A Comparative Study on Software Architectural Styles for Network based Applications

Dipanwita Thakur
Banasthali University
Rajasthan, India

G.N. Purohit
Banasthali university
Rajasthan, India

ABSTRACT
Software architecture defines the components and the interaction in between the components of a system. It also defines how the components are interacting with each other, the dependency in between the components and the interface protocols used for communication. For a network-based application, system performance is based on network communication. Therefore, selection of the appropriate architectural style(s) for use in designing the software architecture can make the difference between success and failure in the deployment of a network-based application.
There are so many architectural styles available to represent different network-based application. According to the behavior of the application we have to choose the appropriate architectural style.
In this paper we have surveyed different architectural styles for Network-based application.

Keywords
Software architecture, software architectural style, network-based application

1. INTRODUCTION
Software architecture has been a focal point for software engineering research in the 1990s. Architecture has emerged as a crucial part of the design process. Choosing a right architectural style for a network based application needs the knowledge of communication and the type of the application.

1.1. Software Architecture
Software architecture gives us the significant decision about the organization of a software system. Software system architecture is a system of computational components and interactions among those components. Components are such things as clients and servers, databases, filters, and layers in a hierarchical system. Interactions among components at this level of design can be simple and familiar, e.g. procedure call and shared variable access.

The architecture not only define the structure and topology of the system, but it also gives the interaction in between the system requirements and elements of the constructed system, thereby providing some rational for the design decisions. At the architectural level, relevant system-level issues typically include properties, e.g. capacity, throughput, consistency, and component compatibility.

Software architecture is the set {Elements, Form, and Rationale}. Thus software architecture is a set of architectural elements that have a particular form. There are three different classes of architectural elements: processing elements, data elements and connecting elements. The processing elements are those components that supply the transformation on the data elements; the data elements are those that contain the information that is used and transformed; the connecting elements are the glue that holds the different pieces of the architecture together. For example, procedure calls, shared data, and messages are different examples of connecting elements that serve to "glue" architectural elements together, [1]

Architecture is the fundamental organization of a system, embodying in its components, their relationships to each other and the environment, and the principles governing its design and evolution.
The software architecture of deployed software is determined by those aspects which are the hardest to change.

1.1.1 Component
A software component is an architectural entity that (i) encapsulates a subset of the system's functionality and/or data, (ii) restricts access to that subset via an explicitly defined interface, and (iii) has explicitly defined dependencies on its required execution context.

1.1.2 Connector
A software connector is an architectural element, effecting and regulating interactions among components.

1.1.3 Configuration
An architectural configuration is a set of specific associations between the components and connectors of a software system's architecture.

1.2. Architectural Styles
A style defines a family of architectures that satisfy the constraints. Styles allow one to apply specialized design knowledge to a particular class of systems and to support that class of system design with style-specific tools, analysis, and implementations.

1.3. Network-based Application
A distributed system is one that looks to its users like an ordinary centralized system, but runs on multiple, independent CPUs. In contrast, network-based systems are those capable

of operation across a network, but not necessarily in a fashion that is transparent to the user, [2].

2. Architectural Styles for Network-based Applications

2.1 Pipe & Filter (PF)

In this style each component has a set of inputs and a set of outputs. A component known as filter reads data steams as its inputs and produces data streams as its outputs. This is usually accomplished by applying a local transformation to the input streams and computing incrementally, so the output begins before input is consumed, [3]. The filters are totally independent entities and do not share state with other filters.

The advantages of the pipe and filter style are as follows. First, they allow the designer to understand the overall input/output behavior of a system as a simple composition of the behaviors of the individual filters. Second, they support reuse: any two filters can be hooked together, provided they agree on the data which is being transmitted between them. Third, systems are easy to maintain and enhance: new filters can be added to existing systems and old filters can be replaced by improved once. Fourth, they permit certain kind of specialized analysis, such as throughput and deadlock analysis. Finally, they naturally support concurrent execution. Each filter can be implemented for a separate task and can be potentially executed in parallel with other filters, [3].

Disadvantages of the PF style are as follows. First, pipe-and-filter systems often lead to batch organization of processing. Although filters can process data incrementally, they are inherently independent, so the designer must think of each filter as providing a complete transformation of input data to output data. In particular, because of their transformational character pipe-and-filter systems are typically not good at handling interactive applications. Second, they may be hampered by having to maintain correspondence between two separate but related streams. Third, depending on the implementation, they may force a lowest common denominator on data transmission, resulting in added work for each filter to parse and unparsed its data. This, in turn, can lead both to loss of performance and to increase in complexity in writing the filters themselves.

2.1.1Uniform Pipe-and-Filter

An improved version of the pipe-filter style is obtained by adding the constraint that all filters must have the same interface. The Unix operating system is the primary example of this style. In the Unix operating system, filter processes have an interface consisting of one input data stream of characters and two output data streams of characters. A new application can be formed by independently developed filters which allows restricted interface. It is very simple to understand the working of a filter.

The disadvantage of the uniform interface is that it may reduce network performance if the data needs to be converted to or from its natural format.

2.2 Client-Server (CS)

It is very popular architecture for network-based applications. There is one server component which performs all the tasks requested by the client component by a connector. The server can reject the request and sends a response back to the client. A client is a triggering process and a server is a reactive process. A client component makes request and waits for a response from the server. The server waits for a request and after receiving the request it responds to that request. Server is a non-terminating process and may serve more than one client, [4].

So many constraints can be added with this client –server to produce a simple server component to make it scalable.

2.2.1 Layered System (LS) and Layered-Client-Server (LCS)

A layered system is organized hierarchically, each layer provids service to the layer above it and serving as a client to the layer below it. [3].

Layered systems have several desirable properties. First, they support design based on increasing levels of abstraction, by which a implementer can partition a complex problem into a sequence of incremental steps.

Second, they support enhancement and finally, they support reuse.

On the contrary there are so many disadvantages with the layered system. First, not all systems are easily structured in a layered fashion. Second, it is quite difficult to find right levels of abstraction.

Layered-Client-Server adds proxy and gateway component with the client-server style. Proxy server is nothing but a shared server for one or more than one client, which accepting the request and forwards them to the server component. A gateway component is a normal server to the client or proxy component which can forward the services to its inner-layer server.

Architecture based on layered-client-server are referred to as two-tiered, three-tiered, or multi-tiered architecture in the information systems literature, [5].

LCS is also a solution for managing identity in a large scale distributed system, where complete knowledge of all servers would be prohibitively expensive. Instead, servers are organized in layers in such a manner that rarely used services are handled by intermediaries rather then directly by each client, [4].

2.2.2 Client-Stateless-Server (CSS)

It is one of the variants of client-server style. After adding the constraint of no session state on server component in the client-server style, it becomes the Client-Stateless-Server style. Whenever client wants to request the server the client component has to provide all the necessary information to the server component to execute the request. No information is stored in the server component.

These improve the quality like visibility, scalability and reliability. But it increases the per-instance overhead.

2.2.3 Client-Cache-Stateless-Server (C$SS)

It is the variant of the Client-stateless-server and cache style by adding the cache components. In this a cache is inserted in between the server component and client Component. Request is received by the cache component first. It improves the efficiency and performance.

This style is used in Sun Microsystems' NFS, [6].

2.2.4 Layered -Client-Cache-Stateless-Server (LC$SS)

It is another variant of the layered-client-server style and client-cache-stateless-server style obtained by adding the proxy and/or gateway component. Its advantages and disadvantages are derived from the advantages and disadvantages of its parent styles.

This style is used in Internet domain name system i.e., DNS and the Hypertext transfer protocol i.e. HTTP.

2.2.5 Remote Session

It is one of the verities of the client-server style. In this style, one session is created in between the client and the server by which the use of client component should minimize compare to server component. In other words it minimizes the complexity or reuse of client component compare to server component.

This style is used in TELNET or FTP.

2.2.6 Remote Data Access (RDA)

The remote data access style, [5] is one of the varieties of the client-server style. It is used in database query. In this a client sends a database request in SQL format to a remote server. The remote server gives response to the query in a large data set which is further used by the client to perform any other operation, like joining of tables and then retrieving the result.

In this style, a huge amount of data size can be reduced on the server side without transmitting it across the network. It improves the efficiency and visibility. Client should know the same manipulation scheme of data as server. It decreases scalability and reliability.

2.3 Mobile Code

It enables code to be transmitted to a remote host for interpretation. This may be due to lack of local computing power, lack of resources, or due to large data set remotely located. In this code is treated as data, [7].

2.3.1 Virtual Machine (VM)

A virtual machine, sometimes called an abstract machine, is a collection of modules that together provide a cohesive set of services that other modules can use without knowing how those services are implemented. It increases portability.

2.3.2 Remote Evaluation (REV)

In remote evaluation, a component on the source host has the know-how but not the resources needed for performing a service. The component is transferred to the destination host, where it is executed using the available resources. The result of the execution is returned to the source host. In remote evaluation a software component is:

1. Redeployed at run time from a source host to a destination host.

2. Installed on the destination host, ensuring that the software system's architectural configuration and any architectural constraints are preserved.

3. Activated.

4. Executed to provide the desired service.

5. Possibly de-activated and de-installed.

2.3.3 Code-on-Demand (COD)

In code-on -demand, the needed resources are available locally, but the know-how is not. The local subsystem thus requests the components providing the know-how from the appropriate remote hosts.

From a software architectural perspective, code-on-demand requires the same steps as remote evaluation; the only difference is that the roles of the target and destination hosts are reversed.

2.3.4 Mobile Agent (MA)

If a component on a given host (i) has the know-how for providing some service, (ii) has some execution state, and (iii) has access to some, though not all, of the resources needed to provide that service, the component, along with its state and local resources, may migrate to the destination host, which may have the remaining resources needed for providing service. The component, along with its state, will be installed on the destination host and will access all of the needed resources to provide the service. Mobile agents are stateful software components.

2.4 Replication

2.4.1 Replicated Repository

In this style more than one process provides the same service which improves the accessibility and scalability. It improves the performance. The client has the illusion that there is only one server which provides the centralized service. Distributed file system is the example of this.

2.4.2 Cache ($)

It is another variety of the replicated repository. Cache is easy to implement. It improves the efficiency of the system.

2.5. Event-based Integration (EBI)

The event-based style is characterized by independent components communicating solely by sending events through

event-bus connectors. Components emit events to the event-bus, which then transmits them to every other component.

The event-based style is highly suited to strongly decoupled concurrent components, where at any given moment a component either may be creating information of potential interest to others or may be consuming information.

2.6 Some other Styles

2.6.1 C2

C2 style is the resulting style of layered & event based styles. It is originally developed to support graphical user interface applications, was found to be beneficial in a wide variety of applications-indeed more so outside the domain of GUIs that within. C2's primary role in this presentation is showing how elements of many styles may be judiciously combined to meet variety of needs.

The advantages of C2 style are as follows: (i) Substrate independent: ease in modifying the application to work with new platforms. (ii) Accommodating heterogeneity: enabling an application to be composed of components written in diverse programming languages and running on multiple, varying hardware platforms, communicating across a network. (iii) Support for product lines: ease of substituting one component for another to achieve similar but difficult applications. (iv) Ability to design in the model-view-controller style: but with very strong separation between the model and the user interface elements. (v) Support for network-distributed applications: wherein communication protocol details are kept out of the components and confined to connectors.

The contribution of C2 is combining selected simple styles into a coherent comprehensive approach.

2.6.2 Distributed Objects (DO)

The distributed objects style represents a combination and adaptation of several simple styles. This style is augmented with the client-server style to provide the notion of distributed objects, with access to those objects from, potentially, different processes executing on different computers. In this style, application functionality broken up into objects that can run on heterogeneous hosts and can be written in heterogeneous programming languages. Objects provide services to other objects through well-defined provide interfaces. Objects invoke methods across host, process, and language boundaries via remote procedure calls (RPCs), generally facilitated by middleware.

Distributed Objects is not an ideal style for every application. Drawbacks include for example, that components in a distributed objects style are required to explicitly specify provided interfaces, but not to specify required interfaces. Dependencies between objects may thus be deeply ingrained.

2.7. Representational State Transfer (REST)

REST describes the architectural style used to guide the development of the standard protocols that constitute the WWW architecture. REST, as a set of design choices, drew from a rich heritage of architectural principles and styles.

There are six REST principles, or RPs:

RP1: The key abstraction of information is a resource, named by an URL. Any information that can be named can be a resource presentation

RP2: The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes. The particular form of the representation can be negotiated between REST components.

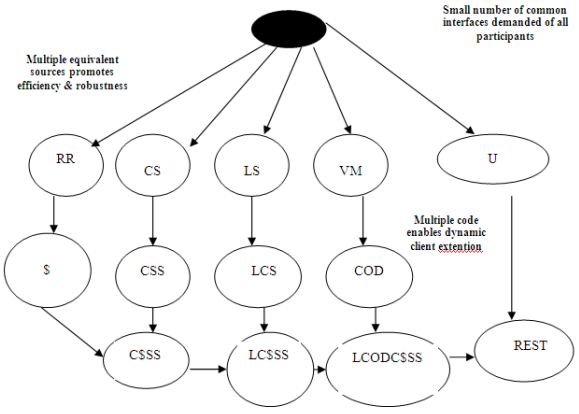
RP3: All interactions are context-free-each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it.

RP4: Components perform only a small set of well-defined methods on a resource producing a representation to capture the current or intended state of that resource and transfer that representation between components. These methods are global to the specific architectural instantiation of REST; for instance, all resources exposed via HTTP are expected to support each operation identically.

RP5: Idempotent operations and representation meta-data are encouraged in support of caching and representation reuse.

RP6: The presence of intermediaries is promoted. Filtering or redirection intermediaries may also use both the meta-data and the representations within request or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the user agent and the origin server.

Derivation of REST tree is as follows -



3. CONCLUSIONS

In this paper, we presented different architectural styles for network based applications. All the basic architectural styles and the derivative architectural styles from the basic one are discussed here. We compared all the architectural styles and discussed there advantages and disadvantages as well.

REST provides a model not only for the development and evaluation of new features, but also for the identification and understanding of broken features.

The World Wide Web is arguably the world's largest distributed application. Understanding the key architectural principles underlying the Web can help explain its technical success and may lead to improvements in other distributed applications, particularly those that are amenable to the same or similar methods of interaction. REST contributes both the rationale behind the modern Web's software architecture and a significant lesson in how software engineering principles can be systematically applied in the design and evaluation of a real software system.

After all the discussions we can conclude that REST is the most useful architectural style for any network-based application.

4. REFERENCES

[1] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes, 17(4), Oct. 1992, pp. 40-52

[2] A. S. Tanenbaum and R. van Renesse. Distributed Operating Systems. ACM Computing Surveys, 17(4), Dec. 1985, pp. 419-470.

[3] D. Garlan and M. Shaw. An introduction to software architecture. Ambriola & Tortola (eds.), Advances in Software Engineering & Knowledge Engineering, vol. II, World

[4] G. Andrews. Paradigms for process interaction in distributed programs. ACM Computing Surveys, 23(1), Mar. 1991, pp. 49-90.

[5] A. Umar. Object-Oriented Client/Server Internet Environments. Prentice Hall PTR, 1997.

[6] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In Proceedings of the Usenix Conference, June 1985, pp. 119-130.

[7] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. IEEE Transactions on Software Engineering, 24(5), May 1998, pp. 342-361.

[8] Roy Thomas. Fielding Architectural Styles and the Design of Networked-based Software Architectures. Ph.D. dissertation, Information and Computer Science, University of California-Irvine, Irvine, CA. 2000.