Journal of the Korea Industrial Information Systems Research Vol. 19, No. 2, Apr. 2014
http://dx.doi.org/10.9723/jksiis.2014.19.2.051

Formal Models of Module Linking Mechanisms for a Single Address Space†

Hiecheol Kim1) and Won-Kee Hong2)

Abstract As WSNs(Wireless Sensor Networks) are being deployed widely in diverse application areas, their management and maintenance become more important. Recent sensor node software takes modular software architectures in pursuit of flexible software management and energy efficient reprogramming. To realize an flexible and efficient modular architecture particularly on resource constrained mote-class sensor nodes that are implemented with MCUs(Micro-Controller Units) of a single address space. an appropriate module linking model is essential to resolve and bind the inter-module global symbols. This paper identifies a design space of module linking model and respectively their implementation frameworks. We then establish a taxonomy for module linking models by exploring the design space of module linking models. Finally, we suggest an implementation framework respectively for each module linking model in the taxonomy. We expect that this work lays the foundations for systematic innovation toward more flexible and efficient modular software architectures for WSNs.

Key Words : Wireless sensor network, Module linking, Modular architecture

1. Introduction

WSNs are being used more widely in diverse areas which include environmental monitoring to capture periodically the physical or chemical volume of interest, military operations to make unmanned surveillance, and disaster prognosis to observe the critical activity of disaster sources such as volcanic activity[1,2]. The widespread of WSNs raises the management and maintenance of sensor node software as one of critical issues. As application logic becomes more complex, we need some modularity to decouple software elements according to their functionality. As in other computing areas, needs for maintenance would occur throughout the life of a deployed WAN. Software update may have to depend on OTA(Over-The-Air) code dissemination as WSNs are often deployed in an unreachable area. This requires energy-efficient software update especially on resource-constrained sensor nodes[5].

A modular software architecture provides a promising solution to efficient software management and maintenance[3,4,5,6,7]. A modular architecture disintegrates the node software suite into a set of independent module; A module encapsulates individual software element such as the OS kernel, the network stack, and an application, and is coupled with

† This research was supported by the Daegu University Research Grant, 2011

1) Daegu University, Dept. of Embedded system engineering, First author

2) Daegu University, Dept. of Multimedia engineering. Corresponding author (wkhong@daegu.ac.kr)

other modules via inter-module function calls. Modular software architectures facilitate efficient software management by organizing complex software into a collection of discrete modules. Furthermore, modular software architectures may enhance energy-efficiency in software update by enabling to replace just the intended module rather than the entire code image.

Over the years, a number of sensor OSes adopt modular approaches to their software architectures [8,9,10,11,12]. These approaches respectively contributes to addressing a wide spectrum of challenging issues stemming from stringent resource constraint, the limited memory model of a single-address space, and hardware restriction of MMU-less MCUs. However, a substantial body of the works do not fully address both flexibility and efficiency issues altogether, failing to delivering the architectural potential of modular software architectures fully into the context of WSNs.

For flexible an efficient modular software architectures for mote-class sensor nodes, a robust module linking model is essential. The module linking in our terminology means to link symbols between modules, and the way for module linking, which we call the module linking model, should always be employed in the design of a modular software architecture. Of important design principles, the underlying module linking model has a dominant effect on the degree of the flexibility and the efficiency of the modular sensor software architecture. Firstly, the underlying module linking model shapes the way to implement bindings for external symbol references which, in turn, determines the tightness of module linkage. As a consequence, the module linking model determines whether a module can be independently built or not, and whether the system allows for dynamic modular update. Secondly, the memory model for inter-module linkage enforced by the module linking model determines both symbol access latency for inter-module symbol references and the amount of memory required for its implementation.

This paper presents a taxonomy of module linking model and respectively their implementation frameworks. We establish a taxonomy for module linking models by exploring the design space of module linking models. We then suggest a implementation framework respectively for each module linking model in the taxonomy.

The remainder of the paper is organized as follows. Section 2 presents the proposed taxonomy of module linking models. Section 3 explains implementation frameworks respectively for module linking models exposed by the taxonomy. Finally, section 4 provides the concluding remark and future search issues.

2. Taxonomy of Module linking models

Module linking models can be crudely characterized by two attributes associated with their organizations. One attribute is where the linking operation is performed; linking can be performed either on the computer with the cross-development environment or th target sensor node. The other attribute is how a binding is implemented in memory; References to external symbols, i.e., external variables or functions, are made either directly to their physical addresses or indirectly to some pointers from which the external symbols can be dereferenced. These two attributes are orthogonal each other; we thus can obtain a two-dimensional classification framework for module linking models.

Offline versus online linking : The offline linking refers to the module linking performed on offline computers as in conventional cross-compilation environments. Under this module linking, given a module to update, the code to be disseminated to target sensor nodes from the offline computer is the fully linked excutable code generated on the offline computers. The online linking refers to the module linking performed on target sensor nodes. In this case, given a module to update, the code disseminated to the target sensor node from the offline computers remains unlinked; The linking process is performed on the target sensor node.

Direct versus indirect binding : In the directly bound code, the physical address of the external symbol is placed in the code image for each external symbol reference. The indirectly bound code, the physical address of a pointer is placed in the code image for each external symbol reference. The pointer is used at runtime for dereferencing its target symbol. Ordinary public or commercial tool-chains of state-of-the-art 8-Bit MCUs, support only a single monolithic code image generated only for a single program that does not allows incremental compilation for late binding. The indirect binding is not feasible without some non-trivial effort to code instrumentation. Therefore, to support indirect binding, an appropriate binding environment is to be designed. The binding environment means the memory organization for symbol pointers to support external symbol references.

Above two attributes provide a simple, but comprehensive framework for classifying linking models because they encompass the effective design space. According to this framework, we can identify four linking models.

3. Implementation frameworks

This section presents the implementation framework for each module linking model exposed by our taxonomy, detailing the development method and memory model for binding environments as well as relative advantages and disadvantages.

3.1 Offline direct module linking(OFD)

This is the module linking model in which linking operations are carried out on offline computers and the external symbol references are implemented by the direct binding. As linking operation is carried out on offline computers, given a module to update, the code to be disseminated to the target sensor node is in the form of a fully linked executable image generated from the linking. Due to direct binding, the module's executable image keeps each external reference bound with the absolute address of the symbol. This form of binding is, in fact, the same with the one resulting from the conventional monolithic development. This is why some projects adopting this model discard it in their future projects[8]

Given a module of interest to update, the binary image for the module cannot be handled independently, but the whole system modules should be rebuilt into a new executable image; the module's fully linked executable image is included in the executable image for the entire modules. This is because both the module of interest and the other modules may require rebuilding as explained below: In general, a module of interest to update may have both public symbols provided for other modules and external symbol references. Hence, the consumer modules which refers to public symbols in the module of interest should also be rebuilt with the module of interest to renew the absolute addresses of the public symbols belonging to the module of interest. Otherwise, the referential inconsistency occurs in consumer modules against all the public symbols owned by the module of interest.

The OFD model provides a simple and efficient way to implement modular software architecture, enabling to benefit from modular approach itself. However, it does not have any substantial architectural merit over the monolithic architecture except the modularity in programming. The practical use of modular dissemination and update is not feasible with this model.

3.2 Online direct module linking(OND)

This is the module linking model in which linking operations are carried out on the target sensor node, and the external symbol references are implemented by direct binding. Due to direct binding, the module's executable image keeps each external reference bound with the absolute address of the symbol. As noted earlier, this form of binding is also the same with the one resulting from the conventional monolithic development.

The OND model has specific requirements respectively for code building and for development tools. Firstly, the code building is separated into two distinct steps. The compilation is made offline under the cross-compiler and the linking is performed on the target sensor node. The main reason for offline compilation is that online compilers on resource constraint of sensor nodes are not available and it is not reasonable to newly implement them. On the other hand, the module linking is made online in accordance with the nature of the OND model. Secondly, in terms of development tools, the OND model demands an online linker that generates the executable code through ordinary linking operations. For the OND model, an online linker must be implemented because off-the-shelf online linkers are not readily available. Some projects show that it is practically possible to implement online linkers even under resource-constrained sensor node[9].

When online linkers are available, the scenario for updating a module is as follows. Given a module to be updated, the intermediate object file is produced through the offline compilation and then is transferred over-the-air to the target sensor node. Taking the intermediate object file, the online linker performs the module linking with the intermediate object files for the remaining modules stored in flash memory, producing a fully linked binary image for the entire node software. Loaded in the system flash, the executable image will be executed, restarting the system.

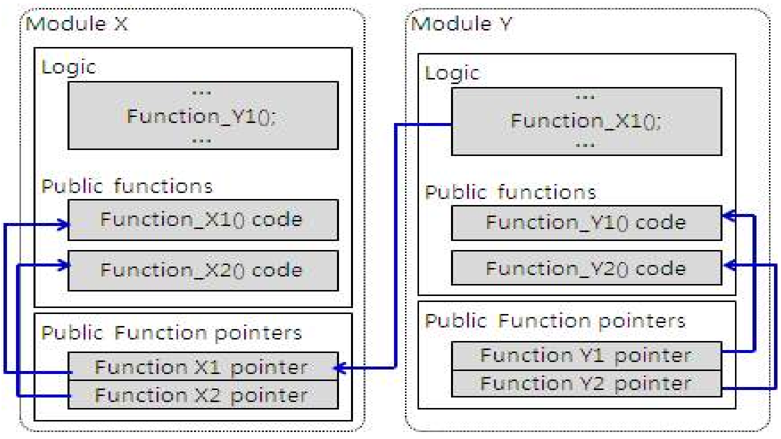
This module linking model allows energy-efficient reprogramming as it supports modular dissemination. The energy efficiency of this model would be quite higher than the OFD model which do not support the modular dissemination. However, it is to some degree lower than other better models such as the OFI and ONI model because the symbol information included in the intermediate object file causes additional cost in transmission energy. Next, as module linking is handled by online linker, developers are not burdened with any supplemental code instrumentation to support module linking.

This module linking model has also some disadvantages and limitations. Firstly, an online linker must be developed. It should be challenging to develop a linker on the small memory capacity of sensor nodes. The online linker should be maximally interoperable enough to support conventional object file formats such as the ELF file format. Another disadvantage is severe memory overhead. This model retains several sources of the memory overhead. The intermediate object files for the rest of modules are to be maintained in the flash memory of the sensor node; The online linker uses them to produce a new executable image for the entire node software when a new module is to be updated. Moreover, it is required that the online linker itself be kept in the flash memory of the sensor nodes.

3.3 Offline indirect module linking(OFI)

This is an OFI model in which linking operations are carried out on offline computers and the external symbol references are implemented by indirect binding. Independent of the other modules, a module is built into a fully linked executable image on an offline computer. In the module's executable image, each external reference in the executable image is encoded with the absolute address of the symbol pointer rather than directly with the symbol address. The symbol pointer keeps the memory address of its symbol.

For this model, the binding environment must



<Fig. 1> Example view of modules in the OFI model

satisfy the following two requirements. The first one is associated with symbol references in the code. All the external symbol references in a module should be able to be set to the addresses of their symbol pointers at compile-time. This implies that the addresses of all the symbol pointers for its external symbol references are explicitly known at module compile-time. The second requirement is associated with symbol pointers in the binding environment. Each symbol pointer should be able to be set statically at build time to the memory address of its symbol.

Our proposal to implementing the OFI model is based on a binding environment in which each module maintains a symbol pointer for each public symbol within the module. These pointers will be called collectively the per-module public symbol table in our terminology. The table provides, as a table entry, a symbol pointer for each of public symbols within the module. Each pointer in the table keeps the address of its public symbol. External references from other modules to the public symbols are made via these table entries. This design satisfies the two requirements for the binding environment as follows. First, addresses of symbol pointers can be explicitly exposed to other modules by just statically allocating the table in a predetermined location memory. Second, as both the table and public symbols belong to the same module source code, setting each table entry to its symbol address can be made without any difficulty.

When employing the proposed binding environment, a module consists of three parts. The first one is the code that implements the module's own logic. The second one consists of public functions that the module provides externally for other modules. The third part is the per-module symbol table where each table entry serve as a function pointer respectively to a public function in the module. <Fig. 1> shows the logical view of the module organization. For brevity, it is assumed that only functions are allowed for external accesses by other modules. In module X, the per-module symbol table contains the addresses of public functions defined in module X. In module Y, an external reference to module X is bound to the address of a per-module symbol table entry in module X.

The OFI model can be viewed as an approach to enhance the architectural flexibility by breaking the tight coupling through the indirection. As the external references are not directly coupled with the physical address of the external symbols, a new version of a module can be seamlessly updated, enabling energy efficient modular dissemination and dynamic modular update. But, the developer has some burden to determine the memory space for the per-module public table.

3.4 Online indirect module linking(ONI)

This is the module linking model in which linking operations are carried out on offline computers and the external symbol references are implemented by indirect binding. In the module's executable image, each external reference is encoded with the absolute address of the symbol pointer. The code from the offline computer remains unlinked; Each symbol pointer in the executable image remains empty without being set to the memory address of its symbol. These pointers are filled later through the module linking model on the target node.

As discussed for the OFI model, the binding environment, i.e., organization of symbol pointers, needs to be addressed because the ONI model also adopts the indirect binding. To support the ONI model, the organization must satisfy the following two requirements. Firstly, as for the bindings for symbol references, it must be possible to set all the external symbol references in the module to the addresses of their symbol pointers at module load time. Secondly, as for the binding for symbol pointers, each symbol pointer should be able to be set at the load time such that it can be dereferenced to the memory address of its symbol. Note that these two conditions is to address respectively the indirect binding and the online linking.

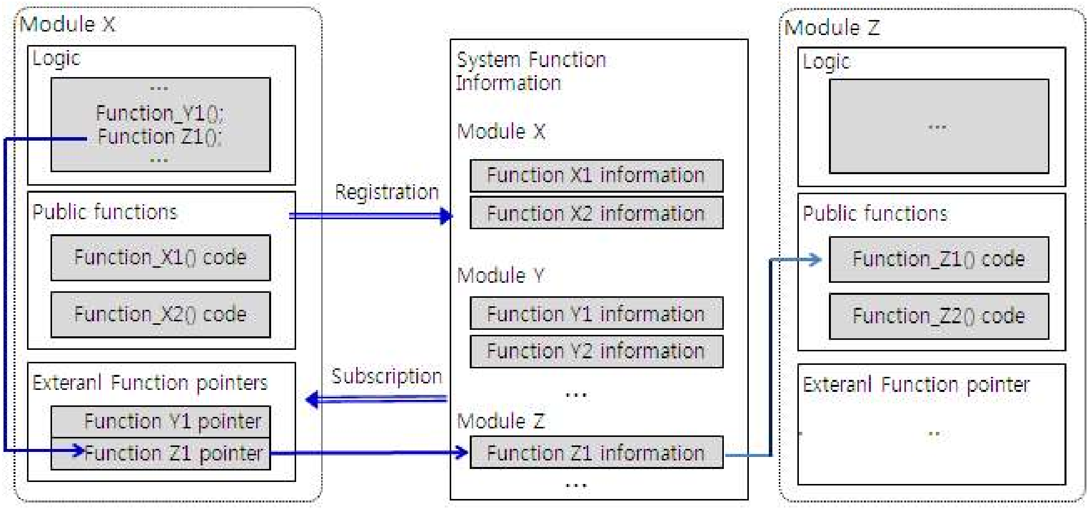
Our proposal to implementing the ONI model is based on a binding environment in which each module maintains a symbol pointer respectively for each external symbol referenced by the module. These pointers will be collectively called the per-module external symbol table. The table provides, as a table entry, a symbol pointer for each of external symbols referenced in the module. Each pointer in the table keeps the address of its external symbol. External references within the module are made via these table entries. The per-module external symbol table satisfies both requirements mentioned earlier. Firstly, symbol pointers for external symbol references are naturally exposed at module build time because the table belongs to the module. The external symbol references can thus be statically bound to addresses of their symbol pointers. Secondly, the problem of binding for symbol pointers can be solved by using some dynamic mechanisms. These will be illustrated shortly.

<Fig. 2> shows the organization of modules based on per-module external symbol tables. Each module consists of the module logic, public functions, and a per-module external symbol table. In module X, an external symbol reference to a symbol in module Z is bound to the address of its per-module external table entry.

The main issue for implementing the per-module external symbol table for the ONI model is how to bind symbol pointers. It can be achieved by registration and subscription concepts as suggested in [10]. Registration refers to the operation that a module notifies the system of the information on its public symbols, and subscription refers to the operation that a module obtains information about their external symbols. The system keeps the information registered by each module and supply the information upon requests.

Given a module, bindings for symbol pointers are made though registration and subscription at module load time. The module registers addresses of its global symbols to the system symbol table at the module load time. A module with any references to external symbols subscribes to the symbols. Through the subscription, the module can obtain absolute addresses of the external symbols and places them to its own per-module external symbol table. Once the module finishes the registration and the subscription, it becomes fully linked with other modules.

When implementing this model based on the per-module external symbol table, we can benefit from two important features. Firstly, each module source code is self-contained without any source-level dependence on other modules. Bindings for symbol pointers are the only information de-



<Fig. 2> Example logical view of modules in the ONI model

pendent on other modules, but they are obtained dynamically. This independency introduces simplicity in module programming and building over th OFI model. The developers are not burdened with static memory space allocation for per-module tables across all the modules. Next, the per-module external symbol table and its associated instrumental code can be programed in conventional high level language such as C language, as does for the OFI model. This allows that ordinary off-the-self compilers and linkers are sufficient in generating the executable image for a module.

This model prioritizes complexity of code instrumentation over the burden of the online linker; The linking process in this model is very light-weight, whereas application developers are burdened with some code instrumentation. In spite of the burden of the instrumentation code, this model supports both modular dissemination and dynamic modular update.

4. Conclusions

The paper provides the first comprehensive exploration on the module linking issue in building the modular software architecture especially within the context of WSNs. We identify two core attributes, the place where module linking is conducted and the memory structure that implements the external symbol binding, as the essential factors that shapes the operation and performance of module linking models. The taxonomy based on those attributes is very simple, but comprehensive for exposing the design space of module linking models. The module linking models exposed by the taxonomy and their implementation framework suggested in this paper would serve as a foundation for designing more flexible and efficient modular software architectures. As a future research, we will investigate the existing systems and their implementation from the perspective of the proposed taxonomy to suggest any enhancement of the implementations of future modular architectures.

References

[1] V. Gallart, S. Felici-Castell, and M. Delamo, A. Foster, and J. Perez, "Evaluation of a Real, Low Cost, Urban WSN Deployment for Accurate Environmental Monitoring", 8th International Conference on Mobile Ad-Hoc and Sensor Systems", IEEE, pp. 634-639, October 2011.

[2] G. Xing, J. Chen, W. Song, and R Huang, "Fusion-based Volcanic Earthquake detection and timing in Wireless Sensor Networks", ACM Transaction on Sensor Networks, Vol. 9, No. 4, pp 1-25, 2013.

[3] M. Strube, R. Kapitza, K. Stengel, M. Daum, and F. Dressler, "Stateful Mobile Modules for Sensor Networks", 6th IEEE International Conference on Distributed Computing in Sensor Systems, Springer-Verlag, pp. 63-76, June 2010.

[4] G. Coulson et al., "A generic component model for building systems software", ACM Transaction on Computer Systems, Vol. 26, Issue 1, pp. 1-42, 2008.

[5] L. Mottola, G. Picco, and A. Sheikh, "FiGaRo: Fine-Grained Software Reconfiguration for Wireless Sensor Networks". 5th European Conference on Wireless Sensor Networks, Springer-Verlag, pp. 286-304, January 2008.

[6] D. Balasubramaniam, A. Dearle, and R. Morrison, "A Composition-based Approach to the Construction and Dynamic Reconfiguration of Wireless Sensor Network Applications", 7th International Conference on Software Composition, Springer-Verlag, pp. 206-214, March 2008.

[7] W. Dong, C. Chen, X. Liu, Y. Liu, J. Bu, and K. Zheng, "SenSpire OS: A Predictable, Flexible, and Efficient Operating System for Wireless Sensor Networks", IEEE Transactions on Computers, Vol. 60, No. 12, pp. 1788-1801, 2011.

[8] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors", 29th Annual IEEE International Conference on Logical Computing Networks, IEEE, pp. 455-462, November 2004.

[9] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks", 4th International Conference on Embedded networked Sensor Systems, ACM, pp 15-28, October 2006.

[10] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A Dynamic Operating System for Sensor Nodes", 3th International Conference on Mobile systems, applications, and services", ACM, pp 163-176, June 2005.

[11] H. Cha, S. Choi, I. Jung, H. Kim, and H. Shin, "RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks," 6th International Conference on Information Processing in Sensor Networks, ACM/IEEE, pp. 148-157, April 2007.

Hiecheol Kim (김 희 철)

• Member
• BS, Dept of Computer Science, Yonsei university
• MS.Dept. of Computer Eng. Univ. of Southern California
• Ph.D .Dept. of Computer Eng. Univ. of Southern California
• Professor, Daegu University, Dept. of Embedded system engineering, Professor
• Interest : Realtime embedded OS, Wireless sensor network

Won-Kee Hong (홍 원 기)

• Member
• BS, Dept of Computer Science, Yonsei university
• MS, Dept of Computer Science, Yonsei university
• Ph.D Dept of Computer Science, Yonsei university
• Associate Professor, Daegu University, Dept. of Multimedia engineering.
• Interest : Realtime embedded OS, Wireless sensor network

논 문 접 수 일 : 2014년 03월 13일
1 차 수 정 완 료 일 : 2014년 04월 07일
2 차 수 정 완 료 일 : 2014년 04월 15일
게 재 확 정 일 : 2014년 04월 22일