Compadres: A Lightweight Component Middleware Framework for Composing Distributed Real-time Embedded Systems with Real-time Java[⋆]

Jie Hu, Shruti Gorappa, Juan A. Colmenares[⋆⋆], and Raymond Klefstad
{jieh, sgorappa, jcolmena, klefstad}@uci.edu

Department of Electrical Engineering and Computer Science
University of California, Irvine, CA 92697, USA

Abstract. Component frameworks simplify development of enterprise systems and enable code reuse, but most frameworks are unpredictable and hence unsuitable for embedded or real-time systems. Similarly, Java is increasingly being used to build embedded system software because of its portability and ease of use. The Real-Time Specification for Java (RTSJ) reduces the unpredictability in Java execution times by eliminating the need for a garbage collector. However, it introduces programming complexity that makes it difficult to build non-trivial applications. To bring the advantages of Java component development to DRE systems, while simultaneously simplifying the use of RTSJ, therefore, we have developed a new lightweight component model for RTSJ called *Compadres*. *Compadres* offers the following advantages: 1) Simple component definition in Java that abstracts away RTSJ memory management complexity; 2) System assembly from components by connecting ports that communicate through strongly-typed objects; 3) The *Compadres* compiler that automatically generates the scoped memory architecture for components, while the component framework handles communication between the components. To validate this work, we construct a non-trivial example application using the component framework, a simple real-time CORBA implementation. We then analyze the performance and efficiency of our component example versus a non-component example, RTZen. Our measurements show that our Compadres example built with components incurs only minor time overhead as compared to a comparable hand-coded example.

KEY WORDS: Real-time Java, RTSJ, component framework, middleware

[⋆] This material is based upon work supported by the National Science Foundation under Grant No. 0410218. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

[⋆⋆] Also with the Applied Computing Institute, College of Engineering, University of Zulia

1 Introduction

Distributed, real-time, embedded (DRE) systems pose significant challenges for software developers. As embedded systems, they typically have limited processing power and memory; as real-time systems, they have timing and predictability constraints; and as distributed systems, they must be able to communicate across heterogeneous platforms. Developing software that meets all of these constraints is costly and time-consuming: each application is typically custom-coded from scratch using C/C++ programming language. The limited space and processing power of DRE systems requires lean, specialized custom code, while the threading and memory control needed for real-time requirements requires highly-skilled programming.

By contrast, two existing technologies currently ease and speed development of non-DRE enterprise systems. First, component technology provides effective reusability for software applications for enterprise systems, allowing assembly of pre-coded, pre-tested subsystems into systems, saving both time and money for system development. Second, Java facilitates software development because not only is it relatively easy to use, eliminating complex memory management, but it also offers a large programmer base, library support, platform independence, and a better memory model that minimizes problems with buffer overruns and illegal references.

Unfortunately, the advantages of both component frameworks and the Java programming language have been unavailable to DRE systems developers. Current component frameworks incur too much memory overhead, decrease efficiency, and fail to support the real-time predictability requirements needed for DRE systems. Furthermore, Java cannot be used for real-time systems because its under-specified thread semantics and automatic memory management cause unpredictability.

In general, two complementary approaches have been proposed to reduce the unpredictability of Java– 1) the Real-Time Specification for Java (RTSJ) [1] and 2) real-time garbage collection [2, 3]. Real-time garbage collectors (RTGCs) can be unsuitable for use in *hard* real-time systems because they cause an inherent minimum latency and large execution overhead [4]. It is also necessary to accurately predict parameters such as average and maximum allocation rates when using a RTGC. On the other hand, The RTSJ adds memory and thread models that enable predictability for real-time systems, but loses much of the ease of programming of Java. We have therefore developed a lightweight component model for RTSJ, called *Compadres*, that brings the advantages of Java component development to DRE systems, while simultaneously simplifying the use of RTSJ and providing real-time predictability. *Compadres* components are fine-grained object-oriented software artifacts that communicate via *ports*; applications can be developed by connecting these ports.

Compadres achieves ease of use, ease of testing, and a high level of reusability in the following ways:

– **Simple component definition in Java:** *Compadres* is a simple component model that hides the programming difficulty of the RTSJ scoped memory and threading model and yet is powerful enough to define most real-time systems.[1] The component implementations are separated from their threading model, allowing developers to implement the business code for the components using Java with some restrictions but without having to deal with the RTSJ memory management rules. Furthermore, any component may be used as an application process. This feature can also be used to convert a component into a stand-alone application. At a higher level, applications may be distributed in a network.

– **Automatic generation of scoped memory architecture:** The *Compadres* compiler (henceforth compiler) processes a user-defined component composition language file to generate the scoped memory architecture required for the application to run based on the RTSJ scope access rules. The compiler thus abstracts away the RTSJ memory management code from the user.
– **Simplified system assembly through composition of components:** *Compadres* provides hierarchical composition and extension; i.e., components may be incrementally composed into larger components. This feature facilitates incremental testing of components as well as final system testing.
– **Simple communication model:** Components are composed by connecting ports that communicate through strongly-typed objects, providing semantic checking at compile time.

The remainder of this paper is organized as follows: Section 2 presents the *Compadres* component model and describes how it abstracts RTSJ programming challenges. Section 3, presents a simple Real-time CORBA ORB built using *Compadres* and compare its performance to RTZen, our Real-time CORBA ORB for RTSJ [5]. Section 4 presents the related work and section 5 presents conclusions and future work.

2 The *Compadres* Component Framework for RTSJ

The process of developing an RTSJ application using *Compadres* is divided into two phases (see Fig. 1):

1. **Component Definition**: In this phase, the application programmer defines the components and their ports in an XML file following the *Component Definition Language* (CDL). The CDL file is compiled to generate the skeletons of the implementation classes of the components and the message handlers associated with the components' `In` ports. The programmer adds the implementation of the component and message handler classes using plain Java.
2. **Component Composition**: In this phase, composite components and connections among components are specified in an XML file according to the *Component Composition Language* (CCL) to form the application. The programmer uses the *Compadres* compiler to validate the CCL file and generate the RTSJ glue code needed to run the main application.

[1] The components may also use an RTSJ-safe library such as Javalution (http://javolution.org/).

Finally, the Java compiler is used for compiling the implementation classes of components and message handlers along with the generated RTSJ glue code to build the RTSJ application. The rest of the section describes in detail the phases for developing a *Compadres*-based RTSJ application.

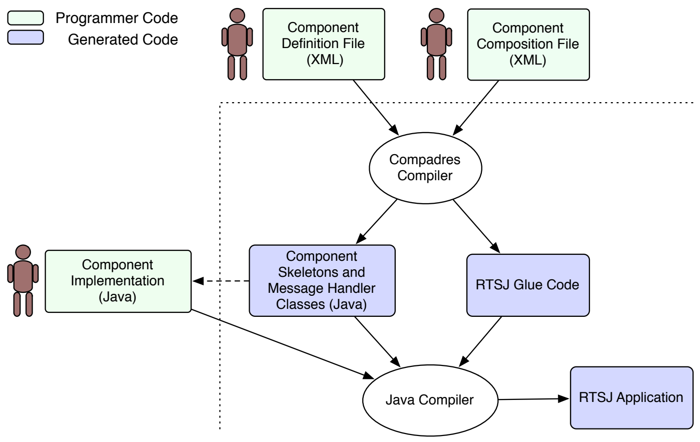
Programmer Code
Generated Code
Component Definition File (XML)
Component Composition File (XML)
Compadres Compiler
Component Implementation (Java)
Component Skeletons and Message Handler Classes (Java)
RTSJ Glue Code
Java Compiler
RTSJ Application

Fig. 1: Generation of a real-time Java application using the *Compadres* framework

2.1 Component Definition

The application programmer writes the CDL file in XML to define the components used in the application as well as the ports of each component. An example of a CDL file is presented in Listing 1.1. The definition of a component comprises the name of the component and the set of its ports. The definition of a port includes its name, its type, and the Java type of the message that is communicated through the port. Ports may be *input ports*, which receive messages or *output ports*, which send messages. Thus, the type of a port may be set to `In` or `Out` in the CDL file; the direction is specified in relation to the component itself. In particular, the port types and message types specified in the CDL file will be used to by the *Compadres* compiler to validate the CCL file.

The *Compadres* compiler parses the CCL file and generates the following Java skeleton classes for each component: 1) a component class and 2) one message handler class per `In` port. The component skeleton class extends the `Component` class, which contains the `addInPort()`, `addOutPort()`, and `_start()` methods. The `addInPort()` method associates a message handler class with the corresponding `In` port, and the `_start()` method is an empty method that may be implemented by the programmer to initialize the component. Each message handler skeleton class extends the `MessageHandler` class, which contains the `process()` method. The `process()` method accept one message object (of any

Java datatype) as a parameter. When a message is sent to an In port, the corresponding process() method is called to handle the incoming message. The process() method of each message handler skeleton class is initially empty, so that the application programmer needs to implement it. The user may allocate objects using new in the implementation of component and message handler classes but does not need to determine which RTSJ memory region to use.

```
<Component>
   <ComponentName>Server</ComponentName>
      <Port>
      <PortName>DataOut</PortName>
      <PortType>Out</PortType>
      <MessageType>String</MessageType>
    </Port>
    <Port>
      <PortName>DataIn</PortName>
      <PortType>In</PortType>
      <MessageType>CustomType</MessageType>
    </Port>
</Component>

<Component>
    <ComponentName>Calculator</ComponentName>
    ....
</Component>
```

Listing 1.1: Component Definition Language file

2.2 Component Composition

A vital characteristic of *Compadres* components is that they are hierarchically *composable*. The Component Composition Language (CCL) file, written in XML format, allows programmers to construct an application from components. The CCL file is written once per application and defines the connections between components, thread priorities, and thread assignment to the components. The CCL decouples the definition of the individual components from their configuration and interaction, thereby enabling component reuse. The component implementations themselves are unaware of the runtime properties; the compiler handles the assignment of the components to memory regions and threads.

Connecting Components via Ports: Components are composed by connecting their appropriate ports, and the port connections are defined in the CCL file; Out ports must be connected to In ports, and the message types (obtained from the CDL file) must match exactly. However, *adapter* components may be introduced to connect two non-matching types.

Connection of ports must follow RTSJ scoping rules to ensure that the compiler can map these components into RTSJ scoped memory areas. In order to enforce the scoping rules, we designate ports as Internal or External in the CCL file. Hierarchically, components created inside another component are the children of that component; two or more components inside the same component are siblings of each other. Internal ports communicate a parent component with its child components; `external` ports communicate a child component with its parent or sibling components. Only sibling components can see the external ports of each other. Components can only exchange messages between their siblings and parent via external ports and between their own children via internal ports. Therefore, only the following port connections are allowed in the *Compadres* model: 1) internal port of the parent component to external port of the child component, 2) external ports of sibling components. When a component sends data from one of its `Out` ports, it relays the data to the `In` port(s) connected to it. When data arrives at an `In` port, the component that owns the port processes the data immediately in a new execution context and may generate outputs at its other ports.

A simple example of a hierarchical composition of five components is illustrated in Fig. 2. The components are constructed in three levels of scoped memory. Component A is the level-1 parent component. It has two child components, B and C, and is connected to them via internal ports. The component C in turn has two nested components, D and E.

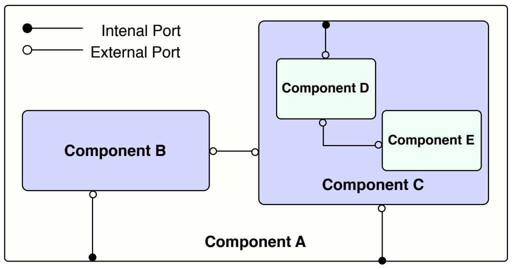

Fig. 2: Hierarchical composition of components via internal and external ports

The CCL file (example in Listing 1.2) contains application information under the following XML tags:

- `ApplicationName`: is the name of the application class to be generated.
- `Component`: specifies each component used in the application. This tag contains tags that indicate the name of the component class, name of the instance, its type (immortal or scoped), and its nesting level if the component is of type scoped. `Component` tags are nested to represent the parent-child relationship among components.
- `Connection`: contained in a `Component` tag, includes the list of ports of the component and their links with ports of other components.
- `Port`: represents a port of a component; it includes the name of the port and its attributes.
- `PortAttributes`: specifies the threading strategy (shared or dedicated), size of threadpool, and buffer size of each `In` port.
- `Link`: represents the end-point and type (internal or external) of a link between two component ports.

– `RTSJAttributes`: includes RTSJ memory pool attributes such as memory size in bytes and scope pool sizes.

```
<Application>
  <ApplicationName>MyApp</ApplicationName>
  <Component>
    <InstanceName>MyServer</InstanceName>
    <ClassName>Server</ClassName>
    <ComponentType>Immortal</ComponentType>
    <Connection>
      <!-- Define Ports -->
      <Port>
        <PortName>DataIn</PortName>
        <PortAttributes>
          <BufferSize>5</BufferSize>
          <Threadpool>Shared</Threadpool>
          <MinThreadpoolSize>2</MinThreadpoolSize>
          <MaxThreadpoolSize>10</MaxThreadpoolSize>
        </PortAttributes>
        <!-- Define connection to Out port of child-->
        <Link>
          <PortType>Internal</PortType>
          <ToComponent>Calculator</ToComponent>
          <ToPort>DataOut</ToPort>
        </Link>
      </Port>
    </Connection>
    <Component>
      <InstanceName>MyCalculator</InstanceName>
      <ClassName>Calculator</ClassName>
      <ComponentType>Scoped</ComponentType>
      <ScopeLevel>1</ScopeLevel>
      <Connection>
        . . . .
      </Connection>
    </Component>
    . . . .
  </Component>

  <RTSJAttributes>
    <ImmortalSize>400000</ImmortalSize>
    <ScopedPool>
      <ScopeLevel>1</ScopeLevel>
      <ScopeSize>200000</ScopeSize>
      <PoolSize>3</PoolSize>
      . . . .
    </ScopedPool>
  </RTSJAttributes>
</Application>
```

Listing 1.2: Component Connection Language file

Any component, whether simple or composite, can be made into an application using the CCL file. In this phase the compiler serves two purposes: validation and glue code generation. First, it uses the CDL file to validate the CCL file for connections (to ensure that `Out(In)` ports are connected to `In(Out)` ports and there are no loops), RTSJ access rules, and message type matching. The connections are checked to ensure that each component's `internal` port is connected to the `external` ports of its children, and that the `external` ports of siblings are connected. This process ensures that message passing will not violate RTSJ memory access rules. The code generation tasks of the compiler in the component composition phase are: 1) allocating memory to components by analyzing the specified memory needs, 2) defining the RTSJ memory structure for the components, 3) generating glue code to create component instances and for component communication, and 4) generating the main application class that includes an empty `_start()` method that the programmer will need to implement.

In order to implement component ports, the compiler generates the code for managing the message buffer and threadpool associated with each `In` port, and the RTSJ glue code for connecting them to the `MessageHandler` of that port. The incoming messages at an `In` port are enqueued in its message buffer. The size of the message buffer is specified in the CCL file. Messages are assigned a priority in the `send()` method of the `Out` port. When a message arrives at an `In` port, a thread from the threadpool is assigned the priority of the incoming message and then calls the `process()` method of the corresponding `MessageHandler`. The number of threads in the pool is initialized to `MinThreadpoolSize` value and can go up to the `MaxThreadpoolSize` value, with both values specified in the CCL file. If these values are 0, the calling thread executes the `process()` method of the `In` port synchronously.

Structure of *Compadres* Component Applications: *Compadres* is a loosely coupled component model because a component can be 1) individually implemented and tested independent of the rest of the system, 2) incrementally deployed in a system, and 3) easily extracted from a system for reuse. Several components can be encapsulated to compose a new component. Composition and communication between components must follow the RTSJ memory access rules. Next, we briefly discuss the RTSJ memory structure and the restrictions it imposes on programming, and describe how the *Compadres* framework serves as an abstraction over the RTSJ memory model.

RTSJ Memory Structure [1]: An application's memory structure is constrained by the rules that govern memory access among the three types of memory regions defined in the RTSJ—heap, scoped, and immortal. Of these, the heap memory is garbage collected; therefore, *Compadres* components support only two types of RTSJ memory, scoped and immortal. *Scoped memory* is a region with a limited lifetime, which ends when there are no more threads executing in the region. Scoped memory can be of two types, linear-time, or variable-time: our memory model only uses linear-time or `LTScopedMemory`, which is allocated in a time proportional to its size and therefore predictable. `ImmortalMemory` is a fixed-sized area whose lifetime is the same as that of the JVM. Objects allocated in immortal memory, however, will never be garbage collected during the lifetime of the application. Scoped memory areas may be nested, producing a scoping structure called a scope stack. Since multiple memory areas can be entered from an existing memory area, this scope stack can form a tree-like structure. One key relationship is as follows: if scope B is entered from scope A, then A is considered the *parent* of B and B, the *child* of A (see Fig. 2). Two rules govern memory access among scopes. Code within a given scoped memory area X can reference memory in another region Y only if it can be guaranteed that the lifetime of the memory region Y is at least as long as that of the first region X. This lifetime can be guaranteed only if the requested object resides in an ancestor region (e.g., a parent or grandparent), immortal, or heap memory. Another important constraint is that a memory region can have only one parent, thereby preventing cycles in the scope stack (the *single parent rule*). The implication is that a single scope cannot have two or more threads from different parent scopes enter it. An important consequence of this restriction on scoping structure is that a real-time thread executing in a given region cannot access memory residing in a sibling region and vice versa. In the event that real-time threads in these two regions need to coordinate to perform some task, they will need to do so through memory stored in a common ancestor region. For example, in Fig. 3, a real-time thread in scope C cannot access scope B. They can only coordinate via objects stored in A or immortal memory. Table 1 depicts the complete access rules among scopes in Fig. 3.

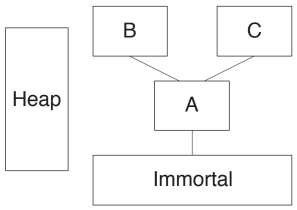

Fig. 3: Nested scopes

Table 1: Access rules for Fig. 3 assuming real-time threads are used. Note that if no-heap real-time threads are used, no references to the heap are permitted.

	to Heap	to Immortal	to A	to B	to C
from Heap	–	yes	no	no	no
from Immortal	yes	–	no	no	no
from A	yes	yes	–	no	no
from B	yes	yes	yes	–	no
from C	yes	yes	yes	no	–

Mapping Components to RTSJ Scopes: Each *Compadres* component is created in a separate (scoped or immortal) memory area. The RTSJ memory scopes in *Compadres* are hierarchical; thus, so are components– they may be nested inside other components. The outer memory area is the parent of the nested memory areas. The nested architecture follows the single parent rule, which ensures that each component has only one parent. The scope in which a component should be placed is based on 1) the lifetime of the component, and 2) its interaction with other components. The following rule determines the lifetime of each scope memory of component: *child components have a shorter lifetime than their parent since they are created in a scoped memory area with a depth greater than that of the parent component*. Therefore, scoped memory components that are triggered by other components and have shorter lifetime should be instantiated as their children.

One method to detect scoped memory regions for allocating objects from Java programs is to generate a directed acyclic graph based on object lifetimes and references and assign RTSJ memory scopes based on the depth of the object in the graph [6]. We use a similar approach, but at the level of components, rather than objects. As the lifetimes of scoped components are different, the scoped memory areas are not bound to components at compile-time, but at runtime. This memory can be reused after the scoped component is reclaimed. The *Compadres* component framework allows component instantiation at application runtime. Components are created in `LTScopedMemory`. Further optimization of component instantiation can be achieved by creating pools of scoped memory areas in immortal memory and reusing these areas at runtime. The size and number of scopes in the pools can be assigned in the CCL file under the `RTSJAttributes` tag (Listing 1.2).

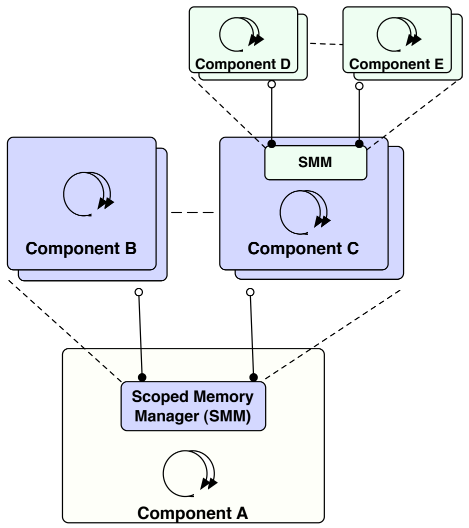

Fig. 4: Parent components communicate with their child components via scoped memory managers (SMMs).

Component Communication via Scoped Memory Managers: References to objects in different components are constrained by the RTSJ memory access rules described previously, but directly exchanging messages across ports may violate these restrictions. We solve this problem by using a *Scoped Memory Manager* (SMM), illustrated in Fig. 4. The SMM is used to connect an internal port of a component to the external port of its child component. In our framework, each parent component needs only one SMM to communicate with all its children. Each SMM of a parent component maintains a virtual proxy for every child component. Upon receiving a message intended for a child component, the SMM checks the proxies for the existing component or, if none are found, creates a new scoped memory component which should receive the message. After the messages are processed by the component, the scoped memory objects are reclaimed. To keep a child alive, the parent component requests a new child scoped memory component; and a `handle` is returned to the parent. The parent can kill the temporary component by calling `disconnect()` with the `handle`. This mechanism is implemented using the wedge thread pattern [7].

One of the most difficult aspects of application development using RTSJ is to implement the mechanisms to pass messages between objects in different scoped memory areas. We have identified three mechanisms to handle cross-scope method invocation and message passing:

– ***Serialization***: The object is serialized and copied to a memory area that is accessible by the other scoped memory component.
– ***Shared Object*** [7]: The object shared by the components is created in a common ancestor memory area. Users need to identify the common ancestor memory area of the two child components and create the shared object in that memory area.
– ***Handoff Pattern*** [7]: A thread created in the source memory can access the destination memory through the memory area of their common ancestor.

The overhead of serialization causes it to be much less efficient than the handoff pattern. However, using the handoff pattern requires that developers know the scoped memory structure of applications. It also results in the component code becoming tightly coupled and difficult to reuse. The shared object approach is an efficient method but may lead to memory leaks if not implemented correctly. Moreover, users need to determine the common ancestor memory area for two threads, which involves tracing the threads at design time. Based on experience, we have found the shared object approach to be the most efficient and easiest to generate as part of the *Compadres* framework. Thereby, the *Compadres* framework reduces the programming effort by handling inter-component message passing transparently. This feature enables programmers to implement their logic inside each component using regular Java and hides the complexity of RTSJ scope access rules from them. The SMM of the parent component contains the message buffer of each external port of its child components. This message buffer serves as the shared object; therefore, the parents and its children can reference the messages from the buffer.

The *Compadres* framework creates a message pool per message type in the parent component's SMM (allocated in the parent component's memory area). To send a message, programmers get a message object from the pool by calling `getMessage()`, set the message data, and then send the message through the port via `send()`. The message is returned to the pool after it is processed by the receiver. This mechanism reuses objects, thus preventing the memory areas of parent components from being exhausted. The only restriction is that message objects should be RTSJ-safe – all the data contained in a message object must be allocated in the same memory area. Hence, *Compadres* is less restrictive than programming profiles such as the Ravenscar [8], which strictly disallow many features such as dynamic task allocation and dynamic priority assignment.

The shared object mechanism is inefficient in the case of message passing between components that do not have the same parent component but have a common ancestor, due to additional and expensive message copying. To optimize this type of communication, we relay the messages from the ancestral memory area using *shadow ports*. The *Compadres* framework provides a *shadow port* for a scoped component to communicate directly with its non-immediate ancestors without having to generate a message for its parent. For example, consider a three-level component structure in which component C needs to communicate with its grandparent A, but not with its parent B, illustrated in Fig. 5. In this case, programmers specifies the direct connection between components C and A. The compiler detects the need for a shadow port and generates the port connection that allows direct communication between C and A. The data structure for a regular port at B will not be generated and the message pool and buffer are created only in the memory area of component A.

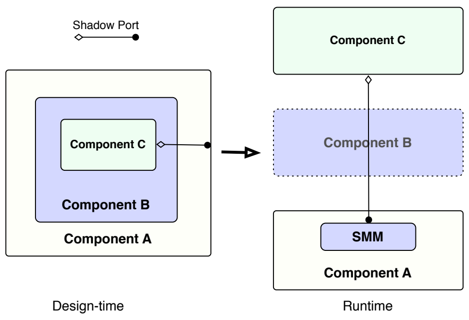

Fig. 5: The *shadow port* allows a child component to communicate with its ancestor directly rather than via its parent.

3 Performance Results

We built and tested two examples using *Compadres*. The first example was designed to test *Compadres'* pure overhead for a simple round-trip co-located client-server request-reply. The second example was designed to test *Compadres'* usefulness in a more complete, real-world example of an ORB.

3.1 Overhead of the Framework

We first implemented a simple co-located client-server example and measured the round-trip time to send a client request message and receive a server reply message. The *Compadres* implementation of the example is illustrated in Fig. 6, and the programmer code is shown in Listings 7 and 8.

At application startup, an instance of an `ImmortalComponent` (`IMC`) and `SMM` are created in immortal memory, and the `_start()` method is called. The `IMC` creates an instance of a scoped memory component (`Client`) in a level-1 scoped memory region, `Client` ports are added, and the message handler `P2_Handler` is associated with the `In` port, `P2`. IMM sends a trigger message via `P1` instructing the `Client` to send a request message to the `Server`. When port `P2` receives this message, the `process()` method of `P2_Handler` is called and sends out a request message to the server via `Out` port `P3`. Since `Client` and `Server` are defined as siblings in the CCL file, the SMM creates the server component using `connect()` in sibling scoped memory region and sends the request message to the `Server`. This invokes the message handler for `In` port `P4`, which processes the request and sends a reply via `P5`. The reply message is received by the message buffer in `SMM` and routed to `Client` via `P6`.

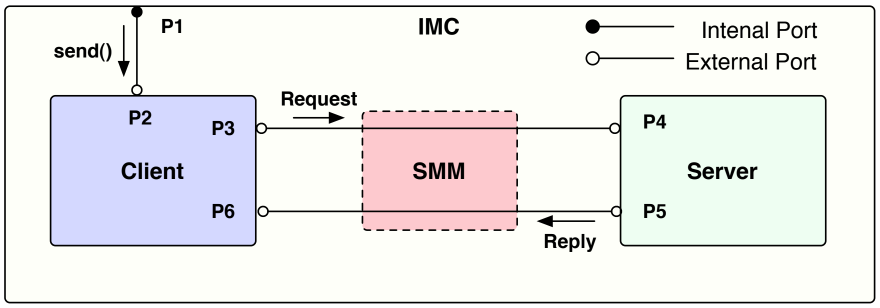

Fig. 6: The client-server scoped memory example

Testing Environment. This first example was tested on three platforms:

1. a non-real-time Pentium system: a 865 MHz Pentium III processor (Coppermine, 256KB Cache) with 512MB PC133 ECC SDRAM, running TimeSys Linux GPL 4.1 based on the Linux kernel 2.4.21, with the non-real-time Java Virtual Machine (JVM) Sun JDK 1.4 default garbage collector;
2. a real-time Pentium system: the same Pentium and OS above, with the RTSJ RI from TimeSys; and
3. a real-time Sun system: a Sun-Fire-V210 with a 1064 MHz UltraSPARC processor, running SunOS 5.0, with Sun's Mackinac[9].

```
public class MyInteger {
    public int value = 0;
}

public class ImmortalComponent extends
      Component {
    // Compadres framework creates
    // ImmortalComponent in immortal memory
    public SMM smm = new SMM(...);
    //Define out-port
    // addOutPort(out-port name, SMM object,
    //   msg type, destination in-port name)
    public OutPort p1 = addOutPort("P1", smm,
           MyInteger.class, "MyClient_P2");
    public void _start(){
        // Get a message from the pool and
        // send it to the client component
        MyInteger m = (MyInteger) p1.
              getMessage();
        // Send trigger msg with priority 2
        p1.send(m, 2);
    }
}

public class Client extends Component {
    // addInPort(in-port name, SMM object,
    //   msg type, buffer size, threadpool
    //   strategy, min pool size, max pool
    //   size, message handler class)
    public InPort p2 = addInPort("P2", imc.
          smm, MyInteger.class, 10, 0, 1, 5,
          P2_MessageHandler.class);
    public OutPort p3 = addOutPort("P3",imc.
          smm,MyInteger.class,"MyServer_P3");
    public InPort p6 = addInPort("P6", imc.
          smm, MyInteger.class, 20, 0, 1, 5,
          P6_MessageHandler.class);
    public void _start() {
    }
}
```

Fig. 7: Implementation classes of immortal and client components

```
public class Server extends Component {
    public InPort p4 = addInPort("P4", imc.
          smm, MyInteger.class, 20, 0, 1, 5,
          P4_MessageHandler.class);
    public OutPort p5 = addOutPort("P5",imc.
          smm,MyInteger.class,"MyClient_P6");
    public void _start(){}
}

public class P2_MessageHandler extends
      MessageHandler{
   public void process(Object data, SMM smm){
       //Get reference to out-port
       //   connected to server
       OutPort p3 = smm.getOutPort("P3");
      MyInteger i=(MyInteger)p3.getMessage();
       i.value = 3;
       // take timestamp ts_0
       . . .
       p3.send(i, 3);
    }
}
public class P4_MessageHandler extends
      MessageHandler {
   public void process(Object data, SMM smm){
       // Get reference to out-port
       //    connected to client
       OutPort p5 = smm.getOutPort("P5");
      MyInteger i=(MyInteger)p5.getMessage();
       i.value = 4;
       p5.send(i, 3);
    }
}
public class P6_MessageHandler extends
      MessageHandler {
   public void process(Object data, SMM smm){
        // take timestamp ts_1
        . . .
    }
}
```

Fig. 8: Implementation classes of server component and message handlers

Measurements. For all tests, measurements were based on steady state observations, where the system is run until the transitory effects of cold starts are eliminated before collecting the measured observations. We used the maximum of 10,000 observations as an estimate of a system's "worst case," a critical measurement for real-time systems that must be designed with the assumption that the system will always deliver the worst possible performance. A sample size at least this large was necessary to observe a reasonable estimate for the maximum latency because the maximum values tended to be extremely low-probability events. The range of the observations, i.e., jitter, was used as another measure of a system's predictability.

Results. Our framework is reasonably predictable on both Mackinac and Timesys RI, with jitter of 92μs and 55μs respectively, well within the 10ms described as typically acceptable for distributed real-time systems [10] The distribution of the round-trip latency values indicating maximum and minimum bounds is shown in Fig. 9, while Table 2 lists the jitter for each platform. The jitter on JDK1.4 is large, most likely caused by the garbage collector preempting the application threads. The jitter on Mackinac is larger than the jitter on Timesys RI; Timesys RI was installed on a real-time OS and Mackinac on SunOS 5.10. Although SunOS 5.10 provides some RT scheduling strategies, it is a non-real-time OS, allowing some system threads to preempt the application threads.

Platform	**Median** (μs)	**Jitter** (μs)
Mackinac	99.58	92.17
RI	114.0	55.0
JDK1.4	56.43	317.27

Table 2: Median and jitter of round-trip times on different platforms

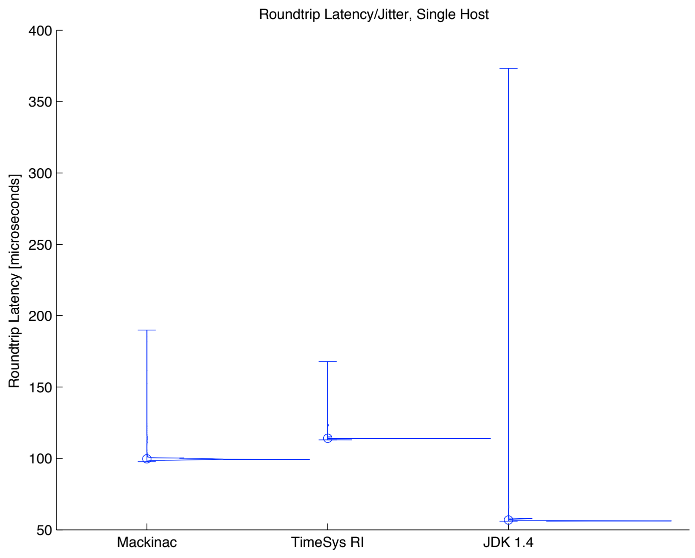

Fig. 9: Comparison of round-trip times of simple message passing

3.2 Constructing a Real-World Example: RT-CORBA

We use previous experience in building RTZen [5], an RTSJ implementation of RT-CORBA, to construct a simple RT-CORBA ORB using *Compadres*. CORBA exposes the ability to create and destroy CORBA components, such as `POA` and `Transport`, to the application. RTZen enables this by assigning scoped memory areas to these components. When the user creates (destroys) one of these components, the associated memory scope is created (freed). The design of the *Compadres* ORB is based on RTZen and is illustrated in Fig. 10. The *Compadres* CORBA client is a 3-level scoped structure. The level-1 memory contains an `ORB` component, which is allocated from immortal memory. Inside the `ORB` component is the `Transport` component, created in the level-2 scoped memory when a request message is received from the `ORB` component. When the client application makes a remote method call to the server, the `ORB` sends a message to the previously created `Transport` component. Upon receiving the message, the `Transport` creates a `MessageProcessing` component to generate the request in the level-3 scoped memory. After the `MessageProcessing` component obtains the reply message from the server, it sends the result back to client application and destroys itself.

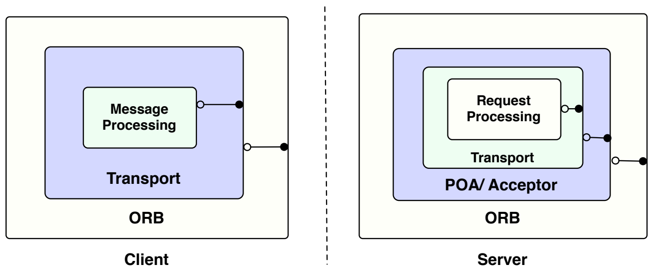

Fig. 10: Component structure of the *Compadres* ORB

The *Compadres* CORBA server is a 4-level scoped structure. Similar to the client, the server-side application creates an `ORB` component in the level-1 immortal memory. A `POA/Acceptor` component is created by the `ORB` component in level-2 scoped memory. The `POA/Acceptor` component listens to and waits for client request messages. Once a client request message comes in, the `POA` component creates a `Transport` component in level-3 scoped memory to wait for client request messages. Once a message is received, a `RequestProcessing` component is created to process the client request in level-4 scoped memory. After processing the request and sending the reply back to the client, this component is destroyed.

With the hierarchical model of *Compadres*, it is easy for us to define and reuse components for the modules of CORBA. Although the `Transport` components of `Client` and `Server` are located in different memory levels and connected to different data processing components, we can reuse the `Transport` component at both the `Client` and `Server`. In addition, binding memory area with components at design time makes the memory hierarchical structure clearer and easier to maintain. Finally, the lifetime of each component matches the lifetime of each CORBA module.

3.3 Comparison of RT-CORBA with *Compadres* and RTZen

We compared our RT-CORBA ORB implementation using the *Compadres* framework's round-trip latency and jitter on a real-time platform with that of RTZen.[2] Both RTZen and the *Compadres* ORB demos were run on the same real-time platform, Timesys Linux and RI. Moreover, both server-side and client-side were run on single machine connected via loopback network. Since performance varied across different message sizes, we compared the two ORBs for the message size from 32 to 1024 bytes.

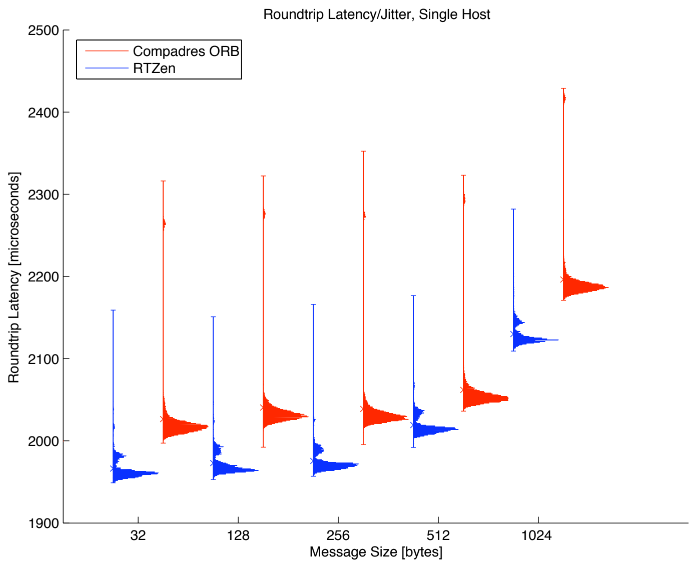

Fig. 11: Comparison of round-trip times of RTZen with the *Compadres* ORB for different message sizes

Both RTZen and the *Compadres* ORB are highly predictable, with the jitter value of 230μs and 300μs respectively. The *Compadres* ORB has a slightly larger jitter, likely caused by the scoped memory managers (SMMs). The distribution

[2] For the purposes of this experiment, the *Compadres* ORB can be considered to be functionally similar to RTZen; it includes marshalling and demarshalling, the most computationally-intensive modules of CORBA. The policy check mechanism has not been implemented, but it is not a computing intensive module and would not incur much overhead.

of the round-trip latency values is illustrated in Fig. 11, with the maximum and minimum bound indicated and with the 'x' representing the median latency. Again, the typical performance and predictability of both RTZen and the *Compadres* ORB are within 10ms, typically acceptable for distributed real-time systems [10]. In general, these jitter values are close to the expected values and highlight the predictability of RTSJ. Hence, our model demonstrates both predictability and low overhead.

4 Related Work

During the last decade several component-based real-time (CBRT) frameworks have been proposed [11–14]. However, none of these CBRT frameworks are based on RTSJ and, therefore, they do not deal with the complexities of the RTSJ's memory model. In the RTSJ domain, our previous work [15] was the first to bring together the ease-of-use of programming in Java with the real-time predictability of RTSJ using a component-based approach. *Compadres* compliments our previous work by providing a component model that supports the notion of ports and enables asynchronous communication. A similar RTSJ-based component framework is presented in [16]. It allows active and passive components to be created in individual scopes, uses a *Connector* component to specify the mode of connection between components, and allows creating and binding sub-components hierarchically.

Scoped Types and Aspects for real-time Java [17] presents an approach to reduce the programming complexity of RTSJ by allocating objects in scopes based on their types. This rule enables static checking and ensures that an assignment does not breach the program structure. However, it requires making minor changes to the virtual machine and uses aspects to separate real-time concerns from the Java code. Reflexes [18] is a an alterative to the RTSJ model; the authors use Java annotation to specify the object type as stable or dynamic, which allows the detection of illegal memory reference at compile time and eliminating runtime memory checks. However, it is very restrictive– it requires assigning one thread per reflex, prevents object reference across reflexes, and does not allow for memory hierarchy.

5 Conclusion

The RTSJ brings real-time performance to Java applications, but presents programming difficulties due to its memory model. We have presented *Compadres*, a component model that abstracts away the programming difficulty of RTSJ, while leveraging the advantages of component-based programming. In its current state, it provides predictability, RTSJ specification compliance, and reduces programming complexity. It provides a solid foundation for further research into implementations of real-time applications based on Java. Future work includes performance optimization of the component framework, developing a graphical user interface for connecting components, and code generation for transparently handling remote communication over a network.

6 Acknowledgements

The authors thank Susan Anderson Klefstad for help with revising the paper and Krishna Raman for his help with implementing the ORB demo. Juan A. Colmenares thanks the University of Zulia (LUZ) for supporting his participation in this research.

References

1. The Real-Time Specification for Java. http://www.rtsj.org.
2. David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press.
3. Sun Microsystems. Sun java real-time system 2.0.
4. F. Pizlo and J. Vitek. An emprical evaluation of memory management alternatives for real-time java. In *27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 35–46, May 2006.
5. Krishna Raman, Yue Zhang, Mark Panahi, Juan A. Colmenares, Raymond Klefstad, and Trevor Harmon. RTZen: highly predictable, Real-Time Java middleware for distributed and embedded systems. In *Proc. of the 6th Int'l Middleware Conference*, December 2005.
6. Morgan Deters and Ron K. Cytron. Automated discovery of scoped memory regions for real-time java. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 25–35, New York, NY, USA, 2002. ACM Press.
7. F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time java scoped memory: Design patterns and semantics. In *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 101–110, May 2004.
8. Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-java: a high-integrity profile for real-time java: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):681–713, 2005.
9. Sun Microsystems. The Real-Time Java platform, June 2004.
10. Peter C. Dibble. *Real-Time Java Platform Programming.* Sun Microsystems Press, 2002.
11. David B. Stewart, Richard A. Volpe, and Pradeep K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23(12), December 1997.
12. K.H. (Kane) Kim. Apis for real-time distributed object programming. *IEEE Computer*, 33(6):72–80, June 2000.
13. Shengquan Wang, Sangig Rho, Zhibin Mai, Riccardo Bettati, and Wei Zhao. Real-time component-based systems. In *Proc. 11th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 428–437, March 2005.

14. Dionisio de Niz, Gaurav Bhatia, and Raj Rajkumar. Model-based development of embedded systems: The sysweaver approach. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 231–242, Washington, DC, USA, 2006. IEEE Computer Society.
15. J. A. Colmenares, S. Gorappa, M. Panahi, and R. Klefstad. A Component Framework for Real-time Java. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium Work-in-Progress (RTAS'06)*, 2006.
16. Jean-Paul Etienne, Julien Cordry, and Samia Bouzefrane. Applying the CBSE paradigm in the real time specification for Java. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 218–226, New York, NY, USA, 2006. ACM Press.
17. Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped types and aspects for real-time java. In *ECOOP*, pages 124–147, 2006.
18. Jesper Honig Spring, Filip Pizlo, Rachid Guerraoui, and Jan Vitek. Reflexes: abstractions for highly responsive systems. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 191–201, New York, NY, USA, 2007. ACM Press.