Master's Thesis

Component-Based Software Development Framework

for Embedded IoT Devices

Supervisor

Professor Toshimitsu Ushio

Assistant Professor Takuya Azumi

By

Takuro Yamamoto

February 2, 2018

Division of Mathematical Science for Social Systems,
Department of Systems Innovation,
Graduate School of Engineering Science, Osaka University

Abstract

In embedded network software running on embedded systems within the Internet of Things (IoT), high levels of runtime efficiency and user productivity are required. To improve productivity, the mruby on TOPPERS embedded component system (TECS) framework, which employs a scripting language (i.e., lightweight Ruby) and supports component-based development, has been proposed. Based on the above information, this thesis proposes an extended mruby on TECS framework for its application in developing software for IoT devices, including sensors and actuators. The proposed framework enables mruby programs to utilize Tomakomai Internetworking (TINET), a TCP/IP protocol stack specifically designed for use in embedded systems. Further, this thesis proposes two component-based functionalities, i.e., a componentized TINET stack called TINET+TECS and a componentized Two-Level Segregate Fit (TLSF) dynamic memory allocator called TLSF+TECS. Here, TINET+TECS improves scalability and configurability and offers software developers high levels of productivity through variable network buffer sizes and the ability to add or remove TCP (or UDP) functionalities. TINET+TECS utilizes a dynamic TECS component connection method to satisfy the original TINET specifications. Further, TLSF+TECS is a thread-safe memory allocator that runs at high speeds and efficiently consumes memory. The experimental results of the comparison between TINET+TECS and the original TINET show that execution time and memory consumption overhead are both reduced; I conclude that configurability is improved. Finally, the TLSF+TECS functionality which obtains and report statistical information regarding mruby's virtual machine (VM) memory usage and frequency of memory requests, helps developers debug and verify their embedded IoT systems.

Contents

Abstract **i**

1 Introduction **1**

2 System Model **4**
2.1 TECS . . . 4
2.1.1 Component Model . . . 4
2.1.2 Component Description . . . 5
2.1.3 Development Flow . . . 7
2.2 mruby . . . 8
2.3 mruby on TECS . . . 8
2.3.1 System Model of mruby on TECS . . . 9
2.3.2 mruby-TECS Bridge . . . 9

3 Proposed Framework **10**
3.1 TINET+TECS . . . 10
3.1.1 TINET . . . 10
3.1.2 Component Design of TINET+TECS . . . 11
3.1.3 Dynamic connection in TECS . . . 15
3.1.4 TECS adapter . . . 17
3.2 TLSF+TECS . . . 18
3.2.1 TLSF . . . 18
3.2.2 TLSF Algorithm . . . 18
3.2.3 Component Design of TLSF+TECS . . . 19
3.3 Use case . . . 24

4 Evaluation **25**
4.1 Performance of TINET+TECS . . . 25
4.2 Dynamic connection . . . 26
4.3 Adapter overhead . . . 28
4.4 Memory statistics using TLSF+TECS . . . 29

5 Related Work **32**

5.1 Scripting languages for embedded systems 32
5.2 TCP/IP protocol stacks for embedded systems 33

6 Conclusions **35**

Acknowledgment **36**

References **37**

Publication List **40**

Chapter 1

Introduction

The Internet of Things (IoT) is an essential next evolutionary step for the Internet [1] in which various items and platforms, such as wearable devices and smart devices, will be connected via the Internet to further enrich people's lives. The IoT uses embedded systems such as data sensors and controlling actuators, as elemental constituents; therefore, these devices must demonstrate consistently high levels of quality and runtime performance. These requirements have consequently led to an increase in their complexity and scale; further, to be successful and widely adopted by users, these systems must have low production costs and short development cycles.

Complex and large-scale software systems can be developed efficiently using component-based techniques [2], [3]. Particularly, component-based development is a design technique that is applicable to the development of reusable software. Given its importance in terms of reliability, verification of component-based systems has been extensively researched [4], [5]. Individual component diagrams enable the visualization of an entire system. Further, component-based systems are flexible in terms of their extensibility and resilience to specification changes. Typical component-based development environments focused on embedded systems include the TOPPERS embedded component system (TECS) [6], AUTOSAR [7], and SaveCCM [8].

In addition to specific development environments, scripting languages, such as Ruby, JavaScript, Perl, Python, and Lua, offer efficient approaches to develop quality software. Currently, most embedded software is implemented using the C language. However, development in C typically results in large code size, high costs, and significant development time. In contrast, the use of scripting languages improves the efficiency of software engineering and can shorten the development period because it effectively supports and promotes the development of reusable scripts.

Embedded systems often have real-time requirements; therefore, real-time properties, such as estimating worst-case execution times, are very important. Although scripting languages are easy to use and read, their execution typically requires more time than that required for executing the code written in C. Therefore, applying scripting languages to embedded systems poses a major difficulty. To address this limitation, in [9], [10], mruby on TECS has been proposed as a component-based framework for efficiently running script-based programs. This framework integrates two key technologies, i.e., mruby, which is a lightweight implementation of Ruby designed specifically for embedded systems [11], [12] and TECS, which is a component-based framework also designed specifically for embedded systems [6].

This thesis proposes an extended framework for mruby on TECS that can be applied to embedded network software development involving IoT devices. The proposed framework makes it possible to utilize Tomakomai Internetworking (TINET) functions from within the mruby programs. Note that TINET is a compact TCP/IP protocol stack for embedded systems [13] that comprises a number of complex source code files, 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. To overcome the above problem, TINET+TECS is a componentized TINET implementation that incorporates TECS to improve the configurability and scalability of TCP/IP software. For IoT applications that should only be sending values obtained by one or more than one sensors, it is ideal to easily customize the minimum configuration of the TCP/IP protocol stack, e.g., by removing unused functions. This improved configurability also leads to satisfying strict memory constraints of IoT devices. In addition to TINET+TECS, this thesis proposes a component-based dynamic memory allocator based on Two-Level Segregate Fit (TLSF) called TLSF+TECS. Note that TLSF is a dynamic memory allocator designed specifically for real-time systems that always runs in constant time (i.e., $O(1)$) and improves memory usage efficiency by dividing memory blocks in two distinct stages. In the current version of TLSF, memory contention may occur when multiple threads run simultaneously. To address this problem, TLSF+TECS is a componentized TLSF memory allocator that is also entirely thread-safe because each component has its own heap area from which memory is actually allocated. In TLSF+TECS, developers can also obtain statistical information regarding memory usage and frequency of memory requests, which is crucial in analyzing memory operations and locating bugs.

Contributions: The proposed framework provides the following three key contributions.

Applicability to various devices. The proposed framework does not depend on real-time operating systems (RTOSs) or specific hardware; instead, it can be utilized by various devices, i.e., mruby code is portable. Therefore, it is possible to run the same program on different devices.

Improved configurability. Because TINET+TECS is a component-based system, its software can flexibly change in response to system configuration changes, such as resizing of network buffers, adding or removing TCP (or UDP) functionality, and supporting both IPv4 or IPv6. Further, the use of individual component diagrams enables visualizations of the entire system.

Thread-safe memory allocation TLSF+TECS safely runs multiple threads without exclusive control even if the threads operate in parallel. To achieve this, each thread can easily set up its own heap area. Further, statistical information is available to help developers debug and verify the specific memory usage of the given system.

Organization: In addition to this introductory section, the remainder of this thesis is organized as follows. Chapter 2 introduces the system model and the fundamental technologies, i.e., TECS, mruby, and mruby on TECS. Next, Chapter 3 describes the design and implementation of the proposed framework, including TINET+TECS and TLSF+TECS. Chapter 4 provides a detailed evaluation of the proposed framework. Finally, related work is discussed in Chapter 5, and Chapter 6 concludes this thesis.

Chapter 2

System Model

This chapter describes the system model of the proposed framework, including basic technologies such as TECS and mruby. The proposed framework is an extension of mruby on TECS framework [9] [10], and utilizes two technologies: mruby and TECS. The system model of the proposed framework is shown in Fig. 2.1. In the proposed framework, each mruby program runs on a RiteVM mapped to a componentized task of an RTOS. mruby programs can call the TINET functions required for network programming through the mruby-TECS bridge, and thus software to be embedded in IoT devices can be developed. Moreover, the proposed framework comprises TLSF+TECS. TLSF+TECS is utilized for memory management of mruby's RiteVMs and TINET+TECS, which helps to improve the efficiency of memory consumption. The following section explains TECS, mruby, and mruby on TECS framework.

2.1 TECS

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

TECS statically performs component deployment and composition. 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.

2.1.1 Component Model

Fig. 2.2 shows a component diagram. A *cell*, which is an instance of a TECS component, consists of *entry* ports, *call* ports, attributes, and variables. An *entry* port is an interface that provides functions to other *cells*, and a *call* port is an interface that enables the use

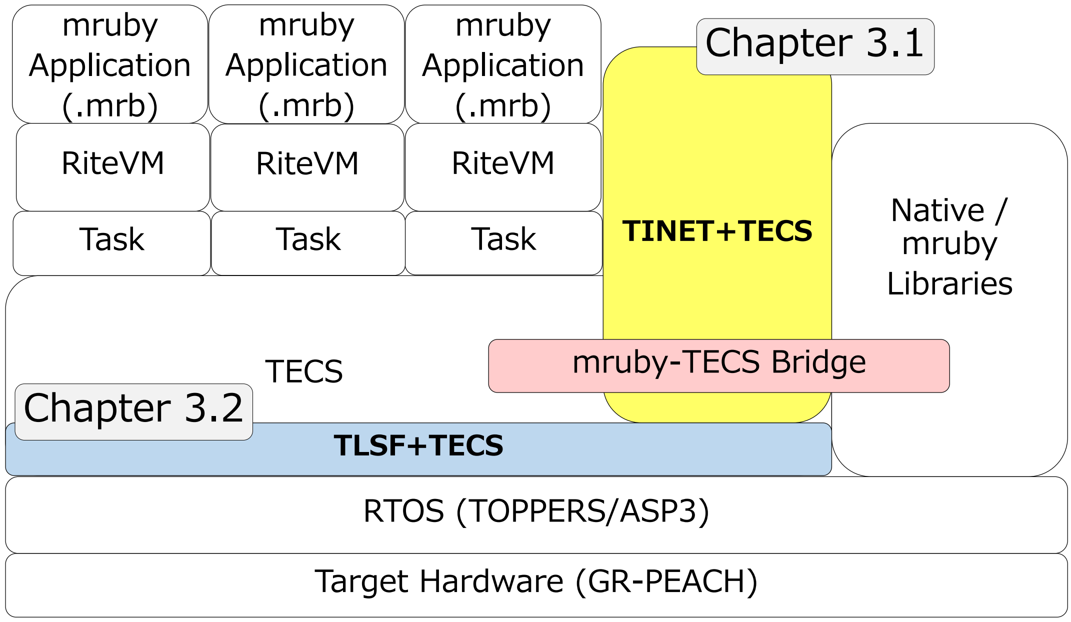

Fig. 2.1 System model of the proposed framework

Fig. 2.2 Component Diagram

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*. Here, *celltype* defines one or more *call*/*entry* ports, attributes, and internal variables of a *cell*.

2.1.2 Component Description

In TECS, components are described by *signature*, *celltype*, and build written in component description language (CDL). 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. 2.3). In TECS, to clarify the function of an interface, specifiers such as [in] and [out] are

```
signature sMotor {
    void initializePort( [in]int32_t type );
    ER setPower( [in]int power );
    ER stop( [in]bool_t brake );
};
```

Fig. 2.3 Signature Description

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

Fig. 2.4 Celltype Description

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

Fig. 2.5 Build Description

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. 2.4). 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.

Build Description The build description is used to instantiate and connect *cell*s. Fig. 2.5 shows an example of a build description. A *celltype* name and *cell* name, e.g., tMotor and Motor, respectively, follow the keyword *cell*. To compose *cell*s, a *call* port, *cell*'s name, and an *entry* port are described in that order. In Fig. 2.5, *entry* port eMotor in *cell* Motor is connected to *call* port cMotor in *cell* Caller. *C_EXP* calls macros defined in C files.

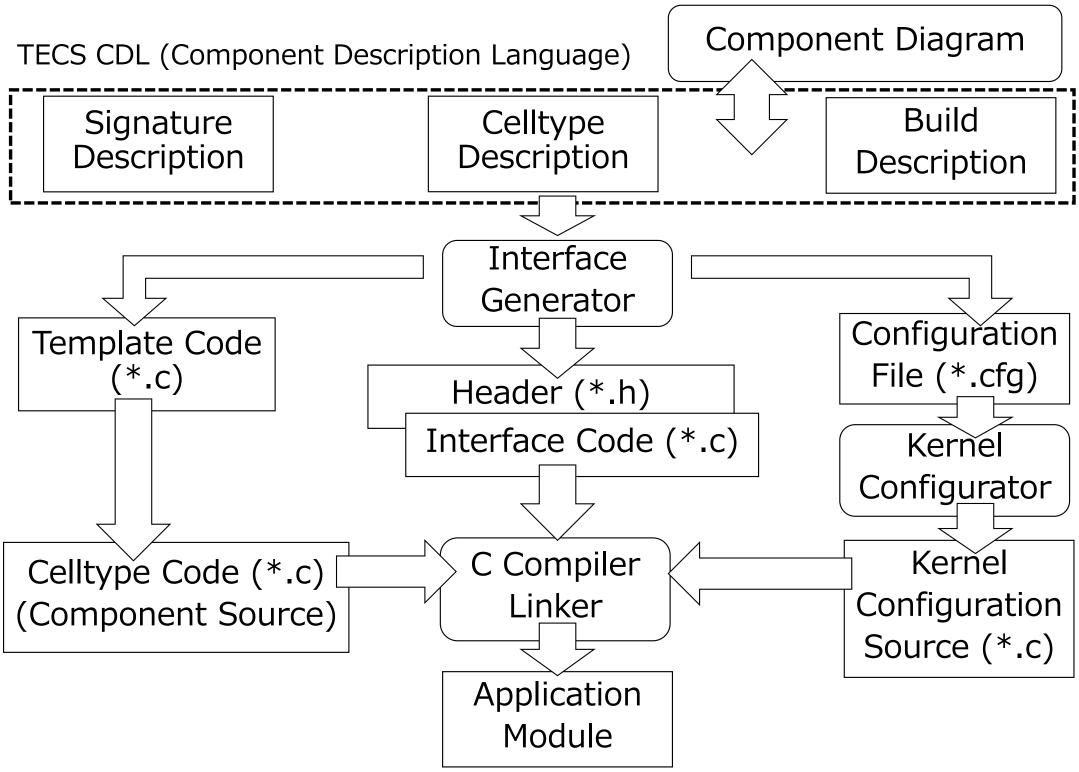

Fig. 2.6 Development flow using TECS

2.1.3 Development Flow

Fig. 2.6 shows the development flow using TECS. TECS generator generates the interface code (.H and .C) and the configure file of the RTOS (.cfg) from the CDL file.

Software developers using TECS can be divided into component designers and application developers. Component designers define *signature*s, which are interfaces between *cell*s, and *celltype*s, which are types of *cell*s. Using the template code generated from the CDL file in which these are defined, component designers implement the functions and behaviors of the component in C language. The source code implementing the function of the component is called a *celltype* code. Application developers develop applications by using component diagrams and predefined *celltype* to connect *cell*s with build description. An application module is generated by compiling and linking the header, the interface code, and the *celltype* code.

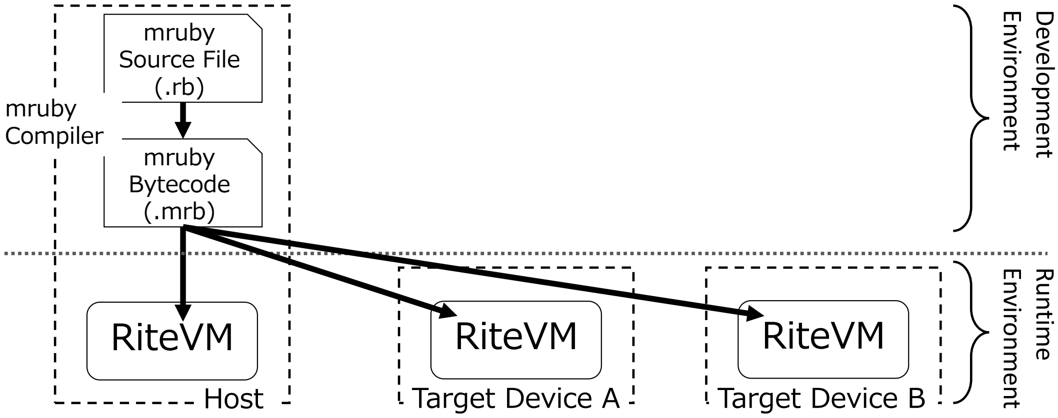

Fig. 2.7 mruby/RiteVM mechanism

2.2 mruby

mruby is a light-weight implementation of the Ruby programming language complying to part of the ISO standard. Ruby is an object-oriented scripting language [14] with classes and methods, exceptions, and garbage collection functions. It is easy to use and read due to its simple grammar and Ruby requires fewer lines of code than C. Ruby improves the productivity of software development due to its simple grammar and object-oriented functions.

mruby, which retains the usability and readability of Ruby, requires fewer resources, and thus, is suitable for embedded systems. In addition, mruby includes a VM mechanism, and thus, mruby programs can run on any operating system as long as a VM is implemented. The mruby/RiteVM mechanism is shown in Fig. 2.7. The mruby compiler translates an mruby code into a bytecode, which can be interpreted by a RiteVM; thus, mruby programs can be executed on any target device with a RiteVM.

2.3 mruby on TECS

mruby on TECS is a component-based framework for running an mruby script language on embedded systems. This framework integrates two technologies, mruby and TECS, and enables to develop embedded software using a script language without slowing down the execution time.

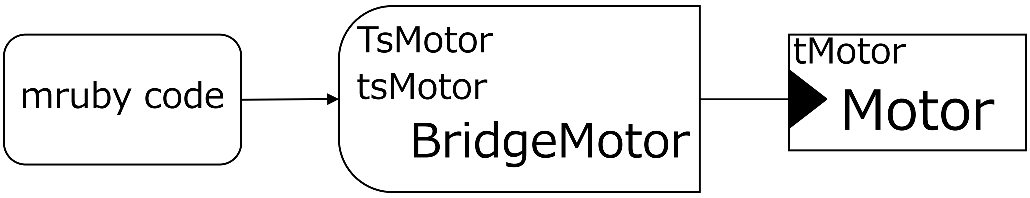

Fig. 2.8 mruby-TECS bridge

2.3.1 System Model of mruby on TECS

Each mruby program, which is a bytecode, runs on own RiteVM as a componentized task of an RTOS. TECS components support various embedded drivers such as motor and sensor drivers. An mruby-TECS bridge provides native libraries for mruby and can call a native program (e.g., C legacy code) from an mruby program. The mruby-TECS bridge also provides TECS components for receiving the invocation from an mruby program.

In this thesis, TOPPERS/ASP3 [15], [16] is the target RTOS and is based on μITRON [17]. However, mruby on TECS does not depend on the RTOS because TECS supports not only TOPPERS/ASP3 but also the other RTOSs such as OSEK [18] and TOPPERS/HRP2 [19], [20].

2.3.2 mruby-TECS Bridge

There is a significant difference between the execution times of mruby and C language codes. According to [9], mruby programs are several hundred times slower than C programs and the execution of an mruby bytecode on a RiteVM is not as efficient as that of C code. Thus, it is difficult to use mruby exclusively.

Using Ruby on embedded devices improves productivity and maintainability because it is easy to use and read. However, some C language codes are required to manipulate actuators and sensors and ensure that critical sections of the code run quickly.

Fig. 2.8 illustrates an mruby-TECS bridge used to control a motor. The left side of BridgeMotor belongs to the mruby program. The right side of BridgeMotor belongs to TECS component. The mruby-TECS bridge generates a *celltype*, which is called from the mruby code, and an mruby class, which corresponds to a developer-specified TECS component to invoke a C function from the mruby program. The generated mruby-TECS bridge supports registration of classes and methods for mruby. Methods in an mruby class are defined by generation codes for an mruby-TECS bridge, such as setPower and stop. Thus, when a method is called in an mruby program, the mruby-TECS bridge calls the function defined in the TECS component such as a Motor *cell*.

Chapter 3

Proposed Framework

The proposed framework is an extended mruby on TECS framework for network programming. It can use TINET+TECS functions from mruby programs. TINET+TECS is a component-based TCP/IP protocol stack comprised in the proposed framework, and it compensates for the original TINET's weak point, that it is hard to maintain, extend, and analyze the software due to many complex source codes and improves the configurability. In addition, TLSF+TECS, a component-based dynamic memory allocator, is used for memory management of mruby's RiteVMs and TCP/IP buffers in the proposed framework. Since each component holds own heap area, TLSF+TECS allows concurrent operation without exclusive control while improving the efficiency of memory consumption which is a merit of TLSF.

3.1 TINET+TECS

3.1.1 TINET

TINET is a compact TCP/IP protocol stack for embedded systems based on the ITRON*1 TCP/IP API Specification [21], developed by the TOPPERS Project [22]. TINET has been released as an open-source tool. To satisfy restrictions for embedded systems in terms of, for example, memory capacity, size, and power consumption, TINET supports the functions such as minimum copy frequency, elimination of dynamic memory control, asynchronous interfacing, error detailing per API.

Overview: TINET runs as middleware on TOPPERS/ASP3 [15] [16], a real-time kernel based on μITRON [17]. As it is compatible with TOPPERS RTOS, TINET also supports other RTOSs such as TOPPERS/ASP and TOPPERS/JSP.

Fig. 3.1 shows the hierarchy diagram of TINET and TOPPERS/ASP3. Users send and receive data using a Communication End Point (CEP), an interface that functions like a

*1 ITRON is an RTOS developed by the TRON project.

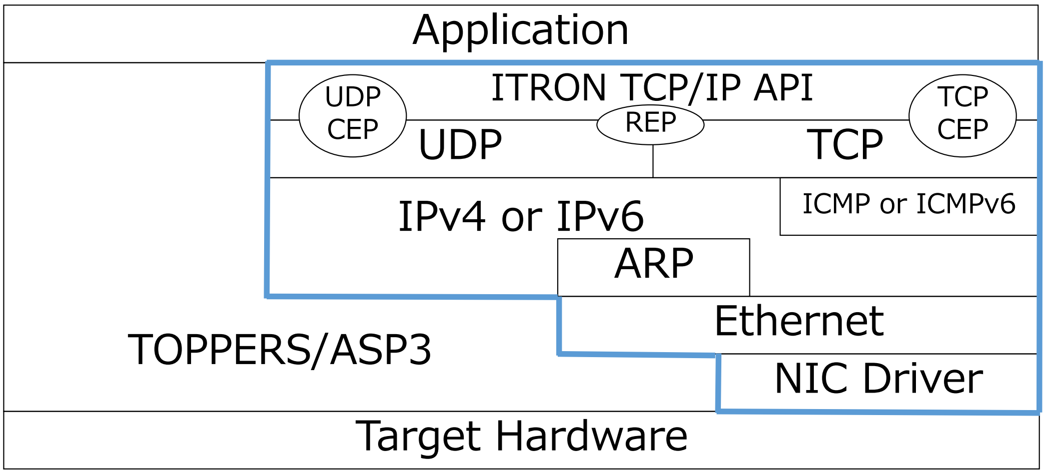

Fig. 3.1 TINET and TOPPERS/ASP3 hierarchy diagrams

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 at each protocol layer 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 pass the pointer of the data buffer between each protocol layer instead of data copying.

3.1.2 Component Design of 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.

Components of a protocol stack

The components of the TINET+TECS protocol stack are shown in Fig. 3.2. 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/output functions are defined

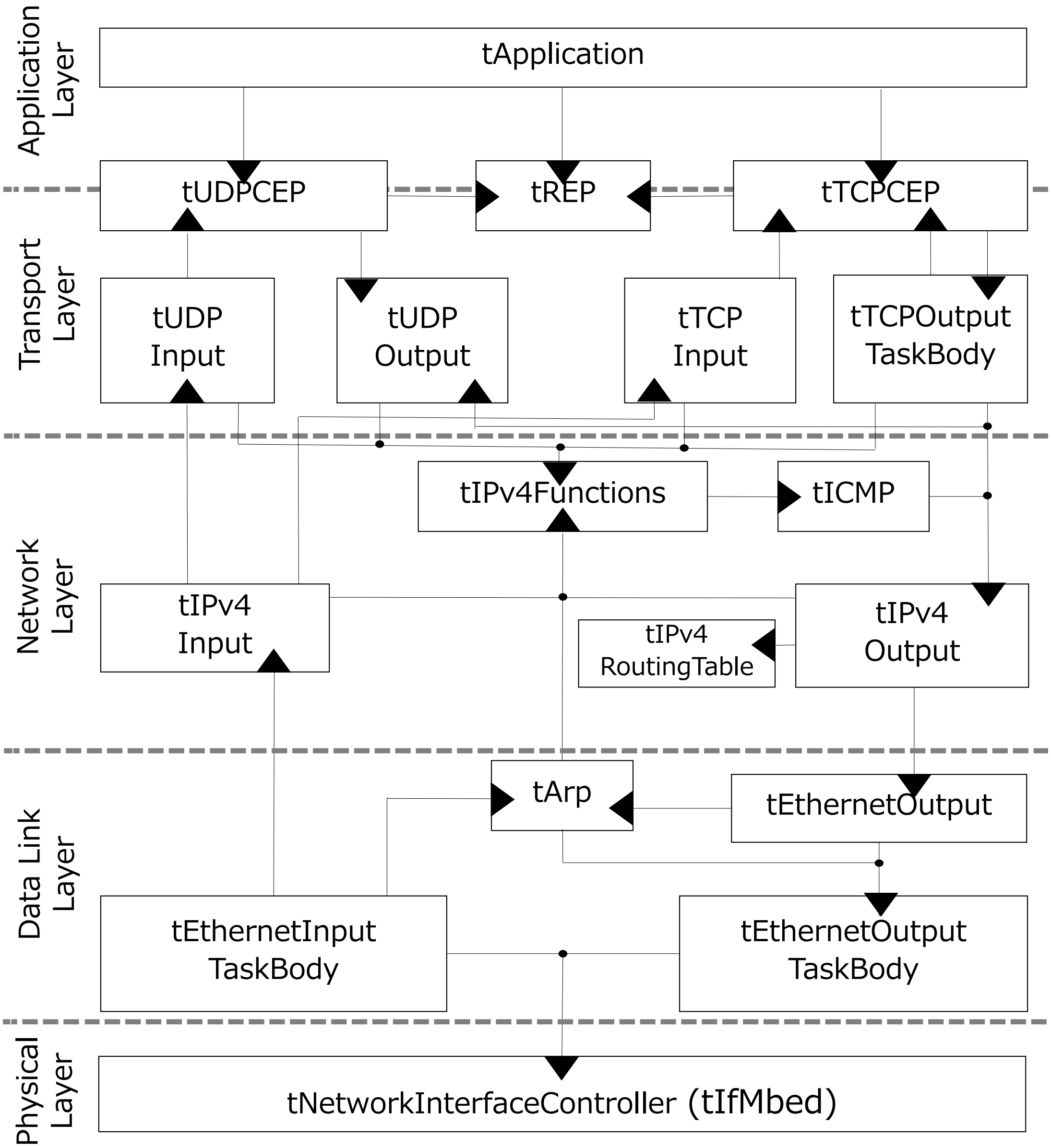

Fig. 3.2 Component diagram of a protocol stack

as respective components. By using such small grain components, software visibility is improved. The components of each protocol are described as follows.

Application layer: An application in TINET+TECS is implemented as a component such as tApplication. Software with TINET uses ITRON TCP/IP API [21] 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, the software can be developed either using existing methods or as TECS components.

Transport layer: tTCPCEP (tUDPCEP) and tREP are, respectively, CEP and REP components. For example, a server program supporting multiple clients can be developed by preparing multiple tTCPCEP components. tTCPInput and tTCPOutput 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 a 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. The 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 applications. The communication object component corresponds to a CEP or REP of the original TINET. Application developers can utilize TINET+TECS functionalities by generating and combining as many components as necessary.

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.

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. 3.3. tFixedSizeMemoryPool is a componentized kernel object of TOPPERS/ASP3 for allocating and releasing 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 compo-

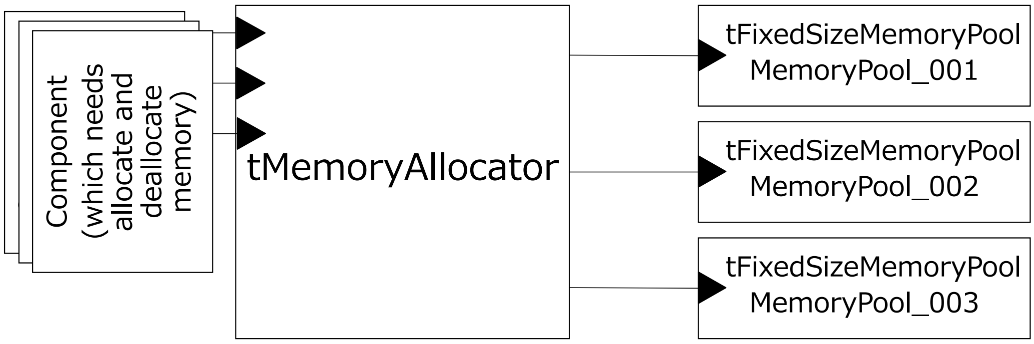

Fig. 3.3 Component diagram of tMemoryAllocator

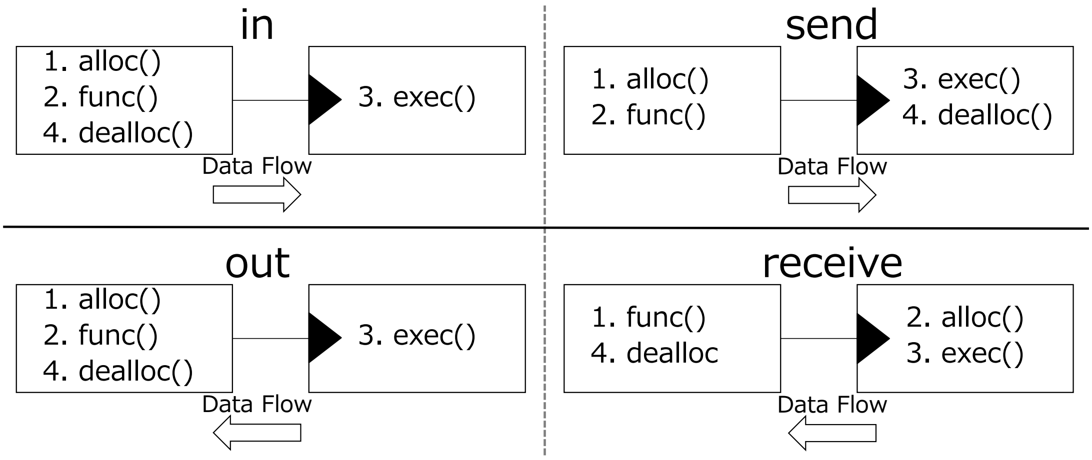

Fig. 3.4 Differences between in/out and send/receive

nents 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 [23]. 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 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. The difference between *in* and *send* is whether the data memory is deallocated in the caller or callee, as shown in Fig. 3.4. In the case of the *in* specifier, both allocating and deallocating of the data memory are performed in the caller. By contrast, in the

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

Fig. 3.5 Signature description of the nic driver (An example of send/receive)

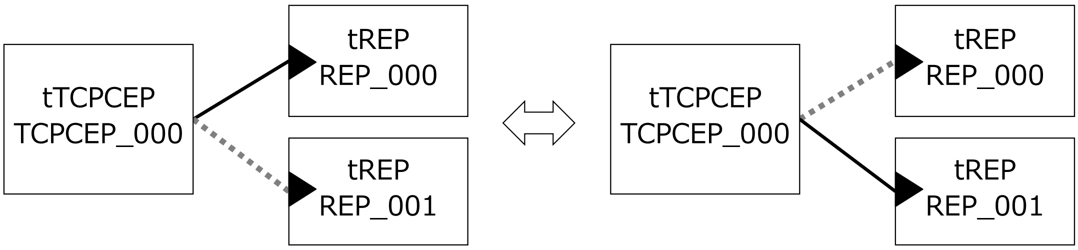

Fig. 3.6 Dynamic connection

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. The difference between *out* and *receive* lies in whether the data memory is allocated in the caller or callee, as shown in Fig. 3.4. 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. 3.5, sending and receiving arguments such as *outputp* and *inputp* are defined using, respectively, the *send/receive* specifier in the signature description. Developers hardly make mistakes of memory operation because these specifiers completely pass an ownership of memory. Common object request broker architecture (CORBA) does not consider memory sharing; CORBA has no functionalities such as *send/receive*.

3.1.3 Dynamic connection in TECS

TECS supports a dynamic connection, a method for switching the binding of components at runtime (Fig. 3.6) as a new functionality. In Fig. 3.6, 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 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 componentiza-

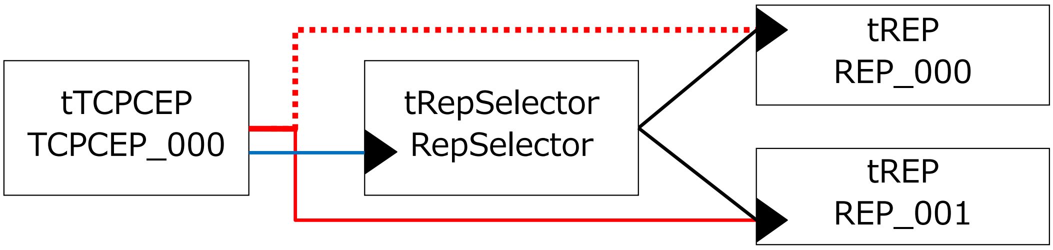

Fig. 3.7 Dynamic connection between CEP and REP

tion in TINET while satisfying the memory constraint because components are statically generated and dynamically connected in TECS.

TINET+TECS utilizes the dynamic connection to switch between CEP and REP components, as shown in Fig. 3.7. In a server application, CEP is associated with REP in the state of waiting for a 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 [24]. The cREP ports form a call port array connecting to connecting to all tREP cells (Line 8 in Fig. 3.8). *[ref_desc]* is used to identify call ports referring to descriptors. In the case of Fig. 3.7, 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. 3.8). 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.

Fig. 3.9 shows a sample code of the dynamic connection. The eAPI_accept function is the function wrapping *tcp_acp_cep* under TECS, which is set as the state waiting for a connection request. The dynamic connection in the function is performed as shown in Fig. 3.9. First, the descriptor of REP to be joined is obtained (Line 3 in Fig. 3.9). 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. 3.9), 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. 3.9).

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

```
signature sRepSelector {
   void getRep( [out]Descriptor(sREP4) *desc, [in]int_t i );
};
celltype tRepSelector {
   entry sRepSelector eRepSelector;
   [ref_desc]
      call sREP4 cREP[NUM_REP];
};
celltype tTCPCEP {
   call sRepSelector cRepSelector;
   [dynamic]
      call sREP4 cREP;
   /* Omit: other call/entry ports */
   /* Omit: attributes and variables */
};
```

Fig. 3.8 Signature and celltype description for the dynamic connection

```
eAPI_accept (.., ..) {
   /* Get a descriptor of intended REP cell */
   cRepSelector_getRep( &desc, repid );
   /* Set the descriptor */
   cREP_set_descriptor( desc );
   /* Call the function of intended REP cell */
   cREP_getEndpoint();
}
```

Fig. 3.9 Accept function (a dynamic connection example)

3.1.4 TECS adapter

TINET+TECS supports legacy codes because TECS supports *Adapter* functionality, which enables to call functions 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. 3.10. 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.

Since existing communication programs are supported, a communication task can be applied without porting, and other tasks can be developed with mruby programs.

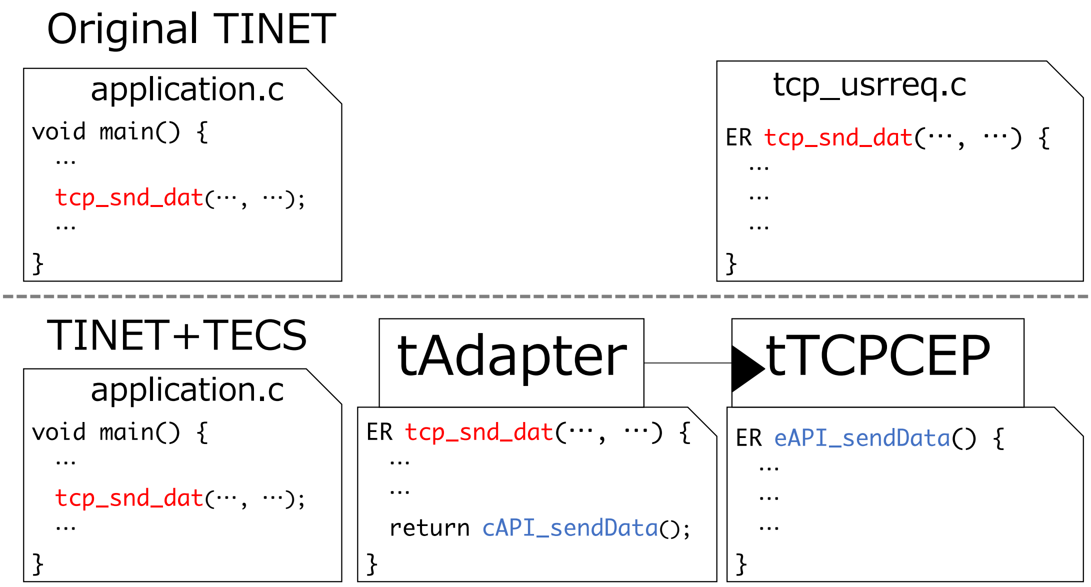

Fig. 3.10 TECS adapter

3.2 TLSF+TECS

3.2.1 TLSF

The TLSF memory allocator [25] [26] is a dynamic memory allocator that is suitable for use in the real-time systems. As such, the TLSF memory allocator provides the following two features.

Real-time property. In TLSF, the worst-case execution time required for allocating and deallocating memory does not depend on the given data size. Instead, TLSF always runs in constant time (i.e., $O(1)$), and it is possible to estimate response time.

Efficient memory consumption. Memory efficiency is improved by suppressing memory fragmentation. Various experiments have achieved an average fragmentation of less than 15 % and a maximum fragmentation of less than 25 %.

3.2.2 TLSF Algorithm

As illustrated in Fig. 3.11, the TLSF algorithm classifies memory blocks into two stages and searches for a memory block that optimally lines up with the requested memory size. Particularly, consider the case in which a request to dynamically allocate 98 bytes is made via *malloc*(). First, the request is classified based on the leftmost 1 bit in the requested memory size. In this example, since 98 is represented in binary as byte 01100010, it falls

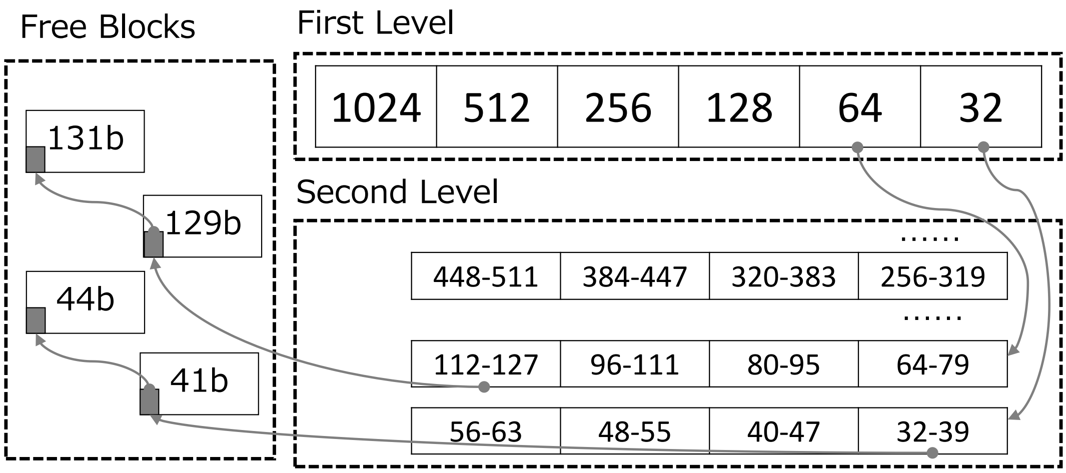

Fig. 3.11 TLSF Algorithm

```
signature sMalloc {
    int initializeMemoryPool( void );
    void *calloc( [in]size_t nelem, [in]size_t size );
    void *malloc( [in]size_t size );
    void *realloc( [in]const void *ptr, [in]size_t size );
    void free( [in]const void *ptr );
};
```

Fig. 3.12 Signature description of memory management

in the range of 64-128 based on the leftmost 1 bit. Second, the request is further classified as follows. The range from 64 to 128 is further divided into four equally sized groups, which 98 falling into the block ranging from 96 to 111. A free block[*3] in this range is then used to fulfill the memory allocation request.

Without using the above approach, a simple fixed-size memory block allocator results in wasted memory blocks of up to 50%. As illustrated above, TLSF classifies the memory allocation process finely in two steps. Therefore, it is a memory efficient algorithm. Fortunately, given its design, TLSF searches for memory blocks in constant time (i.e., $O(1)$).

3.2.3 Component Design of TLSF+TECS

This section describes the component design of the TLSF memory allocator. Note that I use TECS to componentize TLSF. Further, the version of TLSF used is 2.4.6[*4].

[*3] A free block is an available memory block.
[*4] http://www.gii.upv.es/tlsf/main/repo

```
celltype tTLSFMalloc {
    [inline]
        entry sMalloc eMalloc;
    attr {
        size_t memoryPoolSize;
    };
    var {
        [size_is(memoryPoolSize/8)]
            uint64_t *pool;
    };
};
```

Fig. 3.13 Celltype description of TLSF memory allocator component

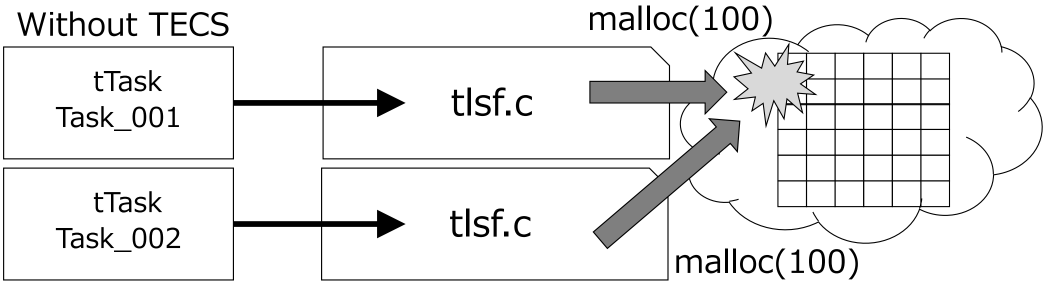

Fig. 3.14 TLSF before componentization

In Fig. 3.12, I summarize the function signatures used by the allocator for memory management. Particularly, this component defines the memory pool initialization function *initializeMemoryPool()*, memory allocation functions *calloc()*, *malloc()*, *realloc()*, and the memory release function *free()*.

Next, Fig. 3.13 shows the celltype description for the TLSF memory allocator component. Here, entry port *eMalloc* is connected to all components that perform memory management, including *malloc()* and *free()*. Further, *[inline]* is a specifier to implement as an inline function. Memory pool size is defined as an attribute, and a pointer to a memory pool is defined as a variable. Note that each component maintains its own heap area. Therefore, even when calling memory management functions simultaneously from different threads, it is possible to operate without any memory contention.

As shown in Fig. 3.14, since TLSF before componentization shares the heap area with multiple threads, if memory is allocated or released simultaneously via multiple threads, memory contention may occur in some cases, thus causing intermittent synchronization problems that can be extremely difficult to debug. In this study, TLSF is componentized using TECS, as shown in Fig. 3.15. It is possible to operate in thread safe without exclusive control because each component independently holds a heap area and manages memory within it.

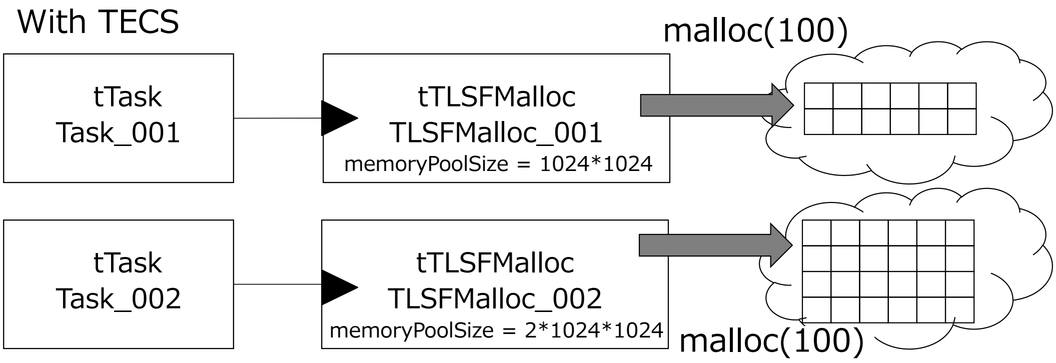

Fig. 3.15 TLSF after componentization

```
1  cell tTask Task_001 {
2      cMalloc = TLSFMalloc_001.eMalloc;
3  };
4  cell tTLSFMalloc TLSFMalloc_001 {
5      memoryPoolSize = 1024*1024; /* 1MB */
6  };
7  cell tTask Task_002 {
8      cMalloc = TLSFMalloc_002.eMalloc;
9  };
10 cell tTLSFMalloc TLSFMalloc_002 {
11     memoryPoolSize = 2*1024*1024; /* 2MB */
12 };
```

Fig. 3.16 Build description of TLSF memory allocator component

Next, Fig. 3.16 shows the build description of the TLSF memory allocator component illustrated in Fig. 3.15*5. Here, two sets of task components and TLSF components are combined. Further, each memory pool size can be configured as a variable (Lines 5 and 11 in Fig. 3.16). Fig. 3.17 presents the code that actually calls functions of the TLSF memory allocator component. The use part shows a function in which the mruby VM allocates memory within the mruby on TECS framework [9] [10], which is introduced in Section 2.3. Line 8 calls the *free()* function of the TLSF memory allocator component. *cMalloc_* represents the name of the call port on line 2 in Fig. 3.16. Similarly, lines 13 and 17 call the memory allocation function. Particularly, the heap area for the *TLSFMalloc_001* component is used if the code from 3.17 is executed in *Task_001*; conversely, if that code is executed in *Task_002*, the heap area of the *TLSFMalloc_002* component is used. Using this approach, in component-based development using TECS, it is possible to operate with the same code without modifying the underlying C code, although the resulting cells

*5 Other call/entry ports, attributes, and valuables are actually described, but it is omitted here for the simplicity.

```
void*
mrb_TECS_allocf( mrb_state *mrb, void *p, size_t size, void *ud )
{
    CELLCB *p_cellcb = (CELLCB *)ud;
    if (size == 0) {
        //tlsf_free( p );
        cMalloc_free( p );
        return NULL;
    }
    else if (p) {
        //return tlsf_realloc( p, size );
        return cMalloc_realloc( p, size );
    }
    else {
        //return tlsf_malloc( size );
        return cMalloc_malloc( size );
    }
}
```

Fig. 3.17 Example of TLSF memory allocator component

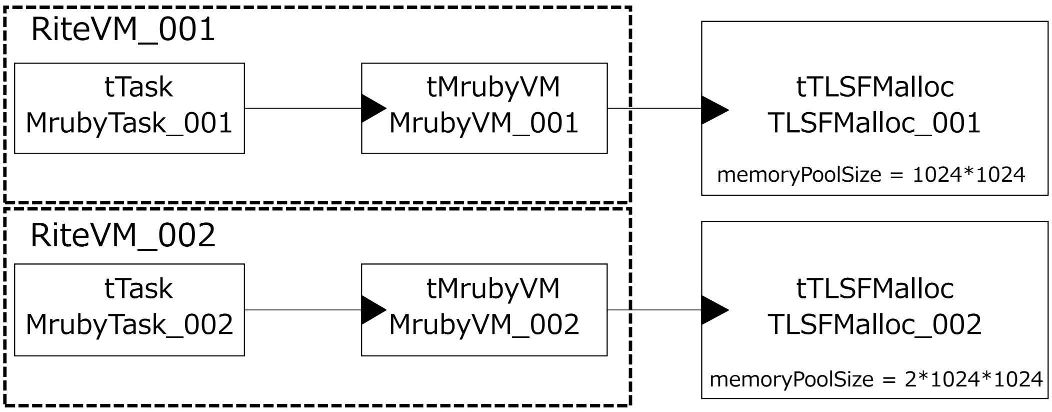

Fig. 3.18 Component description of RiteVM and TLSF+TECS

differ.

Multiple RiteVM Instances

The proposed framework uses the TLSF memory allocator for memory management within RiteVMs; however, since it is difficul to handle multiple memory pools in the existing TLSF, if memory is allocated or deallocated from multiple threads, memory contention will likely occur. Here, as a RiteVM allocates and deallocates memory at high frequencies, memory contention quickly occurs when multiple RiteVMs are instantiated.

Note that the TLSF components are connected to the RiteVMs to ensure that each

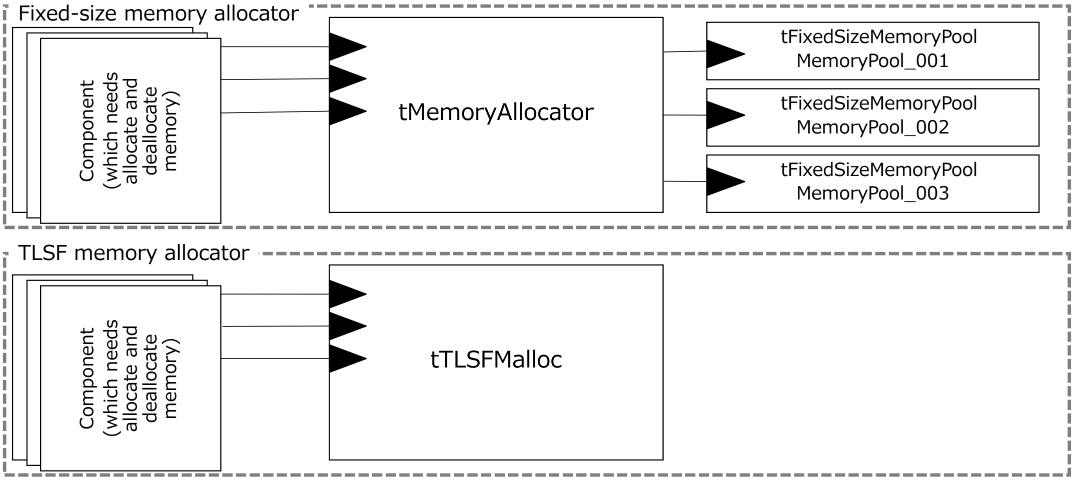

Fig. 3.19 Fixed-size and TLSF memory allocator components

component has its own heap area within each RiteVM, as shown in Fig. 3.18. Since each TLSF component maintains its own memory pool, multiple RiteVMs can be executed without the possibility of any memory contention. In the Fig. 3.18, I observe that the first RiteVM has a heap area of 1 MB (i.e., 1024 × 1024 bytes) and the second RiteVM has a heap area of 2 MB (i.e., 2 × 1024 × 1024 bytes). As illustrated in the figure, it is easy to configure different-sized heap areas for each RiteVM. Further, each RiteVM performs incremental garbage collection (GC), and a RiteVM that has started GC does not disturb the execution of other RiteVMs in its GC execution.

Memory management for sending and receiving data

In the TCP/IP protocol stack, memory allocation and subsequent deallocation are repeated in each layer, including the TCP, IP, Ethernet, and other layers. Therefore, the role of the memory allocator is critical. The TINET+TECS framework combines all components that manage memory within the allocator. The TLSF memory allocator can execute at the same speed as that of the fixed-size memory allocator that TOPPERS/ASP3 supports as standard; further, the TLSF memory allocator can improve memory efficiency. As shown in Fig. 3.19, the fixed-size memory allocator prepares memory pools of different sizes and selects a memory pool whenever it is necessary. Conversely, TLSF+TECS efficiently manages memory without the need to select a memory pool. Finally, TLSF+TECS can be easily extended to TINET+TECS since TINET+TECS is a component-based system.

```
begin
    io = AnalogIO.new(A0, INPUT)
    cep = TCP.new()
    cep.accept
    loop do
        val = io.read
        cep.snd val.to_s + ”\n”
        RTOS.delay(1000)
    end
rescue => e
    puts ”[ERROR]” + e
end
```

Fig. 3.20 Example of mruby application

3.3 Use case

In the proposed framework, applications can call TINET functions, such as specific TCP- and UDP-related functions, from mruby programs because the mruby-TECS bridge automatically generates the code to link mruby and C. Fig. 3.20 shows an example mruby program that transmits the value acquired from a sensor from the sensor to another device. In general, mruby makes it easier to develop applications than using C with the existing TINET framework.

For a simple application, typically only a few functions are used, with numerous unused functions. As an example, the application code shown in Fig. 3.20 only uses a function to send data via TCP. The proposed framework incorporates TINET+TECS and can easily customize the TCP/IP protocol stack by removing functions, such as UDP functions, functions that support only IPv4, or TCP receiving functions. As such, the proposed framework can be applied to embedded systems with strict memory constraints by removing many unused functions.

Chapter 4

Evaluation

This chapter describes the experimental evaluation used to demonstrate the effectiveness of the proposed framework. GR-PEACH was employed as the evaluation board. Detailed specifications of the board are shown in Table 4.1. I also employ TINET 1.5.4 and the compiler arm-none-eabi-gcc 5.2 To pretest the system, I connected the board to a host PC via a LAN cable and evaluated the data sending and receiving.

4.1 Performance of TINET+TECS

To demonstrate the low overhead of TINET+TECS, I compared its execution time and memory consumption with that of TINET, producing the results shown in Fig. 4.1.

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*, I 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. 3.2. For *tcp_rcv_dat*, I 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. 3.2. The execution time of TINET+TECS is close to that of TINET, with an overhead of about three us. 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 4.2. 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 4.3, the code lines for modification were measured to demonstrate the improved configurability. This demonstrated the ability to change the composition

Table 4.1 Evaluation board environment

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

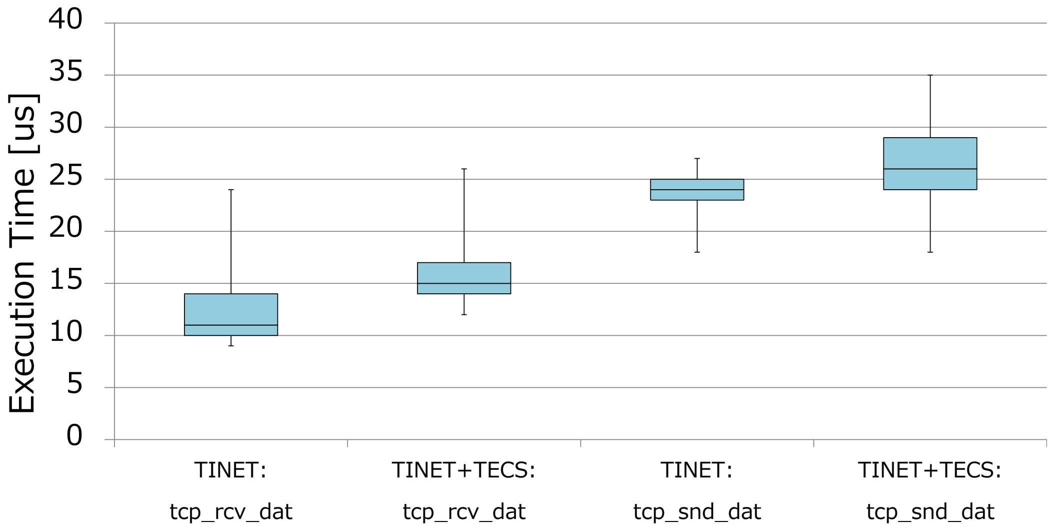

Fig. 4.1 Execution times of TINET and TINET+TECS

Table 4.2 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

of the protocol stack with a small workload, confirming that the proposed framework improves the configurability.

4.2 Dynamic connection

Memory consumption without and with TECS the dynamic connection was then evaluated. As shown in the left of Fig. 4.2, each CEP component should be statically connected to all REP components if the dynamic connection is not used. As the number of REPs increases, additional call ports of CEP are required, in turn increasing the consumption of

Table 4.3 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)

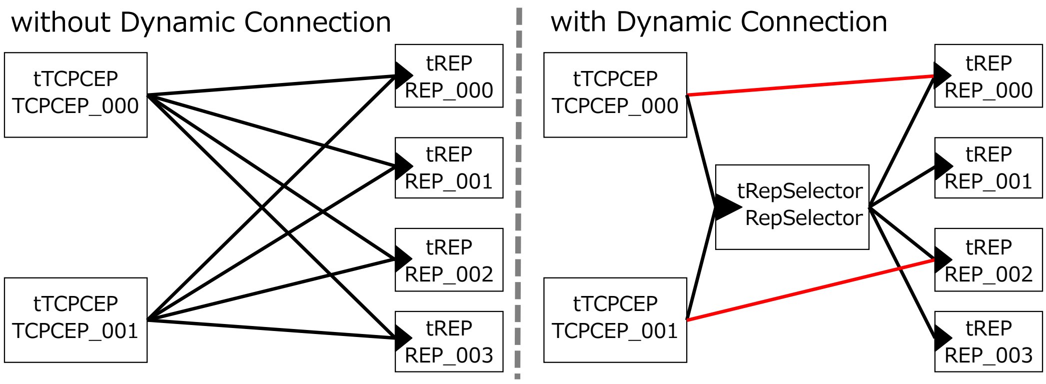

Fig. 4.2 Component diagrams for without/with the dynamic connection

Table 4.4 Memory consumption in two cases (with/without the 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

memory. The 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 of Fig. 4.2. 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 4.4. The dynamic connection case consumes the more RAM memory because, as mentioned in Section 3.1.3, 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 the dynamic connection is shown in TABLE 4.5 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 of Fig. 4.2, each CEP

Table 4.5 CDL code lines of without/with the 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[1] = REP_001.eREP;
    ...
    cREP[n] = REP_00n.eREP;
};
cell tTCPCEP TCPCEP_001 {
    cREP[0] = REP_000.eREP;
    cREP[1] = REP_001.eREP;
    ...
    cREP[n] = REP_00n.eREP;
};
..
cell tTCPCEP TCPCEP_00n {
    cREP[0] = REP_000.eREP;
    cREP[1] = REP_001.eREP;
    ...
    cREP[n] = REP_00n.eREP;
};
```

```
/* with Dynamic Connection */
cell tRepSelector RepSelector {
    cREP[0] = REP_000.eREP;
    cREP[1] = REP_001.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. 4.3 Two CDL codes (without/with the dynamic connection)

connects all REPs as shown in the upper of Fig. 4.3. In the right of Fig. 4.2, a CEP dynamically connects an REP, and only the selector connects all REPs as shown in the lower of Fig. 4.3. It is effective for software that uses many ports because the difference spreads as the number of CEPs and REPs increases.

4.3 Adapter overhead

The TECS adapter which enables to call functions in TECS from existing C codes is implemented between C codes and a TECS component and links a C function to a TECS function. Therefore, the existing applications can be applied to the proposed framework. However, since real-time processing is important for embedded systems, it is meaningless if the execution time is slower than the existing application.

In the section, it is evaluated whether the TECS adapter affects the system. The API execution time when using an adapter such as *tcp_snd_dat/eAPI_sendData* was used to

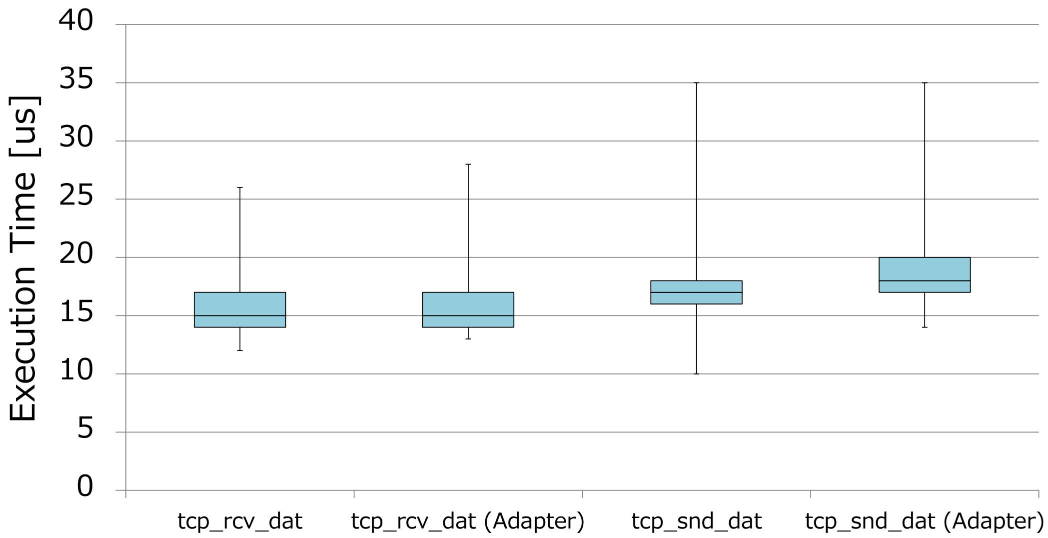

Fig. 4.4 Execution times in two cases (without/with the TECS adapter)

analyze the overhead of the TECS adapter supporting existing applications. As shown in Fig. 4.4, the overhead of the TECS adapter is small because the adapter only passes parameters from C codes to TECS components; thus, the overhead of TECS adapter does not affect the system.

4.4 Memory statistics using TLSF+TECS

The TLSF+TECS framework also provides functionality to acquire statistical information describing dynamic memory usage. Therefore, it is possible to analyze the operational status of the TLSF, GC on RiteVMs, because TLSF+TECS acquires and records the frequency of memory allocations and deallocations, as well as the usage sizes within the heap area. I describe memory usage and frequency below.

Memory usage. Fig. 4.5 shows the memory usage of a RiteVM, with presented data acquired from the statistical information available via the TLSF+TECS framework. From the figure, I observe that when the RiteVM is first activated, a large amount of memory is allocated and initialization is performed for several seconds. Next, the application runs for 20 s and the RiteVM subsequently terminates. The reduction in memory usage that occurs at regular intervals is due to the GC function of the RiteVM. As such, the TLSF+TECS framework provides users with the ability to visualize GC behavior and help verify its operation. Further, when the RiteVM terminates, the memory used by the RiteVM is not completely released, i.e., a few kilobytes remain. This remaining memory causes a memory leak when the number of RiteVMs increases or a RiteVM repeatedly activates and shuts down. Using the proposed environment here proves useful for detecting bugs

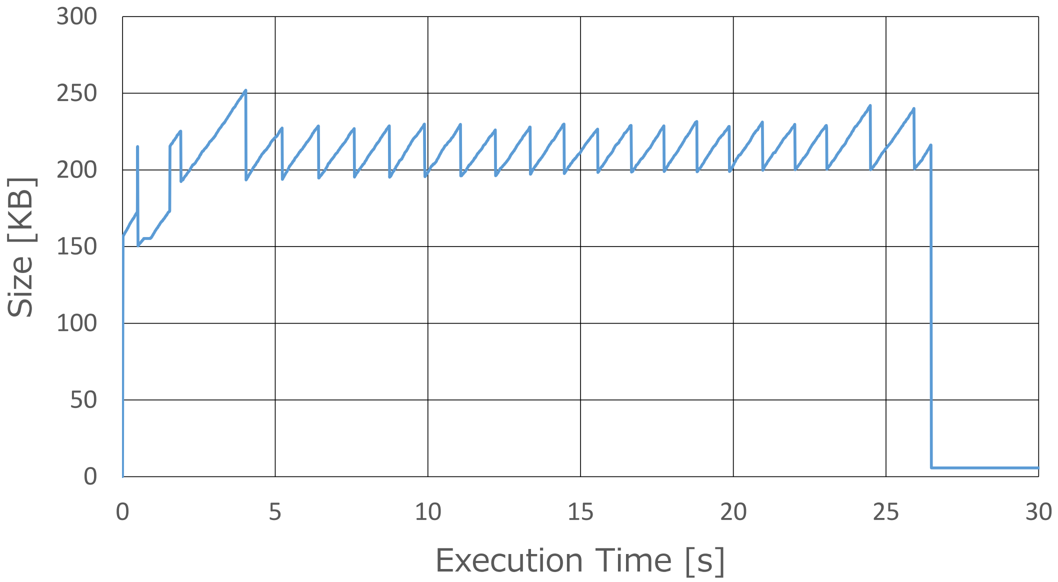

Fig. 4.5 Memory usage of RiteVM (mruby VM) by TLSFStatistics component

related to memory, which in practice can be extremely difficult to detect.

Memory frequency. Fig. 4.6 evaluates the amount of memory and the frequency of memory requests from a RiteVM. In this evaluation, the same application used above (i.e., Fig. 4.5) ran for 30 s, which during which the RiteVM calls *malloc()* rather frequency. When designating heap areas with fixed-size memory sizes, it is necessary to properly configure the memory blocks in response to the given application's specific needs. For example, this configuration may comprise 10 blocks of 64 bytes, 20 blocks of 128 bytes, and three blocks of 64 kilobytes. It is difficult to analyze and determine the number of memory blocks needed with RiteVMs as they consume memory at considerably high frequencies. Conversely, with TLSF+TECS framework, developers need only to prepare a large heap area; therefore, the workload of determining the approach to configure the blocks and areas containing useless memory are eliminated.

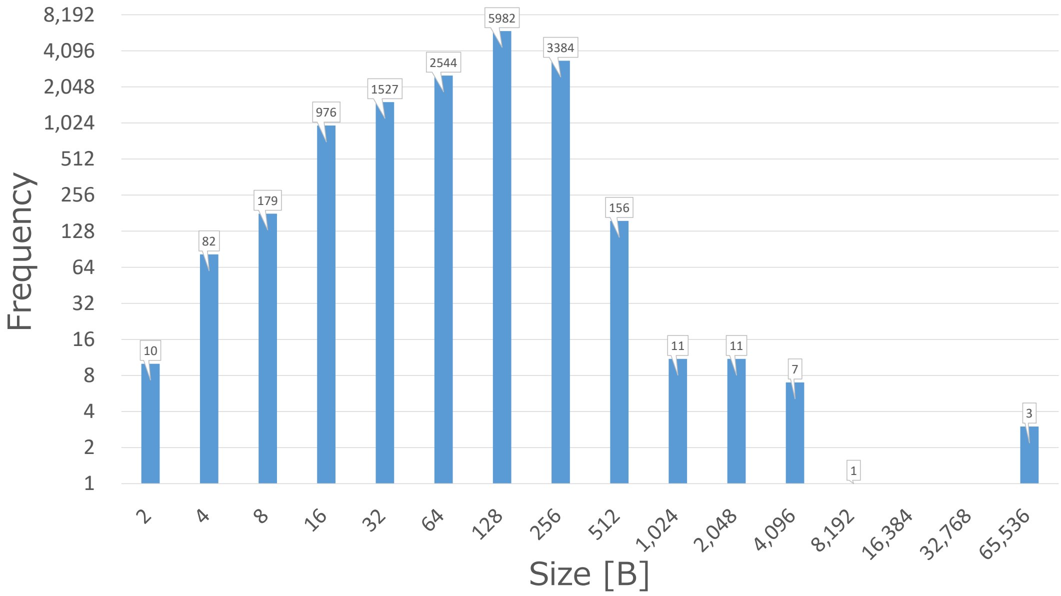

Fig. 4.6 Memory frequency of RiteVM (mruby VM) by TLSFStatistics component

Chapter 5

Related Work

To develop the software of IoT systems, several approaches have been proposed [27] such as Wireless Sensor Network (WSN) macroprogramming, Cloud-based platforms, and Model-Driven Development (MDD), General-purpose Programming Languages (GPLs).

WSN macroprogramming provides abstractions to specify high-level collaborative behaviors, while hiding low-level details such as message passing and state maintenance. nesC, a programming language used to build applications for the TinyOS platform [28], has been proposed. nesC/TinyOS is designed for WSN nodes with limited resources e.g., 8 KB of program memory, 512 bytes of RAM, but not supported TCP/IP implementation.

Cloud-based platform reduces development efforts by providing cloud-base APIs and high-level constructs (e.g., drag-and-drop) [29]. In addition, it offers the ease deployment and evolution because the application logic is centrally located in a cloud platform. However, it is platform-dependent design, and restricts developers in terms of functionality such as in-network aggregation or direct node-to-node communication locally. The cloud-based mruby framework, enzi Board [30], has been proposed. enzi can be developed and simulated on the Web, and developer can download and run the program on the board.

To address the issues of development efforts and platform-dependent design, MDD has been proposed [31]. MDD provides the benefits of reusable, platform-independent, extensible design; however, it needs long development time to build MDD systems.

The development using GPLs such as C, JavaScript, Python, and Android allows the extremely efficient systems based on the complete control over individual devices. However, GPLs need more development effort, and it is difficult to reuse software due to platform-dependent design.

5.1 Scripting languages for embedded systems

Open-source runtime systems for scripting languages have been proposed such as ChaiScript [32], python-on-a-chip [33], the Owl system [34], eLua [35], Squirrel [36], and mruby [11], [12].

ChaiScript: ChaiScript is a scripting language to be embedded in C++ applications. It is a dialect of ECMAScript (a.k.a. JavaScript). ChaiScript is header only, that is, it only needs to include the header files in order to be used it in an application. ChaiScript will support co-routines, but not now.

python-on-a-chip: python-on-a-chip (p14p) is a Python runtime system that uses a reduced Python VM called PyMite. The VM runs a significant subset of the Python language with few resources on a microcontroller. p14p can also run multiple stackless green threads.

Owl system: The Owl system is an embedded Python runtime system. It is a complete system for ARM Cortex-M3 microcontrollers. The Owl toolchain produces relocatable memory images that are directly runnable on the microcontroller from Python code objects. Note that the Owl system interpreter is the same as that of python-on-a-chip.

eLua: eLua (embedded Lua) offers a full implementation of the Lua programming language for embedded systems. Lua is one of the most popular scripting languages for embedded systems [37], [38]. Lua supports a co-routine, which is referred to as cooperative multitasking. A co-routine in Lua is used as an independently executed thread. Note that a co-routine can only suspend and resume multiple routines; thus, a Lua co-routine is not like multitasks in multitask systems.

Squirrel: Squirrel is an object-oriented programming language designed as a lightweight scripting language that satisfies the real-time requirements of applications. Squirrel was inspired by Lua. The Squirrel API is very similar to Lua and the table code is based on that of Lua; Squirrel also supports co-routines.

mruby: mruby, a lightweight implementation of the Ruby language, has been proposed for embedded systems. mruby programs can run on a RiteVM, which is the VM for mruby and reads the mruby bytecode. Note that the RiteVM only supports a single thread. In addition, mruby supports co-routines but does not support multitasking for RTOSs. mruby has the same syntax as Ruby which has advantages to web application development as it uses in Rails framework [39].

Table 5.1 compares the proposed framework to previous work. The proposed framework is a component-based framework for running mruby programs. mruby programs on TECS can be executed approximately 100 times faster than standard mruby programs. Software can be developed using CBD with mruby on TECS. In addition to supporting multitasking and co-routines, the mruby on TECS framework can be utilized RTOS functionalities such as periodic tasks and semaphores and supports legacy code such as a motor driver.

5.2 TCP/IP protocol stacks for embedded systems

Open-source TCP/IP protocol stacks for embedded systems have been developed such uIP [40] and lwIP [41].

Table 5.1 Comparison of open-source scripting Languages for embedded systems

	Call C functions	Legacy code of embedded system	Multitasking	Co-routines	RTOS functionalities
ChaiScript [32]					
python-on-a-chip [33]				Green thread	
Owl system [34]	✓	Partially		Green thread	
eLua [35]	✓	Partially		✓	
Squirrel [36]	✓			✓	
mruby [11]	✓			✓	
Proposed framework	✓	✓	✓	✓	✓

Table 5.2 Comparison of open-source TCP/IP protocol stacks for embedded systems

	Configurability	Generation	Binding	Support CBD	Memory saving
uIP [40]	Partially	Dynamic	Dynamic		✓
lwIP [41]	Partially	Dynamic	Dynamic		✓
TINET [13]		Static	Dynamic		✓
Proposed framework	✓	Static	Dynamic	✓	✓

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 [42], [43], 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. To customize protocols and buffer size, users should handle a lot of macros.

TINET: TINET is a compact TCP/IP protocol stack for embedded systems based on the ITRON TCP/IP API Specification [21], developed by the TOPPERS Project [22]. To satisfy restrictions for embedded systems, TINET supports several functions such as minimum copy frequency and elimination of dynamic memory control. Since TINET is TINET statically generates CEPs and REPs, which are like sockets.

Table 5.2 shows a comparison of TINET+TECS and previous work. Note that the proposed framework, TINET+TECS, has demonstrated all features shown in Table 5.2. TINET+TECS improves configurability more than TINET and is suitable for embedded systems with memory constraint. The proposed framework, TINET+TECS, can configure the TCP/IP protocol stack with minimum set compared to the other platforms.

Chapter 6

Conclusions

This thesis presented a component-based framework that can develop software including network applications for embedded IoT devices using a scripting language. It is an extended framework of mruby on TECS, including TINET+TECS and TLSF+TECS. In the proposed framework, mruby programs can call TINET+TECS functions through the mruby-TECS bridge. Development of software for IoT devices such as sensors and actuators will be more efficient due to the high productivity of mruby.

TINET+TECS is a componentized version of TINET, a compact TCP/IP protocol stack that uses TECS. It improves on TINET 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 thesis also presented the dynamic connection, a new TECS functionality, to enable dynamic processing while reducing memory consumption. TINET+TECS utilizes the dynamic connection to satisfy the TINET specification for supporting the static generation of CEPs and REPs.

TLSF+TECS is a component-based dynamic allocator. Since the TLSF+TECS can hold own heap area, memory contention will not occur even if memory is simultaneously allocated or released from multiple threads. The TLSF+TECS functionality to get statistical information of memory usage helps developers analyze the systems and find bugs.

In addition, the RiteVMs, TINET+TECS, and TLSF+TECS are implemented as components; therefore, developers can add, remove, or reuse their functionalities easily as required. Note that our prototype system and the application programs used in the performance evaluation are all open-source and can be downloaded from the website [44]. In the future, mruby libraries will be supported as mrbgems, which is an mruby distribution packaging system.

Acknowledgment

I would like to express my deep sense of gratitude to my adviser Assistant Professor Takuya Azumi, Professor Toshimitsu Ushio, and Associate Professor Takafumi Kanazawa, Graduate School of Engineering Science, Osaka University, for their invaluable, constructive advice and constant encouragement during this work. Assistant Professor Azumi's deep knowledge and their eyes for detail have inspired me much. I am deeply grateful to my advisers Hiroshi Oyama and Hiroaki Nagashima for their helpful comments on this thesis and warm encouragement. I would like to thank all members of Ushio Laboratory, especially the members of the Azumi team, for their kind help and cooperation. Finally, I would also like to express my gratitude to my family for their moral support and warm encouragements.

References

[1] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos: "Context Aware Computing for The Internet of Things: A Survey," *IEEE Communications Surveys Tutorials*, 16, 1, pp. 414–454, (2014).

[2] I. Crnkovic: "Component-based Software Engineering for Embedded Systems," *In Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pp. 712–713, (2005).

[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)*, pp. 372–379, (2000).

[4] G. Gössler and L. Aştefănoaei: "Blaming in Component-based Real-time Systems," *In Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, (2014).

[5] B. Bonakdarpour and S. S. Kulkarni: "Compositional Verification of Fault-tolerant Real-time Programs," *In Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pp. 29–38, (2009).

[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)*, pp. 46–50, (2007).

[7] "AUTOSAR,". `http://www.autosar.org/`.

[8] 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*, 80, 5, pp. 655–667, (2007).

[9] 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)*, pp. 252–259, (2015).

[10] T. Yamamoto, H. Oyama, and T. Azumi: "Component-Based Framework of Lightweight Ruby for Efficient Embedded Software Development," *JSSST Journal on Computer Software*, 34, 4, pp. 3–16, (2017).

[11] K. Tanaka and H. Higashi: "mruby – Rapid IoT Software Development," *In Proceed-*

ings of 17th International Conference on Computational Science and Its Applications (ICCSA), pp. 733–742, (2017).

[12] "mruby,". `https://github.com/mruby/mruby`.

[13] "TINET,". `https://www.toppers.jp/en/tinet.html`.

[14] "Ruby,". `https://www.ruby-lang.org/en/`.

[15] 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)*, pp. 95–99, (2016).

[16] "TOPPERS/ASP3,". `https://www.toppers.jp/asp3-kernel.html`.

[17] H. Takada and K. Sakamura: "μITRON for Small-Scale Embedded Systems," *IEEE Micro*, 15, 6, pp. 46–54, (1995).

[18] A. Ohno, T. Azumi, and N. Nishio: "TECS Components Providing Functionalities of OSEK Specifications for ITRON OS," *Journal of Information Processing*, 22, 4, pp. 584–594, (2014).

[19] "TOPPERS/HRP2,". `http://www.toppers.jp/en/hrp2-kernel.html`.

[20] 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)*, pp. 1–8, (2013).

[21] "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`.

[22] "TOPPERS Project,". `http://www.toppers.jp/en/index.html`.

[23] 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)*, pp. 204–209, (2008).

[24] 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)*, pp. 182–188, (2009).

[25] M. Masmano, I. Ripoll, A. Crespo, and J. Real: "TLSF: a New Dynamic Memory Allocator for Real-Time Systems," *In Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 79–88, (2004).

[26] "TLSF,". `http://www.gii.upv.es/tlsf/`.

[27] S. Chauhan, P. Patel, F. C. Delicato, and S. Chaudhary: "A Development Framework for Programming Cyber-physical Systems," *In Proceedings of the 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*, pp. 47–53, (2016).

[28] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler: "The nesC Language: A Holistic Approach to Networked Embedded Systems," *ACM SIGPLAN Notices*, 38, 5, pp. 1–11, (2003).

[29] H. Derhamy, J. Eliasson, J. Delsing, and P. Priller: "A survey of commercial frameworks for the Internet of Things," *In Proceedings of the 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pp. 1–8, (2015).

[30] "enzi Board,". `http://enzi.cc/`.

[31] V. Kulkarni and S. Reddy: "Separation of concerns in model-driven development," *IEEE Software*, 20, 5, pp. 64–69, (2003).

[32] "ChaiScript,". `http://chaiscript.com/`.

[33] "python-on-a-chip,". `http://code.google.com/archive/p/python-on-a-chip/`.

[34] T. W. Barr, R. Smith, and S. Rixner: "Design and Implementation of an Embedded Python Run-Time System," *In Proceedings of the USENIX Annual Technical Conference (USENIX ATC 12)*, pp. 297–308, (2012).

[35] "eLua,". `http://www.eluaproject.net`.

[36] "Squirrel,". `http://www.squirrel-lang.org/`.

[37] "Lua,". `http://www.lua.org/`.

[38] R. Ierusalimschy, L. H. d. Figueiredo, and W. Celes: "The Evolution of Lua," *In Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pp. 2–1–2–26, (2007).

[39] "Ruby on Rails,". `http://rubyonrails.org/`.

[40] A. Dunkels: "Full TCP/IP for 8-bit Architectures," *In Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pp. 85–98. ACM, (2003).

[41] A. Dunkels: "Design and Implementation of the lwIP TCP/IP Stack," *Swedish Institute of Computer Science*, 2, p. 77, (2001).

[42] 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*, 370, 1958, pp. 68–84, (2011).

[43] "Contiki OS,". `http://www.contiki-os.org/`.

[44] "TECS,". `http://www.toppers.jp/tecs.html`.

Publication List

Journal

[1] Takuro Yamamoto, Takuma Hara, Takuya Ishikawa, Hiroshi Oyama, Hiroaki Takada, and Takuya Azumi, "Component-Based mruby Platform for IoT Devices," *IPSJ Journal of Information Processing*, 2018 (conditionally accepted).

[2] Takuro Yamamoto, Hiroshi Oyama, and Takuya Azumi, "Component-Based Framework of Lightweight Ruby for Efficient Embedded Software Development," *JSSST Journal on Computer Software*, vol. 34, no. 4, pp. 3-16, 2017.

International Conference

[1] Takuro Yamamoto, Takuma Hara, Takuya Ishikawa, Hiroshi Oyama, Hiroaki Takada and Takuya Azumi, "TINET+TECS: Component-Based TCP/IP Protocol Stack for Embedded Systems," *In Proceedings of the IEEE 14th International Conference on Embedded Software and Systems (ICESS)*, 2017.

[2] Takuro Yamamoto, Hiroshi Oyama, and Takuya Azumi, "Lightweight Ruby Framework for Improving Embedded Software Efficiency," *In Proceedings of the 4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pp. 71-76, 2016.

Domestic Conference

[1] Takuro Yamamoto, Hiroshi Oyama, Takuya Azumi, "Componentized Dynamic Memory Allocator for Embedded Systems," *In Proceedings of IPSJ Kansai-Branch Convention 2017*, A-3, 2017 (in Japanese).

[2] Takuro Yamamoto, Hiroshi Oyama, Takuya Azumi, "mruby Bytecode Loader Using Bluetooth in Multi-VM Environment," *In Proceedings of ETNET2016*, No.8, pp.1-6, 2016 (in Japanese).