Exploring the Design Space for Network Protocol Stacks on Special–Purpose Embedded Systems

Hyun-Wook Jin and Junbeom Yoo

Department of Computer Science and Engineering
Konkuk University
Seoul 143-701, Korea
{jinh,jbyoo}@konkuk.ac.kr

Abstract. Many special-purpose embedded systems such as automobiles and aircrafts consist of multiple embedded controllers connected through embedded network interconnects. Such network interconnects have particular characteristics and thus have different communication requirements. Accordingly, we need to frequently implement new protocol stacks for embedded systems. Implementing new protocol stacks on embedded systems has significant design space but it has not been explored in detail. In this paper, we aim to explore the design space of network protocol stacks for special-purpose embedded systems. We survey several design choices very carefully so that we can choose the best design for a given network with respect to performance, portability, complexity, and flexibility. More precisely we discuss design alternatives for implementing new network protocol stacks over embedded operating systems, methodologies for verifying the network protocols, and the designs for network gateway. Moreover, we perform case studies for the design alternatives and methodologies discussed in this paper.

Keywords: Embedded Networks, Embedded Operating Systems, Network Protocol Stacks, Formal Verification, Protocol Verification, Network Gateway.

1 Introduction

Many special-purpose embedded systems consist of multiple embedded controllers connected through network interconnects. For example, machine control systems such as automobiles and aircrafts are equipped with more than hundreds embedded controllers or boards, which collaborate by communicating each other. Since such embedded systems use special network interconnects and have different communication requirements, there are many cases where new protocol stacks are needed to be implemented. Implementing new protocol stacks on embedded systems has significant design space but it has not been explored in detail. Thus it is highly desirable to analyze the possible design alternatives and present their case studies as references.

In this paper, we aim to explore the design space of network protocol stacks for special-purpose embedded systems. The legacy protocol stacks such as TCP/IP have several implementations already exist, which can help significantly to reduce the time frame of design and implementation phases. Adding new protocol stacks, however, requires significant cost in terms of time and complexity from the industry perspective.

Therefore, we need to consider several design choices very carefully so that we can choose the best design for a given network with respect to performance, portability, complexity, and flexibility. In this paper, we present various design alternatives and compare them in several aspects. Moreover, we perform the case studies for the design alternatives.

The rest of the paper is organized as follows: Section 2 discusses the design alternatives for implementing new network protocol stacks over embedded operating systems. Section 3 describes the methodologies for verifying the network protocols. Section 4 addresses the network interoperability issue and discusses the designs for network gateway. Finally we conclude the paper in Section 5.

2 Protocol Stacks on Embedded Nodes

In this section, we explore the design and implementation alternatives of network protocol stacks on embedded nodes. The designs can highly depend on operating systems and their task models but we try to generalize this discussion as much as possible so that the designs described can be applied to the most of embedded operating systems. One of the most important issues when implement new network protocol stacks is who takes charge of multiplexing and demultiplexing of network packets. Accordingly, we classify the possible designs into two: i) user-level design and ii) kernel-level design.

2.1 User-Level Design

In this design alternative, the protocol stacks are implemented as a user-level thread or process, which performs (de)multiplexing across networking tasks. The user-level protocol stacks can be portable across several embedded operating systems as far as they follow the standard interfaces such as POSIX. The overall designs are shown in Figure 1. As we have mentioned, the way to implement new network protocol stacks are dependent on the task models of operating systems. Many embedded operating systems such as VxWorks [18] and uC/OS-II [19] define the thread-based tasks on top of the flat memory models in which the user-level protocol stacks are implemented as a user thread. On the other hand, some other embedded operating systems such as Embedded Linux and QNX [20] define isolated memory spaces between tasks. In such systems, the user-level protocol stacks are implemented as a user process in general. Though most of these process-based task models also support multiple threads, the design of process-based protocol stacks is still attractive. This is because, in this task model, if we implement the protocol stacks as a thread it can only support the threads belong to the same process. That is, the thread-based protocol stacks over the process-based task models is not suitable to support multiple networking processes.

In either thread or process-based design, the protocol stacks send the network packets by accessing the network device driver directly. Thus the device drivers have to provide the interfaces (e. g., APIs or system calls) for the network protocol stacks. The user-level tasks request the protocol stacks to send their packets through Inter-Process Communication (IPC). In case of thread-based design, since the protocol stacks share the memory space with other tasks, the network data can be directly accessed from the protocol thread without data copy as far as the synchronization is guaranteed by an IPC such as semaphore. On the other hand, the process-based protocol stacks need to pass the network data between the networking tasks and the protocol stacks explicitly by using an IPC such as message queue. This can add message passing overhead because the messaging IPCs usually require memory copy operations to move data between two different memory spaces.

On the receiver side, how it works is similar with the sender side; however, there is a critical design issue of how to detect the incoming new packets. Since the protocol stacks are implemented at the user-level, there is no proper asynchronous signaling mechanism at the device driver to notify new packet arrival to the user-level protocol stacks. Thus, the interfaces provided by the device driver are the only way to check new packet arrival. However, if the interface has a blocking semantic then the protocol stacks cannot handle other requests (e.g., sending requests) from the tasks while waiting a new packet arrived. There are two solutions to overcome this issue. One is to use asynchronous interface and the other one is to have multithreaded protocol stacks. The asynchronous interface is easy to use but it is hard to come up with an optimal strategy of calling the interface in terms of when and how frequently. Thus it is likely to achieve lower performance than what the actual network can provide or waste the processor resources. Instead, the multithreaded protocol stacks can have separate threads to serve the sending and receiving operations respectively. That is, for both thread- and process-based designs, the protocol stacks consist of a set of threads. Only one difference is that the multiple threads belong to the same process in case of the process-based design. The receiving thread can block on waiting a new packet while the sending thread handles the requests from the tasks. Once the new packet has been recognized by returning from the blocked function, the receiving thread of the protocol stacks interpret the header and pass the packet to the corresponding process through an IPC.

Since the protocol stacks are implemented at the user-level, they are scheduled as other user-level tasks by the task scheduler. If we give the same priority to the protocol stacks with other tasks, the execution of the protocol stacks can get delayed, which results in high network latency. Thus it is desired that the protocol stacks have higher priority than general user-level tasks and block on waiting new packets received or sending requests, which allows other tasks to utilize the processor resources if there are no pending jobs of the protocol stacks.

As a case study we have implemented Network Service of Media Oriented System Transport (MOST) [1] at the user-level over Linux [2]. MOST is an automotive high-speed network to support multimedia data streaming. The current MOST standard specifies 25Mbps ~ 150Mbps network bandwidth with QoS support. To meet the demands from various automotive applications, MOST provides three different message channels: control, stream, and packet message channels. Network Service is the transport protocol for the control messages, which covers from layer 3 to parts of layer 7 of OSI 7 layers. In order to implement Network Service, we have applied the process-based design where the protocol stacks consist of sending and receiving threads. We have utilized the ioctl() system call to provide interfaces between the protocol stacks and the device driver. We have also implemented library for applications, which provides interfaces to interact with the protocol stacks using POSIX message queue. The performance results show 0.9ms of one-way latency with 8-byte control message.

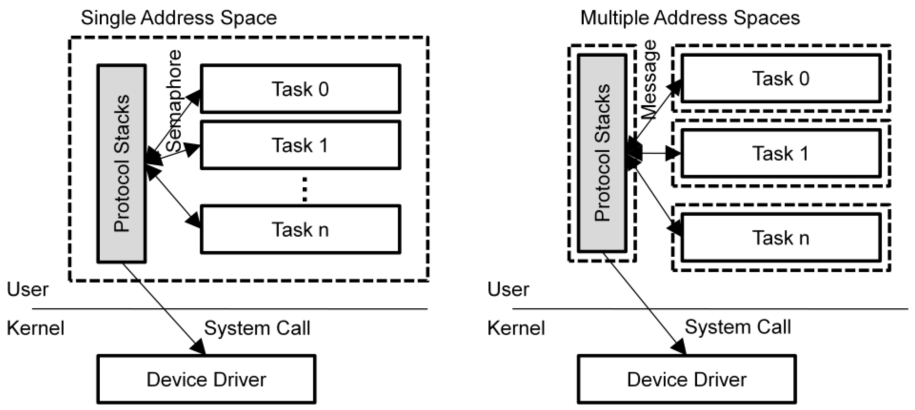

Fig. 1. User-level protocol stacks: (a) thread-based design and (b) process-based design

2.2 Kernel-Level Design

In this design alternative, the protocol stacks are implemented as a part of operating system. Thus we do not need to move data between the device driver and the protocol stacks. This is because both use the kernel memory space and can share the network buffer. In addition, since the kernel context has higher priority than the user context, the kernel-level protocol stacks can guarantee the network performance. Accordingly, it has more potential of achieving better performance than the user-level protocol stacks. This design however may require modifications of the kernel, which is not portable across several operating systems. As shown in Figure 2, we classify the kernel-level design into bottom half based design and device driver based design according to where the protocol stacks are implemented (especially for receiver side). The traditional protocol stacks are implemented as a bottom half in general. In such design, when a packet has been received from the network controller, the interrupt handler simply queues it to a queue shared with the bottom half. Then the bottom half takes care of most of protocol processing including demultiplexing. The bottom half is scheduled by the interrupt handler when there is no interrupts to be processed. On the other hand, in the device driver based design, the entire protocol stacks are implemented in the device driver. Therefore, if the protocol stacks are heavy like TCP/IP then the device driver based design may not be suitable.

In case of the kernel-level design, the user tasks request a sending operation through a system call. The system call eventually passes the request to the device driver. On the sender side, the main difference between two design alternatives is that, in case of the bottom half based design, the kernel performs most of protocol processing before passing down the user request to the device diver. It is to be noted that the data copy operation between the user and kernel spaces should be carefully designed. In either synchronous or asynchronous interface, we can copy the user data into the kernel and return immediately; however, this results in the copy overhead. On the contrary, we can avoid the copy operation by delaying the notification of completion but this can hinder the application's progress.

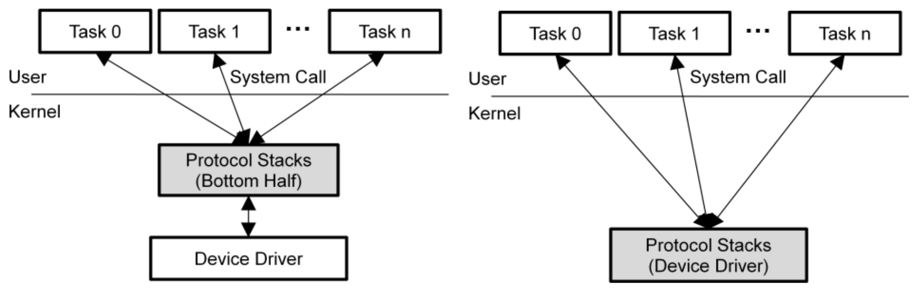

Fig. 2. Kernel-level protocol stacks: (a) bottom half based design and (b) device driver based design

On the receiver side, once a new packet comes in from the network controller, the interrupt handler does urgent process before passing it to the upper layer. In the bottom half based design, as we have mentioned earlier, the bottom half takes care of interpreting the header and demultiplexing. Some of operating systems such as Embedded Linux provide an interface to insert a new bottom half (more precisely tasklet in Embedded Linux) without kernel modification. The microkernel based operating systems such as QNX also allow adding new protocol stacks in a similar manner. In the device driver based design, the bottom half is not taken into account at all. In this design alternative, the protocol stacks are implemented in the system call and the interrupt handler. The distribution of weight between the system call and the interrupt handler can vary in terms of which does more protocol processing but usually the interrupt handler does majority of the protocol processing. This is because doing demultiplexing at the interrupt handler is more efficient. Otherwise, the system call needs to search the incoming packet queue internally, which requires exhaustive searching time and locking overhead between tasks. However, doing more work at the interrupt handler is not desirable because it is supposed to finish its work very quickly. Therefore, this design is valuable when the overhead for protocol processing is low.

As a case study of kernel-level design, we have implemented a device driver based protocol called RTDiP (Real-Time Direct Protocol) in the Embedded Linux over Ethernet [3, 4]. RTDiP is a new transport protocol that can provide priority aware communication, communication semantics for synchronization, and low communication overhead. In the synchronous semantics, the communication protocols do not queue the packets but keep only the last packet received, which is suitable for distributed synchronization over relatively small area embedded networks. The performance results show that RTDiP reports 48us one-way latency with 8-byte message and provides better overhead prediction. We are currently implementing RTDiP over Control Area Network (CAN) as well. In addition we plan to implement it in another embedded operating system such as QNX.

3 Verification Methodologies

Protocol verification [5] is an activity to assure the correctness of network communication protocols. The design alternatives we have studied in Section 2 should be verified thoroughly before proceeding to the implementation. The formal verification has been known as the prominent but cost-ineffective technique. This section introduces formal verification techniques for verifying network protocol stacks. We briefly overview formal verification techniques and then review the techniques from aspect of network protocol stacks verification. We then share our experience of verifying protocol stacks of system air conditioning system.

3.1 Formal Verification

Formal verification and formal specification altogether are called as formal methods [6]. Formal specification [7] is a technique for specifying the system on the basis of mathematics and logic. It has various techniques and notations, e.g. algebra, logic, table, graphics and automata. After completing the formal specification, we can apply formal verification techniques to the specification to prove that the system satisfies required properties. There are two main approaches in formal verification: deductive reasoning and algorithmic verification.

Deductive reasoning is a verification methodology using axioms and proof rules to establish the reasoning. Experts construct the proofs in hands, and it usually requires greater expertise in mathematics and logic. Even if tools called theorem prover have been developed to provide a certain degree of automation, its inherent characteristic makes it difficult to be used widely for verifying recent network protocol stacks. Second methodology is algorithmic verification, usually called model checking [8]. Model checking is a technique verifying finite state systems through exhaustively searching all states space to check whether specified correctness condition is satisfied. It is carried out automatically without almost any intervention of experts, but restricted to the verification of finite state systems. The deductive reasoning, on the other hand, has no such limitations.

With respect to protocol verification, the latter - model checking is more efficient and cost-effective than the former - theorem proving. The former's main drawback, requiring considerable expertise, makes the model checking techniques better suited for protocol stacks verification. Indeed, as the performance of model checking technique has increased rapidly, it can do various verifications more efficiently than when it had been firstly proposed.

3.2 Formal Verification Techniques for Network Protocol Stacks

The formal verification techniques for network protocol stacks fall into several categories. General-purpose model checkers such as Cadence SMV [9] and SPIN [10] can verify protocols efficiently. General-purpose proof tools which are not the model checker but conduct formal verification such as UPPAAL [11] are useful too. We can also use specialized protocol analysis tools (e.g., Meadows' NRL [12] and Cohen's TAPS [13])

Formal specification should be prepared before conducting formal verification. Finite State Machine (FSM) based formal specification technique has been widely used for specifying network protocols and stacks. FSM mainly consists of a set of transition rules. In the traditional FSM model, the environment of the FSM consists of two finite and disjoint sets of signals, input signals and output signals. A number of papers using FSM based formal specification have been reported. Especially, network protocols can be well specified using communicating FSM or extended FSM as reported in [14, 15].

With respect to the formal verification of network protocol stacks, we have to consider two tasks: specification of protocol stacks and modeling of the system implementing the protocol. In the first step, we have to model the protocol algorithm and stack hierarchy using a formal specification method. Then the modeling of the whole embedded network system which includes the implementations of the protocol stacks can proceed. Therefore, verifying the network protocol stacks requires not only the formal specification for the protocol stacks but also the encompassing environment where the protocol stacks are implemented and used.

Formal verification for network protocol stacks totally depends on the formal specification developed beforehand. If we use FSM-based formal specifications (e.g., Statecharts [16] and Specification and Description Language (SDL) [17]), most general-purpose model checkers are available. In case that exact timing constraints should be preserved, timed automata based formal specification like UPPAAL is a good choice. We can also use specialized protocol verification tools, but it is not easy to model the whole system with them. Therefore, the combination of FSM based formal specification and general-purpose model checking tools will be more effective than others.

3.3 SDL-Based Verification of Protocol Stacks

SDL is a formal specification language and tool suite widely used to model the system which consists of a number of independently communicating subsystems. The SDL specification can be translated into FSM forms, and then used as an input for general-purpose model checkers such as SMV and SPIN. Figure 3 describes the architecture of system air conditioning system. We performed the formal verification of the

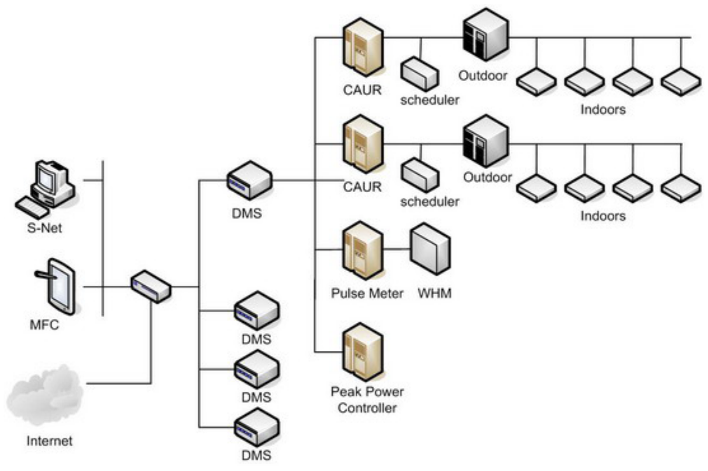

Fig. 3. The architecture of system air conditioning system

network protocol between distributed controllers called DMS (Distributed Management System) and a personal controller called MFC (Multi-Function Controller). A DMS controls all indoor air conditioners, outdoor compressors and network routers under its control. An MFC is a touch-screen based personal controller like PDA.

In our experience, special-purpose embedded network system such as the above can be well specified with SDL and verified formally through general-purpose model checkers such as SPIN. We implemented an automatic translator from SDL into PROMELA, SPIN's input program language, and conducted SPIN model checking. We verified several properties, categorized as feasibility test, responsiveness, environmental assumptions and consistency checking. In addition to the SPIN model checking, the SDL tool has its own validation tool, which checking syntax errors and completeness of the specification.

4 Network Interoperability

Since various network interconnects can be utilized on a distributed embedded system, the network interoperability is a critical requirement in such system. For example, in modern automobile systems, several network interconnects such as CAN, LIN, FlexRay, and MOST are widely used in an integrated manner. In such systems, we need a gateway for interoperation between different networks [2, 21, 22], which is similar with bridges or routers on Internet. Thus the gateway needs to understand several network protocols and convert one into another. In this section, we explore the design alternatives for embedded network gateways. Especially, we classify the gateway designs into two based on how to operate the operating system on the gateway.

4.1 Single OS Based Gateway

In this design alternative, the gateway architecture has a single or multiple homogeneous Micro-Controller Units (MCUs) that run single operating system's image. The MCU can include the network controllers for different network interconnects supported by the gateway or can be connected with the network controllers on the same board through buses such as Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I^2C), etc.

The protocol stacks can be designed and implemented as any of the design choices described in Section 2 but a layer of protocol stacks is required to perform the gateway functions. If the network layer performs the gateway functions, it can be transparent to the networking processes running on the embedded nodes. The network protocols for embedded systems, however, usually have no strict distinction between the network and transport layers because their network layers do not suppose to allow arbitrary transport layers while the Internet Protocol (IP) layer does. In addition, even if the gateway performs the protocol conversion at the network layer, in many cases it is hard to conserve the end-to-end communication semantics due to significant differences between transport protocols of embedded networks.

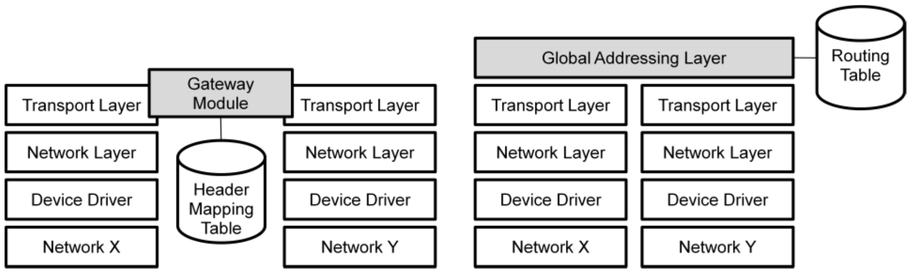

Fig. 4. Single OS based gateway: (a) transport layer based design and (b) global addressing layer based design

Another solution is to introduce a gateway module at the transport layer as shown in Figure 4(a). In this design alternative, the gateway should manage the protocol conversion tables that map between message headers of both network and transport layers for different networks. Since the transport layer translates the protocols internally the legacy applications do not need any modifications. A drawback of this design is the limitation of scalability. The number of possible header patterns can be numerous in some embedded systems, which can result in memory shortage on the gateway node. Therefore, this design is useful only when the number of entries of the protocol conversion tables is predictable. Fortunately, in many embedded systems, we can figure out the number of embedded devices that need to collaborate (i.e., communicate) each other across different networks at the design phase.

We can also add a new layer on top of the transport layer as shown in Figure 4(b). The new layer defines global addressing and APIs to access the layer from the applications. If the gateway uses global addressing the networking processes on every embedded node have to be aware about it. Thus the applications need to be modified but if once this is done they can run transparently on any networks in which the additional layer is inserted. In this case, the gateway only requires managing the routing table and thus the scalability in terms of memory space can be better than the previous design. However, if the most of embedded nodes perform intra-network communication, the overhead of additional layering can harm the performance. Therefore, the decision among the design alternatives described can vary based on the system requirements and characteristics.

As a case study of single OS gateway, we have implemented a gateway between MOST and CAN networks based on the transport layer based design [2]. In this case study, we utilize the MOST Network Service implemented in Section 2.1. The communication semantics of MOST control message are very different with the traditional send/receive semantics. The MOST control message invokes a function on a MOST device. However, CAN does not provide such communication semantics while providing multicast like communication semantics which is not in MOST Network Service. Thus, simple message forwarding with address translation at the network layer does not work. To provide transparent conversion of communication semantics we have suggested a gateway module. In addition, we have implemented the protocol conversion table and defined some entries for performance measurement. The performance results show that the suggested design hardly adds additional overhead, which is about 15% of pure communication overhead, and can deliver control messages very efficiently.

4.2 Multi-OS Based Gateway

Since the embedded nodes on different networks can have different requirements the desirable operating systems can vary. For example, the automobile gateway node can have many kinds of peripheral interfaces such as USB and wireless network for supporting infotainment applications over MOST. Therefore, an operating system that has fluent device drivers such as Embedded Linux is highly expected. On the other hand, the electronic units such as chassis, powertrain and body controllers connected to CAN or LIN demand to guarantee the real-time requirements and thus an RTOS is desirable. Since the gateway needs to meet such various requirements we can consider having multiple operating systems on the gateway node. The address translation issue discussed in Section 4.1 is still applied in the similar manner even in this design alternative. However, an efficient scheme for communication between operating systems has to be taken into account.

A gateway node can be equipped with multiple heterogeneous MCUs that have different network controllers as shown in Figure 5(a). Each MCU can run its own operating system that satisfies the requirements of responsible networks. The MCUs on the gateway node can collaborate by communicating each other through a bus or shared memory module. Since an MCU may do not have all network controllers required to a specific embedded system, we can need several MCUs, which makes the connection architecture between MCUs very complicate. Thus the architecture based on multiple MCUs can be applied to limited cases.

Another approach is to exploit the virtualization technology, which allows running several operating systems on the same MCU as shown in Figure 5(b). The virtualization technology can isolate the system fault from propagating to other operating systems and provide service assurance. In addition, the state-of-the-art virtualization technologies enable low overhead virtualization and better resource scheduling, which lead to high scalability. In addition to the existing optimization technologies, a lighter I/O virtualization can be suggested because the network controllers on the gateway

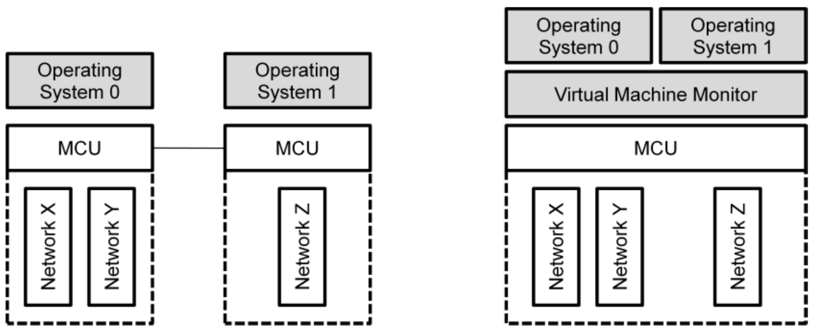

Fig. 5. Multi-OS based gateway: (a) multiple MCUs based design and (b) system virtualization based design

node may not be shared between operating systems. An important issue is how efficiently the operating system domains can communicate each other. In general, the portion of inter-domain communication on a virtualized node is not dominant compared with inter-node communication. However, on the gateway node, many of network messages cause inter-domain communication because they are supposed to be forwarded to another network interface of which another operating system domain may take care.

As a case study of the gateway with multiple operating systems, we are implementing a MOST-CAN gateway using virtualization technology provided by Adeos [23]. Adeos provides a flexible environment for sharing hardware resources among multiple operating systems by forwarding hardware events to appropriate operating system domain. We run Linux and Xenomai [24], a parasitic operating system to Linux, over Adeos. The Linux operating system takes charge of the MOST interface while Xenomai does the CAN interface. The gateway processes are running on each operating system and communicate each other through inter-domain communication interface provided by Xenomai. Since the protocol stacks for MOST and CAN run on different operating systems we perform the protocol conversion at the above of the transport layer but we do not use global addressing. Instead we define a protocol conversion table that maps network connections over different networks.

5 Conclusions

In this paper, we have explored the design space of network protocol stacks for special-purpose embedded systems. We have surveyed several design choices very carefully so that we can choose the best design for a given network with respect to performance, portability, complexity, and flexibility. More precisely we have discussed design alternatives for implementing new network protocol stacks over embedded operating systems, methodologies for verifying the network protocols, and the designs for network gateway. Moreover, we have performed case studies for the design alternatives and methodologies.

Acknowledgments. This work was partly supported by grants #NIPA-2009-C1090-0902-0026 and #NIPA-2009-C1090-0903-0004 by the MKE (Ministry of Knowledge and Economy) under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) and grant #R33-2008-000-10068-0 by MEST (Ministry of Education, Science and Technology) under the WCU (World Class University) support program.

References

1. MOST Cooperation.: MOST Specification. Rev 3.0 (2008)
2. Lee, M.-Y., Chung, S.-M., Jin, H.-W.: Automotive Network Gateway to Control Electronic Units through MOST Network (2009) (under review)
3. Lee, S.-H., Jin, H.-W.: Real-Time Communication Support for Embedded Linux over Ethernet. In: International Conference on Embedded Systems and Applications (ESA 2008), pp. 239–245 (2008)

4. Lee, S.-H., Jin, H.-W.: Communication Primitives for Real-Time Distributed Synchronization over Small Area Networks. In: IEEE International Symposium on Object/component/service-oriented Real-Time distributed Computing (ISORC 2009), pp. 206–210 (2009)
5. Palmer, J.W., Sabnani, K.: A Survey of Protocol Verification Techniques. In: Military Communications Conference - Communications-Computers, pp. 1.5.1–1.5.5 (1986)
6. Peled, D.: Software Reliability Methods. Springer, Heidelberg (2001)
7. Wing, J.M.: A specifier's introduction to formal methods. IEEE Computer 23(9) (1990)
8. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
9. SMV, `http://w2.cadence.com/webforms/cbl_software/index.aspx`
10. SPIN, `http://spinroot.com/spin/whatispin.html`
11. UPPAAL, `http://www.uppaal.com/`
12. Meadows, C.: Analysis of the Internet Key Exchange protocol using the NRL Protocol Analyzer. In: SSP 1999, pp. 216–231 (1999)
13. Cohen, E.: TAPS: A first-order verifier for cryptographic protocols. In: 13th IEEE Comp. Sec. Found. Workshop, pp. 144–158 (2000)
14. Aggarwal, S., Kurshan, R.P., Sabnani, K.: A Calculus for Protocol Specification and Verification. In: Int. Workshop on Protocol Specification, Testing and Verification (1983)
15. Sabnani, K., Wolper, P., Lapone, A.: An Algorithmic Procedure for Protocol Verification. In: Globecom (1985)
16. Harel, D.: Statecharts: A Visual Formalism for complex systems. Science of Computer Programming 8, 231–274 (1987)
17. SDL, `http://www.telelogic.com/products/sdl/index.cfm`
18. Wind River, `http://windriver.com`
19. Labrosse, J.: MicroC/OS-II: The Real-Time Kernel. CMP Books (1998)
20. QNX Software Systems, `http://www.qnx.com`
21. Hergenhan, A., Heiser, G.: Operating Systems Technology for Converged ECUs. In: 7th Embedded Security in Cars Conference (2008)
22. Obermaisser, R.: Formal Specification of Gateways in Integrated Architectures. In: Brinkschulte, U., Givargis, T., Russo, S. (eds.) SEUS 2008. LNCS, vol. 5287, pp. 34–45. Springer, Heidelberg (2008)
23. Yaghmour, K.: Adaptive Domain Environment for Operating Systems (2001), `http://www.opersys.com/adeos`
24. Xenomai, `http://www.xenomai.org`