Comparative Assessment of Network-Centric Software Architectures

LIKHITA KRISHNAMURTHY

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Dr. Osman Balci, Chair
Dr. James D. Arthur
Dr. Roger W. Ehrich

May 1, 2006

Blacksburg, Virginia

Keywords and phrases: architecture, architecture evaluation, comparative assessment, network-centric, software architecture, software architecture comparison, system architecture

Comparative Assessment of Network-Centric Software Architectures

LIKHITA KRISHNAMURTHY

Abstract

The purpose of this thesis is to characterize, compare and contrast four network-centric software architectures, namely Client-Server Architecture (CSA), Distributed Objects Architecture (DOA), Service-Oriented Architecture (SOA) and Peer-to-Peer Architecture (PPA) and seven associated frameworks consisting of .NET, Java EE, CORBA, DCOM, Web Services, Jini and JXTA with respect to a set of derived criteria. Network-centric systems are gaining in popularity as they have the potential to solve more complex problems than we have been able to in the past. However, with the rise of SOA, Web Services, a set of standards widely used for implementing service-oriented solutions, is being touted as the "silver bullet" to all problems afflicting the software engineering domain with the danger of making other architectures seem obsolete. Thus, there is an urgent need to study the various architectures and frameworks in comparison to each other and understand their relative merits and demerits for building network-centric systems.

The architectures studied here were selected on the basis of their fundamentality and generality. The frameworks were chosen on the basis of their popularity and representativeness to build solutions in a particular architecture. The criteria used for comparative assessment are derived from a combination of two approaches – by a close examination of the unique characteristics and requirements of network-centric systems and then by an examination of the constraints and mechanisms present in the architectures and frameworks under consideration that may contribute towards realizing the requirements of network-centric systems. Not all of the criteria are equally relevant for the architectures and frameworks. Some, when relevant, are relevant in a different sense from one architecture (or framework) to another.

One of the conclusions that can be drawn from this study is that the different architectures are not completely different from each other. In fact, CSA, DOA and SOA are a natural evolution in that order and share several characteristics. At the same time, significant differences do exist, so it is clearly possible to judge/differentiate one from the other. All three architectures can coexist in a single system or system of systems. However, the advantages of each architecture become apparent only when they are used in their proper scope. At the same time, a sharp difference can be perceived between these three architectures and the peer-to-peer architecture. This is because PPA aims to solve a totally different class of problems than the other three architectures and hence has certain unique characteristics not observed in the others. Further, all of the frameworks have certain unique architectural features and mechanisms not found in the others that contribute towards achieving network-centric quality characteristics. The two broad frameworks, .NET and Java EE offer almost equivalent capabilities and features; what can be achieved in one can be achieved in the other.

This thesis deals with the study of all the four architectures and their related frameworks. The criteria used, while fairly comprehensive, are not exhaustive. Variants of the fundamental architectures are not considered. However, system/software architects seeking an understanding of the tradeoffs involved in using the various architectures and frameworks and their subtle nuances should benefit considerably from this work.

Acknowledgments

First and foremost, I wish to express my heartfelt gratitude to my advisor, Dr. Osman Balci for giving me the opportunity to work on this thesis. What I have learned from him while working on this thesis, will, I believe, benefit me for a lifetime. I would also like to thank Dr. James D. Arthur for his invaluable guidance and support.

On the personal front, I am grateful to my parents Krishnamurthy Hegde and Laxmi Hegde for their unconditional support as always. Finally, I thank Swapneel Mehta for his constant encouragement and assistance throughout this endeavor.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ADL	Architecture Description Language
ADO	ActiveX Data Objects
AOP	Aspect-Oriented Programming
API	Application Programming Interface
ASMX	Active Server Methods
ASP	Active Server Pages
AV	All View
C4ISR	Command, Control, Communication, Computer, Information, Surveillance and Reconnaissance
CAS	Code Access Control
CBA	Component-Based Architecture
CCM	CORBA Component Model
CLI	Common Language Infrastructure
CLR	Common Language Runtime
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CSA	Client-Server Architecture
DCOM	Distributed Component Object Model
DODAF	DoD Architecture Framework
EJB	Enterprise JavaBean
FoS	Family of Systems
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
IDL	Interface Definition Language
IIOP	Internet Inter Orb Protocol
IL	Intermediate Language
IoC	Inversion of Control
J2EE	Java 2 Platform, Enterprise Edition
JAAS	Java Authentication and Authorization Service
JAX-WS	Java API for XML Web Services
JAXB	Java Architecture for XML Binding
JCA	Java Connector Architecture
JCE	Java Cryptography Extension
JCP	Java Community Process
JDBC	Java Database Connectivity
JITA	Just In Time Activation
JMS	Java Messaging Service
JNDI	Java Naming and Directory Interface
JSF	Java Server Faces
JSP	Java Server Pages
JSTL	JavaServer Pages Standard Tag Library
JVM	Java Virtual Machine
JXTA	Juxtapose

LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
MDB	Message-Driven Bean
MSMQ	Microsoft Message Queueing
MVC	Model-View-Controller
NCW	Network-centric Warfare
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
ORPC	Object Remote Procedure Call
OSJTF	Open Systems Joint Task Force
OV	Operational View
POJO	Plain Old Java Object
PPA	Peer-to-Peer Architecture
RFC	Request For Comments
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SCSL	Sun Community Source License
SEI	Software Engineering Institute
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SoS	System of Systems
SSL	Secure Socket Layer
SV	Systems View
TCP/IP	Transmission Control Protocol/Internet Protocol
TV	Technical Standards View
UDDI	Universal Description, Discovery and Integration
URI	Universal Resource Identifier
VPN	Virtual Private Network
W3C	World Wide Web Consortium
WAN	Wide Area Network
WSDL	Web Services Description Language
WSE	Web Services Extensions
XML	eXtensible Markup Language

Chapter 1: Introduction

The "Network" is everywhere. The omnipresence of the "network", be it the Internet, wireless networks or other kinds of networks like Virtual Private Networks (VPNs), is driving computing towards a network-centric model where systems and applications are deployed over and accessed through a network. Various forces are responsible for this paradigm shift. For single applications, it could be the ease of application deployment, updating and maintenance. Increased commodification of traditional services and the ease of building complex systems by aggregating capabilities rather than constructing them from scratch as a monolithic system are further contributing factors. Moreover, the systems of the future shall be more complex, catering to larger groups of users. Most often than not, they will be System of Systems (SoS) that aggregate the capabilities of many individual systems. The nature and intended function of these complex systems could be inherently network-centric - requiring them to be deployed onto various nodes that communicate and collaborate over a network. Examples of these classes of systems are systems that disseminate/aggregate information and data from various geographically distributed sources and help to form a coherent picture from this aggregated information. For systems and applications that are not required to be geographically distributed, considerations of various quality characteristics like scalability and resilience might require them to be distributed onto different nodes connected by a network, and to collaborate over the network to achieve some common function.

1.1 What is a Network-Centric System?

A network-centric system is an interconnection of hardware, software, and humans that operate together over a network (e.g., Internet, virtual private network, local area network, intranet) to accomplish a set of goals.

The adjective "network-centric" has been coined in the Department of Defense (DoD) to refer to a class of systems, which is mandated for DoD components to build/use for transforming their operations and warfare capabilities to be network-centric. The terms "Network-centric Operations", "Network-centric Warfare", and FORCEnet currently constitute a common terminology in DoD. The Navy has its own flagship organization called NETWARCOM (Naval Network Warfare Command). The major distinguishing characteristic of this class of systems is the fact that the components (or subsystems or modules) of this type of system communicate with each other over a network. For example, the space shuttle or an aircraft is a complex system, but it is not a network-centric system as its components do not communicate over a network. A supply chain system operating over a company's virtual private network with geographically dispersed employees using the system with their PDAs, cell phones, laptops, and PCs is a network-centric system. The adjective "network-centric" is not just for DoD systems, but for any kind of system, which possess characteristics of this class of systems.

While Network-centric systems share many characteristics with Distributed systems, they are not another name for distributed systems. The term "distributed systems" typically implies that the system operation is distributed for performance improvement reasons. It also implies that the system is engineered to have distributed components. The terms "distributed processing" and “parallel processing” are used purely for performance improvement. On the other hand, the term “network-centric” implies that the system can be composed by way of reuse of already existing systems or subsystems over a network. The term “Distributed Systems” refers to the old local area or wide area networks. Thus, in the new era of the Internet with which we have witnessed many paradigm changes, the new term is “Network-centric” is more appropriate.

A network-centric system consists of hardware, software, and humans as depicted in Figure 1.

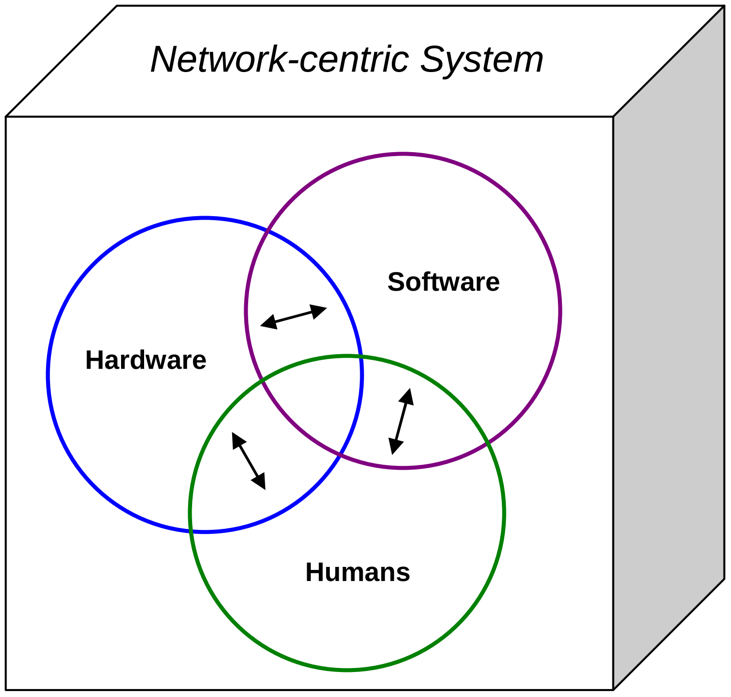

Figure 1 Major components of a Network-Centric system

The term “Network-centric System” refers to a class of systems. Example systems that belong to this class include the following:

1.1.1 System of Systems

A system of systems (SoS) is an interconnection of interdependent systems through a network to provide a given capability. A SoS may be a single platform or consist of a collection of separate, but interdependent, interconnected platforms performing different functions. A military aircraft, for example, is a single platform operating with different systems on board, such as propulsion, weapons, navigation, and communications systems. A ground station dependent on a satellite is an example of interconnected platforms performing different functions. A distinguishing factor for a SoS is that it depends on all of its elements working interactively and continuously within a network to accomplish a pre-specified capability. The loss of any SoS element degrades the performance or capabilities of the entire SoS. A SoS provides a capability not possible with any of the individual elements acting alone. [OUSDATL 2005]

1.1.2 Family of Systems

A family of systems (FoS) is a collection of independent (not interdependent) systems that can be interconnected over a network in various ways to provide different capabilities needed depending on a particular situation. Interoperability of the independent systems is a key consideration in the ad hoc deployment of a FoS. [OUSDATL 2005]

An enterprise-wide system is a system that covers the entire operation of an enterprise such as the U.S. Navy over a network.

1.1.3 Network-Centric Software

From the preceding discussion, we can characterize software components that interact with each other over a network (e.g. Internet, VPN, Local Area Networks, Wireless networks, etc) as network-centric software. This characterization leads us to the following definition of Network-centric software architecture:
"A Network-centric Software Architecture is software architecture with characteristics and organization that make it suitable for building applications and systems that are deployed over networks. Network-centric Software Architecture Frameworks have mechanisms and tactics that support building Network-centric systems".

1.2 Network-Centric Architectures Literature Review

As early as in 2000, the gradual paradigm shift towards Network-centric computing was observed by Garlan [2000]. Garlan also identified challenges and opportunities for research in software architecture due to this paradigm shift.

The term "network-centric" was first used in the military domain in the context of network-centric warfare [DoD 2006a]. As such, a substantial section of the literature on network-centric systems is related to the military domain. Cook [2001] gives a fairly detailed description of SoS and discusses characteristics that differentiate SoS from monolithic systems. This characterization, though primarily from a military perspective, is also applicable to SoS in general. Openness, adaptability, interface-based, loose organization and evolvability are some of the identified characteristics. Fuzak et al. [2001] describe five capabilities required of network-centric architectures based on the seven C4ISR (command, control, communications, computers, intelligence, surveillance, reconnaissance) imperatives [SSC San Diego 2006] that represent command capabilities required by military forces. These include dynamic interoperable connectivity, universal information access, focused sensing and data collection, information operations-assurance and resource planning and management. Fuzak et al. [2001] also envision network-centric systems as a "confederation of pieces" that can evolve through "parts upgrade". Several companies like Oracle [Oracle 2004] and Boeing [Logan 2003, Boeing 2005] have also come up with their own "network-centric" reference architectures.

The four network-centric architectures considered in this thesis, Client-Server Architecture (CSA), Distributed Objects Architecture (DOA), Service-Oriented Architecture (SOA) and Peer-to-Peer Architecture (PPA) have been treated extensively in the literature. Lewandowski [1998] describes CSA in considerable detail. Pressman [2004] describes CSA, DOA, SOA and PPA. Dogac, Dengi, and Öszu [1998] discuss Distributed Objects Architecture in the context of

CORBA. Szyperski [2003] examines components and component characteristics in depth. The benefits and challenges of using components are also discussed. Reiss [2005] recognizes the importance of component Interfaces for building network-based systems and presents an enhanced component model into which semantics for specifying non-functional properties are incorporated. An explosion in literature on SOA can be observed in recent years. Papazoglou and Georgakopoulos [2003], Patrick [2005], Perrey and Lycett [2003], Anand, Padmanabhuni, and Ganesh [2005] and Chung [2005] all deal with various aspects of SOA. Singh [2001] provides an overview of peer-to-peer computing and its main variants. Androutsellis-Theotokis and Spinellis [2004] survey different peer-to-peer content distribution technologies.

For the frameworks considered in this thesis, the primary sources of information are their specifications. Complete specifications for Java EE [Sun Microsystems 2006], CORBA [OMG 2005] (Common Object Request Broker Architecture), Web Services [W3C 2006, OASIS 2006], JXTA [JXTA 2006], and Jini [Jini 2006] are available online and form the definitive sources of information. Microsoft [2005] describes the DCOM architecture in detail. While portions of the .NET framework are standards and specifications are available online, the primary source of information is MSDN (Microsoft Developer Network). [MSDN 2006]

Substantial work has been done on quality attributes by the Carnegie Mellon University Software Engineering Institute. Barbacci et al. [2000] provide a reasoning framework for quality characteristics such as modifiability, scalability, performance, and dependability and discuss architectural tactics for achieving the same. Ellison et al. [2004] provides a similar treatment of the security and survivability quality characteristics. Maeir [2006] and Meyers et al. [2004] highlight the importance of interoperability for building system-of-systems. Acton [2003] highlights the importance of high availability to network-centric systems and elaborates on architectural mechanisms like failover and redundancy that can be used to achieve it.

1.3 Statement of the Problem

To begin with, while substantial research has been done on several quality characteristics, there is a need to identify a fairly comprehensive set of quality characteristics and common capabilities and services that are important to network-centric systems and architectures and characterize them from a network-centric architectural and framework perspective.

Requirements drive the architecture of a software system. The architecture and framework of a system influence its quality attributes and determine its capabilities. Network-centric systems place more emphasis on a certain set of quality characteristics and capabilities than other traditional systems and applications. Identification and characterization of these qualities and capabilities help system designers and architects make more informed decisions about the architecture of the particular system that they are building. Thus, to build network-centric systems that meet expectations, an understanding of the capabilities and deficiencies of the various network-centric architectures in comparison to each other and with respect to network-centric quality characteristics is required.

Several architectures are available currently that can be used to build network-centric systems. Also, whenever a new technology begins to gain momentum in the industry, it is touted as the "silver bullet" to all problems afflicting the discipline. For example, the current trend is to portray SOA as the silver bullet to every problem in enterprise software development. It is often necessary to place this new architecture in context with respect to the other architectures. Thus, choosing the right architecture to satisfy the requirements of a particular network-centric system requires an understanding of the tradeoffs involved in choosing one architecture over another. Therefore, a comparative analysis of these architectures with respect to each other is required for identifying and making these tradeoffs explicit.

Frameworks should be studied in conjunction with their related architectures because software architecture as a discipline has not reached a level of maturity where it can be specified purely in abstract terms without any reference to the underlying platform/framework it is built on. The characteristics of the implementation frameworks influence the architecture in many ways. Their capabilities in the form of implementation support for various abstractions, mechanisms and services can direct the architectural process. Choice of framework dictates many elements of the overall architecture like its structure and nature of components. It also influences the operational environment and often constrains or expands the design choices available. Thus, frameworks add information that is relevant architecturally. Further, the mechanisms and services provided in a framework may be unique or they may be replicated, refined upon and provided in one form or the other in other frameworks. Thus, a comparative study of frameworks in association with their respective network-centric architectures helps to put them in perspective and aids architectural choices.

1.4 Statement of Objectives

The objectives of the research described herein are to identify, compare and contrast various network-centric software architectures and frameworks. A set of network-centric quality characteristics and common capabilities and services are identified and characterized. Following that, four network-centric architectures – CSA, DOA, SOA and PPA are identified, characterized and compared and contrasted with respect to each other from an architectural perspective. For every architecture considered, a set of corresponding frameworks are identified, characterized and compared. Finally, all the architectures and frameworks are evaluated against the identified network-centric qualities, capabilities and services.

1.5 Scope

In this thesis, we consider the software architecture of network-centric systems at the application level. We discuss application-level architectural concerns and protocols. We assume that the underlying network communication fabric (the lower layers comprising the hardware infrastructure used to build the network and the network software and protocols like TCP/IP) provides a certain base-level quality of service. While, as discussed in Chapter 3, certain characteristics associated with network-centricity are often concerns of the communication fabric and less influenced by the architecture at the application layer, we limit our discussion to the application layer. Further, only generic network-centric architectures are considered. Domain specific application architectures, like for military systems, are excluded.

1.6 Overview of Thesis

The remainder of this thesis is organized as follows: Chapter 2 gives a brief description of the identified network-centric architectures: CSA, DOA, SOA and PPA and their associated frameworks: .NET, Java EE, CORBA, DCOM, Web Services, Jini and JXTA. A set of network-centric quality characteristics and common capabilities and services are identified and characterized in Chapter 3. In Chapter 4, the architectures and frameworks are evaluated with respect to architectural and network-centric qualities. Chapter 5 provides concluding remarks.

Chapter 2: Network-Centric Software Architectures

Among the different architectures that are used to build most network-centric systems, there are four dominant architectures, namely, Client-Server Architecture (CSA), Distributed Objects Architecture (DOA), Service-Oriented Architecture (SOA) and Peer-to-Peer Architecture (PPA). Other architectures can be built as variants by composition/combination of these architectures.

2.1 Client-Server Architecture

CSA consists of two kinds of logical entities – clients and servers. “Client/Server” is primarily a relationship between processes running on separate machines where the client is the consumer of the services provided by the server process [Orfali, Harkey, and Edwards 1999]. In CSA, there is a many-to-one relationship between clients and servers. The servers are passive entities that await requests from the clients. The clients always initiate the dialog by requesting a service (Exceptions to this scenario include cases where the client passes a reference to a callback object when it invokes a service).

Different types of CSAs can be distinguished depending on how the application logic is split between the client and the server. In a “fat client” model, more of the application functionality is placed on the client side. Examples for this type of CSA include file servers and database servers. In a “fat server” model or “thin client” model the reverse occurs; most of the application functionality is pushed onto the server side.

Another way of differentiating between different types of CSAs is using the notion of a “Tier”. The idea of “fat clients” and “fat servers” gives an indication of how application logic is partitioned. The notion of a CSA application being N-Tiered is similar, except that it gives more precise information, i.e., how the application logic is partitioned into functional units and the maximum number of machines the application can be distributed onto. The idea of “Tiers” tells us about the physical distribution of logic. The ability to distribute an application onto different machines is achieved by partitioning application logic into distinct logical layers, where each layer performs a set of related functionality. In a 2-tier architecture, most of the application logic is either on the client or on the server. Currently architectures with N usually being three or four are the most popular forms of CSA [SEI 2000]. Examples of 3-Tier applications include Web applications and other kinds of enterprise applications like banking systems. In a 3-tier CSA, the most common functional units are presentation, business/application logic and persistence/data as shown in Figure 2.

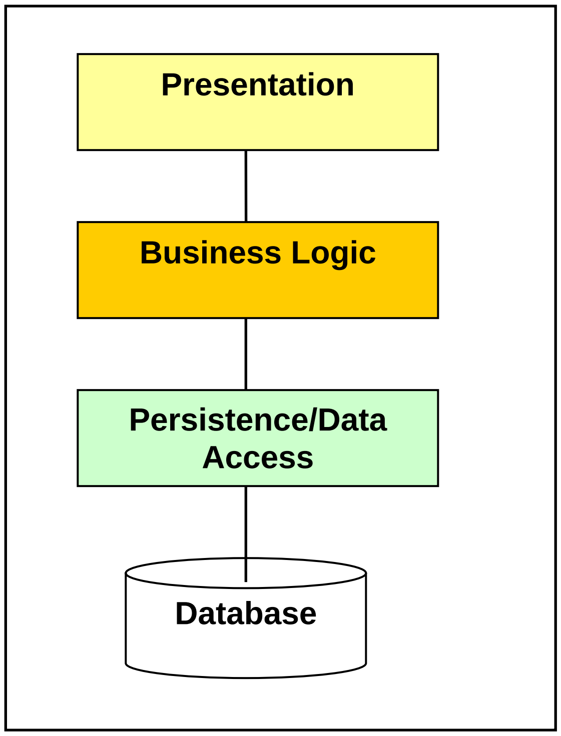

Figure 2 A three-layered application

- Presentation Layer

  The presentation tier is responsible for handling the interaction between the user and the application. It displays information to the user and interprets requests from the user into actions upon the business logic and data source.

- Business/Application Logic Layer

  This layer includes logic for all the business rules, data validation, manipulation, input/output processing and security for the application. Thus, the bulk of the application logic is in this tier.

- Persistence Layer

  This layer is primarily concerned with retrieving, deleting, changing and adding data.

Most early client-server applications were implemented using low-level, conversational peer-to-peer protocols such as sockets, NetBIOS or Named Pipes [Orfali, Harkey, and Edwards 1999]. Currently, communication between the distributed tiers is carried out mainly by using synchronous RPCs (Remote Procedure Calls). Asynchronous communication is also possible using MOM (Message-Oriented Middleware) like MSMQ (Microsoft Message Queuing) or JMS (Java Messaging Service) [SEI 2006g].

The two most popular frameworks used to build CSA applications are the .NET and Java EE frameworks.

2.1.1 The Microsoft .NET Framework

The Microsoft .NET Framework [Microsoft 2006] is a software development platform developed by Microsoft Corporation. As such, it consists of a runtime environment, called the Common Language Runtime (CLR), on which programs developed for .NET run, and a set of types (classes) in the form of libraries. There are two main libraries required for a minimum implementation of .NET: the Base Class Library (BCL), which provides a simple runtime library for modern programming languages, and the Runtime Infrastructure Library, which provides the services needed by a compiler to target the CLR and the facilities needed to dynamically load types from a stream in the file format specified. A hallmark of the .NET framework is its support for multiple languages. Figure 3 provides an overview of the .NET framework architecture.

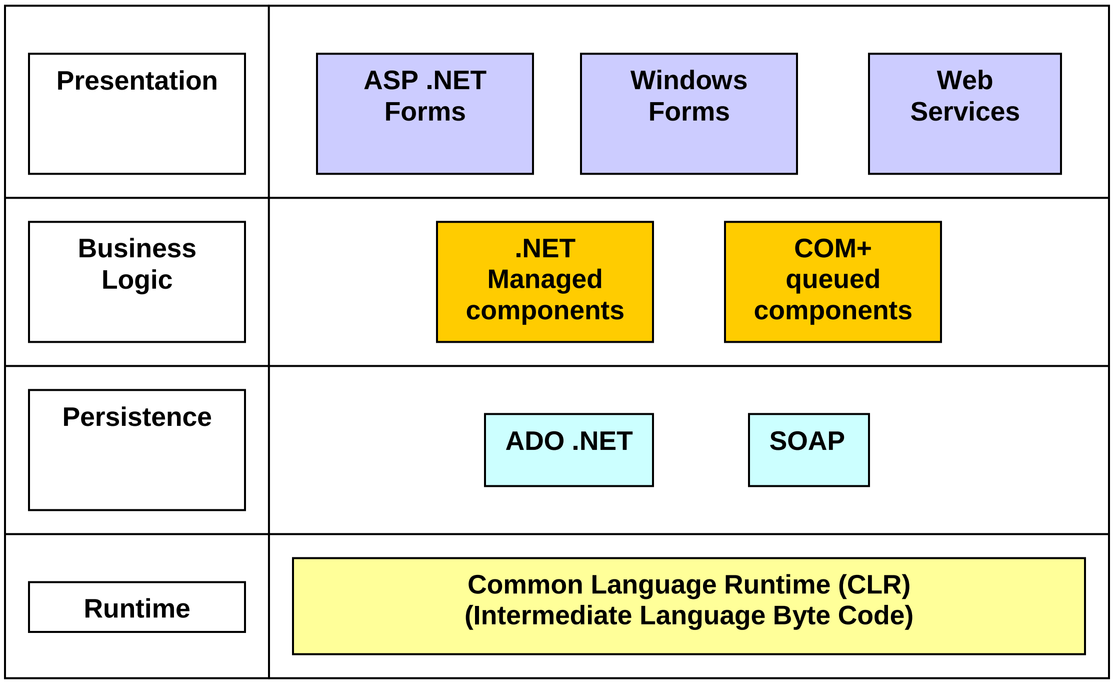

Figure 3 .NET Architecture

The .NET framework is targeted at building two major kinds of applications – 3-tier enterprise applications and service-oriented systems using web services. Therefore, it makes sense to study it in terms of the support it provides for building the tiers of a 3-tier application and Web Services.

2.1.1.1 Presentation

In the .NET framework, the presentation layer for an application is built using either ASP .NET or Windows Forms. ASP .NET is used for thin client web interfaces whereas Windows forms are used for rich client interfaces.

An important feature of ASP .NET is the separation of code and content by using Code-behind files. "Code-behind" means that the code for an ASP.NET page is contained within a separate class file. This permits a clean separation of the HTML from the presentation logic. Another major feature of ASP .NET is its support for data binding through the use of server-side controls. Server-side controls are components that are placed on ASP .NET web forms. When a client requests a page containing these controls, the ASP .NET processor loads and executes them on the server. Data binding is the process of retrieving data from a source and dynamically associating with a property of a visual element [Esposito 2002]. Depending on the context in which the element will be displayed, you can map the element to either an HTML tag or a ASP .NET server side control. Data bound server side controls represent a powerful mechanism for associating rows of data with graphical HTML elements such as drop-down lists or tables.

2.1.1.2 Business Logic

.NET provides the following main services required to build business objects:

2.1.1.2.1 Remoting

The .NET framework includes the remoting subsystem [MSDN 2006f] which allows .NET applications to interact with each other. This includes both cross-process communication and communication across the network from machine to machine.

2.1.1.2.2 Enterprise Services (COM+)

Business logic can be implemented in .NET using classes that leverage COM+ services. When used from .NET, COM+ services are referred to as Enterprise Services. Enterprise Services provides the kind of services provided by an EJB (Enterprise Java Bean) container to components deployed in it. Some of the most commonly used services include:

- Two-phase distributed transactions
- Object pooling
- Queued components
- Role-based security

A .NET component that takes advantage of COM+ services needs to derive from the .NET base class ServicedComponent defined in the System.EnterpriseServices namespace and use various custom attributes to specify the actual services required.

2.1.1.2.3 Queued Components

COM+ Queued Components (QC) service provides a way to invoke and execute components asynchronously using Microsoft Message Queuing (MSMQ).

2.1.1.3 Persistence

In the .NET framework, ADO .NET is used for accessing relational databases and other data sources. ADO.NET includes .NET Framework data providers for connecting to a database, executing commands, and retrieving results. Those results are either processed directly, or placed in an ADO.NET DataSet object in order to be exposed to the user in an ad-hoc manner, combined with data from multiple sources, or remoted between tiers. The ADO.NET DataSet object can also be used independently of a .NET Framework data provider to manage data local to the application or sourced from XML and it provides deep integration with XML.

2.1.1.4 Web Services

.NET provides built-in support for Web Services through ASMX (Active Server Methods) [Skonnard 2006] and WSE (Web Services Extensions) [MSDN 2006j].

2.1.2 The Java Platform, Enterprise Edition (Java EE) Framework

The Java Enterprise Edition (Java EE) Framework is a set of coordinated technologies developed by Sun Microsystems for building multi-tier server side Java-based applications and services. Java Platform, Enterprise Edition 5 (Java EE 5) was earlier known as Java 2 Platform, Enterprise Edition (J2EE). In the latest version Java EE 5, the 2 has been dropped from the platform name in order to simplify it. The Java EE Framework is not an implementation; it is a set of specifications. The technologies covered under the Java EE specification are implemented by various vendors. Like .NET, it includes infrastructure for building Web Services. The new Java EE 5 platform includes the newly redesigned annotations-driven EJB 3 specification as well as JavaServer Faces (JSF), integrated into the platform for the first time. Figure 4 provides a simplified view of the Java EE architecture.

The technologies that comprise the Java EE Framework include:

- Web Services Technologies for implementing Enterprise Web Services
- Component Model technologies

Component Model Technologies can be considered as the heart of the JEE Framework. It consists of specifications for

  - Enterprise Java Beans 3.0 (EJB 3.0),
  - J2EE Connector Architecture 1.5 (JCA)
  - Servlets
  - Java Server Pages (JSP)
  - Java Server Faces (JSF)
  - Java Standard Tag Library (JSTL).

- Management Technologies
- Other Java EE Technologies

Like RFCs for Internet Standards, each Java EE technology is described in a JSR (Java Specification Request) document. Java Specification Requests (JSRs) are the actual descriptions of proposed and final specifications for the Java platform. For example, JSR-220 describes the Enterprise Java Beans 3.0 (EJB 3.0) technology. Java EE specifications are approved and maintained by the JCP (Java Community Process), a consortium that holds the responsibility for the development of Java technology. A specification is initiated by community members and approved for development by the Executive Committee.

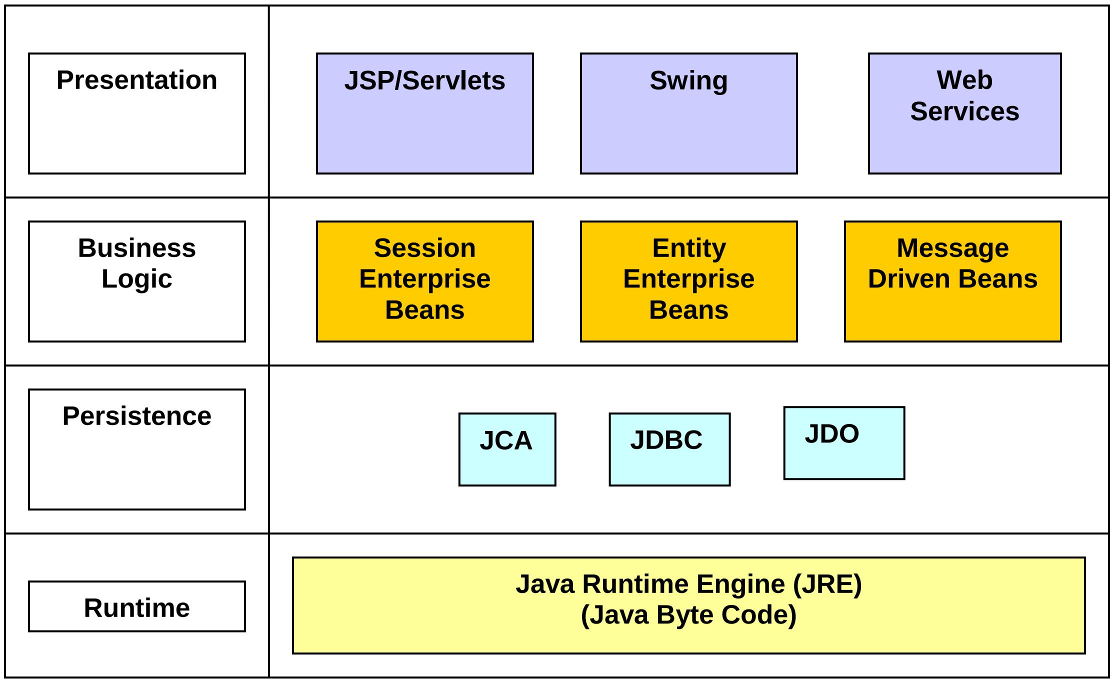

Figure 4 Java EE Architecture

Like .NET, the Java EE platform is intended primarily to build 3-tier enterprise applications and service-oriented systems using Web Services.

2.1.2.1 Java EE Presentation

Java Server Pages (JSPs) are used to build tag-oriented dynamic web pages for accessing remote objects. Dynamic pages can also be built programmatically using Servlets. Swing is used to build rich, interactive GUIs.

2.1.2.2 Java EE Business Logic

In Java EE, Enterprise Java Beans (EJB) hold the application's business logic – the code that implements the functionality of the application. EJBs are server-side components written in the Java programming language [Sun 2005]. There are two kinds of EJBs: Session EJBs and message-driven beans.

2.1.2.2.1 Session EJBs

A session bean represents a single client inside the Application Server. To access an application that is deployed on the server, the client invokes the session bean's methods. As its name suggests, a session bean is similar to an interactive session. A session bean is not shared; it can have only one client, in the same way that an interactive session can have only one user. Like an interactive session, a session bean is not persistent. (That is, its data is not saved to a database.) When the client terminates, its session bean appears to terminate and is no longer associated with the client.

There are two types of session beans: stateful and stateless.

2.1.2.2.1.1 Stateful Session Beans

In a stateful session bean, the instance variables represent the state of a unique client-bean session. Because the client interacts (i.e. talks) with its bean, this state is often called the conversational state. The state is retained for the duration of the client-bean session.

2.1.2.2.1.2 Stateless Session Beans

A stateless session bean does not maintain a conversational state with the client. When a client invokes the method of a stateless bean, the bean's instance variables may contain a state, but only for the duration of the invocation. When the method is finished, the state is no longer retained. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client.

2.1.2.2.2 Message-Driven Beans (MDB)

A message-driven bean is an enterprise bean that allows Java EE applications to process messages asynchronously. It normally acts as a JMS message listener, which is similar to an event listener except that it receives JMS messages instead of events.

2.1.2.3 Persistence

Java EE 5 introduces a new Java Persistence API [JCP 2006a] that greatly simplifies entity bean persistence by using lightweight “entity objects”. Unlike EJB components that use container-managed persistence (CMP), entity objects using the new APIs are no longer components, but POJOs (Plain Old Java Objects). This approach leads to a simpler and more lightweight programming model. The new entity objects provide an object-oriented view of the data stored in a relational database. The specification also standardizes how such object-relational mapping information is provided.

In Java EE, an application communicates with a data storage system using JDBC. The Java Connector Architecture (JCA) allows Java EE components to access different legacy enterprise information systems.

2.1.2.4 Web Services

Web Services are built in Java EE using JAX-WS [JCP 2005a]. JAX-WS stands for Java API for XML Web Services. The starting point for developing a JAX-WS web service is a Java class annotated with the javax.jws.WebService annotation. The WebService annotation defines the class as a web service endpoint.

2.1.2.5 Design Frameworks

The aim of Java EE design frameworks is to make the task of developing Java EE applications easier. A framework provides an abstraction over low level infrastructure APIs. A well designed framework provides structure and consistency to applications. The design frameworks are frameworks "over" Java EE.

2.1.2.5.1 The Spring Design Framework

The Spring Design Framework was developed to deal with the complexity inherent in developing using EJB. The Spring Framework is a "full-stack" Inversion of Control (IoC) Java EE design framework. Spring includes:

- A complete lightweight container,
- A common abstraction layer for transaction management,
- A JDBC abstraction
- Integration with Toplink, Hibernate, JDO, and iBATIS SQL Maps
- AOP (Aspect-Oriented Programming) functionality
- A flexible MVC web application framework

The Spring design framework has been ported to .NET. Spring, at its most base layer has a lightweight container, a glorified object factory that creates objects, configures them and resolves dependencies between them. But on top of that Spring offers things like AOP support, various helper classes for doing EJB development, and sort of a whole slew of transaction management facilities. In some sense there's a lot of the same stack found in EJBs.

2.1.2.5.2 The Hibernate Design Framework

Hibernate is an open source Object Relation Mapping design framework for Java EE. It was developed in response to the perceived cumbersomeness of the persistence mechanism provided with the standard Java EE (the J2EE) specifications – entity beans. Due to their deployment within an EJB container, Entity beans were perceived to be more difficult to test. Also, they were lacking in ability to manage relationships between persistent objects, the query language was inadequate. Entity beans were, in short, perceived to be underspecified [Johnson 2004]. The aim of the Hibernate framework is to provide transparent persistence to plain old java objects (POJO).

2.1.2.5.3 The Struts Design Framework

Struts [Struts 2006] is a design framework for building Web applications based on the Front-Controller design paradigm. The Apache Struts Project now consists of two design Frameworks, namely:

- Struts Action Framework
  Struts Action is the original request-based framework.

- Struts Shale Framework
  Struts Shale is a component-based framework for JavaServer Faces.

2.1.2.5.4 The Tiles Design Framework

Tiles is a Java EE design framework that allows users to provide a consistent user interface, to display portlet-like rectangles of content within a larger page of content, and to download and process just one section of the image at a time, decreasing bandwidth needs. Through a central XML file that defines screens and a set of tags that can be embedded in JSP pages for the insertion of dynamic/static content, Tiles lets users build componentized views and assemble them as they choose.

Figure 5 gives an overview of how the .NET and Java EE frameworks compare against each other for a 3-tier CSA application.

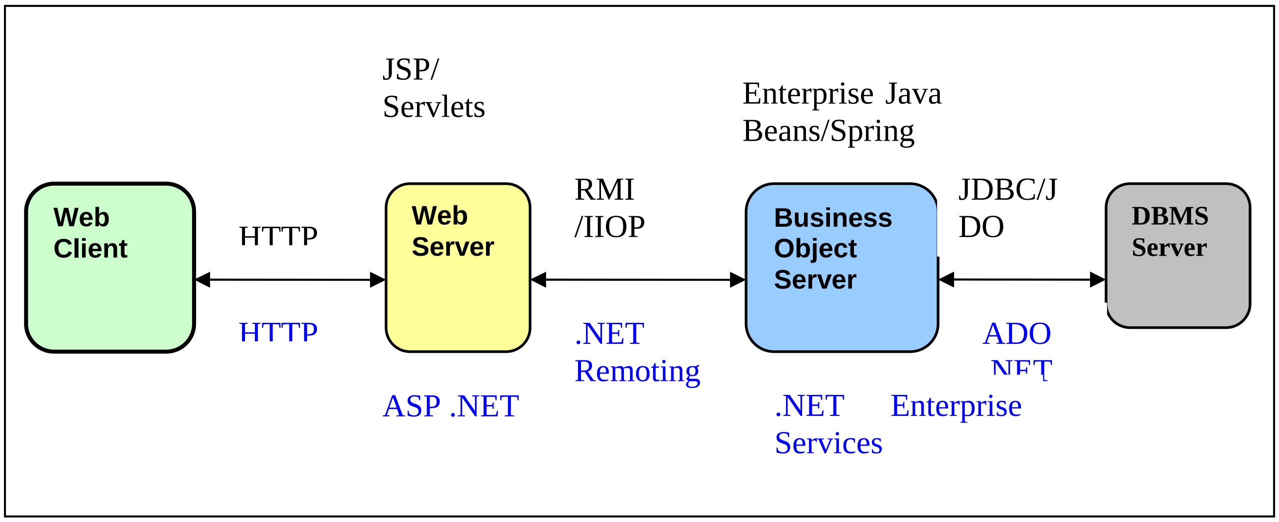

Figure 5 .NET vs. Java EE

2.2 Distributed Objects Architecture

In Client-Server programming, nothing prevents us from using Structured Modular programming or shell scripts to implement both client and server application logic. In DOA, the application logic is organized as objects and distributed over multiple networked hosts. These objects collaborate over the network to provide the overall functionality using method invocation as a communication primitive [Emmerich and Kaveh 2001]. The invoking object is called as the "client object" and the remote object on a different host whose method is being invoked is called as the "server object". Since this invocation happens over a network, a reference to the remote object has to be obtained by the client object. Infrastructure software (often referred to as "middleware") that provides a level of abstraction over the network-layer protocols like TCP/IP is used to achieve this remote invocation of a method.

One thing to be noted is that the distribution of the logic is transparent. The client object thinks it is calling a local object. The task of actually making the call over the network is taken over by the infrastructure software.

The three most famous frameworks in this paradigm are DCM (Distributed Component Model), CORBA and DCOM.

2.2.1 Distributed Component Model

Components are the units of processing in DCM. Syzperski [1998] defines a software component as "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

The core principles of Component-Oriented programming are:

- Separation of interface from implementation

In component-based programming, the basic unit in an application is a binary-compatible interface. An interface defines a set of properties, methods, and events through which external entities can connect to, and communicate with, the component. According to Lowy [2003], this principle contrasts with the object-oriented view of the world that places the object rather than its interface at the center. Lowy [2003] further says that in component-based programming, the server is developed independently of the client.

- Location transparency

Location transparency allows components to be distributed onto different machines without hardcoding their location into the client code. This allows the location of the components to be changed without requiring changes to the client code and recompilation.

Components are usually at a higher level of abstraction than objects and are explicitly geared towards reuse. Components differ from other types of reusable software modules in that they can be modified at design time as binary executables. In contrast, libraries, subroutines, and so on must be modified as source code [Krieger and Adler 1998].

Component standards specify how to build and interconnect software components. They show how a component must present itself to the outside world, independent of its internal implementation. Current popular component standards include .NET, Java EE and CORBA who provide support for the distributed component model through Enterprise Services [Nagel 2005], Enterprise Java Beans (EJB) and CORBA Component Model (CCM) respectively.

Components often exist and operate within containers, which provide a shared context for interaction with other components. Containers also offer common access to system-level services for a component's embedded components (such as process threads and memory resources). Containers are themselves are typically implemented as components, which can be nested in other containers. An example is embedding widget field arrays into panels within GUI windows.

Event-based protocols are commonly used to establish the relationship between a component and its container. Compliant containers all support the same set of interfaces which means that components can freely migrate between different containers at runtime without the need of reconfiguration or recompilation. Containers themselves run on application servers, which offer services offered by the underlying middleware systems such as transactions, security, persistence and notification. Also, server components are often multithreaded, replicated, and pooled, to achieve scalability and reliability. Consequently server components cannot readily be organized into static containment hierarchies.

2.2.2 Common Object Request Broker Architecture

CORBA, an acronym for Common Object Request Broker Architecture, is a suite of specifications being standardized by the Object Management Group [OMG 2005] for a distributed object architecture and infrastructure. The CORBA technology can be used for building applications as a collection of distributed objects/components that collaborate over a network. It provides the mechanism for exposing an object's methods to remote callers (to act as a server) and for discovering such an exposed server object within the CORBA infrastructure (to invoke it as a client). CORBA objects can act as servers and clients simultaneously.

CORBA uses a platform-independent interface definition language (IDL) as a common denominator. It is used for the definition of the calling interfaces and their signatures. An IDL compiler is a tool that a platform vendor must provide. It compiles the IDL file into platform-specific stub code and maps the parameter types to platform-specific types. An IDL compiler can generate both the client stubs and the server skeleton code. 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. Thus, CORBA is language independent, provided that there is a mapping from the language constructs to the IDL. In CORBA lingo, an implementation programming language entity that defines the operations that support a CORBA IDL interface is called as a “Servant”.

The heart of the CORBA specification is the Object Request Broker (ORB), a common communication software bus for objects. An ORB makes it possible for CORBA objects to communicate with each other by connecting objects making requests (clients) with objects servicing requests (servers). Interoperability is implemented by ORB to ORB communication. A CORBA ORB transparently handles object location, object activation, parameter marshalling, fault recovery, and security. Figure 6 shows the structure of an ORB in terms of the various interfaces supported by it.

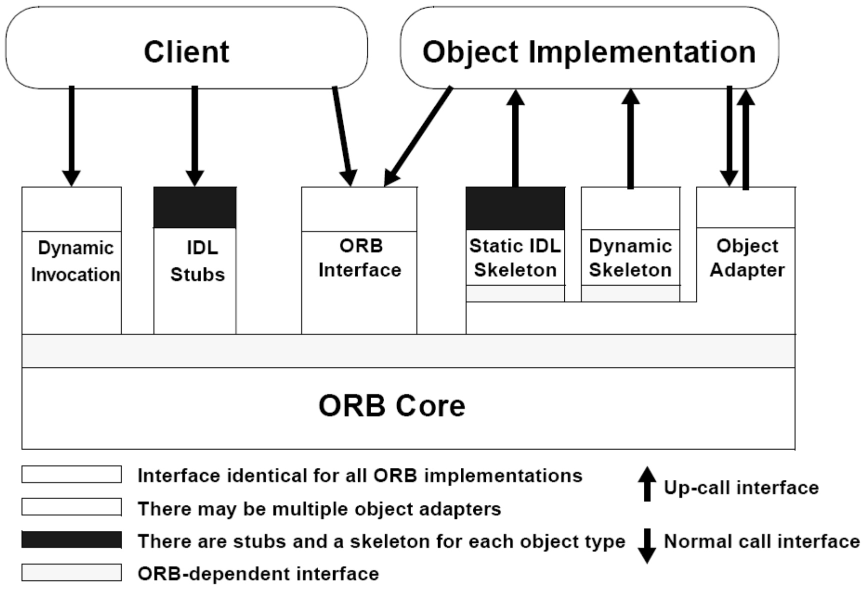

Figure 6 CORBA ORB Interfaces (Image taken from [OMG 2006])

The ORB is also the custodian of the Interface Repository (abbreviated variously IR or IFR), an OMG-standardized distributed database containing IDL interface definitions. The ORB offers a number of services for the manipulation of objects. It provides interface definitions from the IFR, and constructs invocations for use with the Dynamic Invocation Interface (DII).

2.2.2.1 Dynamic Invocation Interface

The DII allows clients to generate requests at run-time. In this approach, the client has no stub connecting it to the server and therefore must dynamically construct its request. To achieve this, the client uses the ORB's DII which provides access to a database containing the descriptions of the interfaces of all the servers that are available in the system. The client thus finds the information about the operations that it can invoke on the objects. This flexibility is useful when an application has no compile-time knowledge of the interface it is accessing. While both static and dynamic invocation support synchronous and one-way communication, only dynamic invocation supports deferred synchronous communication.

2.2.2.2 Dynamic Skeleton Interface (DSI)

The DSI is the server's analogue to the client's DII. The DSI allows an ORB to deliver requests to a servant that has no compile-time knowledge of the IDL interface it is implementing.

2.2.2.3 Object Adapter

An Object Adapter connects an incoming request using an object reference with the proper code to service that request [Sun 2002a]. The Portable Object Adapter (POA) is a kind of object adaptor which is designed to support constructing object implementations that are portable among different ORB implementations, provide support for objects with persistent identities, provide support for transparent object activation and allow a single servant to support multiple object identities simultaneously [Sun 2002a].

2.2.2.4 *Implementation Repository*

The Implementation Repository contains information that allows an ORB to activate servers to process servants. Most of the information in the Implementation Repository is specific to an RB or OS environment. In addition, the Implementation Repository provides a common location to store information associated with servers, such as administrative control, resource allocation, security, and activation modes.

2.2.2.5 *CORBA Architecture*

The CORBA specification is embedded in another embracing architecture, the Object Management Architecture (OMA) shown in Figure 7. In addition to providing users with a language and a platform-neutral remote procedure call specification, CORBA defines commonly needed services such as naming, persistence, life cycle, event notification, transactions and security. These services are implemented in the form of objects connected to the ORB and are described by an IDL interface.

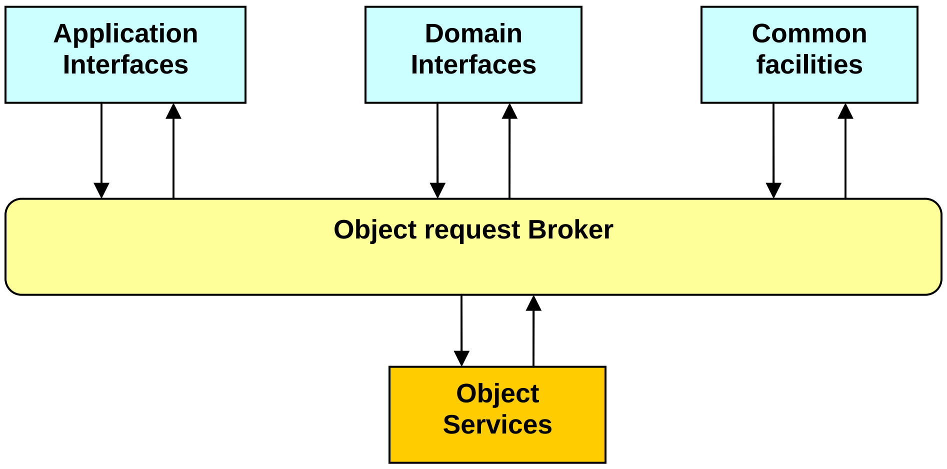

Figure 7 CORBA Architecture

2.2.2.6 *CORBA Component Model*

CORBA Component Model (CCM) is an addition to the family of CORBA definitions. It was introduced with CORBA 3, and it describes standard application framework for CORBA components. It could be used for programming languages other than Java while achieving interoperability with EJB components [Emmerich and Kaveh 2001].

The CCM has a component container, where software components can be installed. The container offers a set of services that the components can use. These services include (but are not limited to) authentication, persistence and transaction management. These are all the most used services a distributed system requires, and by moving the implementation of these services from the software components to the component container, the complexity of the components is dramatically reduced.

2.2.3 Distributed Component Object Model

Distributed Component Object Model (DCOM) is an extension of COM developed by Microsoft in 1996. It allows two objects, one acting as a client and the other acting as the server object, to communicate regardless of whether the two objects are on the same or on different machines. This communication structure is achieved using a proxy object in the client and a stub in the server.

When client and component reside on different machines, DCOM simply replaces the local inter-process communication with a network protocol. The COM run-time provides object-oriented services to clients and components and uses DCE-RPC and the security provider to generate standard network packets that conform to the DCOM wire-protocol standard. Figure 8 provides an overview of the DCOM architecture.

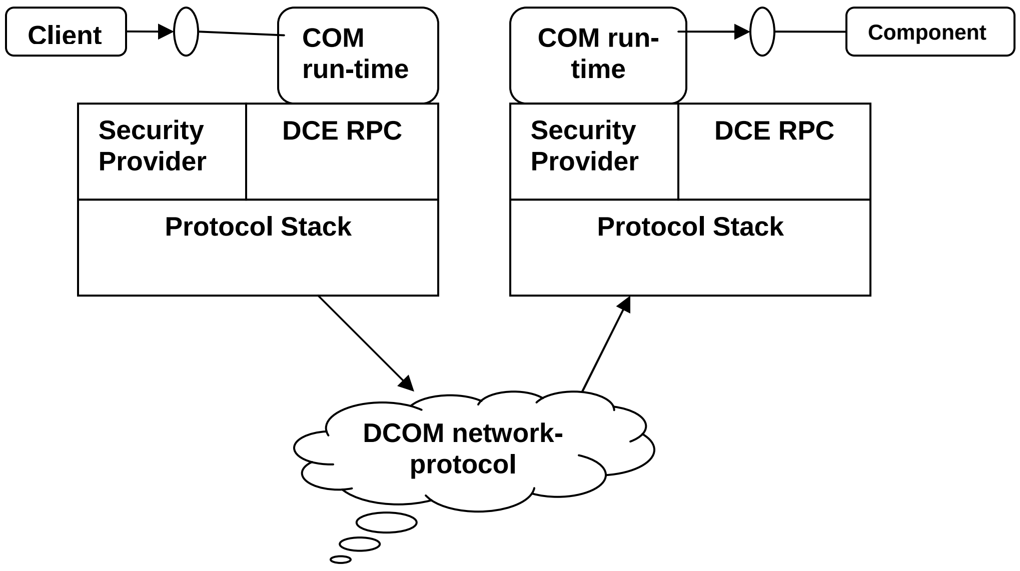

Figure 8 DCOM Architecture

A DCOM object has one or more interfaces that a client accesses via interface pointers. It is not possible to directly access an object itself; it is accessible only through its interfaces. Thus, a DCOM object is completely defined by the interfaces that comprise it. Each DCOM interface is unique in the system. A Globally Unique Identifier (GUID – a 128 bit integer that guarantees uniqueness in space and time for an interface, an object or a class) allows them to be uniquely named. A DCOM interface is not modifiable; if a new function is added or if the semantics of an existing function changes, a new interface is added and a new GUID is assigned to it.

DCOM, like CORBA, provides dynamic invocation and metadata facilities. The DCOM type library, like the CORBA Interface Repository, allows clients to dynamically discover the methods and properties a DCOM server object exposes.

All DCOM components and interfaces must inherit from IUnknown, the base DCOM interface. IUnknown consists of the methods AddRef(), Release() and QueryInterface(). AddRef() and Release() are used to for reference counting and memory management. Essentially, when an object's reference count becomes zero, it must self-destruct.

2.3 Service-Oriented Architecture

Several definitions exist for what constitutes an SOA. Some take a technical perspective, some others a business perspective and a few define SOA from an architectural perspective. For example, the W3C (World Wide Web Consortium) takes a technical perspective and defines SOA as “A set of components which can be invoked, and whose interface descriptions can be published and discovered” [W3C 2004]. This is not very clear as it describes architecture as a technical implementation and not in the sense the term “architecture” is generally used – to describe a style or set of practices.

A more helpful definition of SOA from an architectural perspective is provided in [MSDN 2004 g] where SOA is defined as “an architecture for a system or application that is built using a set of services”. An SOA defines application functionality as a set of shared, reusable services. However, it is not just a system that is built as a set of services. An application or a system built using SOA could still contain code that implements functionality specific to that application. On the other hand, all of the application’s functionality could be made up of services.
Some of the other definitions of SOA include:

- “Service-Oriented Architecture is an approach to organizing information technology in which data, logic, and infrastructure resources are accessed by routing messages between network interfaces.” [Microsoft 2006b]

- “A service-oriented architecture (SOA) is an application framework that takes everyday business applications and breaks them down into individual business functions and processes, called services. An SOA lets you build, deploy and integrate these services independent of applications and the computing platforms on which they run.” [IBM 2006]

The four tenets of SOA define desirable characteristics of a service [Microsoft 2004a]:

- o Service boundaries are explicit.
- o Services are autonomous.
- o Services share schema and contract not types.
- o Service compatibility is based on policy.

The most fundamental form of SOA consists of three components – a Service Consumer, a Service and a Service Directory as shown in Figure 9. These three components interact with each other to provide/achieve automation.

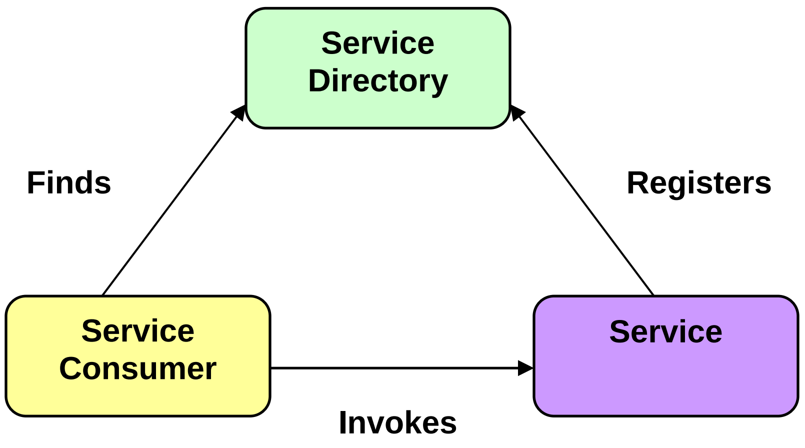

Figure 9 SOA

2.3.1 Service

In a SOA, services are the building blocks from which an application or system is assembled. A service can be defined as "an implementation of a well-defined piece of business functionality, with a published interface that is discoverable and can be used by service consumers when building different applications and business processes" [O'Brian, Bass, and Merson 2005]. For example, there could be a service that performs the following task: "verify a consumer's credit history." The technology used to provide the service, such as a programming language, does not form part of the definition of a service. A service should confirm to the following service-level design principles.

2.3.2 Characteristics of a Service

- Discoverability

A service can be found at both design time and runtime, not only by unique identity but also by interface identity and by kind of service. Service discovery can be facilitated by the use of a directory provider, or, if the address of the service is known during implementation, the address can be hard-coded into the user's software during implementation.

- Interface based definition

Services implement separately defined interfaces. The benefit of this is that multiple services can implement a common interface and a service can implement multiple interfaces. An application or a system whose underlying structure is based on SOA is designed as a collection of discrete services that are wired together using the description of their interfaces.

- Composability

Services may compose other services. This possibility allows logic to be represented at different levels of granularity and promotes reusability and the creation of abstraction layers.

- Reusability

Application logic encapsulated as a service can be reused in different applications. The Service Oriented paradigm encourages reuse in bigger chunks at higher levels of abstraction than other architectural approaches. The software components become very reusable because the interface is defined in a standards-compliant manner. So, for example, a C# service could be used by a Java application.

- Single instance nature

Each service is a single, always running instance with which a number of clients communicate.

- Autonomous nature

Services have distinct boundaries. They should be self-contained and should not depend on the state of other functions or processes.

- Asynchronous nature

In general, services use an asynchronous message-passing approach; however, this is not required. In fact, many services use synchronous message passing at times.

- Stateless

Services should not be required to manage state information, since that can impede their ability to remain loosely coupled. Services should be designed to maximize statelessness even if that means deferring state management elsewhere.

- Granularity

Compared to CSA and DOA, operations on services are frequently implemented to encompass more functionality and operate on larger data sets.

- Loosely coupled nature

SOA is a loosely coupled architecture because it strictly separates the interface from the implementation. Services share schema and contract, not class. Services interact solely on their expression of structures using schema, and behaviors using contract. Further, runtime discovery reduces the dependency between service producers and consumers and makes an SOA even more loosely coupled.

- Services are location transparent.

Service requestors do not have to access a service using its absolute network address. Requestors dynamically discover the location of a service looking up a registry. This feature allows services to move from one location to another without affecting the requestors.

- Abstract underlying logic

The interface definition encapsulates (hides) the vendor and language-specific implementation.

The characteristics of services listed above confer certain defining characteristics and capabilities on SOA. Important among them are the capability for Legacy System Integration, service-based interoperability and integration and loose coupling between elements of the architecture leading to greater flexibility.

A Service-Oriented system is a system based on SOA principles. Web Services are one way of realizing a SOA. In fact, it is possible to build an SOA without Web Services as a SOA can be realized using a host of different technologies other than Web Services. Web Services are a Service-based technology that is currently gaining industry acceptance as a standard to build SOAs. However, Web services are not inherently service oriented. A Web service merely exposes a capability that conforms to Web services protocols. A SOA is the structure that makes service orientation possible.

2.3.3 Web Services

"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards" [W3C 2004].

Web Services consist of Services. They are a way of encapsulating/exposing business logic/application. Unlike the .NET or Java EE frameworks, an entire system of business logic cannot be built using Web Services. A service can be implemented without distributed objects behind [or with it]. It can be anything behind it, including procedural code. Or it could be an entire enterprise application.

Web Services are "transport agnostic" meaning that they can be accessed over any type of transport or application protocol. We can use SOAP to transport messages over HTTP, but we can also use to transfer messages over UDP or TCP. However, most Web Services run over HTTP.

2.3.3.1 Web Service Standards

Interoperability using Web Services is made possible due to industry collaboration, standards organizations and consensus from major software rivals on the Web Services standards. Currently, there are over 60 web services standards and specifications, and that figure is continuing to rise. Figure 10 provides an overview of the major standards and specifications. There are three bodies that are mainly concerned with WS-standards.

The W3C [W3C 2006] is responsible for HTTP, SOAP and XML.

OASIS [OASIS 2006] (the Advancement of Structured Information Standards) maintains WS-Security, UDDI, and WS-Reliability for reliable messaging using SOAP.

WS-I [WS-I 2006] (Web Services Interoperability Organization) provides Profiles, Sample Applications and Testing Tools. Profiles include implementation guidelines for how related web services specifications should be used together for best interoperability. To date, WS-I has finalized the Basic Profile, Attachments Profile and Simple SOAP Binding Profile. Work on a Basic Security Profile is underway too.

The WS-I Basic Profile is a document that clarifies the SOAP 1.1 and WSDL 1.1 specifications in order to promote SOAP interoperability [WS-I 2004].

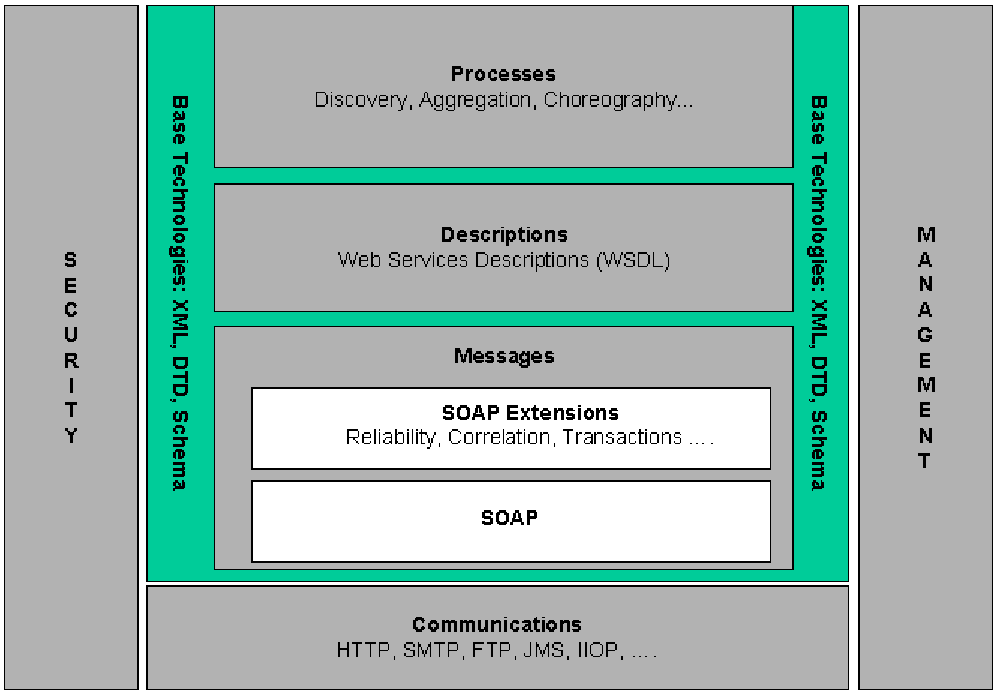
SECURITY
Base Technologies: XML, DTD, Schema
Processes
Discovery, Aggregation, Choreography...
Descriptions
Web Services Descriptions (WSDL)
Messages
SOAP Extensions
Reliability, Correlation, Transactions ....
SOAP
Base Technologies: XML, DTD, Schema
MANAGEMENT
Communications
HTTP, SMTP, FTP, JMS, IIOP, ...

Figure 10 Web Services protocols stack (Image taken from [W3C 2004])

2.3.3.1.1 SOAP

SOAP is an XML-based protocol used for exchanging structured and typed information between Web Services. A SOAP message is formally specified as an XML Infoset which provides an abstract description of its contents. SOAP is fundamentally a stateless, one-way message exchange paradigm [W3C 2003a]. However, applications can create more complex interaction patterns such as request/response or request/multiple responses by combining such one-way exchanges with features provided by an underlying protocol and/or application-specific information. [W3C 2003a] The name "SOAP" was originally an acronym for Simple Object Access Protocol, but the full name was dropped in Version 1.2 of the SOAP specification

2.3.3.1.2 WSDL

Web Services Description Language (WSDL) [W3C 2006a] is a document written in XML that describes a Web service. It specifies the location, the basic format of the request and response messages and the protocol bindings for communicating with the service it describes.

2.3.3.1.3 UDDI

UDDI (Universal Description, Discovery, and Integration) [UDDI 2004] provides a standardized method for publishing and discovering information about Web Services. A UDDI registry is itself an instance of a Web Service. Information in UDDI is conceptually organized into white pages, yellow pages and green pages with white pages providing information about business entities and green pages providing technical information about the registered Web Services. Information in a UDDI registry is modeled using five data structures: businessEntity, businessService, bindingTemplate, tModel and publisherAssertion. An important aspect of UDDI is that each entity, be it a businessService or a tModel, is assigned a unique key or identifier when it is first published to a registry. Within each registry, this key must be unique. Information in the different data structures is related using these keys.

There are two primary types of UDDI registries: private and public. UDDI was originally conceived as a universal public registry called the “UDDI Business Registry” (UBR) where all Web Services would be registered. As this proved to be quite infeasible, Version 3.0.2 (V3) of UDDI places more emphasis on private registries and tries to address the interaction between public and private registries by introducing the concept of root and affiliate registries. Prior to V3, UDDI registry implementations had no relationship to one another. Identical entities having the exact same keys could not be saved into multiple registries. In V3 of UDDI, a root registry acts as the authority for key spaces. Such registries are used to delegate key partitions so that other registries can rely upon the root registry to verify and maintain the uniqueness of such key partitions. The UBR is a good example of a root registry.

2.3.3.1.4 Orchestration and Choreography

2.3.3.1.4.1 Web Services Orchestration

Orchestration refers to an executable business process in which interactions with both internal and external Web services occur at the message level.

2.3.3.1.4.2 Web Services Choreography

A choreography is a model of the sequence of operations, states, and conditions that control the interactions involved in the participating services. The interaction prescribed by a choreography results in the completion of some useful function. Examples include the placement of an order, information about its delivery and eventual payment, or putting the system into a well-defined error state.

2.3.3.1.4.3 Difference Between Orchestration and Choreography

Orchestration always represents control from one party's perspective. This distinguishes it from choreography, which is more collaborative and allows each involved party to describe its part in the interaction and task execution order, and they can span applications and organizations to define a long-lived, transactional, multistep process model [Chung, Lin, and Mathieu 2003].

2.3.4 Jini

The Jini framework is a distributed infrastructure built around the Java programming language and environment. It specifies a way for clients and services to find each other on the network and to work together to get a task accomplished [Sun 2004a]. The two most important components of the Jini framework are the discovery protocol and the Lookup service [Waldo 1999].

A Jini federation is a collection of clients and services communicating with each other using Jini protocols [Ledru 2002]. Entities wishing to join a Jini federation use the discovery protocol to find a Lookup service. The Lookup service acts as a kind of directory service where service providers publish service descriptions and service objects (proxies). Clients query a Lookup service to find services that match their requirements. Lookup services send service objects to interested clients, which then invoke methods on them. The basic communication model of Jini is based on the Java Remote Method Invocation system, in which objects in one Java virtual machine communicate with objects in another by receiving a proxy object that implements the same interface as the remote object. This proxy object deals with all communication details between the two processes.

A client discovers a Lookup service using either a multicast protocol or a unicast protocol. The Jini network does not require that there be a single lookup service. Many lookup services can co-exist on the same network, and a service can register with multiple lookup services depending upon the local policy of the service.

In Jini, when a service registers itself with a lookup service, it has to provide a lease specifying how long it wishes to stay registered. Once registered, it is the service's responsibility to periodically renew its lease if it wants to stay registered. If a service does not renew its lease either because it crashed or for some other reason, its reference will be eventually removed from the lookup service.

The Jini model is java-centric i.e. all participants (service requestors, as well as service providers) need to have Java Virtual Machine capability.

2.4 Peer-to-Peer Architecture

PPA is a self-organizing and decentralized architecture of potentially untrusted, unreliable nodes with symmetric roles for purposes of sharing resources of the participating peers. In PPA, each peer can play the role of both a client and a server, unlike CSA where roles are asymmetric – i.e. a entity can be either a client or a server, but not both.

Each peer that joins the network has to register itself and the provided resources. By joining the network, a new peer automatically registers itself to the network, either by signing up at a central entity or by announcing its presence to the network.

There are mainly two different flavors of PPA, hybrid and pure. In hybrid PPA, a central entity, called a super node or a rendezvous point provides a registry and helps in the discovery process. On the other hand, in pure PPA networks, this is done by active announcement to the network. Active announcement uses various broadcast protocols and algorithms. Thus, PPA networks rely on the existence of a good discovery mechanism.

2.4.1 JXTA

Project JXTA (Juxtapose) [JXTA 2006a] is an attempt to formulate a core set of P2P protocols on top of which P2P applications can be built [Halepovic and Deters 2002]. It started as a research project at Sun Microsystems and was later converted into an open source project. JXTA specifies a set of protocols rather than an API. Thus, JXTA technology can be implemented in any language on any Operating System [JXTA 2006b].

The components that make up a JXTA system are the very same that can be identified in many P2P network implementations:

- Peers and peer groups

- Services

JXTA services are available for shared use by peers within a peer group. In fact, a peer may join a group primarily to use the services available within that group. A set of services, called core services, is essential to the basic operation of a JXTA network. The core services included in the JXTA specification include:

- Pipes
  In the JXTA specification, logical pipes are the mechanisms used to transfer data, files,

information, code, or multimedia content between peers. JXTA pipes are used to send messages (with arbitrary content) between peers.

- Messages
  JXTA messages are XML documents that are passed from one peer to another through pipes. A JXTA message consists of
  - A header
  - Source endpoint information (in URI form)
  - Destination endpoint information (in URI form)
  - A message digest (optional -- for security purposes)

- Advertisements
  JXTA advertisements are also XML documents. The content of an advertisement describes the properties of a JXTA component instance, such as a peer, a peer group, a pipe, or a service. For example, a peer having access to an advertisement of another peer can try to connect directly to that other peer.

- Membership
  Membership determines which peers belong to a peer group; handles arrival and departure of peers within a peer group.

- Access
  Access can be considered as a security service for controlling access to services and resources within a peer group; a sort of security manager for the peergroup

- Discovery
  Discovery is a way peers can discover each other, the existence of other peer groups, pipes, services, and the like

- Resolver
  Resolver allows peers to refer to each other, peer groups, pipes, or services indirectly through a reference (called an advertisement in JXTA lingo); the resolver binds the reference to an implementation at run time

The JXTA protocols define the minimum required network semantic for peers to establish a virtual ad hoc network as shown in Figure 11 (called an “overlay network” in P2P parlance) on top of the Internet and non-IP networks, allowing them to directly interact and self-organize independently of their network connectivity. This can be used to build a wide variety of P2P networks. The JXTA addressing model is based on a uniform and location independent logical addressing model. Every network resource (peer, pipe, data, peergroup, etc.) is assigned a unique JXTA ID. JXTA IDs are abstract objects enabling multiple ID representations (IPv6, MAC) to coexist within the same JXTA network.

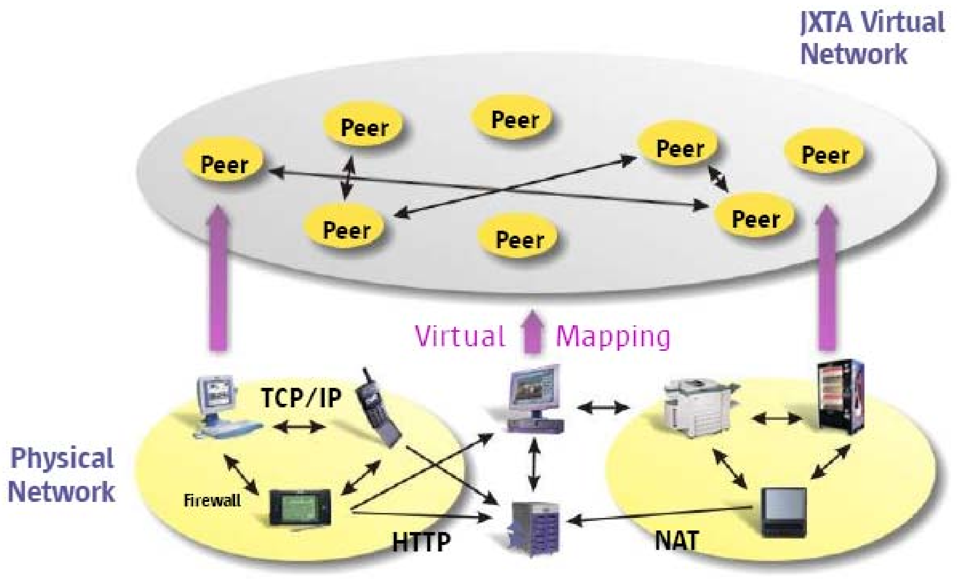

Figure 11 JXTA (Image taken from [Sun 2004])

Figure 12 shows the Project JXTA protocols.

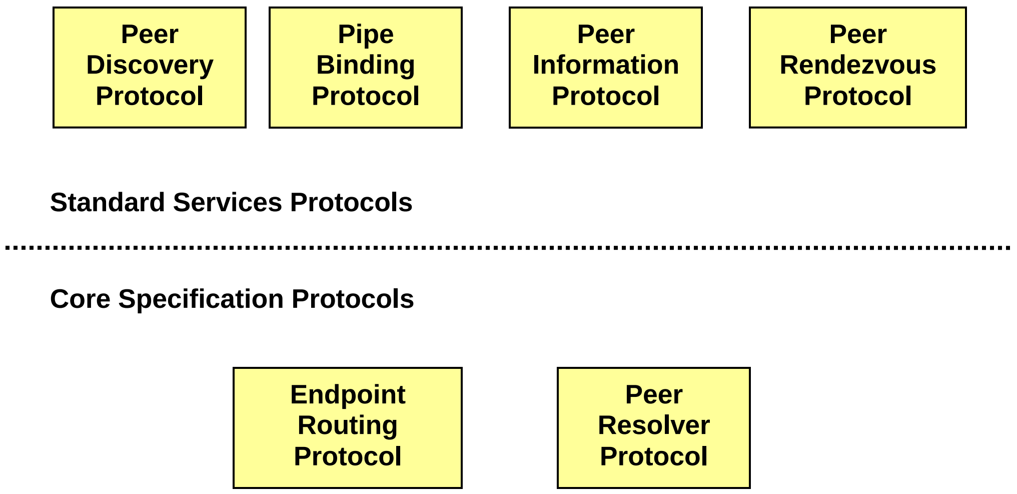
Peer Discovery Protocol
Pipe Binding Protocol
Peer Information Protocol
Peer Rendezvous Protocol
Standard Services Protocols
Core Specification Protocols
Endpoint Routing Protocol
Peer Resolver Protocol

Figure 12 Project JXTA protocols

The Project JXTA protocols are composed of six protocols divided into two categories.

2.4.1.1 Core Specification Protocols

The JXTA Core Specification protocols define the functionality required of all implementations that wish to be JXTA compliant.

The Core Specification defines two protocols:

• The Endpoint Routing Protocol (ERP)

ERP is the protocol by which a peer can discover a route (sequence of hops) used to send a message to another peer. If a peer A wants to send a message to peer C, and there is no direct route between A and C, then peer A needs to find the intermediary peer(s) to route the message to C. ERP is used to manage and determine the routing information. If the network topology has changed such that the route to C can no longer be used, the peer can use ERP to find routes known by other peers to construct a new route to C.

• The Peer Resolver Protocol (PRP)

The PRP is used by a JXTA peer to send a query to another JXTA peer and receive a response.

2.4.1.2 Standard Service Protocols

The JXTA Standard Services protocols are optional JXTA protocols and behaviors. Implementing these services will provide greater interoperability with other implementations and broader functionality.

The Standard Services protocols specification defines four protocols:

- The Rendezvous Protocol (RVP)

The RVP is used for propagating a message within a peer group.

- The Peer Discovery Protocol (PDP)

PDP is the protocol by which a peer publishes its own advertisements, and discovers advertisements from other peers (peer, peergroup, module, pipe and content). PDP uses the Peer Resolver Protocol for sending and propagating discovery advertisement requests.

- The Peer Information Protocol (PIP)

PIP is the protocol by which a peer may obtain status information about other peers, such as state, uptime, traffic load, and capabilities. PIP uses the PRP for sending and propagating peer information requests.

• The Pipe Binding Protocol (PBP)

PBP is the protocol by which a peer can establish a virtual communication channel or pipe between one or more peers. PBP uses the PRP for sending and propagating pipe binding requests. [Traversat et al. 2003]

Chapter 3: Characteristics of Network-Centric Architectures

Modern software systems are complex entities. Their properties, characteristics and capabilities are determined by the various structures of which they are composed. To understand the structure of a software system better, it can be viewed as a succession of stages, where each stage adds a little more detail (and properties and capabilities) to the previous stage as depicted in Figure 13. A network-centric system can be visualized as starting with a style and generic architecture which is extended to create a complete application architecture for an application domain. This is further extended by design and implemented to form the complete system or application. This complete system or application runs over a network infrastructure consisting of a network software layer over the network hardware. Thus, the overall characteristics of a system are a function of the underlying network hardware, the layers of the networking software used, the generic architecture, the completed application architecture and application framework (if any) and then the design and implementation of the application itself.

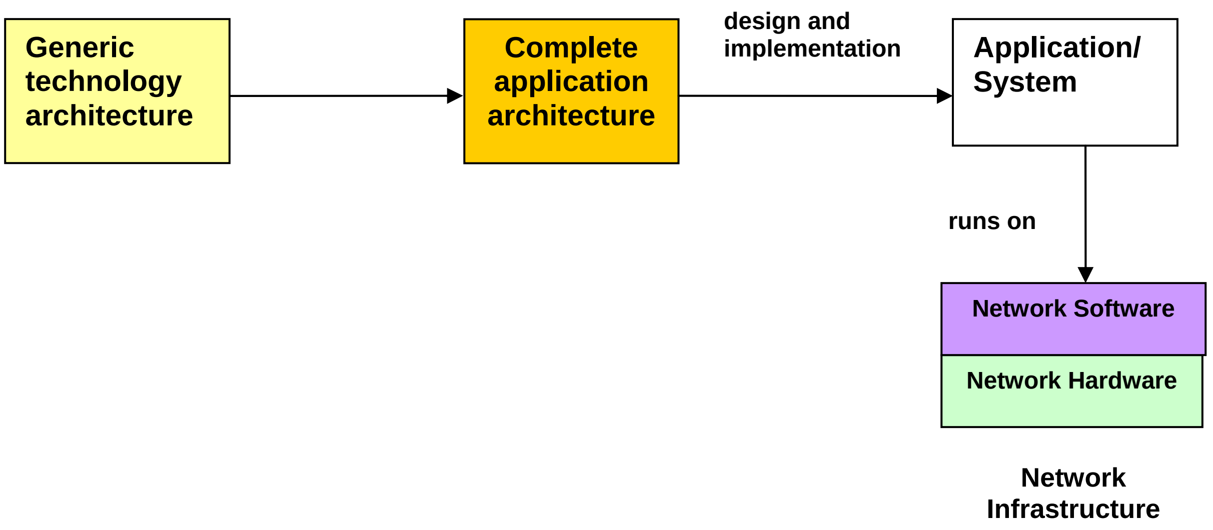
Generic technology architecture
Complete application architecture
design and implementation
Application/ System
runs on
Network Software
Network Hardware
Network Infrastructure

Figure 13 Structures in a system

A survey of existing literature on network-centric systems and architectures provides us with a list of several required and desired characteristics and capabilities in a network-centric system and infrastructure. As stated earlier, the qualities/capabilities exhibited by a system are a function of its various aspects – the operational environment (the network infrastructure), its architecture, design and implementation. These aspects influence the desired qualities and capabilities of a network-centric system to various degrees. Certain attributes such as reach, quality, network assurance and network agility are qualities of the underlying network – they are dominated by the characteristics and capabilities of the underlying hardware and software network infrastructure. In the same vein, certain attributes are the responsibility of the architecture and some of the completed application/completed architecture. However, a large section of the quality characteristics/capabilities are influenced/determined by all aspects of the system.

From the definition and descriptions of the various classes of network-centric systems provided in the previous chapter, it is evident that network-centric systems have the following distinguishing characteristics:

- Components of a network-centric system operate with each other over a network such as Internet, LAN (Local Area Network), WAN (Wide Area Network), VPN (Virtual Private Network) or wireless network.
- The constituent nodes of a network-centric system can run on heterogeneous platforms
- The overall system can cross organizational boundaries
- A network-centric system could often be "dynamic coalition of nodes" – i.e. runtime dynamism.

These characteristics of network-centric systems lead to the following quality requirements and capability requirements for building such systems.

- Openness (enabling interoperability)
    - Support for open standards

- Interoperability
    - Data Elements Interoperability
    - Communications interoperability
    - Interoperability of new and legacy systems
    - Interoperability with the GIG (Global Information Grid) for military systems
    - Plug-and-play of new components

- Integration
    - With disparate and interoperable systems
    - With legacy systems

- Adaptability
    - Modifiability and configurability
    - Reconfigurability of its structure, components - runtime dynamism.

- Dependability
    - High availability
    - Fault tolerance/Survivability/Resilience
    - Security

- Scalability and performance

3.1 Qualities

3.1.1 Openness, Interoperability and Integration

Openness, Interoperability and Integration are terms that are frequently associated with network-centric systems. An understanding of them with respect to each other will help us to understand an important aspect of architecting network-centric systems.

To begin with, let us consider Integration. Integration is a process. SEI [2006d] defines software integration as "the practice of combining individual software components into an integrated whole." Thus, the creation of network-centric systems often involves integration. Dynamic composition, a characteristic of some network-centric systems can be considered as an act of integration at runtime.

Interoperability is defined as "the ability of two or more systems or components to exchange information and to use the information that has been exchanged". [IEEE 1990] Interoperability is required to realize the intended benefit of integration – combining capabilities to derive new capabilities. Thus, interoperability is required to create systems that are composed of nodes running on heterogeneous platforms. It is required irrespective of whether the system is composed statically or it is a dynamic, runtime coalition of nodes.

The use of the terms "Interoperability" and "Integration" depends on the perspective – a system composed of other entities that work together is said to the integrated when viewed from a distance whereas from a closer view the entities are considered to be "interoperating". The term "Interoperability" does not imply that the "interoperating systems or entities" are integrated. It merely expresses that the potential to integrate exists. Thus, proper interoperability is a prerequisite for successful integration. Depending on the level and quality of interoperability existing between nodes, various integration strategies can be adopted.

To arrive at a description of "openness" at the architectural level, it helps to begin with a few popular definitions of an "open system" by various sources. The SEI defines an "open system" as following [SEI 2006e]:

"An open system is a collection of interacting software, hardware, and human components

- designed to satisfy stated needs
- with interface specifications of its components that are
  - fully defined
  - available to the public
  - maintained according to group consensus
- in which the implementations of the components conform to the interface specifications"

Another popular definition of an "Open System" is provided by the IEEE POSIX (Portable Operating System Interface) working group and has been adopted by the DoD's Open Systems Joint Task Force (OSJTF) . POSIX defines an open system as:

“A system that implements sufficient open specifications for interfaces, services, and supporting formats to enable properly engineered components to be utilized across a wide range of systems with minimal changes, to interoperate with other components on local and remote systems, and to interact with users in a style that facilitates portability”.[ACC 2006]

The SEI lists the following characteristics of an open system [SEI 2006f]:

- “well defined, widely used, and non-proprietary interfaces/protocols
- use of standards which are developed/adopted by industrially recognized standards bodies
- definition of all aspects of system interfaces to facilitate new or additional systems capabilities for a wide range of applications
- explicit provision for expansion or upgrading through the incorporation of additional or higher performance elements with minimal impact on the system”

As with the definition of open system, there are various definitions of the concept of open system architecture. A simple one comes from the OSJTF [OSJTF 2006]:

“A system architecture produced by an open systems approach and employing open systems specifications and standards to an appropriate level.”

One definition that is consistent with the more operational definition of open system given above is [SEI 2006f]:

”An open system architecture is a representation of a system in which there is

- a mapping of functionality onto hardware and software components
- a mapping of the software architecture onto the hardware architecture
- a representation of the human interaction with these components
- interface specifications of the components that are
    - fully defined available
    - available to the public
    - maintained according to a consensus process”

In evaluating generic architectures and frameworks in terms of openness, the challenge is to formulate a set of appropriate questions that address those aspects of openness that the architectures should be responsible for at that level.

In terms of implemented systems, the term “Interface” refers to the interfaces of the fully developed and implemented components or services that are the units of reuse. In Figure 13, we brought out the notion of a less complete and abstract “generic architecture” that does not include the domain specific components. The generic architecture prescribes the nature of the components but not the components itself. Thus, at the generic architectural level, it makes more sense to ask whether the nature of the components require them to be described by an Interface; does the architecture mandate or recommend the separation of component Interface specification and implementation?

Openness is often a function of description. It is more influenced by design, documentation and policy than architecture. For software systems, description can be of two types: external documentation and intrinsic or self-description.

External documentation could consist of documents describing the module or component in a natural language like English, or using formal methods like ADLs (Architectural Description Languages) or process algebra that have the potential to be more precise. DODAF (DoD Architecture Framework) [DODAF 2004] is a major effort by the DoD for comprehensively documenting system and enterprise architecture. The framework, partitioned into two volumes and a deskbook offers extensive guidance for documenting architectures from four operational views: overarching All View (AV), Operational View (OV), Systems View (SV), and the Technical Standards View (TV).

One of the dangers of using external documentation is its distance from the actual implementation. It could easily be out of sync with the actual architecture or implementation. For an architecture that relies on external documentation, openness becomes more of a policy for that particular project than an intrinsic property of the overall architecture inherited from the generic architecture.

A mandatory requirement by a generic architecture that a component's interface description/specification be separate from its implementation can be called as "Self-Description". Self-description enforced by the architectural requirement or constraints of a particular architecture makes "openness" an intrinsic property of the architecture. Further, interfaces always evolve together with their implementations. Therefore, they have the added advantage of being more precise and always in sync.

3.1.2 Adaptability

Adaptability can be defined as "the ease with which software satisfies differing system constraints and user needs." [Evans 87] Differing system constraints and user needs can be satisfied by changes to the overall system architecture and corresponding implementation. For a certain class of Network-centric systems, these changes may be needed to be applied online while for others it may suffice to incorporate them offline statically. Modifiability and reconfigurability are two aspects of adaptability.

3.1.2.1 Modifiability

Modifiability is concerned with how the system can accommodate anticipated and unanticipated changes and is largely a measure of how changes can be made locally, without ripple effect on the system at large [Barbacci et al. 2000]. Modifiability is static in the sense that changes are accommodated during development or maintenance. Modifiability scenarios in a technical sense include changes such as addition of components, deletion of components, change in the interface of a component and change in the interface semantics of a component. These changes are required due to various scenarios such as new requirements, changed requirements, bug fixes, and external changes like the change in the technology used. Configurability is a subset of modifiability to adapt the system to a certain environment.

Modifiability is measured in terms of the flexibility of the architecture to change. [Barbacci et al. 2000] identify the following architectural mechanisms that can help a system attain modifiability:

- Location transparency of objects and services including “yellow pages” facility
- Modularity of components in the system.
- Information hiding and abstraction which promote modularity
- Mechanisms to achieve information hiding and abstraction such as layering, virtual machine and using interfaces.

Versioning mechanisms are used to distinguish evolving software artifacts over time. Versioning helps in component upgrades. In general, metadata that specifies different aspects of software components is needed in case of dynamic linkage, where the only information about component usage is the component itself.

3.1.2.2 Reconfigurability

Reconfigurability, also referred to as “Runtime dynamism” is the ability of a system to change its structure and architecture during runtime to accommodate change in requirements and failures. Reconfigurability faces almost similar forces as modifiability except that it is during runtime. The effect in technical terms is the similar to the ones considered for modifiability: addition of an interface or component, deletion and modification. The causes may include node failure, link failure or resource change.

Reflection is a very powerful tool to support dynamicity of software architectures and component configurations. Incorporating an explicitly reflective framework at the architectural level helps to build adaptive systems. A considerable portion of the research on dynamic systems is aimed at exploiting the architectural structure in systems. Architectural approaches using reflection include constructing an architectural model and mapping them to implementation.

The term “reflection” indicates that the system can be viewed as being composed of two levels – a base level and at least one meta level. A reflective system has an internal model of itself – this model can be referred to as the meta level. The actual system (the implementation) can be considered as the base level. The base level and the meta level have a causal connection i.e. any change in the base level is reflected in the meta level. Similarly, any change in the meta level causes corresponding changes in the base level.

Three kinds of runtime reflective capabilities can be identified [Ortin et al. 2005]:

- Introspection: The system’s structure can be consulted, but not modified
- Structural Reflection: The system’s structure can be modified and the changes are reflected during runtime. An example would be to add or delete a member variable to a class and objects.
- Computational (Behavioral) Reflection: The system’s semantics can be modified, changing the runtime behavior of the system. This kind of the reflection is the most powerful as it allows changing the architecture of a running system while still maintaining consistency. Thus, this

concept of computational reflection can be used at the architectural level to define dynamic systems.

At the architectural level, runtime change involves addition and deletion of nodes. It is easier to accommodate addition and deletion of nodes if there is dynamic discovery. Further, the use of certain styles like publish/subscribe (implicit invocation) are more amenable to this kind of dynamism as these styles consider components as almost independent nodes. Architectural mechanisms needed to support such styles include a robust event system.

3.1.3 Dependability

Dependability, the degree to which a system can be relied on, is a composite attribute. The attributes of dependability are availability, reliability, safety and security [Laprie et al. 2000]. Dynamic reconfigurability can be considered as an important dependability mechanism [Shrivastava and Wheater 1998].

Dependability for network-centric systems can be viewed at two levels of abstraction: at the System-of-Systems (SoS) level and the application level. Architectural mechanisms for achieving dependability have to be considered at these levels accordingly for the various architectures and frameworks.

In network-centric systems, failures can occur mainly due to three reasons: process failures, node failures and link failures. Node failures are applicable to systems that are formed as static or dynamic coalitions of nodes. Thus node failures are the dominant failure scenarios for SoS and systems (if a system, in turn, is a composition of services) and so on down the hierarchy until an application that is not composed of discrete services is reached. For an application, process failure is the main failure scenario. Link failures apply both to SoS and applications as a link failure can be interpreted as a node or process failure.

When we talk of Client-Server Architecture (CSA), Distributed Objects Architecture (DOA) or Component Based Architecture (CBA), the focus is on the failure of software entities in individual tiers or the loss of connectivity between entire tiers. The presentation may have to deal with the failure of the business tier, the business tier with the loss of the connection to the database or to the persistence tier. Some of the architectural mechanisms for achieving dependability for an application include:

- Redundancy of software components or services using replication [Laprie et al. 2000, Nikander 2000]
- Transactions. [Tartanoglu et. al 2003] Transactions are a mechanism for maintaining database consistency.
- fail over clustering

Architectural approaches to fault tolerance and resilience at the SoS and system level include:

- Dynamic discovery and composition [Nikandar 2000]
- Lease based resource management [Tichy and Giese 2004, Gray and Cheriton 1989]

3.1.4 Scalability and Performance

3.1.4.1 Scalability

Scalability is an important quality characteristic for network-centric applications and systems as poor scalability can result in poor performance [Bondi 2000]. Two major types of scalability can be observed:

Structural Scalability: Bondi [2000] defines structural scalability as "the ability of a system to expand in a chosen domain without major modifications to its architecture". In a system and SoS context, a major aspect of structural scalability can be interpreted as the ability of the discovery mechanism to accommodate a large number of services, without significant degradation in performance.

Load scalability: Load scalability of an application refers to the ability to handle more workload, typically from the addition of more users. A platform is load scalable if an increase in hardware resources results in a corresponding similar increase in supported user load while maintaining the same response time.

Two types of load scalability can be observed:

- Scaling up or Vertical scalability: the scalability achieved by using faster hardware (single machines).
- Scaling out or Horizontal scalability: the scalability achieved by using more hardware (multiple machines). This has more significant implications for architectural design.

For standalone applications, load scalability is important. In terms of software, scalability is a function of the hardware, operating system and the system architecture. The scalability of the operating system plays a major role in the scalability of a software platform or framework that runs on it as the operating system determines factors such as the kind of processors can be used and how powerful the processors can be.

Some of the architectural and framework mechanisms that can help scalability are:

- Reduction in the amount of communication.
  This helps scalability by reducing network-traffic. This can be achieved by code migration. One form of code migration can be achieved by using value objects.

- Coupling.
  More coupling leads to less scalability as the amount of information exchanged is more.

- Statelessness.

Chapter 4: Comparative Assessment

In this chapter, the network-centric architectures and frameworks described and characterized in the previous chapters are compared based on two sets of criteria: architectural characteristics and quality characteristics. Architectural characteristics are aspects of the architectures themselves that help to distinguish architectures from one another. The quality characteristics chosen here are the ones that were described in chapter 3. Quality characteristics are often influenced by the architectural characteristics. Further, not all architectural and quality characteristics apply equally to all architectures and frameworks. Even when they do apply, the sense in which they apply may differ from one architecture or framework to another. This is indicated whenever appropriate in the following sections.

4.1 Comparison between Architectures and Frameworks based on Architectural Characteristics

In the previous chapter, four network-centric architectures, namely, Client-Server Architecture (CSA), Distributed Objects Architecture (DOA), Service-Oriented Architecture (SOA) and Peer-to-Peer Architecture (PPA) and their associated frameworks are described. Any system that involves a request-response, a piece of software sending out a request over the network to another piece of software and receiving a response in return can technically be termed as having a CSA. Thus, all these architectures have an element of client-server interaction in them. The difference between these architectures lays in architectural design that concern

- how application logic is partitioned
- where the partitioned units of processing logic reside
- how the units of processing logic interact

4.1.1 Partitioning of Application Logic

A network-based system can be viewed as a collection of nodes interacting over a network where the nodes play various roles. The difference between the CSA, DOA, SOA and PPA primarily arises in the roles played by the application level software on these nodes and the nature of the application software on the nodes. To begin with, let us consider CSA and DOA. The most popular style for these architectures is the 3 tier architecture where processing is divided into three distinct layers – presentation, business logic and database/data store.

The heart of an application is the business logic that consists of application logic described by an interface as illustrated in Figure 14. (Henceforth, the terms “application logic” and “business logic” will be used interchangeably). The presentation accesses and incorporates the application logic using the interface.

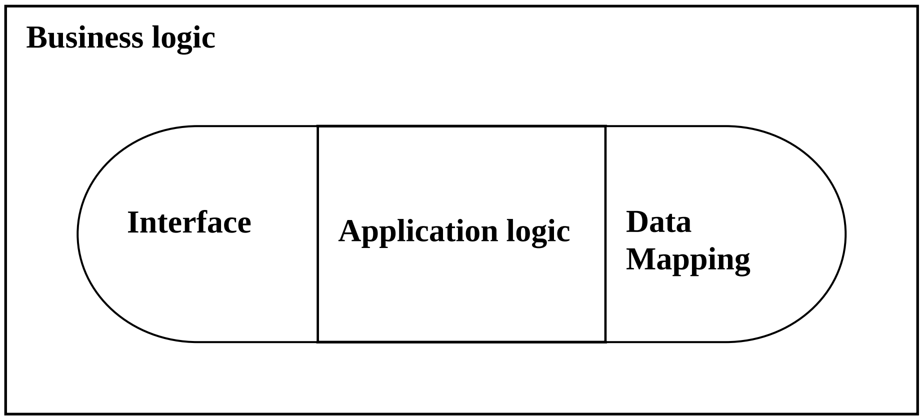

Figure 14 Business logic

This partitioning of an application into layers can be viewed as a horizontal partitioning of the logic. The different layers can be put on different machines. A layer on a different machine constitutes a tier. (A layer can be considered as a logical grouping and separation of functionality, while a "tier" can be considered as physical separation.) However, one layer cannot be split onto different machines. Thus, the number of tiers cannot be more than the number of layers. It is equal or smaller than the number of layers.

Layers are a horizontal partitioning of logic. Seen from this perspective, classic CSA can be seen as consisting of a monolithic business layer. While, this entire business layer can be deployed onto a different server, but it cannot be further divided into pieces and deployed onto different machines. This business layer can be implemented in many ways. Using objects is one possible and popular way of doing it. Other approaches include using procedural programming or scripts. In fact, when the entire server side part of the application resides on the same machine, if the application is not properly layered, there is often no strict separation of presentation from business logic.

However, the logic in the business tier or layer can be vertically partitioned as illustrated in Figure 15. In a CSA system with a monolithic business tier, vertical partitioning of the business layer leads to either DOA or SOA. In SOA, you can cross organizational boundaries.

While DOA and SOA may look similar from a physical perspective, there is a major difference. When dividing logic into discrete pieces, SOA uses and keeps in mind the tenets and principles of service-orientation such as autonomy, statelessness and interface opacity. A subtle characteristic of SOA is that it does not say anything about layering an application. When Web Services are used as a façade in a 3-layered application, it is used at the business layer to encapsulate and expose its functionality. However, SOA does not say anything explicitly about the presentation layer. Nor is the link between the business and data layer a concern of this architecture. It assumes that it need not look beyond the business layer. Thus, SOA says nothing about splitting an application's logic into different horizontal tiers. The internal architecture/design of an application is irrelevant.

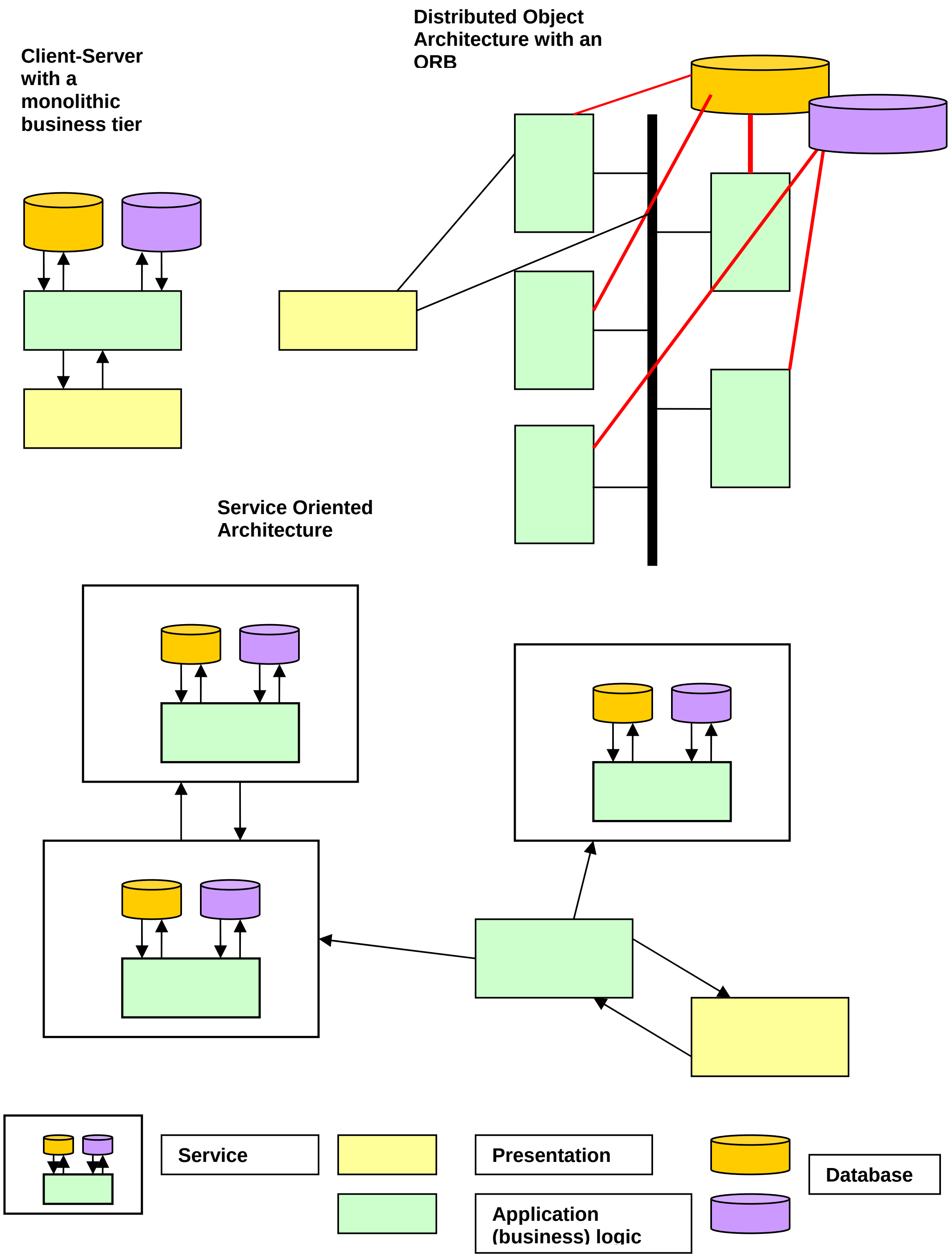
Client-Server with a monolithic business tier
Distributed Object Architecture with an ORB
Service Oriented Architecture
Service
Presentation
Database
Application (business) logic

Figure 15 Structure of applications architected in CSA, DOA and SOA

Theoretically, a service can encompass even the presentation, but this is rarely done in practice due to the drawbacks of depending on presentation data for integration. Encapsulating presentation means depending on the exact layout of presentation data such as HTML which can be very brittle [Trowbridge et al. 2004]. Any change in the position or layout of the fields can break the service encompassing it. Also, when an application is architected with strict separation of concerns, i.e. when there is no mixing of presentation and the business logic, the business logic is the real heart of the application and provides the required functionality. It is the most reusable part. A web service only needs to hook into this. In order to incorporate older applications in a SOA, a web services façade is thrown around the business logic. This becomes one of many other interfaces possible. Other interfaces can exist and the business logic can be used as before. However, this Web Façade around business logic is not SOA. Vendor hype is responsible for this impression. Doing so is using Web Services as a technology to achieve interoperability and integration, but it is not SOA.

One can view Service Orientation as separation of concerns. The opposite of SOA is duplication of functionality in every application that needs it. This means that the functionality is limited to being consumed by only one process/thread instead of being consumed by multiple processes/threads. Here the use of the word application again indicates that SOA is an enterprise level strategy and not within an application. This is because the benefits of SOA begin to look significant at that level. Within an application, SOA looks like Component-Based Architecture (CBA) with Web Services being one more technology to provide an interface façade.

Another major characteristic of SOA is that it can be created using an Enterprise Service Bus (ESB) as a communication fabric as shown in **Error! Reference source not found.**. An ESB can be considered as a "connectivity layer between services". [Schmidt et al. 2005] An ESB forms a layer of abstraction over the underlying communication layer and offers two major features over using just a conventional communication layer like the SOAP/HTTP over the Internet: mediation and metadata in a metadata registry that is accessible from any point on the bus. Mediation involves intercepting and acting on the messages passing between the various services over the bus. Mediation actions could include protocol translation, enforcement of policy (ex. for security), load balancing by delivery redirection or auditing. Thus, by providing mediation, an ESB plays a more active role than other conventional communication fabrics. The ESB may also provide a central registry for metadata such as service contracts and policies that can be made available on a global basis. This registry acts as a catalog for services and supports "assembly-from-parts".

While the CSA, DOA and SOA all allow us to break an application into chunks of functionality that can be deployed onto different machines, PPA on the other hand, does not have this concept. The easiest way to understand PPA and its differences with the other three is to consider the nature of the applications in built using PPA.

PPA applications are usually standalone applications and are not hosted from a web server. Entire applications are distributed onto different nodes of the network. Three major PPA application types can be observed:

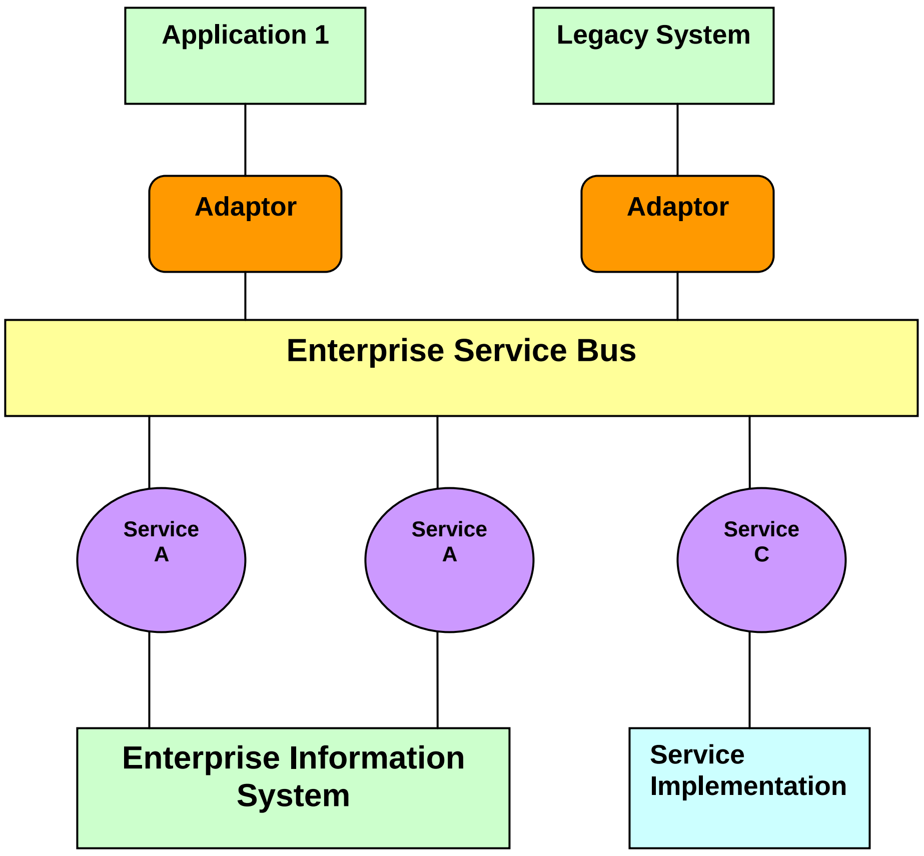

Figure 16 SOA Using Enterprise Service Bus

- Distributed computing/processing Applications: The same application working on different data sets i.e. same computation on different data units. An example of this class of application is the distributed computing project SETI@home [SETI 2006].

- Content distribution Applications: Same application providing storage for resources such as music files and other kinds of files on hard disks of peer like Gnutella [Gnutella 2001, Ripeanu 2001], BitTorrent [BitTorrent 2006] and Kazaa [Kazaa 2005].

- Collaborative Applications: same Application is used by different clients to form an ad hoc group and collaborate. Representative applications are chat clients, online meeting and collaborative editing applications [Groove Networks 2005].

Thus, PPA applications are usually monolithic and are often tightly bound to the underlying protocols. On the other hand, in SOA, different nodes provide different pieces of computation and an application brings them together. Thus, a major difference between the PPA and SOA is in the nature of the nodes and the distribution of computational logic onto them. In SOA, functionality is divided onto different nodes, i.e. it is not the same copy of the application everywhere on the different nodes.

4.1.2 Operational Environment and Scope of Distribution

In PPA, certain assumptions are made about the nature of the nodes hosting the applications and the network itself. PPA is also referred to as "computing on the edge". Peers are most likely to be PCs and workstations that connect to the network transiently rather than powerful servers with dedicated connections to the network that are usually used to host Web Services or other types of Services in a SOA. This means that peers can drop off at anytime i.e. there is a high "peer churn" rate. Because of this, there is a constant need to keep track of peers and their identities which is not required for Web Services as the assumption in a SOA is that of high availability. Web Services are hosted on well known hosts and ports and have fixed addresses.

At a lower level, the lack of availability also creates the need for a reliable messaging facility with an option for redelivery at a later time. Further, peers may alter their location on the network and may not always have the same IP address. This creates a need for the ability to resolve peer names to network addresses. Finally, while Web Services are hosted on external servers and addressable from outside the network, peers are often hidden behind NATs (network address translations) or firewalls and may require the use of a tunneling protocol or relaying services for communication Services.

4.1.3 Level of Abstraction

Complexity drives the level of abstraction. The traditional solution to addressing increased complexity has been to raise the level of abstraction as is evident in the evolution of programming languages – from machine code to assembly to the modern high level languages like C++, Java or C#. This is also the case with CSA and SOA. SOA evolved in response to a set of challenges and requirements that were difficult for the CSA and DOA/CBA to handle as they cannot be used at a sufficiently higher level of abstraction. The driving forces behind the evolution of the SOA were:

- Integration at the application level
- Integration across organization boundaries/ Crossing trust boundaries
- Interoperability
- Discoverability (Dynamic)
- Flexibility of composing an application to meet changing requirements/needs
- Dynamic assembly of applications

The main difference between the CSA, DOA and the SOA is that SOA can be at a higher level of abstraction and it can encapsulate several levels of abstraction as shown in Figure 17 and Figure 18. It is fractal. Multiple levels of abstraction can be achieved because services can encapsulate much larger chunks of functionality in an interface. A service can be composed of other services by using their interfaces. These services can be of any level of granularity from fine to very coarse. This composite service can be in turn used in a higher composition and so on.

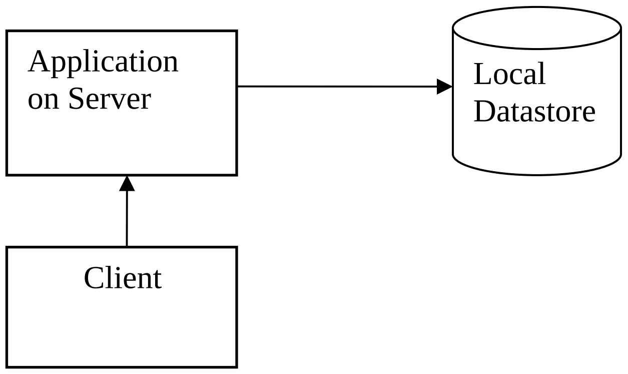

Figure 17 Application based on CSA

Due to the fractal nature of SOA, it is an ideal candidate to build System-of-Systems. The task of engineering a System-of-Systems involves being able to abstract over and encapsulate an entire system. Among all the paradigms, SOA has the capability to do so. While CSA and DOA/CBA can be used to build applications, SOA is better suited for systems at higher levels of abstraction above that viz. systems and system-of-systems.

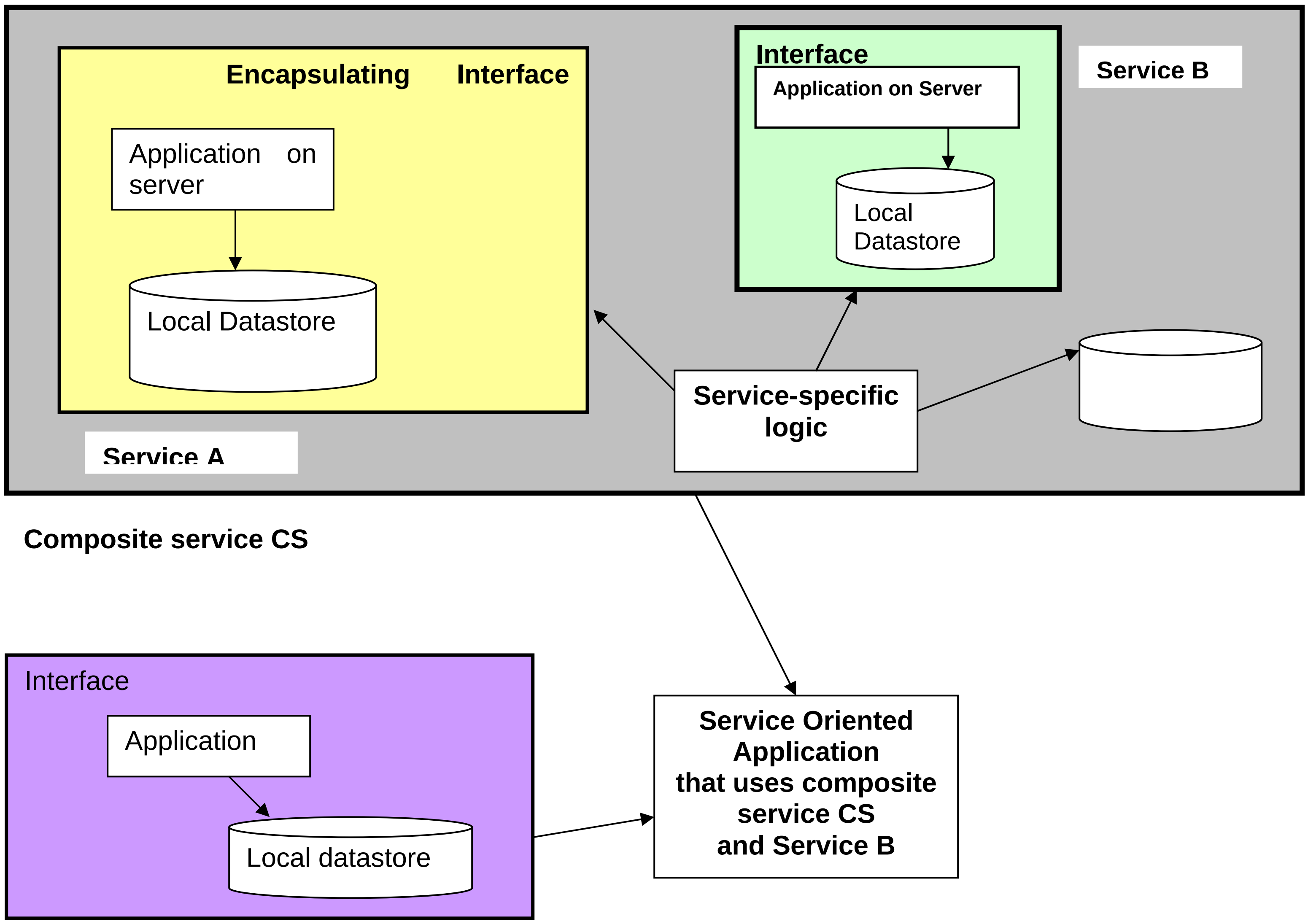

Figure 18 SOA encapsulating entire applications and systems

4.1.4 Granularity and Nature of Software Computing Units

4.1.4.1 Granularity

Levels of abstraction and granularity are different in the sense that granularity of a software entity is defined for a particular level of abstraction. In the DOA, the computing entities are classes of objects that are modeled, designed and implemented using OO principles. In the CBA, components form the units of computing. While a component can be a single class, in practice, a component usually consists of a grouping of several classes thus resulting in it having a coarser granularity than objects. The largest granularity possible is in SOA where the software unit, a service, can encapsulate an entire application as depicted in Figure 19. Since, services are fractal, theoretically, there is no limit to how coarse a service can be. In the same token, nothing prevents a service from being implemented by a single object or component. Thus, a service's granularity may vary from one end of the granularity spectrum to the other. However, since the benefits of using a service-oriented approach might not be apparent below a certain level of granularity, best practices generally suggest using a service at higher levels of granularity, usually at the application level.

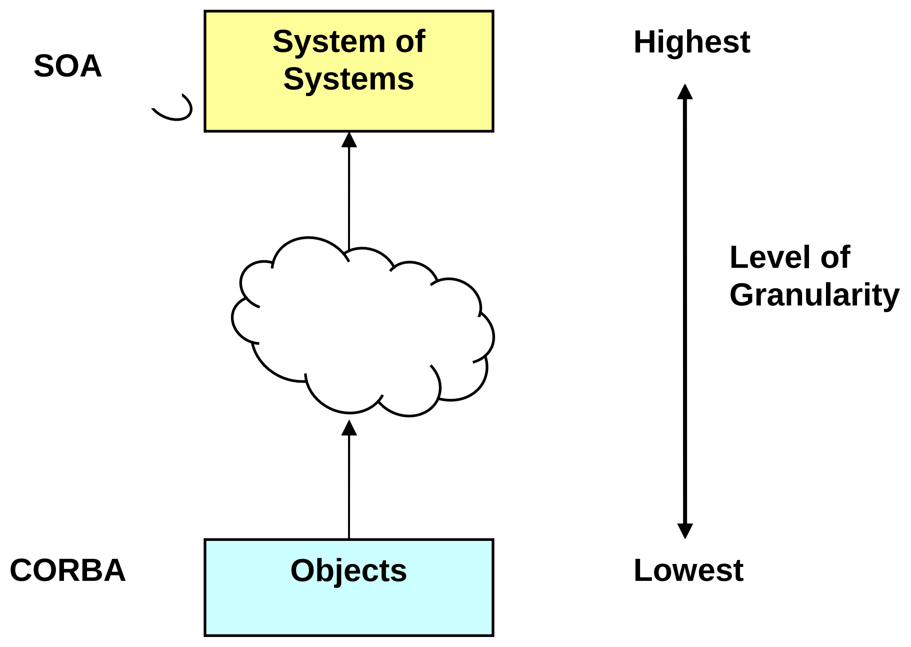

Figure 19 Granularity of the processing elements in the various architectures

4.1.4.2 Nature of the Components

4.1.4.2.1 State

Distributed Objects and components have state. This means that multiple, simultaneous instances of objects and components that are of the same type (class or component type) can exist, with each instance being distinguished from the other by the internal state contained in it.

On the other hand, Web Services have no notion of state that can be seen from the client side. In SOA, services exist irrespective of whether they are being invoked by a consumer or not. That means they are independent of the consumer. As they are not created by consumer invocation or destroyed after the completion of the interaction, their life cycle does not correspond to that of a consumer. Therefore, services do not have an execution context for a particular consumer; they are stateless with respect to consumer invocation unlike objects and components. Rather, it can be said that their existence and state is dependent on the enterprise resource for which they are a front end. An important point to be noted from this discussion is that, when it is said that a Web Service is stateless, it is stateless from a consumer's point of view.

For example, consider a service that provides information about a student given his/her personal identification number (PID). Interaction between a client and a stateful service would proceed as follows:

Client: what is the GPA of the student with PID (Personal Identification Number) 657585960?
Service: 3.6
Client: What is his major?
Service: Physics

In the above example, the Service needs to remember the PID of the student, which is state.

On the other hand, a Client's conversation with a stateless service would proceed as follows:
Client: what is the GPA of the student with PID 77777?
Service: 3.6
Client: What is the major of the student with PID 77777?
Service: Physics

From the above example, it is clear that, state, when required, is maintained in the requesting client application and is passed in messages to the service. Thus, the key to statelessness is intelligent messages that contain all the information required to instigate an action on the part of a web service. The service itself is not required to remember information between one invocation and the other. "Statelessness" means that each invocation is completely independent of the other. The consumer has to assume statelessness. This does not mean that a single instance of a web service proxy on the service side services all requests. Multiple requests may be coming in for a single web service. A Web Service may respond to all these multiple requests by instantiating a new instance for each request. But the client should not assume that the same instance will service consecutive requests from its side.

4.1.4.2.2 Modes of communication

Two models of communication exist for distributed software entities: synchronous request/response and asynchronous message passing/queuing. Traditional CSA uses a blocking, synchronous form of interaction where the server passively waits until it receives a request. The system blocks the client's execution until a reply is received from the server.

In DOA/CBA, interaction is primarily through ORPC (Object Remote Procedure Call). The RPC style of interaction typically involves passing a small number of individual data items in multiple requests, and synchronously getting a small number of reply data items in return. The data is in binary format. In this environment, the decision to invoke a particular synchronous call, and the data to be passed to the call, depends heavily upon the context, which is defined by the previously invoked RPC calls and the data returned by them.

Though CORBA and DCOM primarily use a synchronous Object RPC (ORPC) protocols, they also provide support for interaction styles using messaging – CORBA through CORBA messaging and DCOM through MSMQ (Microsoft Message Queuing).

SOA is based on the message passing style of communication. Web Services and SOA in general use a document-style communications approach. The document-style organizes data within a collection, called business document, and makes far fewer invocations compared with the RPC style. The business document contains all of the information required for business processing and typically contains far more data than the amount passed in RPC-style parameters. While a document processing request/response could be synchronous, an asynchronous approach is far more common. SOAP implements patterns such as request-response pairs as one-way transmissions from a sender to a receiver.

An RPC request call (even when using Web Services and SOAP) maps to a backend method call. As such, the request message contains elements that contain the method name and its parameters. RPC is typically static, requiring changes to the client when the method signature changes. Therefore, strict rules are applied to the data wire format and there is a tight contract between the client and the provider. Document-style on the other hand, does not require a tight contract between the client and the service provider. The contents of a document are described by an XML schema that applies to the whole message itself rather than the parameters alone like in the RPC-style. Thus, using the document style leads to less coupling between the consumers and providers than when using RPC.

In PPA, communication between a pair of peers is based on an asynchronous, symmetric style of interaction. In PPA, the software units on each node are the same and have the capability to play the role of both a client and a server. When these roles are considered in isolation, the overall interaction between two peers can be considered as a pair of client/server interactions – one for each role played by a peer. Thus if A and B are two interacting peers, one interaction would be where peer A acts as a client to peer B playing the role of a server and vice versa in the other. Together, this pair of interactions provides symmetry in PPA interactions. PPA interactions are carried out using low level application protocols over transport layer protocols like TCP/IP like in traditional CSA. However, unlike most CSA applications, the interaction is asynchronous and is typically implemented using callbacks or message queues.

4.1.4.2.3 Autonomy

Services are more standalone. On the other hand, layers in a CSA application, objects or components are not standalone; they do not provide a complete unit of functionality by themselves. Objects and components provide pieces of functionality that can be reused in different applications to build a complete piece of functionality. On the other hand, a service encapsulating an application provides a complete piece of functionality. In PPA, the virtual network itself is considered as an organic whole, an overall functionality. Every peer contains complete, monolithic applications. Thus, the software entities in PPA can be considered as standalone as in SOA. The major difference between SOA and PPA is that services can be composed into applications and other services; where as the nodes in PPA cannot be incorporated hierarchically.

4.1.4.2.4 Life Cycle

A Service is not a "distributed object". Web Services have no notion of objects and object life cycles [Vogols 2003]. In a distributed object environment, communication is by remote procedure invocation on an object. This involves requesting an object instantiation, requesting an operation on that instance of the object, obtaining that result as a reply and releasing the object after use (garbage collection). On the other hand a service is a piece of software that can understand and parse well-defined XML documents that is provided to it through some combination of transport and application protocols. It could be anything from an entire application to a small component and need not necessarily be implemented using object oriented techniques. Thus, communication is document centric – i.e. it is by XML document exchange and does not involve the object life cycle. Therefore, while Web Services can be used to implement RPC style interactions, they are not distributed objects.

Inheritance is another OO principle that has no parallel in SOA. A Service does not inherit from another service like a class does from a base class. A service can be a composition of other services, but this is not the same as inheritance.

The term "service" is overloaded. Most of the confusion regarding what constitutes a "service" stems from the misuse of the word service for any object or component with a Web Service façade [Gamma et al. 1995]. A software unit can be called a "service" in the true SOA sense when it is architected using service-oriented principles and displays the characteristics such as standalone and statelessness listed in the "tenets of SOA".

4.1.5 Intent of Usage and Usage of the Software Units

One of the major yet subtle characteristic of an SOA that differentiates it from other architectures is the intent for the creation of the basic software unit. A service is created with intent of reuse – reuse by customers in ways that cannot be foretold. This requires that the interface be created in such a way that facilities such "repurposing". You cannot see behind the interface of an SOA (opacity). This opacity allows for replacement of parts i.e. modifiability. A service is autonomous in its ability to control its level of opacity. That a single service is an orchestration of several services may not be exposed, but may be inferred by such things as quality of service attributes.

When we work with a component, we work with code. When we work with a service, we work with a contract. Using a component might involve code level activities such as checking it out of a library, connecting it together with other components and compiling. Services on the other hand are not accessed at the source code/binary executable level. They are accessed remotely over a network and they are composed into an application, but not statically linked and compiled.

4.1.6 Nature of the Interface

An interface is a grouping of logically related methods and properties [Lowy 2003]. Interfaces help to control complexity by abstracting away from implementation detail. Separation of the Interface from the implementation and the nature and expressiveness of the interface play a major role in network-centric architectures and systems. Interface-based architecting and development help realize a sizable number of network-centric qualities such as interoperability and dynamic comparison. The nature of the interface helps to do many things like cross heterogeneous platforms and different implementation languages, ownership domains in the case of SOA, allow different versions of a software entity to co-exist and enables lego-style assembly of applications and system of systems. Interfaces make the task of programming large systems significantly easier by enabling interchangeability of components and services.

Interfaces define contracts. The specification of the contract of a component includes the input/output behavior, invariance and dependencies to other components. Architecturally, interfaces encapsulate nodes and provide clear access points. A component or a service is visible exclusively through its interface [Szyperski 2003]. Interfaces are considered to be necessary and sufficient to characterize components. Current interface definition languages include CORBA's IDL (Interface Definition Language) and similar IDL like languages on other platforms and the WSDL (Web Services Description Language) for Web Services.

A closer examination of these languages reveals that they mostly provide support for syntactic specification of the components. They support specifying the functional aspects of the component or node in terms of signatures. The WSDL, the more expressive of the two, also defines operations on services and service end points.

However, information required for the proper use of a component is not confined to the syntactic elements like method signatures or operations. The overall characteristics of a component consist of non-functional properties (quality attributes such as latency and accuracy) and also the internal behavior of the system that may need to be made explicit for proper use of the component. An example would be the "locking" mechanism used in a component and its characteristics. These types of information can be specified by any of the current interface standards.

While, syntactic and functional information might be sufficient to invoke the node, but it might not be sufficient to ensure proper interoperability. To illustrate inadequacy of syntactic functional information, consider a service that provides the current price of the stock of a company when it is provided with the name of the company. Semantic ambiguity would occur when the price is in dollars but is interpreted in euros. Current interface definition practices have no way to accommodate this kind of information into the interface definition. This kind of under specification may work as long as the developers who incorporate the component or service into their application are in the same building. Things can be underspecified by relying on group knowledge and "established practice". However, such assumptions cannot be made for network-centric services. This brings us back to the issue of network-centric systems crossing organizational boundaries. Integration problems occur here as systems are created with some components over which the integrator has less than complete control. Sometimes the integrator application may have no control at all. All these make a firm case for the importance of semantics in the interface definition of network-centric systems.

Several efforts are underway to add semantics to service contracts. An approach championed by the Semantic Web community is to use a different language for service description altogether. Current research centers on the use OWL-S (Ontology Web Language for Services) which was formerly DAML-S (DARPA Agent Markup Language) and its variants as a service description language for Web Services. The most interesting effort however is to add semantics to the WSDL itself. This effort has led to the creation of the Web Service Semantics WSDL-S specification [W3C 2005a]. WSDL-S uses the extensibility elements of WSDL to add semantic annotations to WSDL document elements. This initiative draws upon the OWL-S and METEOR-S [LSDIS 2006] initiatives.

4.1.6.1 Nature of Interfaces in the Different Architectures

Interfaces in CSA tend to be the most fine grained of all. Often, there is no separate interface in the sense of an interface in the component-based or service-oriented development. In fact, one of the major differences between client-server and component-based development is that in component-based development, the focus is on defining and implementing interfaces. In DOA, the solution is modeled using class hierarchies. An interface is not limited to defining methods. An interface can also define properties, indexers and events. Interfaces promote loose coupling between clients and objects because when you use an interface, there is a level of indirection between the client's code and the object implementation.

Both DCOM and CORBA have interfaces defined in an interface definition language (IDL). The IDL describes an object-oriented Interface. Using an IDL can be considered as an external description mechanism. .NET and Java EE are specifications upon programming languages and do not have external descriptions. Both .NET and Java EE do not enforce some core principles of component-based programming, such as separation of interface from implementation, unlike COM and CORBA. While this separation is possible using the programming language "interface" construct in both Java and .NET languages, it is not strictly enforced. In Java EE, using interfaces is mandatory only when EJBs (Enterprise Java Beans). When using POJOs (Plain Old Java Object), interface based development becomes a design choice like in .NET. Thus, it can be said that both .NET and Java EE enable component based concepts, but do not enforce them. Both frameworks allow binary inheritance of implementation.

A service description, typically in WSDL is different from an IDL in that it is not an object-oriented description. It describes types and messages that are grouped into operations. A service contract is totally different from a component interface. In SOA, association of methods in the service interface is a pure logical construct. Service and consequently service interface is effectively a “namespace”, associating together service’s methods, which are otherwise independent entities with their own quality of service requirements, security and versioning strategy. To make a programming language analogy, every method of the service is similar to a FORTRAN subroutine, which can exist and be executed independently from other functions.

While the DOA/CBA and SOA prescribe a standard way for describing interfaces, PPA does not have the concept of service description in terms of interfaces. The description information usually tends to be a simple textual description. In the case of JXTA, it could be a structured XML document, but the format is not specified unlike the WSDL document for Web Services. This is because of the nature of PPA; there is no composition of pieces of functionality into a whole. Every node in a virtual network runs the same application. In P2P networks, the search is for resources and not pieces of computation.

4.1.6.1.1 Component based Interface (IDL) vs. Service Contract (WSDL)

4.1.6.1.1.1 Syntax

IDL is C++ in a different syntax. It is rigid, and not capable of incorporating descriptions of policies. WSDL is based on XML.

4.1.6.1.1.2 Type System

One of the reasons WSDL supports looser coupling than the IDL is because the type system of the WSDL is more flexible. The WSDL type system is based on XML which a mature technology for representing data as self-describing, platform-independent text. Self-describing means several things, first, that data in an XML document identifies itself using element and attribute names, and second that elements identify their type, such as “Integer” using the XML Schema Definition Language (XSD). XSD allows services and clients running on diverse platforms to interoperate over a common type set, and is critical to the success of web services.

4.1.6.1.1.3 Service Endpoints

The WSDL specifies a “Service Endpoint”. The message exchange patterns can be more easily varied, and it is much easier to add new bindings for other protocols and transports.

In IDL, interfaces support multiple interface inheritance. WSDL does not have this construct and therefore interface inheritance is mapped as repetition of the operations declared in the parenting interfaces. Types declared within the parent interface scope are not repeated as that type space is available to the derived interfaces. So, as such, WSDL doesn't have limitation in carrying the information associated with an object that inherits from multiple object definitions. It is just done in a different way than in the IDL.

4.1.7 Degree of Coupling

Loose coupling describes an approach where integration interfaces are developed with minimal assumptions between the sending/receiving parties, thus reducing the risk that a change in one application/module will force a change in another application/module. Loose coupling is enabled by open architectures. Loose coupling enhances the maintainability and reusability of software. It also enhances the scalability and resilience of architectures.

Loose coupling is achieved by the use/incorporation of one or more of the following architectural constraints [Orchard 2004]:

- Vendor and platform independent messages
- Coarse-grained, self-describing and self-contained messages.
- Well-defined interfaces
- Extensible versionable interfaces
- Constrained interfaces
- Stateless messaging
- Human readable strings like Universal Resource Identifiers (URIs) for service and instance addresses
- Stateless messaging where possible and appropriate
- Asynchronous exchange patterns where possible and appropriate
-

4.1.7.1 Coupling and the Different Architectural Paradigms

By comparing the characteristics of CSA with the constraints and mechanisms for achieving loose coupling listed above, it is easy to discern that it has the potential for the highest coupling among all the other architectures. CSA uses synchronous RPC. In a typical 3-tiered architecture, the communication between the tiers (or layers) is very fine grained. It involves the use of entities such as properties, methods, events, delegates and data binding. There is a lot of fine grained communication.

While DOAs lead to strong coupling too, some characteristics of the CBA lead to a looser coupling. Component interfaces may group objects together, thus providing an abstraction of their methods. Thus, communication between layers involves the use of fewer methods as each method is designed to do a relatively large amount of work. CORBA, DCOM, EJB and .NET Remoting use RPC calls where the wire format is binary. This results in much tighter coupling.

SOA ideally operates at an inter-application level. Therefore, communication between services is coarser grained since each invocation of a service results in a much larger amount of work done and hence the communication is not as much as in the CSA and DOA. The third tenet of SOA roughly states that the calls between SOA services and between a client and an SOA service are all XML messages and that only the contracts and schemas are shared between services and clients. This contributes further to loose coupling. Incidentally, CORBA and DCOM cannot be considered as SOAs as they violate this tenet. Both CORBA and DCOM are RPC based and the messages that go inside the wire are binary OO-RPC call.

Another contributing factor to the loose coupling of SOAs is the opaqueness of the service interface. Interface opaqueness means that you cannot see the internals of the implementation behind that interface. You cannot bypass a layer. The SOA interface (or contract) is the most opaque of all. For example, in SOA, you cannot bypass the business layer and access the database layer directly, something which you can do in CSA and DOA. The ability to do this stems not only from the nature of the interface, but also the operational environment and policy. Services are consumed assuming inter-organizational boundaries and thus it is not possible to obtain a description of its internals or access them. Opaqueness can be achieved in a 3 tiered CSA whereas in SOA it is a constraint enforced by the architecture. The nature of SOA is such that opaqueness is an inherent characteristic of the architecture. While separation of concerns is advocated, it is not enforced in the other architectures.

4.1.7.2 Coupling Summary

For Services, loose coupling is relevant at the boundaries of the services and service consumers. It does not matter how things are internally. Boundaries are defined by the interfaces. Therefore, to a great extent it is dependent on the nature of the interface and semantics. While the nature of the interface may be decided to a great extent by the framework chosen, adding semantics and other information is still a matter of architectural decisions.

While using XML protocols like SOAP can greatly enhance loose coupling and flexibility, there are tradeoffs involved. XML can be expensive to parse and is often a larger representation of the data compared to a binary format, so it is more cumbersome to send over a network. The flexibility and loose-coupling offered at the expense of processing efficiency. Tightly-coupled systems that use platform specific binary formats like .NET Remoting or RMI are generally faster. They can transfer data and objects in binary formats that are specifically optimized for the specific implementation language.

4.1.8 Dynamic Discovery (Discoverability) and Composability

4.1.8.1 Dynamic Discovery

Discovery is "the act of locating a machine-processable description of a service that may have been previously unknown and that meets certain functional criteria." [W3C 2004] The goal is to find an appropriate service. Three kinds of discovery mechanisms can be observed [W3C 2004]:

- Directory based
- Index based
- Broadcast

A robust discovery mechanism can be considered as a necessary mechanism for self-healing and recovery from failures such as loss of network-connectivity, or in the case of military situations, disappearance of cooperating components due to physical or cyber attacks, jamming of communication channels and nodes moving out of range. In such volatile environments, service discovery enable systems to rediscover lost components or to find other components that provide essential services needed to accomplish critical tasks [Dabrowski and Mills 2002].

4.1.8.2 Composability

Composability or Compositionality is the composition potential of a software component or service [Belloir et al. 2003]. Composability is a requirement for proper reuse. Composability is influenced by openness, interoperability and modularity. Two kinds of compositions can be observed in practice: composition at compile time (development time) that can be called as static composition and runtime composition. Runtime composition or dynamic composition requires dynamic discoverability and late binding.

Properties of composable software work units include modularity, openness, interoperability, autonomy, explicit statement of dependencies, expressive interfaces, statelessness, discoverability and late binding.

Dynamic discovery and composition facilitate building communities of interest [Lau 2004]. A community of interest (COI) is defined as "any group with a common mission interest and informational needs" [DoD 2006]. Technically, a COI implies a common capability that should be provided by a system or by some combination of system-of-systems (SoS). Without dynamic discovery, it is not possible to create new capabilities by composing existing capabilities even if they are provided as services. Dynamic discovery and composition allows creation of new and different capabilities by combining and recombining the capabilities provided by existing services. If the services are provided using Web Services technologies, this can be achieved by orchestrating Web Services. Thus, dynamic composition helps to meet changing requirements and to create dynamic communities of interest.

4.1.8.3 Dynamic Discovery and Composition in the Various Architectures

CBA and SOA promote modularity to a greater degree than the other paradigms. SOA has the potential to help realize modularity at greater levels of abstraction. Modularity in the Web Services Architecture is promoted by its self-describing nature using WSDL.

CSA does not have the concept of dynamic discovery. Everything is statically coded. CORBA and DCOM can be said to have a limited kind of service description and discovery. The IDL interface can be considered as a form of service description. Both frameworks provide a form of "Naming Service" that can be used to lookup a CORBA based or DCOM based object. Of course, clients must know the correct name in order to discover (or recognize) a service and all functional information is only implicit, being assumed to be known independently.

To elaborate, CORBA's naming service that lets you look up a remote object by name and obtain a remote reference to it. However, in order to use the remote reference by invoking methods on a local stub, CORBA requires that the client has the definition of the stub locally. (In other words, CORBA requires that the code for the stub object be known to the developers that create the client.) CORBA does offer a "dynamic invocation interface" (DII) that enables clients to use remote objects without the stub definition, but it is more complex to use than just invoking methods on a local stub. The difference in complexity is similar to the difference between using a Java object through its interface and using the same Java object via the reflection API. The CORBA trader service is something that comes closer to UDDI for Web Services. In the trader service, instead of just supplying a name with which a remote object is associated, as you do with the CORBA naming service, you describe the type of remote object you are seeking. The CORBA trader service returns a remote reference to a matching remote object which can then be used by the client through a local stub.

SOA is inherently dynamic as one of its cornerstones is dynamic discovery and composition. Discovery in Web Services is accomplished using UDDI and WSDL. The use of UDDI implies that Web Services follow a centralized directory model for discovery.

A service can be said to be much more dynamically composable because it can be discovered dynamically based on its WSDL description and incorporated into the application. Services in an SOA are discoverable in the sense the distributed nature of services is not transparent, it is explicit. In contrast, in distributed objects, remote object invocations are transparent. However, even in SOA, completely automated discovery and composition on the fly to give rise to new capabilities requires a level of semantic negotiation not possible yet.

Jini has a distributed service discovery mechanism in its Lookup service built atop Java. The major difference from the Web Services approach is that several directories can exist within one Jini network and the discovery process is by using a “multicast request protocol”. Jini services may also come to know about the existence of a directory service through an advertisement by a directory service of its existence. Directory services advertise their existence using the multicast announcement protocol.

4.1.8.4 P2P Discovery Process

The discovery process in PPA is different from Web Services. Pure PPA is decentralized. There is no central directory; instead broadcast protocols are used to discover peers. At discovery time, a requester peer queries its neighbors in search of a suitable peer. If any one of them matches the request, then it replies. Otherwise each queries its own neighboring peers and the query propagates through the network until a particular hop count or other termination criterion is reached. One of the advantages of the PPA approach over UDDI is the fact that the registration of peers is done automatically and is very simple in nature.

PPAs do not need a centralized registry, since any node will respond to the queries it receives. Therefore, PPAs do not have a single point of failure, such as a centralized registry. Furthermore, each node may contain its own indexing of the existing peers. Finally, nodes contact each other directly, so the information they receive is known to be current. (In contrast, in the registry or index approach there may be significant latency between the time a Web service is updated and the updated description is reflected in the registry or index.)

Even in hybrid PPA, the directory is distributed. In Web Services, there is only one UDDI directory i.e. it is not yet a distributed directory.

What prevents us from building applications that are coalitions of services on different nodes using the P2P paradigm? The answer lies in the key differences between the SOA and PPA paradigms.

4.1.8.4.1 The Concept of a Virtual Overlay Network

An overlay network is a computer network which is built on top of another network [Wikipedia 2006]. Nodes in the overlay can be thought of as being connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links, in the underlying network. Overlay networks can be constructed in order to permit routing messages to destinations not specified by an IP address.

 P2P architectures have the concept of a virtual overlay network as illustrated in Figure 20. Most peer-to-peer networks are overlay networks because they run on top of the Internet or some other network. Many peer-to-peer protocols including Gnutella and Freenet are overlay network protocols. There is no concept of an Overlay Network in SOA.

This concept of a virtual network over a network, usually a TCP/IP network, results in several interesting properties of P2P networks.

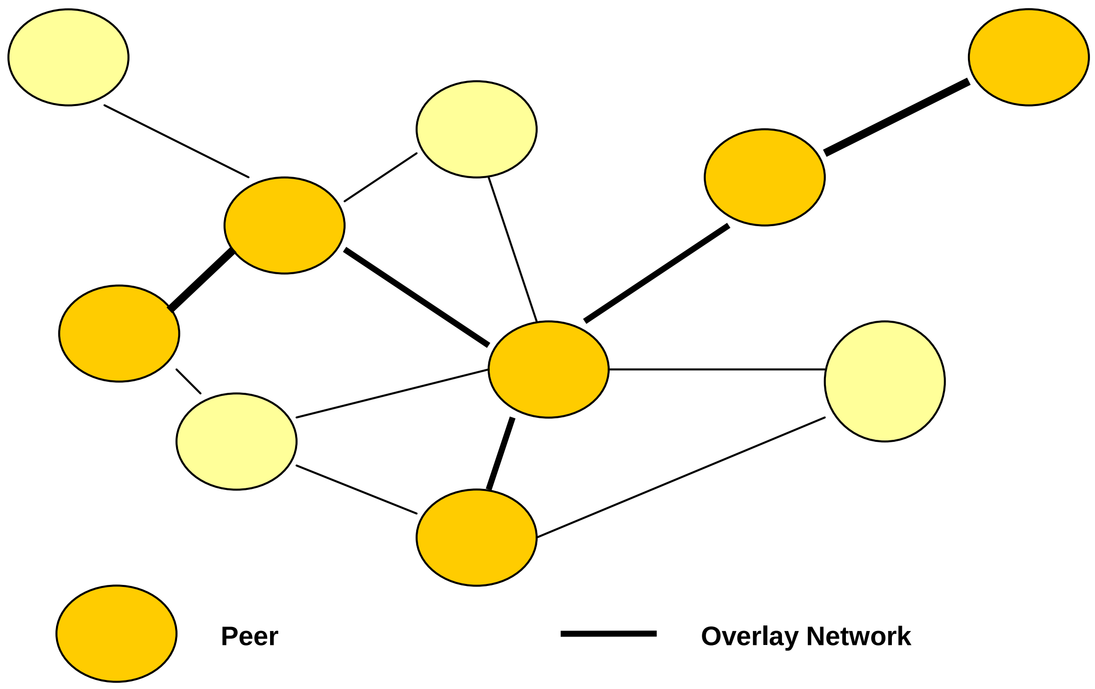

Figure 20 Peer-to-Peer overlay network

4.1.8.4.1.1 Dynamic Addressing

Due to the fact that peers join or leave the virtual overlay network often, peers cannot establish direct contact without discovering each one another first. In a overlay network, nodes are identified not by static URLs but by dynamic IDs. Because it is a virtual network, every node needs an ID that is unique in the virtual network. On the other hand, in SOA which does not have the concept of an application level overlay, you use static URLs, as virtual IDs are not required.

In PPA, every node joining the network has to register itself and the provided resources. There is no such thing in SOA because there is no concept of an overlay network. There is no concept that nodes are connected to each other except that they are available over a network. In PPA being a peer means that it is available on the network at least at the time the information was obtained. Peers mostly provide very simple information during the registration process. Because of this, discovery of information in PPA has to stick to a simple search queries.

4.1.8.4.1.2 SOA vs. PPA for Creating an Application Composed of Services

Currently in SOA, dynamic discovery is not used. Addresses of services are statically embedded into the application at compile time by the programmer. This is not possible in a PPA environment because of the dynamic nature of the nodes and the transient nature of their connection to the network. PPA nodes do not have fixed URLs. Therefore, dynamic discovery is one of the prerequisites for service compositions in PPA. Also, Services in a P2P environment can be considered as transient whereas SOA services are persistent.

In order to bring out the differences between SOA and the different flavors of PPA, let us consider a hypothetical scenario where an application is created using functionality distributed onto different nodes. The application's functionality in this example is partitioned according to SOA principles. In PPA, as in SOA, once the node containing the application has joined the network, it could dynamically search for services that provide the missing piece of functionality and uses them in accomplishing its task. Conceptually, this looks similar to SOA with dynamic binding. However, a closer examination of the details reveals important differences between PPA and Web Services.

While dynamic coalition of services is not a reality even for SOA, it is hard to see how it would work for peer-to-peer topologies where service providers are more transient. Mechanisms for partial or complete failure detection and recovery and for discovery of a semantically equal service would be required – the same challenges faced by web services.

Another difference would be that using the application that leverages the services in PPA would be like using any other PPA application. Users have to download it onto their computers and run it from there instead of accessing it from a server like you could do in on SOA.

From the scenario, it is possible to extrapolate a few things. If replicated versions of applications and services could be deployed, in systems requiring survivability (like military systems), a PPA approach to discovery and redundancy could prove to be more robust. UDDI directories represent single points of failure for Web Services. If using a distributed directory approach or broadcast protocols over a overlay network like in PPA, an alternate replica of the service (without considering semantic equivalence) could be found and used. As long as all the services can be found, PPA may prove to be more robust than SOA. Thus, the SOA idea of decomposing pieces can be combined with PPA's robust connectivity to build highly resilient networks.

4.1.8.5 Dynamic discovery and composition summary

Composability is also related to satisfiability – whether the component or service helped to fulfill the requirement for which it was composed. Hinton [1997] cautions about emergent behaviors, the behaviors that arise because of the interaction of the component behaviors in composite system and how they can play a role in hindering satisfiability of component compositions. He argues that undesirable emergent behaviors are a result of the under-specification of components and services highlighting the importance of openness and complete interface specification in proper composability.

4.2 Comparison between Architectures and Frameworks based on Quality Characteristics

4.2.1 Openness

4.2.1.1 Evaluating Frameworks for Openness

Frameworks are more concrete than generic architectures. They are either complete specifications for implementations or actual implementations. Therefore, when evaluating a framework for openness, the first question to ask is about the openness of the framework itself considered as a product. Are the specifications of the framework open and available? For commercial frameworks whose specifications are not available, how open and well documented are the interfaces supported by the framework? Their interfaces are usually the APIs supported by the programming models for that platform. The next consideration is whether the frameworks aid in the construction of open systems. Aspects to consider in evaluation include:

- Are the protocols used by the framework open and standards based?
- What concrete mechanisms do they provide for self-description (in the form of interface definition?)
- Is the interface definition language (IDL) expressive enough for complete component description? Is it a standard? Is the IDL language independent or specific?

4.2.1.2 CSA

In CSA, there is no self-description in terms of interfaces. In fact, separation of interface and implementation is a design decision as opposed to an architectural dictate. Documentation is through external means such as ADLs (Architecture Description Language), plain text using natural languages or other external tools like diagrams. Openness of CSA with a monolithic middle layer is thus a matter of detailed architectural specification and design and has to be decided on a case by case basis.

4.2.1.3 .NET

Microsoft has secured certification for both C# and CLI (Common Language Infrastructure) from ECMA and ISO/IEC as industry standards. CLI is Standard ECMA-335 while C# is Standard ECMA-334. In doing so, Microsoft released all intellectual property in the core C#/CLI platform to the public domain which means that a license is not needed to implement C#/CLI. This will also help in better understanding the implementations of C# and CLI which are at the core of .Net platform. Microsoft also provides “The Microsoft® Shared Source CLI Implementation” which is a file archive containing working source code for the ECMA-334 (C#) and ECMA-335 (CLI) standards.

However, the whole of .NET platform is not a public domain standard; Microsoft still owns complete intellectual property rights to several class libraries and APIs within the .Net platform. The non-standard parts of the .NET platform include Windows.Forms, ADO.NET and Web Services.

The major communication protocols used in .NET are mainly DCOM and SOAP. While SOAP is an open protocol, DCOM is proprietary.

A major feature of the .NET framework is its support for XML. .NET has a suite of XML classes for creating and parsing XML documents that fully conform to the current W3C recommended standards of XML like Namespaces, XSLT, XPath, Schema, and the Document Object Model (DOM) [MSDN 2006d]. These classes support the W3C XML Schema Definition language (XSD) 1.0 recommendation [MSDN 2006d]. XML can be considered as a core technology in .NET as the other parts of the .NET Framework including ASP.NET, Web Services and ADO.NET use XML as their native data representation format [Skonnard 2001].

4.2.1.3.1 Assembly Metadata

.NET assemblies contain metadata in a standard format. Metadata makes assemblies self describing by providing information on types (classes), methods, dependencies on other modules, etc. However, since the .NET metadata is in binary format, it has to be retrieved at runtime using reflection.

Lumpe [2002] argues that the custom attribute facility in .NET provides a way to represent functional and non-functional properties of components. If we take a broader view of openness as not just explicit documentation, but information in general that can be obtained about a component, custom attributes can be seen as contributing to openness. Custom attributes have the potential to provide more information about a component as they are highly accurate being in such close proximity with the code. Reflection can be used for discovering attributes at runtime.

4.2.1.4 Java EE

The Java EE platform is a set of specifications maintained by the JCP (Java Community Process). While only a subset of the .NET APIs are covered by ECMA standards, all the Java specifications are defined by the JCP.

Communication between Java components is achieved using RMI (Remote Method Invocation). JDK 1.5 also supports version 2.3.1 of the CORBA IIOP (Internet InterORB Protocol). Java EE also supports the RMI-IIOP protocol, which is Java RMI but using the IIOP protocol for communication.

Java EE provides support for WS standards through the JAX-WS (Java API for XML-based Web Services) API. Support for XML Schema and parsing is provided in JAXB (Java Architecture for XML Binding). Java EE 5 also provides “annotations” which is similar to .NET attributes.

4.2.1.5 DOA/CBA

Clear separation of interface and implementation is an architectural constraint in DOA and CBA. Components are specified using an Interface Definition Language (IDL). IDL is used to define the interfaces for accessing and operating upon service components. However, it provides a signatures only specification. The primary purpose of IDL based specifications is type checking between client code and independently developed components.

4.2.1.6 CORBA

CORBA is an open specification maintained by the OMG [OMG 2005]. As such, the interfaces of the standard CORBA services and the communication protocols used are open. The inherent openness of CORBA applications are a function of the open protocols used for communication and the expressiveness of the IDL.

4.2.1.7 DCOM

The COM, Active-X and DCOM specifications was turned over to the Open Group [Open Group 2006] by Microsoft in 1998. Thus, DCOM is often considered to be an open group standard [MSDN 1997a].

DCOM supports two interchange formats for contract description: Microsoft's IDL (MIDL) and type library (TLB) files. The MIDL's expressiveness is similar to that of the CORBA IDL. COM has no support to describe component dependencies. The lack of dependency information makes it difficult to determine what DLLs would be needed to deploy COM-based components. The COM contract description format is also not extensible.

4.2.1.8 Web Services

Web Services themselves are a suite of open specifications. Web Services are able to describe their own input and output requirements using the Web Services Description Language (WSDL). Web Services use existing Internet protocols like HTTP and SOAP which are open and standard.

4.2.1.9 Jini

Jini's licensing model places restrictions on the way derivative works using Jini can be redistributed. This detracts from its openness. The Jini specifications were first released by Sun Microsystems under the Sun Community Source License (SCSL) in 1998 [Sun Microsystems 2006b]. This licensing model proved to be quite complex when it came to defining how developers who used Jini could license their work and was considered to be restrictive. In response to this, Sun Microsystems released the Jini technology specifications under the less restrictive Apache 2 license [Apache 2004]. In terms of service description, the Jini service contract is a Java interface like for EJBs.

4.2.1.10 PPA

In PPA, there is no concept of interface based development. The applications are monolithic. Openness in this paradigm can be interpreted as the "openness" of the protocols used for purposes such as communication between the peers or for forming the virtual network.

4.2.1.11 JXTA

The JXTA protocols can be considered as "open" since they are the work of an open source project. However, true openness in PPA is a function of design and policy like in CSA as the application level protocol determines whether an application on a peer can interoperate with a particular virtual network of peers or not. Details of this protocol cannot be captured inherently using the JXTA framework.

4.2.2 Interoperability

4.2.2.1 .NET

4.2.2.1.1 Interoperability with Applications Developed Using the Same Platform

For interoperability from a syntactic point of view, .NET provides seamless interoperability with other applications developed using the same platform i.e. .NET. This is because Microsoft is the only vendor who provides a full implementation of the platform and so there are unlikely to be any inconsistencies. Even though, other open source implementations of the CLR exist, like Mono [Novell 2005], most of the application development is done using the implementation provided by Microsoft as it is the most comprehensive and widely deployed.

4.2.2.1.2 Interoperability with Applications Developed Using Other Platforms

Interoperability of .NET with Java EE without using Web Services can be achieved using third-party vendor's products like runtime bridges. The most popular runtime bridges include:

- Ja.NET and J-Integra.

Ja.NET makes it possible to write clients for Enterprise Java Beans in a .NET language targeting the .NET platform. In essence, Ja.NET java components act as though they are .NET components, and vice versa because Ja.NET leverages .NET remoting [Peltzer 2004].

- JNBridgePro [JNBridge LLC 2006] is another runtime bridge for .NET/Java EE interoperability.

Using an ORB is also a feasible interoperability strategy. However, to use CORBA, a ORB implementation for the .NET platform is required. MiddCor.NET [Middsol 2006] is one such product. MiddCor.NET is a CORBA ORB implementation for the .NET platform. Since there is an ORB implementation for Java EE, interoperability between .NET components and Java EE components could be achieved. However, it should be noted that this capacity is not a native capability of the platform, but depends on the use of an external product from a vendor. .NET does not provide native support for CORBA's interoperability protocol IIOP nor does it provide an IDL compiler.

4.2.2.1.3 .NET and Web Services

While Web Services technology is platform-neutral, .NET provides several APIs and tools for developing Web Services and wrapping and exposing existing applications as Web Services through its ASMX (Active Server Methods) technology. It also supports Basic Profile 1.0.

4.2.2.2 Java EE

4.2.2.2.1 Interoperability with Applications Using the Same Platform

With Java EE, since it is a specification, the implementation is provided by different, often competing vendors. Vendors often add extensions that do not confirm to the Java EE specifications. Minute differences in implementation of the specifications for the application servers and containers can impede seamless interoperability. For example, an application designed specifically for BEA's application server won't necessarily be able to run on IBM's WebSphere Application Server. To address this, Sun has provided the J2EE Certification program. The J2EE certification program is a suite of tests and reference material to ensure that J2EE applications don’t target any vendor-specific extensions and thus are portable across all these different application servers.

4.2.2.2.2 Interoperability with Applications Using Other Platforms

Java provides Java IDL and supports the IIOP protocol for interop with CORBA. Java EE also supports the Web Services protocol stack via JAX-WS 2.0. JAX-WS 2.0 supports the Web Services Interoperability (WS-I) Basic Profile Version 1.1 [JCP 2005a]. To support WS-I Basic Profile Version 1.1, JAX-WS has the following features: The JAX-WS runtime supports doc/literal and rpc/literal encodings for services, static ports, dynamic proxies, and DII.

4.2.2.3 CORBA

Interoperability in CORBA is through the use of ORBs and the CORBA protocols. When you use CORBA, you need an ORB at both ends. Differences in vendor implementation of an ORB may hinder proper interoperability due to slight differences in interpretation and implementation by vendors of the CORBA standards [Ironside et al. 2001]. CORBA provides a quite restricted Interface Definition Language, which allows one to specify only operational signatures of objects. CORBA does not offer mechanisms for semantic interoperability.

4.2.2.4 DCOM

Interoperability of DCOM with components developed on platforms other than DCOM and Windows is achieved using a bridging strategy. DCOM interoperability is hindered by the fact that the DCOM standard is not as widely accepted in the industry as CORBA. Like CORBA, DCOM, which uses an IDL, has no facilities for semantic interoperability.

4.2.2.5 Web Services

Using only the official tenets of service orientation, it is quite possible, in fact, to build a service-oriented application using proprietary message formats and communication protocols. Doing so creates a flexible system that can't talk to anything but itself. However, since interoperability is a fundamental requirement, the technology choices for these shared assumptions must facilitate interoperability, not reduce the ability to work together.

A major reason for the meteoric rise of Web Services was the promise of seamless interoperability. Web Services use document style messages that offer the flexibility and pervasiveness that CORBA and DCOM cannot provide. Web Services through the use of open and widely accepted standards, foster what can be called “intrinsic” interoperability. “Intrinsic” interoperability does not require the use of runtime bridges and other mechanisms. As long as the software is built using standards, interoperability just happens. Because of the wide acceptance of standards, services provide not only interoperability, but also interchangeability preventing vendor lock in. Thus, services provide more choice than other “open” paradigms like CORBA which do not offer both at the same time.

4.2.2.6 Jini

Jini is Java-centric as it is an extension of the Java programming language. A Jini object is essentially a serialized java object, and Jini uses Java RMI API in order to provide a communication mechanism for activating, locating and removing object groups. While developers can write a service implementation in a language other than java, each object must be encapsulated using JNI so that the java environment can dynamically load the objects. This means that every device registered within a Jini community must have the ability to execute a JVM. Jini’s Java-centricity severely hampers its capacity to interoperate.

4.2.2.7 JXTA

The project JXTA specification is generic. JXTA achieves interoperability in terms of the six underlying JXTA protocols written in XML in terms of deployment platform, implementation language and network protocol. JXTA applications can be developed in any language and can interoperate with other JXTA applications regardless of their implementation language and underlying operating system as long as they confirm to the JXTA protocol specifications.

4.2.2.8 Interoperability Summary

Interoperability in CSA, DOA and SOA is for integration of logic, for combining software business logic. On the other hand, in PPA, it is more for exchange of information.

4.2.2.9 Legacy System Integration

Integration, in general, can be achieved in three ways [Hophe et al. 2003]:

- Presentation Integration
- Functional Integration
- Data Integration

Many large corporations have existing code have a number of legacy systems, such as CICS/COBOL, SAP R/3 or Siebel. With legacy systems, it might not always be possible to achieve functional integration, which is the most preferred for reasons of ease and stability of integration and maintaining the integrity of the system. For many older systems, “screen scraping” is the technique used for achieving integration.

There are several ways to achieve legacy integration using Java EE, including:

- Java Message Service (JMS) to integrate with existing messaging systems
- Web services to integrate with any system
- CORBA for interfacing with code written in other languages that may exist on remote machines.
- JNI for loading native libraries and calling them locally.

- J2EE Connector Architecture (JCA). The JCA is a specification for plugging in resource adapters that understand how to communicate with existing systems, such as SAP R/3, CICS/COBOL, Siebel, and so-on. If such adapters are not available, you can write your own adapter. These adapters are reusable in any container that supports the JCA.

In .NET, legacy system integration can be achieved through:

- Host Integration Server 2004 for IBM platform interoperability. Host Integration Server can be used to integrate IBM host applications, data sources, messaging and security systems.
- Web Services for any system that can be encapsulated as a service.
- COM Transaction Integrator (COM TI) can be used for collaborating transactions across mainframe systems.
- Microsoft Message Queue (MSMQ) can integrate with legacy systems built using IBM MQSeries.
- Finally, BizTalk Server 2004 can be used for process integration.

4.2.3 Adaptability

4.2.3.1 .NET

4.2.3.1.1 Modifiability

.NET while providing the ability to architect systems using layering strictly, gives more leeway to write code that mixes up all the layers. This is possible due to the programming model where the presentation can be developed using server side controls and powerful data binding capabilities. This often results in the presentation and business logic layer interface becoming very fine grained or non-existent as the presentation and business logic code being mixed together. .NET does not enforce strict layering as is possible to write all the business logic in the code-behind files in ASP.NET, thus achieving more of a flatter model

4.2.3.1.2 .NET Attributes and Contexts

The .NET framework can use contexts as an object's execution scope and intercept calls going to and from the object, similar to the way COM+ provides component services. What is new with this mechanism is that the runtime allows developers to take part in the interception chain and add powerful services, thus extending existing component services. This in turn decouples the business logic from the system plumbing and simplifies long-term maintenance.

4.2.3.1.3 .NET Versioning

The versioning mechanisms of the .NET framework are one of the most advanced as it allows existence and usage of multiple versions of one component. .NET assemblies are the objects of versioning where the version number is a quadruple of 16-bit integers which is specified by the developer. Next to private assemblies, which can be used by local assemblies only, one can deploy multiple versions of an assembly to the global assembly cache (GAC) to share it with applications in the computer system. Shared assemblies must be extended with a strong name, which is some kind of UID based on public key signature for authenticity and integrity.

However, changes which are not reflected correctly by the version numbers will cause unpredictable effects.

The manifest of a .NET assembly records all dependencies to external assemblies specified by their name, their version number and the string names, if existing. The manifest may contain, in addition to the version number, some metadata like name of the developer and other description which can be retrieved at runtime by reflection.

4.2.3.1.4 Reflection and Metadata

The .NET platform provides reflection at the introspection level in the System.Reflection namespace [MSDN 2006a]. Using the System.Reflection namespace, it is possible to obtain information about classes, fields, methods at runtime. However, the .NET framework offers the facility to dynamically generate MSIL code at runtime in a limited way (it only permits to create new types, not add methods and other members to existing classes and objects) by means of the System.Reflection.Emit [MSDN 2006b] namespace. Structural reflection capabilities can be used to provide a degree of computational reflection capability by wrapping method invocations, etc. Ortin et al. [2005] reports on an attempt to extend the CLI by providing it with a set of structural reflection primitives.

4.2.3.2 Java EE

4.2.3.2.1 Modifiability

While architectures without containers are possible in Java EE (using just POJOs), the use of containers in the Java EE framework can be thought of as a way of enforcing layering automatically making it a little less easy if not impossible to write mixed code. The use of EJBs can be seen as an encapsulation mechanism. A lot of plumbing issues like security, transaction, pooling and caching issues are delegated to the application server with the use of EJB.

4.2.3.2.2 Session Beans

A client can access a session bean only through the methods defined in the bean's business interface. The business interface defines the client's view of a bean. All other aspects of the bean--method implementations and deployment settings, are hidden from the client. Well-designed interfaces simplify the development and maintenance of Java EE applications. Not only do clean interfaces shield the clients from any complexities in the EJB tier, but they also allow the beans to change internally without affecting the clients. For example, if you change a session bean from a stateless to a stateful session bean, you won't have to alter the client code. But if you were to change the method definitions in the interfaces, then you might have to modify the client code as well. Therefore, it is important that you design the interfaces carefully to isolate your clients from possible changes in the beans.

4.2.3.2.3 Entity Beans

The use of entity beans improves modifiability by providing a nice object-oriented abstraction to persistent data in a relational database, but they demand a lot of memory and can incur a high number of database calls. If not used in the right situations or not configured properly, entity beans may yield poor performance. An analysis may be needed to determine if the use of entity beans in this situation hampers performance and if additional risk mitigation activities are required. In addition, another tradeoff can involve application server specific optimizations that reduce portability of the system.

On the flipside, Java EE's EJB has often been accused of being too cumbersome to program raising questions about its modifiability [Krastev and Galletly 2003]. It adds more complexity compared to POJOs as every session bean consists of at least three Java classes, while every entity bean comprises at least four. This led to the rise of lightweight containers like Spring in the first place. To deal with this, EJB 3.0 introduced dependency injection using "annotations" which can be used to achieve a programming model similar to what can be achieved using design frameworks like Spring. What is interesting to note is that it also increases the similarity between Java EE's programming model and that of .NET. Other than cumbersomeness, another major problem reported with respect to EJB has been its performance [Prechelt 2003]. EJBs have been reported to degrade performance considerably. Also, the persistence provided by entity beans have been considered to be not enough or expressive enough.

4.2.3.2.4 Location Transparency

To a remote client, the location of an enterprise bean is transparent. This facility is provided by JNDI and LDAP.

4.2.3.2.5 Annotations and Deployment Descriptors

To create an enterprise bean that has remote access, you must annotate the business interface of the enterprise bean as a @Remote interface. The remote interface defines the business and lifecycle methods that are specific to the bean. For example, the remote interface of a bean named BankAccountBean might have business methods named deposit and credit.

4.2.3.2.6 Patterns

While patterns are available for Java EE, it is debatable whether this contributed to modifiability as excessive use of patterns can prove to be detrimental [Johnson 2004]. Several frameworks are available that use these patterns.

4.2.3.2.7 Versioning

In Java EE, packages are the objects of versioning. Without using complicated workarounds, it is not possible for multiple package versions to exist in one system in Java EE [Stuckenholz 2005]. Some of the conflicts that arise are listed in [Poddar 2004]. They include class loading conflicts, servlet path conflicts, JNDI namespace conflicts, etc. Workarounds include using multiple classloaders and keeping each version of the application component class in different JAR files.

4.2.3.2.8 Reflection and Metadata for Dynamic Reconfiguration

Like .NET, Java EE has a reflection API that provides runtime introspection.

4.2.3.3 Web Services

4.2.3.3.1 Modifiability

In SOA, the focus is on composition rather than building applications. In SOA, the "service" can provide an interface based abstraction over software entities of any granularity. Perhaps, the strongest feature of the SOA paradigm is that a service can be used to abstract an entire system by hiding its technology, implementation, etc. and provide a standard, message based interface to it.

4.2.3.3.2 Versioning

There is no direct way to version web services yet. The current workaround is to use XML namespaces. An XML namespace string is unique. A date or version stamp can be appended to this namespace.

4.2.3.3.3 Dynamic Reconfiguration

In SOA, an application can be seen in terms of a coalition of nodes. Thus, changing the architecture often means changing the topology in terms of the nodes involved. One of the cornerstones of SOA is dynamic discovery. Architectural reconfiguration using dynamic discovery is a distinguishing capability of SOA (and Web Services). Architectural dynamism may also be achieved in Web Services using an event based style.

4.2.3.4 CORBA

4.2.3.4.1 Modifiability

Changeability and extensibility of components are provided through the use of interfaces in CORBA. Changes to server implementations are transparent to clients if they don't change interfaces. Changes to internal broker implementation does not affect clients and servers. Thus, one can change communication mechanisms without changing client and server code.

CORBA provides location transparency as CORBA clients/servers do not care where servers and clients are located. This is through the Common Object Service (COS) Naming which provides a registry to hold references to CORBA objects. COS Naming is conceptually similar to the RMI registry.

4.2.3.4.2 Versioning

CORBA has versioning problems [Stuckenholz 2005]. The CORBA specification [OMG 2005] does not contain any approaches to handle component evolution at all. On the basis of the current CORBA specification, it is neither possible to enrich components with version information, nor to run more than one version of a single component in a system.

4.2.3.4.3 Reflection

CORBA also supports dynamically discovering information about remote objects at runtime. The IDL compiler generates type information for each method in an interface and stores it in the Interface Repository (IR). A client can thus query the IR to get run-time information about a particular interface and then use that information to create and invoke a method on the remote CORBA server object dynamically through the Dynamic Invocation Interface (DII). Similarly, on the server side, the Dynamic Skeleton Interface (DSI) allows a client to invoke an operation of a remote CORBA Server object that has no compile time knowledge of the type of object it is implementing.

4.2.3.5 DCOM

COM does not use a centralized registration and identification service. The most common way to do a component upgrade in COM is to remove the old component and replace it with a newer component.

4.2.3.5.1 DCOM Versioning

Once an interface of a DCOM component is published, it gets a unique identifier (IID) that is unique also beyond the boundaries of all computers. In DCOM, an interface, once published, cannot be changed. If the component offers new functionality, or modified functionality, rather than changing an interface, this is exposed through a new interface that gets a new IID [MSDN 1996b]. This practice is useful to ensure that component clients are never disabled by installing a newer version of a component. But this also prevents the client from knowing the features of the new component versions without rebuilding them.

4.2.3.5.2 Reflection and Metadata

All COM interfaces must derive from IUnknown which supports three methods; Addref, Release and QueryInterface. The QueryInterface method can be used to perform runtime introspection on a COM object.

4.2.3.6 Jini

Jini, like Web Services, also provides dynamic discovery of services through its Lookup service and discovery protocols.

4.2.3.7 JXTA

As P2P applications are traditionally monolithic and JXTA is a set of protocols for communication, there isn't much that can be said in terms of application modifiability. PPA is not about how to construct an application. Hence, JXTA has almost nothing on this subject in its specification. Modifiability (and other quality characteristics like performance and scalability) is often used in a different sense in PPA. In PPA, the focus is on the system. The “system” is a network of computing nodes running usually the same application or sometimes different applications.

Since applications are tightly coupled with the protocols in PPA, modifiability may mean being able to swap different versions of the application and still being able to function in the network. Modifiability may also mean being able to use different types of applications within the same network and to introduce new kinds of applications within the same network. Since JXTA is an open set of fairly generic protocols, this may be easy to do so. However, since JXTA protocols are fairly low level, they might not be enough for this scenario.

4.2.4 Security

The traditional approach to security can be termed as the "islands of security" model. In this, security is applied/considered at the application level. Each application has its own security mechanisms and policies. Also, the applications do not span domains of autonomy. They work with the knowledge of how they will be used. Unique security challenges arise in network-centric systems architected using SOA that comprise of a collection of services as these services can be composed into applications that span operational boundaries. Enforcing and maintaining end-to-end security is the biggest challenge in network-centric systems when nodes involved in an application are geographically distributed, run on heterogeneous platforms and span autonomous regions. This is because tried and tested security policies and mechanisms that worked for non network-centric applications are no longer valid. This requires formulating policies at the highest level and not leaving it to the individual applications to implement it. It also requires propagating the policies to the various nodes involved and ensuring compliance, which is a difficult task.

From the perspective of a single service, security challenges arise as SOAs provide an additional layer of abstraction that exposes business functionality as services that are both location independent and discoverable on the network. However, this leads to a breakdown of traditional models of security. Consider the problem of authentication and authorization. Since a service can encapsulate a system, an SOA can consist of backend systems. The various backend systems can have various security mechanisms and policies i.e. users may have different passwords and privileges with each system. So, when users access a composite system or service, they still need to be authenticated to the backend systems. But since the service composition layer acts as an abstraction layer, and masks the underlying technology and implementation details from the users, the service, in effect masks the user identity context from the underlying applications. This makes it difficult to associate the users of the overall functionality, since the SOA provides no security context. For example, consider an accounts system that is exposed as a service. One of the functionalities it offers is the ability to retrieve the salary of an employee, given the employee ID through the get_salary API. When a call on this API comes from a service interface, it is difficult to distinguish whether the call is authorized or not. The calling party could be the authorized "expenditure" service or the service composition software that the service runs on. The "islands_of_security" approach of traditional applications breaks down in a network-centric model.

Providing security with a global perspective for network-centric systems is still an immature area unlike security for traditional monolithic applications that have been extensively studied. If the end-to-end security solutions involve using additional data in the messages, it may impact other quality attributes like performance.

In this section, security for Web Services are discussed first as they have to be considered both for .NET and J2EE.

4.2.4.1 Web Services

The security model for Web Services is currently provided by specifications and standards from various organizations. Some of the important standards include WS Security (WSS) and Security Assertion Markup Language [OASIS 2005a] (SAML). WS-Security is a message security mechanism that uses XML Encryption and XML Digital Signature to secure web services messages sent over SOAP. The WS-Security specification defines the use of various security tokens including X.509 certificates, SAML assertions, and username/password tokens to authenticate and encrypt SOAP web services messages. This specification also defines an extensible, general-purpose mechanism for associating security tokens with message content, as well as how to encode binary security tokens, a framework for XML-based tokens, and how to include opaque encrypted keys. The SAML specification defines an XML-based mechanism for securing Business-to-Business (B2B) and Business-to-Consumer (B2C) e-commerce transactions. SAML defines an XML framework for exchanging authentication and authorization information. Like for the core Web Service specifications, the WS-I provides the Basic Security Profile (BSP)[WS-I 2005b].

4.2.4.2 .NET

In both J2EE and .NET, security mechanisms exist both at the transport level and application level.

4.2.4.2.1 Applications

.NET provides extensive support for the traditional security mechanisms. For web applications, ASP.NET provides Windows and Forms based authentication. A role based security mechanism can be used for components (Enterprise service components) that allows defining different access to components, interfaces and methods. Impersonation and delegation allow accessing resources with the same identity of the caller. The authentication level settings make it possible to encrypt the data that is sent across the network. .NET provides Code Access Security (CAS) to limit access to code and other resources. By employing permissions you can limit what users can access.

4.2.4.2.2 Web Services

The technology for building Web Services using .NET is ASMX 2.0. However, ASMX provides support for the Web Services standards specified in basic Profile and not the security stack. Support for Web Services security specifications is provided in the Web Services Extension (WSE) technology which is an add on to the .NET 2.0 framework. WSE 3.0 provides support for WS-Security [Skonnard 2006]. Thus, message level security for Web Services can be considered as part of the .NET framework.

4.2.4.3 Java EE

4.2.4.3.1 Applications

The Java EE security model also addresses authentication, authorization, delegation, and data integrity for the components that make up a Java EE environment. Java provides for security in two ways. The Java Cryptography Architecture and Java Cryptography Extension (JCA/JCE) provide for user authentication and authorization and signing of digital messages. Both the JCA and JCE are “provider based”. A provider implements a cryptographic service such as generating random numbers or random numbers or creating digital signatures. JCA forms the core of the Java security API. JCE provides other security services like Encryption/decryption of messages, Password-Based Encryption, Cipher, key Agreement and Message Authentication Code (MAC). Java Authentication and Authorization Services (JAAS) provide programmatic access control and user authorization similar to CAS in .NET. JAAS grants a set of the program’s features based on permissions and security policies.

4.2.4.3.2 Web Services

Message Security is not yet a part of the Java EE platform [Sun 2006a]. While sun provides support for WS-Security in its application server for Java EE called the “Sun Java System Application Server”. Sun’s Java Web Services Developer Pack (Java WSDP) also includes XML and Web Services Security (XWSS). However, since these are provided as proprietary enhancements to Sun products, and are not required to be provided by all the Java EE vendors, they cannot be considered as part of the Java EE standard.

4.2.4.4 CORBA

OMG provides a series of specifications for addressing CORBA security [OMG 2006a]. The main specification is the CORBA security service specification. CORBA implementations may come with a Security Service based on the specifications of the Object Management Group's standards. These standards define three levels of service in this context: Level 0 simply incorporates SSL (Secure Socket Layer). Level 1 is intended for applications that may need to be secure, but where the code itself need not be aware of security issues. In such a case, all security operations should be handled by the underlying object request broker (ORB). Level 2 supports other advanced security features, and the application is likely to be aware of these. There are plenty of variances between CORBA implementations that anyone choosing CORBA should consider carefully. For example, many implementations of CORBA do not contain a Security Service at all. Others may only implement part of the specification.

4.2.4.5 DCOM

The DCOM specification provides similar functionality to CORBA even though it looks completely different. Authentication, data integrity, and secrecy are all wrapped up into a single property called the authentication level. Authentication levels only apply to server objects, and each object can have its own level set. Higher levels provide additional security, but at a greater cost. Authentication levels vary from 1 to 7, which each level building upon the capabilities of the previous level. Usually, a DCOM user chooses the authentication level on a per-application basis. The user may also set a default authentication level for a server, which will be applied to all applications on the machine for which specific authentication levels are not specified. DCOM also provides multiple levels of impersonation. COM+ provides role-based security to DCOM.

4.2.4.6 Jini

Jini Security features are quite similar to that of Java EE. It is based upon the twin notions of the principal and access control lists (ACLs). While the principal refers to a particular network user, access to a resource depends upon the contents of the ACL associated with that object. While these security features of Jini are enough for a trusted workgroup, problems arise when unknown clients are introduced to a Jini network of any size. Jini has no provision for data encryption or authentication beyond that provided by the standard capabilities of Java and RMI. Hasselmeyer et al. [2000] discuss Jini's dynamically downloaded proxies as a security concern as the client who downloads them does not know what the code of the proxy might be doing. The Jini Security Architecture by Sun's Davis project [jini.org 2006] tries to address that. The Jini Security Architecture mainly defines security as a deployment-time option. Using the new JSK (Jini Starter Kit) it is in fact possible to deploy an existing service in a secure way. In this respect then, Jini security is similar to Java EE security. In the Jini Security Framework both the client and the service provider can impose constraints on the service object (or proxy). For instance, once a service's proxy has been downloaded, it is possible to restrict which client (on the same device) can invoke which proxy's methods. Similarly, the client may impose certain constraints on the service provider such as that it authenticates and achieves integrity and confidentiality.

4.2.4.7 JXTA

Since JXTA is a set of protocols and infrastructure for building peer-to-peer applications, it makes more sense to discuss security in the context of PPA itself. The challenges in P2P computing are different from the ones in traditional client-server computing. The lack of a single information owner means that it is extremely difficult to establish a single security policy across an entire network, or implement traditional CSA security measures such as authentication, authorization, challenge/response, filtering, and logging. Some of the security problems associated with P2P networks includes poisoning [Daswani and Garcia-Molina 2004] and violation of privacy [Good and Krekelberg 2003]. However, the lack of central authority may also be advantageous sometimes as it can make Denial-of-Service attacks difficult. A malicious user cannot monitor the entire network by snooping on server communications as peer activity is usually limited to a small locality.

4.2.5 Dependability

4.2.5.1 .NET and Java EE

Both .NET and Java EE provide similar support for the traditional mechanisms to achieve dependability at the application level.

4.2.5.1.1 Transactions

.NET provides transactions to components through .NET Enterprise Services. Similarly, the Java EE EJB container provides built-in support for transactions.

4.2.5.1.2 State Management and Failover Clustering

When state is stored in a separate node or persisted in a database, it can be used for failover load balancing. Clustering helps to achieve both availability (through redundancy) and scalability. .NET and Java EE support for clustering is described under scalability.

ASP.NET allows several modes to store session state and enables on demand backing up of state to an independent node. State in Java EE applications can be in HttpSession (for Web Applications) or in stateful session beans. Java EE vendors provide failover clustering solutions at the HttpSession and EJB level. The biggest difference among the Java EE servers is support for automatic failover. Some vendor servers do not provide it, while others allow failover of stateful session beans by using in-memory state replication [Sun 2006c].

4.2.5.1.3 Support for Asynchronous Communication

.NET provides loosely coupled events and queued components. Java EE provides support for asynchronous communication with persistent JMS and message-driven beans.

4.2.5.2 CORBA

Fault Tolerant CORBA [OMG 2005] is the part of the CORBA 3.0 specification that can be used to provide fault-tolerance to CORBA objects. The Fault tolerant CORBA uses the entity redundancy paradigm (i.e. replication of objects) to provide fault tolerance to CORBA objects.

4.2.5.3 DCOM

DCOM’s reliability and consistency capabilities are mainly provided in conjunction with COM+ and MTS (Microsoft Transaction Server). DCOM also provides a “pinging mechanism” for fault tolerance at the protocol level [MSDN 1996b]. A basic idea is for client machines to keep sending “ping messages” periodically to a DCOM server object they are accessing. If the server object does not receive a message from a client for a specified period of time, that client is considered “dead”. This can be considered as a form of network failure detection mechanism.

4.2.5.4 Web Services

Dependability for Web Services can be considered from two perspectives: reliability of the messages sent between services over unreliable channels and the reliability of a service itself. A service that is architected in accordance with the SOA principles will be ideally stateless. Keeping this in mind and the fact that SOA includes dynamic discovery and composition, providing for fault tolerance at the service level might be much easier using replicated Web Services. If Web Services providing identical functionality are available, dealing with a crashed web server or web service node might be as simple as routing messages for them to an alternate service using dynamic discovery. WS-Management [Arora et al. 2004] might help in this regard.

Stateful Web Services, on the other hand, might pose problems as dealing with failover and will require state management and migration. This will need to be incorporated into the code.

Message reliability in Web Services is addressed using the WS-Reliability [OASIS 2004] which is an OASIS standard and WS-ReliableMessaging [IBM 2005] developed by IBM, BEA, Microsoft, and TIBCO Software. WS-Reliability aims to provide guaranteed delivery of messages, duplicate elimination and message order.

Other Web Services specifications related to dependability include WS-Transactions and WS-Coordination proposed by a consortium of companies led by Microsoft, IBM and BEA. However, these are not industry accepted standards as either the W3C nor OASIS has ratified them. WS-Transaction provides for the implementation of two different types of transactions, atomic and long running [BEA 2004]. WS-Transaction is built upon the WS-Coordination specification that provides protocols that coordinate the actions of distributed applications.

In terms of dynamic discovery, the location of the UDDI service is hardcoded into the application or service. This may be a less robust approach to the bootstrapping problem (i.e. in this case, finding a directory that facilitates service publishing and discovery) than dynamically discovering the directory using multicasting or other approaches.

4.2.5.5 Jini

In distributed systems, the most common failure scenario is the one in which some, but not all system components can be accessed. This partial failure can be the result of a host machine failure, a network partition, a software failure, or simple neglect (say, for example, one component decides to cease responding to another component). One of the distinguishing characteristics of the Jini framework is the concept of “lease” which is a useful mechanism to deal with partial failure scenarios. A Jini service is leased to a client in order to handle network failure and provide reliability. Each lease is negotiated between the service provider and service consumer as part of the network protocol and when a lease expires, a client must renew that lease in order to continue using that service.

A lease is a contract between a client and a server where a server grants a client privileges for a certain period of time. A lease can be considered as a failure detection mechanism in that the expiration of a lease that would have otherwise be expected to be renewed can be construed as a network or service failure on the part of the server [Bowers et al. 2003, Jai et al. 2000]. Similarly, if the server fails to respond to a renew request, the client detects that an error has occurred to the server. Usually, failure detection is achieved by monitoring a software entity or by the entity sending out heartbeats. The renewal of the lease can be considered as an heartbeat [Jai et al. 2000]. The rapidity of failure detection may be affected by the lease period [Bowers et al. 2003].

4.2.5.6 PPA Dependability

One of the hallmarks of PPA is that it is a decentralized architecture; applications and resources are replicated on the various nodes of the virtual network and the nodes collectively form the system without any central coordination. Because of this decentralization, there are no single points of failure.

Further, most PPA applications build an overlay network at the application layer. The graph-theoretic properties of this application layer overlay influence the routing efficiency and resilience to node failures of the network [Loguinov 2005]. Distributed Hash Tables (DHT) is an approach that has proved to be very efficient for PPA application networks in achieving properties such as resilience, performance and scalability [Chawathe et al. 2003]. DHTs are an approach to building PPA applications in which an abstract keyspace is partitioned among the participating nodes. The overlay network that connects these nodes can be used to find any node by using its key using hash table like semantics. Thus, DHTs help to locate resources in a PPA network more efficiently than using other approaches like flooding as fewer peers are visited and network communication is reduced. This approach has been proposed in structured P2P networks like Chord [Stoica et al. 2001], Pastry [Rowstron and Druschel 2001] and Tapestry [Zhao et al. 2001]. Another approach that can cut down on message overhead when trying to locate resources is a random walk [Lv et al. 2002].

4.2.5.6.1 JXTA

The JXTA platform provides a de-centralized environment that minimizes single-points of failure and is not dependent on any centralized services. Both centralized and de-centralized services can be developed on top of the JXTA platform.

In JXTA, all network resources such as peers, pipes, peergroups and services are represented by advertisements. JXTA uses a hybrid approach to provide for dependability and scalability. It combines a loosely consistent DHT with a limited range rendezvous walker to search for advertisements in a JXTA network [Traversat et al. 2003a].

JXTA advertisements are published with an expiration lifetime and they are purged from the caches of peers when it expires. This can be considered as a mechanism for dependability on the lines of Jini leases.

4.2.6 Scalability and Performance

4.2.6.1 .NET and Java EE

While clustering, load-balancing and failover fall outside the Java EE specification, the major implementations for application servers based on the Java EE specification provide suitable mechanisms for it. On the .NET side, Microsoft Application Center [Microsoft 2006a] provides support for load-balancing technology that enables a cluster of machines to collaborate and service user load that scales over time (scale-out). Both object clustering or clustering of whole deployments is possible as in Java EE.

.NET enterprise applications typically execute in the context of COM+ applications, which are typically used to provide automatic transaction and Just-in-Time activation (JITA) support. JITA helps to reduce load on a server. Java enterprise applications execute within a EJB container such as Websphere or Weblogic, which also provide automatic transaction and activation support.

The earlier perception that .NET does not scale vertically (by deployment on faster processors) as well as Java EE because of its dependence on the Windows operating system does not hold anymore. The advent of Windows Server 2003, has changed this. The 64 bit datacenter Edition of Windows Server 2003 [Microsoft 2002a] has the 3rd position in the TPC-C benchmark [TPC 2006]. The TPC-C benchmark models basic OLTP (OnLine Transaction Processing) functions used in OLTP environments. TPC-C as a measurement of scalability is important as a considerable number of enterprise applications are deployed in such environments. Thus, platform wise, Windows can scale as well as a UNIX or Solaris machine.

Scalability and performance can also be improved by reducing communication overhead. Both .NET and Java EE provide value objects and caching that can be used to achieve this. The connection with the database can become another major bottleneck. Once again, both .NET and Java EE provide database connection pooling that helps to improve this.

4.2.6.2 DCOM

For DCOM, platform scalability is the same as that for .NET as it is used primarily on Windows. DCOM achieves scalability by distributing objects onto different machines (location transparency), by providing support for symmetric multiprocessing for certain applications that use a free thread model and parallel deployment [MSDN 1996a, 1996b].

In DCOM, a client talks to a server component only through method calls. The client obtains the addresses of methods from a simple method address table called the "vtable" [MSDN 1996b]. If the method resides in a different process or machine, the DCOM RPC mechanism is used to make the call. This method is more efficient than using a component to intercept a client request as the overhead involved in sending out a call is reduced to a lookup in a vtable.

4.2.6.3 CORBA

The Portable Object Adapter (POA) can be considered as a mechanism for scalability for CORBA. POA is the piece of the ORB that manages server-side resources for scalability. By deactivating objects' servants when they have no work to do, and activating them again when they are needed, it helps to extend the same amount of hardware to service many more clients.

CORBA does not have a load balancing service [OMG 2006b] or other specific scalability services. Like in DCOM, multi-threading can be used to achieve a degree of scalability. The OMG [OMG 2006b] does suggest a load balancing mechanism that can be implemented based on the features of the GIOP, OMG's protocol for CORBA. But since issues like how it is implemented (in terms of interface provided) and whether it is implemented at all is dependent completely on the vendors, and no standardized interfaces are specified as part of the CORBA specifications to access this mechanism from clients. It cannot be considered as a core feature of CORBA.

An observation that can be made from the preceding sections on DCOM and CORBA scalability is that, in both frameworks, features for scalability are not transparently supported. Incorporating scalability into an application built on these frameworks requires extensive technical knowledge of the various client-server interactions involved.

4.2.6.4 Web Services

Scalability in the SOA context can be considered from several perspectives. From one point of view, scalability can be considered as the ability to accommodate increasing numbers of services and types of services. Other perspectives include the scalability of an individual service that is atomic (i.e. not composed of other services) and the scalability of an application or service that is created by federating a set of services.

When dynamic discovery is used, scalability, in the sense of being able to accommodate large numbers of services becomes the scalability of the approach that is used for dynamic discovery. In the case of a directory-based approach, it is then the scalability of the directory. Scalability of the discovery process having a single directory may become a bottleneck as the number of registered services increases exponentially. In Web Services, the discovery process using UDDI represents a centralized approach.

From the perspective of a single service, scalability depends on the technology used to implement it and is the same as for a single web application. The scalability scenario becomes more interesting when you consider an application incorporating a set of services from different providers. The overall scalability often becomes the salability of the least scalable service.

Scalability and performance is also affected by the communication protocols used. In Web Services, the communication protocol is SOAP which is verbose and text based. This may result in communication overhead leading to network congestion. The implementation of the soap stack used in creating a service may also have an impact on performance. The soap stack is responsible for providing libraries for parsing the soap messages received from various clients and performance and scalability depend on the speed of this parsing. Also, the encoding style used for soap may have an impact on the scalability [Cohen 2003]. Cohen [2003] describes an experiment for testing the performance of various SOAP stacks with the document style performing better than others.

Another factor that affects the performance of Web Services is the latency of the network over which they are accessed. This could be quite high for the Internet and is not deterministic.

Achieving scalability by load balancing might be easy as web services are stateless. Further, Web Services support the asynchronous communication model which can be exploited to enhance the performance of a services based application.

4.2.6.5 Jini

Sollins [2003] postulates that it is possible to create very large scale networks by using a grouping and partitioning mechanism. These large scale networks are formed by the interconnection of smaller, autonomous networks called “regions” where a region can be considered as a group of networked-entities with a boundary. While Jini was envisioned for use mostly in LANs and small scale networks, large scale networks can be formed using the concept of Jini federations. A Jini federation is a group of Jini services and clients that come together to form a community. These federations can, in turn, link together to form a larger federation and so on to form a hierarchy. Thus, large Jini federations can be formed out of smaller federations. This is possible because a Jini lookup service can register itself in other federations acting as the interface for sharing its resources with other federation's clients.

4.2.6.6 JXTA

Scalability for JXTA is discussed in the context of P2P virtual networks as the overall scalability is influenced more by the characteristics of the P2P paradigm than anything else. Since the nodes in a P2P paradigm play both role of client and server, scalability in the P2P network may be in terms of the number of nodes that the virtual network can accommodate gracefully. Scalability may be affected by overhead of routing, locating and synchronizing.

Since a large class of P2P networks exist for the sharing of resources, scalability and performance for P2P can also be interpreted as the ability of the virtual network to handle sudden spikes in the demand for particular resources. Rubenstein and Sahu [2005] report that even simple P2P solutions are capable of naturally handling sudden spikes in demand gracefully without much adverse effect on time and performance. Like resilience, the performance of a PPA application network is dependent on the graph-theoretic properties of the application layer overlay.

The same grouping and partitioning behavior that can be achieved using Jini federations for scalability can also be achieved using JXTA peer groups. Peers in JXTA self organize into peer groups like services and clients in Jini self organize into federations. Thus, peergroups enable subdividing a JXTA network into "regions" which can be used as boundaries for propagation of discovery and search requests. JXTA rendezvous and relay peers can be used as bridges between JXTA PPA networks.

4.3 Comparative Assessment Summary

This section provides a summary of the comparative analysis of the four architectures and seven frameworks based on architectural and quality characteristics provided in the previous sections of this chapter. Table 1 summarizes the similarities and differences between CSA, DOA, SOA and PPA. Table 2 summarizes the similarities and differences between the .NET framework and Java EE. Table 3 summarizes the similarities and differences between CORBA and DCOM. Table 4 summarizes the similarities and differences between Web Services, Jini and JXTA.

Table 1 Summary of comparison between CSA, DOA/CBA, SOA and PPA

Comparison Criteria		**CSA**	**DOA/CBA**	**SOA**	**PPA**
Interface based definition/description		No separate interface based definition	Interfaces defined in object-oriented IDL	Document-like Service contracts	No concept of interface
Partitioning of application logic		Horizontal	Horizontal and vertical	Vertical	NA
Discoverability (static and dynamic)	**Discoverable Entity**	None	Objects	Services	Peers and resources
	Discoverable?	None	Directory lookup of object location	Services are dynamically discoverable	Peers are dynamically discoverable
Autonomy of software units		None	Fair	Excellent	Excellent
Composability of software units		None	Good	Excellent	None
Coupling		Very tight	Tight	Loose	Tight to Loose
Software units have state?		Yes	Depends on design. Both stateless and stateful components are possible	Ideally, does not have state	NA
Granularity of processing units		Objects/functions	Distributed objects, Components	Varies. Usually a service encapsulates an application	Monolithic application on a node
Distribution scope		Application	Enterprise/organization	Inter-organizational	Global
Assumptions about operational environment		Stable	Stable	Stable	Unstable, transient connections, "computing on the edge"
Openness		External documentation, design and policy	Expressiveness of Interfaces, external documentation	Expressiveness of Service Contracts, External Documentation	Openness of application level communication protocol, policy
Interaction mode		Blocking synchronous request/reply	Primarily synchronous, finegrained ORPC, asynchronous comm.. Possible using polling, callback and message queues	Coarse grained document-style message passing. Both synchronous and asynchronous semantics can be achieved	Aynchronous, symmetric messaging/ request/reply
Life cycle		NA	Objects have life cycles	No notion of object life cycle	NA

Table 2 .NET vs. Java EE

Comparison Criteria		.NET	Java EE
Interface based definition		Not enforced, possible using “interface” language construct	Enforced only if using EJB. Otherwise, optional using Java “interface” construct
Openness	Framework	Partially open	Open specifications
	support for building open applications	Interfaces, Custom attributes, metadata, support for WS-* standards, XML standards	Interfaces, Annotations, support for WS-* standards
Interoperability and Integration		Bridging, Web Services (support for Basic Profile 1.0)	Bridging, RMI-IIOP, Web Services (support for Basic Profile 1.0)
Adaptability		Strict layering not enforced, but possible .NET attributes and contexts Strong versioning mechanism Manifest metadata Reflection.Emit namespace	Strict layering enforced if using EJB or an Inversion-of-Control container Entity beans provide object-oriented abstraction for relational data Weak versioning mechanism JNDI, annotations, deployment descriptors
Dependability		Transactions ASP.NET state management Loosely coupled events, queued components	Transactions Vendor support for automatic failover varies JMS, MDB
Security	Applications	Support for traditional security mechanisms Role based security, impersonation and delegation, multiple authentication and authorization levels Programmatic security through CAS	Support for traditional security mechanisms using JCE Role based security, programmatic access control through JAAS
	Web Services	Support for WS-Security	Support for WS-Security not part of Java EE spec, proprietary standards provided.
Scalability and Performance		Support for clustering, load balancing and fail over using Microsoft Application Server Runs on Windows Server 2003 that scales both horizontally and vertically Value objects, Automatic transactions, JITA, caching, database connection pooling.	Vendor support in application server for clustering, load balancing and failover Runs on almost all operating systems Provides value objects, Automatic transactions, JITA, caching, database connection pooling.

Table 3 CORBA vs. DCOM

Comparison Criteria		**CORBA**	**DCOM**
Interfaces		Object-oriented CORBA IDL, Multiple inheritance implemented using inheritance	MIDL, Multiple interfaces implemented via aggregation
Object		Uses references to identify objects	Only access to the interfaces of objects is possible
Client communication		Communication with the server object is through an ORB intermediary	Access to the server object can use RPC
Openness	Framework	Open specifications maintained by OMG	Specifications turned over to Open Group
	Support to build open applications	Expressiveness of IDL, open protocol	Expressiveness of MIDL, open protocol
Interoperability and Integration		Using ORB-to-ORB communication.	DCE-RPC protocol
Adaptability		Location transparency, DII Has problems with versioning	Location transparency, Strong Interface Versioning,
Security		CORBA security specification provides 3 levels of security Delegation of privileges Not much vendor support for CORBA security specifications	Multiple authentication levels and impersonation levels
Dependability		Fault tolerant CORBA using entity redundancy	Mechanisms provided by COM+ and MTS DCOM protocol's "pinging mechanism"
Scalability and Performance		Pros: Portable Object Adaptor Cons: no load balancing service Scalability and performance features not transparently supported	Symmetric multiprocessing, vtable, parallel deployment Scalability and performance features not transparently supported

Table 4 Web Services vs. Jini vs. JXTA

Comparison Criteria		**Web Services**	**Jini**	**JXTA**
Interface		WSDL	Java Interface	Not applicable
Openness	Framework	Web Services is a set of open and widely accepted standards	Jini specifications are openly available, licensing restrictions on developed code	Core protocols and specification open, openness of application
	Support for building open applications	Service contracts, open protocols	Service contracts are Java interfaces, RMI is the protocol used	JXTA does not influence this, matter of policy for openness of application protocols
Interoperability and Integration		Intrinsic interoperability	Java-centric	Depends on openness of application protocols
Adaptability		Dynamic discovery of services, contractual description, services are modular, dynamic reconfiguration easy No direct way for versioning	Dynamic discovery of services	Dynamic discovery of peers, resources
Security		Security with global perspective problematic, Message based security, policies. Standards: WS-Security, SAML, Basic Security Profile 1.0	Based on principles and ACLs, similar to Java EE Downloadable smart proxies can create problems	Single security policy difficult Pros: DoS attacks, snooping difficult Cons: Providing traditional CSA security mechanisms difficult
Dependability		Dynamic discovery, loose coupling, statelessness, WS-Reliability, WS-ReliableMessaging Cons: UDDI directory single point of failure	Dynamic discovery, partial decentralization through multiple Lookup services and direct discovery through multicast, Leases, distributed events	Ad hoc decentralized networks, redundancy – application and data replication, JXTA advertisements with expiration lifetime, JXTA forms a loosely consistent DHT with limited range random walker
Scalability and Performance		Pros: service statelessness, autonomy, loose coupling, coarse interfaces Cons: UDDI - performance bottleneck SOAP and XML parsing overhead, unpredictable network latency (esp. if Internet) overall scalability least common denominator of all services in an orchestration	Downloadable smart proxies, distributed directories, Jini communities cons: uses multicast for discovery	Influenced by graph-theoretic properties of the overlay network formed, routing algorithm of overlay network. JXTA peer groups, loosely consistent DHTs.

Chapter 5: Concluding Remarks

This thesis examined four mainstream architectures and seven popular associated frameworks. One trend that can be observed from the preceding comparative analysis of these architectures and frameworks is the need to consider architecture from a global perspective at a much higher level of abstraction than it is today to accommodate the unique needs of systems and System-of-Systems (SoS).The Service-Oriented Architecture (SOA) emerges as a natural candidate for architecting systems and SoS at this level. At the same time, it becomes evident that the Distributed-Objects Architecture (DOA) and Component Based Architecture (CBA) are more suitable for building single applications or to implement services. The rise of SOA does not render these architectures obsolete as SOA is an evolutionary architecture that rose in response to requirements that could only be solved at a higher level of abstraction than was possible to achieve with CSA, DOA or CBA. Many of the concepts found in peer-to-peer architecture are also very relevant to building network-centric systems. Both SOA and PPA view a system as a collection of cooperating nodes- they both take the focus away from the internals of a node and deal with it as a blackbox. If the PPA architectural constraint that each node in a PPA network is the same computationally is removed, the difference between SOA and PPA begins to blur.

Another trend that can be observed is that the two major platforms, .NET and Java EE are being cross pollinated with features and ideas from each other and are almost equally powerful. What can be achieved in one, can be achieved in the other, with perhaps different degrees of ease, cost, and methods.

This thesis presented a conceptual framework consisting of two sets of criteria- architectural and quality-based, for comparing network-centric software architectures and frameworks. Using this conceptual framework, a number of unique features, constraints and mechanisms that contribute towards realizing desirable characteristics of network-centric systems in each architecture and framework were identified and characterized. For example, Jini has leasing mechanisms for dependability; JXTA has an efficient method for resource location in a transient environment and so on. This characterization helps to understand the tradeoffs involved in using a particular architecture or framework to build a network-centric system.

The conceptual framework used for evaluating the architectures and frameworks could be further refined and extended. Future research may also involve evaluating academic and other domain specific architectures with respect to network-centricity. Further, a new architecture could be created by synthesizing these features or extending current architectures with useful features from the others.

Bibliography

ACC (2006), “Terrms and Definitions,” Acquisition Community Connection,” https://acc.dau.mil/simplify/ev.php?ID=93327_201&ID2=DO_TOPIC

Acton, M. (2003), “Designing Highly Available Web-Based Software Systems,” *CrossTalk 16*, 8, 4-8.

Adler, R.M. (1995), "Distributed Coordination Models for Client/Server Computing," *Computer* 28, 4, 14-22.

Anand, S., Padmanabhuni, S. and Ganesh, J. (2005), "Perspectives on Service Oriented Architecture," In *Proceedings of the IEEE International Conference on Services Computing, IEEE computer society press, Los Alamitos, CA*, pp. xvii.

Androutsellis-Theotokis, S. and Spinellis, D. (2004), “A Survey of Peer-to-Peer Content Distribution Technologies,*” ACM Computing Survey 36*, 4, 335-371.

Apache (2004), “Apache License, Version 2.0,” http://www.apache.org/licenses/LICENSE-2.0.html

Arora, A., Geller, A., He, J., Kaler, C., McCollum, R., Milenkovic, M., et al. (2004), “Web Services for Management,” http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-management1004.pdf

Barbacci, M., R., Ellison, R., J., Weinstock, C., B. and Wood W. G. (2000), “Quality Attribute Workshop Participants Handbook,” Special report, Software Engineering Institute, CMU/SEI-2000-SR-001.

BEA (2004), “Web Services Transaction (WS-Transaction),” http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html

Belloir, N., Bruel, J. and Barbier, F. (2003), “Whole-Part Relationships for Software Component Combination,” In *Proceedings of the 29th Conference on EUROMICRO*, IEEE Computer Society, Washington, DC, pp. 86.

BitTorrent (2006), “BitTorrent homepage,” http://www.bittorrent.com/

Boeing (2005), "Strategic Architecture Reference Model", The Boeing Company, Anaheim, CA, http://www.boeing.com/ids/stratarch/docs/sarm.pdf

Bowers, K., Mills, K., and Rose, S. (2003), “Self-Adaptive Leasing for Jini,” *In Proceedings of the First IEEE international Conference on Pervasive Computing and Communications*, IEEE Computer Society, Washington, DC, pp. 539.

Box, D. and Pattison, T. (2002), *Essential .Net: the Common Language Runtime*, Addison-Wesley Longman Publishing Co., Inc.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996), *Pattern-Oriented Software Architecture: a System of Patterns*, John Wiley & Sons, Inc.

Casagni, M. and Lyell, M. (2003), “Comparison of two component frameworks: the FIPA-compliant multi-agent system and the web-centric J2EE platform,” In *Proceedings of the 25th international Conference on Software Engineering*, IEEE Computer Society, Washington, DC, pp. 341-351.

Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N. and Shenker, S. (2003), “Making Gnutella-like P2P Systems Scalable,” *In Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications*, ACM Press, New York, NY, pp. 407-418.

Chung, J. (2005), "An Industry View on Service-Oriented Architecture and Web Services," in *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering*, IEEE computer society, Washington, DC, pp. 59.

Chung J., Lin K. and Mathieu, R.G. (2003), "Web Services Computing: Advancing Software Interoperability," *Computer 36*, 10, 35- 37.

Clayton, C. W. (2000), “Scalability with the Use of Object Pooling in an E-Commerce Environment,” Microsoft Developer Network, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomser/html/scalobjpool.asp

Cohen C. (2003), “Discover SOAP Encoding's Impact on Web Service Performance,” http://www-128.ibm.com/developerworks/webservices/library/ws-soapenc/

Cook, S.C. (2001), “On the Acquisition of Systems of Systems,” http://www.unisa.edu.au/seec/pubs/01papers/SoS%20Acquisition.PDF

Cotroneo, D., Graziano, A. and Russo, S. (2004), “Security Requirements in Service Oriented Architectures for Ubiquitous Computing,” In *Proceedings of the 2nd Workshop on Middleware For Pervasive and Ad-Hoc Computing*, ACM Press, New York, NY, pp. 172-177.

Bailes, J. E. and Templeton, G. F. (2004), “Managing P2P security,” *Communications of the ACM 47*, 9, 95-98.

Bondi, A. B. (2000), “Characteristics of Scalability and their Impact on Performance,” In *Proceedings of the 2nd international Workshop on Software and Performance*, ACM Press, New York, NY, pp. 195-203.

Dabrowski, C. and Mills, K. (2002), “Understanding Self-healing in Service-Discovery Systems,” In *Proceedings of the First Workshop on Self-Healing Systems*, ACM Press, New York, NY, pp. 15-20.

Dashofy, E. M., Medvidovic, N. and Taylor, R. N. (1999), “Using Off-the-shelf Middleware to Implement Connectors in Distributed Software Architectures,” In *Proceedings of the 21st international Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 3-12.

Daswani, N. and Garcia-Molina, H. (2004), “Pong-cache Poisoning in GUESS,” In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ACM Press, New York, NY, pp. 98-109.

DoD (2006), “Horizontal Fusion: FAQs,” http://horizontalfusion.dtic.mil/faq/

DoD (2006a), “Network-Centric Warfare: Department of Defense Report to Congress,” Department of Defense, http://www.defenselink.mil/nii/NCW/

DODAF (2004), “DoD Architecture Framework Version 1.0,” http://www.defenselink.mil/nii/doc/DoDAF_v1_Volume_I.pdf

Dogac, A., Dengi, C., and Öszu, M. T. (1998), “Distributed Object Computing Platforms,” *Communications of the ACM 41*, 9, 95-103.

ECMA International (2005), “Standard ECMA-335 -- Common Language Infrastructure (CLI),” Third Edition, Technical report ECMA-335, http://www.ecma-international.org/publications/standards/Ecma-335.htm

Eddon, G. and Eddon, H. (1998), “Understanding the DCOM Wire Protocol by Analyzing Network Data Packets,” Microsoft Systems Journal, http://www.microsoft.com/msj/0398/dcom.aspx

Ellison, R. J., Moore, A. P., Bass, L., Klein, M. and Bachmann, F. (2004), “Security and Survivability Reasoning Frameworks and Architectural Design Tactics,” http://www.sei.cmu.edu/publications/documents/04.reports/04tn022.html

Emmerich, W. and Kaveh, N. (2001), “Component Technologies: Java Beans, COM, CORBA, RMI, EJB and the CORBA Component Model,” In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT international Symposium on Foundations of Software Engineering*, ACM Press, New York, NY, pp. 311-312.

Erl, T. (2005), “A Look Ahead to the Service-Oriented World: Defining SOA When There’s No Single, Official Definition,” http://weblogic.sys-con.com/read/48928.htm

Esposito, D. (2002), *Building Web Solutions with ASP.NET and ADO.NET*, Microsoft Press Redmond, WA, USA.

Evans, M. W. and Marciniak, J. (1987), *Software Quality Assurance and Management*,John Wiley & Sons, Inc, New York, NY.

Fayad, M. and Schmidt, D. C. (1997), “Object-Oriented Application Frameworks,” *Communications of the ACM 40*, 10, 32-38.

Forster, F., and De Meer, H. (2004), “Discovery of Web Services With a P2P Network,” In *Proceedings of the 4th International Conference on Computational Science (ICCS) 2004*, Springer-Verlag, Berlin, pp. 90-97.

Fowler, M. (2002), *Patterns of Enterprise Application Architecture*, Addison-Wesley Longman Publishing Co., Inc.

Fuzak, C., Carper, W. L., Gmitruk, M., Aitkenhead, J. W., Mattoon, T. and Monteleon, V. J. (2001), “C4ISR Imperatives -- Cornerstones of a Network-Centric Architecture,”
http://stinet.dtic.mil/cgi-bin/GetTRDoc?AD=ADA434121&Location=U2&doc=GetTRDoc.pdf

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Longman Publishing Co., Inc.

Garlan, D., (2000), “Software Architecture: a Roadmap,” In *Proceedings of the Conference on the Future of Software Engineering ICSE '00*, ACM Press, New York, NY, pp. 91-101.

Gnutella (2001), “Gnutella Homepage” http://www.gnutella.com/

Good, N. S. and Krekelberg, A. (2003), “Usability and Privacy: a Study of Kazaa P2P File-sharing,” In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press, New York, NY, pp. 137-144

Govindaraju, M., Slominski, A., Choppella, V., Bramley, R., and Gannon, D. (2000), “Requirements for and Evaluation of RMI protocols for Scientific Computing,” *In Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, IEEE Computer Society, Washington, DC, pp. 61.

Gray, C. and Cheriton, D. (1989), “Leases: an Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency,” In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, ACM Press, New York, NY, pp. 202-210.

Groove Networks (2005), “Groove Virtual Office at a Glance: Architecture,” http://www.groove.net/pdf/gaag/gaag-architecture.pdf

Halepovic, E. and Deters, R. (2002), “Building a P2P Forum System with JXTA,” In *Proceedings of the Second international Conference on Peer-To-Peer Computing*, IEEE Computer Society, Washington, DC, p. 41.

Hasselmeyer, P., Kehr, R. and Vob, M. (2000), “Tradeoffs in a Secure Jini Service Architecture,“ In *Proceedings of the 3rd IFIP/GI International Conference on Trends towards a Universal Service Market,* Springer-Verlag, London, UK, pp. 190-201.

Hinton, H. M. (1997), “Under-specification, Composition and Emergent Properties,” In *Proceedings of the 1997 Workshop on New Security*, ACM Press, New York, NY, pp. 83-93.

Hohpe, G. and Woolf, B. (2004), *Enterprise Integration patterns: designing, building and deploying messaging solutions*, Addison-Wesley

IBM (2006), “IBM Homepage,” http://www.ibm.com/us/

IBM (2005), “Web Services Reliable Messaging,” http://www-128.ibm.com/developerworks/library/specification/ws-rm/

IEEE (1990), “IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries,” Institute of Electrical and Electronics Engineers, New York, NY.

Ironside, E., Etzkorn, L., and Zajac, D. (2001), “Examining CORBA Interoperability,” *Dr. Dobb's J. 26*, 6, 111-122.

Jai, B., Ogg, M. and Ricciardi, A. (2000), “Effortless Software Interoperability with Jini Connection Technology,” *Bell Labs Technical Journal 5*, 2, 88-101

JCP (2005a), “JSR 224: Java API for XML-Based Web Services (JAX-WS) 2.0,” Java Community Process, http://jcp.org/aboutJava/communityprocess/pfd/jsr224/index.html

Jini (2006), “Jini Standards,” http://www.jini.org/standards/

jini.org (2006), “Davis Project Home,” http://davis.jini.org/

JNBridge LLC. (2006), “JNBridgePro: High Performance Java/.NET Interoperability,” http://www.jnbridge.com/jnbpro.htm

Johnson, R., Hoeller, J., Arendsen, A., Risberg, T., and Kopylenko, D. (2005), *Professional Java Development with the Spring Framework*, Wrox Press Ltd.

Johnson, R. (2005), “J2EE Development Frameworks,” *Computer 38*, 1, 107-110.

JXTA (2006a),”Project JXTA,” http://www.jxta.org/

JXTA (2006b), “General JXTA™ FAQ,” http://www.jxta.org/JXTAFAQ.html#whatisProjectJXTA

Kazaa (2005), “Kazaa homepage,” http://www.kazaa.com/us/index.htm

Krastev, A. and Galletly, J. (2003), "Do We Really Need EJB?" In *Proceedings of the 4th international Conference on Computer Systems and Technologies: E-Learning*, ACM Press, New York, NY, pp. 190-195.

Krieger, D. and Adler, R.M. (1998), "The Emergence of Distributed Component Platforms," *Computer 31*, 3, 43-53.

Laprie, J., C., Avizienis, A. and Randell, B. (2000), "Fundamental Concepts of Dependability," UCLA CSD Report #010028, UCLA, Los Angeles, CA

Lau, Y. (2004), "Service-Oriented Architecture and the C4ISR Framework," *CrossTalk 17*, 9, 11-14.

Ledru, P. (2002), "Smart proxies for Jini Services," *SIGPLAN Notes 37*, 4, 57-61.

Lewandowski, S. M. (1998), "Frameworks for Component-based Client/Server Computing," *ACM Computing Surveys 30*, 1, 3-27.

Logan, B. C. (2003). "Technical Reference Model for Network-centric Operations," *Crosstalk 16*, 8, 22-25

Loguinov, D., Casas, J. and Wang, X. (2005), "Graph-theoretic Analysis of Structured Peer-to-Peer Systems: Routing Distances and Fault Resilience," *IEEE/ACM Transactions on Networks 13*, 5, 1107-1120.

Lowy, J. (2003), *Programming .NET Components*, O'Reilly & Associates Inc, Sebastopol, CA, USA

LSDIS (2006), "MWSAF: METEOR-S Web Service Annotation Framework," Large Scale Distributed Information Systems, http://lsdis.cs.uga.edu/projects/meteor-s/mwsaf/

Lumpe, M. (2002), "On the Representation and Use of Metadata," http://www.cs.iastate.edu/~lumpe/WCL2002/Camera/Lumpe.pdf

Lv, Q., Cao, P., Cohen, E., Li, K., and Shenker, S. (2002), "Search and Replication in Unstructured Peer-to-Peer Networks," In *Proceedings of the 16th international Conference on Supercomputing*, ACM Press, New York, NY, pp. 84-95.

Maeir, M. W. (2006), "Architecting Principles for System-of-Systems," http://www.infoed.com/Open/PAPERS/systems.htm

Medvidovic, N. (2002), "On the Role of Middleware in Architecture-based Software Development," In *Proceedings of the 14th international Conference on Software Engineering and Knowledge Engineering*, ACM Press, New York, NY, pp. 299-306.

Meredith, L. G. and Bjorg, S. (2003), "Contracts and Types," *Communications of the ACM 46*, 10, 41-47.

Meyers, B. C., Levine, L., Morris, E., Place, P. and Plakosh D. (2004), "SOSI: System-of-Systems Interoperability," Software Engineering Institute, http://www.sei.cmu.edu/products/events/acquisition/2004-presentations/meyers/

Middsol (2006), "MiddCor.NET," http://www.middsol.de/MiddCor/doc/MiddCor.pdf

Microsoft (2006), "Microsoft .NET Homepage," http://www.microsoft.com/net/default.mspx

Microsoft (2006a), "Application Center Product Overview," http://www.microsoft.com/applicationcenter/evaluation/overview/default.mspx

Microsoft (2006b), "Microsoft Corporation Homepage," http://www.microsoft.com/

Microsoft (2005), "Understanding the Distributed Object Component Model (DCOM) Architecture," http://www.microsoft.com/ntserver/techresources/appserv/COM/dcom_architecture.asp

Microsoft (2004), "How to use ASP.NET Session State SQL Server Mode in a Failover Cluster," http://support.microsoft.com/default.aspx?scid=kb;en-us;323262

Microsoft (2004a), "Service Orientation and Its Role in Your Connected Systems Strategy," http://msdn.microsoft.com/architecture/default.aspx?pull=/library/en-us/dnbda/html/srorientwp.asp

Microsoft (2002a), "Overview of Windows Server 2003 R2, Datacenter Edition," http://www.microsoft.com/windowsserver2003/evaluation/overview/datacenter.mspx

Monarch, I. and Wessel, J. (2005), "Autonomy and Interoperability in System of Systems Requirements Development," in *Proceedings of the Fifth International Workshop on for requirements for High Assurance Systems*, Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, http://www.sei.cmu.edu/community/rhas-workshop/monarch.pdf

MSDN (2006), "MSDN Home Page," Microsoft Corporation, Redmond, WA, http://msdn1.microsoft.com/en-us/default.aspx

MSDN (2006a), "The System.Reflection Namespace," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemreflection.asp

MSDN (2006b), "The System.Reflection.Emit Namespace," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemreflectionemit.asp

MSDN (2006c), "Session-State Modes," http://msdn2.microsoft.com/en-us/library/ms178586(VS.80).aspx

MSDN (2006d), "Architectural Overview of XML in the .NET framework," http://msdn2.microsoft.com/en-us/library/hfkahe27(VS.80).aspx

MSDN (2006f), ".NET Remoting Overview," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconnetremotingoverview.asp

MSDN (2006j), "Web Services Enhancements," http://msdn.microsoft.com/webservices/webservices/building/wse/default.aspx

MSDN (1997a), "DCOM Architecture," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomarch.asp

MSDN (1996a), "Cariplo: Distributed Object Component Model," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomhb.asp

MSDN (1996b), "DCOM Technical Review," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomtec.asp

Nagel, C. (2005), *Enterprise Services with the .NET Framework : Developing Distributed Business Solutions with .NET Enterprise Services (Microsoft Net Development Series)*, Addison-Wesley Professional, Boston, MA

Nikander, P. (2000), "Fault Tolerance in Decentralized and Loosely Coupled Systems," In *Ericsson Conference on Software Engineering*, Stockholm, Sweden, http://www.tml.tkk.fi/~pnr/publications/Ecse2000.pdf

OASIS (2006), Organization for the Advancement of Structured Information Standards, http://www.oasis-open.org/home/index.php

OASIS (2004), "WS-Reliability 1.1," http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf

OMG (2006a) "OMG Security," Object Management Group, http://www.omg.org/technology/documents/formal/omg_security.htm#Security_Service

OMG (2006b), "ORB Basics," Object Management Group, http://www.omg.org/gettingstarted/orb_basics.htm#g:LoadBalancing

OMG (2005), "CORBA/IIOP Specification," Object Management Group, http://www.omg.org/technology/documents/formal/corba_iiop.htm

Open Group (2006), "The Open Group: Boundaryless information flow through Interoperability," http://www.opengroup.org/

Open Group (1997), "DCE 1.1: Remote Procedure Call – Transfer Syntax NDR," http://www.opengroup.org/onlinepubs/9629399/chap14.htm

Oracle (2004), "Building a Network-centric Warfare Architecture", Oracle Corporation, Redwood Shores, CA, http://www.oracle.com/industries/government/ncwwhitepaperr1.pdf

Orchard, D. (2004), "Achieving Loose Coupling," http://dev2dev.bea.com/pub/a/2004/02/orchard.html

Orfali, R., Harkey, D. and Edwards, J. (1999), *Client/Server Survival Guide*, 3rd Edition, John Wiley & Sons.

Ortin, F., Redondo, J., Vinuesa, L. and Cueva, J.M. (2005), "Adding Structural Reflection to the SSCLI," http://dotnet.zcu.cz/NET_2005/Papers%5CFull%5CA67-full.pdf

OSJTF (2006), "Open Systems Joint Task Force," http://www.acq.osd.mil/osjtf/

OUSDATL (2005), "System-of-Systems and Family-of-Systems Frequently Asked Questions," Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics, http://www.acq.osd.mil/dpap/Docs/FAQs-- SoS & FoS.doc

OWL-S (2004), "DAML Services," http://www.daml.org/services/owl-s/

Papazoglou, M. P. and Georgakopoulos, D. (2003), "Introduction," *Communications of the ACM 46*, 10, 24-28.

Patrick, P. (2005), "Impact of SOA on Enterprise Information Architectures," *In Proceedings of the 2005 ACM SIGMOD international Conference on Management of Data*, ACM Press, New York, NY, pp. 844-848.

Peltzer, D. (2004), *.NET and J2EE Interoperability*, McGraw-Hill/Osborne.

Perrey, R. and Lycett, M. (2003), "Service-Oriented Architecture," In *Proceedings of the Symposium on Applications and the Internet Workshops*, IEEE Computer Society, Washington, DC, pp. 116-119.

Pinzger, M., Oberleitner, J. and Gall, H. (2003), "Analyzing and Understanding Architectural Characteristics of COM+ Components," In *proceedings of the 11th IEEE International Workshop on Program Comprehension*, IEEE Computer Society Press, Washington, DC, pp. 54- 63.

Poddar, I. (2004), "Co-hosting multiple versions of J2EE applications," IBM WebSphere Developer Technical Journal, IBM, http://www-128.ibm.com/developerworks/websphere/techjournal/0405_poddar/0405_poddar.html

Prechelt, L. (2003) "The Co-evolution of a Hype and a Software Architecture: Experience of Component-Producing Large-Scale EJB Early Adopters," In *Proceedings of the 25th international Conference on Software Engineering*, IEEE Computer Society, Washington, DC, pp. 553-556.

Pressman, R. S. (2005), *Software Engineering: A Practitioner's Approach*, McGraw-Hill Professional, New York, NY, 6th Edition.

Reiss, S. P. (2005), "A Component Model for Internet-Scale Applications," In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, ACM Press, New York, NY, pp. 34-43.

Ripeanu, M. (2001), "Peer-to-Peer Architecture Case Study: Gnutella Network," In *proceedings of the First International Conference on Peer-to-Peer Computing*, IEEE Computer Society, Washington, DC, p. 0099.

Rowstron, A. I. and Druschel, P. (2001), "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," *In Proceedings of the IFIP/ACM international Conference on Distributed Systems Platforms Heidelberg*, Springer-Verlag, London, pp. 329-350.

Rubenstein, D. and Sahu, S. (2005), "Can Unstructured P2P Protocols Survive Flash Crowds?" *IEEE/ACM Transactions on Networks 13*, 3, 501-512.

Schmidt, M.-T, Hutchison, B., Lambros, P. and Phippen, R. (2005), "The Enterprise Service Bus: Making Service-Oriented Architecture Real," *IBM Systems Journal 44*, 4, 781-797.

SEI (2006a), "The Open System Approach at the SEI," Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, http://www.sei.cmu.edu/opensystems/

SEI (2006b), "Integrating the SoS", Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, http://www.sei.cmu.edu/isis/guide/engineering/integration.htm

SEI (2006c) "Component Frameworks", Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, http://www.sei.cmu.edu/isis/guide/engineering/integration.htm

SEI (2006d), "A Framework for Software Product Line Practice Version 4.2," Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, http://www.sei.cmu.edu/productlines/frame_report/softwareSI.htm

SEI (2006e),"The Open Systems Approach at the SEI," Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, http://www.sei.cmu.edu/opensystems/

SEI (2006f), "Open Systems Frequently Asked Questions (FAQ)," Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, http://www.sei.cmu.edu/opensystems/faq.html

SEI (2006g), "Three tier software architectures," Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, http://www.sei.cmu.edu/str/descriptions/threetier_body.html

SEI (2005), "Component Object Model (COM), DCOM, and Related Capabilities," Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, http://www.sei.cmu.edu/str/descriptions/com.html

SEI (2000), "Three Tier Software Architectures," Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, http://www.sei.cmu.edu/str/descriptions/threetier.html

SETI (2006), "SETI@home Homepage," http://setiathome.ssl.berkeley.edu/

Shrivastava, S., K. and Wheater, S., M. (1998), "Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications," In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, IEEE Computer Society, Washington, DC, pp.10-17

Singh, M. (2001), "Peering at Peer-to-Peer Computing," *IEEE Internet Computing 5*, 6, 4- 5.

Singh, I., Stearns, B., and Johnson, M. (2002), *Designing Enterprise Applications with the J2EE Platform*, Second Edition, Addison-Wesley Longman Publishing Co., Inc.

Skonnard, A. (2006), "Service Station: All about ASMX 2.0, WSE 2.0 and WCF," MSDN Magazine, http://msdn.microsoft.com/webservices/default.aspx?pull=/msdnmag/issues/06/01/servicestation/default.aspx

Skonnard, A. (2001)," XML in .NET: .NET Framework XML Classes and C# Offer Simple, Scalable Data Manipulation," MSDN Magazine, http://msdn.microsoft.com/msdnmag/issues/01/01/xml/

Sollins, K. R. (2003), "Designing for Scale and Differentiation," *In Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, ACM Press, New York, NY, pp. 267-276.

Spring (2006), "The Spring Framework," http://www.springframework.org/

SSC San Diego (2006), "SSC San Diego: Programs," http://www.spawar.navy.mil/sandiego/html/programs_body.html

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001), "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ACM Press, New York, NY, pp. 149-160.

Stuckenholz, A. (2005), "Component Evolution and Versioning State of the Art," *SIGSOFT Software Engineering Notes* 30, 1, 7.

Sun Microsystems (2006), "Java Platform, Enterprise Edition (Java EE) 5 Technologies," Sun Microsystems, Santa Clara, CA, http://java.sun.com/javaee/5/javatech.html

Sun Microsystems (2006a), "Using Message Security with Java EE," Sun Microsystems, Santa Clara, CA, http://java.sun.com/javaee/5/docs/tutorial/doc/Security-WebSvcs7.html

Sun Microsystems (2006b), "Jini Network Technology - Sun Community Source License (SCSL) Overview," Sun Microsystems, Santa Clara, CA, http://www.sun.com/software/jini/licensing/overview.xml

Sun (2006c), "High Availability for J2EE Platform-Based Applications," Sun Microsystems, Santa Clara, CA, http://java.sun.com/developer/technicalArticles/J2EE/applications/index.html

Sun (2006d), "Jini Specifications," Sun Microsystems, Santa Clara, CA, http://www.sun.com/software/jini/specs/

Sun (2005), "JXTA v2.3.x: Java Programmer's Guide," Sun Microsystems, Santa Clara, CA, http://www.jxta.org/docs/JxtaProgGuide_v2.3.pdf

Sun (2005a), "Java EE Tutorial," Sun Microsystems, Santa Clara, CA, http://java.sun.com/javaee/5/docs/tutorial/doc/JavaEETutorial.pdf

Sun (2004), "JXTA^{TM} Technology: Creating Connected Communities," Sun Microsystems, http://www.jxta.org/docs/JXTA-Exec-Brief.pdf

Sun (2004a), "Jini Architecture Specification," http://www.sun.com/software/jini/specs/jini1.2html/jiniTOC.html

Sun (2002a), "Portable Object Adaptor (POA)," http://java.sun.com/j2se/1.4.2/docs/guide/idl/POA.html

Szyperski, C. (2003), "Component Technology: What, Where, and How?" *In Proceedings of the 25th international Conference on Software Engineering*, IEEE Computer Society, Washington, DC, pp. 684-693.

Szyperski, C. (1998), *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, Reading MA, Harlow, England.

Tartanoglu, F., Issarny, V., Romanovsky, A. and Levy, N. (2003), "Dependability in the Web Services Architecture," in *Architecting Dependable Systems*, LNCS 2677, http://www-rocq.inria.fr/~tartanog/publi/wads/

Tichy, M. and Giese, H. (2004), "A Self-optimizing Run-Time Architecture for Configurable Dependability of Services," Lecture Notes in Computer Science, Springer Berlin/Heidelberg, Volume 3069, pp. 25-50.

TPC (2006), "Top Ten TPC-C by Performance," Transaction Processing Performance Council, http://www.tpc.org/tpcc/results/tpcc_perf_results.asp

Traversat, B., Arora, A., Abdelaziz, M., Duigou, M., Haywood, C., Hugly, J. , Pouyoul, E. and Yeager, B. (2003), "Project JXTA 2.0 Super-Peer Virtual Network," Sun Microsystems, http://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf

Traversat, B., Abdelaziz, M. and Pouyoul, E. (2003a), "Project JXTA: A Loosely-Consistent DHT Rendezvous Walker," http://www.jxta.org/docs/jxta-dht.pdf

Trowbridge, D., Roxburgh, U., Hohpe, G., Manolescu, D. and Nadhan, E. G. (2004), "Integration Patterns," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/intpatt.asp

UDDI (2004), "UDDI Version 3.0.2," http://uddi.org/pubs/uddi_v3.htm

Vogols, W. (2003), "Web Services Are Not Distributed Objects," *IEEE Internet Computing 7*, 6, 59-63.

W3C (2006), "Web Services Activity," http://www.w3.org/2002/ws/

W3C (2006a), "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," http://www.w3.org/TR/wsdl20/#component_model

W3C (2005a), "Web Service Semantics WSDL-S," http://www.w3.org/Submission/WSDL-S/

W3C (2004), "Web Services Architecture," http://www.w3.org/TR/ws-arch/

W3C (2003a), "SOAP Version 1.2 Part 1: Messaging Framework," http://www.w3.org/TR/soap12-part1/

Waldo, J. (1999), "The Jini Architecture for Network-Centric Computing," *Communications of the ACM 42*, 7, 76-82.

Wikipedia (2006), “Overlay Network,” http://en.wikipedia.org/wiki/Overlay_network

WS-I (2006), Web Services Interoperability Organization, http://www.ws-i.org/

WS-I (2004), “Basic Profile Version 1.1,” Web Services Interoperability Organization, http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html

Yang, Z. and Duddy, K. (1996), “CORBA: a Platform for Distributed Object Computing,” *ACM SIGOPS Operating Systems Review 30*, 2, 4-31

Zhao, B., Y., Kubiatowicz, J., D. and Joseph, A. D. (2001), “Tapestry: an Infrastructure for Fault-Tolerant Wide-Area Location and Routing”, Technical Report. UMI Order Number: CSD-01-1141, University of California at Berkeley, Berkeley, CA.

VITA

LIKHITA KRISHNAMURTHY

Education

M.S. in Computer Science, May 2006
Virginia Polytechnic Institute & State University (Virginia Tech), Blacksburg, VA

B.S. in Information Science and Engineering, July 2004
Vishveswaraiah Technological University, Belgaum, India

Computer Skills

Programming languages: C/C++, Java, C#, VB, Perl, ASP .NET, PHP,COBOL
Databases: MS SQL server 2000, Oracle, MS Access, MySQL
Operating systems: Windows XP, Windows server 2003, Linux
Software: Rational Rose, Rational RequisitePro, Rational SoDA

Work Experience

Summer Internship (SDET), Microsoft Corporation, 2005

Search Digger

Search Digger is a tool that lets you monitor the performance as seen by the end user of a large-scale distributed system like the MSN Search engine. It lets you run a set of queries against a search engine, collect data on how long each portion of the query execution took as moves through the various components in the back-end and to be able to graphically drill in to individual components to identify bottlenecks. Additionally, long-term statistics are kept about all aspects of system performance, providing a good baseline for analysis as well as anomaly and regression detection. The repository used was SQL Server 2000. Data was collected by a client written in C++ and stored in the SQL Server repository. The interface was a C# windows application that queried the database and displayed the statistics using XCeed, a graphics library.

MSN Search API Web Service

- Worked extensively with Web Services technologies like WSDL, SOAP and XML Schema as part of my work with the MSN Search API Web Service.
- Developed several sample applications for the API using the .NET platform and in Java using Apache Axis.
- Did functional, interoperability and conformance testing and wrote code to automate the testing process.

Graduate Research Assistant, Dept. of Agriculture and Life Sciences, Fall 2005

Conceptualized, designed and implemented the ASP .NET Web Application Cyber Sheep. Cyber Sheep makes it makes it possible for students to play a stochastic simulation game of sire referencing schemes written in Fortran over the Internet using a browser.

Internship at Dimension Cybertech, Bangalore, India. (2004)

Developed a reader in C to read objects files in the Common Object File Format (COFF) and populate appropriate data structures that could be later plugged into Simulators/Emulators of various vendors for the Texas Instruments' C6000 family of Digital signal processors.

Graduate Teaching Assistant, Intermediate Software Design and Engineering course, Virginia Tech, Spring 2005

Graduate Teaching Assistant, Network Architecture and Programming course, Virginia Tech, Spring 2006

Course Projects

EtanaBrowse, Virginia Tech, Fall 2004
Enhanced, tested, debugged the 5SGraph tool – a graphical modeling tool written in Java that enables users to build their own instance of a digital library specified in the 5SL language automatically

Outback Queue Manager, Outback steakhouse, Christiansburg
The Outback Queue Manager (OQM) is a waitlist system for managing in-person and Call-Ahead customers for the Outback Steakhouse restaurant in Christiansburg. Hosts/Hostesses on the floor will use OQM via electronic Tablet PCs to manage a single waitlist. The application will keep track of waiting customers, table availability, and wait times. This system increases customer satisfaction by reducing wait time, and it simplifies business operations by automating the management of the waitlist between multiple hosts/hostesses. Developed a prototype in VC# using MS Access as the database and ADO .NET

Courses

Information storage and retrieval, Internet Programming, File structures in C/C++, Analysis and design of algorithms, DBMS, UNIX system programming, Software engineering, Usability engineering, Operating systems, Distributed Computing, Computer networks, Data communication, Computer architecture, Management Information Systems, Artificial Intelligence, Pattern Recognition, Neural Networks, Object Oriented System Development, Network Architecture and Protocols, Verification and Validation, Software Architecture

Activities & Awards

- Ranked first in a class of 34 in all four years of undergraduate studies
- Selected as Microsoft student champ for Vivekananda Institute of Technology, Bangalore, India
- Ranked 5th in the Karnataka state (India) matriculation exams (an estimated 600,000 students took the exams)

Other Interests

- Drama – Acted in a play "Antigone" that was presented at the Virginia Tech Department of Arts
- Creative fiction writing and poetry, painting and music
- Trained in two Indian classical dances
- Philosophy