Applied Mathematics & Information Sciences
An International Journal

Design and Realization of Multi-Thread Structure for an LLRP Server

Yun-Ho Kim[1], Tae-Yeong Lee[1], Yeong Rak Seong[1] and Ha-Ryoung Oh[1]
[1]*Department of Electrical Engineering, Kookmin University, Seoul, Korea*
Email: yeong@kookmin.ac.kr

Received 01 Jul. 2011; Revised 13 Nov. 2011; Accepted 22 Jan. 2012

Abstract: This paper presents multi-thread structure for an LLRP server. To design the structure, (i) the requirements for an LLRP server are deduced; (ii) threads are divided to satisfy the requirements; (iii) the operation procedure of an LLRP system is investigated at the thread level; and (iv) the function of each thread and interaction among threads are specified. Then to validate the designed structure, first the design structure is modeled by using the DEVS formalism which specifies discrete event systems, and then it is validated by simulating the DEVS model. Finally, the structure is realized to a C++ program. To maximally utilize the validation result, the program is built on the basis of the simulation code. The test input sequences used during the validation process are also applied to the realized program. From the test result, we can confirm that the proposed structure satisfies the specified requirements.

Keywords: LLRP, RFID, multi-thread, DEVS formalism, simulation

1 Introduction

An RFID (Radio Frequency Identification) system is composed of tags, readers, and the upper-layer applications. Recently, RFID has been studied in a lot of researches [1-3]. RFID has great potential to be applied to various industrial domains. Nonetheless the technology has not been widely employed yet. One of the reasons is that most RFID readers employ distinct vendor-specific protocols while communicating with upper-layer applications, although they comply the standard protocols (e.g. EPCGlobal class 1 generation 2 [4]) while communicating with tags. To remove the obstacle, EPCGlobal announces a new standard protocol which defines interfaces between upper-layer applications and RFID readers, called LLRP (Low-Level Reader Protocol) [5].

An LLRP system has client-server structure over the TCP/IP network. In the client-server structure, a server provides a set of services, whereas a client requests the services. In this respect, upper-layer applications correspond to clients, whereas RFID readers correspond to servers. Several researches have been conducted on the client side [7-9]. However, implementation of LLRP on the server side has been scarcely reported. In this paper, a software program which controls physical RFID readers according to the LLRP messages sent from upper-layer applications is referred to as an LLRP server.

This paper presents multi-thread structure for an LLRP server. To design the structure, (i) the requirements for an LLRP server are deduced; (ii) threads are divided to satisfy the requirements; (iii) the operation procedure of an LLRP system is investigated at the thread level; and (iv) the function of each thread and interaction among threads are specified.

Then the designed software structure is validated. For validation, the design result should be transformed into an executable form. In the classical approach, first the design result is implemented into a program, and then the design result is validated by executing the program. However it is very hard to validate a multi-thread program due to non-deterministic characteristic of thread scheduling. Especially, if the program interacts with a hardware

device which has a time-critical operation, the validation becomes more difficult. In this paper, to reduce complexity in validation process, first the designed software structure is modeled by using the DEVS formalism [6] which specifies discrete event systems, and then it is validated by simulating the DEVS model. In DEVS, execution of components is strictly controlled by a synchronization mechanism based on virtual time. Thus non-determinism in thread scheduling can be eliminated. Moreover, time-critical behavior of hardware devices can be also modeled by the DEVS formalism. One of the advantages of the DEVS formalism is that DEVS models can be easily simulated by using the DEVS abstract simulator algorithm [6]. In this paper, DEVSim++ [10] which implements the algorithm in C++ is employed for the simulation.

Finally, the validated structure is realized to a C++ program. To maximally utilize the validation result, the program is built on the basis of the DEVSim++ simulation code. The test input sequences used during the validation process are also applied to the realized program. From the test result, we can conclude that the proposed structure satisfies the specified requirements.

2 Design

An LLRP server is designed in this section. To provide the functions defined in the LLRP standard, the following requirements should be considered.

(1) An LLRP server should concurrently process request messages, which are asynchronously generated by LLRP clients, as well as errors and exceptions, which are abruptly occurred while the server operates. Thus, an LLRP server should be able to communicate with LLRP clients without regarding to its state.
(2) An LLRP server should be able to store and manage configuration parameters contained in the messages transferred from LLRP clients.
(3) An LLRP server should be able to process events and actions at the scheduled time.
(4) An LLRP server should be able to provide low-level access to the RFID air interface.
(5) An LLRP server should be able to support various RFID readers. In order to do this, the hardware-specific part of the LLRP server should be localized.
(6) An LLRP server should be able to collect the communication results with tags, manage the state of readers, and report them to LLRP clients.

To satisfy those requirements simultaneously, it is more appropriate that the LLRP sever is designed and realized with multi-thread structure. Now, let's discuss how threads are separated.

To satisfy (1), two threads which manage uplink/downlink communication with upper-layer applications should be separated from the main thread. The communication response time of the LLRP server would be reduced with the separated threads. Moreover, the LLRP server can be easily adapted to the environments by modifying only the related thread when the uplink/downlink communication protocol needs to change in real application environments. These roles are assigned to the *SEND/RECV* threads.

To satisfy (3), a separated thread is required to manage a real-time clock. The thread receives requests for alarm calls from other threads, and sends alarm messages to them at the scheduled time instances. The *TRIGGER* thread has the duty.

To satisfy (4) and (5), threads which manipulate RFID reader devices without regarding to operation of other threads are required. Meanwhile, an LLRP server would have many RFID readers. Moreover, each RFID reader vendor has its own API. Therefore, it would be desirable that each RFID reader is controlled by distinct threads, and the code independent on certain API is separated from others. In the API-independent code, RFID reader operations contained in an LLRP message are decomposed and sent to the threads which manipulate RFID readers. In this paper, an instance of the *uHANDLER* thread is assigned to each RFID reader, and the *HANDLER* thread is devoted to the API-independent codes.

To satisfy (6), the *REPORT* thread is used. It collects and manages results of tag operations and status of physical RFID readers. Also the thread converts the information into LLRP messages, and then reports the messages to related clients.

Finally, to satisfy (2) and to supervise overall system operation, the *MANAGER* thread is used. It decodes and stores LLRP messages sent by clients. Also, when one of the stored LLRP messages is ready to be executed, the thread initiates to execute the message.

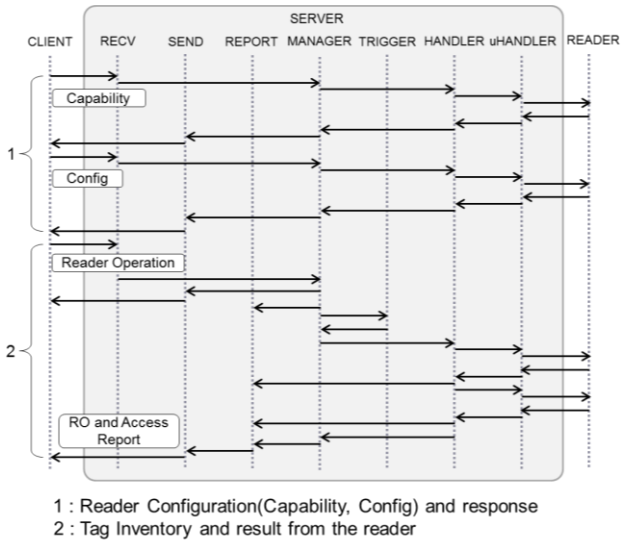

Figure 1: A typical operation scenario of an LLRP system.

With the separated threads, many operation scenarios of an LLRP server are recomposed at the thread level. For example, figure 1 shows the interaction among the separated threads for a typical operation scenario of an LLRP system. By investigating these interaction diagrams, the function of each thread and the control and information flows among threads are analyzed.

Due to the page limit, a simplified version of the *HANDLER* thread is explained as an example in this paper. In the full version, the *HANDLER* thread becomes more complex for supporting more delicate operations. As shown in figure 1, execution of the *HANDLER* thread is initiated by the *MANAGER* thread. When the *HANDLER* thread is idle and an LLRP message which includes RFID reader operations is ready to be executed, the *MANAGER* thread sends the message to the *HANDLER* thread. Then, the *HANDLER* thread decomposes the message into many control commands, and sends them one by one to the *uHANDLER* thread. The *uHANDLER* thread directly interacts with the RFID reader, and reports the communication results between the reader and tags to the *HANDLER* thread. Then, the *HANDLER* thread updates its status and forwards the reported result to the *MANAGER* and *REPORT* threads. When the *HANDLER* thread has no more control commands to be executed or detects unrecoverable errors and exceptions, it informs its status to the *MANAGER* thread, and becomes idle.

Figure 2 illustrates control and information flows among the threads. To increase concurrency, each thread communicates with others by use of asynchronous message passing.

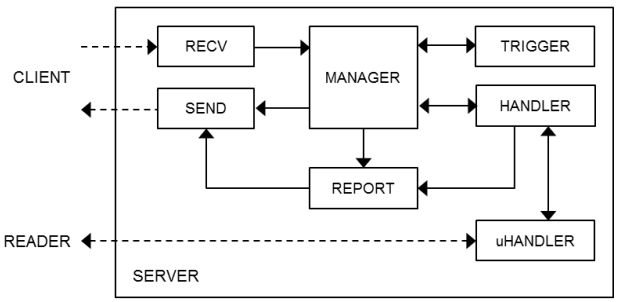

Figure 2: Control and information flows among the threads.

3 Modeling and Simulation

In this section, the proposed multi-thread structure is validated. In this paper, first the structure is modeled and then it is validated by simulating the model. To achieve this, the DEVS formalism [6] is employed. The DEVS formalism specifies discrete event systems in a hierarchical, modular manner. The reasons why the DEVS formalism is employed in the paper are that (i) a multi-thread system can be modeled as a discrete event system; (ii) unlike a general programming language environment, execution of components is strictly controlled by a synchronization mechanism based on virtual simulation time in DEVS; and (iii) DEVS models can be easily simulated by using the DEVS abstract simulator algorithm [10]. Moreover, behavior of the environments which surrounds the designed software structure can be also modeled by using the DEVS formalism.

The software development techniques using models have been attempted in many researches. Model-driven architecture (MDA) [11-12] launched by the Object Management Group (www.omg.org) is a well-known methodology of them. To specify a model, MDA most often uses the Unified Modeling Language (UML) [13-14]. Understandably, the models in those approaches are models that are related to developing software. In contrast, the models in the DEVS formalism are models that are related to specifying general discrete event systems. Thus, the DEVS formalism provides a more rigorous and systematic way for modeling and simulation. Additionally, it has many useful features for modeling the LLRP server as stated above. For those reasons, it is employed in this paper.

There are two types of models in the DEVS formalism: atomic models and coupled models. An atomic DEVS model specifies dynamic behavior of an elementary component, whereas a coupled model specifies how to couple several components together to form a new compound component. An atomic DEVS model has a set of input ports, a set of

output ports, and a set of state variables. When it receives a message from an input port, it executes the external transition function; and it produces output messages by using the output function, just before it executes the internal transition function. The timing of internal transitions is strictly controlled by the time advance function. The detail description of the DEVS formalism can be found in [6].

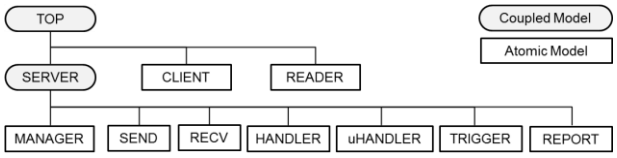

Figure 3: Hierarchical structure of a simple LLRP system model.

Figure 3 shows the hierarchical structure of a simple LLRP system model. Coupled model TOP represents an LLRP system which consists of an LLRP server, an LLRP client, and a population of tags. This paper assumes that the LLRP server has a single RFID reader. Coupled model SERVER corresponds to the proposed LLRP server and is further decomposed to the seven threads. Each of the threads, the LLRP client, and the population of tags are modeled as atomic models.

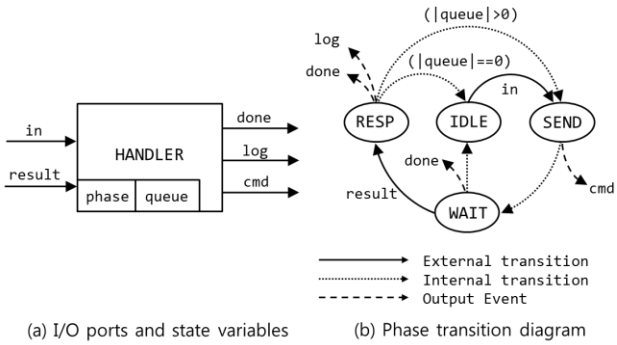

Figure 4: Atomic model HANDLER

The simplified version of the *HANDLER* thread is used again as an example of DEVS modeling. Figure 4(a) shows I/O ports and the state variables of the HANDLER model. The model has two input ports, and three output ports. From a global viewpoint, HANDLER would communicate with the MANAGER, REPORT, and uHANDLER models as shown in figure 2. However, since the DEVS formalism specifies a system in a hierarchical, modular way, HANDLER communicates with others only through its own input/output ports. It receives an LLRP message from the in port, sends control commands to the cmd port, receives tag communication result from the result port, forwards the result to the log port, and reports its status to the done port. Coupling relationships between I/O ports are specified in the parent coupled model SERVER.

Meanwhile, HANDLER has two state variables; phase specifies processing stages of the model, and queue stores control commands. Figure 4(b) illustrates phase transition of the HANDLER model. Initially, the model is in IDLE phase. When it receives an LLRP message from the in port, HANDLER translates the message to a sequence of control commands, stores the commands in queue, and goes in SEND phase. Then, it retrieves a control command from queue, outputs the command to the cmd port and goes in WAIT phase. In WAIT phase, if a status input arrives from the result port within a predetermined time interval, HANDLER goes in RESP phase; otherwise, it transmits an error message to the done port, and goes in IDLE phase. At the end of RESP phase, HANDLER forwards the received status information to the done port and sends a log message to the log port. Also, if queue is empty, HANDLER goes in IDLE phase; otherwise, it goes in SEND phase again to send the next control command to the cmd port.

For simulating the DEVS models, DEVSim++ [10] which implements the DEVS abstract simulator algorithm in C++ is employed. Figure 5 shows the DEVSim++ code of the HANDLER model in figure 4. The external transition function, the internal transition function, the output function, and the time advanced function are implemented as separated C++ functions. The two external transitions in figure 4(b) are implemented in ExtTransFn( ). For example, when HANDLER receives a message from the in port during IDLE phase, it translates the message into control commands, stores the commands in queue by using MakeCmdSet( ), and goes in SEND phase. Three internal transitions are implemented in IntTransFn( ). As mentioned earlier, the output function OutputFn( ) is called just before IntTransFn( ) is called. For example, at the end of SEND phase, HANDLER retrieves a control command from queue by using GetCmd( ), outputs a control command to the cmd port by using SetPortValue( ) in OutputFn( ); and goes in WAIT phase in IntTransFn( ). The time advance function TimeAdvanceFn( ) defines how long HANDLER remains in each phase. For example, in

```
Handler::ExtTransFn(CMessage& X)
{
  port = X.GetPort();
  pMsg = X.GetValue();
  if (GetPhase() == IDLE
    && port == "in") {
      MakeCmdSet(m_pMsg);
      SetPhase(SEND);
  }
  else if (GetPhase() == WAIT
    && port == "result") {
      SaveCommStat(m_pMsg);
      SetPhase(RESP);
  }
}

Handler::IntTransFn()
{
  switch (GetPhase()) {
  case SEND:
    SetPhase(WAIT);
  case WAIT:
    SetPhase(IDLE);
  case RESP:
    if (GetCmdCount() > 0)
      SetPhase(SEND);
    else
      SetPhase(IDLE);
  }
}
```

```
Handler::OutputFn(CMessage& Y)
{
  switch (GetPhase()) {
  case SEND:
    pMsg = GetCmd();
    Y.SetPortValue("cmd", pMsg);
  case WAIT:
    pMsg->type = NO_RESPONSE;
    Y.SetPortValue("done", pMsg);
  case RESP:
    pMsg = GetCommLog();
    Y.SetPortValue("log", pMsg);
    pMsg = GetCommStat();
    Y.SetPortValue("done", pMsg);
  }
}

TimeType Handler::TimeAdvanceFn()
{
  switch (GetPhase()) {
  case IDLE: return Infinity;
  case SEND: return SEND_TIME;
  case WAIT: return TIMEOUT;
  case RESP: return RESPONSE_TIME;
  }
}
```

Figure 5: DEVSim++ code of the HANDLER model.

IDLE phase, HANDLER has no control commands to be executed. Then, it should wait arrival of a new input. Thus, TimeAdvanceFn( ) returns Infinity (∞).

With the same way, other threads are also modeled and implemented. Many cases, including the typical operation scenario in figure 1, are tested on the implemented DEVSim++ code. The simulation result shows that the proposed LLRP server structure produces expected behavior for each test case.

4 Realization

In this section, the proposed LLRP server is realized to a C++ program. To maximally utilize the validation result, the program is built on the basis of the DEVSim++ simulation code.

In this paper, nine atomic models and two coupled models are developed and validated. Each of the seven atomic models representing threads must be realized as an actual thread. The CLIENT and READER models represent a software program or a hardware device. For testing the realized LLRP server, they are briefly realized. Coupled model TOP representing the overall system is used to set up the test environment. Coupled model SERVER integrates the seven threads into a single program and specifies communication paths among threads. The specification of SERVER is utilized while the seven threads are realized.

Figure 6 shows the simplified code of the *HANDLER* thread. Four characteristic functions of the DEVSim++ code in figure 5 are integrated together in a thread. The thread executes an infinite loop. To execute external and internal transitions in a unified way, two classes of messages are used. The *INPUT* class corresponds to external transition, whereas the *TIME_EXPIRED* class corresponds to internal transition. In figure 6, the code which matches with ExtTransFn( ) in figure 5 can be easily found. However, IntTransFn( ) and OutputFn( ) may not be easily matched with figure 6, since the two functions separated in the DEVS formalism is merged in the code. The second switch statement corresponds to the code. For each case statement, IntTransFn( ) and OutputFn( ) are merged. SendMsg( ) in figure 6 corresponds to SetPortValue( ) in figure 5. The target thread of each output port is originally specified in the parent coupled model SERVER. The code which corresponds to TimeAdvanceFn( ) is shown in the end of figure 6.

The other threads are also realized with a similar way, and linked to a single program. The program is verified with the test cases used during the validation process. From the test results, we can confirm that the proposed structure satisfies the specified requirements.

```
Handler thread()
{
  while (true) {
    msg = RecvWait(RecvQ);
    switch (msg.class) {
    case INPUT:
      if (phase == IDLE &&
        msg.GetPort() == "in"
          MakeCmdSet(msg.GetValue());
          phase = SEND;
      }
      else if (phase == WAIT &&
        msg.GetPort() == "result") {
          SaveCommStat(msg.GetValue());
          phase = RESP;
      }

    case TIME_EXPIRED:
      switch (phase) {
      case SEND:
        SendMsg("cmd", GetCmd());
        phase = WAIT;
      case WAIT:
        pMsg = ErrorMsg(NO_RESPONSE);
        SendMsg("done", pMsg);
        phase = IDLE;
      case RESP:
        pMsg = GetCommLog();
        SendMsg("log", pMsg);
        pMsg = GetCommStat();
        SendMsg("done", pMsg);
        if (GetCmdCount())
          phase = SEND;
        else
          phase = IDLE;
      }
    }

    switch (phase) {
    case IDLE: t = INFINITY;
    case SEND: t = SEND_TIME;
    case RESP: t = RESP_TIME;
    case WAIT: t = TIMEOUT;
    }
    if (t != INFINITY)
      SetTimer(t, handlerQid);
  }
}
```

Figure 6: C++ code of the *HANDLER* thread.

5 Conclusion

The key novelty of this paper is that it proposes novel multi-thread structure for an LLRP server. The server is decomposed into seven threads to support the requirements of the target system simultaneously. The function of each thread and interactions among threads are specified by investigating various operation scenarios of the system. The second key contribution is that the designed multi-thread structure is modeled as a discrete event system by using the DEVS formalism, and simulated by using DEVSim++ to reduce complexity in the validation process,. The simulation result shows that the proposed structure produces expected behavior for each simulated case. The last is that the structure is realized to a program. From the testing result of the program, we can confirm that the proposed LLRP server properly operates with fast responsiveness. The proposed structure can be employed in many RFID systems. Meanwhile, the validation method used in the paper can be a very promising solution for developing other multi-thread applications.

Acknowledgements

This work was supported by research program 2012 of Kookmin University in Korea.

References

[1] C. Floerkemeier and S. Sarma, An Overview of RFID System Interfaces and Reader Protocols. 2008 IEEE International Conference on RFID. (2008), 232-240.

[2] S.Y. Choi, H.M. Jung, K.S. Bang, W.Y. Lee and Y.W. Ko, Real-time Data Stream Management System for Large Volume of RFID Events. 2008 International Conference on Convergence and Hybrid Information Technology. (2008), 515-521.

[3] S.W. Ahn, W.S. Ryu, B.H. Hong, H.S. Chae and J.H. Lee, Dynamic Thread Management for Scalable RFID Middleware. 2010 International Conference on Information Science and Applications. (2010), 1-8.

[4] EPCGlobal Inc., UHF Class 1 Gen 2 Standard, Version 1.2.0, (2008), http://www.gs1.org/gsmp/kc/epcglobal/uhfc1g2/uhfc1g2_1_2_0-standard-20080511.pdf

[5] EPCGlobal Inc., Low Level Reader Protocol (LLRP), Version 1.1, (2010), http://www.gs1.org/gsmp/kc/epcglobal/llrp/llrp_1_1-standard-20101013.pdf

[6] B.P. Zeigler, Object-Oriented Simulation with Hierarchical, Modular Models, Academic Press, (1990)

[7] S.S. Kang, G.J. Park, Design and Implementation of ALE v1.1 Middleware in RFID systems. International Conference on New Trends in Information and Service Science. (2009), 815-821.

[8] Q. Wang, W. Ryu, S. Kim, and B. Hong, Demonstration of an RFID Middleware: LIT ALE Manager. 18th ACM Conference on Information and Knowledge Management.

(2011), 2071-2072.

[9] M. Buettner, R. Prasad, M. Philipose and D. Wetherall, Recognizing daily activities with RFID-based sensors. 11th international conference on Ubiquitous computing. (2009), 51-60.

[10] T.G. Kim, DEVSim++ User's Manua: C++ Based Simulation with Hierarchical Modular DEVS Models. (1994)

[11] Object Management Group, MDA Guide, version 1.0.1, (2003)

[12] A. Kleppe, J. Warmer, and W. Bast, MDA Explained: The Model Driven Architecture– Practice and Promise, Addison-Wesley Professional. (2003)

[13] Object Management Group, OMG Unified Modeling Language(OMG UML) Superstructure, version 2.4, (2011)

[14] R. Miles and K. Hamilton, Learning UML 2.0, O'Reilly Media, Inc., (2006)

Yun-Ho Kim received the M.S. degree in Department of Electrical Engineering from Kookmin University, He is currently an Student in Department of Electrical Engineering, Kookmin University. His research interests are in the areas of RFID, real-time processing, discrete event system modeling and simulation.

Tae-Yeong Lee received the M.S. degree in Department of Electrical Engineering from Kookmin University. His research interests are in the areas of discrete event system modeling and simulation.

Yeong Rak Seong received the B.S. degree in electronics engineering from Hanyang University, Seoul, Korea, in 1989 and the M.S. and Ph.D. degrees in electrical engineering from Korea Advanced Institute of Science and Technology, Daejeon, Korea, in 1991 and 1995, respectively. Since 1996, he has been with Kookmin University, Seoul, where he is currently a professor. His current research interests include real-time systems, wireless sensor networks, and discrete event system modeling and simulation.

Ha-Ryoung Oh was born in Busan, Korea, in 1961. He received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 1983 and the M.S. and Ph.D. degrees in electrical engineering from Korea Advanced Institute of Science and Technology, Daejeon, Korea, in 1988 and 1992, respectively. Since 1992, he has been a professor with Kookmin University, Seoul. His current research interests include RFID system, wireless sensor network, and embedded system.