Software Component Models from a Technical perspective

Mohd Hasan Selamat, Hamid Sanatnama, Abdul Azim Abd Ghani, and Rodziah Atan
Faculty of Computer Science and Information Technology University Putra Malaysia 43400 UPM Serdang

Summary
Component-based Software Development is an approach that has many benefits, such as improving application developer productivity, reducing costs and complexity. Programming within this approach is like assembly rather than development, which reduce skill requirements, and allow expertise focus on domain problems. The foundation of any CBSD methodology is its underlying component model, which defines what components are, how they can be constructed, and specifies the standards and conventions that are needed to enable composition of independently developed component. This paper presents a survey of the current available component technologies with focus on the technical perspective of each component model in order to have better understanding for developing a new component model. We have categorized them based on Distributed Application Support and Interaction mechanism.

Key words:
Component Models, Component Compositions, Interoperability, Remote Procedure Call

1. Introduction

Component-based software technology is becoming an increasingly popular approach to facilitate the development of evolving systems, and has many benefits such as improving application developer productivity, reducing costs and complexity by reusing of existing code. Programming within this approach is like assembly rather than development, which reduce skill requirements, and allow expertise focus on domain problems.

The foundation of any CBSD methodology is its underlying *component model*, which defines what components are, how they can be constructed, how they can be composed or assembled. Within CBSD we also distinguish development of components from development of systems. In component-based system development, we focus on identification of reusable entities and selection of components that fulfills system's requirements, but in developing component our focus is on reusability.

Components communicate with their environment only through their interfaces, so it is the interface which provides all the information needed. The current component technologies is designed to allow clients to communicate transparently with objects, regardless of where those objects are running—in the same process, on the same machine, or on a different machine. As none of existing models support composition in both design and deployment phase [16], in this paper we present widely used component models from a technical view in order to have better understanding for developing a new component model. Since components are supposed to be used as building blocks from a repository and assembled or plugged together into larger blocks, composition is a central issue in CBSD.

For composition, existing approaches usually adopt message passing, which allows components to invoke one another's operations by remote procedure calls, either directly or indirectly via a channel such a bus. Examples of direct message passing are Remote Procedure Calls (RPC) and Event Delegation. Indirect message-passing in the other hand uses connectors which are separate entities and are the basis of many software models [3].

1.1 Interoperability

ISO/IEC 2382-01, interoperability is *"The capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units"*. As procedural interactions were confined to process boundaries, operating systems support a wide variety of mechanism for Inter-process Communication (IPC), such as files, pipes, sockets, and shared-memory [1].

1.2 Remote Procedure Calls

One of the reasons for proposing RPCs by Birrel and Nelson [28], which is build on top of the IPC, is that all these IPC mechanisms operate on the level of bits and bytes – quite far from well-ordered world of procedures with typed parameters. The evolution of certain component models shows the achievement on interoperability on all levels to solve the problem of interaction/connectivity of software across process boundaries.

The current component technologies are designed to allow clients to communicate transparently with objects, regardless of where those objects are running; in the same process, on the same machine, or on a different machine. Semantic of RPCs allows a client to invoke a procedure on

Manuscript received October 5, 2007
Manuscript revised October 20, 2007

a remote host, which looks like a local procedure call. This is done by providing a **stub** on the client side, and also for each remote procedure. The stub marshals and unmarsalling parameters which should be transmit between callee and caller.

The most well-known service implementing RPCs across heterogeneous platforms is Distributed Computing Environment (DCE), a standard of the Open Software Foundation (OSF). DCE has introduced an Interface Definition/Description Language (IDL), which is a key component of the CORBA standard and is recommended as the software interface specification due to its language, platform, and vendor independence. It is IDL which create automatically stub and specify, for each remotely callable procedure, number, passing modes, and the type of the parameters, as well as the type of possible return value. DCE also introduced the concept of universally unique identifiers which guarantees uniqueness until year 3500.

1.2.1 IDL's weaknesses and strengths

IDL supports the basic specification for distributed components, such as the operations and attributes provided by the component. Some IDLs are used with a specific programming language, but others can be used with various programming languages, such as OMG IDL. But IDL can not describe all of the information, such as, the pre/post conditions, and semantic descriptions of functionality, of the distributed component. Moreover, IDLs do not provide any additional information about the server's external dependencies such as, the callback invocation of a client's method. Although IDL is human-readable in terms of its syntax, it is a type of program level specification and can be compiled into executable code.

1.2.2 Microsoft's RPC

A decade later Microsoft adopted DCE/RPC as the basis of their Microsoft Remote Procedure Call (MSRPC) mechanism, and implemented Distributed Component Object Model DCOM on top of it. There is also a lightweight version of remote procedure call (LRPC) [22], which can be used for inter-process communication on a single machine. MSRPC include support for Unicode strings, implicit handles, inheritance of interfaces which are extensively used in DCOM, and complex calculations in the variable-length string and structure paradigms already present in DCE/RPC.

1.2.3 Object invocation vs. Procedural invocation

The object invocation is not the same as procedural invocation. A method call inspects the class of receiving object and picks method implementation provided by that class. Also a method always provides, as another parameter a reference to the object to which the message was sent. Many RPCs models share these two properties.

There are two ways to make this possible. Firstly implementing method call on top of the machinery that implements procedure calls. Examples of these approaches are, System Object Model from IBM and CORBA ORBs and also Microsoft's COM, although it does rely on tables of procedure variables. Secondly to define a new virtual machine level with build-in support for method calls, such as JVM and .NET common language runtime (CLR).

1.2.4 Interface and Object reference specification

Components communicate with their environment only through the interface, so it is only the interface which provides all the information needed. All current approaches uniformly define an interface as a collection of named operations, each with a defined signature and possibly a return type.

Approaches [1] for connecting interfaces to object are:

- Traditionally, one-to-one relation between interfaces and object. (CORBA 2, SOM)
- Many interfaces with a single object. (JAVA, CLR)
- Many interfaces with many part objects in a component.(COM,CCM in CORBA 3)

Interfaces are specified by IDL in all approaches which follow the DCE. Among competing proposal, the OMG IDL and COM IDL are the strongest. On object references, there are also different approaches, but all have mechanisms to map locally meaningful references that retain meaning across process, machine and network boundaries.

1.2.5 Connectors

Components and connectors are the basis of many software models [3]. ADLs have always defined software systems in terms of components and connectors. Even component models that do not use connectors explicitly often have composition operators that can be interpreted as connectors at different level of abstraction.

In existing component models, connectors are channels for coordinating the control flow between components. This provides a mechanism for message passing, which allows components to invoke one another's operations by method calls or remote procedure calls, either directly or indirectly via a channel such a bus.

2. Existing Software Component Models

The two concepts component models and component frameworks sometimes are intermixed. A component model defines a set of standards and conventions used by component developer. A component framework is a support infrastructure for component model.

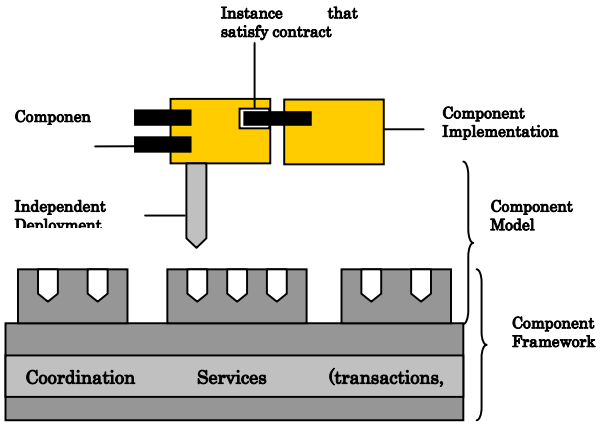

Figure 1, Component-based Technology

There are many different component models, because there are many different domains with different requirements on component-based systems. Existing component models adopt different component definitions and composition operators [4]. We will consider the following software component models: JavaBeans [2] [4], EJB [5, 6], COM [7] [8] [9], CORBA [10] [11] [12] [13] [14], Koala [15] [16], SOFA [17] , ADLs [18].

2.1 JavaBeans

A Java Bean is a reusable software component that can be manipulated visually in a builder tool. One of the goals of the JavaBeans APIs is to define a *software component model* for Java, so that third party can create and ship Java components that can be composed together into applications by end users. Another reason for giving birth to JavaBeans is that, there was no standard technology to help programmer build java components which can interact with one other in common way.

Portability as one of the main goals of the JavaBeans architecture provides platform neutral component architecture. When a Bean is nested inside another Bean then we will provide a full functionality implementation on all platforms. However, at the top level when the root Bean is embedded in some platform specific container such as or Visual Basic or Netscape Navigator then the JavaBeans APIs should be integrated into the platform's local component architecture. This means that on the Microsoft platforms the JavaBeans APIs will be bridged through into COM and ActiveX. Similarly, it will be possible to treat a bean as a Live Object part, or to integrate a bean with LiveConnect inside Netscape Navigator.

The three most important features of a Java Bean are the set of *properties* it exposes, the set of *methods* it allows other components to call, and the set of *events* it fires.

The basic run-time model for Java Bean components is that they run within the same address space as their container. So for example, if the container is a Java application, then the contained bean is run in the same Java virtual machine as its container. If the container is a non-Java application, then the Java Bean will run in a Java virtual machine that is directly associated with the application.

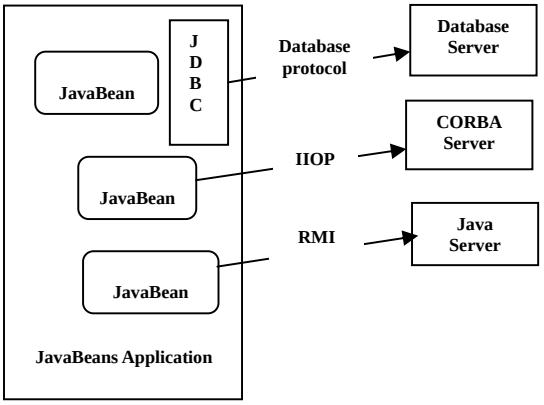

Figure 2, JavaBeans interaction with different servers/platforms [14]

Although Java environment supports multi-threading, having a bean running in several threads at the same time can cause problem. Many Java Beans will have a GUI representation. When composing beans with a GUI application builder it may often be this GUI representation that is the most obvious and compelling part of the beans architecture. However it is also possible to implement invisible beans that have no GUI representation. These beans are still able to call methods, fire events, save persistent state, etc. They will also be editable in a GUI builder using either standard property sheets or customizers. They simply happen to have no screen appearance of their own.

2.2 Enterprise JavaBeans (EJB)

EJB, besides having competitive advantage it offers in term of distribution and platform independency, has been recognized as an excellent platform for creating enterprise solution, especially for *distributed server-side* applications. It combines server-side components with distributed object technologies such as CORBA and Java RMI to greatly simplify that task of application development. Server-side component model defines architecture for developing *distributed business objects*, and are used on the middle-tier application server, which manage the components at runtime and make them available to remote clients.

There are three different kinds of enterprise beans:

- **Entity beans,** model business data; Java objects that cache database information.
- **Session beans,** model business processes; Java objects that act as agents performing tasks and services.

- **Message-Driven beans,** model message-related business processes; Java objects that act as message listeners, which can be triggered be receiving messages from other beans.

The EJB architecture will be the standard component architecture for building distributed object-oriented business applications in the Java™ programming language and will support the development, deployment, and use of web services. EJB architecture applications, application developers do not have to understand low-level transaction and state management details, multi-threading, connection pooling, or other complex low-level APIs (EJB applications will follow the Write Once, Run Anywhere™ philosophy of the Java programming language). The EJB architecture will also address the development, deployment, and runtime aspects of an enterprise application's life cycle and define the contracts that enable tools from multiple vendors to develop and deploy components that can interoperate at runtime. Using EJB architecture it is possible to build applications by combining components developed using tools from different vendors and provide interoperability between enterprise beans and J2EE components as well as non-Java programming language applications. The EJB architecture is compatible with existing server platforms. Vendors are able to extend their existing products to support Enterprise JavaBeans and will be compatible with other Java programming language APIs. The EJB architecture is also compatible with the CORBA protocols.

2.2.1 Client-Side Objects in a Distributed Environment

When the RMI-IIOP protocol or similar distribution protocols are used, the remote client communicates with the enterprise bean use *stubs* for the server-side objects which implement the remote home and remote interfaces.

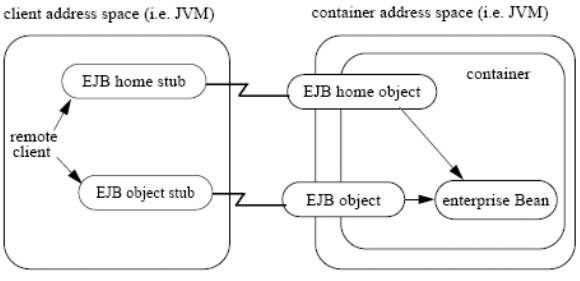

Figure 3, Location of EJB Client Stubs [8]

The communication stubs used on the client side are artifacts generated at the enterprise bean's deployment time by the Container Provider's tools. The stubs used on the client are specific to the wire protocol used for the remote invocation.

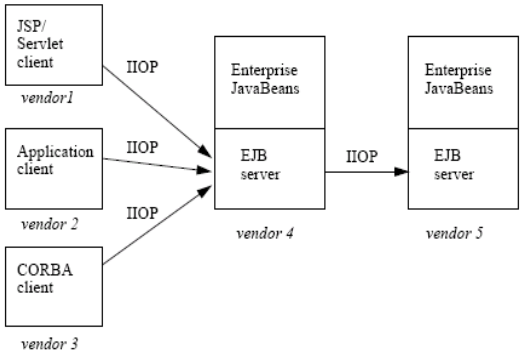

Figure 4, Heterogeneous EJB Environment [8]

2.2.2 Support for Distribution

The remote home and remote interfaces of an enterprise bean's remote client view are defined as Java™ RMI interfaces. This allows the container to implement the remote home and remote interfaces as *distributed objects*. A client using the remote home and remote interfaces can reside on a different machine than the enterprise bean and the object references of the remote home and remote interfaces can be passed over the network to other applications. Comparing with original JavaBeans, which are intended to be used for *intra*process purpose, EJB are designed to be used as *inter*process components.

2.2.3 Interoperability Goals

The goals of the interoperability requirements are:

- To allow clients in one application deployed in J2EE containers from one server provider to access services from session and entity beans in another application that is deployed in an EJB container from a different server provider. For example, web components that are deployed on a J2EE-compliant web server provided by one server provider must be able to invoke the business methods of enterprise beans that are deployed on a J2EE-compliant EJB server from another server provider.
- To achieve interoperability without any new requirements on the J2EE application developer.
- To ensure out-of-the-box interoperability between compliant J2EE products. It must be possible for an enterprise customer to install multiple J2EE server products from different server providers, deploy applications in the J2EE servers, and have the multiple applications interoperated.
- To leverage the interoperability work done by standards bodies (IETF, W3C, and OMG), so that customers can work with industry standards and use standard protocols to access enterprise beans.

2.3 COM

Component Object Model (COM) is a Microsoft

platform for software componentry introduced by Microsoft in 1993. It is used to enable *interprocess communication* and dynamic object creation in any programming language that supports the technology. It means that the language for the source code of a component can be any programming language that supports function call via pointers, such as C, C++, and Ada. The term COM is often used in the software development world as an umbrella term that encompasses the OLE, OLE Automation, ActiveX, COM+ and DCOM technologies.

COM is a binary compatibility specification and associated implementation that allows clients to invoke services provided by COM objects. As shown in **Figure 5**, services implemented by COM objects are exposed through a set of interfaces that represent the only point of contact between clients and the object.

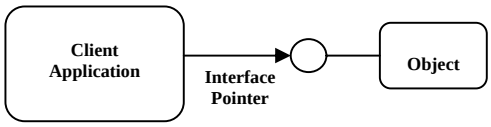

Figure 5, services provided by COM object through interface pointer

Obviously, the only point of contact between the client and the object is a set of interfaces. Component interfaces are specified in Microsoft IDL (COM IDL). Each interface specifies the signatures of the functions it implements. A COM component can implement multiple interfaces. Every component implement an **IUnknown** interface, which is a special interface that implements some essential functionality.

IUnknown has three methods:

AddRef()- Tells the COM object to increment its reference count.

Release()- Tells the COM object to decrement its reference count.

QueryInterface()- Requests an interface pointer from a COM object.

For example, **Figure 6** expose a COM object that emulates a clock. IClock, IAlarm, ITimer, and IUnknown are the interfaces of the clock object.

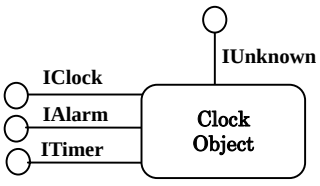

Figure 6 A Clock COM object

2.3.1 Interface

An interface which provides a grouped collection of related methods, its name starts with "***I"*** and may *inherit* from other interfaces. The implementation of the interfaces is in a *coclass* (**c**omponent **o**bject **c**lass). A COM server is a binary (DLL or EXE) that contains on or more coclasses. To avoid name collisions, each object and interface must have a GUID (Globally Unique IDentifier) which is a 128-bit number, and COM's language-independent way of identifying things. UUIDs from OMG is similar to COM GUIDs. A *class ID*, or *CLSID*, is a GUID that names a coclass. An *interface ID*, or *IID*, is a GUID that names an interface. An *HRESULT* is an integral type used by COM to return error and success codes. Interfaces are considered logically immutable. Once an interface is defined, it should not be changed. If new functionality has to be added to a component, it can be exposed through a different interface.

2.3.2 Interoperability

COM defines a binary structure for the interface between the client and the object which provides the basis for *interoperability* between software components written in arbitrary languages. Every COM object runs inside a server. A single server can support multiple COM objects. As shown in **Figure 7**, there are three ways in which a client can access COM objects provided by a server:

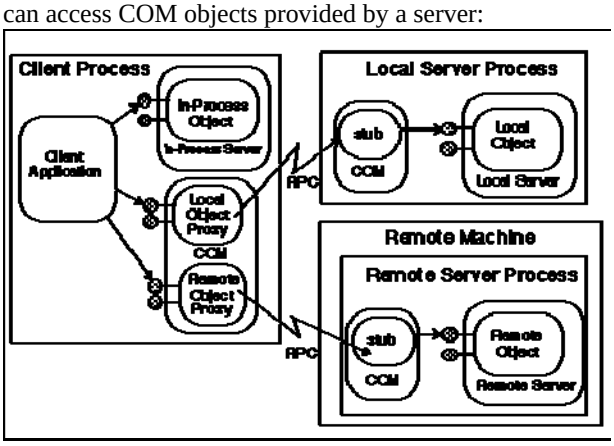

Figure 7, Three methods for accessing COM objects [10]

- In-process server: The client can link directly to a library containing the server. The client and server execute in the same process. Communication is accomplished through function calls.
- Local Object Proxy: The client can access a server running in a different process but on the same machine through an inter-process communication mechanism. This mechanism is actually a lightweight Remote Procedure Call (RPC).
- Remote Object Proxy: The client can access a remote server running on another machine. The network communication between client and server is accomplished through DCE RPC. The mechanism supporting access to remote servers is called DCOM.

If the client and server are in the same process, the sharing of data between the two is simple. However, when the server process is separate from the client process, as in a local server or remote server, COM must format and bundle the data in order to share it. This process of preparing the data is called marshalling.

In COM marshalling accomplishes through a "proxy" object and a "stub" object that handle the cross-process communication details for any particular interface. The following figure exposes the client call to the server through the proxy, which marshals the parameters and passes them to the server stub. The stub unmarshals the parameters and makes the actual call inside the server object. When the call completes, the stub marshals return values and passes them to the proxy, which in turn returns them to the client. The same proxy/stub mechanism is used when the client and server are on different machines. However, the internal implementation of marshalling and unmarshalling differs depending on whether the client and server operate on the same machine (COM) or on different machines (DCOM).

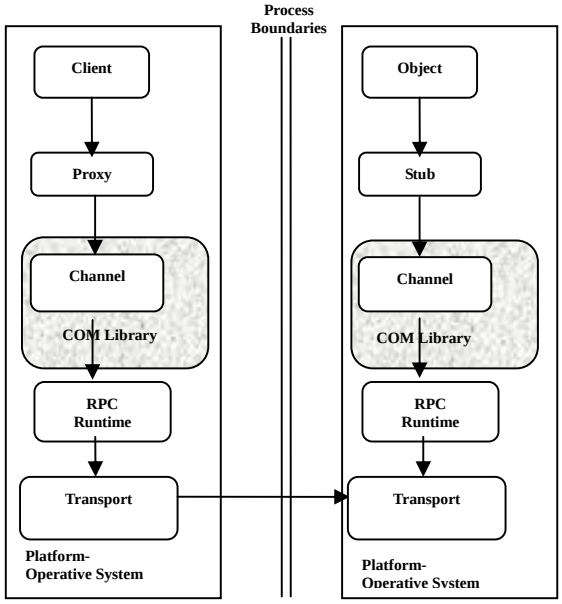

Figure 8, Cross-process communication in COM [10]

2.3.3 *The structure of proxy*

Proxy support standard marshalling of parameters belonging to two interfaces: **IA1** and **IA2**. Each interface proxy implements **IRpcProxyBuffer** for internal communication between the aggregate pieces. When the proxy is ready to pass its marshaled parameters across the process boundary, it calls methods in the **IRpcChannelBuffer** interface, which is implemented by the channel. The channel in turn forwards the call to the RPC run-time library so that it can reach its destination in the object. It is also proxy that generates the appropriate remote procedure call.

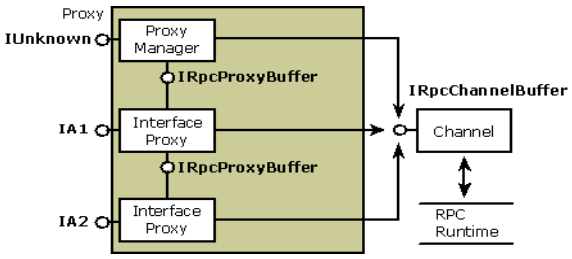

Figure 9, Structure of proxy in COM [7]

2.3.4 *The structure of stub*

COM creates the "stub" in the object's server process and has the stub manage the real interface pointer. COM then creates the "proxy" in the client's process, and connects it to the stub. The proxy then supplies the interface pointer to the client. The differences between the stub and the proxy are

- Stub represents the client in the object's address space.
- The stub is not implemented as an aggregate object.
- The interface stubs are private rather than public.
- The interface stubs implement **IRpcStubBuffer**, not **IRpcProxyBuffer**.
- Instead of packaging parameters to be marshaled, the stub unpackages them after they have been marshaled and then packages the reply.

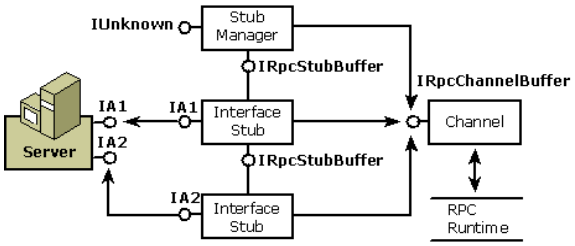

Figure 10, Structure of the stub in COM [7]

Microsoft RPC is a model for programming in a distributed computing environment. The goal of RPC is to provide transparent communication so that the client appears to be directly communicating with the server. Microsoft's implementation of RPC is compatible with the Open Software Foundation (OSF) Distributed Computing Environment (DCE) RPC. Microsoft RPC includes the Interface Definition Language (IDL) and its compiler.

Given an IDL file, the Microsoft IDL compiler can create default proxy and stub code that performs all necessary marshalling and unmarshalling. COM technology includes interfaces and API functions that expose operating system services, as well as other mechanisms necessary for a distributed environment.

Some clients need runtime access to type information about COM objects which is generated by the Microsoft IDL compiler and is stored in a type library. COM provides interfaces to navigate the type library.

COM objects need a way to store their data when they

are not running. COM supports object persistence through "Structured Storage", which creates an analog of a file system within a file. Individual COM objects can store data within the file, thus providing persistence.

Clients often require a way to allow them to connect to the exact same object instance with the exact same state at a later point in time. This support is provided via "monikers". Uniform Data Transfer provides for data transfers and notifications of data changes between a source called the data object, and something that uses the data, called the consumer object. COM allows such objects to define outgoing interfaces to clients as well as incoming interfaces. This enables two-way communication between the client and the component.

2.4 CORBA

Common **O**bject **R**equest **B**roker **A**rchitecture 1.1 was introduced in 1991 by OMG and defined the IDL and the API that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). The standard Internet Inter-Orb Protocol (IIOP) is a protocol for communication between CORBA ORBs that has been published by the OMG. IIOP is an implementation of the General InterORB Protocol (GIOP) for use over an internet, and provides a mapping between GIOP messages and the TCP/IP layer.

Using IIOP a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network. Because of the easy way that CORBA integrates machines from so many vendors, with sizes ranging from mainframes through minis and desktops to hand-helds and embedded systems. The most important, are most frequent uses is in servers that must handle large number of clients, at high hit rates, with high reliability.

CORBA applications are composed of *objects*, individual units of running software that combine functionality and data, and that frequently represent something in the real world.The IDL interface definition is independent of programming language, but *maps* to all of the popular programming languages via OMG standards: OMG has standardized mappings from IDL to C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLscript. In CORBA, every object instance has its own unique *object reference*, an identifying electronic token.

2.4.1 Ports

CORBA components support a variety of surface features through which clients and other elements of an application environment may interact with a component. These surface features are called *ports*. The component model supports four basic kinds of ports:

Facets, which are distinct named interfaces provided by the component for client interaction.

Receptacles, which are named connection points that describe the component's ability to use a reference supplied by some external agent.

Event sources, which are named connection points that emit events of a specified type to one or more interested event consumers, or to an event channel.

Event sinks, which are named connection points into which events of a specified type may be pushed.

Attributes, which are named values exposed through accessor and mutator operations. Attributes are primarily intended to be used for component configuration, although they may be used in a variety of other ways.

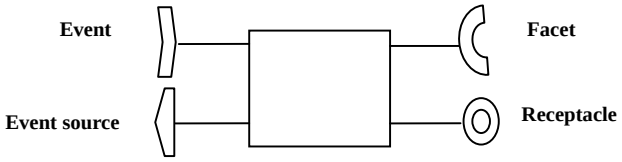

Figure 11, CORBA Component

The repository of CORBA components is a CCM container hosted and managed by an application server, and CORBA components are assembled by method and event delegation in a way that Facets match Receptacles and Event sources match Event sink in the design phase.

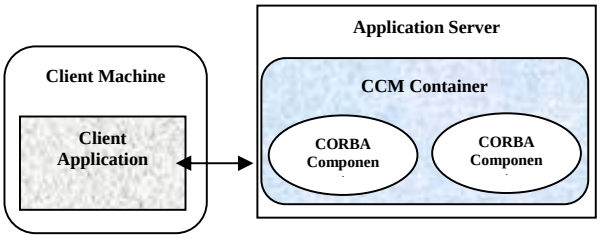

Figure 12, CCM Container

2.4.2 Interoperability

The client acts as if it's invoking an operation on the object instance, but it's actually invoking on the IDL stub which acts as a proxy. Passing through the stub on the client side, the invocation continues through the ORB, and the skeleton on the implementation side, to get to the object where it is executed. Any client that wants to invoke an operation on the object *must* use this IDL interface to specify the operation it wants to perform, and to marshal the arguments that it sends. When the invocation reaches the target object, the *same* interface definition is used there to unmarshal the arguments so that the object can perform the requested operation with them.

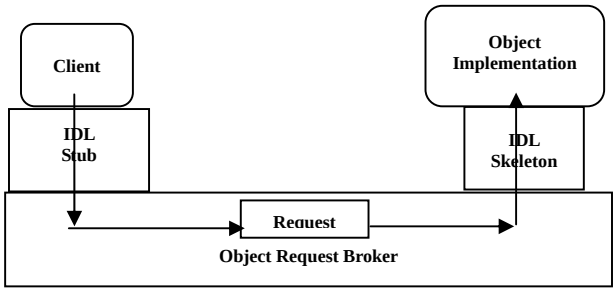

Figure 13, A request flow from client to the object implementation [2]

2.4.3 Remote Invocation

To make the remote invocation, the client uses the same code that it used in the local invocation, substituting the object reference for the remote instance. When the ORB examines the object reference and discovers that the target object is remote, it routes the invocation out over the network to the remote object's ORB. In order to invoke the remote object instance, the client first obtains its object reference. Naming Service and Trader Service are the easy ways.

2.4.4 Naming Service

To avoid deal directly with machine representation of object references, The Naming Service allows the binding of an object reference with user-friendly names. A name binding is always defined relative to naming context. A naming context is an object that contains a set of name bindings in which each name is unique. The Naming Service provides the principle mechanism through which most clients of an ORB-based system located objects that they intend to use. Given an initial naming context, clients navigate naming context retrieving lists of the names bound to that context. The CORBA naming service is defined as IDL module **CosNaming**. This module defines two data types: naming component and name, which is a sequence of name components. This module also supplies two interfaces: naming context and binding iterator. The Naming Context interface provides the necessary operation to bind a name to an object, and to look up a name in order to obtain the associated object reference.

```
module CosNaming [
  typedef string Istring;
  struct NameComponent {Istring id; Istring kind;};
  typedef sequence <NameComponent> Name;
  enum BindingType {nobject, ncontext};
  struct Binding {Name binding_name;
          BindingType binding_type;};
   typedef sequence <Binding> BindingList;
   interface BindingIterator; interface NamingContext {
   exception CannotProceed {
            NamingContext cxt; Name rest_of_name;};
   void bind (in Name n, in object obj)
      raises (CannotProceed);
   void list (in unsigned long how_many,
            out BindingList bl, out BindingIterator bi};
   interface BindingIterator {boolean next_n
            (in unsigned long how_many,
             out BindingList bl); };};
] //other declaration not shown
```

Example 1, CosNaming IDL

2.4.5 Trader Service

Trading Object Service facilitates the offering and discovery of instances of services of particular types. Discovering services is called “import”, and offering a service is called “export”. Export and import facilitate dynamic discovery of, and late binding to, services.

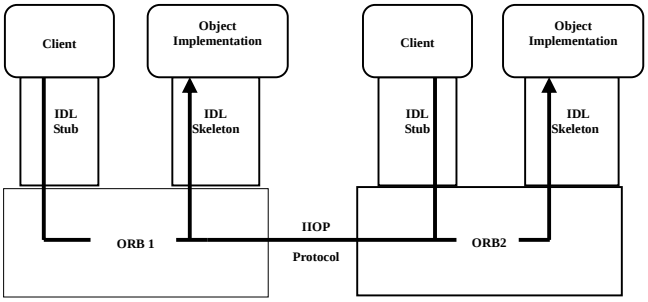

Figure 14, Interoperability by ORB-to-ORB communication [2]

OMG has standardized this process at two key levels. First client stub and object skeleton are generated from the *same IDL*, that makes the client knows the type object it’s calling, and also knows exactly which operation it may invoke. When the invocation reaches the target, everything is there and in the right place. Second client's ORB and object's ORB must agree on a *common protocol* - that is, a representation to specify the target object, operation, all parameters of every type that they may use, and how all of this is represented over the wire. OMG has defined this also - it's the standard protocol IIOP.

2.4.6 OMG Interface Definition Language

For each object type, you define an interface in OMG IDL. The interface is the syntax part of the contract that the server object offers to the clients that invoke it. The separation of interface from implementation, enabled by OMG IDL, is the core of CORBA which enables interoperability. In contrast, the *implementation* of an object - its running code, and its data - is hidden from the rest of the system behind a boundary that the client may not cross. IDL compiles into client stubs and object skeletons, and write object and a client for it. Stubs and skeletons serve as proxies for clients and servers, respectively. Because IDL defines interfaces so strictly,

the stub on the client side has no trouble meshing perfectly with the skeleton on the server side, even if the two are compiled into different programming languages, or even running on different ORBs from different vendors.

2.5 Koala

Most Consumer Electronics (CE) today contain embedded software and have a diversity of features. The past years has shown that the size and complexity of the software, the required diversity of products and developing time are the significant problems which are major challenges. Embedded software in CE provides basic control of hardware, signal and data processing has shifted from hardware to software, and it make new product features possible. That's why CE products have become member of complex product-family structures

Koala's primary goals are to manage increasing software complexity by using software components, explicit architecture. Architecture description in the first place instead of using round-trip engineering techniques to extract design information from the actual code. Koala manage diversity by reusing of software components, different configuration–compound components, and parameterization of component. The answer of "Why software components?" question in Koala model is to handling the diversity, which is a central key in embedded software in CE, is the reuse of software components in different product. The classical approach of reusability is good for other domain, such as scientific and graphical libraries and reuse of low-level codes doesn't help much in managing the similarities and differences in structure of application.

Component-Oriented approach which is an ideal way to handle the diversity of software in CE allows construction of multiple configurations in both variation and structure. The explicit interface of a component designed in such a way that can be used in many different configurations. Late binding and reusability of software let us apply the same software in different products, which saves product development effort.

Koala got inspiration from Darwin ADL which provides the combination of component model and architectural language. The explicit hierarchical structure of components with provides, interfaces, and bindings make developing CE product families much easier.

The solution key to have different type of configuration is to take out the binding knowledge out of the components.

Koala component model strictly separate component development and configuration, and interface definition uses simple IDL (resembles COM and Java interface description) for defining function prototype in C syntax.

```
interface IVolumeControl {
        void Set Volume (Volume v);
        Volume GetVolume(void); }
```

Component description uses CDL:

```
component Amplifier{
        provides VolumeControl vol;
        requires VolumeControl drv;}
```

Each interface is labeled with two names, the long name which is also the interface type and is globally unique in a particular description in an interfaces repository. The other name is the instance name which is a local name inside the component. This convention makes it possible to have two interfaces on the border of a component with the same interface type.

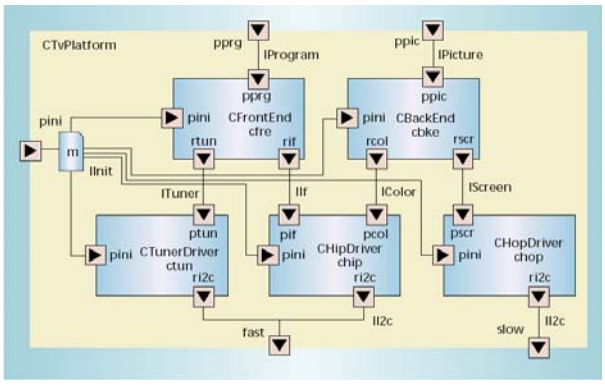

Figure 15, A Koala Compound Component [15]

Configuration is the process of connecting a set of components to form a product. The configuration can be done in different scenarios:

- Many required interface of same type to one provides interface of the same type.
- One provides interface to one/zero required interface of same type.

```
component CTvPlatform{
  provides IProgram pprg; requires II2c slow, fast;
  contains component CFrontEnd cfre;
           component CTunerDriver ctun;
  connects
  pprg =cfre.pprg; cfre.rtun=ctun.ptun; ctun.ri2c=fast; }
```

Module which is interfaceless component solve the problem of initialization interfaces (each subcomponent provides an initialization ➔ glue interfaces). For each module, koala generates a header file with renaming macros.

2.5.1 Implementation

A component is a set of C and header files in a single directory, which may freely include and use each other but may not have any reference to any file outside of the directory. A function f in a provides interface p of a component C with short name c is implemented in C as c_p_f. A function f in a requires-interface r of a component is called as r_f. How does a call of r_f in one component arrive at c_p_f in another component? Simply by a:

#define r_f(...) c_p_f(...)

Such statements are generated by a small tool called Koala that reads CDL and IDL and produces header files to be included by component implementations. Note that the name c_p_f must be globally unique hence the use of c, but the name r_f has as scope only the calling component.

2.6 SOFA

SOFA (SOFtware Appliances) is a project aiming to provide a platform for software components. In SOFA component model, applications are viewed as a hierarchy of nested components. There are two types of components, primitive with no subcomponents or composed building up of other components. In SOFA a component is described by its frame and architecture. A frame, looks like a black-box which defines provides-interfaces and required-interfaces. On the other hand architecture views as a grey-box that defines first level of nesting in a component hierarchy.

There are four types of interface connections:

- Binding of a required-interface to provides-interface
- Delegation from a provides-interface of a component to provides-interface of a subcomponent
- Subsuming from a subcomponent's requires-interface to a requires-interface of component
- Exempting an interface of a subcomponent from any ties.

In SOFA, interfaces, frames, and architectures are described in the Component Description Language (CDL), which is based on OMG IDL.

```
interface Login {
    CentralPlayerServices login(in string who); };
frame Client { provides: Client iClient;
   requires: Login iLogin; CentralPlayerServices iCPS; };
architecture CUNI GameGen implements
GameGenerator {
    inst GameGeneratorDBServices aGGDBS;
    inst ConfigurationFileParser aConfig;
    inst GameGeneratorFunctionality func;
    bind func:iConfig to aConfig:iConfig;
    bind func:iGGDB to aGGDBS:iGGDB;
    subsume aGGDBS:iDatabase to iDatabase; };
```

Example 2, A sample declaration in CDL

The compiled descriptions (interfaces, frame, and architecture) are stored in the Type Information Repository (TIR), which mange an evolution of component's description and can store the several versions of every element defined in CDL. Connectors are first-class entities like component in SOFA. A connector is described in a manner as a component by connector frame and connector architecture. Behavior in SOFA can be defined as communications among SOFA components can be captured formally. The events (method call, returns) in SOFA are denoted by event tokens. For example if there is method m, there are tokens that stand for different events as shown in the table below.

Table 1 Event tokens in SOFA

Token	Event
!m^	emitting a method call
?m^	accepting a method call
!m$	emitting a return
?m$	accepting a return

A sequence of event tokens form a *trace* <!m^; ?m$>. Behavior of a SOFA entity is the set of all traces, which can be produced by the entity. A regular-like expression on the set of all event tokens is called *behavior protocol*. There are three types of behavior protocols, interface protocol which is written by programmer into CDL. Frame protocol which is written by programmer into CDL, and architecture protocol – generated by CDL.

```
interface I {void m(); void n();protocol: m; (n + (n; n))};
frame F{provides: I i1;
        protocol: ?i1.m; (?i1.n + (?i1.n; ?i1.n)); };
```

Example 3, An interface and frame protocols

Dynamic Component UPdating (DCUP) architecture is a specific form of SOFA components which enables their safe updating at runtime. It extends the component model with implementation object and by a technique for updating a component at runtime. A DCUP component can be divided into permanent (is not subject of the dynamic update) and replaceable (is specific for each version of the component)

SOFAnode is a single environment for developing, distributing and running SOFA applications. SOFAnode consists of five logical parts: Template repository, MADE, CDL compiler, Template Information Repository and Code generator

One SOFAnode is not tied with one host – it can be distributed across a network.

2.7 Architecture Description Languages ADLs

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." [18] with other words an ADL is a language that provides features for modeling a software system's *conceptual* architecture, distinguished from the system's *implementation*. Structure is the components that comprise a system, the behavioral specification for those

components, and the patterns and mechanisms for interaction among them. ADLs are emerging as viable tools for formally representing the architectures of systems. While growing in number, they vary widely in terms of the abstraction they support and analysis capabilities they provide.

In contrast to Module Interconnection Languages (MILS), which only describe the structure of an implemented system, ADLs are used to define and model system architecture prior to system implementation. The ADL community generally agrees that Software Architecture is a set of components and connections among them. In addition to identify the components and connectors of a system, ADL typically address the component behavioral specification. ADLs typically provide support for specifying both functional and non-functional characteristics of components. Component protocol specification in some ADLs, such as Wright and Rapide, support the specification of relatively complex component communication protocols. Others such as Unicon allow the type of a component to be specified which in turn restrict the type of the connector that can be used with it. Connector specification in ADL contains structure for specifying properties of connectors, where connectors are used to define the interaction between components.

ADLs are well-suited for representing the architecture of a system and formal, compilable languages. Thus to understand and use ADL technology and architecture concepts effectively, developers need training. Because the structure of a software system can be explicitly represented in an ADL specification, separate documentation describing software structure is not necessary. Examples of ADLs are AADL, ACME, ADML, Aesop, ArTek, C2SADEL, Darwin, Koala, MetaH, Rapide, SADL, UML, UniCon, Weaves, Wright, xADL.

2.7.1 ACME

It was developed jointly by Monroe, Garlan from Carnegie Mellon University and Wile from University of Southern California. It was originally designed to be a lowest common denominator interchange language. Systems are first order entities in Acme, and may also define properties which describe "system-level" attributes. The following is a simple example of a client server system with a single client represented in Acme.

```
System ClientServerSystem = {
   Component server = {Port requests; };
   Component client1 = {Port makeRequest; };
   Connector req = {Role requestor; Role requestee; };
   Attachments { server.requests to req.requestor;
                 client.makeRequest to req.requestee;}}
```

2.7.2 Rapide

It was developed by Luckham at Stanford and has been designed with an emphasis on simulation yielding partially ordered sets of events. Rapide is a language that includes data types, operations, and generation therefore specifications are executable. Rapide is a concurrent, object-oriented, event-based simulation language. In Rapide a *Process d*efines and simulates behavior of distributed object system architectures and produces a simulation represented by a set of events. Posets enable visualization and analysis of an execution. Architecture elements in Rapide are components, connections, and constraints. Component have interfaces which will be implemented by the Architecture and Module. Connection connects send interfaces to receive interfaces via calling actions or functions in its own interface. Three types of connections are basic connection (A to B), pipe connection (A => B), agent connection (A ||> B).

The following is a simple example specified in Rapide.

```
type Producer (Max : Positive) is interface
   action out Send (N: Integer);
   action in Reply(N : Integer);
behavior
  Start => send(0);(?X in Integer) Reply(?X)
  where ?X<Max => Send(?X+1); end Producer;

type Consumer is interface
  action in Receive(N: Integer);action out Ack(N : Integer);
behavior
   (?X in Integer) Receive(?X) => Ack(?X); end Consumer
architecture ProdCon() return SomeType is
  Prod : Producer(100); Cons : Consumer;
  connect
   (?n in Integer) Prod.Send(?n) => Cons.Receive(?n);
   Cons.Ack(?n) => Prod.Reply(?n); end architecture
ProdCon;
```

2.7.3 Wright

It was developed by Garlan at CMU. Wright has been designed with an emphasis on analysis of communication protocols. It uses a variation of Communicating Sequential Processes (CSP) to specify the behaviors of component, connectors, and systems. A language primarily focuses on the basic component./connector/system paradigm. The following is a simple example specified in Wright.

```
System simple_cs
Component client = port send-request = [behavioral spec]
                   spec = [behavioral spec]
Component server =
   port receive-request= [behavioral spec]
   spec = [behavioral spec]
Connector rpc = role
   caller = (request!x -> result?x ->caller) ^ STOP role
   callee = (invoke?x -> return!x -> callee) [] STOP
   glue = (caller.request?x -> callee.invoke!x ->
        callee.return?x -> callee.result!x -> glue) [] STOP
Instances
```

```
  s : server    c : client   r : rpc
Attachments :
  client.send-request as rpc.caller
  server.receive-request as rpc.callee
end simple_cs.
```

2.7.4 xADL

An extension to ADLs which is ADL-neutral interchange format [19]. xADL is designed for representing architectures as hypertext. The ArchStudio project under development in the University of California, Irvine has created xADL as for its "enabling architecture-centric tool integration with XML." ArchStudio is "an extensible, integrated software architecture development environment that uses an XML-based representation for the underlying architecture.

3. Categories of Component Models

The Component Models can be classified based on Distributed Application Support (Middleware, Client-Server), and Interaction Mechanism (Direct-message passing, Indirect-message passing)

An application made up of distinct components running in separate runtime environments, usually on different platforms connected via a network. Typical **distributed applications** are two-tier (**client-server**), three-tier (**client-middleware-server**), and multi-tier (client-multiple middleware-multiple servers)

Table 2, Distributed Application Support

Support Distributed Application	Component Models
Client-Middleware-Server	CORBA,COM,DECOM,EJB
Client-Server	CORBA,COM,DECOM,EJB,ADLs
Client-side	JavaBeans, Koala,

Connectors are meant to encapsulate interaction or communication between components. This leads to mechanism for message passing which can be either directly or indirectly by invoking one another's operations by method call via a channel such as a bus.

Table 3, Interaction Mechanism

Interaction Mechanism	Component Models
Direct message-passing	COM, CORBA, EJB
Indirect message- passing	JavaBeans, ADLs, Koala, SOFA

4. Concluding Remark

In this paper we reviewed the widely used component models from a technical and strategy view in order to have better understanding for developing a new component model. We also showed in which area or domain they are mostly used. The evolution of certain component models shows the achievement of interoperability on all levels to solve the problem of interaction/connectivity of software across process boundaries. The current component technologies are designed to allow clients to communicate transparently with objects, regardless of where those objects are running, in the same process, on the same machine, or on different machines. Although a decade of research in the area of CBSD, it seems there is still a long way to what component-based development promises. As in [16] they have pointed out, there is still no universally accepted terminology, and that's why a software component, is defined in many different ways. Although we can refer to the most widely used definition of software component [1], but it seems that we cannot give a well-defined and generally-accepted answer to the questions *"what is a software component?"* and *"how do we correctly compose software components?*

This research is supported by eScience Fund SF0704, Ministry of Science Technology and Innovation

5. Reference

[1] Clemens Szyperski, D. Gruntz., and S. Murer, *Component Software:Beyond Object-Oriented Programming*. 2002: Addison-Wesley, second edition.
[2] Hamilton, G., *Sun Microsystems, JavaBean*, S.M. Inc., Editor. 1997, August 8.
[3] Kung-Kiu Lau, P.V.E., Zheng Wang. *Exogenous Connectors for Software Components*. in *Proceedings of Eighth International SIGSOFT Symposium on Component-based Software Engineering*. 2005: Springer-Verlag.
[4] Jubin, H., *JavaBeans by example*. 1997: Prentice-Hall, Inc.
[5] Sun Microsystems, L.G.d., *Enterprise JavaBeans Specification, Version 2.1*. 2003.
[6] Manson-Haefel, R., *Enterprise JavaBeans*, ed. M. Loukides. 1999: O'Reilly & Associates, Inc.
[7] MSDN, M. *Inter-Object ommunication.* 2007 [cited; Available from: http://msdn2.microsoft.com/en-us/library/ms694309.aspx.
[8] Comella-Dorda, S. *Component Object Model (COM), DCOM, Related Capabilities.* 2001 [cited; Available from: http://www.sei.cmu.edu/str/descriptions/com.html.
[9] *The Component Object Model Specification Version 0.9*. 1995, October 24 [cited; Available from: http://www.microsoft.com/com/resources/comdocs.asp.
[10] Inc, O. *CORBA® BASICS.* 2007 [cited; Available from: http://www.omg.org/gettingstarted/corbafaq.htm.
[11] OMG, I., *CORBA cOMPONENT mODEL sPECIFICATION*, in *OMG Available Specification*. 2006.
[12] OMG Inc, *Trading Object Services Specification*. 2004.
[13] Group, O.M., *Naming Service Specification*. 2004.
[14] Inc, O. *Information comparing DCOM (ActiveX)to CORBA*.1997 [cited; Available from: http://www.omg.org.
[15] Rob van Ommering, F.v.d.L., Jeff Kramer, Jeff Magee, *The Koala Component Model for Consumer Electronics Software*. 2000.

[16] Kung-Kiu Lau, Z.W. *A Survey of Software Component Models.* in *Software Engineering and Advanced Applications*. 2005. 31 st EUROMICRO Conference: IEEE Computer Socity.
[17] František Plášil, D.B., Radovan Janecek. *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*. in *International Conference on Configurable Distributed Systems*. 1998.
[18] Bass, C., and Kazman, *Software Architecture in Practice, Second Edition.* 2003, Boston, MA: Addision-Wesley.
[19] Rohit Khare, M.G., Peyman Oreizy. *xADL: Enabling Architecture-Centric Tool Integration With XML*. in *34th Annual Hawaii International Conference on System Sciences (HICSS-34)*. 2001. Hawaii.
[20] Inc, O., *Object Management Group Documents*. 2004.
[21] Dušan Bálek, F.P. *SOFTWARE CONNECTORS AND THEIR ROLE IN COMPONENT DEPLOYMENT*. in *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*. 2001: Kluwer, B.V.
[22] Cynthia Della Torre Cicalese, S.R., *Behavioral Specification of Distributed Software Component Interfaces.* IEEE Computer Socity Press, 1999. **32**(7): p. 46-53.
[23] Cory Vondrak, T., Redondo Beach, CA. *Remote Procedure Call*. 1997 [cited February 2007]; Available from: http://www.sei.cmu.edu/str/descriptions.
[24] Bernstein, P.A., *"Middleware: A Model for Distributed Services."* Communication of the ACM, 1996. **39**(2): p. 86-97.
[25] Chappell, D. *DCE and Objects*. 1996 [cited; Available from: http://www.opengroup.org/dce/info/dce_objects.htm.
[26] Jim Waldo, G.W., Ann Wollrath, Sam Kendall, *A Note on Distributed Computing.* Sun Microsystems Laboratories, Inc, 1994.
[27] Brain N. Bershad, T.E.A., Edward D. Lazowska, and Henry M. Levy, *Lightweight Remote Procedure Call.* ACM, 1989. **089791-338-3**.
[28] Andrew D. Birrel and Bruce Jay Nelsonl, *Implementing Remote Procedure Calls* , ACM Transaction on Computer System, Vol. 2, No. 1, February 1984, Pages 39-59.

Mohd Hasan Selamat received his M.S. degrees in Computer Science from Essex University in 1981 and MPhil in Information System from East Anglia University, United Kingdom in 1989. His research areas include software engineering and information system. He is now a full-time lecturer and Head Department of Information System in the Faculty of Computer Science and Information Technology, University Putra of Malaysia. He has published a number of papers related to these areas.

Hamid Sanatnama, received his M. Sc. form Gothenburg University in 1998. After working as a software engineer from (1999) and (2003) in C&S Healthcare consultant company and Volvo technological development department in Sweden respectively, and from (2003) he started as instructor in the Shahid Bahonar University of Kerman. He is now a PhD student at University Putra Malaysia and doing his research in field of software engineering.

Abdul Azim Abdul Ghanis received his M.S. degrees in Computer Science from University of Miami, Florida, U.S.A in 1984 and Ph.D. in Computer Science from University of Strathclyde, Scotland, U.K in 1993. His research areas include software engineering, software metric and software quality. He is now a full-time lecturer in Department of Information System and Dean of the Faculty of Computer Science and Information Technology, University Putra of Malaysia. He has published a number of papers related to software quality areas.

Rodziah Binti Atan received her M.S. degree in Computer Science in 2001 and PhD in Software Engineering in 2006 from University Putra Malaysia. Her research areas include software process modeling, software measurement. She is now a fulltime lecturer in Department of Information System.