Modelling of Half-Duplex Radio Access for HopeMesh Experimental WMN Using Petri Nets

Remigiusz Olejnik

West Pomeranian University of Technology, Szczecin
Faculty of Computer Science and Information Technology
ul. Żołnierska 49, 71-210 Szczecin, Poland
r.olejnik@ieee.org

Abstract. The article presents a problem of shared access to a radio channel in an experimental Wireless Mesh Network. Half-duplex radio access algorithm used by the radio modules used in the network has been shown, then the access problem has been thoroughly defined. Finally a shared radio channel access problem has been modelled using Petri nets.

Keywords: Wireless Mesh Network, Petri net, wireless networks.

1 Introduction

According to Akyildiz [1] "a Wireless Mesh Network (WMN) consists of mesh routers and clients where mesh routers have minimal mobility and form a mesh of self-configuring, self-healing links among themselves". The article presents a half-duplex radio access algorithm used by the radio modules in the experimental HopeMesh wireless mesh network along with proposal of its modelling using Petri nets.

HopeMesh experimental Wireless Mesh Network is composed of simple nodes based on AVR ATmega162 microcontroller with external 62256 SRAM memory chip that offers additional 32 KiB and HopeRF RFM12B radio module. Available memory can keep routing data for a maximum number of 2838 nodes – one entry in the network routing table needs of a total 11 bytes. The network stack for the network has been described in another paper in this volume.

2 Half-Duplex Radio Access

2.1 Problem Definition

The implementation outlined in [2] used an identical (physically) blocking implementation in order to send or receive data via the RFM12B hardware module. Listing 1.1 shows the algorithm used for sending data.

A. Kwiecień, P. Gaj, and P. Stera (Eds.): CN 2014, CCIS 431, pp. 108–117, 2014.
© Springer International Publishing Switzerland 2014

Listing 1.1. Sender routine for the RFM12B hardware module [2]

```
void rfTx(uint8_t data)
{
        while(WAIT_NIRQ_LOW());
        rfCmd(0xB800 + data);
}
```

This implementation physically blocks the main loop the same way as the UART algorithm shown in [2]. In this case the algorithm does not wait for the status of an internal register to send data but rather waits for the external nIRQ pin from the RFM12B hardware module to go low. The official "RFM12B programming guide" [3] also proposes a physically blocking algorithm.

The goal of this paper is to show improvement of the algorithm in a similar fashion as the UART algorithm. The nIRQ pin of the RFM12B was connected to the INT0 pin of the ATmega162 microprocessor allowing to execute the SIG_INTERRUPT0 interrupt service routine asynchronously. But it turned out that the implementation could not be reused at all. The RFM12B radio hardware imposes the following algorithmic challenges for the driver implementation:

- **Single interrupt request for multiple events.** The RFM12 radio module uses only one nIRQ pin in order to generate an interrupt for the following events [4]:
  - The TX register is ready to receive the next byte (RGIT).
  - The RX FIFO has received the preprogrammed amount of bits (FFIT).

  The state management has to be implemented in software otherwise the current state of operation (sending or receiving) is undefined.
- **Half-Duplex operation.** The RFM12 radio module only allows either to receive or to send data at a time but not simultaneously.

The operation of the RFM12B driver algorithm was abstracted as a (proto)thread. Interestingly enough the thread has a state modelled as a state machine depending whether it receives or sends data. The following states are valid:

- **RX:** the receiving state – the thread (logically) blocks until a complete packet has been received. Whether a packet is complete or not depends on the upper network stack layers.
- **TX:** the sending state – the thread (logically) blocks until a complete packet has been sent. Again the upper network stack layers decide whether the transmission is complete or not.

The abstract algorithm is shown in Algorithm 1. Receiving data is *non-deterministic*. A packet can arrive at any time and thus the invocation of the SIG_INTERRUPT0 interrupt service routine. Therefore the algorithm sets the RX state as the *default* state for the radio thread. After receiving the whole packet the driver has to signal the receiver thread that it can process the packet.

Sending data on the other hand is *deterministic*. When a user hits the Enter key via the UART module a packet can be constructed. A sender thread

Algorithm 1. RFM12B driver thread algorithm

```
while true do
  if state is RX then
    receive data
    signal completion to receiver thread
  else if state is TX then
    send data
    signal completion to sender thread
  end if
  set state to RX
end while
```

has to inform the radio driver thread to change its state to TX and wait until the packet has been fully transmitted. It was implemented as a concurrency problem between three threads and a single resource:

- **Sender Thread:** inside the main loop – wants to acquire the control over the radio module until the transmission of a packet is complete.
- **Receiver Thread:** inside the main loop – wants to acquire the control over the radio module until the reception of a packet is complete.
- **Radio Thread (ISR):** also wants to acquire the control over the radio module until the packet reception is complete if it is in the RX state or until the packet transmission is complete if it is in the TX state.
- **Single resource:** in this case is the radio module; only one thread at a time can own the radio hardware resource.

The question is who controls the state of the radio thread and who and when acquires and releases the lock on the single resource (the radio module). This is rather a complicated algorithm which needs further research and investigation. The solution to this problem is presented in the paper.

2.2 Petri Net Model

The purpose of the model is to validate the correct behaviour of the complete algorithm. The most common models for parallel processing are: **Process calculus**, **Actor model** and **Petri nets**.

The Petri net [5, 6] model has been chosen, well known from modelling of RTOS systems [7]. This solution allows also for a visual modelling of the concurrent algorithm. Many different formal definitions for Petri nets exists. According to Peterson [5] a Petri net is composed of:

- A set of Places $P = \{p_1, p_2, \ldots p_n\}$.
- A set of Transitions $T = \{t_1, t_2, \ldots t_m\}$.
- An input function I who maps from a transition t_j to a collection of input places $I(t_j) = \{p_0, p_1, \ldots, p_i\}$.
- An output function O who maps from a transition t_j to a collection of output places $O(t_j) = \{p_0, p_1, \ldots, p_o\}$.

The dynamic behaviour of a Petri net can be modelled using marking. A marked Petri net contains one or more tokens which can only reside inside places (not transitions). Peterson [5] expresses the marking of a Petri net as a vector $\mu = \mu_1, \mu_2, \ldots, \mu_n$ which stores the number of tokens for each place p_i in a Petri net. The marking of a Petri net is not constant through the life-cycle but rather changes over time. The change of a marking will be expressed as a token movement from a place "A" p_a to a place "B" p_b. This abstraction allows to animate the change of Petri net marking by moving tokens from one place to another.

The symbols used in the paper are presented in Figure 1.

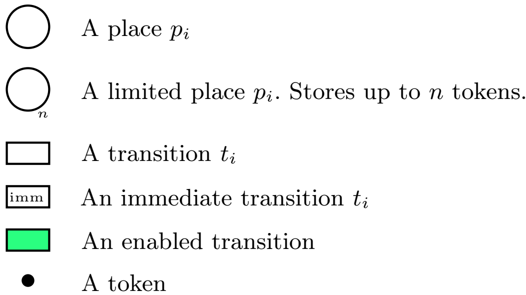

Fig. 1. Petri net symbols

But how can this abstraction be used to model concurrency? The key aspect is that a marking of a Petri net at a discrete point of time simply expresses the current state of the complete system. The location of a token (the place it currently resides in) defines the system state. A concurrently running system includes multiple states, one for each concurrently running module as was shown above. Thus a concurrent system includes multiple locations in which tokens can reside.

Regarding the initial problem we can as an example define two concurrently running modules which have to share a common resource. The following states can be defined:

- **Module 1 state.** The location of this token abstracts the current state of the first module.
- **Module 2 state.** The location of this token abstracts the current state of the second module.
- **Lock state.** The two modules both have critical section of code which must run mutually exclusive because they share a common resource. A lock (mutex) has to be introduced. The mutex token location represents the state of the mutex lock.

The common Petri net model for mutual exclusion as proposed by Peterson [5] can be seen in Fig. 2. It was used for the development of the algorithm.

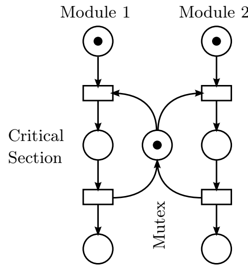

Fig. 2. Mutual exclusion Peterson Petri net model

2.3 RFM12 Driver Petri Net Model

The final algorithm for the RFM12 driver is shown in Fig. 3. It shows the three above mentioned threads in the initial states:

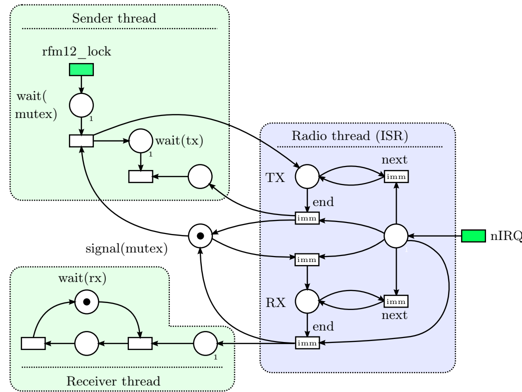

Fig. 3. Half duplex algorithm modeled as a Petri net

- **Sender thread.** The sender thread includes an always enabled transition which emits a token whenever the user prompts to send a message. When this happens it waits until it can acquire the control over the radio module in order to send a complete packet.
- **Receiver thread.** The receiver thread by default always logically blocks in a waiting state until a packet reception is complete.
- **Radio ISR thread.** The radio thread is in the RX (or idle) state by default. All transitions in the radio thread are immediate transitions. This is because the interrupt service routine itself cannot be interrupted which is a constraint defined by the hardware of the used CPU. A context switch to other threads therefore can only then happen when there are no enabled transitions in the ISR.

The Petri net behaviour is used to validate the correctness of the presented algorithm.

Data Reception. Whenever the radio module detects a valid sync pattern it will fill data into its FIFO buffer. When the reception of one byte is complete the radio hardware pulls the nIRQ pin low triggering the ISR. The model shown in Fig. 4 shows this behaviour as an infinitely enabled transition labelled as "nIRQ" which can fire at any non-deterministic time. The following Fig. 4 and the corresponding events describe the behaviour when a new byte is received by the radio.

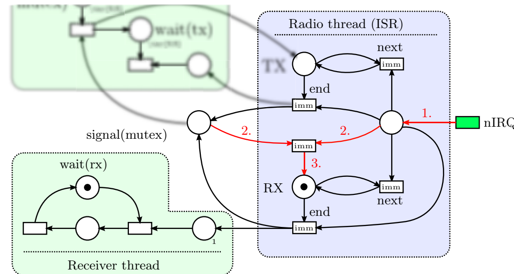

Fig. 4. nIRQ triggers reception

1. The nIRQ transition fires the ISR. An interrupt occurs caused by the radio module.
2. The radio thread being in the RX state by default tries to acquire the lock (mutex) on the radio and succeeds. Since the mutex is free the interrupt source *must* be caused by the reception of a byte.
3. The radio thread can begin its critical section by taking the received byte and delegating it to the upper network layers. The radio thread stays in the RX state and blocks the radio by not releasing the mutex.

Since the radio module has acquired the lock on the radio a sender thread will have to wait until the mutex will be released. All following nIRQ interrupt sources therefore also *must* be caused by a reception of a next byte which is shown in Fig. 5.

1. The nIRQ transition fires the ISR. An interrupt occurs caused by the radio module.
2. The radio thread is in the RX state and already acquired the lock (mutex) on the radio. It is ready to receive the next byte and delegates it to the upper network layers.
3. The upper network layers did not indicate that the reception is complete so the radio thread stays in the RX state.

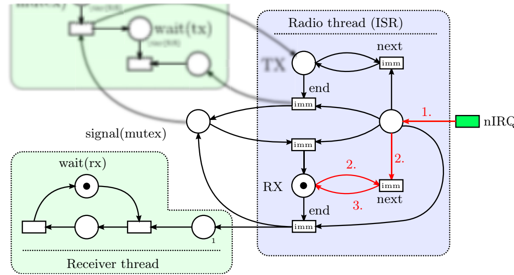

Fig. 5. Next byte reception

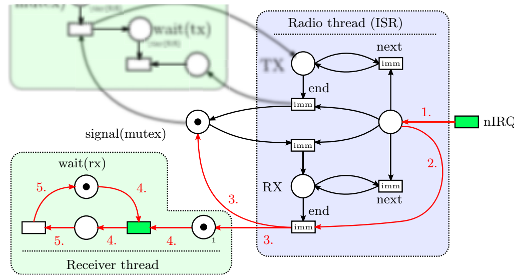

Fig. 6. Final byte reception

Still no sender thread will be able to acquire the mutex lock. The above described reception of bytes happen until the upper network layers detect the end of a frame or packet which is described in Fig. 6.

1. The nIRQ transition fires the ISR. An interrupt occurs caused by the radio module.
2. The upper network layers decide that the reception is complete. The mutex on the radio can be released.
3. After releasing the mutex the receiver thread is signalled by the radio thread that the reception of a packet is complete. The signal itself is emitted by upper network layers who need to wait for the receiver thread to process the incoming packet.
   Note that this is a limited place (currently with the limit of one token). The maximum number of token corresponds to the maximum number of packets which have to be buffered by the network stack. If the receiver thread is too slow to process incoming packets all further incoming packets will be dropped.
4. The receiver thread now changes its state. From a waiting state it switches to a processing state where it processes the received packet and i.e. displays it on the console.
5. The receiver thread finally switches back to a waiting state in order for the next packet to arrive.

Data Transmission. The data transmission in contrast to data reception is triggered by the sender thread. Thus Figure 7 shows an infinitely enabled transition in the sender thread which triggers a token (a packet to be sent) whenever the prompts a new send command. Figure 7 describes the behaviour when a new packet is sent by the sender thread.

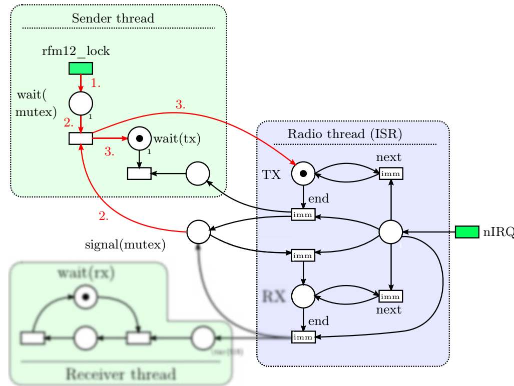

Fig. 7. Sender thread triggers transmission

1. The sender thread tries to send a new packet by changing its state to a mutex waiting state. It waits for the mutex to become free in order to acquire the lock on the radio module.
2. If the radio module becomes free the mutex can be acquired. The radio module transmitter hardware and the radio module transmitter FIFO is enabled.
3. Because the sender thread took over control it resets the radio driver thread to the new state TX. Afterwards the sender thread puts itself into a waiting state until the packet transmission is complete.

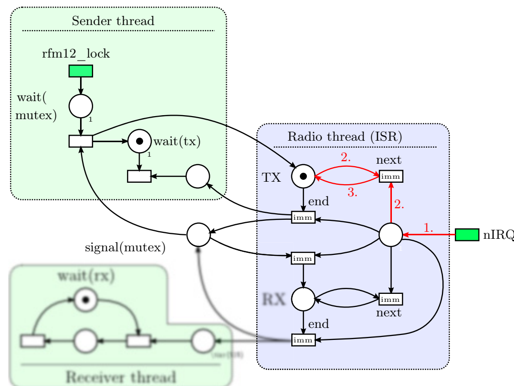

Fig. 8. Next byte transmission

The sender thread is in a different state now. It waits for the transmission of the complete packet. The transmission of the first (or next) byte is triggered now by the radio module itself. Whenever the transmission FIFO of the radio is ready the radio module hardware triggers nIRQ and the ISR fires as shown in Fig. 8.

1. The nIRQ transition fires the ISR. An interrupt occurs caused by the radio module.
2. Since the radio module is in the TX state it is ready to transmit the next byte. The upper network layers return the next byte to the radio driver.
3. The upper network layers do not indicate that the transmission is complete so the radio thread stays in the TX state.

The end of the transmission (final byte) is shown in Fig. 9.

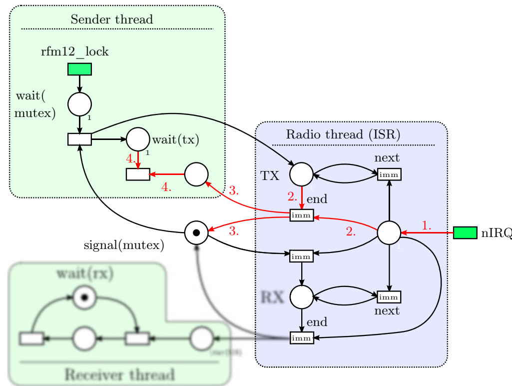

Fig. 9. Final byte transmission

1. The nIRQ transition fires the ISR. An interrupt occurs caused by the radio module.
2. The upper network layers return the next byte and indicate that this is the last byte to be transmitted.
3. The radio thread releases the mutex and signals the sender thread the transmission of the last byte.
4. The sender thread leaves the waiting state and finishes the transmission of the packet. Whenever the user prompts a new send command a new packet can be transmitted.

3 Conclusion

The paper presents the problem of shared access to radio module in HopeMesh experimental WMN. The original algorithm used for RFM12B access has been

presented, then Petri net model along with mutual exclusion problem has been introduced. Finally the RFM12B driver has been successfully modelled with Petri nets – all possible states for data reception and transmission: triggering, sending/receiving of bytes and sending/receiving of final bytes have been shown. The algorithms has been used for the implementation of data link layer as a part of the network stack of HopeMesh experimental WMN.

Acknowledgments. I would like to thank my graduate student, Sergiusz Urbaniak, who implemented my preliminary ideas of RFM12B and AVR ATmega16 based wireless mesh network in his master dissertation [8].

References

1. Akyildiz, I.F., Wang, X., Wang, W.: Wireless Mesh Networks: a Survey. Computer Networks 47(4), 445–487 (2004)
2. Olejnik, R.: An Experimental Wireless Mesh Network Node Based on AVR ATmega16 Microcontroller and RFM12B Radio Module. In: Kwiecień, A., Gaj, P., Stera, P. (eds.) CN 2010. CCIS, vol. 79, pp. 96–105. Springer, Heidelberg (2010)
3. Datasheet: RFM12B programming guide. HOPE Microelectronics (2011)
4. Datasheet: Si4421 Universal ISM Band FSK Transceiver. Silicon Labs (2008)
5. Peterson, J.L.: Petri net theory and the modeling of systems. Prentice-Hall (1981)
6. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE 77(4), 541–580 (1989)
7. Rzońca, D., Trybus, B.: Hierarchical Petri Net for the CPDev Virtual Machine with Communications. In: Kwiecień, A., Gaj, P., Stera, P. (eds.) CN 2009. CCIS, vol. 39, pp. 264–271. Springer, Heidelberg (2009)
8. Urbaniak, S.: Communication algorithms and principles for a prototype of a wireless mesh network. Master thesis, West Pomeranian University of Technology, Szczecin (2011)