE_RX_SLOTs*. Furthermore, commands and data can be sent and received by only using these slots as long as the traffic is not very high. When nodes are required to transmit data to a central location, they would need transmission slots, and such an example is given in Section 6.1.2. When the sink is also required to disseminate a large amount of data, a schedule like in Section 6.1.3 can be used. Furthermore, if some nodes work as sensors, and some as actuators, transmission and reception slots in the schedule can be set for them.

The duty cycles of the nodes can be calculated from these schedules. For

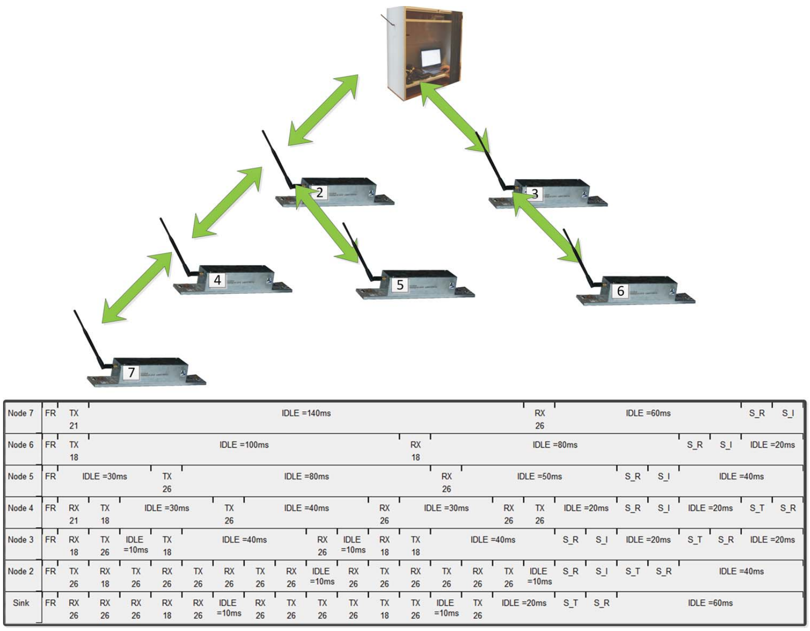

2
3
4
5
6
7
Node 7 FR TX 21 IDLE =140ms RX 26 IDLE =60ms S_R S_I
Node 6 FR TX 18 IDLE =100ms RX 18 IDLE =80ms S_R S_I IDLE =20ms
Node 5 FR IDLE =30ms TX 26 IDLE =80ms RX 26 IDLE =50ms S_R S_I IDLE =40ms
Node 4 FR RX 21 TX 18 IDLE =30ms TX 26 IDLE =40ms RX 26 IDLE =30ms RX 26 TX 26 IDLE =20ms S_R S_I IDLE =20ms S_T S_R
Node 3 FR RX 18 TX 26 IDLE =10ms TX 18 IDLE =40ms RX 26 IDLE =10ms RX 18 TX 18 IDLE =40ms S_R S_I IDLE =20ms S_T S_R IDLE =20ms
Node 2 FR TX 26 RX 18 TX 26 RX 26 TX 26 RX 26 TX 26 RX 26 IDLE =10ms RX 26 TX 26 RX 26 TX 26 RX 26 TX 26 IDLE =10ms S_R S_I S_T S_R IDLE =40ms
Sink FR RX 26 RX 26 RX 26 RX 18 RX 26 IDLE =10ms RX 26 TX 26 TX 26 TX 26 TX 18 TX 26 IDLE =10ms TX 26 IDLE =20ms S_T S_R IDLE =60ms

Figure 6.3. 3 Hop network topology and schedule for data collection and dissemination. Schedule length is 245 ms.

example in Fig. 6.3, Node 7 has one TX slot, one RX slot and one *S_R* slot. Thus minimum duty cycle would be

(RX_slot_length+S_R_slot length) / superframe_length=(20 / 245)=8.2%

Maximum duty cycle would be

(RX_slot_length+TX_slot_length+S_R_slot length)/super_frame_length =(30/245)=0.112=11.2%

If there is no packet to transmit, the radio is not turned on during TX or *S_T* slots. This is why there is a minimum and maximum duty cycle. Note also that the superframe frequency is another parameter that affects the duty cycles. If it is set to 1, then all the superframes are active superframes, but if it is set to 5, then the actual duty cycles would be one fifth of what was calculated above.

6.2 A-Stack Tests

A-Stack is tested and verified by using a series of tests with varying configurations. The aim of these tests is to show the operation, verify the system reliability and to observe the achievable communication reliability of time-synchronized and time-slotted communication over different radio channels.

6.2.1 Application and Development

Schedules generated for this application are given in Fig. 6.4 and Fig. 6.5. These schedules utilize only one channel for the entire network, i.e. operation is not multi-channel. However, throughout the tests the channel used is updated in order to evaluate performance of different channels at different links at different times. Same tests can be done with a multi-channel schedule as well; however in this case we wanted to see the performance of a single channel at different links at the same time and also the variation of this performance over time.

A sample application is created for the tests. In these tests nodes are assigned 10 $msec$ communication slots. A guard time offset of 3 $msec$ is added to the transmission slots. Synchronization and advertisement beacons are sent in every 10 super frames. Note that these tests were done with an older version of A-Stack in which advertisement and synchronization beacons are sent separately. In the current version, they are sent together within *SERVICE_TX* slots. In this application, the sink node polls all the nodes one by one by sending a 96 Byte packet. One packet is sent from the sink to each node at every second. The node that receives a packet will reply to the sink. Both way latency, received signal strength indicator (RSSI), link quality indicator (LQI) and packet delivery ratio (PDR) are saved for every node in the PC using MATLAB. Description of the tests is given in Table 6.3.

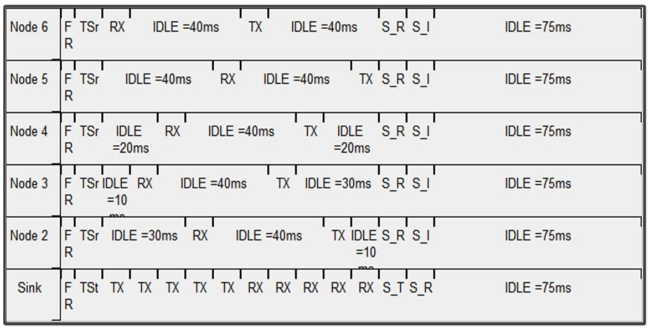

Node 6 | FR | TSr | RX | IDLE =40ms | TX | IDLE =40ms | S_R | S_I | IDLE =75ms
Node 5 | FR | TSr | IDLE =40ms | RX | IDLE =40ms | TX | S_R | S_I | IDLE =75ms
Node 4 | FR | TSr | IDLE =20ms | RX | IDLE =40ms | TX | IDLE =20ms | S_R | S_I | IDLE =75ms
Node 3 | FR | TSr | IDLE =10 ms | RX | IDLE =40ms | TX | IDLE =30ms | S_R | S_I | IDLE =75ms
Node 2 | FR | TSr | IDLE =30ms | RX | IDLE =40ms | TX | IDLE =10 ms | S_R | S_I | IDLE =75ms
Sink | FR | TSt | TX | TX | TX | TX | TX | RX | RX | RX | RX | RX | S_T | S_R | IDLE =75ms

Figure 6.4. Schedule for one-hop tests with a super frame length of 210 msec.

Deployments used in tests are given in Fig. 6.6. When creating multi-hop schedules, transmission pipelining is used in order to decrease latencies. A pipeline between sink node and node 6 is indicated in the schedule given in Fig. 6.5. During the tests, channels used in TX and RX slots were updated periodically by a service message created by controller in PC. Six

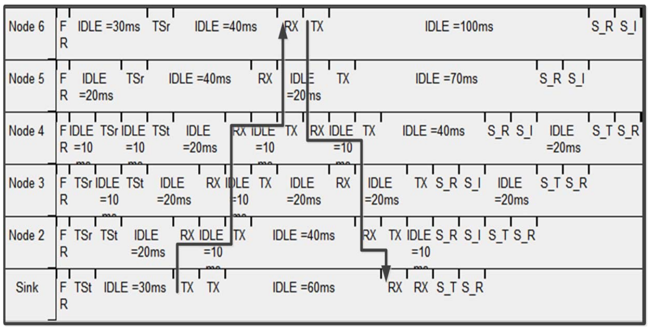

Node 6 FR IDLE =30ms TSr IDLE =40ms RX TX IDLE =100ms S_R S_I
Node 5 FR IDLE =20ms TSr IDLE =40ms RX IDLE =20ms TX IDLE =70ms S_R S_I
Node 4 FR IDLE =10 TSr IDLE =10 TSt IDLE =20ms RX IDLE =10 TX RX IDLE =10 TX IDLE =40ms S_R S_I IDLE =20ms S_T S_R
Node 3 FR TSr IDLE =10 TSt IDLE =20ms RX IDLE =10 TX IDLE =20ms RX IDLE =20ms TX S_R S_I IDLE =20ms S_T S_R
Node 2 FR TSr TSt IDLE =20ms RX IDLE =10 TX IDLE =40ms RX TX IDLE =10 S_R S_I S_T S_R
Sink FR TSt IDLE =30ms TX TX IDLE =60ms RX RX S_T S_R

Figure 6.5. Schedule used in multi-hop tests. Superframe length is 500 msec. (Last IDLE slots in the schedule are not shown). Transmission pipelining is shown on the schedule by using the arrows.

channels (13, 15, 18, 21, 24, and 26) out of 16 available channels (11-26) are used in tests 2, 3 and 4. After polling all nodes 10 times, the network shifts to a new channel and data collected in the previous round is saved so that the wireless channels performance over time can be tracked.

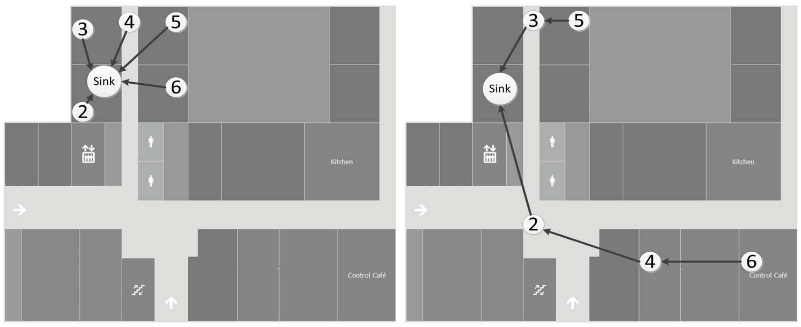


Figure 6.6. Test deployments: Test 3 on the left hand side and Test 4 and on the right hand side. Single room deployments, Test 1 and Test 2, are not shown here.

Table 6.3. Description of the tests

Test	Length	Polls	# of Nodes	Description
1	3 hours	1980	5	single room, multi-channel, one-hop
2	26 hours	2040	5	single room, multi-channel, one-hop
3	139 hours	11640	5	multi-room, multi-channel, one-hop
4	69 hours	6470	5	multi-room, multi-channel, multi-hop

6.2.2 Results

Table 6.4 shows the packet delivery ratios obtained during the tests. Best performing channels are shown in boldface. Number of polls per node and per channel can be found in Table 6.3. Note that when a node is polled, there are 2 packet transmissions (sink to node and node to sink) and corresponding acknowledgements. PDRs are calculated by using this two way communication results.

Table 6.4 shows that channel selection has a high impact (more than 10% in some cases) on the communication performance especially when the nodes are not within line of sight such as in tests 3 and 4. In test 3, node 5 ran out of battery which is the reason for low reception rates. In all the tests, the best performing channels seem to vary for different links. Figure 6.7 shows RSSI measurements of node 2 during test 3. Every measurement point in the Fig. 6.7 is an average of 10 measurements. It is seen that RSSI does not vary during the nights and weekends since the activity in the laboratory environment is limited, but it varies considerably at daytime due to Wi-Fi, Bluetooth interference and human movements. These tests give an insight on how to choose and update channels for different wireless links over long periods of time.

Table 6.4. Packet Delivery Ratios (PDRs) Achieved During the Tests

		Node ID				
Test No.	**CH**	**2**	**3**	**4**	**5**	**6**
1	13	99.85	99.75	99.95	99.9	99.9
	15	99.95	**100**	**100**	99.85	**100**
	18	99.95	**100**	99.26	99.8	99.9
	21	**100**	**100**	**100**	99.75	99.85
	24	**100**	**100**	99.9	99.9	**100**
	26	**100**	**100**	**100**	**100**	**100**
2	13	**100**	99.97	99.9	**66.44**	**95.4**
	15	**100**	**100**	99.98	64.55	93.86
	18	**100**	99.99	99.64	64.66	86.17
	21	**100**	94.38	**100**	64.6	91.73
	24	99.98	99.99	99.99	64.63	94.05
	26	**100**	**100**	**100**	64.5	87.59
3	13	95.75	**100**	91.79	99.89	88.62
	15	**100**	**100**	96.58	**99.97**	95.55
	18	99.95	**100**	94.91	97.74	94.1
	21	99.95	**100**	99.15	99.4	98.83
	24	99.91	**100**	99.55	**99.97**	99.49
	26	99.74	**100**	**99.61**	99.67	**99.55**

As explained in Section 5.3.1, nodes disconnect themselves by going to a JOIN state when they stop hearing the advertisements within their SER-

VICE_RX_SLOTs and they send a join request. During test 4, nodes 2, 4 and 6 could not receive synchronization packets for 1.5 minutes, and they disconnected from the network, but they recovered after a short while. This demonstrates that the long-term operation of the stack is maintained, even though the individual nodes disconnect from the network.

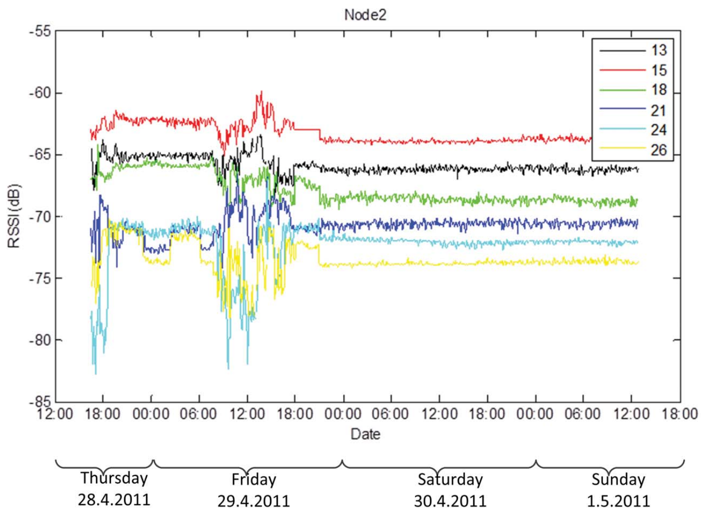


Figure 6.7. Node 2 RSSI measurements for different channels in test 3.

Table 6.5 shows measured latencies for node 5 in test 4. The unit used in the table is super frame. "0" means that the packet is received within the same super frame. As it can be seen from the schedule in Fig.6.5, Node 5 receives packets from sink through node 3 within 30 $msec$ in the same super frame. Sink to node latencies are higher since the sink node generates packets independent of the schedule, whereas the nodes respond right after they receive a packet.

The PDR results can be improved by implementing an end-to-end transport protocol; however, it is meaningful only if data is transferred within a real-time deadline which is application dependent. In these test cases, we wanted to see the system reliability, achievable PDRs and variation in RSSI in different radio channels over time. Note that in the tests, channels are changed from a central controller, which means that a higher level logic that takes into account the channel characteristics for updating the channels can easily be implemented.

Table 6.5. Measured Latencies in Terms of Frames (For Node 5, Test 4).

	Channel number					
	13	**15**	**18**	**21**	**24**	**26**
Sink to Node						
Min.	0	0	0	0	0	0
Max.	2	1	2	1	1	1
Avg.	0.89	0.89	0.89	0.89	0.89	0.89
Node to Sink						
Min.	0	0	0	0	0	0
Max.	1	1	1	1	1	1
Avg.	~0	~0	~0	~0	~0	~0

6.3 Structural Health Monitoring Deployment using A-Stack

Civil structures deteriorate over time due to harsh environmental conditions, hurricanes, earthquakes, corrosion and fatigue. Structural health monitoring (SHM) can diagnose the structure's condition during its lifetime using data collected from sensors. Intelligent Structural Health Monitoring (ISMO) project in Aalto University aims at developing efficient and effective monitoring systems that would provide reliable information about the structure's condition and replace visual inspections. Wireless sensor networks are a promising technology for SHM due to their ease of installation and low costs. On the other hand, high data rates, accurate and time-synchronized sampling required for SHM brings challenges. Furthermore a wireless SHM system should consider energy management, scalability, reliability and usability.

A-Stack has been used in a SHM deployment on a footbridge in Espoo, Finland in November 2011. Figure 6.8 shows the footbridge and wireless sensor nodes deployed. In this deployment, a 3-hop WSN consisting of 9 nodes are placed on the bridge, vibration measurements were collected and evaluated at a central PC. Results are presented to a client application running at a remote location.

3 axis digital accelerometers (LIS3LV02DQ by STMicroelectronics, 7 x 7 x 1.5 mm) were used for measuring vibrations. The 30 second measurements at 125 Hz resulted in 3.75 MB data per node per session. Sampling at the nodes is triggered from a timer callback function based on the synchronized clock. Operation interval was set to 10 minutes. This interval includes synchronous sampling, collection of 33.75 MB data from the net-

work, recovering lost packets, and transferring collected data to a client running on a remote PC. Schedule length was 235 $msec$.

A-Stack enables multiple data flows at the same time. All the nodes in the network can send one packet within one superframe, which means 9 data packets would reach the sink node within 235 $msec$. One packet includes 72 bytes of measurement data which results in 22 $Kbps$ effective data transfer rate excluding MAC and application layer overheads. Two types of schedules are used in the deployment: active and passive. Active frames are used when collecting the data and all the communication slots were active during these frames. WCP_APP.superframe_freq variable was set to 1 during active frames. In passive frames, frequency of active slots was reduced 9 times, i.e. WCP_APP.superframe_freq variable was set to 9, in order to reduce power consumption and increase the network lifetime.

Operation flow is given in the Figure 6.9. Network topology and associated schedule is shown in Figure 6.10. A-Stack's multi-hop synchronized network enabled synchronous accelerometer sampling required for SHM. A-Stack has shown to be a useful tool for SHM applications that require high data rates, accurate sampling, low power consumption and long term reliable operation.

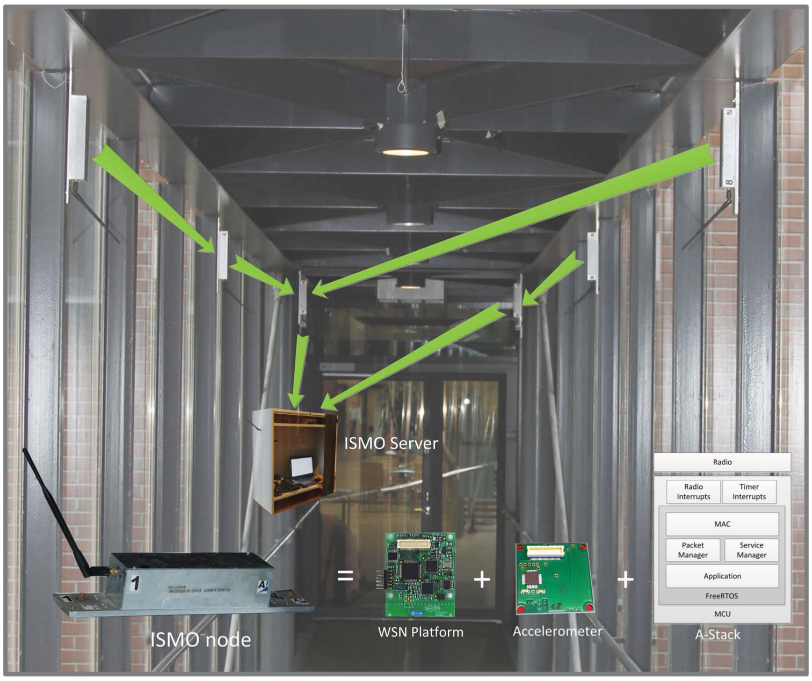


Figure 6.8. Structural health monitoring system deployment on a footbridge using A-Stack

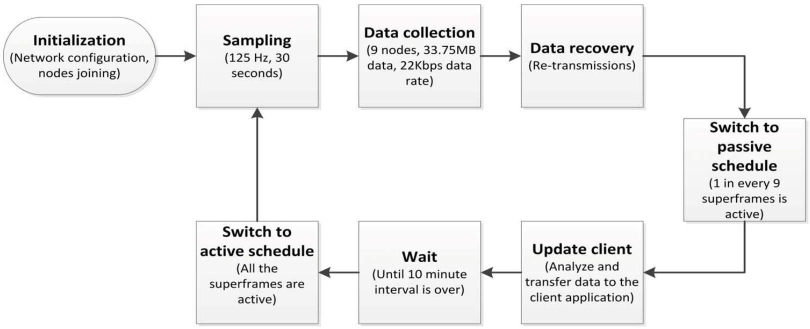

Initialization
(Network configuration, nodes joining)
Sampling
(125 Hz, 30 seconds)
Data collection
(9 nodes, 33.75MB data, 22Kbps data rate)
Data recovery
(Re-transmissions)
Switch to passive schedule
(1 in every 9 superframes is active)
Update client
(Analyze and transfer data to the client application)
Wait
(Until 10 minute interval is over)
Switch to active schedule
(All the superframes are active)

Figure 6.9. Operation flow of the structural health monitoring system

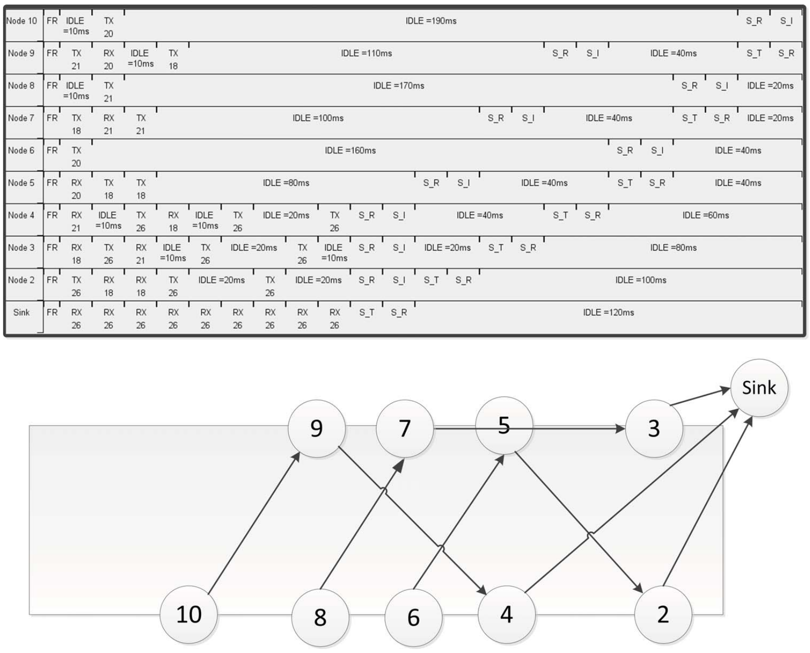

Node 10 FR IDLE =10ms TX 20 IDLE =190ms S_R S_I
Node 9 FR TX 21 RX 20 IDLE =10ms TX 18 IDLE =110ms S_R S_I IDLE =40ms S_T S_R
Node 8 FR IDLE =10ms TX 21 IDLE =170ms S_R S_I IDLE =20ms
Node 7 FR TX 18 RX 21 TX 21 IDLE =100ms S_R S_I IDLE =40ms S_T S_R IDLE =20ms
Node 6 FR TX 20 IDLE =160ms S_R S_I IDLE =40ms
Node 5 FR RX 20 TX 18 TX 18 IDLE =80ms S_R S_I IDLE =40ms S_T S_R IDLE =40ms
Node 4 FR RX 21 IDLE =10ms TX 26 RX 18 IDLE =10ms TX 26 IDLE =20ms TX 26 S_R S_I IDLE =40ms S_T S_R IDLE =60ms
Node 3 FR RX 18 TX 26 RX 21 IDLE =10ms TX 26 IDLE =20ms TX 26 IDLE =10ms S_R S_I IDLE =20ms S_T S_R IDLE =80ms
Node 2 FR TX 26 RX 18 RX 18 TX 26 IDLE =20ms TX 26 IDLE =20ms S_R S_I S_T S_R IDLE =100ms
Sink FR RX 26 RX 26 RX 26 RX 26 RX 26 RX 26 RX 26 RX 26 RX 26 S_T S_R IDLE =120ms
Sink
9
7
5
3
10
8
6
4
2

Figure 6.10. Schedule and network topology used in the structural health monitoring system

7. Future Development and Conclusions

In this chapter, we present the open issues and future development of A-Stack and summarize our contributions.

7.1 Future Development and Known Issues

When using A-Stack, as well as NanoStack, one has to be careful about allocation and freeing of the buffers. If there is a mismatch in this dual operation, nodes will get stuck eventually.

Task priorities should be chosen carefully. If one needs to give higher priority to the application, packet manager, and service manager task priorities can be decreased, and application task priority can be increased. However, the MAC task is highly time-critical, and it is not recommended to decrease its priority.

When running long term tests, debug lines should be commented as it decreases robustness, due to non-deterministic behavior. This point was mentioned also in NanoStack documentation.

In the current implementation, the sink node is quite resource limited due to the overhead caused by the communication with the PC. On the other hand, there should be plenty of coding space in the network nodes. In case memory problems are faced when programming, one should optimize the memory allocation parameters in app_config.h, as described in Section 5.4.3.

Packets are using MAC headers, but there is no footer that would validate the correctness of the packet after it is received in the radio. A CRC check should be implemented to increase reliability.

In the communication between PC (MATLAB) and sink node, there is no checksum implementation and this can be added in the future.

7.2 Conclusions

This report presents the A-Stack, a real-time protocol stack for time-synchronized multi-channel and slotted communication in multi-hop wireless networks. A-Stack is designed to provide flexibility when prototyping reliable and real-time WSN applications. An important characteristic of the stack is that multi-hop and multi-channel schedules can be formed in a PC, making A-Stack adaptive to various network topologies and application communication requirements. The real-time operating system used in A-Stack enables application tasks to be easily developed and re-used. The stack provides online configuration and network reliability tools such as node joining and re-joining services as well as dynamic channel hopping. Tests done under varying configurations shows that A-Stack provides a useful environment for developing low-latency and high reliability real-time WSN applications and protocols.

Bibliography

[1] K. S. J. Pister, L. Doherty, "TSMP: Time Synchronized Mesh Protocol," In Proc. of IASTED Int. Symposium on Distributed Sensor Networks (DSN), Florida, USA, 2008.

[2] K. K. Chintalapudi, L. Venkatraman, "On the Design of MAC protocols for Low-Latency Hard Real-Time Discrete Control Applications Over 802.15.4 Hardware," In Proc. of Int. Conference on Information Processing in Sensor Networks, (IPSN'08), 2008.

[3] J. Yick, B. Mukherjee and D. Ghosal, "Wireless Sensor Network Survey," Computer Networks, 52, pp. 2292-2330, 2008.

[4] P. Di Marco, P. Park, C. Fischione, K.H. Johansson, "TREnD: A Timely, Reliable, Energy-Efficient and Dynamic WSN Protocol for Control Applications," 2010 IEEE International Conference on Communications (ICC), pp.1-6, 23-27 May 2010.

[5] TinyOS, [online] http://www.tinyos.net/, 2011.

[6] Contiki, [online] http://www.sics.se/contiki/, 2011.

[7] FreeRTOS, [online] http://www.freertos.org/, 2011.

[8] J. Song, S. Han, A.K. Mok, D. Chen, M. Lucas, M. Nixon, "WirelessHART: Applying Wireless Technology in Real-Time Industrial Process Control," IEEE Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08, pp.377-386, 22-24 April 2008.

[9] R. Flury, R. Wattenhofer, "Slotted Programming for Sensor Networks," IPSN '10 Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, 2010.

[10] P. Suriyachai, J. Brown, U. Roedig, "Time-Critical Data Delivery in Wireless Sensor Networks," International Conference on Distributed Computing in Sensor Systems (DCOSS10), 2010.

[11] B. Raman, K. Chebrolu, S. Bijwe, and V. Gabale, "PIP: a connection-oriented, multi-hop, multi-channel TDMA-based MAC for high throughput bulk transfer," in Proc. SenSys, 2010.

[12] E. I. Cosar, A. Mahmood, M. Björkbom, "A-Stack: A Real-Time Protocol Stack for IEEE 802.15.4 Radios", IEEE SenseApp, 2011.

[13] Nanostack, http://www.sensinode.com/

[14] E. Mutanen, "WirelessTools v1.1 documentation", Report, Aalto University, Department of Automation and Systems Technology, 2011.

[15] J. Pesonen, "Stochastic Estimation and Control over WirelessHART Networks: Theory and Implementation," Masters Thesis, Royal Institute of Technology, KTH, Feb.2010.

[16] S. Lembo, J. Kuusisto, J. Manner, "In-Depth Breakdown of a 6LoWPAN Stack for Sensor Networks," International Journal of Computer Networks and Communications (IJCNC) Vol.2, No.6, 2010.

[17] A. Mahmood, A. and R. Jäntti, "Time Synchronization Accuracy in Real time Sensor Networks," In Proc. of 9th IEEE Malaysia Int. Conf. on Communications (MICC'09), Kuala Lumpur, Malaysia, 2009.

[18] M. Bocca, A. Mahmood, L. M. Eriksson, J. Kullaa, and R. Jäntti, "A Synchronized Wireless Sensor Network for Experimental Modal Analysis in Structural Health Monitoring," In Computer-Aided Civil and Infrastructure Engineering (CACAIE), December 2010.

[19] M. Maréti, B. Kusy, G. Simon, Á. Lédeczi, "The flooding time synchronization protocol," In Proc. of 2nd Int. Conference on Embedded networked sensor systems, Baltimore, USA, pp. 39-49, 2004.

[20] Q. M. Chaudhari, E. Serpedin, "Clock estimation for long-term synchronization in wireless sensor networks with exponential delays," EURASIP J. on Advances in Signal Processing, vol. 2008, no.27, 2008.

[21] H. Yigitler, A. Mahmood, R. Virrankoski, R. Jäntti, "Time Synchronization for Wireless Network Nodes: From Oscillators to Network Time," submitted to IET-Wireless Sensor Systems, 2011

Cosar et al. A-Stack

Aalto-ST 2/2012

Wireless
Sensor
Systems
Group

9 789526 044675

ISBN 978-952-60-4467-5
ISBN 978-952-60-4468-2 (pdf)
ISSN-L 1799-4896
ISSN 1799-4896
ISSN 1799-490X (pdf)

Aalto University
School of Electrical Engineering
Department of Automation and Systems Technology
www.aalto.fi

Aalto University