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*—High productivity embedded network software is required to run embedded systems within the Internet of Things (IoT). Tomakomai InterNETworking (TINET) is a Transmission Control Protocol/Internet Protocol (TCP/IP) protocol stack for use in embedded systems. Although TINET is a compact protocol stack, it comprises many complex source codes and is difficult to maintain, extend, and analyze. To improve scalability and configurability, this paper proposes TINET componentized with the Toyohashi Open Platform for Embedded Real-time Systems (TOPPERS) embedded component system (TINET+TECS), a component-based TCP/IP protocol stack for embedded systems. This component-based TINET offers software developers high productivity through variable network buffer sizes and the ability to add or remove TCP (or UDP) functionality. TINET+TECS utilizes a dynamic TECS component connection method to satisfy the original TINET specifications. The results of an experimental comparison between the proposed component-based and original TINETs show that the execution time and memory consumption overhead are reduced and the configurability is improved.**

I. Introduction

The Internet of Things (IoT) is an essential next evolutionary step for the Internet [1] [2] in which various items and platforms, for example, wearable devices and smart devices, will be connected via the Internet to further enrich people's lives. However, as the IoT uses embedded systems such as data sensors and controlling actuators as elemental constituents, it is often not practical to implement the same Transmission Control Protocol/Internet Protocol (TCP/IP) protocol stacks used by traditional computing systems because embedded systems face restrictions in terms of low memory capacity.

Tomakomai InterNETworking (TINET) is a compact TCP/IP protocol stack for embedded systems [3]. As TINET supports functionalities such as a minimum copy frequency and the elimination of dynamic memory control, it requires significantly reduced memory for its TCP/IP protocol stack and is therefore suitable for embedded systems. However, TINET comprises many complex source codes, i.e., it contains many files and defines many macros, which can be problematic for software developers seeking to maintain, extend, and analyze the software. Thus, embedded network software is required for high productivity and quality.

One approach to improving software productivity is component-based development, a design technique that can be applied in reusable software development for embedded systems [4] [5] such as TECS [6] [7], AUTOSAR [8], or SaveCCM [9]. Component-based systems are flexible to software extension and specification changes.

This paper proposes a component-based TCP/IP protocol stack for embedded systems–TINET componentized with the Toyohashi Open Platform for Embedded Real-time Systems (TOPPERS) embedded component system (TINET+TECS)–to improve the configurability and scalability of TCP/IP software. Because it is a component system suitable for embedded systems, TECS is used to componentize TINET in the proposed protocol stack. As TECS supports static configurations that statically define component behaviors and interconnections, it can optimize the componentization overhead.

To satisfy the original TINET specifications, TINET+TECS utilizes the TECS dynamic connection method to dynamically switch component bindings. Although general TCP/IP protocol servers dynamically process requested ports, i.e., Hypertext Transfer Protocol (HTTP, port:80) and Hypertext Transfer Protocol Secure (HTTPS, port:443), embedded systems are restricted in their dynamic processing ability owing to strict memory constraints. TINET supports the static generation of Communication Endpoints (CEPs) and Reception Points (REPs), which are similar to sockets. As TINET+TECS, like TINET, statically generates components and dynamically combines them in the same manner, TINET+TECS reduces the dynamic increase of memory consumption.

In the proposed framework, software applications can be developed using both TECS method and existing methods. Software applications can be developed as TECS components because TINET+TECS is a component-based framework. Furthermore, TECS supports the use of an adapter to call TECS component functions from non-TECS codes, which allows for 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 changes needed to add and remove functionalities to improve configurability with small overheads. Furthermore, the advantages of dynamic connection in terms of the memory consumption and low overhead of the TECS adapter are demonstrated.

This paper provides the following contributions:

Improve configurability: Because TINET+TECS is a component-based system, its software can flexibly change as system's configuration by, for example, resizing network buffer, adding/removing TCP (or UDP) functionality, or supporting either IPv4 or IPv6. In addition, the use by

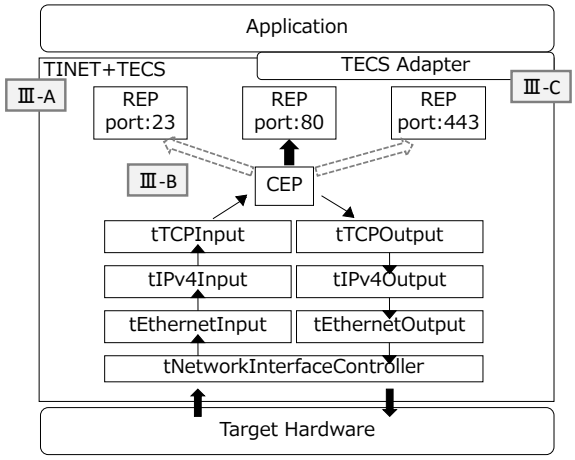

Fig. 1. System model of TINET+TECS

TINET+TECS of individual component diagrams enables visualization of an entire system.

Dynamic connection method: Dynamically switching the binding of components, that is, switching between a TINET communication endpoint and REP, realizes a TCP/IP protocol stack for an embedded system.

Support legacy codes: TINET+TECS can be applied to existing applications because TECS supports the ability of the adapter to call TECS functions from C codes.

The remainder of this paper is organized as follows. Section II introduces the system model and its 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 its 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 basic technologies such as TINET and TECS. A system model of the proposed framework is shown in Fig. 1. TINET+TECS is a component-based TCP/IP protocol stack in which the TCP output task (tTCPOutput) and Ethernet input task (tEhternetInput) are implemented as TECS components (Section III-A). CEPs and REPs (Section II-A), which are also implemented as TECS components, dynamically switch bindings using the TECS method (Section III-B). Moreover, the TECS adapter supports the legacy codes for existing TCP/IP applications (Section III-C).

A. TINET

TINET is a compact TCP/IP protocol stack for embedded systems based on the ITRON[1] TCP/IP API Specification [10], developed by the TOPPERS Project [11]. TINET has been released as an open-source tool.

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

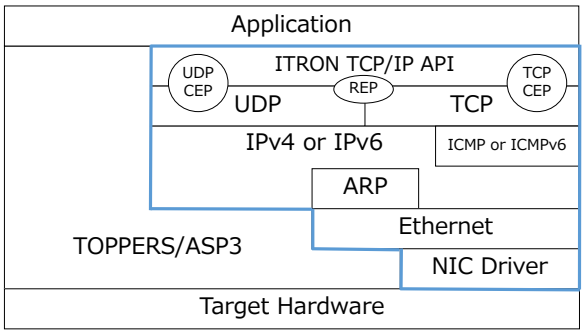

Fig. 2. TINET and TOPPERS/ASP3 hierarchy diagrams

To satisfy restrictions for embedded systems in terms of, for example, memory capacity, size, and power consumption, TINET supports the following functions:

- minimum copy frequency,
- elimination of dynamic memory control,
- asynchronous interfacing,
- error detailing per API.

1) Overview: TINET runs as middleware on TOPPERS/ASP3 [12] [13], a real-time kernel based on μITRON [14]. As it is compatible with TOPPERS RTOS, TINET also supports other RTOSs such as TOPPERS/ASP and TOPPERS/JSP.

Fig. 2 shows the hierarchy diagram of TINET and TOPPERS/ASP3. Users transmit and receive data using a Communication End Point (CEP), an interface that functions like a socket. In the transmission process, headers are attached to the data body passed to the CEP at each protocol layer before the data are transmitted from the network device. In the reception process, the headers of the data bodies received by the network device are analyzed at each protocol layer, and the data are then passed to the CEP.

A TCP reception point called the REP stands by to receive connection requests from the partner side. The REP has an IP address (*myaddr*) and a port number (*myportno*) as attributes and performs functions such *bind()* and *listen()*.

In TINET, the amount of data copying between each protocol layers is minimized. In standard computing systems, the TCP/IP protocol stack has large overheads in terms of execution time and memory consumption because the data are copied at each protocol layer. To solve this problem, TINET does passes the pointer of the data buffer between each protocol layer instead of performing data copying.

B. TECS

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

In TECS, component deployment and composition are performed statically, and as a consequently, component connection does not incur significant overhead, and memory requirements can be reduced. TECS has various features such

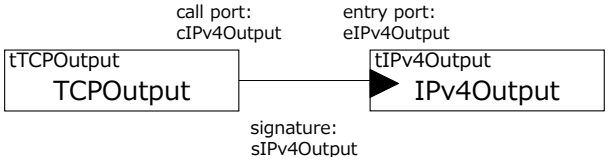

Fig. 3. Component diagram

```
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

```
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

as source level portability and fine-grained components and can be implemented in C.

1) Component model: Fig. 3 shows a component diagram. *Cells*, which are instance of components in TECS, consist of *entry* ports, *call* ports, attributes, and internal variables. An *entry* port is an interface that provides functions to other *cells*, whereas a *call* port is an interface that enables the use of other *cells*' functions. Each *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*, a set of functions that defines the interface definition of the *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 variables of a *cell*.

2) Component description: In TECS, components are described using 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"–for instance, sIPv4Output (Fig. 4). To clarify the function of an interface, specifiers such as [in], [out], and [inout] are used in TECS to represent the input, output, and input/output, respectively. Similarly, [size_is(*len*)] represents an array of size *len*.

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

Build Description: The build description is used to instantiate and bind *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*. In Fig. 6, *entry* port eIPv4Output in *cell* IPv4Output is connected to *call* port cIPv4Output in *cell* TCPOutput. *C_EXP* can be used to call macros defined in C files.

III. Design and Implementation

This section describes the design and implementation of the proposed TINET+TECS framework. The proposed framework is a component-based TCP/IP protocol stack for embedded systems: in other words, a componentized TINET using TECS. In this section, a new TECS functionality–the dynamic connection method–and the TECS adapter to support legacy codes are described via a use case of the proposed framework.

A. TINET+TECS

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

1) Components of a protocol stack: The components of a TINET+TECS protocol stack are shown in Fig. 7. Note that some small particle components, such as a kernel object, data queues, and semaphores, are omitted to simplify the component diagram. In TINET+TECS, the components are divided for each protocol, and functionalities such as input and output functions are defined as respective components. By using such small grain components, software visibility is improved. The components of each protocol are described in the following.

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, in TINET+TECS supporting a TECS adapter (**??**), an existing application with TINET can run on the TINET+TECS framework without transporting, and therefore, software can be developed either using existing methods or as TECS components.

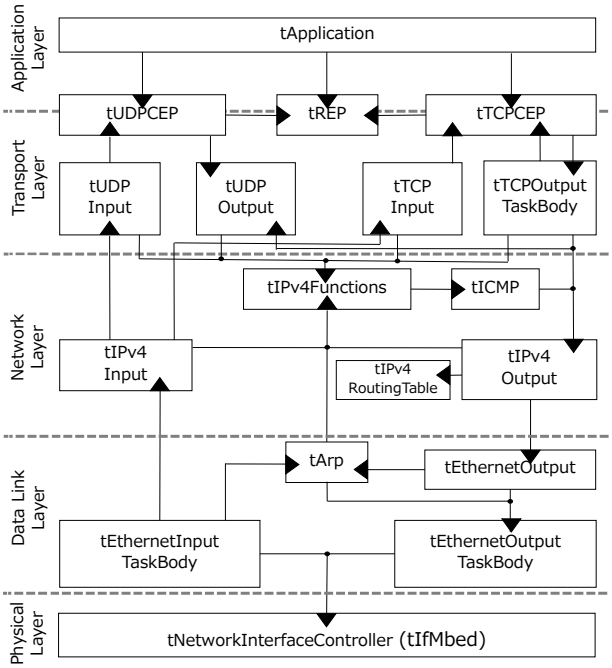

Fig. 7. Component diagram of a protocol stack

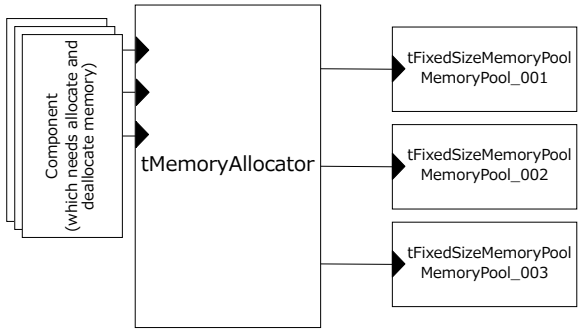

Fig. 8. Component diagram of tMemoryAllocator

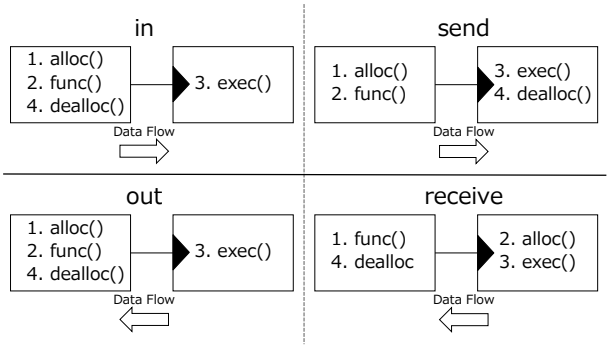

Fig. 9. Differences between in/out and send/receive

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

Network layer: The tIPv4Input and tIPv4Output components perform, respectively, the receiving and sending processing in the network layer. The tIPv4Functions component performs functions such as checksum, the tICMP component is used for the Internet Control Message Protocol (ICMP), and the tIPv4RoutingTable component operates a routing table.

Data link layer: tEthernetInputTaskBody and tEthernetOutputTaskBody (tEthernetOutput) are components for performing, respectively, receiving and sending processing in the data link layer. The tArp component is for implementing the Address Resolution Protocol (ARP).

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

To utilize the protocol stack in the same manner in 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 corresponds 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 the coexistence of multiple protocols. Though its use of IPv6 and Point-to-Point Protocol (PPP) components, TINET+TECS can make IPv4 and IPv6 coexist and support PPP without modification of component implementation.

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

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

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

Send/receive specifiers: TECS supports *send*/*receive* interface specifiers [15]. 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* when executing the

```
signature sNicDriver {
  void start(
    [send(sNetworkAlloc),size_is(size)]
       int8_t *outputp,
    [in]int32_t size,
    [in]uint8_t align);
  void read(
    [receive(sNetworkAlloc),size_is(*size)]
       int8_t **inputp,
    [out]int32_t *size,
    [in]uint8_t align);
    /* Omit: other functions */
};
```

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

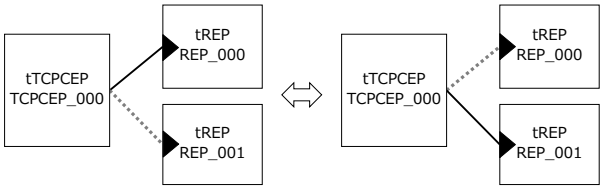

Fig. 11. Dynamic connection

callee function. When the processing returns to the caller side, the caller can reuse and deallocate the memory.

- *send* is another specifier for transferring data to a callee from a caller such as *in*. The difference between *in* and *send* is whether the data memory is deallocated in the caller or callee, as shown in Fig. 9. In the case of the *in* specifier, both allocating and deallocating of the data memory are performed in the caller. By contrast, in the case of *send*, the caller allocates the data memory and the callee deallocates it.
- *out* is a specifier for output arguments through which a callee writes data in the memory allocated by a caller while the caller receives the data.
- *receive* is another specifier for a caller receiving data from a callee such as *out*. The difference between *out* and *receive* lies in whether the data memory is allocated in the caller or callee, as shown in Fig. 9. In the case of *out*, the callee writes data in the memory allocated by a caller, whereas in the *receive* case, the callee allocates the data memory. Deallocating of the memory is performed in the caller in both cases.

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

B. Dynamic connection in TECS

TECS supports dynamic connection, a method for switching the binding of components at runtime (Fig. 11) as a new functionality. In Fig. 11, the solid line represents binding and the dotted line represents non-binding. Note that all components are statically generated in TECS, which can optimize the overhead of componentization because components are statically configured. Dynamically generating components

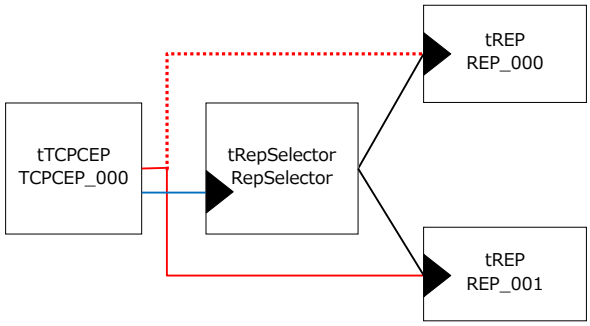

Fig. 12. Dynamic connection between CEP and REP

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

Fig. 13. Signature and celltype description for dynamic connection

causes a good deal of memory consumption, which is a serious problem for embedded systems with strict memory constraints. The proposed framework can take advantage of the componentization in TINET while satisfying the memory constraint because components are statically generated and dynamically connected in TECS.

TINET+TECS utilizes 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 of waiting for connection request from clients[2]. For example, when processing with the HTTP protocol, CEP passively opens with an REP of port number 80.

To utilize dynamic connectivity, a selector should be defined. A selector connects all components that can be dynamically connected under a common descriptor that serves as an identifier to access each component [16]. The cREP ports form a call port array connecting to connecting to all tREP cells (Line 8 in Fig. 13). *[ref_desc]* is used to identify call ports referring to descriptors. In the case shown in Fig. 12, the tRepSelector cell connects all tREP cells.

A CEP component has two call ports: the cRepSelector port, which connects to the eRepSelector port of tRepSelector cell, and the cREP4 port, which connects to either of the tREP cells (Lines 11–13 in Fig. 13). The cREP port is defined using *[dynamic]* to identify the call port used to dynamically switch the components. The call port with the *[dynamic]* specifier is not optimized and is allocated in RAM using a plug-in.

[2]tcp_acp_cep(ID cepid, ID repid, T_IPV4EP *p_dstaddr, TMO tmout).

```
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 (Dynamic connection example)

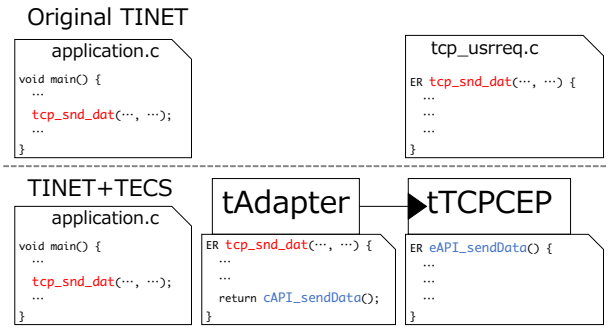

Fig. 15. TECS adapter

TABLE I
EVALUATION BOARD ENVIRONMENT

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

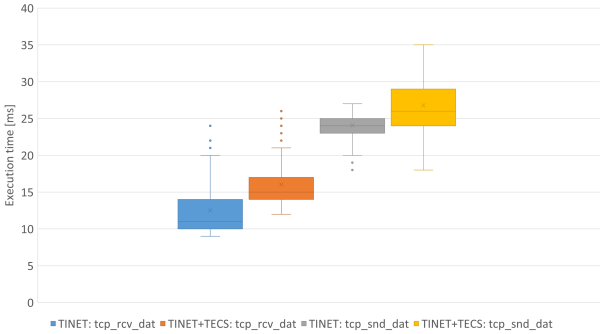

Fig. 16. Execution times of TINET and TINET+TECS

Fig. 14 shows a sample code of dynamic connection. The eAPI_accept function is the function wrapping *tcp_acp_cep* under TECS, which is set as the state waiting for connection request. Dynamic connection in the function is performed as shown in Fig. 14. First, the descriptor of REP to be joined is obtained (Line 3 in Fig. 14). The first argument, *&desc*, is a variable used to store the descriptor information, whereas the second argument, *repid*, is the index of tREP cells. Next, the descriptor is set (Line 5 in Fig. 14), and the cREP port combines the tREP cell specified by the descriptor, enabling the tCEP cell to call the function of the tREP cell to be joined (Line 7 in Fig. 14).

C. TECS adapter

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

IV. EVALUATION

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

A. Evaluation environment

GR-PEACH was employed as the evaluation board. Detailed specifications of the board are shown in TABLE I. We also employ TINET 1.5.4 and the compiler arm-none-eabi-gcc 5.2 To pretest the system, we connected the board to a host PC via a LAN cable and evaluated the data sending and receiving.

B. Performance of TINET+TECS

To demonstrate the low overhead of TINET+TECS, we compared its execution time and memory consumption with that of TINET, producing the results shown in Fig. 16.

The *tcp_snd_dat* and *tcp_rcv_dat* APIs were used in the evaluation to, respectively, send and receive TCP data. For *tcp_snd_dat*, we measured the executing time starting from the API call through the application until the return of the processing result. In TINET+TECS, this process is performed in the order tApplication, tTCPCEP, tTCPOutputTaskBody, tIPv4Output, tEthernetOutput, tArp, tEthernetOutputTaskBody, and tIfMbed, as shown in Fig. 7. For *tcp_rcv_dat*, we measured the execution time from the data receipt in the LAN driver until data acquisition in the application. In TINET+TECS, the process is performed in the order tIfMbed, tEthernetInputTaskBody, tIPv4Input, tTCPInput, tTCPCEP, and tApplication, as shown in Fig. 7. The execution time of TINET+TECS is close to that of TINET, with an overhead of about 3 ms. As the use of the *send/receive* specifier enables accessing of the buffer address without data copying, the componentization overhead does not affect the execution time.

The memory consumptions of TINET and TINET+TECS are compared in TABLE II. The memory consumption of TINET+TECS is about 1% higher than that of TINET, as the data and processes such as initialization of cells, descriptors, function tables, and skeleton functions needed to manage TECS components increase memory consumption.

As shown in TABLE III, the code lines for modification were measured to demonstrate the improved configurability. This demonstrated the ability to change the composition of the protocol stack with a small workload, confirming that the proposed framework improves the configurability.

TABLE II
MEMORY CONSUMPTION OF TINET AND TINET+TECS

	text	data	bss	total
TINET	183.94 KB	5.37 KB	132.03 KB	322.34 KB
TINET+TECS	170.73 KB	5.37 KB	149.13 KB	325.23 KB

Including the application and kernel objects

TABLE III
MODIFIED CODE LINES OF CDL

	Size	Size (– Default)	CDL
Default	325.23 KB	0 KB	0 lines
I	305.40 KB	– 19.83 KB	18 lines
I + II	304.12 KB	– 21.10 KB	27 lines
I + II + III	303.45 KB	– 21.77 KB	32 lines

I: Remove TCP
II: Remove ICMP
III: Change network buffer (Remove memory pools)

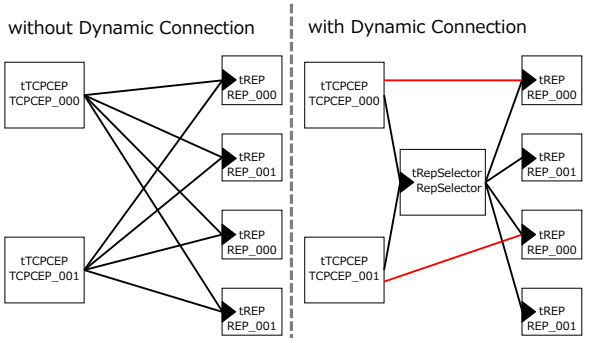

Fig. 17. Component diagrams for without/with dynamic connection cases

TABLE IV
MEMORY CONSUMPTION IN TWO CASES
(WITH/WITHOUT DYNAMIC CONNECTION)

	CEP:1 REP:1	CEP:1 REP:5	CEP:2 REP:5	CEP:5 REP:10
without	324.98 KB	325.34 KB	326.39 KB	331.68 KB
with	325.23 KB	325.32 KB	327.24 KB	330.48 KB

C. Dynamic connection

Memory consumption without and with TECS dynamic connection was then evaluated. As shown in the left panel of Fig. 17, each CEP component should be statically connected to all REP components if dynamic connection is not used. As the number of REPs increases, additional call ports of CEP are required, in turn increasing the consumption of memory. Dynamic connection reduces memory consumption because only one CEP-to-REP call port is required per CEP, as illustrated with red lines in the right panel of Fig. 17. Even if the number of REPs increases, additional call ports can be joined through the selector, instead of the CEPs.

Memory consumption of without and with dynamic connection is shown in TABLE IV. The dynamic connection case consumes the more RAM memory because, as mentioned in Section III-B, call ports with *[dynamic]* are not optimized and allocated in RAM areas. However, the overall memory consumption is lower under the proposed framework.

The code lines in CDL of without and with dynamic connection is shown in TABLE V to demonstrate improved configurability. As the number of CEPs and REPs increases, the amount of CDL code lines to be added increases. In the left panel of Fig. 17, each CEP connects all REPs as shown in the upper of Fig. 18. In the right panel of Fig. 17, a CEP dynamically connects an REP, and only the selector connects all REPs as shown in the lower of Fig. 18. It is effective for software that uses many ports because the difference spreads as the number of CEPs and REPs increases.

TABLE V
CDL CODE LINES OF WITHOUT/WITH DYNAMIC CONNECTION

	without	with	Diff
CEP:1 REP:1	344 lines	347 lines	-3 lines
CEP:1 REP:5	369 lines	367 lines	2 lines
CEP:2 REP:5	387 lines	382 lines	5 lines
CEP:5 REP:10	485 lines	445 lines	40 lines

```
/* without Dynamic Connection */
cell tTCPCEP TCPCEP_000 {
    cREP[0] = REP_000.eREP;
    ..
    cREP[n] = REP_00n.eREP;
};
cell tTCPCEP TCPCEP_001 {
    cREP[0] = REP_000.eREP;
    ..
    cREP[n] = REP_00n.eREP;
};
..
cell tTCPCEP TCPCEP_00n {
    cREP[0] = REP_000.eREP;
    ..
    cREP[n] = REP_00n.eREP;
};
```

```
/* with Dynamic Connection */
cell tRepSelector RepSelector {
    cREP[0] = REP_000.eREP;
    ..
    cREP[n] = REP_00n.eREP;
};
cell tTCPCEP TCPCEP_000 {
    cRepSelector = RepSelector.eRepSelector;
};
cell tTCPCEP TCPCEP_001 {
    cRepSelector = RepSelector.eRepSelector;
};
..
cell tTCPCEP TCPCEP_00n {
    cRepSelector = RepSelector.eRepSelector;
};
```

Fig. 18. Two CDL codes (without/with dynamic connection)

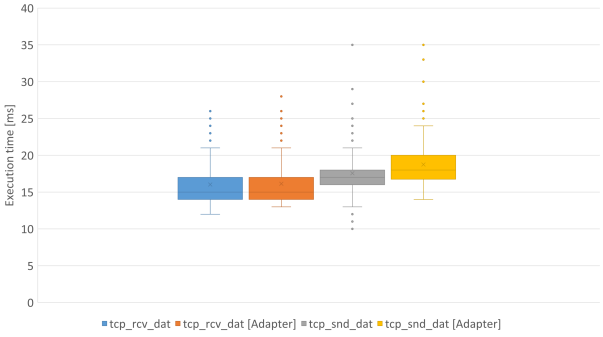

Fig. 19. Execution times in two cases (without/with TECS adapter)

D. Adapter overhead

The API execution time when using an adapter such as *tcp_snd_dat/eAPI_sendData* was used to analyze the overhead of the TECS adapter supporting existing applications. As shown in Fig. 19, the overhead of the TECS adapter is very small because the adapter only passes parameters from C codes to TECS components; thus, the TECS adapter overhead does not affect the system.

V. Related Work

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

uIP: uIP (microIP) is a very small TCP/IP stack intended for tiny 8- and 16-bit microcontrollers. uIP requires only about 5 KB 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. Following the release of ver. 1.0, later versions of uIP, including uIPv6, have been integrated with Contiki OS [19], [20], an operating system to connect tiny microcontrollers to the Internet.

lwIP: lwIP (lightweightIP) is a small TCP/IP implementation for embedded systems that is intended to reduce memory resource usage while still maintaining 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 a componentized version of TINET, a compact TCP/IP protocol stack that uses TECS. Because TINET comprises many macros and complicated codes, its software productivity is low. The proposed framework improves on TINET's configurability while suppressing the overhead of componentization. Scalability is also improved because the component-based framework simplifies to add/remove and change protocols such as TCP/UDP, IPv4/IPv6, and Ethernet/PPP.

This paper also presented dynamic connection, a new TECS method, to enable dynamic processing while reducing memory consumption. TINET+TECS utilizes dynamic connection to satisfy the TINET specification for supporting static generation of CEPs and REPs. As the TECS adapter supports legacy codes, existing TCP/IP applications can run without modification in the proposed framework. TINET+TECS and application programs used in the evaluation are all open-source, and may be downloaded from our website [21].

In future work, we will adapt the proposed framework to cooperate with mruby on TECS [22] to more easily manage IoT devices. Note that mruby is a scripting language for embedded systems [23]. We will support functionalities in which TINET functions can be utilized from mruby programs as an extension of mruby-socket [24].

Acknowledgment

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

References

[1] L. D. Xu, W. He, and S. Li, "Internet of Things in Industries: A Survey," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 4, pp. 2233–2243, Nov 2014.
[2] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context Aware Computing for The Internet of Things: A Survey," *IEEE Communications Surveys Tutorials*, vol. 16, no. 1, pp. 414–454, First 2014.
[3] TOPPERS, "TINET," https://www.toppers.jp/en/tinet.html.
[4] I. Crnkovic, "Component-based Software Engineering for Embedded Systems," in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005, pp. 712–713.
[5] 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.
[6] 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.
[7] T. Ishikawa, T. Azumi, H. Oyama, and H. Takada, "HR-TECS: Component technology for embedded systems with memory protection," in *Proceedings of the 16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2013, pp. 1–8.
[8] "AUTOSAR," http://www.autosar.org/.
[9] M. Åkerholm, 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.
[10] "ITRON TCP/IP API Specification (Ver. 2.00.00)," http://www.tron.org/wp-content/themes/dp-magjam/pdf/specifications/en_US/TEF024-S003-02.00.00_en.pdf.
[11] "TOPPERS Project," http://www.toppers.jp/en/index.html.
[12] T. Kawada, T. Azumi, H. Oyama, and H. Takada, "Componentizing an Operating System Feature Using a TECS Plugin," in *Proceedings of the 4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2016, pp. 95–99.
[13] TOPPERS, "TOPPERS/ASP3 kernel," https://www.toppers.jp/asp3-kernel.html.
[14] H. Takada and K. Sakamura, "μITRON for Small-Scale Embedded Systems," *IEEE Micro*, vol. 15, no. 6, pp. 46–54, 1995.
[15] 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 (SEA)*, 2008, pp. 204–209.
[16] T. Azumi, H. Takada, and H. Oyama, "Optimization of Component Connections for an Embedded Component System," in *Processings of the 7th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, vol. 2, Aug 2009, pp. 182–188.
[17] 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. ACM, 2003, pp. 85–98.
[18] ——, "Design and Implementation of the lwIP TCP/IP Stack," *Swedish Institute of Computer Science*, vol. 2, p. 77, 2001.
[19] 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.
[20] "Contiki OS," http://www.contiki-os.org/.
[21] "TINET+TECS," http://www.toppers.jp/tecs.html.
[22] T. Azumi, Y. Nagahara, H. Oyama, and N. Nishio, "mruby on TECS: Component-Based Framework for Running Script Program," in *Proceedings of the 18th IEEE International Symposium on Real-Time Computing (ISORC)*, 2015, pp. 252–259.
[23] K. Tanaka, A. D. Nagumanthri, and Y. Matsumoto, "mruby – Rapid Software Development for Embedded Systems," in *Proceedings of the 15th International Conference on Computational Science and Its Applications (ICCSA)*, 2015, pp. 27–32.
[24] "Contiki OS," http://mruby.org/libraries/.