Extended RT-Component Framework for RT-Middleware

Hiroyuki Chishiro, Yuji Fujita, Akira Takeda, Yuta Kojima, Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki
School of Science for Open and Environment Systems
Keio University, Yokohama, Japan
{chishiro,fujita,takeda,kojima,funaoka,shinpei,yamasaki}@ny.ics.keio.ac.jp

Abstract

Modular component-based robot systems require not only an infrastructure for component management, but also scalability as well as real-time properties. Robot Technology (RT)-Middleware is a software platform for such component-based robot systems. Each component in the RT-Middleware, so-called "RT-Component" supporting particular robot functions, is based on Common Object Request Broker Architecture (CORBA). Unfortunately, the RT-Middleware lacks the mechanism for real-time control. In this paper, we extend the framework of the RT-Components to take care of timing constraints. We first enable tasks to have different periods within each RT-Component. We then modify the packet format of the General Inter-ORB Protocol (GIOP) to transfer the information of timing constraints over RT-Components. The performance evaluation on ART-Linux shows that the extended RT-Component framework improves the schedulability of distributed real-time tasks, without causing critical overheads in unmarshaling the modified GIOP packets.

1. Introduction

In recent years, the system integration of modular robots based on distributed control has received considerable attention in robotics. To this end, component-based middleware frameworks are useful, since each application logic can be decoupled from the QoS configuration of an entire system. CORBA Component Model [9] is a fundamental specification for component-based programming, which extends Common Object Request Broker Architecture (CORBA) [10, 11, 12] so that it manages components beyond objects. Unfortunately, this model is not mainly designed for component-based robot systems.

Robot Technology (RT)-Middleware [5] is a software platform for component-based robot systems, which is intended to establish basic technologies for the integration of robot functions. In order to provide easy-to-use interfaces for the development of component-based robot systems, the RT-Middleware is wrapping CORBA interfaces. For instance, OpenRTM-aist [2], an open-source implementation of the RT-Middleware, makes use of omniORB [1] as a component framework. ACE [15] is also used as a platform to run CORBA objects. "RT-Component" [13] is a basic software unit of the RT-Middleware. Robot developers adapt an RT-Component to each robot element, such as head, body, arm, and leg. A module is then composed of a couple of RT-Components on one processor. Every RT-Component communicates with each other, using General Inter-ORB Protocol (GIOP), an abstract protocol between ORBs, as shown in Figure 1.

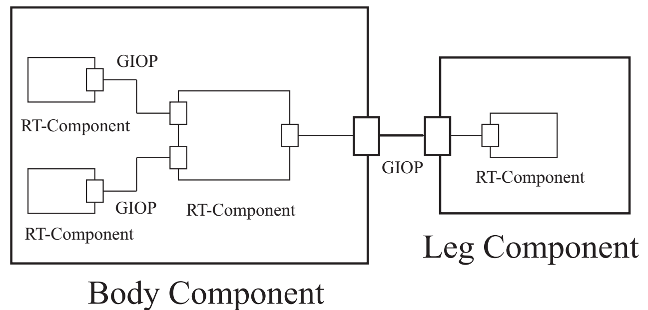


Figure 1. RT-Middleware usage example

Given the distributed control of modular robots, RT-Components are required to run with real-time properties which may be different from each other. For examples, the feedback control task and the image processing task have different periods and must be completed within their periods in robot systems. However, it is not straightforward to apply RT-Components in disributed real-time systems, because an RT-Component does not know the attributes of tasks, such as periods and worst-case execution times, in other RT-Components, and thus it is not able to guarantee that the response is returned from other RT-Components be-

fore the deadline of the control loop.

Composite component [4] accommodates multiple RT-Components, which is called *internal RT-Components*. The attributes of tasks in different RT-Components are managed by their container composite component to take care of timing constraints. The internal RT-Components within a composite component are executed sequentially in a static preconfigured order. An issue of concern is the overhead of communications among the internal RT-Components within a composite component, since marshaling and unmarshaling are needed every time an RT-Component communicates with another RT-Component.

In this paper, we extend the framework of the RT-Components to take care of timing constraints. The primary contribution of this paper is to improve real-time performance as well as robot system integration. We first enable tasks to have different periods within each RT-Component for more sophisticated control of modular robot systems. The task is prioritized in order of periods and executed in the RT-Component, not in the composite component, thus the overhead of communications among the RT-Components can be eliminated. We then modify the packet format of the General Inter-ORB Protocol (GIOP) to transfer the information of timing constraints over RT-Components.

The remainder of this paper is organized as follows: Section 2 provides the background of RT-Component. Section 3 describes a brief of extended RT-Component. Section 4 evaluates the schedulability of RT-Middleware. In Section 5, we compare our work with related work. Finally we offer concluding remarks in Section 6.

2. RT-Component

In this section, we explain the detail of RT-Component and how RT-Components are used to develop the development of component-based robot systems. Figure 2 shows the architecture of RT-Component. An RT-Component has an execution context that is an abstract thread in platform independence and ports that represent interaction points between a classifier and its environment. An RT-Component also has a state machine because the RT-Component is an autonomous object and works as a task. The interfaces associated with the port specify the nature of the interactions that may occur over a port. The required interfaces of the port characterize the requests that may be made from the classifier to its environment through this port. The provided interfaces of the port characterize requests to the classifier that its environment may make through this port. The current implementation of RT-Component has two kinds of ports: *DataPort* and *ServicePort*. The DataPort supports data centric communication, adopts publisher/subscriber model, and defines it as *Inport/OutPort*. The OutPort sends

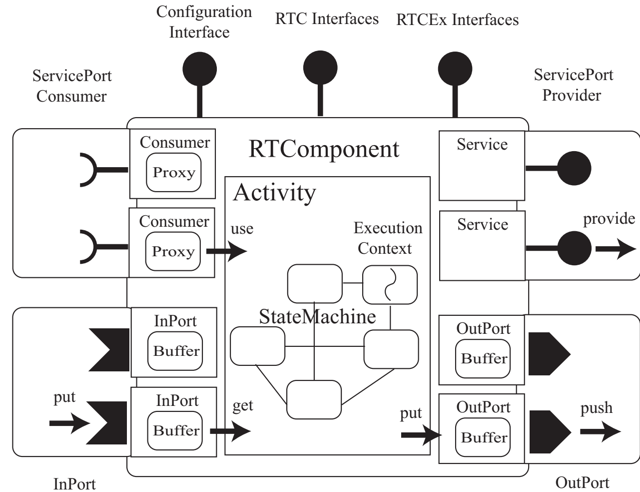


Figure 2. Architecture of RT-Component

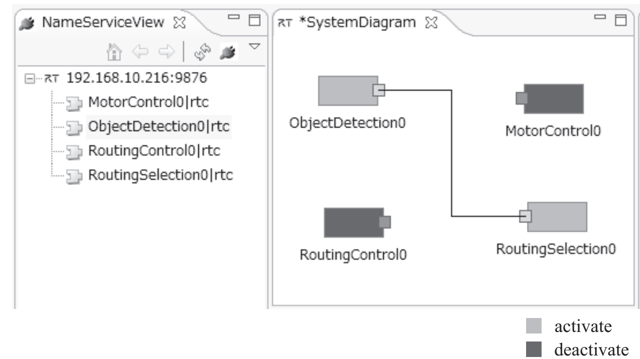


Figure 3. Development of wheel robot on RtcLink

data to the InPort and the InPort receives data from the OutPort. The ServicePort has user defined service interfaces and the *ServicePort Provider/Consumer* has some interfaces to provide/consume services.

Figure 3 shows the development of the wheel robot on the *RtcLink* that is a GUI component manager on an *eclipse*. RtcLink can make RT-Components activated/deactivated and connected/disconnected to other RT-Components. The *NamingServiceView* shows the CORBA name servers and RT-Components which robot developers registered. This robot has four RT-Components as modules of the wheel robot. The *ObjectDetection* and *RoutingSelection* components are activated and connected with ServicePorts each other and the *RoutingControl* and *MotorControl* components are deactivated and disconnected.

Using RT-Components, robot developers can simplify distributed control programming and reduce complicated implementation because they can use the RT-Middleware

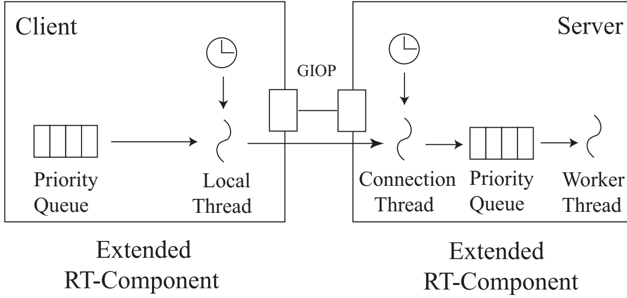


Figure 4. Architecture of extended RT-Component

that wraps operating system-specific APIs and develop reusable robot modules that do not depend on operating systems such as Windows and Linux.

3. Extended RT-Component

We describe the system model in our extended RT-Component framework. We assume that each robot module has one processor and all tasks are executed periodically and nonpreemptive. It has an advantage of throughput. Therefore, our extended RT-Component adopts the non-preemptive model. We present extended RT-Components for distributed control of modular robots. We first describe the thread management model of RT-Component. We then explain the packet format of modified GIOP and the RT-Component interfaces of our extended RT-Component framework. Finally we introduce usage examples of our extended RT-Component.

3.1. Thread Management

Figure 4 shows the architecture of the extended RT-Component. The RT-Component has only one period of task. In contrast, the extended RT-Component can have different periods of tasks in the priority queue. Communications among the internal tasks in the extended RT-Component is less overhead than that among the internal RT-Components in the composite component [4] because the former uses global variables and the latter uses GIOP. There are three different kinds of threads: *local thread*, *connection thread*, and *worker thread*. Local threads execute the activities of their RT-Components and are managed by ACE. Connection threads check periodically whether requests arrive and assign them to worker threads and are managed by CORBA. Worker threads execute requests assigned by connection threads. If there is no request in the priority queue, worker threads wait until requests arrive. The task is prioritized in order of periods.

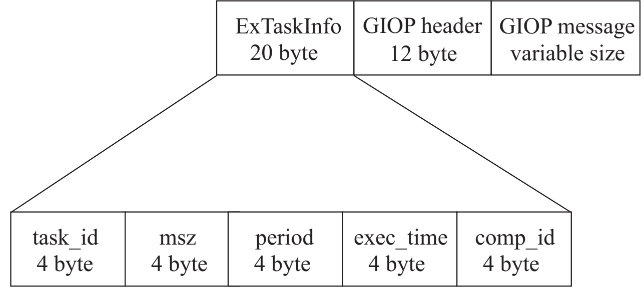


Figure 5. Modified GIOP packet

Our extended RT-Middleware selects omniORB that is usually used in CORBA-compliant middleware. The connection and thread management in omniORB has two primary modes of operation: *thread per connection* and *thread pool*. In thread per connection mode, each connection has a single thread dedicated to it. The thread is blocked and waits for a request. When it receives one, it unmarshals the arguments, makes the up-call to the application code, marshals the reply, and goes back to watching the connection. There is thus no thread switch along the call chain, meeting the call is very efficient. In thread pool mode, a single thread watches all incoming connections. When a call arrives on one of them, a thread is chosen from a pool of threads, and sets to work unmarshaling the arguments and performing the up-call such as the Leader-Followers pattern [17]. There is at least one thread switch for each call. In contrast, when a lot of connections occur, omniORB creates equal numbers of threads in thread per connection mode, which causes the lack of predictability. Therefore, omniORB selects *thread pool mode*.

3.2. Modified GIOP Packet

Figure 5 shows modified GIOP packet. A modified GIOP packet has an `ExTaskInfo` and a GIOP packet that includes a GIOP header and message to transfer the information of timing constraints. An `ExTaskInfo` has the following five attributes: (1) `task_id`, an identifier of a task. (2) `msz`, a sum of a GIOP packet size. (3) `period`, a period of a task. (4) `exec_time`, an execution time of a task. (5) `comp_id`, an identifier of a component. Adding `ExTaskInfo`, the extended RT-Component can determine the order by the attributes of tasks.

We change the algorithm of unmarshaling the modified GIOP packets. Pseudo code of unmarshaling the modified GIOP packets is shown in Figure 6. When the connection thread receives modified GIOP packets into the `buffer` (line 1), omniORB unmarshals them to modified GIOP packets (line 3-6) and inserts them to the priority queue (line 7). If modified GIOP packets in the `buffer` were all unmarshaled (line 9), the connection thread dequeues the `req`

```
1: recv_packet(buffer);
2: repeat
3:    task_info = get_next_task_info(buffer + offset);
4:    msz = task_info.msz;
5:    offset = offset + 20; // Size of ExTaskInfo
6:    packet = get_next_packet(buffer + offset, msz);
7:    enqueue_request(packet, task_info);
8:    offset = offset + msz;
9: until end of buffer
10: req = dequeue_request();
11: execute(req);
```

Figure 6. Pseudo code of unmarshaling modified GIOP packets

from the priority queue (line 10) and executes it (line 11). The overhead of unmarshaling the modified GIOP packets is ignorable in our experiment.

The Portable Interceptor [10] is another way to add or get the attributes of tasks for real-time control. However, it intercepts requests after they are dequeued from the request queue and ready to start the execution, thus it is not able to execute requests in order of priority. In contrast, in our mechanism, the attributes of tasks are known before they are enqueued to the request queue, thus requests can be serialized and enqueued to the request queue in order of priority according to their attributes.

3.3. Extended RT-Component Interface

Figure 7 shows the class diagram of the extended RT-Component. We show multiple important variables and functions in these classes due to space limitations. *Interfaces* in the Interface Definition Language is defined by *omniidl* that is the IDL compiler of omniORB.

We present the classes that are closely related with the classes of the extended RT-Component in the RT-Middleware. When the `Manager` class creates an RT-Component, it calls the `createComponent` function and applies to its `FactoryManager` whether the `module_name` of the RT-Component is registered in the CORBA name server. If it exists, the `FactoryManager` creates the `RTObject_impl` class that is a class to be a base of each RT-Component and then calls the `bindExecutionContext` function. The `RTObject_impl` class gets the object reference of the `RTObject` class that is derived from the `LightweightRTObject` class that has multiple operations for the activity of the RT-Component. If the `Manager` class can narrow the `RTObject` class to the `DataFlowComponent` class that is a base of an execution context, the `Manager` class applies to its `ECFactoryManager` whether the type of the execution context is registered in the `ECFactoryBase` class. If it exists, the `ECFactoryManager` creates the `PeriodicExecutionContext` class and then adds the `RTObject` class to it.

Now we describe the extended RT-Component interface to manage different periods of tasks. Our extended RT-Component is added to three classes: `ExManager`, `ExTask`, and `ExPeriodicExecutionContext`. The `ExManager` class that is derived from the `Manager` class has two overloaded functions: `createComponent` and `bindExecutionContext`. The arguments of these functions are the `module_name` of the RT-Component and the `vector` of the `ExTask` class. Each `ExTask` class has a period and an execution time. Each internal task in the extended RT-Component has its activity. The `ExPeriodicExecutionContext` class can change multiple states of internal tasks in the extended RT-Component by calling these overloaded functions: `activate_component`, `deactivate_component`, `reset_component`, `add`, and `remove`. The arguments of each function are the pointer of the `RTObject` class and the `vector` of the `ExTask` class.

The ports of the internal tasks are delegated to the extended RT-Component. Our extended RT-Component framework has backward compatibility. Therefore, robot developers can make use of both the conventional and extended RT-Component frameworks together.

3.4 Implementation

The extended RT-Component has different periods of tasks and the local thread executes these tasks. Figure 8 shows the execution of periodic tasks in the `svc` function. The argument of the `svc` function is the reference of the `vector` of the `ExTask` class. The `LCM` is the least common multiple and the `GCD` is the greatest common divisor of the periods of the internal tasks by microseconds in the extended RT-Component. If the local thread executes all the tasks of the execution timing, it waits until the next multiple of the `GCD`.

We explain usage examples of extended RT-Component interfaces for moving a wheel robot. The extended RT-Component has three tasks: *object detection*, *routing selection*, and *motor control*. The task of the object detection checks whether there are objects around the wheel robot. The task of the routing selection selects a movable route and notifies the route to the task of the motor control. The task of the motor control controls the torque of the wheel. Executing a series of treatment, the extended RT-Component can control the wheel robot.

Figure 9 shows an example of creating an extended RT-Component. The argument of the `create` function is the

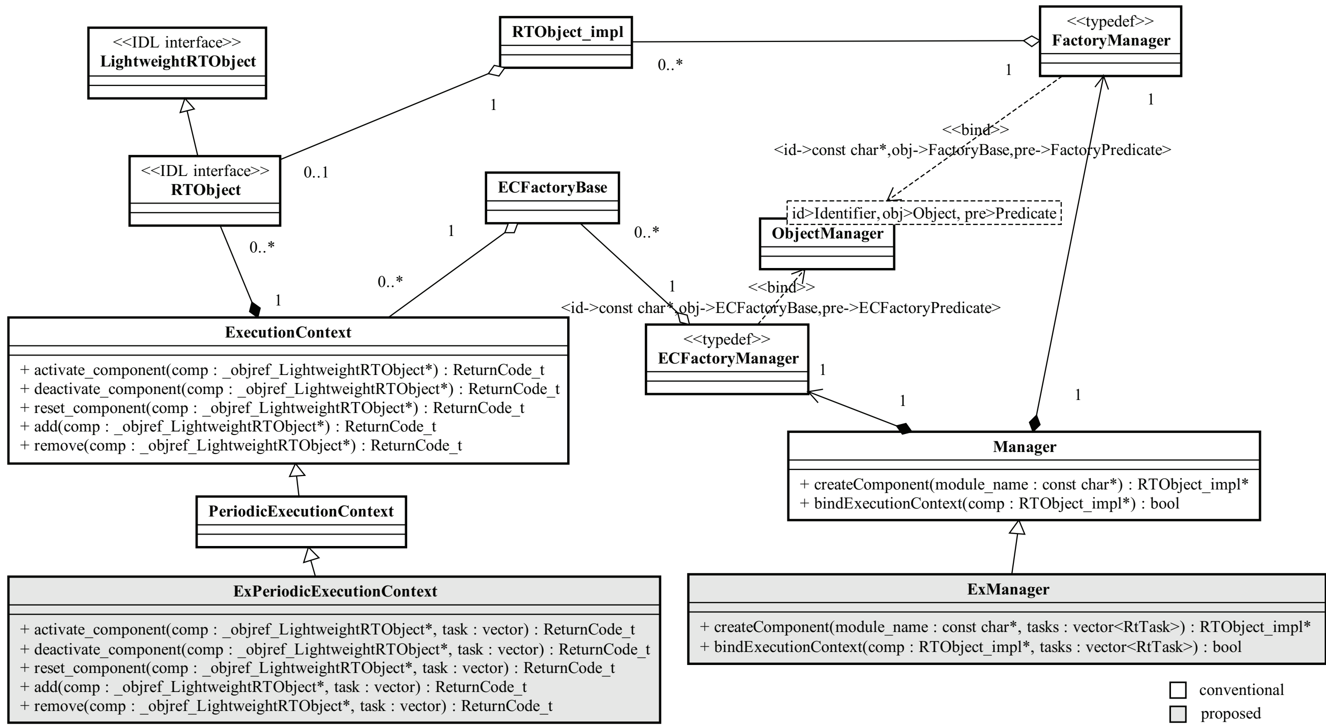


Figure 7. Class diagram of extended RT-Component

pointer to the `ExManager` class. The `task` pushes back three `ExTask` classes for real-time control. The first argument of `ExTask` class is its period and the second argument is its execution time by microseconds. The `Manager` class in the RT-Middleware calls its `createComponent` function, the argument of which is only the name of the RT-Component. The arguments of the `ExManager` class are not only the name of the extended RT-Component but also the `vector` of the `ExTask` class.

Figure 10 shows an example of executing the periodic tasks. The argument of the `execute` function is a `task_id` that is an identifier of a task. Each task calls `object_detection`, `routing_selection`, and `motor_control` functions in the `execute` function. The `execute` function is called by the local thread, as shown in Figure 8. Robot developers can design and implement the extended RT-Component that has these different periods of tasks collectively.

```
void svc(vector<ExTask>& task)
{
  int i, j;
  for (i = 0; i < LCM; i += GCD) {
    for (j = 0; j < task.size(); j++) {
      if (i % task.at(j).period == 0) {
        execute(j);
      }
    }
    wait();
  }
}
```

Figure 8. Execution of periodic tasks

4. Empirical Evaluation

In this section, we present that the results of experiments run on a Linux testbed. The experiments were performed using the OpenRTM-aist version 0.4.1. The OpenRTM-aist applies ACE version 5.6.0 and omniORB version 4.0.7, on a testbed consisting of two machines connected by a 100 Mbps Ethernet switch. One is Pentium-IV 2.53 GHz machine and the other is Pentium-IV 2.26 GHz machine. Each of them has 512 MB RAM, and runs version 2.6.22 of ART-Linux operating system [14].

The `RDTSC` (read-time stamp counter) instruction [7] is used to measure execution times of tasks. The priority of the real-time thread is higher than that of the kernel on ART-Linux. The RT-Middleware connects RT-Components using ServicePorts in all experiences. In order to account for the communication delay in the schedulability analysis on our experimental platform, we called requests between two machines 100 times and obtained the *approximate* maxi-

```
void create(ExManager* ex_mgr)
{
  vector<ExTask> task;
  task.push_back(ExTask(500000, 6000,
    OBJECT_DETECTION));
  task.push_back(ExTask(20000, 500,
    ROUTING_SELECTION));
  task.push_back(ExTask(1000, 30,
    MOTOR_CONTROL));
  ex_mgr->createComponent("MyRTC", task);
}
```

Figure 9. Example of creating extended RT-Component

```
void execute(int task_id)
{
  switch (task_id) {
    case OBJECT_DETECTION:
      object_detection();
      break;
    case ROUTING_SELECTION:
      routing_selection();
      break;
    case MOTOR_CONTROL:
      motor_control();
      break;
  }
}
```

Figure 10. Example of executing periodic tasks

mum communication delay, which was 9.8ms. In practice component-based robot system environments such as those for distributed control of modular robots usually make use of real-time networks that can provide guarantees on the worst-case communication delay. Therefore, we assume the worst-case communication delay as 10ms. Each task τ_i is defined by tupple (C_i, T_i) where C_i is an execution time and T_i is its period, then $U_i = C_i/T_i$ indicates a processor utilization of τ_i. Each instance of a task is a *job*. The total utilization in the system is $\sum_{i=0}^{n} U_i$. C_i and T_i are determined in the range of [10, 15, ... 300] ms and [100, 150, ... 1000] ms randomly.

Figure 11 shows the sequence diagram of the client/server. Two arguments of the `request/callback` operations, `task_id` and `seq_num`, are required to measure execution times of tasks. All messages are asynchronous (i.e., the operation has the *oneway* attribute in CORBA). This model is used in all experiments.

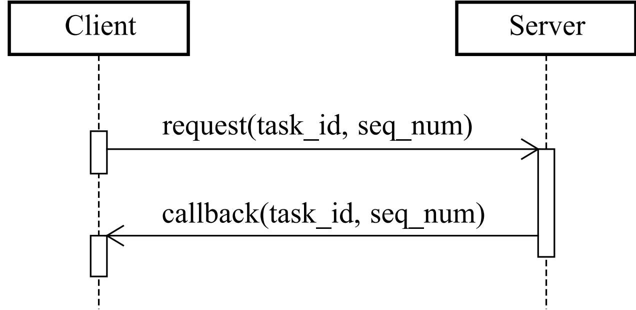


Figure 11. Sequence diagram of client/server

Table 1. Overhead of worker thread

Operation	Avg. [μs]
Unmarshaling modified GIOP packets	7.6
Assigning requests	194.9

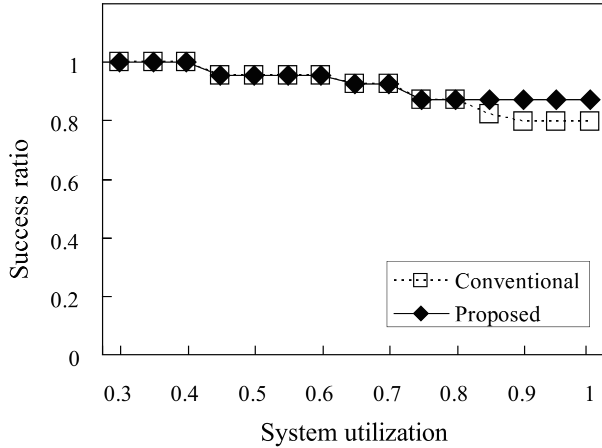
4.1 Overhead Measurement

In this experiment, we measure the overhead of unmarshaling the extended GIOP packets and assigning the requests by the worker thread. The tasks of assigning the requests include that of unmarshaling the modified GIOP packets (line 2-9), as shown in Figure 6. We assume that one machine has a client and server RT-Components and compare the overhead of communications among internal RT-Components in the composite component [4] with that of communications among internal tasks in the extended RT-Component. However, the latter is very little because every internal task in the extended RT-Component can communicate with each other using global variables so that we measure only the former. The server RT-Component receives modified GIOP packets from the client using ServicePorts. The real-time thread in this experiment is only the worker thread in the server RT-Component.

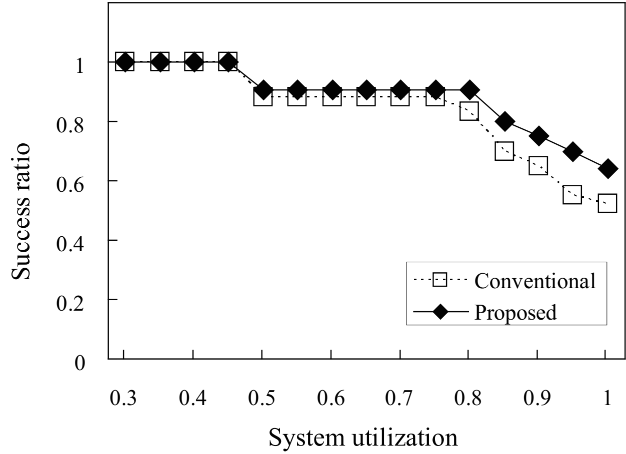
Table 1 shows the overhead of the worker thread. Each value in this table is the average of 100 measurements. The overhead of unmarshaling the modified GIOP packets is about 4% that of assigning the request so that it is ignorable in the worker thread. As previously discussed, the overhead of communications among the internal tasks in the extended RT-Component is ignorable because the communications do not make use of GIOP. If they occur frequently, the performance of the extended RT-Component improves dramatically.

4.2 Success Ratio

In this experiment, we evaluate the success ratio of our extended RT-Middleware. We assume that each processor



(a) # of request tasks is 3



(b) # of request tasks is 5

Figure 12. # of request tasks is (a) 3 and (b) 5

has one RT-Component. One is client and the other is server RT-Component. We define two task types as follows: (1) *Request Task*, a task that the worker thread executes in omniORB, (2) *Local Task*, a task that the local thread executes in the RT-Middleware. The priority of the local thread is higher than that of the worker thread. The RT-Component executes jobs of request and local tasks until the hyperperiod of the system in 100 times. The system utilization is determined within the range of $[0.3, 1.0]$. The success ratio is defined as follows:

$$\frac{\#\ of\ jobs\ of\ request\ tasks\ within\ deadline}{sum\ of\ jobs\ of\ request\ tasks}$$

Figure 12 shows the success ratio of jobs of request tasks. The success ratio of the proposed technique is as much as that of the conventional technique in the system utilization ranges from 0.3 to 0.75. On the other hand, the success ratio of the proposed technique is more than that of the conventional technique in the system utilization ranges from 0.8 to 1.0. In Figure 12(a), the proposed technique is maximum 8% higher than the conventional technique. In contrast, this is maximum 15% higher in Figure 12(b) due to the overhead of the context switch of the real-time thread and communications of RT-Components. Therefore, when the RT-Middleware uses a real-time thread for every real-time task instead of grouping them into single super-tasks which are scheduled using a thread pool, the results will be worse.

5. Related Work

Component-based middlewares are an effective way of achieving customizable reuse of software artifacts. In these middlewares, components are units of implementation and composition that collaborate with other components via ports. Groups of related components are connected together via their ports to form component assemblies. The ports define the collaborations of the components in terms of provided and required interfaces, event sources, and attributes. The ports isolate the contexts of the components from their actual implementations. Component-based middleware platforms configure and deploy component assemblies, and provide execution environments and common middleware services.

Feedback Controlled ORB (FC-ORB) [18] integrates end-to-end scheduling, adaptive QoS control, and fault-tolerant mechanisms that are optimized for unpredictable environments. FC-ORB can significantly improve the end-to-end real-time performance of distributed real-time and embedded (DRE) middleware in face of a broad set of dynamic uncertainties and fluctuations in execution times of tasks, resource contentions from external workloads, and processor failures. However, FC-ORB does not provide a component-based middleware framework.

Component-Integrated ACE ORB (CIAO) [6] is an implementation of lightweight CCM [9] and Real-time CORBA [8]. CIAO provides the component paradigm to the domain of DRE systems by abstracting DRE-critical systemic aspects, such as real-time QoS policies, as installable/configurable units supported by the component framework. CIAO implements the Component Implementation Definition Language (CIDL) compiler that extends the IDL compiler. In contrast, the RT-Middleware wraps CORBA interfaces, applies multiple CORBA-compliant middlewares, and implements *RtcTemplate* that can generate skeletons, stubs, and implementation files of the services for robot system integration. The generation of the imple-

mentation code uses the IDL compiler. Using RtcTemplate, robot developers can reduce the times of design and implementation.

Component Synthesis using Model Integrated Computing (CoSMIC) [16] is domain-specific tools for composing and deploying DRE middleware-based applications. The CoSMIC toolsuite is designed to model and analyze DRE application functionality and QoS requirements and synthesize CCM-specific deployment metadata for CIAO. In contrast, the RT-Middleware provides *RtcLink*, a GUI component manager, in which RT-Components can be activated and connected to other components. The RtcLink is implemented as a plugin on an *eclipse*.

Composite component framework [4] provides robot developers for real-time systems in the RT-Middleware. A composite component can include components to manage them and manage activities of internal components. The ports of the internal components are delegated to the composite component.

6. Concluding Remarks

In this paper, we extend the framework of the RT-Components to take care of timing constraints. We design and implement extended RT-Component interface that provides the priority management and manages multiple periodic tasks and modified GIOP packets to notify the attributes of tasks to the other RT-Component. Our extended RT-Component framework has backward compatibility. Therefore, robot developers can make use of both the conventional and extended RT-Component frameworks together. The performance evaluation on ART-Linux shows that the extended RT-Component framework improves the schedulability of distributed real-time tasks, without causing critical overheads in unmarshaling the modified GIOP packets.

In the RT-Middleware, GIOP usually makes use of TCP which is a non real-time communication protocol. For real-time control, GIOP is required to run over the real-time communication protocol, such as CAN [3]. The implementation of GIOP over such protocol is our future work.

Acknowledgement

A part of this research was supported by CREST, JST.

References

[1] omniORB. `http://omniorb.sourceforge.net/`.

[2] OpenRTM-aist. `http://www.is.aist.go.jp/rt/OpenRTM-aist/`.

[3] I. 11898. Road vehicles - Interchange of digital information - Controller area network (CAN) for high-speed communication, 1993.

[4] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W. K. Yoon. Composite Component Framework for RT-Middleware (Robot Technology Middleware). In *Proceedings of the 2005 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pages 1330–1335, July 2005.

[5] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W. K. Yoon. RT-Middleware: Distributed Component Middleware for RT (Robot Technology). In *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3555–3560, Aug. 2005.

[6] K. Balasubramanian, N. Wang, C. Gill, and D. C. Schmidt. Towards Composable Distributed Real-time and Embedded Software. In *Object-Oriented Real-Time Dependable Systems*, pages 226–233, Jan. 2003.

[7] I. Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*, Nov. 2007.

[8] O. M. Group. *Real-time CORBA Specification*, formal/05-01-04 edition, Jan. 2005.

[9] O. M. Group. *CORBA Component Model Specification*, formal/06-04-01 edition, Apr. 2006.

[10] O. M. Group. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1 Part 1: CORBA Interfaces*, formal/08-01-04 edition, Jan. 2008.

[11] O. M. Group. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1 Part 2: CORBA Interoperability*, formal/08-01-07 edition, Jan. 2008.

[12] O. M. Group. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1 Part 3: CORBA Component Model*, formal/08-01-08 edition, Jan. 2008.

[13] O. M. Group. *Robotic Technology Component Specification*, formal/08-04-04 edition, Apr. 2008.

[14] Y. Ishiwata and T. Matsui. A Real-Time Operating System that can Share Device Drivers with General Purpose OS. *IEICE technical report*, pages 41–48, 1998.

[15] D. C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Comunication Software. In *Proceedings of the 12th Annual Sun Users Group Conference*, Dec. 1993.

[16] D. C. Schmidt, A. Gokhale, B. Natarajan, S. Neema, T. Bapty, and J. Parsons. CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Component Middleware and Applications. In *Proceedings of the ACM OOPSLA 2002 Workshop on Generative Techniques in the Context of the Model Driven Architecture*, Nov. 2002.

[17] D. C. Schmidt and C. O'Ryan. Leader/Followers: A Design Pattern for Efficient Multi-Threaded Event Demultiplexing and Dispatching. In *Proceedings of the 7th Pattern Languages of Programs Conference*, Aug. 2000.

[18] X. Wang, Y. Chen, C. Lu, and X. Koutsoukos. FC-ORB: A Robust Distributed Real-time Embedded Middleware with End-to-End Utilization Control. *System and Software*, 80(7):938–950, July 2007.