International Journal of Ethics in Engineering & Management Education
Website: www.ijeee.in (ISSN: 2348-4748, Volume 3, Issue 1, January 2016)

Architectural Styles and the Design of Network-based Software Architectures

Satyam Arragokula
Assistant Professor
University College of Science
Saifabad, O.U, Hyderabad, Ts
satya.anu11@gmail.com

M. Yesu Ratnam
Assistant Professor
Nizam College
Hyderabad, Ts, India
yesurathnammotamarry@gmail.com

Abstract: **The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an Internet-scale distributed hypermedia system. The Web has been iteratively developed over the past ten years through a series of modifications to the standards that define its architecture. In order to identify those aspects of the Web that needed improvement and avoid undesirable modifications, a model for the modern Web architecture was needed to guide its design, definition, and deployment. Software architecture research investigates methods for determining how best to partition a system, how components identify and communicate with each other, how information is communicated, how elements of a system can evolve independently, and how all of the above can be described using formal and informal notations. My work is motivated by the desire to understand and evaluate the architectural design of network- based application software through principled use of architectural constraints, thereby obtaining the functional, performance, and social properties desired of architecture. An architectural style is a named, coordinated set of architectural constraints. This dissertation defines a framework for understanding software architecture via architectural styles and demonstrates how styles can be used to guide the architectural design of network-based application software. A survey of architectural styles for network-based applications is used to classify styles according to the architectural properties they induce on architecture for distributed hypermedia. I then introduce the Representational State Transfer (REST) architectural style and describe how REST has been used to guide the design and development of the architecture for the modern Web.**

***Keywords*: architecture, Web, HTTP, WWW, Code, Network, HTML**

1. INTRODUCTION

As predicted by Perry and Wolf, software architecture has been a focal point for software engineering research in the 1990s. The complexity of modern software systems have necessitated a greater emphasis on componentized systems, where the implementation is partitioned into independent components that communicate to perform a desired task. Software architecture research investigates methods for determining how best to partition a system, how components identify and communicate with each other, how information is communicated, how elements of a system can evolve independently, and how all of the above can be described using formal and informal notations.

A good architecture is not created in a vacuum. All design decisions at the architectural level should be made within the context of the functional, behavioral, and social requirements of the system being designed, which is a principle that applies equally to both software architecture and the traditional field of building architecture. The guideline that “form follows function” comes from hundreds of years of experience with failed building projects, but is often ignored by software practitioners. The funny bit within the Monty Python sketch, cited above, is the absurd notion that an architect, when faced with the goal of designing an urban block of flats (apartments), would present a building design with all the components of a modern slaughterhouse. It might very well be the best slaughterhouse design ever conceived, but that would be of little comfort to the prospective tenants as they are whisked along hallways containing rotating knives.

The hyperbole of *The Architects Sketch* may seem ridiculous, but consider how often we see software projects begin with adoption of the latest fad in architectural design, and only later discover whether or not the system requirements call for such an architecture. Design-by-buzzword is a common occurrence. At least some of this behavior within the software industry is due to a lack of understanding of *why* a given set of architectural constraints is useful. In other words, the reasoning behind good software architectures is not apparent to designers when those architectures are selected for reuse.

This dissertation explores a junction on the frontiers of two research disciplines in computer science: software and networking. Software research has long been concerned with the categorization of software designs and the development of design methodologies, but has rarely been able to objectively evaluate the impact of various design choices on system behavior. Networking research, in contrast, is focused on the details of generic communication behavior between systems and improving the performance of particular communication techniques, often ignoring the fact that changing the interaction style of an application can have more impact on performance than the communication protocols used for that interaction. My work is motivated by the desire to understand and evaluate the architectural design of network-based application software through principled use of architectural constraints, thereby obtaining the functional, performance, and social properties desired of an architecture. When given a name, a coordinated set of architectural constraints becomes an architectural style.

The first three chapters of this dissertation define a framework for understanding software architecture via architectural styles, revealing how styles can be used to guide the architectural design of network-based application software. Common architectural styles are surveyed and classified according to the architectural properties they induce when applied to an architecture for network-based hypermedia. This classification is used to identify a set of architectural constraints that could be used to improve the architecture of the early World Wide Web.

Software Architecture: In spite of the interest in software architecture as a field of research, there is little agreement among researchers as to what exactly should be included in the definition of architecture. In many cases, this has led to important aspects of architectural design being overlooked by past research. This chapter defines a self-consistent terminology for software architecture based on an examination of existing definitions within the literature and my own insight with respect to network-based application architectures. Each definition, highlighted within a box for ease of reference, is followed by a discussion of how it is derived from, or compares to, related research.

Run-time Abstraction: Software architecture is an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture. At the heart of software architecture is the principle of abstraction: hiding some of the details of a system through encapsulation in order to better identify and sustain its properties. A complex system will contain many levels of abstraction, each with its own architecture. Architecture represents an abstraction of system behavior at that level, such that architectural elements are delineated by the abstract interfaces they provide to other elements at that level . Within each element may be found another architecture, defining the system of sub-elements that implement the behavior represented by the parent element's abstract interface. This recursion of architectures continues down to the most basic system elements: those that cannot be decomposed into less abstract elements.

Perry and Wolf define processing elements as "transformers of data," while Shaw et al. describe components as "the locus of computation and state." This is further clarified in Shaw and Clements "A component is a unit of software that performs some function at run-time. Examples include programs, objects, processes, and filters." This raises an important distinction between software architecture and what is typically referred to as software structure: the former is an abstraction of the run-time behavior of a software system, whereas the latter is a property of the static software source code. Although there are advantages to having the modular structure of the source code match the decomposition of behavior within a running system, there are also advantages to having independent software components be implemented using parts of the same code (e.g., shared libraries). We separate the view of software architecture from that of the source code in order to focus on the software's run-time characteristics independent of a given component's implementation. Therefore, architectural design and source code structural design, though closely related, are separate design activities. Unfortunately, some descriptions of software architecture fail to make this distinction (e.g., [9]).

Elements: Software architecture is defined by a configuration of architectural elements—components, connectors, and data—constrained in their relationships in order to achieve a desired set of architectural properties. A comprehensive examination of the scope and intellectual basis for software architecture can be found in Perry and Wolf. They present a model that defines a software architecture as a set of architectural *elements* that have a particular *form*, explicated by a set of *rationale.* Architectural elements include processing, data, and connecting elements. Form is defined by the properties of the elements and the relationships among the elements — that is, the constraints on the elements. The rationale provides the underlying basis for the architecture by capturing the motivation for the choice of architectural style, the choice of elements, and the form.

My definitions for software architecture are an elaborated version of those within the Perry and Wolf model, except that I exclude rationale. Although rationale is an important aspect of software architecture research and of architectural description in particular, including it within the definition of software architecture would imply that design documentation is part of the run-time system. The presence or absence of rationale can influence the evolution of an architecture, but, once constituted, the architecture is independent of its reasons for being. Reflective systems can use the characteristics of past performance to change future behavior, but in doing so they are replacing one lower- level architecture with another lower-level architecture, rather than encompassing rationale within those architectures.

Configurations: A configuration is the structure of architectural relationships among components, connectors, and data during a period of system run-time. Abowd et al. [1] define architectural description as supporting the description of systems in terms of three basic syntactic classes: components, which are the locus of computation; connectors, which define the interactions between components; and configurations, which are collections of interacting components and connectors. Various style-specific concrete notations may be used to represent these visually, facilitate the description of legal computations and interactions, and constrain the set of desirable systems.

Strictly speaking, one might think of a configuration as being equivalent to a set of specific constraints on component interaction. For example, Perry and Wolf include topology in their definition of architectural form relationships. However, separating the active topology from more general constraints allows an architect to more easily distinguish the active configuration from the potential domain of all legitimate configurations. Additional rationale for distinguishing configurations within architectural description languages is presented in Medvidovic and Taylor .

Properties: The set of architectural properties of a software architecture includes all properties that derive from the selection and arrangement of components, connectors, and data within the system. Examples include both the functional properties achieved by the system and non-functional properties, such as relative ease of evolution, reusability of components, efficiency, and dynamic extensibility, often referred to as quality attributes [9].

Properties are induced by the set of constraints within architecture. Constraints are often motivated by the application of a software engineering principle to an aspect of the architectural elements. For example, the *uniform pipe-and-filter* style obtains the qualities of reusability of components and configurability of the application by applying generality to its component interfaces — constraining the components to a single interface type. Hence, the architectural constraint is “uniform component interface,” motivated by the generality principle, in order to obtain two desirable qualities that will become the architectural properties of reusable and configurable components when that style is instantiated within an architecture.

The goal of architectural design is to create an architecture with a set of architectural properties that form a superset of the system requirements. The relative importance of the various architectural properties depends on the nature of the intended system. Section 2.3 examines the properties that are of particular interest to network-based application architectures.

Styles: An architectural style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style. Since an architecture embodies both functional and non-functional properties, it can be difficult to directly compare architectures for different types of systems, or for even the same type of system set in different environments. Styles are a mechanism for categorizing architectures and for defining their common characteristics. Each style provides an abstraction for the interactions of components, capturing the essence of a pattern of interaction by ignoring the incidental details of the rest of the architecture.

Perry and Wolf define architectural style as an abstraction of element types and formal aspects from various specific architectures, perhaps concentrating on only certain aspects of an architecture. An architectural style encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their relationships. This definition allows for styles that focus only on the connectors of an architecture, or on specific aspects of the component interfaces.

Patterns and Pattern Languages: In parallel with the software engineering research in architectural styles, the object- oriented programming community has been exploring the use of design patterns and pattern languages to describe recurring abstractions in object-based software development. A design pattern is defined as an important and recurring system construct. A pattern language is a system of patterns organized in a structure that guides the patterns’ application. Both concepts are based on the writings of Alexander et al with regard to building architecture.

The design space of patterns includes implementation concerns specific to the techniques of object-oriented programming, such as class inheritance and interface composition, as well as the higher-level design issues addressed by architectural styles. In some cases, architectural style descriptions have been recast as architectural patterns. However, a primary benefit of patterns is that they can describe relatively complex protocols of interactions between objects as a single abstraction, thus including both constraints on behavior and specifics of the implementation. In general, a pattern, or pattern language in the case of multiple integrated patterns, can be thought of as a recipe for implementing a desired set of interactions among objects. In other words, a pattern defines a process for solving a problem by following a path of design and implementation choices.

2. NETWORK-BASED APPLICATION ARCHITECTURES

This chapter continues our discussion of background material by focusing on network- based application architectures and describing how styles can be used to guide their architectural design.

Scope*:* Architecture is found at multiple levels within software systems. This dissertation examines the highest level of abstraction in software architecture, where the interactions among components are capable of being realized in network communication. We limit our discussion to styles for network-based application architectures in order to reduce the dimensions of variance among the styles studied.
Network-based vs. Distributed: The primary distinction between network-based architectures and software architectures in general is that communication between components is restricted to message passing [6], or the equivalent of message passing if a more efficient mechanism can be selected at runtime based on the location of components.

Tanenbaum and van Renesse make a distinction between distributed systems and network-based systems: 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. In some cases it is desirable for the user to be aware of the difference between an action that requires a network request and one that is satisfiable on their local system, particularly when network usage implies an extra transaction cost. This dissertation covers network-based systems by not limiting the candidate styles to those that preserve transparency for the user.

Application Software vs. Networking Software: Another restriction on the scope of this dissertation is that we limit our discussion to application architectures, excluding the operating system, networking software, and some architectural styles that would only use a network for system support (e.g., process control styles). Applications represent the "business-aware" functionality of a system.

Application software architecture is an abstraction level of an overall system, in which the goals of a user action are representable as functional architectural properties. For example, a hypermedia application must be concerned with the location of information pages, performing requests, and rendering data streams. This is in contrast to a networking abstraction, where the goal is to move bits from one location to another without regard to why those bits are being moved. It is only at the application level that we can evaluate design trade-offs based on the number of interactions per user action, the location of application state, the effective throughput of all data streams (as opposed to the potential throughput of a single data stream), the extent of communication being performed per user action, etc.

Evaluating the Design of Application Architectures: One of the goals of this dissertation is to provide design guidance for the task of selecting or creating the most appropriate architecture for a given application domain, keeping in mind that an architecture is the realization of an architectural design and not the design itself. An architecture can be evaluated by its run-time characteristics, but we would obviously prefer an evaluation mechanism that could be applied to the candidate architectural designs before having to implement all of them. Unfortunately, architectural designs are notoriously hard to evaluate and compare in an objective manner. Like most artifacts of creative design, architectures are normally presented as a completed work, as if the design simply sprung fully-formed from the architect's mind. In order to evaluate an architectural design, we need to examine the design rationale behind the constraints it places on a system, and compare the properties derived from those constraints to the target application's objectives.

The first level of evaluation is set by the application's functional requirements. For example, it makes no sense to evaluate the design of a process control architecture against the requirements of a distributed hypermedia system, since the comparison is moot if the architecture would not function. Although this will eliminate some candidates, in most cases there will remain many other architectural designs that are capable of meeting the application's functional needs. The remainder differ by their relative emphasis on the non-functional requirements—the degree to which each architecture would support the various non-functional architectural properties that have been identified as necessary for the system. Since properties are created by the application of architectural constraints, it is possible to evaluate and compare different architectural designs by identifying the constraints within each architecture, evaluating the set of properties induced by each constraint, and comparing the cumulative properties of the design to those properties required of the application.

As described in the previous chapter, an architectural style is a coordinated set of architectural constraints that has been given a name for ease of reference. Each architectural design decision can be seen as an application of a style. Since the addition of a constraint may derive a new style, we can think of the space of all possible architectural styles as a derivation tree, with its root being the null style (empty set of constraints). When their constraints do not conflict, styles can be combined to form hybrid styles, eventually culminating in a hybrid style that represents a complete abstraction of the architectural design. An architectural design can therefore be analyzed by breaking-down its set of constraints into a derivation tree and evaluating the cumulative effect of the constraints represented by that tree. If we understand the properties induced by each basic style, then traversing the derivation tree gives us an understanding of the overall design's architectural properties. The specific needs of an application can then be matched against the properties of the design. Comparison becomes a relatively simple matter of identifying which architectural design satisfies the most desired properties for that application.

Architectural Properties of Key Interest: This section describes the architectural properties used to differentiate and classify architectural styles in this dissertation. It is not intended to be a comprehensive list. I have included only those properties that are clearly influenced by the restricted set of styles surveyed. Additional properties, sometimes referred to as software qualities, are covered by most textbooks on software engineering. Bass et al examine qualities in regards to software architecture.

3. NETWORK-BASED ARCHITECTURAL STYLES

This chapter presents a survey of common architectural styles for network-based application software within a classification framework that evaluates each style according to the architectural properties it would induce if applied to architecture for a prototypical network-based hypermedia system.

Classification Methodology: The purpose of building software is not to create a specific topology of interactions or use a particular component type — it is to create a system that meets or exceeds the application needs. The architectural styles chosen for a system's design must conform to those needs, not the other way around. Therefore, in order to provide useful design guidance, a classification of architectural styles should be based on the architectural properties induced by those styles.

Pipe and Filter (PF): In a pipe and filter style, each component (filter) reads streams of data on its inputs and produces streams of data on its outputs, usually while applying a transformation to the input streams and processing them incrementally so that output begins before the input is completely consumed. This style is also referred to as a one-way data flow network [6]. The constraint is that a filter must be completely independent of other filters (zero coupling): it must not share state, control thread, or identity with the other filters on its upstream and downstream interfaces.

Abowd et al. [1] provide an extensive formal description of the pipe and filter style using the Z language. The *Khoros* software development environment for image processing provides a good example of using the pipe and filter style to build a range of applications.

Garlan and Shaw describe the advantageous properties of the pipe and filter style as follows. First, PF allows the designer to understand the overall input/output of the system as a simple composition of the behaviors of the individual filters (simplicity). Second, PF supports reuse: any two filters can be hooked together, provided they agree on the data that is being transmitted between them (reusability). Third, PF systems can be easily maintained and enhanced: new filters can be added to existing systems(extensibility) and old filters can be replaced by improved ones (evolvability). Fourth, they permit certain kinds of specialized analysis (verifiability), such as throughput and deadlock analysis. Finally, they naturally support concurrent execution (user-perceived performance).

Disadvantages of the PF style include: propagation delay is added through long pipelines, batch sequential processing occurs if a filter cannot incrementally process its inputs, and no interactivity is allowed. A filter cannot interact with its environment because it cannot know that any particular output stream shares a controller with any particular input stream. These properties decrease user-perceived performance if the problem being addressed does not fit the pattern of a data flow stream.

One aspect of PF styles that is rarely mentioned is that there is an implied "invisible hand" that arranges the configuration of filters in order to establish the overall application. A network of filters is typically arranged just prior to each activation, allowing the application to specify the configuration of filter components based on the task at hand and the nature of the data streams (configurability). This controller function is considered a separate operational phase of the system, and hence a separate architecture, even though one cannot exist without the other.

Mobile Code Styles: Mobile code styles use mobility in order to dynamically change the distance between the processing and source of data or destination of results. These styles are comprehensively examined in Fuggetta et al. A site abstraction is introduced at the architectural level, as part of the active configuration, in order to take into account the location of the different components. Introducing the concept of location makes it possible to model the cost of an interaction between components at the design level. In particular, an interaction between components that share the same location is considered to have negligible cost when compared to an interaction involving communication through the network. By changing its location, a component may improve the proximity and quality of its interaction, reducing interaction costs and thereby improving efficiency and user-perceived performance.

Limitations*:* Each architectural style promotes a certain type of interaction among components. When components are distributed across a wide-area network, use or misuse of the network drives application usability. By characterizing styles by their influence on architectural properties, and particularly on the network-based application performance of a distributed hypermedia system, we gain the ability to better choose a software design that is appropriate for the application. There are, however, a couple limitations with the chosen classification.

The first limitation is that the evaluation is specific to the needs of distributed hypermedia. For example, many of the good qualities of the pipe-and-filter style disappear if the communication is fine-grained control messages, and are not applicable at all if the communication requires user interactivity. Likewise, layered caching only adds to latency, without any benefit, if none of the responses to client requests are cacheable. This type of distinction does not appear in the classification, and is only addressed informally in the discussion of each style. I believe this limitation can be overcome by creating separate classification tables for each type of communication problem. Example problem areas would include, among others, large grain data retrieval, remote information monitoring, search, remote control systems, and distributed processing. A second limitation is with the grouping of architectural properties. In some cases, it is better to identify the specific aspects of, for example, understandability and verifiability induced by an architectural style, rather than lumping them together under the rubric of simplicity. This is particularly the case for styles which might improve verifiability at the expense of understandability. However, the more abstract notion of a property also has value as a single metric, since we do not want to make the classification so specific that no two styles impact the same category. One solution would be a classification that presented both the specific properties and a summary property.

Classification of Architectural Styles and Patterns: The area of research most directly related to this chapter is the identification and classification of architectural styles and architecture-level patterns. Shaw describes a few architectural styles, later expanded in Garlan and Shaw. A preliminary classification of these styles is presented in Shaw and Clements and repeated in Bass et al. [9], in which a two-dimensional, tabular classification strategy is used with control and data issues as the primary axes, organized by the following categories of features: which kinds of components and connectors are used in the style; how control is shared, allocated, and transferred among the components; how data is communicated through the system; how data and control interact; and, what type of reasoning is compatible with the style. The primary purpose of the taxonomy is to identify style characteristics, rather than to assist in their comparison. It concludes with a small set of "rules of thumb" as a form of design guidance

4. DESIGNING THE WEB ARCHITECTURE: PROBLEMS AND INSIGHTS

This chapter presents the requirements of the World Wide Web architecture and the problems faced in designing and evaluating proposed improvements to its key communication protocols. I use the insights garnered from the survey and classification of architectural styles for network-based hypermedia systems to hypothesize methods for developing an architectural style that would be used to guide the design of improvements for the modern Web architecture.

WWW *Application Domain Requirements*: Berners-Lee writes that the "Web's major goal was to be a shared information space through which people and machines could communicate." What was needed was a way for people to store and structure their own information, whether permanent or ephemeral in nature, such that it could be usable by themselves and others, and to be able to reference and structure the information stored by others so that it would not be necessary for everyone to keep and maintain local copies.

The intended end-users of this system were located around the world, at various university and government high-energy physics research labs connected via the Internet. Their machines were a heterogeneous collection of terminals, workstations, servers and supercomputers, requiring a hodge podge of operating system software and file formats. The information ranged from personal research notes to organizational phone listings. The challenge was to build a system that would provide a universally consistent interface to this structured information, available on as many platforms as possible, and incrementally deployable as new people and organizations joined the project. Problem Working groups within the Internet Engineering Taskforce were formed to work on the Web's three primary standards: URI, HTTP, and HTML. The charter of these groups was to define the subset of existing architectural communication that was commonly and consistently implemented in the early Web architecture, identify problems within that architecture, and then specify a set of standards to solve those problems. This presented us with a challenge: how do we introduce a new set of functionality to an architecture that is already widely deployed, and how do we ensure that its introduction does not adversely impact, or even destroy, the architectural properties that have enabled the Web to succeed.

Approach: The early Web architecture was based on solid principles—separation of concerns, simplicity, and generality—but lacked an architectural description and rationale. The design was based on a set of informal hypertext notes [14], two early papers oriented towards the user community [12, 13], and archived discussions on the Web developer community mailing list (www-talk@info.cern.ch). In reality, however, the only true description of the early Web architecture was found within the implementations of libwww (the CERN protocol library for clients and servers), Mosaic (the NCSA browser client), and an assortment of other implementations that interoperated with them.

An architectural style can be used to define the principles behind the Web architecture such that they are visible to future architects. As discussed in Chapter 1, a style is a named set of constraints on architectural elements that induces the set of properties desired of the architecture. The first step in my approach, therefore, is to identify the constraints placed

5. REPRESENTATIONAL STATE TRANSFER (REST)

This chapter introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles. REST is a hybrid style derived from several of the network-based architectural styles described in Chapter 3 and combined with additional constraints that define a uniform connector interface. The software architecture framework of Chapter 1 is used to define the architectural elements of REST and examine sample process, connector, and data views of prototypical architectures.

Deriving REST: The design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture. By examining the impact of each constraint as it is added to the evolving style, we can identify the properties induced by the Web's constraints. Additional constraints can then be applied to form a new architectural style that better reflects the desired properties of a modern Web architecture. This section provides a general overview of REST by walking through the process of deriving it as an architectural style. Later sections will describe in more detail the specific constraints that compose the REST style.

Starting with the Null Style: There are two common perspectives on the process of architectural design, whether it be for buildings or for software. The first is that a designer starts with nothing—a blank slate, whiteboard, or drawing board—and builds-up an architecture from familiar components until it satisfies the needs of the intended system. The second is that a designer starts with the system needs as a whole, without constraints, and then incrementally identifies and applies constraints to elements of the system in order to differentiate the design space and allow the forces that influence system behavior to flow naturally, in harmony with the system. Where the first emphasizes creativity and unbounded vision, the second emphasizes restraint and understanding of the system context. REST has been developed using the latter process. Figures through depict this graphically in terms of how the applied constraints would differentiate the process view of an architecture as the incremental set of constraints is applied.

The Null style (Figure 5-1) is simply an empty set of constraints. From an architectural perspective, the null style describes a system in which there are no distinguished boundaries between components. It is the starting point for our description of REST.

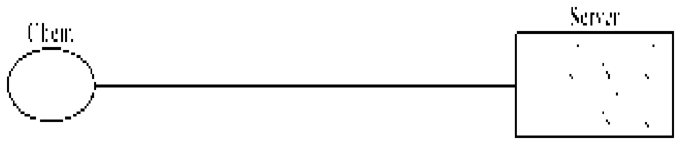

Fig.1.Client-Server

Client-Server: The first constraints added to our hybrid style are those of the client-server architectural style, described Separation of concerns is the principle behind the client-server constraints. By separating the user interface concerns from the data storage concerns, we improve the portability of

the user interface across multiple platforms and improve scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting Cache
In order to improve network efficiency, we add cache constraints to form the client-cache- stateless-server style of Section. Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or noncacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

The advantage of adding cache constraints is that they have the potential to partially or completely eliminate some interactions, improving efficiency, scalability, and user-perceived performance by reducing the average latency of a series of interactions. The trade-off, however, is that a cache can decrease reliability if stale data within the cache differs significantly from the data that would have been obtained had the request been sent directly to the server.

The early Web architecture, as portrayed by the diagram in Figure, was defined by the client-cache-stateless-server set of constraints. That is, the design rationale presented for the Web architecture prior to 1994 focused on stateless client-server interaction for the exchange of static documents over the Internet. The protocols for communicating interactions had rudimentary support for non-shared caches, but did not constrain the interface to a consistent set of semantics for all resources. Instead, the Web relied on the use of a common client-server implementation library (CERN libwww) to maintain consistency across Web applications.

Developers of Web implementations had already exceeded the early design. In addition to static documents, requests could identify services that dynamically generated responses, such as image-maps [Kevin Hughes] and server-side scripts [Rob McCool].Browsers

Work had also begun on intermediary components, in the form of proxies and shared caches, but extensions to the protocols were needed in order for them to communicate reliably. The following sections describe the constraints added

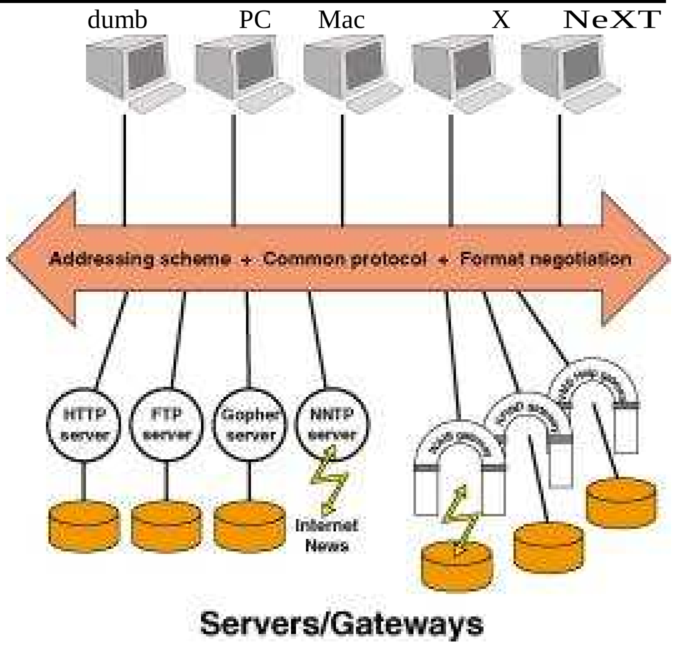

Fig.2. Early WWW Architecture Diagram

to the Web's architectural style in order to guide the extensions that form the modern Web architecture.

Uniform Interface: The central feature that distinguishes the REST architectural style from other network- based styles is its emphasis on a uniform interface between components. By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is **improved.** Implementations
The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs. The REST interface is designed to be efficient for large- grain hypermedia data transfer, optimizing for the common case of the Web, but resulting in an interface that is not optimal for other forms of architectural interaction.

REST Architectural Elements: The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. It encompasses the fundamental constraints upon components, connectors, and data that define the basis of the Web architecture, and thus the essence of its behavior as a network-based application.
REST Architectural Views: Now that we have an understanding of the REST architectural elements in isolation, we can use architectural views to describe how the elements work together to form an architecture. Three types of view—process, connector, and data—are useful for illuminating the design principles of REST.

6. EXPERIENCE AND EVALUATION

Since 1994, the REST architectural style has been used to guide the design and development of the architecture for the modern Web. This chapter describes the experience and lessons learned from applying REST while authoring the Internet standards for the Hypertext Transfer Protocol (HTTP) and Uniform Resource Identifiers (URI), the two specifications that define the generic interface used by all component interactions on the Web, as well as from the deployment of these technologies in the form of the libwww-perl client library, the Apache HTTP Server Project, and other implementations of the protocol standards.

Standardizing the Web: As described in Chapter 4, the motivation for developing REST was to create an architectural model for how the Web should work, such that it could serve as the guiding framework for the Web protocol standards. REST has been applied to describe the desired Web architecture, help identify existing problems, compare alternative solutions, and ensure that protocol extensions would not violate the core constraints that make the Web successful. This work was done as part of the Internet Engineering Taskforce (IETF) and World Wide Web Consortium (W3C) efforts to define the architectural standards for the Web: HTTP, URI, and HTML.

My involvement in the Web standards process began in late 1993, while developing the libwww-perl protocol library that served as the client connector interface for MOM spider. At the time, the Web's architecture was described by a set of informal hypertext notes, two early introductory papers, draft hypertext specifications representing proposed features for the Web (some of which had already been implemented), and the archive of the public www-talk mailing list that was used for informal discussion among the participants in the WWW project worldwide. Each of the specifications were significantly out of date when compared with Web implementations, mostly due to the rapid evolution of the Web after the introduction of the Mosaic graphical browser [NCSA]. Several experimental extensions had been added to HTTP to allow for proxies, but for the most part the protocol assumed a direct connection between the user agent and either an HTTP origin server or a gateway to legacy systems. There was no awareness within the architecture of caching, proxies, or spiders, even though implementations were readily available and running amok. Many other extensions were being proposed for inclusion in the next versions of the protocols.

REST Applied to URI: Uniform Resource Identifiers (URI) are both the simplest element of the Web architecture and the most important. URI have been known by many names: WWW addresses, Universal Document Identifiers, Universal Resource Identifiers, and finally the combination of Uniform Resource Locators (URL) and Names (URN). Aside from its name, the URI syntax has remained relatively unchanged since 1992. However, the specification of Web addresses also defines the scope and semantics of what we mean by *resource*, which has changed since the early Web architecture. REST was used to define the term resource for the URI standard, as well as the overall semantics of the generic interface for manipulating resources via their representations.

REST Applied to HTTP: The Hypertext Transfer Protocol (HTTP) has a special role in the Web architecture as both the primary application-level protocol for communication between Web components and the only protocol designed specifically for the transfer of resource representations. Unlike URI, there were a large number of changes needed in order for HTTP to support the modern Web architecture. The developers of HTTP implementations have been conservative in their adoption of proposed enhancements, and thus extensions needed to be proven and subjected to standards review before they could be deployed. REST was used to identify problems with the existing HTTP implementations, specify an interoperable subset of that protocol as HTTP/1.0 [19], analyze proposed extensions for HTTP/1.1, and provide motivating rationale for deploying HTTP/1.1.

The key problem areas in HTTP that were identified by REST included planning for the deployment of new protocol versions, separating message parsing from HTTP semantics and the underlying transport layer (TCP), distinguishing between authoritative and non-authoritative responses, fine-grained control of caching, and various aspects ofthe protocol that failed to be self-descriptive. REST has also been used to model the performance of Web applications based on HTTP and anticipate the impact of such extensions as persistent connections and content negotiation. Finally, REST has been used to limit the scope of standardized HTTP extensions to those that fit within the architectural model, rather than allowing the applications that misuse HTTP to equally influence the standard.

Technology Transfer: Although REST had its most direct influence over the authoring of Web standards, validation of its use as an architectural design model came through the deployment of the standards in the form of commercial-grade implementations. My involvement in the definition of Web standards began with development of the maintenance robot MOMspider and its associated protocol library, libwww-perl. Modeled after the original libwww developed by Tim Berners-Lee and the WWW project at CERN, libwww-perl provided a uniform interface for making Web requests and interpreting Web responses for client applications written in the Perl language [134]. It was the first Web protocol library to be developed independent of the original CERN project, reflecting a more modern interpretation of the Web interface than was present in older code bases. This interface became the basis for designing REST.

libwww-perl consisted of a single request interface that used Perl's self-evaluating code features to dynamically load the appropriate transport protocol package based on the scheme of the requested URI. For example, when asked to make a "GET" request on the URL <http://www.ebuilt.com/>, libwww-perl would extract the scheme from the URL ("http") and use it to construct a call to *wwwhttp'request()*, using an interface that was common to all types of resources (HTTP, FTP, WAIS, local files, etc.). In order to achieve this generic interface, the library treated all calls in much the same way as an HTTP proxy. It provided an interface using Perl data structures that had the same semantics as an HTTP request, regardless of the type of resource.

Architectural Lessons: There are a number of general architectural lessons to be learned from the modern Web architecture and the problems identified by REST.

Advantages of a Network-based API: What distinguishes the modern Web from other middleware is the way in which it uses HTTP as a network-based Application Programming Interface (API). This was not always the case. The early Web design made use of a library package, CERN libwww, as the single implementation library for all clients and servers. CERN libwww provided a library-based API for building interoperable Web components.

A library-based API provides a set of code entry points and associated symbol/ parameter sets so that a programmer can use someone else's code to do the dirty work of maintaining the actual interface between like systems, provided that the programmer obeys the architectural and language restrictions that come with that code. The assumption is that all sides of the communication use the same API, and therefore the internals of the interface are only important to the API developer and not the application developer.

The single library approach ended in 1993 because it did not match the social dynamics of the organizations involved in developing the Web. When the team at NCSA increased the pace of Web development with a much larger development team than had ever been present at CERN, the libwww source was "forked" (split into separately maintained code bases) so that the folks at NCSA would not have to wait for CERN to catch-up with their improvements. At the same time, independent developers such as myself began developing protocol libraries for languages and platforms not yet supported by the CERN code. The design of the Web had to shift from the development of a reference protocol library to the development of a network-based API, extending the desired semantics of the Web across multiple platforms and implementations.

CONCLUSION

At the beginning of our efforts within the Internet Engineering Taskforce to define the existing Hypertext Transfer Protocol (HTTP/1.0) [19] and design the extensions for the new standards of HTTP/1.1 [42] and Uniform Resource Identifiers (URI), I recognized the need for a model of how the World Wide Web *should* work. This idealized model of the interactions within an overall Web application, referred to as the Representational State Transfer (REST) architectural style, became the foundation for the modern Web architecture, providing the guiding principles by which flaws in the preexisting architecture could be identified and extensions validated prior to deployment.

REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations. This is achieved by placing constraints on connector semantics where other styles have focused on component semantics. REST enables the caching and reuse of interactions, dynamic substitutability of components, and processing of actions by intermediaries, thereby meeting the needs of an Internet-scale distributed hypermedia system.

REFERENCES

[1]. G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology,* 4(4), Oct. 1995, pp. 319-364.

[2]. A shorter version also appeared as: Using style to understand descriptions of software architecture. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'93),* Los Angeles, CA, Dec. 1993, pp. 9-20.

[3]. Adobe Systems Inc. *PostScript Language Reference Manual.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

[4]. C. Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[5]. C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language.* Oxford University Press, New York, 1977.

[6]. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology,* 6(3), July 1997.

[7]. A shorter version also appeared as: Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering,* Sorrento, Italy, May 1994, pp. 71-80.

[8]. Also as: Beyond Definition/Use: Architectural Interconnection. In *Proceedings of the ACM Interface Definition Language Workshop,* Portland, Oregon, *SIGPLAN Notices,* 29(8), Aug. 1994.

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

[10]. F. Anklesaria, et al. The Internet Gopher protocol (a distributed document search and retrieval protocol). *Internet RFC 1436,* Mar. 1993.

[11]. D. J. Barrett, L. A. Clarke, P. L. Tarr, A. E. Wise. A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology,* 5(4), Oct. 1996, pp. 378-421.

[12]. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison Wesley, Reading, Mass., 1998.

[13]. D. Batory, L. Coglianese, S. Shafer, and W. Tracz. The ADAGE avionics reference architecture. In *Proceedings of AIAA Computing in Aerospace 10,* San Antonio, 1995.

[14]. T. Berners-Lee, R. Cailliau, and J.-F. Groff. World Wide Web. Flyer distributed at the *3rd Joint European Networking Conference,* Innsbruck, Austria, May 1992.

[15]. T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann. World-Wide Web: The information universe. *Electronic Networking: Research, Applications and Policy,* 2(1), Meckler Publishing, Westport, CT, Spring 1992, pp. 52-58.

[16]. T. Berners-Lee and R. Cailliau. World-Wide Web. In *Proceedings of Computing in High Energy Physics 92,* Annecy, France, 23-27 Sep. 1992.

[17]. T. Berners-Lee, R. Cailliau, C. Barker, and J.-F. Groff. W3 Project: Assorted design notes. Published on the Web, Nov. 1992. Archived at <http://www.w3.org/History/19921103-hypertext/hypertext/WWW/WorkingNotes/Overview.html>, Sep. 2000.

[18]. T. Berners-Lee. Universal Resource Identifiers in WWW. *Internet RFC 1630,* June 1994.

[19]. T. Berners-Lee, R. Cailliau, A. Luotonen, H. Frystyk Nielsen, and A. Secret. The World-Wide Web. *Communications of the ACM,* 37(8), Aug. 1994, pp. 76-82.

[20]. T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). *Internet RFC 1738,* Dec. 1994.
[21]. T. Berners-Lee and D. Connolly. Hypertext Markup Language — 2.0. *Internet RFC 1866,* Nov. 1995.
[22]. T. Berners-Lee, R. T. Fielding, and H. F. Nielsen. Hypertext Transfer Protocol — HTTP/1.0. *Internet RFC 1945,* May 1996.
[23]. T. Berners-Lee. WWW: Past, present, and future. *IEEE Computer,* 29(10), Oct. 1996, pp. 69-77.

About the Authors:

Satyam Arragokula working as an Assistant Professor in University College Of Science, Saifabad, Hyderabad, he completed M.E From Osmania University and Completed CSE (MSC(IS)) from Osmania University, Hyderabad.

M. Yesu Ratnam working as an Assistant Professor in Nizam College, he completed M.Tech from JNTU Hyderabad and completed MCA from Osmania University Hyderabad