Analysis of TSCH Code on Wismote

Technical Report · July 2016

DOI: 10.13140/RG.2.1.1402.0082

2 authors:

Abby P Joby
National Institute of Technology Calicut
10 PUBLICATIONS **9** CITATIONS
SEE PROFILE

Arun Sasi
National Institute of Technology Calicut
3 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Analysis of TSCH Code on Wismote

Abby P Joby
Department of Electronics and
Communication Engineering
National Institute of Technology, Calicut

Arun Sasi
Department of Electronics and
Communication Engineering
National Institute of Technology, Calicut

***Abstract*—Time slotted channel hopping (TSCH) is a combination of two efficient methods namely time synchronisation and channel hopping attempted at creating a low power, highly reliable operation. It has been implemented successfully in the field of Internet of Things using the Contiki operating system in NXP JN516x, Tmote Sky, Zolertia Z1, CC2538DK, Zolertia Zoul and CC2650 hardware platforms. Implementing a minimal configuration of TSCH (A basic RPL + TSCH network) on platforms involve configuring their radio drivers and other timing related configuration parameters. Porting TSCH code to wismote (msp430f5437 + cc2520) has been unsuccessful owing to synchronisation issues. TSCH involves real time task scheduling of communication slots using the inherent rtimer. Any synchronisation problems would have come as a result of portability issues with the TSCH code, Radio driver configuration issues or timer desynchronisation. Attempts at decoding the root cause would involve an elaborate study on the TSCH code and continuous debugging of the hardware. We show that these problems were a result of rtimer discrepancies on the different wismotes.**

I. Introduction

Wireless sensor networks (WSN) consist of a large number of tiny wireless devices called nodes connected together. A node has several parts such as a radio transceiver, a microcontroller and a power source (typically battery). The microcontroller used in the nodes are highly resource constrained because they have to be compact and low power devices. Typically these microcontrollers have code memory of the order of 100 Kilobytes, and less than 20 Kilobytes of RAM. These limitations in association with a requirement for implementing internet protocols on these small devices motivated developers to come up with lightweight operating systems that run on these microcontrollers.

A. 6LoWPAN

First of all, to connect these nodes to the internet and ensure it is compatible with the existing internet infrastructure, there arises a need to assign IP addresses to each node which is easily accomplished by assigning an IPv6 address to the nodes. The IETF (internet engineering task force) 6LoWPAN working group was created to tackle the problems of implementing internet protocols on small embedded devices.
IEEE 802.15.4 is a standard that targets the low-power personal area networks(LoWPAN) and 6LoWPAN provides specifications for transmitting IPv6 over IEEE 802.15.4. Also IEEE 802.15.4e is an amendment to the medium access control (MAC) protocol defined by the IEEE 802.15.4 standard. The TSCH mode of IEEE 802.15.4e has been implemented in Contiki across various platforms.

B. Contiki

Contiki is an operating system for wireless microcontrollers that makes it possible for them to connect to the Internet. It is the software that is burned into the flash memory of the microcontroller. Contiki is implemented in the C language and has been ported to a number of microcontroller architectures, including Texas Instruments MSP430 and the Atmel AVR. Instant-Contiki is basically a version of Ubuntu Linux that has all the tools pre-installed to program the wireless sensor nodes called motes. It also includes the cooja simulation environment. A typical contiki configuration is 2 kilobytes of RAM and 40 kilobytes of ROM [1].

C. TSCH using Contiki

Even though Time Slotted Channel Hopping (TSCH) has already been ported to various hardware platforms, the process involved in developing driver codes for a new platform is highly technical and time consuming. In a TSCH network, time is sliced up into slots. A time slot is long enough to accommodate a MAC frame of maximum size and it's ACK. A slotframe consists of 10's to 1000's of time slots grouped together and it repeats over time. A TSCH schedule instructs each node what to do in each time slot - transmit, receive or sleep.

II. Process of deduction

A. Data acquisition

The process began with setting up the environment - Cooja simulator on the Instant-Contiki v2.7 and getting familiarised with the simulation process using simple examples running on Rime communication stack and using basic utilities of the given mote. It was followed by a deep understanding of the various timers in Contiki such as timer, stimer, etimer, ctimer and rtimer in which the rtimer allows real time task scheduling. TSCH is implemented on a frame format that derives itself from RFC7554 based on IEEE 802.15.4e and is implemented on the basis of Minimal 6TiSCH Configuration[2].
This was followed by an extensive study on the executable TSCH code(examples/ipv6/rpl-tsch/node.c), the TSCH header files and their associated '.c' files(core/net/mac/tsch/) and the driver codes (dev/cc2520/).

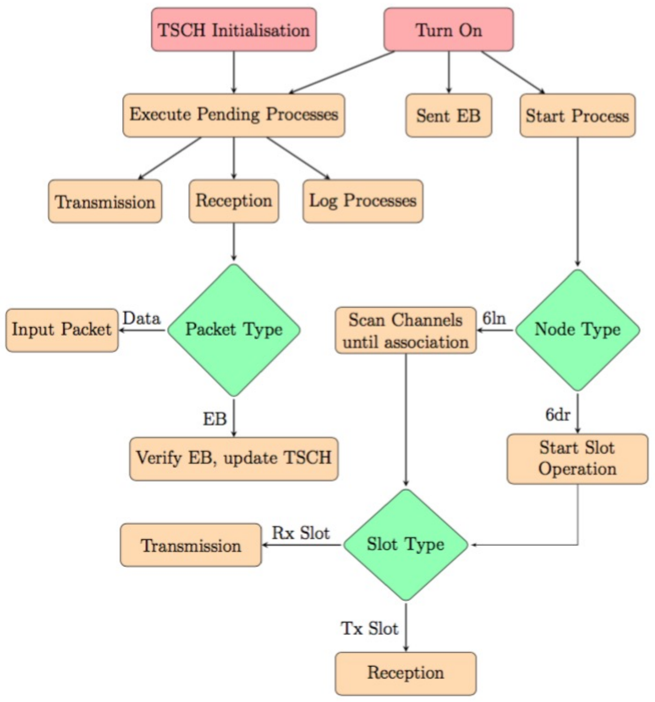

Fig. 1. TSCH process flow chart in contiki

B. *Software analysis*

Upon analysing the code, flow of the TSCH algorithm was keenly observed as shown in Figure 1. The program began with one of the nodes continuously hopping it's channel and sending out EB packets. A new node that wishes to join this network would randomly listen on a particular channel and when it receives an EB, it attempts to join the network. The contents of EB is used to synchronise the nodes at the time of association. Once association is done, keep-alive packets are sent at regular intervals to resynchronise the nodes and the TSCH slot operation process kicks to determines the active slots.

In the TSCH code, time synchronisation can be verified by two means- by observing the continuity in association between two nodes and by verifying packet reception using suitable acknowledgements. For the TSCH code provided (with time offsets at TsRxTx, TsRxAckDelay and RxWait), simulations on wismote seemed to be successful with continuous association and regular ACK receptions. By removing the offsets, the expected drift diverged from the conventional value and the motes tended to oscillate between association and disassociation states. However, the significance of these offset parameters was reduced by the fact that they needed to be altered when introducing extra code segments within the code. The variable for capturing the arrival time(triggered by the FIFOP interrupt) was updated by the interrupt function within the driver code, but got passed to the TSCH code only when the arrived packet was parsed. Changing this capture process into the TSCH code was without any immediate effect on time synchronisation. These observations led to a conclusion that the alterations in TSCH code could not provide a sustainable solution to our problem.

C. *Hardware analysis*

Test involving TSCH code on the Zolertia Z1 hardware platform, provided a result similar to that of the simulation, with associations lasting for a long time with minimum drifts. This was conclusive of the fact that the given TSCH code or its accompanied header files could not be held responsible for the drifts occurring in wismotes. It was observed that after association the estimated drift value remained small usually less than 100 ticks (1 tick = 30.5 microseconds) , with the help of continuous drift corrections from the respective time sources(as determined by the rpl layer).
However, on the wismote platform, there occurred a persistent divergence in the drift where the two motes could neither agree upon the tsch slot timings nor undergo a proper drift correction. On the hardware, the motes would initially associate, send one packet and sometimes even properly receive an ACK. But eventually(usually within a couple of seconds), the ASN (slot numbers) would desynchronise and the motes would disassociate.This is followed by continuous attempts at associations in which one successful association would lead to the process repeating itself. Turning off the drift correction within the TSCH code led to an almost constant drift between any two given motes revealing the fact that the drift correction was indeed working towards reducing the drift but was unable to do so due to some unknown discrepancy.
These observations coupled with the successful execution of the modified TSCH code in the simulation enabled us to verify the validity of the Radio driver configurations. Next, we had to move on to checking the accuracy of timers and delays in communication. If the inherent timers of similar motes are not in synch upto a certain degree of accuracy, it would be extremely unlikely for them to execute TSCH properly. Using a program that triggers the rtimer at discrete intervals of time, we were able to conclude that there were no drifts in the timers. The wall clock time intervals between these triggers remained the same assuring a steady rtimer, although the precision of the experiment was not enough to compare the rtimer timings to that of the wall clock.
This was followed by a transmission-reception experiment between two motes that would enable us to measure the delay in packet reception and to find any drifts introduced by the radio driver. But, the delay introduced was not only consistent with that in the simulations (total delay = tx driver delay + packet transmission delay + packet propagation delay + rx driver delay) but also remained unwavering throughout the experiment. This experiment was attempted at verifying the steadiness of the driver delays and it proved to be successful. All these communication programs used the Rime broadcast module to transmit and receive packets while using rtimer to schedule the transmissions. (Note: Radio Duty Cycling needs to be turned off before using rtimer in the broadcast module). Having ruled out the drifts in timers and drivers, we could now

TABLE I
RELATIVE COMPARISON BETWEEN WISMOTES

Obs No	Tx	Rx	Tx Slot Difference	Rx Slot Difference	Ratio	Extra ms per second
0	w1	w2	2000	1997	1.0015	1.5
1	w1	w3	2000	1990	1.005	5
2	w2	w1	2000	2004	1.002	2
3	w2	w3	2000	1994	1.003	3
4	w3	w1	2000	2010	1.005	5
5	w3	w2	2000	2007	1.0035	3.5
6	z1	z2	2000	2000	1	0
7	z2	z1	2000	2000	1	0

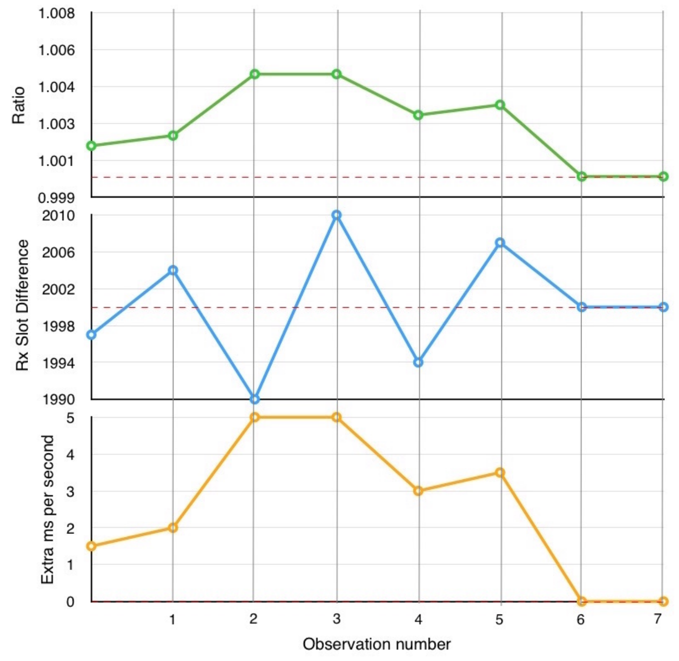

Fig. 2. Relative comparison between Wismotes

TABLE II
EXPERIMENTAL RX SLOT DIFFERENCE VALUES FOR DIFFERENT WISMOTES

Z1,Sky	w1	w2	w3	w4	w5	w6
2000	1990	1993	1999	1985	1989	1988
2000	1990	1993	1999	1985	1990	1988
2000	1990	1993	2000	1985	1989	1988
2000	1990	1993	1999	1985	1989	1988
2000	1990	1993	1999	1985	1989	1988
2000	1990	1993	2000	1985	1990	1988
2000	1990	1993	1999	1985	1989	1988
2000	1990	1993	1999	1985	1989	1988
2000	1990	1993	2000	1985	1989	1988
2000	1990	1993	1999	1985	1989	1988

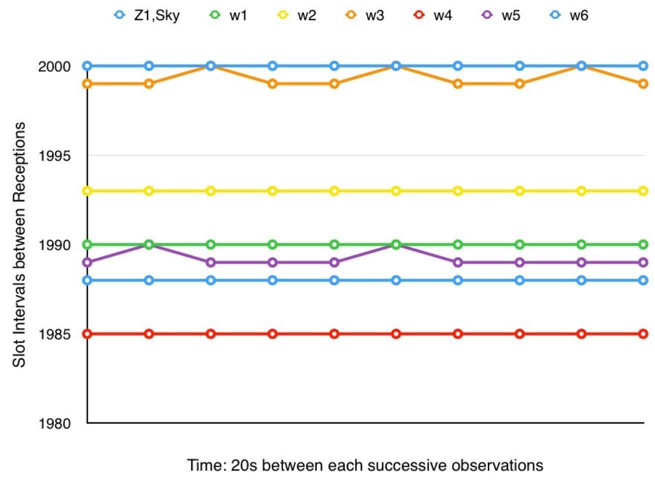

Fig. 3. Experimental Rx slot difference values for different Wismotes

move on to attempting a sample slotted scheduling program (an exoskeleton of the TSCH code) that would have to reveal the ultimate reason behind the desynchronisations.

III. APPROACHING THE PROBLEM

A. The Experiment

Being the most important timer in the context of TSCH, the rtimer permits pre-emptive task scheduling. When an rtimer fires, the function associated with it is immediately executed regardless of the process being executed at that instant.

Since the fact that rtimer of a particular wismote has no drift has been clearly established in the hardware analysis section, the main focus of this experiment was to test the accuracy of rtimers between different wismotes. Using a program that would register time in terms of the slot number and reveal it upon transmission and reception, we could simulate all the complexities of a TSCH code that might have generated the problems. This experiment involved creating slots of 10ms duration using rtimer and incrementing a count value for each slot with the count wrapping around at a value of 2000. The receiver also contained such a time slotting and count in addition to a difference variable that would also increment along with the count , but got reset only when the packet was received. The transmitter would transmit as soon as the count reached 2000 and the receiver would display the experiment variables upon obtaining the packet. The packet contained sequence numbers that would help identify the receptions corresponding to respective transmissions. The count value provided the slot at which the packets were received while the difference variable counted the number of slots between successful receptions.

The first experiment involved one of the wismotes transmitting packets at an interval of 20 seconds while the others receiving it and displaying experiment parameters. The experiment contained usage of three wismotes in order to find their relative speed differences. The values obtained from the experiment included the number of slots between consecutive receptions(Rx Slot Difference) from which the speed ratio and timer inaccuracies could be calculated as shown in Table I and plotted in Figure 2.

In order to conclude the observations in the previous experiment, we had to check the validity of the results against a working standard. The Sky and Z1 motes were used as controls in order to verify the accuracy of the code to be 5 microseconds every 20 seconds. All six of the wismotes were

TABLE III
COMPARISON OF WISMOTES AGAINST ONE Z1 MOTE

Obs No	Rx	Tx	Tx Slot Difference	Rx Slot Difference	Ratio	Extra ms per second
0	z1	w1	2000	1990	1.005	5
1	z1	w2	2000	1993	1.0035	3.5
2	z1	w3	2000	1999	1.0005	0.5
3	z1	w4	2000	1985	1.0075	7.5
4	z1	w5	2000	1989	1.0055	5.5
5	z1	w6	2000	1988	1.006	6
6	z1	z1	2000	2000	1	0

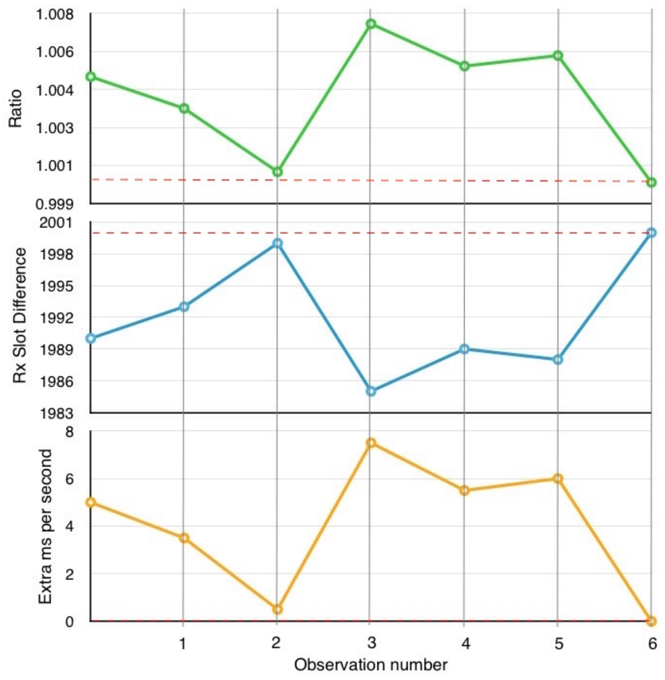

Fig. 4. Comparison of Wismotes against one Z1 mote

tested against a Z1 mote to obtain the readings as shown in Table II and plotted in Figure 3. From these observations, other parameters are calculated and recorded in Table III which are extrapolated visually in Figure 4.

B. Observations

The transmitter transmitted at a regular interval of 2000 slots (20s). This time gap was precise enough to rule out the interferences due to delay in print statements and to compare the rtimer values of the two motes in the communication pair. The synchronization issues in the actual TSCH code had to manifest in some manner in this experiment and the only untested variant in this experiment was the validity of rtimer ticks.

From the first experiment, it was evident that the slot timings for different wismotes were different leading to a conclusion that there were irregularities in their clocks. Using two Z1 motes in a control experiment, we could verify the above conclusion. The timer jitter were of the order of 1 to 5 milliseconds for every second which was observed as 1 to 10 slot differences for every 20 seconds.

In the next experiment, the Rx Slot Differences for the various wismotes against a standard Z1 mote was found to comply with the observations of the previous experiment. The jitter were of the order of 1 to 10 milliseconds for every second which was observed as 1 to 20 slot differences for every 20 seconds.

C. Verification of the Results

It was conclusive from the above observations that there existed an actual timing discrepancy between the two motes which would make it impossible to run a time sensitive code like TSCH. But if two wismotes having the same timing error could be employed in executing a TSCH code, that would serve as a conclusive proof to the experiment. The wismotes w1 and w5 were found to interpret 1 second as 1.005 seconds and 1.0055 seconds respectively. This error of 0.5 milliseconds was not large enough to drift the two motes while executing a TSCH code with active drift correction. The observations included continuous association between the two motes for as long as 5 minutes and with edr values ranging from 50 to 500 ticks. This was a successful implementation of TSCH code with results agreeing to that in simulations. Therefore, it could be verified that the timing error was the only cause behind the unsuccessful attempts at implementing TSCH code in wismotes.

IV. CONCLUSION

From the experimental observations, it can be concluded that the wismote hardware is incapable of providing a time that is in accordance with the wall clock time. This causes an inability to transfer timing information between any two motes that considers time differently. This must be the reason behind the unsuccessful drift corrections where no two wismotes are able to agree upon a fixed time though they are able to reduce the drifts to a lower value before it diverges again.

It is evident that an initial study on the timing hardware could have served as an easier method to identify the synchronisation errors. The method that we have proposed could serve as this initial experiment where the standard times of different motes could be compared and verified against any other standard.

REFERENCES

[1] Adam Dunkels, Bjo rn Gro nvall and Thiemo Voigt, *Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors*, Swedish Institute of Computer Science.
[2] Internet Engineering Task Force 6TiSCH minimal configuration web page, *https://tools.ietf.org/html/draft-ietf-6tisch-minimal-16*, 2016
[3] Internet Engineering Task Force RFC7554 web page, *https://tools.ietf.org/html/rfc7554*, 2015.
[4] Fredrik Osterlind, *A Sensor Network Simulator for the Contiki OS*, Swedish Institute of Computer Science, 2006.
[5] Hossam Ashtawy, Troy Brown, Xiaojun Wang and Yuan Zhang, *Take A Hike: A Trek Through the Contiki Operating System*, Department of Computer Science and Engineering Michigan State University.
[6] Texas Instruments, *CC2520 Datasheet 2.4 GHZ IEEE 802.15.4/ZigBee-ready RF Transceiver*, 2007.
[7] Texas Instruments, *CC2420 Datasheet 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver*.
[8] Sourceforge website, *http://contiki.sourceforge.net/docs/2.6/index.html*
[9] Contiki website, *http://www.contiki-os.org*.
[10] ANRG website, *http://anrg.usc.edu/contiki/index.php*.