

# Applicability of Component-Based Development in High-Performance Systems

Amir Zeid, Michael Messiha, Sami Youssef

Department of Computer Science, the American University in Cairo (AUC)  
Email: [azeid@aucegypt.edu](mailto:azeid@aucegypt.edu) , [mrmessiha@hotmail.com](mailto:mrmessiha@hotmail.com) , [samiyous@hotmail.com](mailto:samiyous@hotmail.com)

Email:

## Abstract

*Component-based development (CBD) has been widely applied in different areas and has proven to be successful; especially in interactive service oriented systems. In such systems, the user usually need not interact with the whole system at one time; thus the overhead incurred in CBD can be tolerated. However, the applicability of such development methodology in high-performance systems is questioned since it involves overheads that will degrade system performance and render it of low quality. In this paper, we claim that CBD is not the right choice for high performance system. As a proof of concept, we will use the Apache HTTP web server as an example of a performance-entailing system.*

**Keywords:** Component-Based Development, CBD, Component object Model, COM, High performance systems, Web servers, Apache, C++.

## 1. Introduction

Component based development is a software development methodology that emerged in the second half of the 1990s with the sole target of achieving reuse and flexibility. The Apache web server was chosen to be the test subject for this paper because it is one of the most popular and the best-performing HTTP servers. Apache is coded using classical C language. Using CBD to reengineer existing high performance systems need to be fully justified. In our research we aimed at reverse engineering Apache then reconstruct it using CBD .

The rest of this paper is organized as follows: In the beginning, we will start by giving an introduction about the history of components; following, we will excavate the specifications of the component object model. Finally, we will tackle the Apache HTTP web server and try to re-design it in a component-based form and thus explain the negative consequences of such approach.

## 2. Component-Based Development

### 2.1 Background

Different engineering disciplines have relied on component based development for decades and were able to reduce both the time and cost of development by assembling already built components. The same idea emerged in software engineering as a result of the limited success of Object Oriented Development (OOD) to properly realize the benefits of reuse. In OOD, a system developer, trying to use other party classes, has to investigate the internal details of these classes before being able to use them. In addition, recompilation and re-linking of the whole system is needed in order to make the system functional [1]. Throughout the last few years, CBD

has been used in different disciplines such as business sectors and telecoms. In addition, some major companies have already started on the road to serious CBD such as Forte, IBM, Microsoft, SAP, Sterling and Sun. It has also been applied in the development of operating systems such as BITS, Pebble, PURE and Choices. Other areas of applicability include Graphical User Interfaces (GUIs) like OLEs, ActiveX controls typically used in web development.

### 2.2 CBD fundamentals

The main ideas that lie underneath CBD are reuse and flexibility. It's important to realize that developers no longer need to program and reprogram classes; they need only to reuse what was already implemented. Flexibility is another useful trait which allows for components to be easily versioned, replaced, and thus easily and transparently maintained. Transparency to the client is a feature that comes embedded in flexibility. The client needs not know about the component it's going to use, its physical location, its lifetime, or its present state. When a client needs a component, it just asks for it either by name or by a published identifier. The entity responsible for managing components and investigating their physical locations and their status is called the COM library. It hides all details from the client. Not surprisingly, the client itself may be a component.

### 2.3 What are components?

Components are reusable pieces of software. They can interact with each other using some mediator called the COM library which will be referenced later in the paper. As requirements evolve, new components can replace

existing ones that are seen outdated or less efficient. Component technology adds very much to systems that need multiple updates and extra customization (such as GUI(s) and OS(s)). Components can be packed into libraries and distributed as binary packages. However, in such case, some standard has to exist to unify communication between these binary packages and client programs. These standards are called interfaces that are explained in the coming sections. Through interfaces, components can achieve distribution and offer transparency to their clients because they need not know where such service providers exist whether locally or remotely. Components are also named Objects since you can map a class ID (CLSID) to a component and thus a client can request a component by ID.

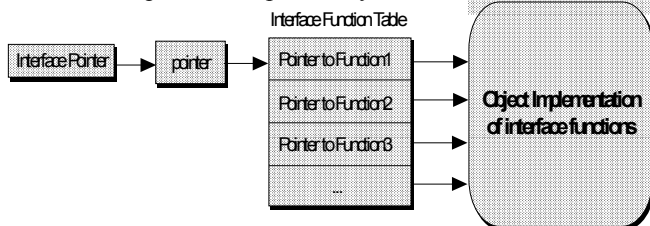


Figure 1: Interface Structure

## 2.4 Interfaces and Objects

An interface is simply a pointer to an array of function pointers representing the operations that can be accessed through this interface (as shown in Figure 1). The interface cannot exist without an instance of its object. An object implements one or more interfaces. When an object implements an interface, it has to implement each and every function of that interface. Clients only have access to interfaces and no direct access to implementation

## 2.5 COM Library

The COM Library is the entity responsible for managing components and determining their physical location. It initializes itself and locates the objects requested by different clients interacting with the system. It also manages the Memory needed to allocate parameters, arguments, indirect calls and the like. Its main functionality is to locate the object implementers or the servers hosting the components requested by the clients. Once located, the library loads the required component. It also checks when no clients are using a loaded component and unloads it. The unloading is done using something called reference counting, which is a counter of the references to a certain object.

## 2.6 Service Request Scenario

After explaining the main building blocks of the COM, let us now investigate a simple scenario<sup>1</sup> showing how a client can request a service from a component through the

<sup>1</sup> The scenario presented here is a very simple one for illustration purposes. The details of the lengthy full process of acquiring an interface pointer are hidden not to confuse the reader. For more details, contact the authors. The methods written in the scenario are the ones used in practice.

COM library (refer to Figure 2 as you read<sup>2</sup>). For instance, assume that a client wants to use a service published by a certain component. What the client has only to do is ask for the interface to the component providing that service.

The COM library loads that component (object) and returns to the client an interface pointer through which it can execute the requested service. The client executes the designated operation and then declares that it no longer needs that object. If that client was the only one using that object, the COM unloads this object. If not, the object will remain active until all clients are done with it; only then, the object is unloaded.

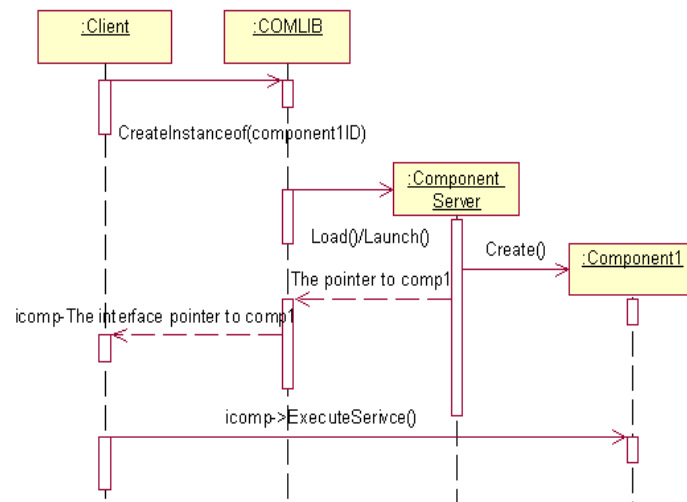


Figure 2: A possible scenario

## 3. Apache HTTP Web Server

The Apache HTTP web server will be the subject of our analysis as an example of a high performance system. Specialized papers describe in detail the architecture of the Apache web server such as the Apache Modeling Project [7] and the Conceptual Architecture of the Apache Web Server [6] from which we will be referencing some information. In the following sections, we will not focus on the inner details of design but rather on the external entities. We will start by describing the anatomy of the web server in its modular form; then the mapping of this design onto a component-based structure will be presented and a discussion of the negative effect of such a decision on the web server performance will take place to backup the paper's purpose.

### 3.1 Apache Web Server Design

The Apache design can be divided into two main parts: the core and the modules (as shown in figure 3). The core is the central web server dispatcher that uses the modules in order to satisfy clients' requests. At the beginning, the parent server starts by reading the configuration files (simple text files) in order to apply the administrator special requirements. Following, the parent server creates

<sup>2</sup> Ignore extra classes (if they exist) and focus only on the client, the COM library, and the component.

a number of child servers to handle clients' requests. Each of the child servers receives a request from the queue of waiting requests and starts serving it. In order to have a particular function handled, the parent server loops on all modules asking about the ones which declared interest in handling this function. At least one of those modules is able to handle the child server inquiry. After completely handling a request, the child server sends the saved response to the client and loops back to handle another coming request.

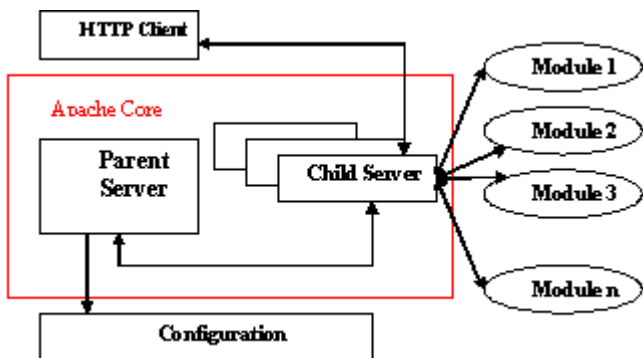


Figure 3: Apache Internal Architecture

## 4. CB Design of Apache

The component-based (CB) paradigm that we have adopted for the web server design tries to achieve a high degree of reuse and flexibility. Following is a description of how each of these requirements is realized in our design.

The Apache web server can be decomposed into the following components:

### a. Servers

These include both the parent server and child servers.

One possible interface for the parent server component includes:

```

RunServer()
StopServer()
RestartServer()
  
```

### b. Memory Management

Apache has a special memory management schema for efficient use of memory which divides the internal web server memory into pools and blocks. Memory allocation is always served from within a pool. Destroying a pool automatically deallocates memory within this pool.

One possible interface for this component includes:

```

CreatePool()    MakeSubPool()
ClearPool()     DestroyPool()
  
```

### c. Application Programming Interfaces (APIs)

Apache has special array and table APIs which serve as data repositories for the web server configuration as well as incoming request headers. They are also used during the request-response loop to save the outgoing headers that will be dumped to the client.

One possible interface for this component includes:

```

MakeArray()    CopyArray()    MakeTable()
CopyTable()
MergeTables()  TableSet()
TableGet()
  
```

### d. Configuration

The web server configuration process is usually done by reading text files that include directives specifying the special requirements of the system administrator. While the web server is reading these files, it saves the information in apache tables and arrays, specifying how each "module" should handle incoming requests.

One possible interface for this component includes:

```

MakeDirConfig()    MergeDirConfig()
MakeServerConfig()  InitConfig()
MergeServerConfig()
  
```

### e. Request Handlers

These are components hosted by one component server. Once the child server receives a request, they apply the previously read configuration to fetch the client request being a static resource or the result of running a script. Each of these entities is considered a different component by itself and has an interface different from other handlers.

One possible interface for one of the request-handler components includes:

```

TranslateName()
CheckMimeType()
HandleRequest()
  
```

By dividing the system into these components, a high degree of reusability is achieved. For example, the memory management component can be used in any continuously running application in order to improve the system efficiency and minimize memory usage. The APIs component is very beneficial and can be reused in applications requiring advanced data structures.

Flexibility is highly achieved since the system decomposition, as presented in this design, is logically acceptable. For example, any change in the configuration directives can be incurred into the system by only changing the configuration component. Another example that gives evidence of flexibility is in the request-handlers component. The emergence of a new mime type can be easily incorporated in the request-handlers component by adding the extra functions capable of handling this new mime type.

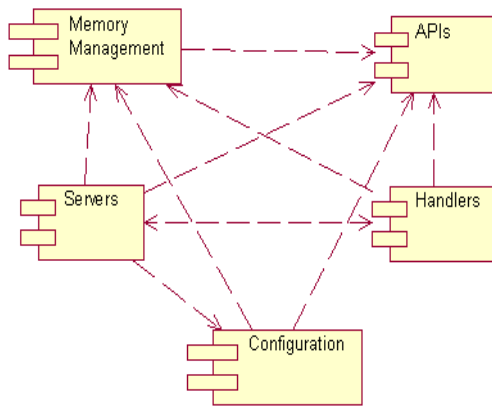


Figure 4: Apache's New Design and Components' Interaction

#### 4.1 The Apache CB Design in Action

A possible deployment of components is in component servers (loaded DLLs or launched EXEs). Components do not live alone; they have to live within a component server. In figure 4, components servers are not shown; they are merely outer containers for the drawn components. Parent and Child server classes are encapsulated in one component server which is loaded in memory by the COM library once the client application creates an instance of the parent server. Only the interface to the parent server component is available to external clients since they are not aware of the existence of child servers. It is important to notice that this component server is permanently loaded in memory as long as the server is running because the client holding a reference to the server object will never release it due to the infinite loop.

The memory management component server is a crucial component for the functionality of the whole web server. Other components constantly act as clients for this component due to the need for continuous memory allocation in the course of serving clients' requests. According to the COM specification, a component server is unloaded from memory once no clients hold references to it.

If a request-handler component asks for a service provided by the memory management component, and then requests no further memory allocation, it thus releases the reference to the memory component interface. The COM library will notice that no clients are referencing the memory management component and will consequently unload it from memory. It is possible that shortly after unloading, another request-handler component requests a memory management service. This will necessitate the re-loading of the memory management component server in memory and lose precious time that can boost the request-handling time.

The same applies to any of the other components. For instance, if one of the child servers received a request for a file with a mime type "image/gif"; this request can be handled by request-handler component "A". After handling the child server request, the COM library unloads the request-handlers component server holding all handlers components. Again, an incoming new request could be received shortly and requires the reloading of the

request-handlers component server thus decreasing the whole web server performance.

One possible solution to the problem of loading and unloading is to lock the component server<sup>3</sup> permanently in memory. If the COM library notices that no more clients use any of the components hosted by the component server, it does not unload the server and keeps it locked<sup>4</sup> in memory. However, resorting to such a solution is not quite practical in the context of a web server since a long time gap could elapse between any two successive requests. This scenario will invalidate the benefits of using components.

Although the problem of loading and unloading of component servers can be somehow alleviated, the communication overhead cannot be overcome. As shown in figure 3, all components use the memory management and use the APIs components extensively besides interacting with each other. A client requesting a component service has to pass through multiple levels of indirection starting by requesting an interface pointer from the COM library and ending by actually invoking the method through that pointer. Think of what chains of inter-component calls would incur – certainly poor performance. Such scenario illustrates the expensive subsystem communication involved in CBD that one cannot even ignore or bypass.

### 5. Challenges to CBD Performance

Some performance-related challenges exist when we tackle CBD in relation to performance-entailing systems. As was illustrated, the COM library unloads a component when it's no longer in need by any client; thus, it saves memory. Here comes the critical issue; what if a client comes shortly after the component was unloaded asking for some service. Again, the COM library will load this given component to fulfill the client's request - a real threat to performance.

Being only loaded when needed is a bright idea in CBD, but when it contradicts with a main requirement of a system as in high-performance, a sacrifice has to happen. Indeed, Loading and Unloading of components is beneficial to the system's memory usage but, unfortunately, it negatively affects performance.

Another threat to performance is shown in the interface programmatic structure (see Figure 1&2) and in the excessive use of pointers in this COM library itself. The Interface structure described above imposes an overhead caused by multiple levels of indirection. The COM library, as well, uses pointers to objects. Of course, there is no other way through which we could pass objects but through pointers; it's an expense that CBD has to pay. Anyone that uses pointers knows that, the more pointers, the more levels of indirection, which will eventually degrade the performance of the system. It will even

<sup>3</sup> 'Component Server' is the host that holds the component(s). Do not mix it with 'server' representing the web HTTP server

<sup>4</sup> Server Locking happens using some interface that exist in the component server called IClassFactory. The idea is related to reference counting; you have a locking counter that you initialize it with one instead of zero to lock the server in memory and prevent its unloading.

degrade it more when a service is to use any external services that exist in other components.

## 6. Conclusion

In his book, *Software Engineering*, Sommerville describes performance [5] as follows:

“Performance is a critical requirement; this suggests that the architecture should be designed to localize critical operations within a small number of sub-systems with as little communication as possible between these sub-systems.”

When this basic definition is applied to component-based high performance systems, a key contradiction arises due to the fact that a component-based system does not minimize communication since it is necessary to link the independently developed components. As proven in the Apache web server, adopting component-based design harms the server more than benefits it by depreciating performance to a very low level. Therefore, we believe that CBD should not be used in high-performance systems; instead, this development paradigm is very applicable to user-interactive systems (GUIs) and possibly in (OSes).

## References

[1] Crnkovic, Ivica, Magnus Larsson. *Challenges of Component-based Development*.  
[www.mrtc.mdh.se/publications/0327.pdf](http://www.mrtc.mdh.se/publications/0327.pdf).

[2] Dragoi, Octavian Andrei. *The Conceptual Architecture of the Apache Web Server*.

[www.math.uwaterloo.ca/~oadragoi/CS746G/a1/apache\\_conceptual\\_arch.html](http://www.math.uwaterloo.ca/~oadragoi/CS746G/a1/apache_conceptual_arch.html).

Department of Computer Science, University of Waterloo. January 26, 1999.

[3] Microsoft Corporation. **The COM Specification**. 1995

[4] Rogerson, Dale. Microsoft Press Product. **Inside COM**. 1997

[5] Sommerville, Ian. **Software Engineering**, Six Ed. Addison-Wisley Publishers Limited. 2001

[6] The Apache Software Foundation.  
<http://httpd.apache.org/>

[7] The Apache Modeling Project.  
<http://apache.hpi.uni-potsdam.de/document/Contents.html>