CREAM: A Generic Build-time Component Framework for Distributed Embedded Systems

Chetan Raj, Jiyong Park and Seongsoo Hong

Real-Time Operating Systems Laboratory
Seoul National University, Seoul 151-744, Korea
{chetan, parkjy, sshong}@redwood.snu.ac.kr

Abstract

A component framework plays an important role in CBSD as it determines how software components are developed, packaged, assembled and deployed. A desirable component framework for developing diverse cross-domain embedded applications should meet such requirements as (1) lightweight on memory use, (2) integrated task execution model, (3) fast inter-component communication, (4) support for distributed processing, and (5) transparency from underlying communication middleware. Although current embedded system component frameworks address some of the above requirements, they fail to meet all of them taken together. We thus propose a new embedded system component framework called CREAM (Component-based Remote-communicating Embedded Application Model). It achieves these goals by using build-time code generation, explicit control of task creation and execution in the component framework, static analysis of component composition to generate efficient component binding, and abstraction of the component’s application logic from the communication middleware. We have implemented the CREAM component framework and conducted a series of experiments to compare its performance characteristics to a raw socket-based communication implementation and the Lightweight-CCM implementation by MicoCCM.

Keywords: CBSD, Component Frameworks, Component Models, CCM, Koala, AUTOSAR, CORBA

1. Introduction

The ever increasing complexity of software has led to the wide adoption of component-based software development (CBSD) [1]. The CBSD is an engineering methodology used to build a software system by composing software components [2]. These components are software artifacts that are specially designed to be used in diverse contexts. The CBSD requires less time to assemble components than to design, code, test and debug the entire system. This development methodology greatly reduces the software cost and the time to market.

In order for independently developed components to be seamlessly integrated with each other, there must be certain rules that govern how components are developed, packaged, assembled and deployed. The component framework enforces the component to adhere to these rules [2, 3] by providing gluing mechanisms for component composition, communication, synchronization, deployment and execution. Since the component framework determines the overall system structure and implements fundamental services, the final system performance and efficiency is greatly affected by the underlying component framework.

The current component frameworks for embedded systems have been designed either based on existing enterprise computing component frameworks or from scratch to suit a particular application domain. Popular embedded system component frameworks such as Lightweight-CCM [4], SCA [5] and .NET compact framework [6] are designed based on existing enterprise computing component frameworks. In designing such frameworks, features that are highly specific to enterprise computing are removed in order to minimize the complexity for embedded systems. However, they still require heavy resources and have significant performance overhead as they retain many of the fundamental features to guarantee the backward compatibility with their base component frameworks. For example, Lightweight-CCM is based on CCM [7] and they both use the CORBA [8] middleware for inter-component communications. Therefore, the use of such component frameworks is limited to high-end embedded devices such as telecommunication network devices and military radio devices [9].

There have been component frameworks designed from scratch for the embedded systems where compact versions of enterprise component frameworks are not suitable. Koala [10], AUTOSAR [11], OROCOS [12] and PECOS [13] are widely known examples. They have been specially designed for consumer electronics, automobile control systems, industrial robots and field devices, respectively. However, since they are highly optimized for specific application domains, it is almost impossible to use them in other domains. For example, AUTOSAR uses domain-specific real-time control networks such as CAN and FlexRay [14, 15]. Therefore, AUTOSAR is not suitable for generic in-vehicle entertainment systems where those control networks are
seldom used. Moreover, research on extending AUTOSAR for other domains is still in its early stages.

In this paper, we propose CREAM (Component-based Remote-communicating Embedded Application Model) as a generic build-time component framework for embedded systems. It is designed to be used in a wide range of embedded systems while using less system resources and providing performance efficiency. Specifically, CREAM is designed for following five requirements essential for developing the current-generation of cross-domain embedded applications.

1. Lightweight on memory usage  
2. Integrated task execution model  
3. Fast inter-component communication  
4. Support for distributed processing  
5. Transparency from underlying communication middleware

To the best of our knowledge, CREAM is the only component framework that strives to achieve all the above design requirement taken together. The existing component frameworks meet only subsets of these requirements. For example, Koala, OROCOS and PECOS lack support for distributed processing. AUTOSAR is highly dependent on the OSEK-COM [16] communication middleware. Lightweight-CCM and SCA require a significant amount of memory and CPU time. Moreover, all inter-component communications in CCM and SCA are done through the CORBA communication middleware which significantly increases the inter-component communication time [17].

The main idea of CREAM is to utilize build-time information and static analysis of the final component-composed system in order to improve the run-time performance and reduce the usage of system resources. In particular, the build-time information describes how components are composed, deployed and executed. The analysis of this build-time information to statically configure the component-composed system removes the costly run-time features from the final system.

One of the main contributions of the CREAM component framework is the separation of the component model from the underlying communication middleware. Unlike some other component frameworks such as the CCM, the CREAM components’ business-logic is completely abstracted from the communication middleware used for remote inter-component communication. This mechanism enables CREAM to support different communication middleware without modifying the component business-logic source code.

We have implemented the CREAM component framework and supporting tools such as an automatic code generator. We have conducted a series of experiments comparing CREAM’s communication performance characteristics to a raw socket-based communication implementation and to a popular Lightweight-CCM implementation by MicoCCM which is used in distributed real-time embedded systems [18, 19]. The experimental results showed that CREAM incurred little overhead compared to the raw communication implementation. On the other hand, the CREAM achieved improved inter-component communication time and used lesser run-time memory as compared to the MicoCCM.

The remainder of this paper is organized as follows. In Section 2, we enumerate the design requirements for our component framework. In Section 3, we present the CREAM component framework along with its component model. In Section 4, we explain the key mechanisms used in CREAM to achieve the design requirements. In Section 5, we describe the application development process using an example of a networked robot application. In Section 6, we describe the CREAM implementation and experimental results. Finally, in Section 7, we provide our conclusions.

### 2. Design Requirements

The CREAM component framework strives to meet the following five design requirements that are essential for developing the current-generation of cross-domain embedded systems applications. The failures to meet these requirements by popular embedded system component frameworks are also mentioned.

1. Lightweight on memory usage: Despite decreases in prices of solid state memory devices, memory is still a precious resource in embedded systems. Embedded system applications generally run on little memory. However, many existing component frameworks require a huge amount of run-time and code memory, and hence they are not suitable for resource-constrained embedded systems. Those component frameworks support complex features such as naming service, dynamic component composition and runtime XML processing that are rarely useful in embedded system applications.

2. Integrated task execution model: Embedded systems applications generally have many active components with independent threads of control. Moreover, many embedded systems applications have real-time constraints. In such systems, handling of task creation and execution forms an important activity. Explicitly controlling those activities in the component framework provides greater predictability and analyzability of the embedded systems applications. In particular,
with the knowledge of the entire task model of an application, the component framework could provide automatic synchronization among shared components. In addition, the component framework could assist in WCET evaluation and configuring real-time properties such as period, deadline and priorities.

Many of the current component frameworks do not support a task execution model. For example, Koala, CCM and SCA lack handling of task creation and execution at the component framework level [4, 5, 10].

3. Fast inter-component communication: Components can communicate with each other using various methods. If they are located in the same address space, a simple direct method call is sufficient. On the other hand, remote procedure call (RPC) should be used when components are in different address spaces or in different physical nodes. Therefore, a suitable communication mechanism must be chosen depending on components deployment location.

However, many of the current generic distributed component frameworks such as CCM and SCA route all inter-component method calls through their underlying ORB (object request broker) [17]. The appropriate communication method is then dynamically chosen by the ORB. This mechanism causes significant performance overhead for embedded systems.

4. Support for distributed processing: Many embedded systems such as automobile systems consist of tens of distributed nodes [20]. Recently, mobile phones are also equipped with multiple processors connected via on-chip networks [21]. Therefore, the support for distributed processing is becoming a prerequisite for an embedded system component framework.

However, many of current embedded component frameworks such as Koala and PECOS do not have support for this feature. Those component frameworks only support components in the same address space [10, 13].

5. Transparency from underlying communication middleware: A component framework useful for developing cross-domain applications should be independent of communication middleware and the underlying networks. For example, a networked home service robot having its own communication middleware needs to co-operate with home networking appliances using another communication middleware.

However, many of the distributed embedded component frameworks are tightly coupled with their underlying communication middleware. For example, AUTOSAR is strongly coupled to OSEK COM [11, 16]. CCM and SCA are coupled to the CORBA communication middleware [5, 7, 8].

The support offered by existing embedded system component frameworks for these design requirements are as shown in Table 1. As shown in the table, we observe that the CREAM component framework achieves all of the aforementioned design requirements.

### Table 1. Comparisons of Component Frameworks

<table>
<thead>
<tr>
<th>Component Framework</th>
<th>Lightweight on memory</th>
<th>Integrated task-execution model</th>
<th>Fast inter-component communication</th>
<th>Support for distributed Processing</th>
<th>Transparency from communication middleware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Koala</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>-NA-</td>
</tr>
<tr>
<td>PECOS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-NA-</td>
</tr>
<tr>
<td>AUTOSAR</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>CCM</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SCA</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CREAM</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### 3. The CREAM Component Framework

The CREAM component framework manages the underlying component model. It uses services of an object-based communication middleware to support remote inter-component communication. The CREAM component framework defines the component composition and deployment semantics. It makes use of XML based domain-profiles to describe, configure and deploy components in the final component-composed system.

#### 3.1. Component Model of CREAM

The component model used in CREAM is similar to that of other popular component frameworks such as CCM [4] and AUTOSAR [11]. This component model can be visualized as shown in Figure 1 (a). A component interacts with other components and its environment using ports [2, 3]. The interconnection of different component’s ports forms the composition of those components as shown in Figure 1 (b).

#### 3.1.1. Components’ Port

A port is defined as a point of interaction between a component and its environment. These interactions occur through well-defined interfaces [2]. The ports in CREAM can be further categorized into client-server ports and event-service ports.
method calls can be used. The CREAM code generator can be extended to support any object-based communication middleware without requiring the costly re-coding of existing components’ business logic.

3.3. Component Composition and Deployment

Component composition is defined as a process of integrating two or more components into a single unit. In CREAM, the composition of client-server ports involves associating requires port object references of one component with provides port object instances of another component. The event-service ports are composed together by associating event publisher and subscriber ports to a common event channel as accomplished in other push-type event models [22].

In CREAM, deploying components involves grouping of component instances into different partitions. A partition is executed as an OS process. All component instances of the same partition form collocated components and share the same address space. These partitions are managed by a separate standalone DomainExecutionManager which waits for the boot up of all partitions. It can then be used to start and stop the execution of partitions in the system.

3.4. Domain Profiles

The CREAM component framework makes use of XML based domain profiles as its component definition language [2] for describing various operations on components. These XML domain profiles are modeled on the lines of CCM [4] and SCA [5] XML descriptors. These domain profiles are – (1) Software Component Descriptor (SCD) used for specifying and developing components, (2) Software Packaging Descriptor (SPD) for describing the software component package, (3) Component Properties Descriptor (CPD) for describing the custom properties of component instances, (4) Software Assembly Descriptor (SAD) for describing the assembly of components to form an assembly, and lastly (5) Software Deployment Descriptor (SDD) which provides the partitioning and deploying information. These domain profiles are consumed by the CREAM’s code-generator to produce the final component assemblies as shown in above Figure 2.
4. Key Mechanisms of CREAM

The key mechanisms of CREAM that achieve the aforementioned design requirements are explained in this section.

4.1. Build-Time Code Generation for Developing a Lightweight System

The CREAM is a build-time component framework. The component framework binds all component references and dependencies at build-time. This analysis helps remove costly memory consuming features such as XML-parsers, naming-services and dynamic component binding to achieve a lightweight system.

The CREAM code generator analyzes the domain profiles and extracts required information at build-time. This information includes the components’ interfaces and ports, inter-connection of components’ ports, custom properties of component instances, partition and deployment information. The code generator uses this information to generate statically configured code that instantiates the components, inter-connects the components’ ports and deploys the composed components. This static analysis removes the need for a heavy run-time in the final deployed system.

The CREAM component framework eliminates costly naming-service from the final deployed system. Naming-service is used to locate object-references at run-time. However, the CREAM’s code generator statically binds the component instances and embeds all the required code to locate the external components. This static binding removes the need for the naming service.

4.2. Integrated Task Execution Model

Handling of task creation and execution forms an important activity in embedded software systems. These systems usually have many active elements that need their own threads of control. Manual coding of task creation and execution for such active elements causes the strong coupling of applications to target platforms. Moreover, manual coding for task creation leads to difficulties in predictability and analyzability of the embedded application system. To address this problem, the CREAM has integrated the task execution model into the component framework. The CREAM defines the execution model of the component composed system and explicitly controls the creation and execution of all tasks in the system. This integrated task model enables automatic synchronization among shared component instances and helps analyze the WCET of tasks.

The CREAM component framework manages the creation and execution of a task (implemented as a thread in Linux and Windows). In CREAM, components are of two types: (1) active components, with an independent thread of control, and (2) passive components, with no independent thread of control. In CREAM, active components implement a \texttt{run} method. The CREAM component framework creates a task and initializes its entry point to the \texttt{run} method for each active component.
The task model of CREAM enables automatic synchronization among shared component instances. This mechanism is described in Figure 4. The CREAM code generator statically parses the software assembly descriptor (SAD) to analyze for shared component instances used by two or more active components. The code generator then automatically embeds code that uses underlying OS task synchronizing primitives such as a mutex and semaphore to coordinate access to these shared component instances.

The integrated task model of CREAM helps in using external WCET analysis tools within the CREAM component framework. The WCET analyzer tools such as Bound-T, SWEET and MTime [25] are widely used in the development of real-time applications. These tools analyze the worst case execution time for tasks in the system and aid schedulability analysis for various real-time scheduling algorithms. The CREAM component framework, having the complete knowledge of all the tasks in the system, can automatically configure these WCET tools to evaluate the worst case execution time for all tasks.

4.3. Fast Inter-component Communication

The CREAM achieves inter-component communication performance efficiency for collocated inter-component method calls by mapping collocated components’ port composition to local function calls and remote components’ port composition to communication middleware based remote function calls.

The composition optimization is achieved using polymorphism and the delegator design pattern. The port interface type is associated with an abstract class. This abstract class has two implementations: (1) the actual business logic implementation of interface methods and (2) the delegation implementation to a proxy that handles remote object communication. CREAM’s code generator automatically generates the second implementation. The collocated inter-component calls are mapped to the actual business logic implementation method. The remote inter-component calls are mapped to the auto generated delegation implementation method.

The entire mechanism is visualized in Figure 5. Interface Printer with method print is processed by the CREAM’s code generator. It transforms this IDL code into the communication middleware’s IDL code. The middleware’s IDL processor then creates PrinterProxy that handles remote communication with the Printer type object. The CREAM code generator then generates the abstract PrinterAbstract base class. The component developer implements the business logic of PrinterAbstract methods in the PrinterImpl derived class. The PrinterRemote derived class is auto-generated by the CREAM’s code-generator. The PrinterRemote class uses PrinterProxy to delegate the print method. Then, the PrinterProxy uses the communication middleware to invoke PrinterImpl method on a remote host.

```
interface Printer
{
  void print(string someMessage);
};
```

```
CREAM Code Generator

Middleware IDL

Middleware IDL processor

PrinterAbstract

PrinterProxy

<<realization>>

PrinterImpl
- string buffer;
+ void print(string message)
  { buffer = message; cout<< buffer; }

PrinterRemote
- PrinterProxy proxy;
+ void print(string message)
  { return proxy->print(message); }

Business logic, implemented by component developer

Delegator, auto generated by CREAM code generator

Middleware IDL processor

PrinterAbstract

PrinterProxy

<<dependency>>

PrinterImpl

PrinterRemote

<<realization>>

Figure 5: Interface Methods Local and Remote Implementation.
```

The build-time binding of component ports to appropriate local or remote references provides optimization and efficiency over that of run-time component frameworks. Those component frameworks usually bind all component ports at run-time and use their communication middleware for inter-component method calls. This communication middleware overhead for collocated inter-component communication is completely avoided in CREAM.
4.4. Transparency from Underlying Communication Middleware

In the CREAM component framework, the component model and operations on components such as component construction, composition and deployment are made independent of the underlying communication middleware. This separation is achieved by developing a thin abstraction layer for the communication middleware, having minimal requirements on object-based communication middleware, and code-generation tools. This mechanism enables the support for different communication middleware without changing the business logic of components and applications.

Figure 6 shows the abstraction of the communication middleware from the component model in the architecture. This abstraction is achieved through a thin communication services layer. This layer can be implemented with any object-based communication middleware such as Ice-E [23] and minimum CORBA [24].

The CREAM component framework has minimal requirements on the communication middleware. Specifically, it only requires object methods marshalling and un-marshalling support from the communication middleware. Any communication middleware which support this minimal requirement can be used in the CREAM component framework.

The CREAM code generator helps achieve the separation of application logic code from the communication mechanisms. The code generator automatically associates the application business logic object to communication middleware’s object servants. It then extracts the business logic object’s information from communication middleware’s object proxies. Therefore, the CREAM’s code generator is the only piece of code that must be extended to support any new communication middleware. This preserves the investment done on developing the actual business logic of components and enables the components to be deployed over different communication middlewares.

5. Application Development in CREAM

The CREAM component framework is currently being used to develop networked home service robots. We shall explain the application development process in CREAM using this robot application. The hardware platform for these robots consists of ERSP Scorpion robot from Evolution Robotics [26], a microcontroller, and a desktop computer. The software platform consists of two processing nodes, one on a desktop computer and another on the microcontroller. The microcontroller is mounted on top of the Scorpion robot and connected to the robot via USB 2.0. The desktop computer and the microcontroller are connected via 802.11b Wireless LAN. Although this configuration is not as complicated as the actual networked home service robots, it has all the features to demonstrate application of the CREAM component framework for developing cross-domain embedded system components.
The application development process in CREAM involves four major steps as shown in Figure 7. The first step is the component construction according to the CREAM component model. This involves describing the SCD profiles of all the components used in the networked home service robots. The CREAM code generator uses the SCD profiles to generate component skeleton code. A component developer then implements the business logic of the robot component in the skeleton code.

The second step is the compiling and packaging of developed components into binary entities according to microcontroller and desktop computer target platforms. The CREAM’s code generator creates the necessary Makefiles and scripts to package the implemented skeleton code into component libraries. Along with the binary component library, the package contains the components SCD, IDLs, header files, and default CPD properties file of the component. This binary entity can then be placed in a component repository and referenced by the SPD profile.

The third step is the assembly of different components to form component assemblies as described in SAD profiles. This step inter-connects the components in the desktop computer node with the components in the microcontroller node of the networked home service robot.

The last step is the deployment of the component assemblies according to the SDD profile as executable entities in the desktop computer and the microcontroller. This step involves grouping the component assemblies into one partition in the desktop computer node and another partition in the controller node. The CREAM’s code-generator uses the SDD profile to finally create an executable entity for each partition.

6. Implementation and Experimental Results

We have implemented the CREAM framework using the standard C++ programming language. The only additional library used in the CREAM framework is the POSIX pthread library for thread creation. This renders the CREAM framework easily portable to any platform.

We have developed and tested the CREAM framework on two OS platforms: Linux (2.6.22 kernel) and Windows XP. On Linux, gcc (4.1.3) compiler, and autoconf (2.61) and automake (1.10) build-toolsets were used. On Windows, Visual Studio 2005 was used to develop the CREAM.

The CREAM code generator is implemented in the Perl scripting language. It uses the perl-libxml2 library to parse the domain profiles. The use of the Perl scripting language makes the code generator a cross-platform tool.

We have compared the CREAM performance characteristics to a socket based raw implementation and the MicoCCM. In the raw implementation, method calls between collocated components were handled through local function calls, and method calls between two partitions were handled through socket communication. This raw implementation allows us to compare the communication performance for best obtainable values. On the other hand, MicoCCM is a popular C++ implementation of CCM by FPX [18] and has been used in many distributed real-time embedded system applications [19].

For performance comparisons, we conducted experiments using three component instances and their interfaces shown in Figure 8. Component C1 invoked the print method with a string of 1000 characters for 1000 times on components C2 and C3. In the CREAM experiment, we configured the experimental system such that two component instances were in one partition and another in a different partition. Similarly, in the MicoCCM experiment, two component instances were in one container that manages component instances in CCM and another in a different container. The component instances within the same partition (CREAM) or container (CCM) share the same address space.

We used two computing hosts with the following configuration for our experiments: Intel Centrino 2.80 GHz running Linux 2.6.22 kernel and having 1 GB of RAM memory. The CREAM component framework made use of the Ice-E [23] communication middleware in these experiments.

We measured inter-component communication time for three scenarios. First, the inter-component communication time for components in the same address space was measured. This measurement was obtained using either of the experimental setups in Figure 8 (a) and (b). Second, the inter-component communication time for components residing in different address spaces, but within the same host was measured. This measurement was performed using the setup in Figure 8 (a). This experiment helps us analyze the performance of underlying mechanisms to achieve remote communication without the overhead of the physical communication channel. Third, the inter-component communication for remote components residing in different hosts was measured using the setup in Figure 8 (b).
interface Printer
{
    void print(string someMessage);
};

(a) Interface used in inter-component communication.

Partition A (Container A)

C1

<<Co-located Components>>

C2

Partition B (Container B)

C3

Partition A and B deployed on the same Computer 1

(b) Remotely located components in the same computer host.

Partition A (Container A)

C1

<<Co-located Components>>

C2

Partition B (Container B)

C3

Partition A deployed on Computer 1

Partition B deployed on Computer 2

(c) Remotely located components in different computer host.

Table 2. Inter-component Communication time

<table>
<thead>
<tr>
<th></th>
<th>CREAM</th>
<th>Raw</th>
<th>MicoCCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collocated components</td>
<td>1.43 µs</td>
<td>1.10 µs</td>
<td>2.74 µs</td>
</tr>
<tr>
<td>in the same address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>space</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remotely located</td>
<td>43.4 µs</td>
<td>37.2 µs</td>
<td>76.5 µs</td>
</tr>
<tr>
<td>components in the same</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>host</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remotely located</td>
<td>351 µs</td>
<td>332 µs</td>
<td>387 µs</td>
</tr>
<tr>
<td>components in different</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hosts</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As shown in Table 2, for collocated components in the same address space, the communication overhead of CREAM compared to the raw implementation is 30% whereas MicoCCM causes 149%. For remotely located components in the same and different hosts, the overhead of CREAM is 16.6% and 5.7%, respectively. Compared to this, the overhead of MicoCCM was 106% and 16.6%, respectively. This shows that the communication performance of CREAM exceeds that of MicoCCM and is close to the best obtainable performance in raw implementation.

These fast inter-component communications in the CREAM component framework are achieved through mapping of collocated inter-component method calls to local function calls and having minimal requirements on the communication middleware.

We also measured the total run-time memory used by the CREAM and MicoCCM component frameworks for the experimental setup of Figure 8 (c).

Table 3. CREAM Processes and Memory Use

<table>
<thead>
<tr>
<th></th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainExecutionManager</td>
<td>48</td>
</tr>
<tr>
<td>Partition A (on computer 1)</td>
<td>48</td>
</tr>
<tr>
<td>Partition B (on computer 2)</td>
<td>40</td>
</tr>
<tr>
<td>Total</td>
<td>128</td>
</tr>
</tbody>
</table>

Table 4. MicoCCM Processes and Memory Use

<table>
<thead>
<tr>
<th></th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naming-service</td>
<td>24</td>
</tr>
<tr>
<td>mico-ccmd (daemon on computer 1)</td>
<td>48</td>
</tr>
<tr>
<td>component-server (on computer 1)</td>
<td>56</td>
</tr>
<tr>
<td>mico-ccmd (daemon on computer 2)</td>
<td>40</td>
</tr>
<tr>
<td>component-server (on computer 2)</td>
<td>48</td>
</tr>
<tr>
<td>Total</td>
<td>216</td>
</tr>
</tbody>
</table>

Table 3 shows that the CREAM component framework makes use of three OS processes: DomainExecutionManager, PartitionA and PartitionB to implement the experimental system of Figure 8 (c) on two hosts. On the other hand, Table 4 shows that MicoCCM uses five OS processes: Naming-service, two mico-ccmd processes and two component-server processes for the same experimental setup. As can be inferred from Tables 3 and 4, CREAM uses 40.7% less memory than MicoCCM. This is achieved through the removal of naming-service and dynamic-composition features of CCM which are rarely needed for an embedded application.
7. Conclusion

In this paper, we have proposed the CREAM as a new generic component framework for distributed embedded systems. We have identified the design requirements of a component framework that meets the challenges of distributed cross-domain applications. We have designed and implemented the CREAM component framework, which is lightweight on memory usage, has integrated task-execution model, efficiently handles inter-component communication, and supports distributed processing in a communication middleware transparent manner.

The CREAM component framework was evaluated and compared to a raw socket-based implementation and the MicoCCM. The experimental results showed that the CREAM’s inter-component communication time is comparable to that of the raw implementation. Moreover, the CREAM achieves faster inter-component communication and less total run-time memory usage than the MicoCCM.

There are several possible future research directions to improve the CREAM component framework. First, the CREAM component framework can be extended to support other desired features such as quality-of-service (QoS) guarantees for inter-component communication. Secondly, the CREAM component framework can be made more user-friendly through development of GUI toolsets and integrating with existing IDE such as the Eclipse [27].

References

[27] Eclipse - an open development platform: http://www.eclipse.org