Information Technology Journal 3 (1): 12-16, 2004
ISSN 1682-6027

Component Based Programming Model for Linux

Muhammad Ali Raza, Muhammad Omer, Sikander Hayat Khiyal and [1]Imran A. Gondal
Department of Computer Science, International Islamic University, Islamabad, Pakistan
[1]Development Manager, Ultimus, Pakistan

Abstract: Contemporary component model development is getting more and more important in software industry. Academic world is spending long time on development and refinement of their component models and rarely do they consider the alternative of not using a bridge. We propose the Component based Programming Model (CBPM) for Linux which removes the usage of bridge while conforming to Component Object Model (COM). CBPM aims to lower software development cost by providing sophisticated facilities for COM based component reuse on Linux. CBPM also focuses on eliminating the bridging overheads for using COM component. CBPM defines a standard for component interoperability, is not dependent on any particular programming language and is extensible.

Key words: Component based development, compatibility, component object model, CORBA, bridging

INTRODUCTION

To support rapid software development, applications are currently constructed from reusable components which form more than 50% of the total software development[1]. Using this approach, the architecture of an application can be considered as a collection of interconnected components. The notion of components enjoys significant interest in the software engineering community. Components are considered to be useful units of code sharing and reuse, as well as useful building blocks of software architectures. While the former view is supported by practical component system[2-4], the latter view appear to be lagging behind. Many applications are designed by using components based on COM standard[2]. The problem arises when we need to use a COM based component on Linux platform, where arises a need of bridging for interoperability of COM based components with OMG based components. The goal of present study is to introduce an alternative of bridging by providing a COM based component development Model, so that Linux applications can use COM based components with out going through the complications of COM and CORBA components interoperability in the form of bridge.

Bridging and its disadvantage: Component Object Model and OMG's CORBA components share many characteristics. Roughly speaking, COM component interfaces are similar to CORBA interfaces. In addition, Component Object Model interface pointers are similar to CORBA object references. There is enough similarity in calling conventions and data types to provide a appropriate level of interworking through mappings. For example, a Windows client can interact with a CORBA object (target object) via a COM based "view object", which provides similar interfaces and methods. Similarly, a client living in the CORBA world interacts with a COM based target object via a CORBA view object. The client treats the view object as if it were the real object. The view object, in turn, transparently provides the communications (for example, Remote Procedure Calls) that connects with the target object. View objects are components in the COM/CORBA (and CORBA/COM) bridge.

A project conducted some investigations of commercially available products for providing interoperability at the protocol and interface level COM and CORBA[5,6]. A set of criteria regarding conformance with the COM/CORBA standard, security, dependability, scalability, platform independence, language independence, openness, configuration, bundling with other products and price were defined. A template was created before the products were investigated. Products from BEA, Brokat, O3sis IT AG, ExperSoft, ICL, Inprise, Hitachi, Rogue Wave, Visual Edge and IONA were surveyed. A number of these were selected for practical tests, but the tests showed that in general, getting code to work that was developed for other bridges proved to be very difficult. Compiler compatibility issues, problems locating COM objects, poor quality documentation and technical support all contributed to the problem. It can be

Corresponding Author: Muhammad Omer, Department of Computer Science, International Islamic University, Islamabad, Pakistan

said that the development cycle is long and complex; also the error messages are few and cryptic. Overall the experiments, which when run showed that the technology at the time was still quite immature. The overall conclusion from this experiment was that bridging is not a trivial and straightforward task.

MATERIALS AND METHODS

There are at least two strong arguments for employing component based programming: Firstly around the world there are a number of software modules which offer services, therefore reusing them is desirable in order to facilitate the software development process[7-8]. Secondly Programming using component technology is more effective for several reasons: it eliminates debugging of the reused parts, there are more opportunities for visual manipulation[9].

Our proposed Model CBPM focus on identifying a way to define interfaces of software components and to specify the structure of an application in terms of interface interconnections. What makes CBPM interesting is that it provides a component interoperability standard. When we use CBPM to connect pieces of software, deviations from the standard can cause interoperability problems.

The creation of object instances and the use of interfaces obtained from those instances are often done by separate pieces of code. This separation allows the interface user to be attached to different classes at different times and is an important feature of CBPM. Most existing software assumes great knowledge about the implementation details of the code that it calls. By deliberately avoiding such coupling, CBPM encourages a style of programming in which different pieces of code can be truly independent of one another. Rather than assuming knowledge about the implementation of some called code, a CBPM client assumes a service from an interface or collection of interfaces. Different implementations can provide that service. CBPM interfaces provide a standard means by which a client of an object can perform an initial negotiation for a particular service and then assume some behavior from that service that conforms to a known contract.

Design of CBPM: The Component based Programming Model defines several fundamental concepts based on COM standard that provide the model's structural underpinnings. These include:

- A binary standard for function calling between components.
- A provision for strongly-typed groupings of functions into interfaces.
- A base interface providing:
  A way for components to dynamically discover themselves when appropriate.
  Reference counting to allow components to track their own lifetime and delete itself
- A mechanism to uniquely identify components and their interfaces.
- A "component loader" to set up component interactions through interfaces implemented by other components.

Binary standard: For any given platform (hardware and operating system combination), CBPM follows COM standard to lay out virtual function table (vtable) in memory and a standard way to call functions through the vtable (Fig. 1). Thus, any language that can call functions via pointers can be used to write components that can interoperate with other components written to the same binary standard. The double indirection (the client holds a pointer to a pointer to a vtable) allows for vtable sharing among multiple instances of the same object class. On a system with hundreds of object instances, vtable sharing can reduce memory requirements.

Objects and components: Component objects support a base interface, along with any combination of other interfaces, depending on what functionality the component object chooses to expose.

Component objects usually have some associated data, but unlike C++ objects, a given component object will never have direct access to another component object in its entirety. Instead, component objects always access other component objects through interface pointers. This is a primary architectural feature of the CBPM, because it allows CBPM to completely preserve encapsulation of data and processing, a fundamental requirement of a true component software standard.

The Base interface contains three methods that must be implemented in every CBPM component. These methods provide the functionality of interface discovery and object life cycle management.

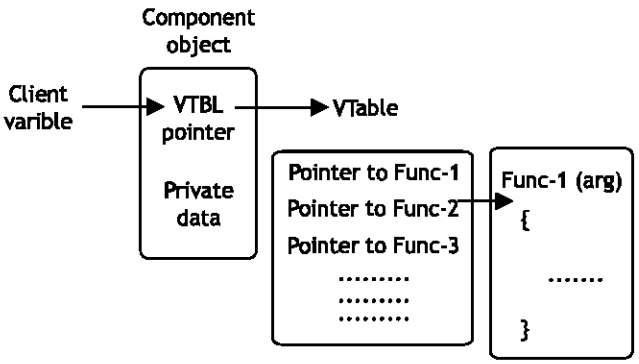

Fig. 1: Vtable function calling mechanism

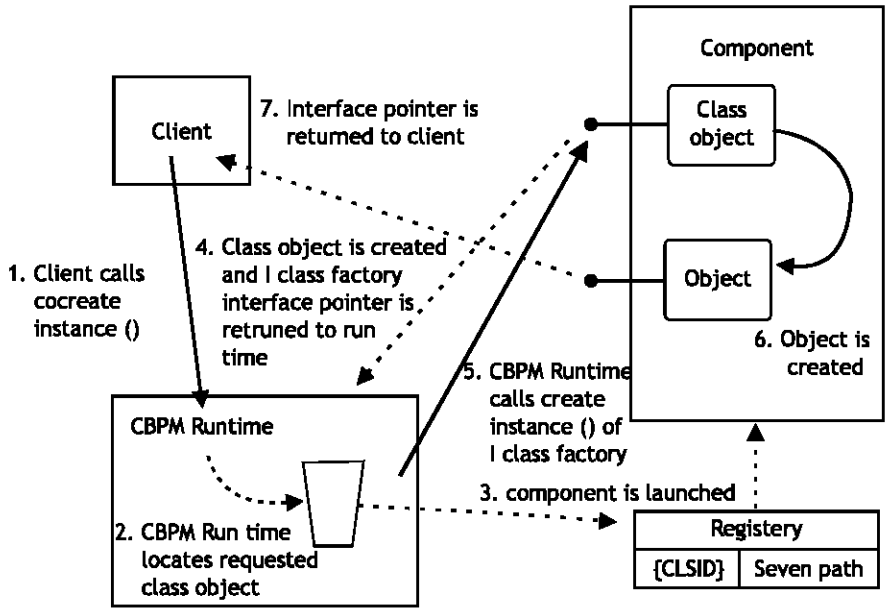

Fig. 2: CBPM internals

Globally unique identifier: The Unique identifiers are needed to locate components, as well as interfaces. One approach would be to use some type of string identifier, but this raises several problems, the most important being that it's very difficult to guarantee you've picked a truly unique identifier.

A GUID is a 128-bit integer which is used to uniquely identify components and interfaces. The algorithm used to generate GUIDs is statistically guaranteed to generate unique numbers.

When a CBPM component is installed onto a computer, it must be registered by creating entries in the computer's registry. To register a CBPM component, entries must be created under the predefined CLSID and server path fields. These entries enable the CBPM libraries to locate and load CBPM component.

Working mechanism: The Component Object Library is a system component that provides the working mechanics of CBPM. The Component Object Library provides the ability to make base interface calls; it also encapsulates all the "legwork" associated with launching components and establishing connections between components. Typically when an application creates a component object, it passes the CLSID of that component object class to the Component Object Library. The Component Object Library uses that CLSID to look up the associated server code in the registration database.

A class factory is the mechanism in CBPM used to instantiate new component objects. Every CBPM class has an associated class object. A class object knows how to create instances of a single CBPM component. The CBPM specification defines a standard API function for creating class objects and a standard interface for communicating with class objects. Thus, clients need to learn only one mechanism in order to create any type of CBPM object.

Figure 2 shows what happens at run time when an object is created. A portion of the CBPM run time looks in its internal data structures to locate the requested class object. It looks in the system registry to locate the code that knows how to create the requested class object and does whatever is necessary to load and execute that code.

If that code happens to be a shared library, CBPM loads the shared library. CBPM uses the component's standard interface to ask the class factory to create an instance of the component object and returns a pointer to the requested interface back to the calling application. The calling application neither knows nor cares where the server application is run; It simply uses the returned interface pointer to communicate with the newly created component object.

Finally, the Component Object Library is implemented as a part of the operating system and provides the "legwork" associated with finding and launching component objects.

RESULTS AND DISCUSSION

We evaluate the effectiveness of CBPM using three criteria: code size, development time and performance. We show how code size is reduced dramatically when using CBPM and the corresponding improvements in development and porting times. We also show

Table 1: Results (in ms) of OrbixCOM et and object bridege C++ server and client

Bridges	C and B on X, S on Y	C and X, S and B on Y	C on X, B on Y and S on Z
OrbixCOMet	250.87	250.87	251.73
Object brige	1.60	1.70	1.83

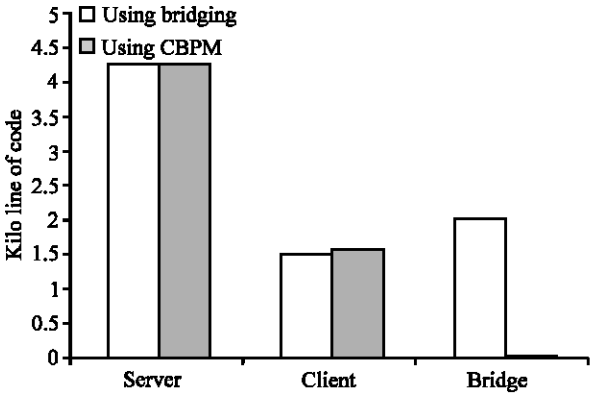

Fig. 3: Reduction in code size

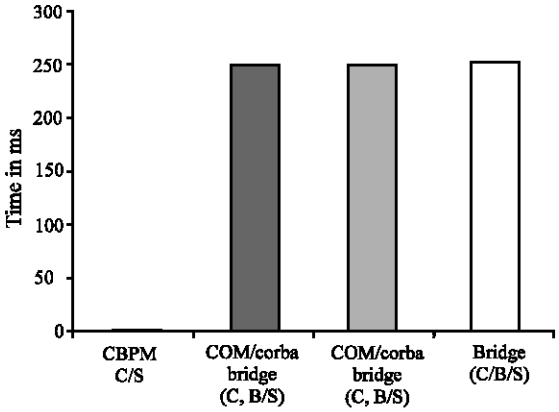

Fig. 4: Results (in ms) of OrbixCOMet and CBPM

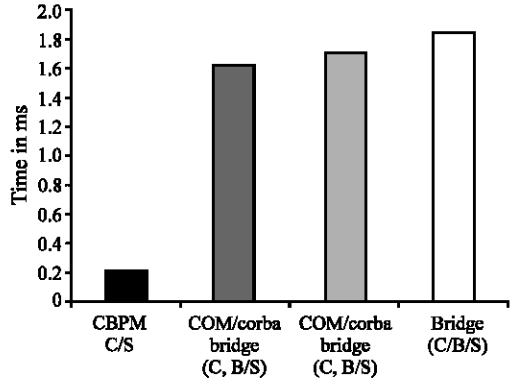

Fig. 5: Results (in ms) of ObjectBridge and CBPM

comparison between performances of using CBPM and bridging. We Constructed applications for Linux which used DCOM/CORBA bridges to access COM based Components on Windows along with an application which was based on CBPM. We report results based on our experimentation mentioned above.

When we used the bridging techniques, we needed to write more code compared to CBPM based application, because we had to write extra code for bridging; which in turn also increased the development time.

To measure performance, two DCOM-CORBA bridges were considered in our study: ObjectBridge by Visual Edge[10] and OrbixCOMet by IONA Technologies[11]. Both bridges are bi-directional and dynamic. ObjectBridge is also ORB-neutral while OrbixCOMet works with IONA Orbix3 only. Both of them are supposed to be compliant with the Inter-working Specification; however, we found that Objectbridge might lack support for two of the specification configurations. The count program was used to measure the performance of the two bridges. Each bridge was tested for three configurations. The performance was measured with and without the bridges for comparison. The count program was implemented with C++. In addition, the client was implemented with C++. In all cases, the same language was used for both the client and the server.

The DCOM-CORBA experiments were conducted using Windows 2000 and Linux 8.0 running on Pentium II machines with 333 MHz and 128 Mbytes of memory. Several software tools were needed for this study, including: Object Brident Enterprise Client, VisiBroker for C++, Orbix3, OrbixCOMet, MS Visual C++, DCOM Configuration and OLECOM Object View.

Table 1 shows the results for different configurations. All data were collected when the network (Ethernet) is observed to be idle. During performing the experiments, it was observed that network traffic could greatly affect the results. For all these tables, the following notation is used: C: Client, S: Server, B: Bridge and X, Y and Z are three machines.

Along with the bridging experiment we tested the same server and client based on CBPM at a single Linux system. CBPM client and server took 0.23 ms to run. The comparisons of CBPM with OrbixCOMet and ObjectBridge (with three different configurations) are shown in Fig. 4 and 5 respectively.

The DCOM-CORBA results for using OrbixCOMet and Objectbridge (Table 1, Fig. 4 and 5) show that CBPM is significantly faster than OrbixCOMet and Objectbridge by a great factor for most cases.

The full potential of component-based software architectural styles cannot be realized unless reusing code

developed by others becomes a common practice. A new architectural style can become a standard in its domain only if it makes reuse easier. We believe that CBPM is such a style for component development, with the potential for broader applicability. A key idea is to introduce an alternative of bridging between COM and CORBA by proposing a COM compatible architecture for Linux.

We have implemented and tested the CBPM with the hardware and software configurations. We found that the results readily support the effectiveness of the CBPM architecture. In some cases the development time was reduced by as much as 700%. We will like to highlight the importance of the new architecture with the cost involved and believe that the use of this architecture will be an efficient way of reducing the overhead.

Future work includes communication between components across process boundaries, network boundaries and threading Model.

REFERENCES

1. Miryam, W., 1997. Special Section Software Reuse, CIO Magazine, CIO Communication Inc.
2. Microsoft, 1995. Component Object Model Specification 0.9.
3. Object Management Group, 2001. CORBA Component Model Specification, ptc/01-11-03.
4. Sun Microsystems, 2002. Enterprise JavaBeans Specification 2.0.
5. Peter, L. and O. Risnes, 2001a. Technology Assessment of Middleware for Telecommunications, (EDIN 0094-0910).
6. Peter, L. and O. Risnes, 2001b. Market scan of CORBA-COM bridge products, (EDIN 0122-0910).
7. P. Becker, 1996. An Embeddable and Extendable Language for Large-Scale Programming on the Internet, In Proceedings of the 16th ICDCS.
8. Purtilo, M., 1994. Software Bus, ACM Transactions on Programming Languages and Systems, The Polylith.
9. Sun Microsystems, 1997. JavaBeans 1.0 Specification.
10. Visual Edge, 1998. The Premier OLE/COM-CORBA Interoperability Solution, ObjectBridge.
11. IONA., 1998.OrbixCOMet Desktop Programmer's Guide and References.