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 proposes TINET+TECS, 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. TINET+TECS utilizes a dynamic connection method of TECS components to satisfy the original TINET specification. We evaluate the component-based TINET compared with the original TINET. Experimental results show that the overheads of execution time and memory consumption are low, and that the configurability is improved.**

I. Introduction

Internet of Things (IoT) is an essential keyword for the next era [1] [2]. Various things, such as wearable devices, smart devices, and smart homes, connected to the Internet will enrich our lives. Embedded systems are the elements constituting IoT, e.g., sensing data and controlling actuators. It is not practical to implement the same TCP/IP protocol stack 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 [3]. 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 file and define 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.

An approach to improve software productivity is component-based development, which is a design technique that can be applied to reusable software development for embedded systems [4] [5], such as TECS [6] [7], AUTOSAR [8], and SaveCCM [9]. Component-based systems are fl xible to software extension and specificatio changes. In addition, individual component diagrams enable the visualization of an entire system.

This paper proposes a component-based TCP/IP protocol stack for embedded systems, i.e., TINET+TECS, to improve the configurabilit and scalability of TCP/IP software. TECS (TOPPERS Embedded Component System) [6] [7] is utilized to componentize TINET, because TECS is a component system suitable for embedded systems. TECS supports static configuration which statically define component behaviors and interconnections. Thus, TECS can optimize the overhead of componentization.

In addition, satisfying the original TINET specification the proposed framework utilizes dynamic connection, a method of TECS, to dynamically switch component bindings. A general TCP/IP protocol server dynamically processes the requested port, i.e., HTTP (port: 80) and HTTPS (port: 443). However, embedded systems are restricted dynamic processing due to the strict memory restriction. TINET supports static generation of CEPs and REPs which are like sockets. As TINET+TECS statically generates components and dynamically combines them as well as TINET, TINET+TECS reduces dynamically increasing memory consumption.

In the proposed framework, software applications can be developed by not only TECS method but also existing method. The software applications can be developed as a TECS component because TINET+TECS is a component-based framework using TECS. Moreover, TECS supports an adapter to call functions of TECS components from non TECS codes. The adapter allows the use of existing TCP/IP applications without modification

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 configurabilit with small overheads. Moreover, the advantages of dynamic connection in terms of memory consumption and the low overhead of the TECS adapter are demonstrated.

Contributions: This paper provides the following contributions.

1) **Improve configurability**
   Since TINET+TECS is a component-based system, the software is fl xible to change the configuratio such as resizing network buffer, adding/removing TCP (or UDP) functions, and supporting both IPv4 and IPv6. In addition,

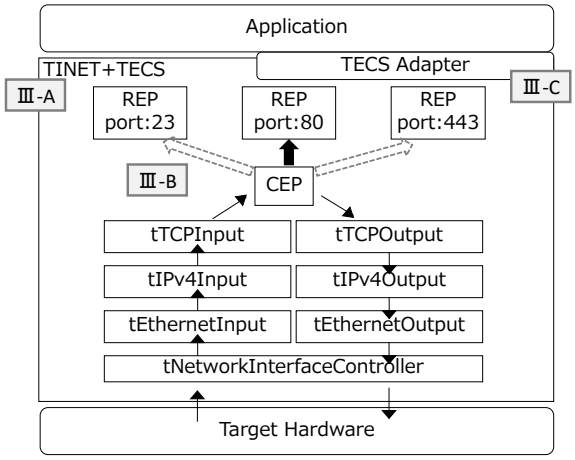

Fig. 1. System model of TINET+TECS

a component diagram provides visualization of TINET, a complicated system.

2) **Dynamic connection method**
Dynamically switching the binding of components, that is switching between a communication endpoint and reception point of TINET, realizes a TCP/IP protocol stack for embedded systems.

3) **Support legacy codes**
TINET+TECS can be applied to an existing application because TECS supports the adapter to call TECS functions from C codes.

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 and demonstrates the advantages. Related work is discussed in Section V. Conclusions and suggestions for future work are presented in Section VI.

II. System Model

This section describes the system model of TINET+TECS, including the basic technologies such as TINET and TECS. System model of the proposed framework is shown in Fig. 1. TINET+TECS is a component-based TCP/IP protocol stack, and the TCP output task (tTCPOutput) and Ethernet input task (tEhternetInput) are implemented as TECS components. CEPs and REPs (Section II-A), which are also implemented as TECS components, dynamically switch bindings by TECS method. Moreover, TECS adapter supports the legacy codes for existing TCP/IP applications.

A. TINET

TINET is a compact TCP/IP protocol stack for embedded systems based on the ITRON[1] TCP/IP API Specificatio [10], developed by TOPPERS (Toyohashi OPen Platform for Embedded Real-time Systems) Project [11]. TINET has been released as open source.

[1]ITRON is a realtime operating system (RTOS) developed by TRON project.

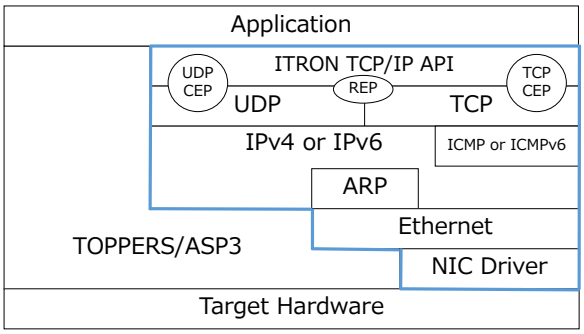

Fig. 2. Hierarchy diagram of TINET and TOPPERS/ASP3

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 [12] [13], which is a realtime kernel based on μITRON [14]. 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.

A TCP reception point called Reception Point (REP) is prepared to wait for a connection request from the partner side. An REP has an IP address (*myaddr*) and a port number (*myportno*) as attributes, and performs functions like *bind()* and *listen()*.

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.

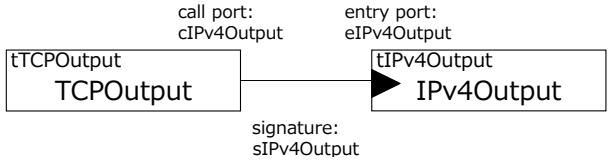

Fig. 3. Component Diagram

```
signature sIPv4Output {
  T_IN4_ADDR getIPv4Address(void);
  ER getOffset([inout]T_OFF_BUF *offset);
  ER setHeader([inout,size_is(size)]
                  int8_t *outputp,
                  [in]int32_t size,
                  [in]T_IN4_ADDR dstaddr,
                  [in]T_IN4_ADDR srcaddr);
    /* Omit: other functions */
};
```

Fig. 4. Signature description

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

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 define by a *signature*, which is a set of functions. A *signature* is the interface definitio 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* define 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* define a *cell* interface. The *signature* name follows the keyword *signature* and takes the prefi "s", e.g., sIPv4Output (Fig. 4). In TECS, to clarify the function of an interface, specifier such as [in], [out], and [inout] are used, which represent input, output, and input/output respectively. [size_is(*len*)] represents an array of size *len*.

Celltype Description

The *celltype* define *entry* ports, *call* ports, attributes, and variables. A *celltype* name with the prefi "t" follows the keyword *celltype*, e.g., tIPv4Output (Fig. 5). To defin *entry* ports, a *signature*, e.g., sIPv4Output, and an *entry* port name, e.g., eIPv4Output, follow the keyword *entry*. *Call* ports are define similarly. Attributes and variables follow the keywords *attr* and *var*, respectively.

```
celltype tIPv4Output {
   /* Entry port */
   entry sIPv4Output eOutput;

   /* Call port */
   call sEthernetOutput cEthernetOutput;
   /* Omit: other call ports */

   attr { /* Attribute */
      uint16_t fragInit = 0;
   };
   var { /* Variable */
      uint16_t fragId = fragInit;
   };
};
```

Fig. 5. Celltype description

```
cell tIPv4Output IPv4Output {
   /* Omit: other build description */

   fragInit = 0; /* Attribute */
};
cell tTCPOutput TCPOutput {
   cIPv4Output = IPv4Output.eOutput;
   /* Omit: other build description */
};
```

Fig. 6. Build 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., tIPv4Output and IPv4Output, 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 eIPv4Output in *cell* IPv4Output is connected to *call* port cIPv4Output in *cell* TCPOutput. *C_EXP* calls macros define in C files

III. Design and Implementation

This section describes design and implementation of the proposed framework, TINET+TECS. The proposed framework is component-based TCP/IP protocol stack for embedded systems, i.e., componentized TINET using TECS. In addition, A TECS novel functionality, dynamic connection method, and TECS adapter to support legacy codes are described with the use case of the proposed framework.

A. TINET+TECS

TINET+TECS, the proposed componentized TCP/IP protocol stack, consists of some TECS components. This section describes the components of TINET+TECS framework with component diagrams.

1) Components of protocol stack: The components of a protocol stack for TINET+TECS is shown in 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

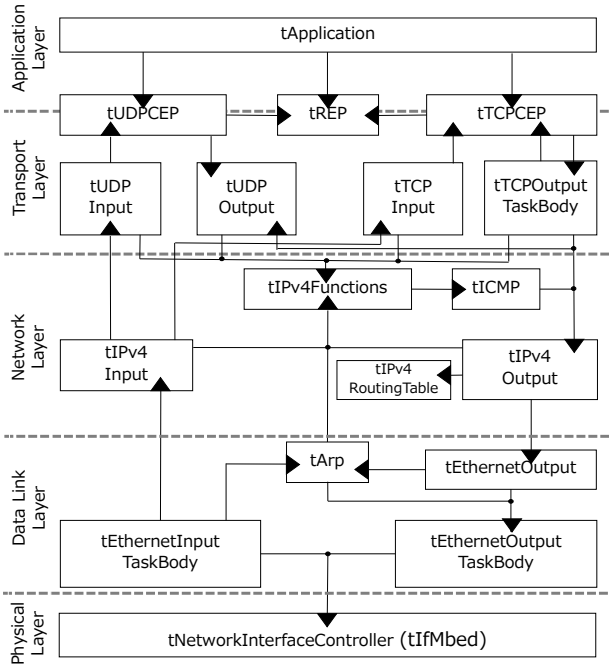

Fig. 7. Component diagram of protocol stack

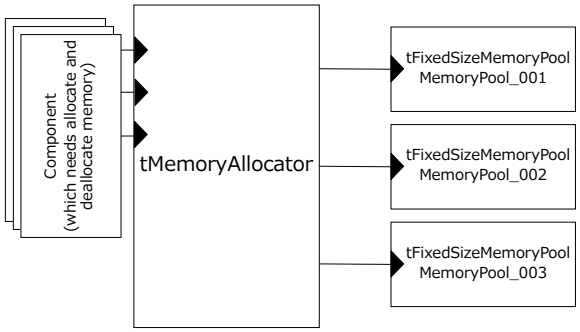

Fig. 8. Component diagram of tMemoryAllocator

divided for each protocol, and the functionalities such as input function and output function are define as each a component. Therefore, software visibility is improving because of small grain components.

The components of each protocol are described below.

Application layer: An application in TINET+TECS is implemented as a component such as tApplication. Software with TINET uses ITRON TCP/IP API [10] such as *tcp_snd_dat* and *tcp_rcv_dat*. In TINET+TECS, the application component calls TECS functions such as *cTCPAPI_sendData* and *cTCPAPI_receiveData*. Moreover, TINET+TECS supporting a TECS adapter (III-C), an existing application with TINET can run on TINET+TECS framework without transporting. Therefore, software can be developed either with existing methods or as a TECS component.

Transport layer: tTCPCEP (tUDPCEP) is a CEP component for TCP (UDP), and tREP is a REP component. For example, a server program supporting multiple clients can be developed by preparing the multiple tTCPCEP components. tTCPInput (tUDPInput) and tTCPOutput (tUDPOutput) are components performing the receiving and sending processing respectively in the transport layer.

Network layer: tIPv4Input and tIPv4Output are components performing the receiving and sending processing respectively in the network layer. tIPv4Functions component performs some functions such as checksum, tICMP component is for the ICMP protocol, and tIPv4RoutingTable component operates a routing table.

Data link layer: tEthernetInputTaskBody and tEthernetOutputTaskBody (tEthernetOutput) are components performing the receiving and sending processing respectively in the data link layer. tArp component is for the ARP protocol.

Physical layer: tNetworkInterfaceContoroller component implements a network device driver. Software can run on other devices by replacing the component because only the component depends on the target device.

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

2) Memory allocator component: The original TINET eliminates dynamic memory control to meet the severe memory restriction of embedded systems. A memory area for sending/receiving data in the protocol stack is allocated and released within a predetermined area. Memory allocator component performs the elimination of dynamic memory control in TINET+TECS. The component provides a requested memory area from the memory area statically allocated.

The memory allocator component connects to as many tFixedSizeMemoryPool as needed, shown in Fig. 8. tFixedSizeMemoryPool is a componentized kernel object of TOPPERS/ASP3 to alloc and release a memory area of the requested size. tFixedSizeMemoryPool components with various sizes are prepared, and an appropriate memory area can be allocated according to the used data size. On the other hands, all components which need allocate and deallocate memory connect to the memory allocator component, e.g., tTCPInput and tEthernetOutput.

In addition, TINET+TECS utilizes *send/receive* specifie of TECS to minimize the memory copy frequency, which is a functionality supported by TINET.

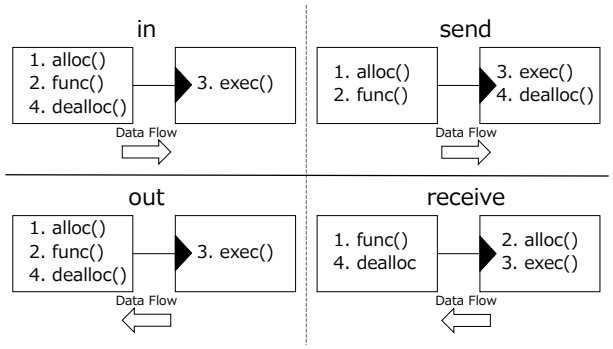

Fig. 9. Differences between in/out and send/receive

```
1  signature sNicDriver {
2    void start(
3      [send(sNetworkAlloc),size_is(size)]
4         int8_t *outputp,
5      [in]int32_t size,
6      [in]uint8_t align);
7    void read(
8      [receive(sNetworkAlloc),size_is(*size)]
9         int8_t **inputp,
10     [out]int32_t *size,
11     [in]uint8_t align);
12   /* Omit: other functions */
13 };
```

Fig. 10. Signature description of the nic driver (An example of send/receive)

send/receive specifier: TECS supports *send*/*receive* specifiers which are interface specifier [15]. TINET+TECS uses *send* and *receive* specifier instead of *in* and *out* to reduce the number of copies.

in is a specifie 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 specifie 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 in Fig. 9. In case of *in* specifie , 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.

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

receive is also a specifie 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 in Fig. 9. 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.

As shown in Fig. 10, the arguments of sending and receiving data such as *outputp* and *inputp* are define with the *send/receive* specifier in the signature description.

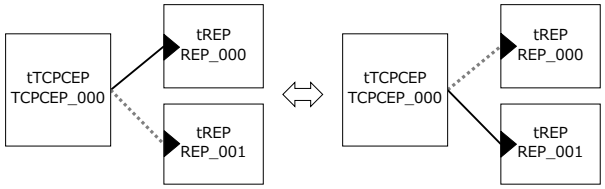

Fig. 11. Dynamic connection

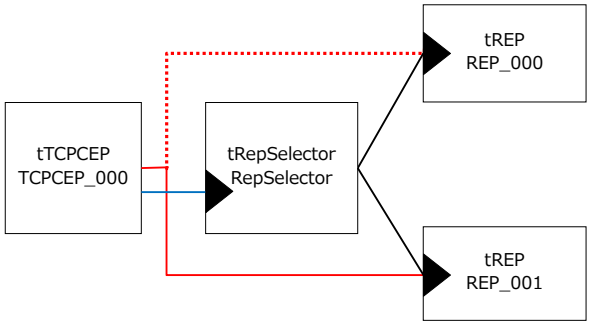

Fig. 12. Dynamic connection between CEP and REP

B. Dynamic Connection of TECS

TECS supports dynamic connection as a new functionality. Dynamic connection is a method to switch the binding of components at runtime as shown in Fig. 11. Note that all components are statically generated in TECS. TECS can optimize the overhead of componentization because components are statically configured Dynamically generating the components causes a lot of memory consumption, which is a serious problem for embedded systems with strict memory constraint. The proposed framework realizes the componentization of TINET while satisfying the memory constraint, because components statically generate and dynamically connect in TECS.

TINET+TECS utilizes the dynamic connection to switch between CEP and REP components as shown in Fig. 12. In a server application, CEP is associated with REP in the state waiting for connection request from clients[2]. For example, when processing with HTTP protocol, CEP passively opens with REP of port number 80.

To utilize dynamic connection, the selector should be defined The selector connects all the components that can be dynamically connnected, to refer the descriptor of them. Descriptor is an identifie to access the component [16]. cREP ports are call ports array, which connect the number of tREP cells (Line 9 in Fig. 13). *[ref_desc]* is described to identify the call port refers to descriptors. In the case of Fig. 12, the tRepSelector cell connects all tREP cells.

A CEP component has two call ports. cRepSelector port connects eRepSelector port of tRepSelector cell and cREP4

[2]ITRON TCP/IP API Specificatio [10]
tcp_acp_cep(ID cepid, ID repid, T_IPV4EP *p_dstaddr, TMO tmout)

```
1  signature sRepSelector {
2     void getRep([out]Descriptor(sREP4) *desc,
3                 [in]int_t i);
4  };
5
6  celltype tRepSelector {
7     entry sRepSelector eRepSelector;
8     [ref_desc]
9        call sREP4 cREP[NUM_REP];
10 };
11
12 celltype tTCPCEP {
13    call sRepSelector cRepSelector;
14    [dynamic]
15       call sREP4 cREP;
16    /* Omit: other call/entry ports */
17    /* Omit: attributes and variables */
18 };
```

Fig. 13. Signature and celltype description for dynamic connection

```
1  eAPI_accept (.., ..) {
2     /* Get a descriptor of intended REP cell */
3     cRepSelector_getRep(&desc, repid);
4     /* Set the descriptor */
5     cREP_set_descriptor(desc);
6     /* Call the function of intended REP cell */
7     cREP_getEndpoint();
8  }
```

Fig. 14. Accept function (a dynamic connection example)

port connects either of tREP cells (Lines 13-15 in Fig. 13). cREP port is define with *[dynamic]* to identify the call port dynamically switch the components. The call port with *[dynamic]* specifie is not optimized and allocated in RAM by a plug-in.

Fig. 14 shows a sample code of dynamic connection. The eAPI_accept function is the function wrapping *tcp_acp_cep* with TECS, which is utilized to be the state waiting for connection request. In the function, dynamic connection is performed as shown in Fig. 14. First, get the descriptor of REP to be joined (Line 3 in Fig. 14). The firs argument, *&desc*, is a variable to store the descriptor information, and the second argument, *repid*, is the index of tREP cells. Next, set the desctriptor (Line 5 in Fig. 14). cREP port combines the tREP cell that descriptor specify. Thus, the tCEP cell can call the function of tREP cell to be joined (Line 7 in Fig. 14).

C. TECS Adapter

TECS supports *Adapter* functionality which enables to call a function in TECS from existing C codes. An adapter is implemented between C codes and a TECS component, and links a C function to a TECS function shown in Fig. 15. In TINET+TECS, when the application calls an API such as *tcp_snd_dat*, the adapter component calls the function of tTCPCEP such as *eAPI_sendData*. Note that *tcp_snd_dat* is define with the name *eAPI_sendData* in TINET+TECS. The adapter wraps the APIs used in the existing applications into TECS functions. Therefore, software developers can utilize an existing TCP/IP application by using the adapter.

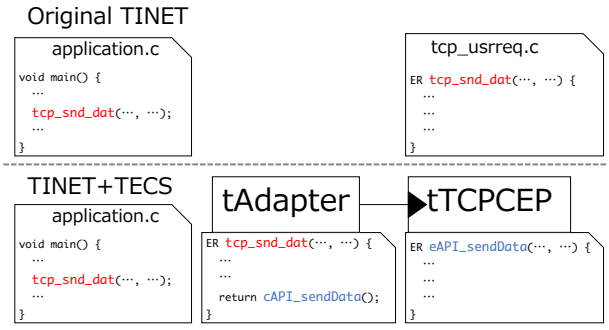

Fig. 15. TECS adapter

TABLE I
EVALUATION BOARD ENVIRONMENT

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

IV. EVALUATION

This section describes our experimental evaluation to demonstrate the effectiveness of the proposed framework.

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 specificatio of the board is shown in TABLE I. We also employ TINET 1.5.4 and the compiler arm-none-eabi–gcc 5.2

B. Performance of TINET+TECS

To demonstrate the low overhead of TINET+TECS, we evaluated the execution time and the memory consumption of TINET+TECS compared with TINET.

The comparison of execution time between TINET and TINET+TECS is shown in Fig. 16. The APIs used for the evaluation are *tcp_snd_dat* to send TCP data and *tcp_rcv_dat* to receive TCP data. For *tcp_snd_dat*, we measured the executing time from the API calling by the application to the return of the processing result. In TINET+TECS, the process is performed in the order of tApplication, tTCPCEP, tTCPOutputTaskBody, tIPv4Output, tEthernetOutput, tArp, tEthernetOutputTaskBody, and tIfMbed of Fig. 7. For *tcp_rcv_dat*, we measured the execution time from the data receiving in the LAN driver to the data acquisition in the application. In TINET+TECS, the process is performed in the order of tIfMbed, tEthernetInputTaskBody, tIPv4Input, tTCPInput, tTCPCEP, and tApplication of Fig. 7. The execution time of TINET+TECS is as well as that of TINET; the overhead is about 3 usec. *send/receive* specifier enable to access the buffer address without data copies. Therefore, the overhead of componentization does not effect the execution time.

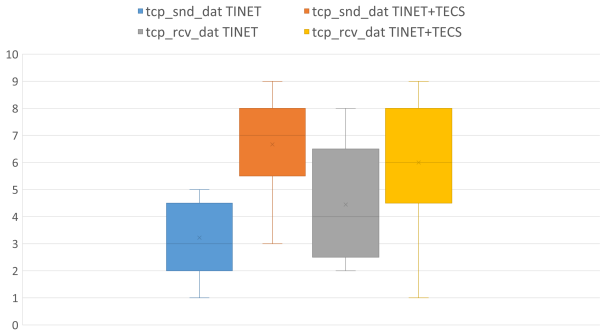

Fig. 16. Execution time of TINET and TINET+TECS

TABLE II
MEMORY CONSUMPTION OF TINET AND TINET+TECS

	TINET	TINET+TECS
ROM	74.93 KB	76.62 KB
RAM	27.36 KB	28.24 KB
ROM + RAM	102.29 KB	103.86 KB

TABLE III
MODIFIED CODE LINES OF CDL

	Size	Size (- Default)	CDL
Default	104.86 KB	0 KB	0
I	80.79 KB	- 24.07 KB	18
I + II	80.50 KB	- 26.35 KB	26
I + II + III	72.03 KB	- 32.83 KB	31

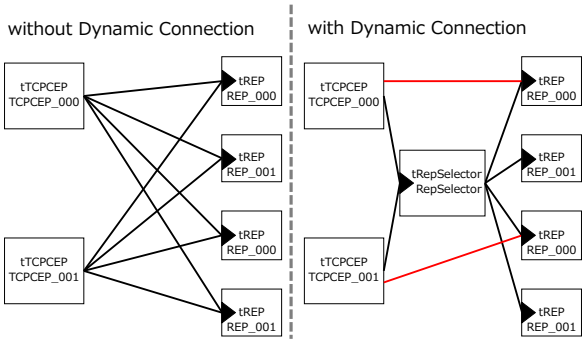

Fig. 17. Component diagrams of two cases (without/with dynamic connection)

The memory consumptions of TINET and TINET+TECS are compared in TABLE II. The memory consumption of TINET+TECS increases about 2.5% compared with TINET. The processes and data to manage TECS components, such as initialization of cells, descriptors, function tables, and skeleton functions, cause the increase memory consumption.

As shown in TABLE **??**, the code lines for modificatio were measured to demonstrate the improved configurabilit . We can change the composition of the protocol stack with a small workload. Thus, the proposed framework improves the configurabilit .

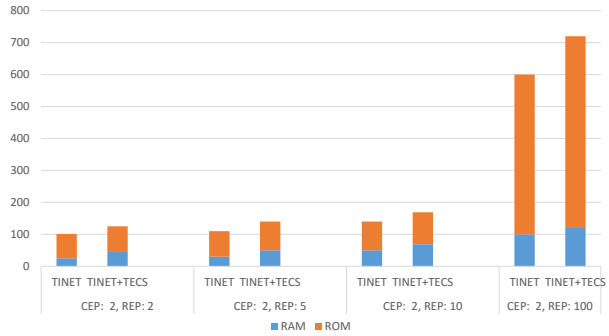

Fig. 18. Memory consumption of two cases (with/without dynamic connection)

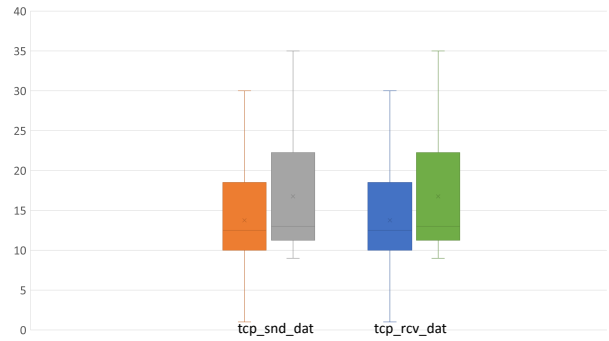

Fig. 19. Execution time of two cases (without/with TECS adapter)

C. Dynamic Connection

The memory consumption without and with TECS dynamic connection was evaluated. As shown in the left of Fig. 17, a CEP component should statically connect to all REP components in the case dynamic connection is not used. As the number of REPs increases, call ports of CEP are required for that. Thus, it consumes a lot of memory. Dynamic connection reduces the memory consumption because only one call port of CEP for REP, which is drawn with red lines in the right of Fig. 17, is required per CEP. Even when the number of REPs increases, a call port of the selector, not the CEP, is joined.

The memory consumption of two cases, without/with dynamic connection, is shown in Fig. 18. The case of dynamic connection consumes the RAM memory more than the other case because the call ports with *[dynamic]* is not optimized and allocated in RAM areas as mentioned in Section III-B. The total memory consumption is fewer in the proposed framework.

D. Overhead of Adapter

The execution time of API with the adapter such as *tcp_snd_dat/eAPI_sendData* to analyze the overhead of TECS adapter which supports existing applications. As shown in Fig. 19, the overhead of TECS adapter is very small because the adapter only passes the parameters to TECS components from C codes. Therefore, TECS adapter does not affect the system.

V. Related Work

Open-source TCP/IP protocol stacks for embedded systems have been developed such uIP [17], lwIP [18].

uIP: uIP (microIP) is a very small TCP/IP stack intended for tiny 8-bit and 16-bit microcontrollers. uIP only requires about f ve KB of code size and several hundreds 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 [19], [20], 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 KB of ROM and tens of KB of RAM. lwIP is larger than uIP, but provides better throughput.

VI. Conclusion

This paper proposed TINET+TECS, a component-based TCP/IP protocol stack for embedded systems. TINET+TECS is componentized TINET, which is a compact TCP/IP protocol stack, using TECS. TINET consists of many macros and complicated codes and the software productivity are low. The proposed framework improves the configurabilit while suppressing the overhead of componentization. The proposed framework also improves the scalability because the component-based framework simplifie to add/remove and change protocols such as TCP/UDP, IPv4/IPv6, and Ethernet/PPP.

In addition, this paper presents dynamic connection, a new method of TECS, to enable dynamic processing while reducing memory consumption. To satisfy TINET specificatio that TINET support static generation of CEPs and REPs, TINET+TECS utilizes dynamic connection. TECS adapter supports legacy codes, existing TCP/IP applications can run without modificatio in the proposed framework.

In the future, the proposed framework will cooperate with mruby on TECS [21] to easily manage IoT devices. Note that mruby is a scripting language for embedded systems [22]. We will support the functionalities that TINET functions can be utilized from mruby programs as an extension of mruby-socket [23].