TINET+TECS: Component-based TCP/IP Protocol Stack for Embedded Systems

Takuro Yamamoto*, Takuma Hara†, Takuya Ishikawa†, Hiroshi Oyama‡, Hiroaki Takada† and Takuya Azumi*
*Graduate School of Engineering Science, Osaka University
†Graduate School of Information Science, Nagoya University
‡OKUMA Corporation

***Abstract*—Embedded systems are applied to Internet of Things (IoT), and the high productivity of embedded network software is required. TINET is a TCP/IP protocol stack for embedded systems. Although TINET is a compact TCP/IP protocol stack, it consists of many complex source codes. Therefore, it is difficult to maintain, extend, and analysis the software.**

To improve the scalability and configurability, this paper has proposed a component-based TCP/IP protocol stack for embedded systems: TINET componentized with TOPPERS embedded component system (TECS). The component-based TINET provides software developers high productivity such as change of network buffer size and adding/removing TCP (or UDP) function. We evaluate the component-based TINET compared with the original TINET. We confirm that the overheads of execution time and memory consumption are low, and that the configurability is improved.

In this paper, we also demonstrates a use case of the dynamic component connection method of TECS with a network application.

I. Introduction

Internet of Things (IoT) is an essential keyword for the next era. Such as wearable devices, smart devices, and smart homes, various things are connected to the Internet, enriching our lives. Embedded systems are the elements constituting IoT, e.g., sensing data and controlling actuators. It is not practical to implement the same communication function as a general computer because embedded systems have several restrictions such as low memory capacity.

TINET (Tomakomai InterNETworking) is a compact TCP/IP protocol stack for embedded systems [1]. TINET supports the ability such as minimum copy frequency and elimination of dynamic memory control. TINET needs only small memory for its TCP/IP protocol stack; therefore it is suitable for embedded systems. However, there are several issues that TINET consists of many complex source codes. In other words, TINET is composed of many files and defines many macros. This may take a lot of time for software developers to maintain, extend, and analysis the software. Embedded network software is required for the high productivity and quality.

Component-based development is an approach to improve software productivity. Component-based development is a design technique that can be applied to reusable software development [2] [3], such as TECS [4], AUTOSAR [5], and SaveCCM [6]. Component-based systems are flexible to software extension and specification changes. In addition, individual component diagrams enable the visualization of an entire system.

In this paper, component-based TCP/IP protocol stack, i.e., componentized TINET, is proposed to improve the configurability and scalability of software. TECS (TOPPERS Embedded Component System) [4] is utilized to componentize TINET, because TECS is a component system suitable for embedded systems. TECS supports static configuration, which statically defines component behaviors and interconnections, thus TECS can optimize the overhead of componentization. This paper evaluates the overheads of execution time and memory consumption and the amount of code line change for adding/removing the functionalities, which demonstrates TINET+TECS can improve the configurability with small overheads.

Contributions: This paper provides the following contributions.

1) **Improve configurability**
   TINET+TECS is a component-based system, therefore the software is flexible to change the configuration such as resizing network buffer, adding/removing TCP (or UDP) functions, and support both IPv4 and IPv6. In addition, a component diagram provides visualization of TINET, a complicated system.
2) **Dynamic connection method**
3) **Support legacy codes**
   TECS supports the adapter to call TECS functions from existing C codes, thus TINET+TECS can be applied to an existing application.

Organization: The remainder of this paper is organized as follows. Section II introduces the system model and basic technologies, i.e., TINET and TECS. Section III describes the design and implementation of the proposed framework. Section IV evaluates the proposed framework. Related work is discussed in Section V. Conclusions and suggestions for future work are presented in Section VI.

II. System Model

This section describes system model, including the basic technologies such as TINET and TECS. System model of the proposed system is shown as Fig. 1.

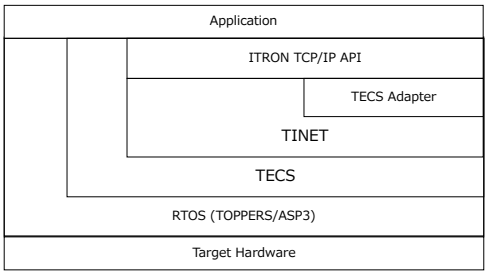

Fig. 1. System model

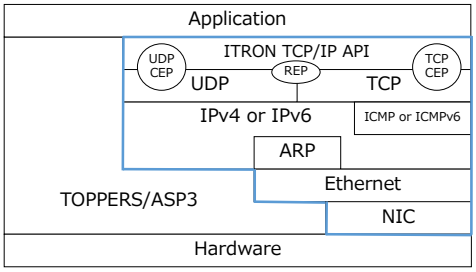

Fig. 2. Hierarchy diagram of TINET and TOPPERS/ASP3

A. TINET

TINET is a compact TCP/IP protocol stack for embedded systems based on the ITRON TCP/IP API Specification [7], developed by TOPPERS (Toyohashi OPen Platform for Embedded Real-time Systems) Project [8]. TINET has been released as open source.

To satisfy restrictions for embedded systems such as memory capacity, size, and power consumption, TINET supports following functions:

- Minimum copy frequency
- Elimination of dynamic memory control
- Asynchronous interface
- Error detailed per API

1) Overview: TINET runs as middleware on TOPPERS/ASP3 [9], which is a realtime kernel based on μITRON [10]. TINET also supports other RTOSs such as TOPPERS/ASP and TOPPERS/JSP because TINET is compatible with TOPPERS RTOS. Fig. 2 shows the hierarchy diagram of TINET and TOPPERS/ASP3. Users transmit and receive the data using a Communication End Point (CEP) which is an interface like a socket. In transmission process, headers are attached to the data body passed to the CEP at each protocol layer, and the data is transmitted from the network device. In reception process, headers of the data body received in the network device are analyzed at each protocol layer, and the data is passed to the CEP.

In TINET, the number of the data copy between each protocol layers is minimized. A TCP/IP protocol stack for general computers has large overheads of execution time and memory consumption because the data is copied at each protocol layers. To solve the problem, TINET does pass the pointer of the data buffer between each protocol layers, not perform data copy.

B. TECS

TECS is a component system suitable for embedded systems. TECS can increase productivity and reduce development costs owing to improved reusability of software components. TECS also provides component diagrams, which help developers visualize the overall structure of a system.

In TECS, component deployment and composition are performed statically. Consequently, connecting components does not incur significant overhead and memory requirements can be reduced. TECS can be implemented in C, and demonstrates various feature such as source level portability and fine-grained components.

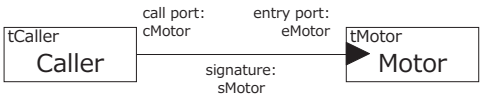

Fig. 3. Component Diagram

1) Component Model: Fig.3 shows a component diagram. A *cell*, which is an instance of a component in TECS, consists of *entry* ports, *call* ports, attributes and internal variables. An *entry* port is an interface that provides functions to other *cell*s, and a *call* port is an interface that enables the use of other *cell*'s functions. A *cell* has one or more *entry* ports and *call* ports. *Cell* functions are implemented in C.

The type of *entry*/*call* port is defined by a *signature*, which is a set of functions. A *signature* is the interface definition of a *cell*. The *cell*'s *call* port can be connected to the *entry* port of another *cell* by the same *signature*. Note that *celltype* defines one or more *call*/*entry* ports, attributes, and internal variables of a *cell*.

2) Component Description: In TECS, components are described with component description language (CDL). CDL can be divided into three categories: *signature*, *celltype*, and build description. These components are described as follows.

Signature Description

The *signature* defines a *cell* interface. The *signature* name follows the keyword *signature* and takes the prefix "s", e.g., sMotor (Fig.4). In TECS, to clarify the function of an interface, specifiers such as [in] and [out] are used, which represent input and output respectively.

Celltype Description

The *celltype* defines *entry* ports, *call* ports, attributes, and variables. A *celltype* name with the prefix "t" follows the keyword *celltype*, e.g., tCaller (Fig.5). To define *entry* ports, a *signature*, e.g., sMotor, and an *entry* port name, e.g., eMotor, follow the keyword *entry*. *Call* ports are defined similarly. Attributes and variables follow the keywords *attr* and *var*, respectively.

```
signature sMotor {
   int32_t getCounts( void );
   ER resetCounts( void );
   ER setPower( [in]int power );
   ER stop( [in] bool_t brake );
   ER rotate( [in] int degrees,
              [in] uint32_t speed_abs,
               [in] bool_t blocking );
   void initializePort( [in]int32_t type
       );
};
```

Fig. 4. Signature Description

```
celltype tCaller {
   call sMotor cMotor;
};
celltype tMotor {
   entry sMotor eMotor;
   attr {
      int32_t port;
   };
   var {
      int32_t currentSpeed = 0;
   };
};
```

Fig. 5. Celltype Description

Build Description

The build description is used to instantiate and connect *cell*s. Fig.6 shows an example of a build description. A *celltype* name and *cell* name, e.g., tMotor and Motor, respectively, follow the keyword *cell*. A *call* port, *cell*'s name, and an *entry* port are described in that order to compose *cell*s, In Fig. 6, *entry* port eMotor in *cell* Motor is connected to *call* port cMotor in *cell* Caller. *C_EXP* calls macros defined in C files.

III. Design and Implementation

TINET+TECS, the proposed componentized TCP/IP protocol stack, consists of some TECS components. This section describes the components with component diagrams. In addition, TECS functionalities applied to the proposed system such as *send*/*receive* specifier and adapter are explained.

A. Components of protocol stack

The components of protocol stack for TINET+TECS is shown as Fig. 7. Note that some small particle components such as a kernel object, dataqueques, and semaphores are omitted to simplify the component diagram. In TINET+TECS, the components are divided for each protocol, and the functionalities such as input function and output function are defined as each a component. Therefore, software visibility is improving because of small grain components.

To utilize the protocol stack as same as the original TINET, communication object components such as tTCPCEP, tUDPCEP, and tREP are defined as an interface between TINET+TECS and an application. The communication object component is a component corresponding to a CEP or REP of

```
cell tMotor Motor {
   port = C_EXP("PORT_A");
};
cell tCaller Caller {
   cMotor = Motor.eMotor;
};
```

Fig. 6. Build Description

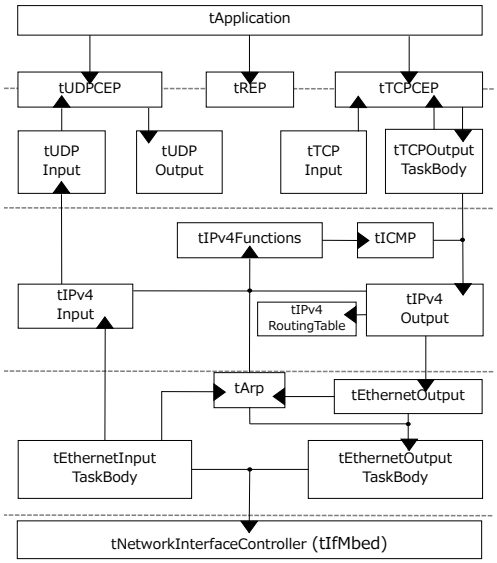

Fig. 7. Component diagram of protocol stack

the original TINET. An application developers can utilize the TINET+TECS functionalities by generating and combining as many components as necessary.

The protocol stack of TINET+TECS supports coexistence of multiple protocols. By developing the IPv6 and Point-to-Point Protocol (PPP) components, TINET+TECS can make IPv4 and IPv6 coexist and support PPP without a modification of component implementation.

B. Memory allocator component

1) send/receive specifier: TECS supports *send*/*receive* specifiers, which are interface specifiers [11]. TINET+TECS uses *send* and *receive* specifiers instead of *in* and *out* to reduce the number of copies.

in is a specifier for input arguments. A callee side uses the memory of arguments with *in* during executing the callee function. When the processing returns to the caller side, the caller can reuse and deallocate the memory.

send is also a specifier for transferring data to a callee from a caller such as *in*. The difference between *in* and *send* is whether to deallocate the data memory in a caller or callee, shown as Fig. 8. In case of *in* specifier, both allocating and deallocating the data memory are performed in the caller. On the other hand, in case of *send*, the caller allocates the data memory and the callee deallocates it.

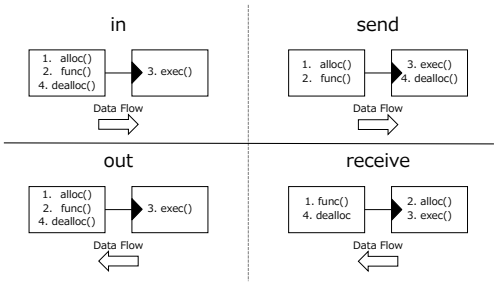

Fig. 8. Differences between in/out and send/receive

```
1 signature sNicDriver {
2   void start([send(sNetworkAlloc),size_is(
        size)]int8_t *outputp,
3              [in]int32_t size,
4              [in]uint8_t align);
5   void read([receive(sNetworkAlloc),
        size_is(*size)]int8_t **inputp,
6             [out]int32_t *size,
7             [in]uint8_t align);
8     /* Omit: other functions */
9 };
```

Fig. 9. An example of send and receive

out is a specifier for output arguments. A callee writes data in the memory allocated by a caller, and the caller receives the data.

receive is also a specifier for a caller receiving data from a callee such as *out*. The difference between *out* and *receive* is whether to allocate the data memory in a caller or callee, shown as Fig. 8. While, in case of *out*, a callee writes data in the memory allocated by a caller, in case of *receive*, the callee allocates the data memory. Deallocating the memory is performed in the caller in both cases.

C. Network timer component

D. TECS Adapter

TECS supports *Adapter* functionality which enables to call a function in TECS from existing C codes. An adapter connects a TECS component, and links a C function to a TECS function shown as Fig. 10. Software developers can utilize an existing application for TECS owing to the adapter.

IV. Evaluation and Demonstration

A. Evaluation environment

GR-PEACH is employed as the evaluation board. We connected the board and the host PC with a LAN cable, and evaluated the data transmission and reception. The detail specification of the board is shown in TABLE I. We also employ TINET 1.5.4 and the compiler arm-none-eabi–gcc 5.2.

V. Related Work

Open-source TCP/IP protocol stacks for embedded systems have been developed such uIP [12].

uIP: uIP (microIP) is a very small TCP/IP stack intended for tiny 8-bit and 16-bit microcontrollers. uIP only requires about five kilobytes of code size and several hundred bytes of RAM. uIP has been ported to various systems and has found its way into many commercial products. After ver. 1.0 is released, later versions of uIP, including uIPv6, are integrated with Contiki OS [13], [14], an operating system to connect tiny microcontroller to the Internet.

lwIP: lwIP (lightweightIP) is a small TCP/IP implementation for embedded systems. The focus of lwIP is to reduce memory resource while still having a full scale TCP. lwIP requires about 40 kilobytes of ROM and tens of kilobytes of RAM. lwIP is larger than uIP, but provides better throughput.

CiAO/IP:

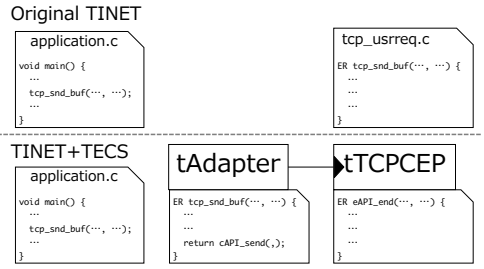

Fig. 10. TECS Adapter

TABLE I
Evaluation Board Environment

Board	GR-PEACH
CPU	Cortex-A9 RZ/A1H 400MHz
Flash ROM	8 MB
RAM	10 MB
LAN Controller	LAN8710A

VI. Conclusion

Acknowledgment

The authers would like to thank OO for supporting this research. This work is supported by JSPS KAKENHI Grant Number 15H05305.

References

[1] TOPPERS, “TINET,” https://www.toppers.jp/en/tinet.html.
[2] I. Crnkovic, “Component-based Software Engineering for Embedded Systems,” in *Proceedings of the 27th International Conference on Software Engineering*, 2005, pp. 712–713.
[3] X. Cai, M. R. Lyu, K.-F. Wong, and R. Ko, “Component-based software engineering: technologies, development frameworks, and quality assurance schemes,” in *Processings of 7th Asia-Pacific Software Engineering Conference (APSEC)*, 2000, pp. 372–379.
[4] T. Azumi, M. Yamamoto, Y. Kominami, N. Takagi, H. Oyama, and H. Takada, “A New Specification of Software Components for Embedded Systems,” in *Proceedings of the 10th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2007, pp. 46–50.
[5] “AUTOSAR,” http://www.autosar.org/.
[6] M. kerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli, “The SAVE Approach to Component-based Development of Vehicular Systems,” *Journal of Systems and Software*, vol. 80, no. 5, pp. 655–667, 2007.
[7] “ITRON TCP/IP API Specification (Ver. 1.00.01, only Japanese version is available),” http://www.ertl.jp/ITRON/SPEC/tcpip-e.html.
[8] “TOPPERS Project,” http://www.toppers.jp/en/index.html.

[9] TOPPERS, “TOPPERS/ASP3 kernel,” https://www.toppers.jp/asp3-kernel.html.
[10] H. Takada and K. Sakamura, “μITRON for Small-Scale Embedded Systems,” *IEEE Micro*, vol. 15, no. 6, pp. 46–54, 1995.
[11] T. Azumi, H. Oyama, and H. Takada, “Memory Allocator for Efficient Task Communications by Using RPC Channels in an Embedded Component System,” in *Proceedings of the 9th IASTED International Conference on Software Engineering and Applications*, 2008, pp. 204–209.
[12] A. Dunkels, “Full tcp/ip for 8-bit architectures,” in *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, ser. MobiSys ’03. New York, NY, USA: ACM, 2003, pp. 85–98. [Online]. Available: http://doi.acm.org/10.1145/1066116.1066118
[13] P. Dutta and A. Dunkels, “Operating systems and network protocols for wireless sensor networks,” *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 370, no. 1958, pp. 68–84, 2011. [Online]. Available: http://rsta.royalsocietypublishing.org/content/370/1958/68
[14] “Contiki OS,” http://www.contiki-os.org/.