Delft University of Technology
Software Engineering Research Group
Technical Report Series

A Component- and Push-based Architectural Style for Ajax Applications

Ali Mesbah and Arie van Deursen

Report TUD-SERG-2008-013

TUDelft

SERG

TUD-SERG-2008-013

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note: Accepted for publication in the Journal of Systems and Software (JSS), Elsevier, 2008.

© copyright 2008, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

A Component- and Push-based Architectural Style for Ajax Applications [1]

Ali Mesbah [a] Arie van Deursen [b]

[a] *Delft University of Technology*
Software Engineering Research Group
The Netherlands

[b] *Delft University of Technology & CWI*
Software Engineering Research Group
The Netherlands

Abstract

A new breed of web application, dubbed Ajax, is emerging in response to a limited degree of interactivity in large-grain stateless Web interactions. At the heart of this new approach lies a single page interaction model that facilitates rich interactivity. Also push-based solutions from the distributed systems are being adopted on the web for Ajax applications. The field is, however, characterized by the lack of a coherent and precisely described set of architectural concepts. As a consequence, it is rather difficult to understand, assess, and compare the existing approaches. We have studied and experimented with several Ajax frameworks trying to understand their architectural properties. In this paper, we summarize four of these frameworks and examine their properties and introduce the Spiar architectural style which captures the essence of Ajax applications. We describe the guiding software engineering principles and the constraints chosen to induce the desired properties. The style emphasizes user interface component development, intermediary delta-communication between client/server components, and push-based event notification of state changes through the components, to improve a number of properties such as user interactivity, user-perceived latency, data coherence, and ease of development. In addition, we use the concepts and principles to discuss various open issues in Ajax frameworks and application development.

Key words: Ajax, web architectural style, web engineering, single page interface, rich internet application

Email addresses: `A.Mesbah@tudelft.nl` (Ali Mesbah), `Arie.vanDeursen@tudelft.nl` (Arie van Deursen).

[1] This is a substantially revised and extended version of our paper 'An Architectural

1 Introduction

Over the course of the past decade, the move from desktop applications towards web applications has gained much attention and acceptance. Within this movement, however, a great deal of user interactiveness has been lost. Classical web applications are based on a *multi page interface* model, in which interactions are based on a page-sequence paradigm. While simple and elegant in design for exchanging documents, this model has many limitations for developing modern web applications with user friendly human-computer interaction.

Recently, there has been a shift in the direction of web development towards the new generation of Web 2.0 applications. A new breed of web application, dubbed Ajax (Asynchronous JavaScript And XML) [25], has been emerging in response to the limited degree of interactivity in large-grain stateless web interactions. Ajax utilizes a set of existing web technologies, previously known as *Dynamic HTML (DHTML)* and *remote scripting* [15], to provide a more interactive web-based user interface.

At the heart of this new approach lies a *single page interface* model that facilitates rich interactivity. In this model, changes are made to individual user interface components contained in a web page, as opposed to (refreshing) the entire page.

Another recent development, under the same umbrella, is applying the push-based concepts from the distributed systems to the web [29]. For applications that require *real-time event notifications*, the client-initiated pull model is very inefficient and might lead to network congestion. The push-based style, where the server broadcasts the state changes to the clients asynchronously every time its state changes, is emerging as an alternative on the web, which is known as Comet [44] or Reverse Ajax [16]. Each of these options has its own architectural trade-offs.

Thanks to the momentum of Ajax, the technology has attracted a strong interest in the web application development community. After the name Ajax was coined in February 2005 [25], numerous frameworks[2] and libraries have appeared, many web applications have adopted one or more of the ideas underpinning Ajax, and an overwhelming number of articles in developer sites and professional magazines have appeared.

Style for Ajax' which appeared in the *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2007 [33].

[2] At the time of writing more than 150 frameworks are listed at `http://ajaxpatterns.org/Frameworks`.

Adopting AJAX-based techniques is a serious option not only for newly developed applications, but also for ajaxifying [34] existing web sites if their user friendliness is inadequate.

A software engineer considering adopting AJAX, however, is faced with a number of challenges. What are the fundamental architectural trade-offs between designing a legacy web application and an AJAX web application? How would introducing a push-based style affect the scalability of web applications? What are the different characteristics of AJAX frameworks? What do these frameworks hide? Is there enough support for designing such applications? What problems can one expect during the development phase? Will there be some sort of convergence between the many different technologies? Which architectural elements will remain, and which ones will be replaced by more elegant or more powerful solutions? Addressing these questions calls for a more abstract perspective on AJAX web applications. However, despite all the attention the technology is receiving in the web community, there is a lack of a coherent and precisely described set of architectural formalisms for AJAX enabled web applications. In this paper we explore whether concepts and principles as developed in the software architecture research community can be of help to answer such questions.

To gain an abstract perspective, we have studied a number of AJAX frameworks, abstracted their features, and documented their common architectural elements and desired properties. In particular, we propose SPIAR, the Single Page Internet Application aRchitectural style, which emphasizes user interface component-based development, intermediary delta-communication between client/server components, and push-based event notification of state changes through the components, to improve a number of properties such as user interactivity, user-perceived latency, data coherence, and ease of development. The style can be used when high user interaction and responsiveness are desired in web applications.

One of the challenges of proposing an architectural style is the difficulty of evaluating the success of the style. As also observed by Fielding [20], the success of an architecture is ultimately determined by the question whether a system built using the style actually meets the stated requirements. Since this is impossible to determine in advance, we will evaluate our style in the following ways:

(1) We investigate how well existing AJAX frameworks such as GWT or Echo2 are covered by the style;
(2) We discuss how a number of typical AJAX architectures (client-centric, server centric, push-based) are covered by the style;
(3) We show how the style can be used to discuss various tradeoffs in the design of AJAX applications, related to, architectural properties such as

scalability and adaptability.

This paper is organized as follows. We start out, in Section 2 by exploring AJAX, studying four frameworks (Google's GWT, Backbase, Echo2, and the push-based Dojo/Cometd framework) that have made substantially different design choices. Then, in Section 3, we survey existing architectural styles (such as the Representational State Transfer architectural style REST on which the World Wide Web is based [21]), and analyze their suitability for characterizing AJAX. In Sections 4–7, we introduce SPIAR, describing the architectural properties, elements, views of this style, and the constraints. Given SPIAR, in Section 8 we use its concepts and principles to discuss various open issues in AJAX frameworks and application development and evaluate the style itself. We conclude with a summary of related work, contributions, and an outlook to future work.

2 Ajax Technology

2.1 AJAX

The ability to update a Web page incrementally has been around for several years but it was not until recently when the AJAX acronym was coined that the approach gained wide appeal.

AJAX is actually the name given to a set of modern web application development technologies, previously known as *Dynamic HTML (DHTML)* and emphremote scripting, to provide a more interactive web-based user interface.

As defined originally by Garrett [25], AJAX incorporates standards-based presentation using XHTML and CSS, dynamic display and interaction using the Document Object Model, data interchange and manipulation, asynchronous data retrieval using XMLHttpRequest, and JavaScript binding everything together. This definition, however, merely focuses on the client side of the web application spectrum. As we will see in this paper, the use of AJAX has important server side implications as well.

AJAX represents an approach to web application development utilizing a combination of established web technologies. It is the combination of these technologies that makes AJAX useful on the Web.

Rather than the classical model of 'request-wait-response' to the server and reloading the entire page for each user action, AJAX allows the interaction with the server to take place asynchronously in the background to update

portions of the page. This *behind the scenes* approach can provide a much more responsive and faster experience on the Web.

Even before the term AJAX was coined, its power was becoming evident by web applications such as Google Suggest, Google Docs or Google Map. Other well known examples are Gmail, the recent version of Yahoo! Mail, Flickr and Digg. For more technical details of AJAX we refer to [1, 15].

2.2 Reverse AJAX*:* COMET

The traditional web model requires all communication between the browser and the web server to be initiated by the client, i.e., the end user clicks on a button or link and thereby requests a new page from the server. Thus, once a complete response is returned, there is no further way for the server to send data back to the client browser. In this scheme, each interaction between the client and the server is independent of the other interactions. No 'permanent' connection is established between the client and the server maintains no state information about the clients. request and the server returns a response for this particular request. This scheme helps scalability, but precludes servers from sending asynchronous notifications. There are many use cases where it is important to update the client user interface in response to server-side changes. Examples include:

- An auction web site, where the users need to be averted that another bidder has made a higher bid. In a site such as eBay, the user has to continuously press the 'refresh' button of his or her browser, to see if somebody has made a higher bid.
- A stock ticker, where stock prices are updated,
- A chat application, where new sent messages are delivered to all the subscribers,
- A news portal, where news items are pushed to the subscriber browser when they are published.

With current web technologies, these types of applications requiring real-time *event notification* and *data delivery* are mainly implemented using a pull style, where the client actively requests the state changes using client-side timeouts. The pull style has many drawbacks, such as the low level of coherence between client and server side data, and increased network usage. If the updates are not frequent, clients will make unnecessary requests. On the other hand, if the clients pull infrequently, they might miss some valuable updates [9].

An alternative to such a pull style is the *push-based style* [26, 9], where the server broadcasts the state changes to the clients asynchronously every time its state changes. The concept of pushing or *streaming* web data by the server

was first introduced in 1992 by Netscape, under the name 'dynamic document' [38]. This method simply consists of streaming server data in the response of a long- lived HTTP connection. Most web servers do some processing, send back a response, and immediately exit. But in this pattern, the connection is kept open by running a long loop. The server script uses event registration or some other technique to detect any state changes. As soon as a state change occurs, it streams the new data and flushes it, but does not actually close the connection.

Comet [44] or Reverse Ajax [16] is the new name given to this style of interaction on the web. Comet uses the XMLHttpRequest object to have an open connection. This brings some flexibility regarding the length and frequency of connections. After the initial request from the client, the server does not close the connection, nor does it give a full response. As the new data becomes available, the server returns it to the client using the existing connection.

This new paradigm on the web introduces some potential architectural trade-offs while having valuable benefits for the aforementioned use-cases.

2.3 Frameworks

Web application developers have struggled constantly with the limits of the HTML page-sequence experience, and the complexities of client-side JavaScript programming to add some degree of dynamism to the user interface. Issues regarding cross-browser compatibility are, for instance, known to everyone who has built a real-world web application. The rich user interface (UI) experience Ajax promises comes at the price of facing all such problems. Developers are required to have advanced skills in a variety of Web technologies, if they are to build robust Ajax applications. Also, much effort has to be spent on testing these applications before going in production. This is where frameworks come to the rescue. At least many of them claim to.

Because of the momentum Ajax has gained, a vast number of frameworks are being developed. The importance of bringing order to this competitive chaotic world becomes evident when we learn that 'almost one new framework per day' is being added to the list of known frameworks[3].

We have studied and experimented with several Ajax frameworks trying to understand their architectural properties. We summarize four of these frameworks in this section. Our selection includes a widely used open source framework called Echo2, the web framework offered by Google called GWT, the

[3] http://ajaxpatterns.org/wiki/index.php?title=AJAXFrameworks

commercial package delivered by Backbase and the Dojo/Cometd push-based comet framework. All four frameworks are major players in the Ajax market, and their underlying technologies differ substantially.

2.3.1 Echo2

Echo2[4] is an open-source Ajax framework which allows the developer to create web applications using an object-oriented, UI component-based, and event-driven paradigm for Web development. Its Java *Application Framework* provides the APIs (for UI components, property objects, and event/listeners) to represent and manage the state of an application and its user interface.

All functionality for rendering a component or for communicating with the client browser is specifically assembled in a separate module called the *Web Rendering Engine*. The engine consists of a server-side portion (written in Java/J2EE) and a client-side portion (JavaScript). The client/server interaction protocol is hidden behind this module and as such, it is entirely decoupled from other modules. Echo2 has an *Update Manager* which is responsible for tracking updates to the user interface component model, and for processing input received from the rendering agent and communicating it to the components.

The *Echo2 Client Engine* runs in the client browser and provides a remote user interface to the server-side application. Its main activity is to synchronize client/server state when user operations occur on the interface.

A *ClientMessage* in XML format is used to transfer the client state changes to the server by explicitly stating the nature of the change and the component ID affected. The server processes the ClientMessage, updating the component model to reflect the user's actions. Events are fired on interested listeners, possibly resulting in further changes to the server-side state of the application. The server responds by rendering a *ServerMessage* which is again an XML message containing directives to perform partial updates to the DOM representation on the client.

2.3.2 GWT

Google has a novel approach to implementing its Ajax framework, the Google Web Toolkit (GWT)[5]. Just like Echo2, GWT facilitates the development of UIs in a fashion similar to AWT or Swing and comes with a library of widgets that can be used. The unique character of GWT lies in the way it renders

4 Echo2 2.0.0, `www.nextapp.com/platform/echo2/echo/`.
5 http://code.google.com/webtoolkit/

the client-side UI. Instead of keeping the UI components on the server and communicating the state changes, GWT compiles all the Java UI components to JavaScript code (compile-time). Within the components the developer is allowed to use a subset of Java 1.4 API to implement needed functionality.

GWT uses a small generic client engine and, using the compiler, all the UI functionality becomes available to the user on the client. This approach decreases round-trips to the server drastically. The server is only consulted if raw data is needed to populate the client-side UI components. This is carried out by making server calls to defined services in an RPC-based style. The services (which are not the same as Web Services) are implemented in Java and data is passed both ways over the network, in JSON format, using serialization techniques.

2.3.3 Backbase

Backbase[6] is an Amsterdam-based company that provided one of the first commercial Ajax frameworks. The framework is still in continuous development, and in use by numerous customers world wide.

A key element of the Backbase framework is the Backbase Client Run-time (BCR). This a standards-based Ajax engine written in JavaScript that runs in the web browser. It can be programmed via a declarative user interface language called XEL. XEL provides an application-level alternative to JavaScript and manages asynchronous operations that might be tedious to program and manage using JavaScript.

BCR's main functionality is to:

- create a single page interface and manage the widget tree (view tree),
- interpret JavaScript as well as the XEL language,
- take care of the synchronization and state management with the server by using delta-communication, and asynchronous interaction with the user through the manipulation of the representational model.

The Backbase framework provides a markup language called Backbase Tag Library (BTL). BTL offers a library of widgets, UI controls, a mechanism for attaching actions to them, as well as facilities for connecting to the server asynchronously.

The server side of the Backbase framework is formed by BJS, the Backbase Jsf Server. It is built on top of JavaServer Faces (Jsf)[7], the new J2EE presenta-

[6] http://www.backbase.com
[7] JavaServer Faces Specification v1.1, http://java.sun.com/j2ee/javaserverfaces/

tion architecture. JSF provides a user interface component-based framework following the model-view-controller pattern. Backbase JSF Server utilizes all standard JSF mechanisms such as validation, conversion and event processing through the JSF life-cycle phases. The interaction in JSF is, however, based on the classical page sequence model, making integration in a single page framework non trivial. Backbase extends the JSF request life-cycle to work in a single-page interface environment. It also manages the server-side event handlers and the server-side control tree.

Any Java class that offers getters and setters for its properties can be directly assigned to a UI component property. Developers can use the components declaratively (web-scripting) to build an AJAX application. The framework renders each declared server-side UI component to a corresponding client-side XEL UI component, and keeps track of changes on both component trees for synchronization.

The state changes on the client are sent to the server on certain defined events. Backbase uses DOM events to delegate user actions to BCR which handles the events asynchronously. The events can initiate a client-side (local) change in the representational model but at the same time these events can serve as triggers for server-side event listeners. The server translates these state changes and identifies the corresponding component(s) in the server component tree. After the required action, the server renders the changes to be responded to the engine, again in XEL format.

2.3.4 Dojo and Cometd

The final framework we consider is the combination of the client-side Dojo and the server-side Cometd frameworks, which together support a push-based client-server communication. The framework is based on the BAYEUX protocol which the Cometd group [8] has recently released, as a response to the lack of communication standards. For more details see [9, 8].

The BAYEUX message format is defined in JSON (JavaScript Object Notation) [9] which is a data-interchange format based on a subset of the JavaScript Programming Language. The protocol has recently been implemented and included in a number of web servers including Jetty [10] and IBM Websphere [11].

The frameworks that implement BAYEUX currently provide a connection type

[8] http://www.cometd.com
[9] http://www.json.org
[10] http://www.mortbay.org
[11] http://www-306.ibm.com/software/websphere/

called *Long Polling* for HTTP push. In Long Polling, the server holds on to the client request, until data becomes available. If an event occurs, the server sends the data to the client and the client has to reconnect. Otherwise, the server holds on to the connection for a finite period of time, after which it asks the client to reconnect again. If the data publish interval is low, the system will act like a pure pull, because the clients will have to reconnect (make a request) often. If the data publish interval is high, then the system will act like a pure push.

Bayeux defines the following phases in order to establish a Comet connection. The client:

(1) performs a handshake with the server and receives a client ID,
(2) sends a connection request with its ID,
(3) subscribes to a channel and receives updates.

Bayeux is supported by the client-side Ajax framework called Dojo [12]. It is currently written entirely in JavaScript and there are plans to adopt a markup language in the near future. Dojo provides a number of ready-to-use UI widgets which are prepackaged components of JavaScript code, as well as an abstracted wrapper (`dojo.io.bind`) around various browsers' implementations of the XMLHttpRequest object to communicate with the server. Dojo facilitates the `dojo.io.cometd` library, to make the connection handshake and subscribe to a particular channel.

On the server-side, Bayeux is supported by Cometd [13]. This is an HTTP-based event routing framework that implements the Comet style of interaction. It is currently implemented as a module in Jetty.

2.4 Features

While different in many ways, these frameworks share some common architectural characteristics. Generally, the goals of these frameworks can be summarized as follows:

- Hide the complexity of developing Ajax applications - which is a tedious, difficult, and error-prone task,
- Hide the incompatibilities between different web browsers and platforms,
- Hide the client/server communication complexities,
- All this to achieve rich interactivity and portability for end users, and ease of development for developers.

[12] `http://dojotoolkit.org`
[13] `http://www.cometd.com`

The frameworks achieve these goals by providing a library of user interface components and a development environment to create reusable custom components. The architectures have a well defined protocol for small interactions among known client/server components. Data needed to be transferred over the network is significantly reduced. This can result in faster response data transfers. Their architecture takes advantage of client side processing resulting in improved user interactivity, smaller number of round-trips, and a reduced web server load.

The architectural decisions behind these frameworks change the way we develop web applications. Instead of thinking in terms of sequences of Web pages, Web developers can now program their applications in the more intuitive (single page) component- and event-based fashion along the lines of, e.g., AWT and Swing.

3 Architectural Styles

In this section, we first introduce the architectural terminology used in this paper and explore whether styles and principles as developed in the software architecture research community, and specifically those related to network-based environments, can be of help in formalizing the architectural properties of Ajax applications.

3.1 Terminology

In this paper we use the software architectural concepts and terminology as used by Fielding [20] which in turn is based on the work of Perry and Wolf [42]. Thus, a software architecture is defined [42] as a configuration of architectural elements — processing, connectors, and data — constrained in their relationships in order to achieve a desired set of architectural properties.

An architectural style, in turn, [20] is a coordinated set of architectural constraints that restricts the roles of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style. An architectural style constrains both the design elements and the relationships among them [42] in such a way as to result in software systems with certain desired properties. Clements et al.[14] define an architectural style as a specialization of element and relation types, together with a set of constraints on how they can be used. A style can also be seen as an abstraction of recurring composition and interaction characteristics in a set of architectures.

An architectural system can be composed of multiple styles and a style can be a hybrid of other styles [45]. Styles can be seen as reusable [36] common architectural patterns within different system architectures and hence the term *architectural pattern* is also used to describe the same concept [4].

The benefits of using styles can be summarized as follows:

- Design reuse: well-understood solutions applied to new problems
- Code reuse: shared implementations of invariant aspects of a style
- Understandability and ease of communication: phrases such as 'client-server' or 'Rest' make use of a vocabulary conveying a wealth of implicit information.
- Interoperability: supported by style standardization
- Specific trade-off analysis: enabled by the constrained design space
- Visualizations: specific depictions matching mental models

In our view, being able to understand the tradeoffs inherent in the architectures [28] of Ajax systems is the essence of using architectural styles. An architectural style enables us to pin-point relevant tradeoffs in different instantiated architectures.

3.2 Existing Styles

Client/server [46], n-tier [51], and Mobile Code [12, 24], are all different network-based architectural styles [20], which are relevant when considering the characteristics of Ajax applications.

In addition, user interface applications generally make use of popular styles such as Module/View/Controller [32] to describe large scale architecture and, in more specific cases, styles like C2 [49] to rely on asynchronous notification of state changes and request messages between independent components.

There are also a number of interactional styles, such as event observation and notification [43], publish/subscribe [19], the component and communication model [26], and ARRESTED [31], which model the client/server push paradigm for distributed systems.

In our view, the most complete and appropriate style for the Web, thus far, is the REpresentational State Transfer (Rest) [21]. Rest emphasizes the abstraction of data and services as resources that can be requested by clients using the resource's name and address, specified as a Uniform Resource Locator (URL) [5]. The style inherits characteristics from a number of other styles such as client/server, pipe-and-filter, and distributed objects.

The style is a description of the main features of the Web architecture through architectural constraints which have contributed significantly to the success of the Web.

It revolves around five fundamental notions: a *resource* which can be anything that has identity, e.g., a document or image, the *representation of a resource* which is in the form of a media type, *synchronous request-response interaction* over HTTP to obtain or modify representations, a *web page* as an instance of the application state, and *engines* (e.g., browser, crawler) to move from one state to the next.

REST specifies a client-stateless-server architecture in which a series of proxies, caches, and filters can be used and each request is independent of the previous ones, inducing the property of scalability. It also emphasizes a uniform interface between components constraining information to be transferred in a standardized form.

3.3 A Style for Ajax

AJAX applications can be seen as a hybrid of desktop and web applications, inheriting characteristics from both worlds. Table 1 summarizes the differences between what REST provides and what modern AJAX (with COMET) applications demand. AJAX frameworks provide back-end services through UI components to the client in an event-driven or push style. Such architectures are not so easily captured in REST, due to the following differences:

- While REST is suited for large-grain hypermedia data transfers, because of its uniform interface constraint it is not optimal for small data interactions required in AJAX applications.
- REST focuses on a hyper-linked resource-based interaction in which the client requests a specific *resource*. In contrast, in AJAX applications the user interacts with the system much like in a desktop application, requesting a response to a specific *action*.
- All interactions for obtaining a resource's representation are performed through a synchronous request-response pair in REST. AJAX applications, however, require a model for asynchronous communication.
- REST explicitly constrains the server to be stateless, i.e., each request from the client must contain all the information necessary for the server to understand the request. While this constraint can improve scalability, the tradeoffs with respect to network performance and user interactivity are of greater importance when designing an AJAX architecture.
- REST is cache-based while AJAX facilitates real-time data retrieval.
- Every request must be initiated by a client, and every response must be

Table 1
What REST provides versus what AJAX demands

REST provides	AJAX demands
Large-grain hypermedia data transfers	Small data interactions
Resource-based	UI component-based
Hyper-linked	Action- Event-based
Synchronous request-response	Asynchronous interaction
Stateless	Stateful
Cache-based	Real-time data retrieval
Poll-based	Poll and Push

generated immediately; every request can only generate a single response [31]. COMET requires a model which enables pushing data from the server to the client.

These requirement mismatches call for a new architectural style capable of meeting the desired properties.

4 Architectural Properties

The architectural properties of a software architecture include both the functional properties achieved by the system and non-functional properties, often referred to as quality attributes [4, 40]. The properties could also be seen as requirements since architecting a system requires an understanding of its requirements.

Below we discuss a number of architectural properties that relate to the essence of AJAX. Other properties, such as extensibility or security, that may be desirable for any system but are less directly affected by a decision to adopt AJAX, are not taken into account. Note that some of the properties discussed below are related to each other: for instance, user interactivity is influenced by user-perceived latency, which in turn is affected by network performance.

4.1 User Interactivity

The Human-computer interaction literature defines interactivity as the degree to which participants in a communication process have control over, and can exchange roles in their mutual discourse. User interactivity is closely related

to *usability* [22], the term used in software architecture literature. Teo *et al.* [50] provide a thorough study of user interactivity on commercial web applications. Their results suggest that an increased level of interactivity has positive effects on user's perceived satisfaction, effectiveness, efficiency, value, and overall attitude towards a Web site. Improving this property on the Web has been the main motivating force behind the AJAX movement.

4.2 User-perceived Latency

User-perceived latency is defined as the period between the moment a user issues a request and the first indication of a response from the system. Generally, there are two primary ways to improve user-perceived performance. First, by reducing the round-trip time, defined as time elapsed for a message from the browser to a server and back again, and second, by allowing the user to interact asynchronously with the system. This is an important property in all distributed applications with a front-end to the user.

4.3 Network Performance

Network performance is influenced by *throughput* which is the rate of data transmitted on the network and *bandwidth*, i.e., a measure of the maximum available throughput. Network performance can be improved by means of reducing the amount and the granularity of transmitted data.

4.4 Simplicity

Simplicity or development effort is defined as the effort that is needed to understand, design, implement, maintain and evolve a web application. It is an important factor for the usage and acceptance of any new approach.

4.5 Scalability

In distributed environments scalability is defined by the degree of a systems ability to handle a growing number of components. In Web engineering, a system's scalability is determined, for instance, by the degree to which a client can be served by different servers without affecting the results. A scalable Web architecture can be easily configured to serve a growing number of client requests.

4.6 Portability

Software that can be used in different environments is said to be portable. On the Web, being able to use the Web browser without the need for any extra actions required from the user, e.g., downloading plug-ins, induces the property of portability.

4.7 Visibility

Visibility [20] is determined by the degree to which an external mediator is able to understand the interactions between two components, i.e., the easier it is for the mediator to understand the interactions, the more visible the interaction between the two components will be. Looking at the current implementations of Ajax frameworks, visibility in the client/server interactions is low, as they are based on proprietary protocols. Although a high level of visibility makes the interaction more comprehensible, the corresponding high observability can also have negative effects on security issues. Thus low visibility is not per se an inferior characteristic, depending on the desired system property and tradeoffs made.

4.8 Reliability

Reliability is defined as the continuity of correct service [2]. The success of any software system depends greatly on its reliability. On the Internet, web applications that depend on unreliable software and do not work well, will lose customers [40]. Testing (test automation, unit and regression testing) resources can improve the reliability level of an application. However, web applications are generally known to be poorly tested compared to traditional desktop applications. In addition to the short time-to-market pressure, the multi-page interaction style of the web makes it difficult to test. Adopting a single page component-based style of web application development can improve the testability of the system and as a consequence its reliability.

4.9 Data Coherence

An important aspect of real-time event notification of web data that need to be available and communicated to the user as soon as they happen, e.g., stock prices, is the maintenance of data coherence [6]. A piece of data is defined as coherent, if the data on the server and the client is synchronized. In

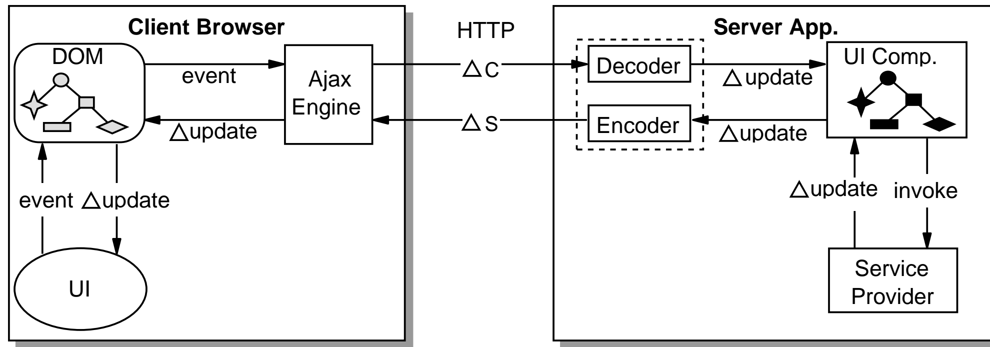

Fig. 1. Processing View of a SPIAR-based architecture.

web applications adhering to the HTTP protocol, clients need to frequently pull the data based on a pre-defined interval. In contrast, servers that adopt push capability maintain state information pertaining to clients and stream the changes to users as they happen. These two techniques have different properties with respect to the data coherence achieved [9].

4.10 Adaptability

Adaptability is defined as the ease with which a system or parts of the system may be adapted to the changing environment. In web applications, an architecture that allows changes on the server to be propagated to the clients is called adaptable. We use the notion of *code mobility* [24] to compare the dynamic behavior of different AJAX architectures in terms of changeability and adaptability. Mobile code is, generally, software code obtained from remote servers, transferred across a network, and then downloaded and executed on the client without explicit installation or execution by the recipient.

5 SPIAR Architectural Elements

Following [20, 42], the key architectural elements of SPIAR are divided into three categories, namely processing (components), data, and connecting elements. An overview of the elements is shown in Figure 1. In this section we explain the elements themselves, while in the next section we discuss their interaction.

5.1 Processing Elements

The processing elements are defined as those components that supply the transformation on the data elements.

The *Client Browser* offers support for a set of standards such as HTTP, HTML, Cascading Style Sheets, JavaScript, and Document Object Model. It processes the representational model of a web page to produce the user interface. The user interaction can be based on a single page user interface model. All the visual transitions and effects are presented to the user through this interface. Just like a desktop client application, it consists of a single main page with a set of identifiable widgets. The properties of widgets can be manipulated individually while changes are made in-place without requiring a page refresh.

The AJAX *Engine* is a client engine that loads and runs in the client browser. There is no need for a plug-in for the web application to function. However, downloading the engine does introduce an initial latency for the user which can be compensated by the smaller data transfers once the engine is in place. The engine is responsible for the initialization and manipulation of the representational model. As can be seen in Figure 1, the engine handles the events initiated by the user, communicates with the server, and has the ability to perform client-side processing.

The *Server Application* resides on the server and operates by accepting HTTP-based requests from the network, and providing responses to the requester. All server-side functionality resides in the server application processing element.

The *Service Provider* represents the logic engine of the server and processes state changes and user requested actions. It is capable of accessing any resource (e.g., database, Web Services) needed to carry out its action. A Service Provider's functionality is invoked by event listeners, attached to components, initiated by incoming requests.

The *Delta Encoder/Decoder* processes outgoing/incoming delta messages. It is at this point that the communication protocol between the client and the server is defined and hidden behind an interface. This element supports delta communication between client and server which improves user-perceived latency and network performance.

UI Components consist of a set of server-side UI components. The component model on the server is capable of rendering the representational model on the client. Each server-side component contains the data and behavior of that part of the corresponding client-side widget which is relevant for state changes; There are different approaches as when and how to render the client-side UI code. GWT, for instance, renders the entire client-side UI code compile-time

from the server-side Java components. Echo2 which has a real component-based architecture, on the other hand, renders the components at run-time and keeps a tree of components on both client and server side. These UI components have event listeners that can be attached to client-side user initiated events such as clicking on a button. This element enhances simplicity by providing off-the-shelf components to build web applications.

A *Push Server* resides as a separate module on the server application. This processing element has the ability to keep an HTTP connection open to push data from the server to the client. The Service Provider can publish new data (state changes) to this element.

A *Push Client* element resides within the client. It can be a separate module, or a part of the Ajax Engine. This element can subscribe to a particular channel on the Push Server element and receive real-time publication data from the server.

5.2 Data Elements

The data elements contain the information that is used and transformed by the processing elements.

The *Representation* element consists of any media type just like in Rest. HTML, CSS, and images are all members of this data element.

The *Representational Model* is a run-time abstraction of how a UI is represented on the client browser. The Document Object Model inside the browser has gained a very important role in Ajax applications. It is through dynamically manipulating this representational model that rich effects have been made possible. Some frameworks such as Backbase use a domain-specific language to declaratively define the structure and behavior of the representational model. Others like GWT use a direct approach by utilizing JavaScript.

Delta communicating messages form the means of the delta communication protocol between client and server. Spiar makes a distinction between the client delta data (delta-client) and the server delta data (delta-server). The former is created by the client to represent the client-side state changes and the corresponding actions causing those changes, while the latter is the response of the server as a result of those actions on the server components. The delta communicating data are found in a variety of formats in the current frameworks, e.g., XML, JavaScript Object Notation (JSON), or pure JavaScript. The client delta messages contain the needed information for the server to know for instance which action on which component has to be carried out.

```
REQUEST (DELTA-CLIENT):

POST http://demo.nextapp.com/Demo/app/Demo/app?serviceId=Echo.Synchronize
Content-Type: text/xml; charset=UTF-8
<client-message trans-id="1">
 <message-part processor="EchoAction">
  <action component-id="c_7" name="click"/>
 </message-part>
</client-message>

RESPONSE (DELTA-SERVER):

<?xml version="1.0" encoding="UTF-8"?>
<server-message
 xmlns="http://www.nextapp.com/products/echo2/svrmsg/servermessage"
 trans-id="2">
 <message-part-group id="update">
  ...
  <message-part processor="EchoDomUpdate.MessageProcessor">
   <dom-add>
    <content parent-id="c_35_content">
     <div id="c_36_cell_c_37" style="padding:0px;">
      <span id="c_37" style="font-size:10pt;font-weight:bold;">
        Welcome to the Echo2 Demonstration Application.
      </span>
     </div>
    </content>
   </dom-add>
  </message-part>
  ...
 </message-part-group>
</server-message>
```

Fig. 2. An example of Echo2 delta-communication.

As an example, Figure 2 illustrates delta-communication in Echo2. After the user clicks on a component (button) with ID `c_7`, the client-side engine detects this `click` event and creates the DELTA-CLIENT (in Echo2 called `client-message`) and posts it to the server as shown in the REQUEST part of Figure 2. The server then, using the information in the DELTA-CLIENT which, in this case, is composed of the action, component ID, and the event type, responds with a DELTA-SERVER in XML format. As can be seen, the DELTA-SERVER tells the client-side engine exactly how to update the DOM state with new style and textual content on a particular parent component with ID `c_35_content`.

We distinguish between three types of code that can change the state of the client: *presentational code*, *functional code*, and *textual data*.

Presentational code as its name suggests has influence on the visual style and presentation of the application, e.g., CSS, or HTML. Textual data is simply pure data. The functional code can be executed on the client, e.g., JavaScript code, or commands in XML format (e.g., `dom-add` in Figure 2). The DELTA-SERVER can be composed of any of these three types of code. These three types of code can influence the Representational model (DOM) of the client application which is the run-time abstraction of the presentational

code, executed functional code and textual data.

GWT uses an RPC style of calling services in which the DELTA-SERVER is mainly composed of textual data, while in Backbase and Echo2 a component-based approach is implemented to invoke event listeners, in a mixture of presentational and functional code.

5.3 Connecting Elements

The connecting elements serve as the glue that holds the components together by enabling them to communicate.

Events form the basis of the interaction model in SPIAR. An event is initiated by each action of the user on the interface, which propagates to the engine. Depending on the type of the event, a request to the server, or a partial update of the interface might be needed. The event can be handled asynchronously, if desired, in which case the control is immediately returned to the user.

On the server, the request initiated by an event *invokes* a service. The service can be either invoked directly or through the corresponding UI component's event listeners.

Delta connectors are light-weight communication media connecting the engine and the server using a request/response mechanism over HTTP.

Delta updates are used to update the representational model on the client and the component model on the server to reflect the state changes. While a delta update of the representational model results in a direct apparent result on the user interface, an update of the component model invokes the appropriate listeners. These updates are usually through procedural invocations of methods.

Channels are the connecting elements between the push consumer and producer. A consumer (receiver) *subscribes* to a channel, through a handshake procedure, and receives any information that is sent on the channel by the producer (information source) as *delta push server*.

6 Architectural Views

Given the processing, data, and connecting elements, we can use different architectural views to describe how the elements work together to form an architecture. Here we make use of two processing views, which concentrate

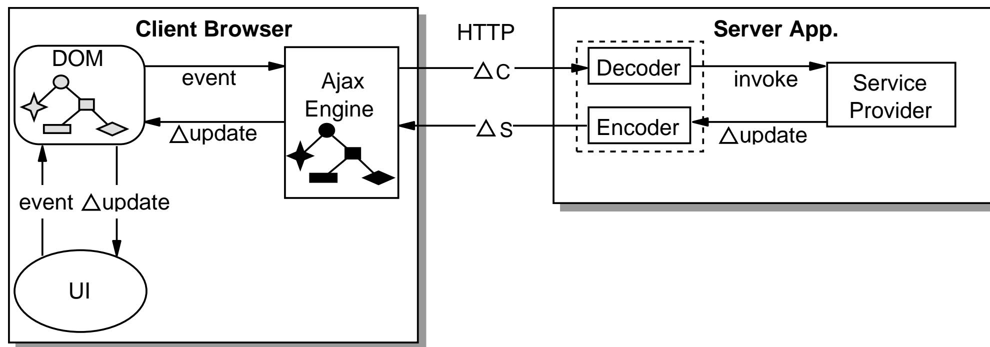

Fig. 3. Processing View of an RPC-based GWT architecture.

on the data flow and some aspects of the connections among the processing elements with respect to the data [20]. Such views fit in the Components and Connectors viewtype as discussed by Clements et al.[14]. We discuss one processing view for a pure component-based Ajax solution, one for an RPC-based Ajax application, and one view for the push-based variant.

6.1 Ajax *view*

Figure 1 depicts the processing view of an Spiar-based architecture based on run-time components rendering as in, e.g., Echo2. The view shows the interaction of the different components some time after the initial page request (the engine is running on the client).

User activity on the user interface fires off an event to indicate some kind of component-defined action which is delegated to the Ajax engine. If a listener on a server-side component has registered itself with the event, the engine will make a delta-client message of the current state changes with the corresponding events and send it to the server. On the server, the decoder will convert the message, and identify and notify the relevant components in the component tree. The changed components will ultimately invoke the event listeners of the service provider. The service provider, after handling the actions, will update the corresponding components with the new state which will be rendered by the encoder. The rendered delta-server message is then sent back to the engine which will be used to update the representational model and eventually the interface. The engine has also the ability to update the representational model directly after an event, if no round-trip to the server is required.

The run-time processing view of the GWT framework is depicted in Figure 3. As can be seen, GWT does not maintain a server-side component tree. Instead the server-side UI components are transformed into client-side components at compile-time. The client engine knows the set and location of all available com-

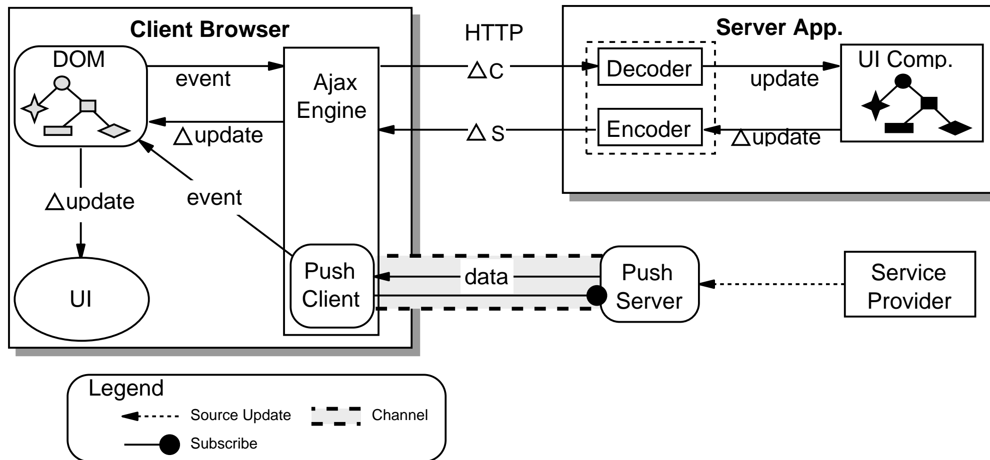

Fig. 4. Processing View of a push-based SPIAR architecture.

ponents at run-time. The RPC-based interaction with the server is however still conducted in a delta-communication style. Here, the encoder and decoder talk directly to the Service Provider without going through the server-side component model.

Note that each framework uses a different set of SPIAR's architectural elements to present the run-time architectural processing view. See also Section 8.1 for a discussion on how each approach fits in SPIAR.

6.2 COMET *view*

In the majority of the current COMET frameworks the data is pushed directly to the client as shown in Figure 4. This direct approach is fine for implementations that are not component-based. However, for UI component-based frameworks, if the push data is directly sent to the client, the client has to handle this data itself and update its components locally. To notify UI components on the server, the client has to send a client delta back to the server. This is inefficient, since in many cases, the push server and the application server are in the same machine or network. The SPIAR architectural style thus reveals an interesting tension between the UI component-based and the push-based constraint.

A possible solution [8] would be to take a short-cut for this synchronization process. Whenever an event arrives from the Service Provider (state change, new data), instead of publishing the new data directly to the client, first the state changes are announced to the UI Components for all the subscribed clients. The changes are then passed through the encoder to the Push Server and then passed as push delta-server to the push client.

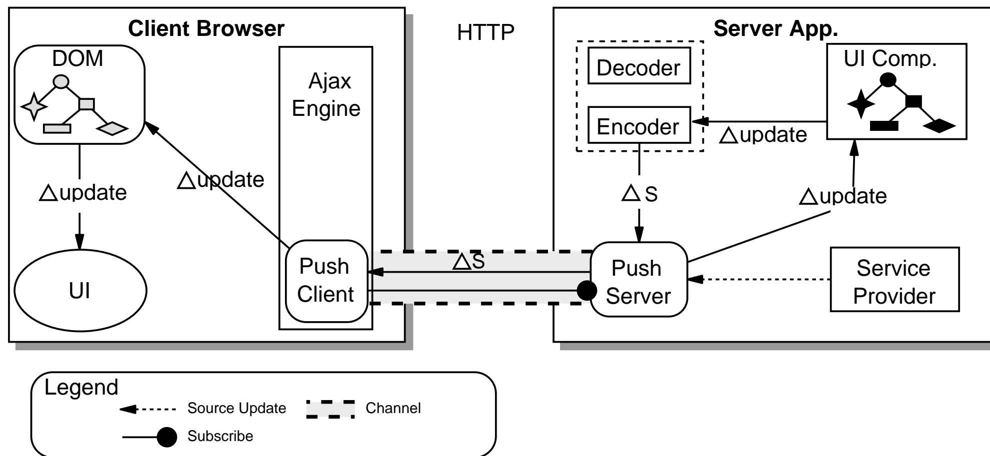

Fig. 5. Proposed push-based integration.

This approach makes sure that the state on the server is synchronized with the state on the client for each notification. Figure 5 depicts our proposed push-based view. The push client subscribes to a particular channel through the push server, and the changes are passed, through the component model, as push delta server, real-time to the client.

Note that the normal interaction of the client/server as depicted on Figure 1 can continue unhindered by the introduction of this push module. There are two advantages of this solution. First of all, it allows ΔS to be sent directly to the user in one step.

Second advantage is the simplicity for the application programmer. Without this solution, the programmer has to write explicit JavaScript functions in order to process the incoming push data. In the proposed solution, no such function is needed, since the response will be in an expected ΔS format, which will be processed by the Ajax Engine automatically.

7 Architectural Constraints

Architectural constraints can be used as restrictions on the roles of the architectural elements to induce the architectural properties desired of a system. Table 2 presents an overview of the constraints and induced properties. A "+" marks a direct positive effect, whereas a "–" indicates a direct negative effect.

Spiar rests upon the following constraints chosen to retain the properties identified previously in this paper.

7.1 Single Page Interface

SPIAR is based on the client-server style which is presumably the best known architecture for distributed applications, taking advantage of the separation of concerns principle in a network environment. The main constraint that distinguishes this style from the traditional Web architecture is its emphasis on a single page interface instead of the page-sequence model. This constraint induces the property of user interactivity. User interactivity is improved because the interaction is on a component level and the user does not have to wait for the entire page to be rendered again as a result of each action. Figure 6 and Figure 7 show the interaction style in a traditional web and in a single-page client-centric AJAX application respectively.

7.2 Asynchronous Interaction

AJAX applications are designed to have a high user interactivity and a low user-perceived latency. *Asynchronous interaction* allows the user to, subsequently, initiate a request to the server at any time, and receive the control back from the client instantly. The requests are handled by the client at the background and the interface is updated according to server responses. This model of interaction is substantially different from the classic synchronous request, wait for response, and continue model.

7.3 Delta-communication

Redundant data transfer which is mainly attributed to retransmissions of unchanged pages is one of the limitations of classic web applications. Many techniques such as caching, proxy servers and fragment-based resource change estimation and reduction [7], have been adopted in order to reduce data redundancy. Delta-encoding [35] uses caching techniques to reduce network traffic. However, it does not reduce the computational load since the server still needs to generate the entire page for each request [37].

SPIAR goes one step further, and uses a *delta-communication* style of interaction. Here merely the state changes are interchanged between the client and the server as opposed to the full-page retrieval approach in classic web applications. Delta-communication is based on delta-encoding architectural principles but is different: delta-communication does not rely on caching and as a result, the client only needs to process the deltas. All AJAX frameworks hide the delta-communication details from the developers.

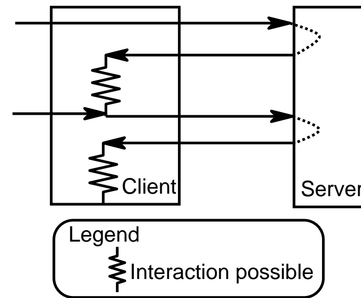

Fig. 6. Traditional multi-page Web Interaction.

This constraint induces the properties of network performance directly and as a consequence user-perceived latency and user interactivity. Network performance is improved because there are less redundant data (merely the delta) being transported. Data coherence is also improved because of the fine-grain nature of the data which can be transferred to the user faster than when dealing with data contained in large-grain web pages.

7.4 User Interface Component-based

Spiar relies on a single page user interface with components similar to that of desktop applications, e.g., AWT's UI component model. This model defines the state and behavior of UI components and the way they can interact.

UI component programming improves simplicity because developers can use reusable components to assemble a Web page either declaratively or programmatically. User interactivity is improved because the user can interact with the application on a component level, similar to desktop applications. In addition, testing component-based software is inherently easier than testing traditional page-based web applications, which induces the property of reliability.

Frameworks adhering to this constraint are very adaptable in terms of code mobility since state changes in the three code types (5.2) can be propagated to the client.

7.5 Web standards-based

Constraining the Web elements to a set of standardized formats is one way of inducing portability on the Web. This constraint excludes approaches that need extra functionality (e.g., plug-ins, virtual machine) to run on the Web browser, such as Flash and Java applets, and makes the client cross-browser compatible. This constraint limits the nature of the data elements to those that are supported by web browsers. Also using web standards, web browsers that

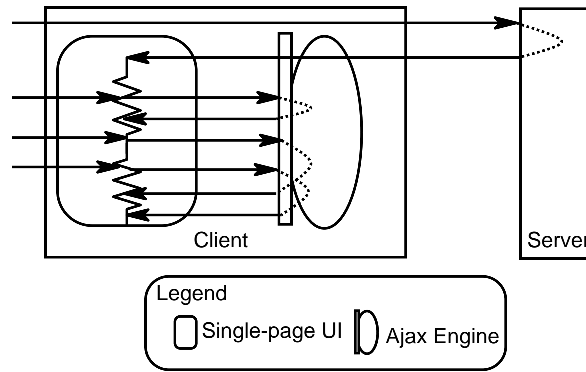

Fig. 7. Client-centric AJAX Interaction.

Table 2
Constraints and induced properties

	User Interactivity	User-perceived Latency	Network Performance	Simplicity	Scalability	Portability	Visibility	Data Coherence	Reliability	Adaptability
Single-page Interface	+									
Asynchronous Interaction	+	+								
Delta Communication	+	+	+		−		−	+		
Client-side processing	+	+	+							
UI Component-based	+			+					+	+
Web standards-based				+		+			+	
Stateful	+	+	+		−		−			
Push-based Publish/Subscribe		+	+		−		−	+		+

abide by standards are easily supported and hence some degree of reliability is induced [2].

7.6 Client-side Processing

Client-side processing improves user interactivity and user-perceived latency through round-trip reduction. For instance, client-side form validation reduces unnecessary server-side error reports and reentry messages. Additionally, some server-side processing (e.g., sorting items) can be off-loaded to clients using mobile code that will improve server performance and increase the availability to more simultaneous connections. As a tradeoff, client performance can become an issue if many widgets need processing resources on the client. GWT takes advantage of client-side processing to the fullest, by generating all the UI client-side code as JavaScript and run it on the client.

7.7 Stateful

A stateless server is one which treats each request as an independent transaction, unrelated to any previous request, i.e., each request must contain all of the information necessary to understand it, and cannot take advantage of any stored context on the server [21]. Even though the Web architecture and HTTP are designed to be stateless, it is difficult to think of stateless Web applications. Within a Web application, the order of interactions is relevant, making interactions depend on each other, which requires an awareness of the overall component topology. The statefulness is imitated by a combination of HTTP, client-side cookies, and server-side session management.

Unlike Rest, Spiar does not constrain the nature of the state explicitly. Nevertheless, since a stateless approach may decrease network performance (by increasing the repetitive data), and because of the component-based nature of the user interactions, a stateful solution might become favorable at the cost of scalability and visibility.

7.8 Push-based Publish/Subscribe

The client-server interaction can be realized in both a push- or pull-based style. In a push-based style [26], the server broadcasts the state changes to the clients asynchronously every time its state changes. Event-based Integration [3] and Asynchronous Rest [31] are event-based styles allowing asynchronous notification of state changes by the server. This style of interaction has mainly been supported in peer-to-peer architectural environments.

In a pull-based style, client components actively request state changes. Event-driven [39] architectures are found in distributed applications that require asynchronous communication, for instance, a desktop application, where user initiated UI inputs serve as the events that activate a process.

Comet enables us to mimic a push-based publish/subscribe [19] style of interaction on the web. This ability improves the network performance [10] because unnecessary poll requests are avoided. User-perceived latency, and adaptability are also improved by allowing a real-time event notification of state changes to clients. The results of our empirical study [9] show that data coherence is improved significantly by this constraint, but at the same time the server application performance and reliability can be deteriorated and as a result scalability negatively influenced.

Table 3
Ajax configurations and properties.

	User Interactivity	User-perceived Latency	Network Performance	Simplicity	Scalability	Portability	Visibility	Data Coherence	Reliability	Adaptability
Rest-based Classic Web	−	−	−	+	+	+	+	−	+−	−
Client-centric Ajax	+	+		−		+				−
ARPC Ajax	+	+	+	+			−			−
Push-based Ajax		+	+		−		−	+		+
Spiar-based Ajax	+	+	+	+	+−	+	−	+	+	+

8 Discussion and Evaluation

In this section we evaluate Spiar by investigating how well existing Ajax frameworks and typical Ajax architectures are covered by the style, and discuss various decisions and tradeoffs in the design of Ajax applications in terms of the architectural properties.

8.1 Retrofitting Frameworks onto Spiar

Each framework presented in Section 2 can be an architectural instance of Spiar, even if not fully complying with all the architectural constraints of Spiar. *Echo2* is the best representative of Spiar because of its fully event-driven and component-based architecture. The Jsf-based *Backbase* architecture is also well covered by Spiar even though Jsf is not a real event-based approach. *GWT*, on the other hand, is an interesting architecture. Although the architecture uses UI components during the development phase, these components are compiled to client-side code. GWT does not rely on a server-side component-based architecture and hence, does not fully comply with Spiar. None of these three frameworks has push-based elements. While the push-based constraint is well represented in the *Dojo and Cometd* framework, the component-based constraint is missing here.

Spiar abstracts and combines the component- and push-based styles of these Ajax frameworks into a new style.

8.2 *Typical* AJAX *Configurations*

Many industrial frameworks have started supporting the AJAX style of interaction on the web. However, because of the multitude of these systems it is difficult to capture their commonalities and draw sharp lines between their main variations. Using SPIAR as a reference point, commonalities and divergences can be identified.

Table 3 shows a number of AJAX configurations along with the induced architectural properties. The first entry is the REST-based classic Web configuration. While simple and scalable in design, it has, a very low degree of responsiveness, high user-perceived latency, and there is a huge amount of redundant data transferred over the network.

The second configuration is the Client-centric AJAX. Most AJAX frameworks started by focusing on the client-side features. Frameworks such as Dojo, Ext [14], and jQuery [15] all provide rich UI widgets on the client, facilitating a client-centric style of interaction in which most of the functionality is off-loaded to the browser. Generally, an interaction between components that share the same location is considered to have a negligible cost when compared to interaction that is carried out through a network [12]. This variant provides a high degree of user interactivity and very low user-perceived latency. There is, however, no support for adaptability as all the code is off-loaded to the client and that makes this variant static in terms of code changes from the server.

Frameworks such as GWT, DWR [16], and JSON-RPC-Java [17] support the *Asynchronous Remote Procedure Call* (ARPC) style of interaction. In this configuration, all the presentational and functional code is off-loaded to the browser and the server is only asynchronously contacted in case of a change in terms of raw textual data. Low user-perceived latency, high user interactivity and reduced server round-trips are the characteristics of this configuration. Even though the textual data can be dynamically requested from the server, there is a limited degree of adaptability for the presentational and functional code.

The fourth configuration is a pure push-based interaction in which the state changes are streamed to the client (by keeping a connection alive), without any explicit request from the client. High level of data coherence and improved network performance compared to the traditional pull style on the web are the

[14] `http://extjs.com`
[15] `http://jquery.com`
[16] `http://getahead.org/dwr`
[17] `http://oss.metaparadigm.com/jsonrpc/`

main positive properties of this variant. High server load and scalability issues are mainly due to the fact that the server has to maintain state information about the clients and the corresponding connections.

For the sake of comparison, the last entry in Table 3 presents the component- and push-based SPIAR style itself.

8.3 Issues with push AJAX

Scalability is the main issue in a push model with a traditional server model. COMET uses persistent connections, so a TCP connection between the server and the client is kept alive until an explicit disconnect, timeout or network error. So the server has to cope with many connections if the event occurs infrequently, since it needs to have one or more threads for every client. This will bring problems on scaling to thousands of simultaneous users. There is a need for better event-based tools on the server. According to our latest findings [10], push can handle a higher number of clients if new techniques, such as the continuations [27] mechanism, are adopted by server applications. However, when the number of users increases, the reliability in receiving messages decreases.

The results of our empirical study [9, 10] show that push provides high data coherence and high network performance, but at the same time a COMET server application consumes more CPU cycles as in pull.

A stateful server is more resistant to failures, because the server can save the state at any given time and recreate it when a client comes back. A push model, however, due to its list of subscribers is less resilient to failures. The server has to keep the state, so when the state changes, it will broadcast the necessary updates. The amount of state that needs to be maintained can be large, especially for popular data items [6]. This extra cost of maintaining a state and a list of subscribers will also have a negative effect on scalability.

These scalability issues are also inherited by SPIAR as can be seen in Table 3.

8.4 Resource-based versus Component-based

The architecture of the World Wide Web [53] is based on resources identified by Uniform Resource Identifiers (URI), and on the protocols that support the interaction between agents and resources. Using a generic interface and providing identification that is common across the Web for resources has been one of the key success factors of the Web.

The nature of Web architecture which deals with Web pages as resources causes redundant data transfers [7]. The delta-communication way of interaction in SPIAR is based on the component level and does not comply with the Resource/URI constraint of the Web architecture. The question is whether this choice is justifiable. To be able to answer this question we need to take a look at the nature of interactions within single page applications: safe versus unsafe interactions.

8.5 Safe versus Unsafe Interactions

Generally, client/server interactions in a Web application can be divided into two categories of *Safe* and *Unsafe* interactions [52]. A safe interaction is one where the user is not to be held accountable for the result of the interaction, e.g., simple queries with `GET`, in which the state of the resources (on the server) is not changed. An unsafe interaction is one where a user request has the potential to change the state of the resources, such as a `POST` with parameters to change the database.

The web architecture proposes to have unique resource-based addressing (URL) for safe interactions, while the unsafe ones do not necessarily have to correspond to one. One of the issues concerning AJAX applications is that browser history and bookmarks of classic web applications are broken if not implemented specifically. In AJAX applications, where interaction becomes more and more desktop-like, where eventually *Undo/Redo* replaces *Back/Forward*, the safe interactions can remain using specific addressing while the unsafe ones (POST requests) can be carried out at the background. Both variants use delta-communication, however, the safe interactions should have unique addressing and the unsafe one do not necessarily correspond to any REST-based resource identified by a URL.

To provide the means of linking to the safe operations in AJAX, the URI's *fragment identifier* (the part after `#` in the URL) can be adopted. Interpretation of the fragment identifier is then performed by the engine that dereferences a URI to identify and represent a state of the application. Libraries such as the jQuery history/remote plugin [18] or the Really Simple History [19] support ways of programatically registering state changes with the browser history through the fragment identifier.

[18] `http://stilbuero.de/jquery/history/`
[19] `http://code.google.com/p/reallysimplehistory/`

8.6 Client- or server-side processing

Within the current frameworks it is not possible for developers to choose whether some certain functionality should be processed on the client or on the server. How the computation is distributed can be an important factor in tuning a web application. Ajax frameworks architectures should provide the means for the developer to decide if and to what extent computation should be done on the client. Also adopting adaptive techniques to choose between the server or client for processing purposes needs more attention.

8.7 Asynchronous Synchronization

The asynchronous interaction in Ajax applications may cause race conditions if not implemented with care. The user can send a request to the server before a previous one has been responded. In a server processor that handles the requests in parallel, the second request can potentially be processed before the first one. This behavior could have drastic effects on the synchronization and state of the entire application. A possible solution would be handling the event-triggered requests for each client sequentially at the cost of server performance.

8.8 Communication Protocol

As we have seen, currently each Ajax framework has implemented its own specific communication protocol. This makes the visibility of client/server interactions poor as one must know the exact protocol to be able to make sense of the delta messages. It also results in a low level of portability for these applications. For a client to be able to communicate with an Ajax server, again it needs to know the protocol of that server application. These two properties can be improved by defining a standard protocol specification for the communication by and for the Ajax community.

If we look at the current push approaches, we see different techniques on achieving the push solution itself, but also different measures to deal with portability. Without a standard here, it will be difficult for a mediator to understand the interactions between system components, therefore the system itself will be less visible. The definition and adoption of the Bayeux protocol is a first attempt in the right direction which will improve both visibility and portability.

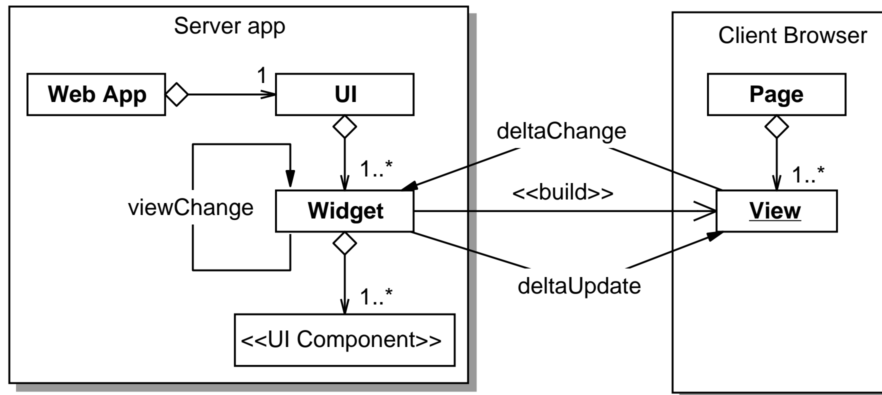

Fig. 8. A single page web application composed of UI components.

8.9 *Design Models*

Figure 8 shows a meta-model of an Ajax web application. The UI is composed of widgets of UI components. The client single page is built by the server-side widgets. Delta changes as well as view changes occur on the widget level. A view change, can be seen as navigating through the available widgets. Ajax frameworks should provide clear navigational models for developers. Research is needed to propose design models for Ajax developers by for instance extending the UML language to model user interaction, navigation through components, asynchronous/synchronous actions and client versus server side processing.

8.10 *Scope of* Spiar

The essential requirements for an Ajax application are speed of execution and improved user experience, small size of client/server data, and very specific interaction behavior. Spiar is a coordinated set of architectural constraints that attempts to minimize user-perceived latency and network usage, and improve data coherence and ultimately user experience. Because of these properties, the components of Ajax frameworks are tightly coupled. Loose coupling is thus not a property included in Spiar. This inherent tight coupling also encompasses some scalability tradeoffs.

The style focuses on the front-end of the new breed of web applications, i.e., the Service Provider is an abstract component that could be composed of middle en back-end software. Service-oriented architecture solutions could therefore easily be combined with Spiar, e.g., by replacing the Service Provider with a SOAP server. Spiar elaborates only those parts of the architecture that are considered indispensable for Ajax interaction.

9 Related Work

While the attention for rich Internet applications in general and Ajax in particular in professional magazines and Internet technology related web sites has been overwhelming, few research papers have been published on the topic so far.

A number of technical books have appeared on the subject of developing Ajax applications. Asleson and Schutta [1], for instance, focus primarily on the client side aspects of the technology and remain 'pretty agnostic' to the server side. Crane et al.[15] provide an in-depth presentation of Ajax web programming techniques and prescriptions for best practices with detailed discussions of relevant design patterns. They also mention improved user experience and reduced network latency by introducing asynchronous interactions as the main features of such applications. While these books focus mainly on the implementation issues, our work examines the architectural design decisions and properties from an abstraction level by focusing on the interactions between the different client/server components.

The push-based style has received extensive attention within the distributed systems research community. However, most of the work focuses on client/server distributed systems and non-HTTP multimedia streaming or multicasting with a single publisher [23, 26]. The only work that currently focuses on Ajax is the white-paper of Khare [29]. Khare discusses the limits of the pull approach and proposes a push-based approach for Ajax. However, the white-paper does not evaluate possible issues with this push approach, such as scalability and performance. Their work on the `mod_pubsub` event router over HTTP [30] is highly related to the concepts of Ajax push implementations.

The page-sequence model of the traditional web architecture makes it difficult to treat portions of web pages (fragments), independently. Fragment-based research [7, 11, 13] aims at providing mechanisms to efficiently assemble a web page from different parts to be able to cache the fragments. Recently proposed approaches include several server-side and cache-side mechanisms. Server-side techniques aim at reducing the load on the server by allowing reuse of previously generated content to serve user requests. Cache-based techniques attempt to reduce the latency by moving some functionality to the edge of the network. These fragment-based techniques can improve network and server performance, and user-perceived latency by allowing only the modified or new fragments to be retrieved. Although the fragments can be retrieved independently, these techniques lack the user interface component interactivity required in interactive applications. The UI component-based model of the Spiar style in conjunction with its delta-communication provides a means for a client/server interaction based on state changes that does not rely on

caching.

The SPIAR style itself draws from many existing styles [31, 39, 46, 49] and software fields [20, 35, 42], discussed and referenced in the paper. Our work relates closely to the software engineering principles of the REST style [21]. While REST deals with the architecture of the Web [53] as a whole, SPIAR focuses on the specific architectural decisions of AJAX frameworks.

Parsons [41] provides an overview of the current state of the web by exploring the evolving web architectural patterns. After the literature on the core patterns of traditional web application architectures is presented, the paper discusses some new emerging patterns, by focusing on the recent literature on WEB 2.0 in general and AJAX in particular.

On the architectural styles front the following styles can be summarized: Pace [48] an event- based architectural style for trust management in decentralized applications, TIGRA [17] a distributed system style for integrating front-office systems with middle- and back-office applications, and Aura [47] an architectural framework for user mobility in ubiquitous environments which uses models of user tasks as first class entities to set up, monitor and adapt computing environments.

Khare and Taylor [31] also evaluate and extend REST for decentralized settings and represent an event-based architectural style called ARRESTED. The asynchronous extension of REST, called A+REST, permits a server to broadcast notifications of its state changes to 'watchers'.

Recently, Erenkrantz et al. [18] have re-evaluated the REST style for new emerging web architectures. They have also come to the conclusion that REST is silent on the area that AJAX expands. They recognize the importance of the AJAX engine which is seen as the interpretation environment for delivered content. They also notice, that REST's goal was to reduce server-side state load, while AJAX reduces server-side computational load by adopting client-side processing, and increases responsivity. Their new style extends REST and is called Computational REST (CREST). CREST requires a transparent exchange of computation so that the client no longer is seen as merely a presentation agent for delivered content; 'it is now an execution environment explicitly supporting computation'. In other words, CREST much like SPIAR recognizes the significance of the AJAX engine as a processing component. On the other hand, CREST ignores other important architectural characteristics of AJAX applications, such as the the delta-communication and asynchronous interaction covered in SPIAR.

10 Concluding Remarks

Ajax is a promising solution for the design and implementation of responsive rich web applications, since it overcomes many of the limitations of the classical client-server approach. However, most efforts in this field have been focused on the implementation of different Ajax tools and frameworks, with little attention to the formulation of a conceptual architecture for the technology.

In this paper we have discussed Spiar, an architectural style for Ajax. The contributions of this paper are in two research fields: web application development and software architecture

From a software architecture perspective, our contribution consists of the use of concepts and methodologies obtained from software architecture research in the setting of Ajax web applications. Our paper further illustrates how the architectural concepts such as properties, constraints, and different types of architectural elements can help to organize and understand a complex and dynamic field such as single page Ajax development. In order to do this, our paper builds upon the foundations offered by the Rest style, and offers a further analysis of this style for the purpose of building web applications with rich user interactivity.

From a web engineering perspective, our contribution consists of an evaluation of different variants of Ajax client/server interactions, the Spiar style itself, which captures the guiding software engineering principles that practitioners can use when constructing and analyzing Ajax applications and evaluating the tradeoffs of different properties of the architecture. We further propose a component- push-based architecture capable of synchronizing the events both on the server and the client efficiently.

The style is based on an analysis of various Ajax frameworks and configurations, and we have used it to address various design tradeoffs and open issues in Ajax applications.

Ajax development field is young, dynamic and changing rapidly. Certainly, the work presented in this paper needs to be incrementally enriched and revised, taking into account experiences, results, and innovations as they emerge from the web community.

Future work encompasses the use of Spiar to analyze and influence Ajax developments. One route we foresee is the extension of Spiar to incorporate additional models for representing, e.g., navigation or UI components, thus making it possible to adopt a model-driven approach to Ajax development. At the time of writing, we are using Spiar in the context of enriching existing web applications with Ajax capabilities.

Acknowledgments We thank Engin Bozdag (TU Delft) for his feedback on our paper, particularly for his help on the push-based extension of the style.

References

[1] R. Asleson and N. T. Schutta. *Foundations of Ajax*. Apress, 2005.
[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33, 2004.
[3] D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise. A framework for event-based software integration. *ACM Trans. Softw. Eng. Methodol.*, 5(4):378–421, 1996.
[4] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice, 2nd ed.* Addison-Wesley, 2003.
[5] T. Berners-Lee, L. Masinter, and M. McCahill. *RFC 1738: Uniform Resource Locators (URL)*, 1994.
[6] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Disseminating dynamic web data. *IEEE Trans. Comput.*, 51(6):652–668, 2002.
[7] C. Bouras and A. Konidaris. Estimating and eliminating redundant data transfers over the Web: a fragment based approach: Research articles. *Int. J. Commun. Syst.*, 18(2):119–142, 2005.
[8] E. Bozdag. Integration of HTTP push with a JSF Ajax framework. Master's thesis, Delft University of Technology, December 2007.
[9] E. Bozdag, A. Mesbah, and A. van Deursen. A comparison of push and pull techniques for Ajax. In *Proceedings of the 9th IEEE International Symposium on Web Site Evolution (WSE'07)*, pages 15–22. IEEE Computer Society, 2007.
[10] E. Bozdag, A. Mesbah, and A. van Deursen. Performance testing of data delivery techniques for Ajax applications. Technical Report TUD-SERG-2008-009, Delft University of Technology, 2008.
[11] D. Brodie, A. Gupta, and W. Shi. Accelerating dynamic web content delivery using keyword-based fragment detection. *J. Web Eng.*, 4(1):079–099, 2005.
[12] A. Carzaniga, G. P. Picco, and G. Vigna. Designing distributed applications with mobile code paradigms. In *ICSE '97: 19th International Conference on Software Engineering*, pages 22–32. ACM Press, 1997.
[13] J. Challenger, P. Dantzig, A. Iyengar, and K. Witting. A Fragment-based approach for efficiently creating dynamic Web content. *ACM Trans. Inter. Tech.*, 5(2):359–389, 2005.
[14] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[15] D. Crane, E. Pascarello, and D. James. *Ajax in Action*. Manning Publications Co., 2005.

[16] Direct Web Remoting. Reverse Ajax documentation. `http://getahead.org/dwr/reverse-ajax`, 2007.

[17] W. Emmerich, E. Ellmer, and H. Fieglein. TIGRA an architectural style for enterprise application integration. In *ICSE '01: 23rd International Conference on Software Engineering*, pages 567–576. IEEE Computer Society, 2001.

[18] J. R. Erenkrantz, M. Gorlick, G. Suryanarayana, and R. N. Taylor. From representations to computations: the evolution of web architectures. In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE'07)*, pages 255–264. ACM, 2007.

[19] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[20] R. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, UC, Irvine, Information and Computer Science, 2000.

[21] R. Fielding and R. N. Taylor. Principled design of the modern Web architecture. *ACM Trans. Inter. Tech. (TOIT)*, 2(2):115–150, 2002.

[22] E. Folmer. *Software Architecture analysis of Usability*. PhD thesis, Univ. of Groningen, Mathematics and Computer Science, 2005.

[23] M. Franklin and S. Zdonik. data in your face: push technology in perspective. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 516–519. ACM Press, 1998.

[24] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Trans. Softw. Eng.*, 24(5):342–361, 1998.

[25] J. Garrett. Ajax: A new approach to web applications. Adaptive path, 2005. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

[26] M. Hauswirth and M. Jazayeri. A component and communication model for push systems. In *7th European Software Engineering Conference (ESEC/FSE-7)*, pages 20–38. Springer-Verlag, 1999.

[27] Jetty. Jetty webserver documentation - continuations. Mortbay Consulting, `http://docs.codehaus.org/display/JETTY/Continuations`, 2006.

[28] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *4th IEEE International Conference on Engineering of Complex Computer Systems*, pages 68–78. IEEE Computer Society, 1998.

[29] R. Khare. Beyond Ajax: Accelerating web applications with Real-Time event notification. `http://www.knownow.com/products/docs/whitepapers/KN-Beyond-AJAX.pdf`, 2005.

[30] R. Khare, A. Rifkin, K. Sitaker, and B. Sittler. mod_pubsub: an open-source event router for Apache, 2002.

[31] R. Khare and R. N. Taylor. Extending the Representational State Transfer (REST) architectural style for decentralized systems. In *ICSE '04: 26th International Conference on Software Engineering*, pages 428–437. IEEE Computer Society, 2004.

[32] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Program*, 1(3):26–49, 1988.

[33] A. Mesbah and A. van Deursen. An architectural style for Ajax. In *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, pages 44–53. IEEE Computer Society, 2007.

[34] A. Mesbah and A. van Deursen. Migrating multi-page web applications to single-page Ajax interfaces. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 181–190. IEEE Computer Society, 2007.

[35] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *ACM SIGCOMM Conf. on Applications, technologies, architectures, and protocols for computer communication*, pages 181–194. ACM, 1997.

[36] R. T. Monroe and D. Garlan. Style-based reuse for software architectures. In *ICSR '96: 4th International Conference on Software Reuse*, pages 84–93. IEEE Computer Society, 1996.

[37] M. Naaman, H. Garcia-Molina, and A. Paepcke. Evaluation of ESI and class-based delta encoding. In *8th International Workshop Web content caching and distribution*, pages 323–343. Kluwer Academic Publishers, 2004.

[38] Netscape. An exploration of dynamic documents. `http://wp.netscape.com/assist/net_sites/pushpull.html`, 1996.

[39] W. M. Newman and R. F. Sproull. *Principles of Interactive Computer Graphics*. McGraw-Hill, 1979. 2nd Edition.

[40] J. Offutt. Quality attributes of web software applications. *IEEE Softw.*, 19(2):25–32, 2002.

[41] D. Parsons. Evolving architectural patterns for web applications. In *Proceedings of the 11th Pacific Asia Conference on Information Systems (PACIS)*, pages 120–126, 2007.

[42] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[43] D. S. Rosenblum and A. L. Wolf. A design framework for internet-scale event observation and notification. In *ESEC/FSE '97: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 344–360. Springer-Verlag New York, Inc., 1997.

[44] A. Russell. Comet: Low latency data for the browser. `http://alex.dojotoolkit.org/?p=545`, 2006.

[45] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[46] A. Sinha. Client-server computing. *Communications of the ACM*, 35(7):77–98, 1992.

[47] J. P. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *WICSA 3: IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 29–43. Kluwer, B.V., 2002.

[48] G. Suryanarayana, J. R. Erenkrantz, S. A. Hendrickson, and R. N. Taylor. PACE: An architectural style for trust management in decentralized applications. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, page 221. IEEE Computer Society, 2004.

[49] R. N. Taylor, N. Medvidovic, K. M. Anderson, J. E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Trans. Softw. Eng.*, 22(6):390–406, 1996.

[50] H.-H. Teo, L.-B. Oh, C. Liu, and K.-K. Wei. An empirical study of the effects of interactivity on web user attitude. *Int. J. Hum.-Comput. Stud.*, 58(3):281–305, 2003.

[51] A. Umar. *Object-oriented client/server Internet environments*. Prentice Hall Press, 1997.

[52] W3C. URIs, Addressability, and the use of HTTP GET and POST, Mar. 21 2004. W3C Tag Finding.

[53] W3C Technical Architecture Group. Architecture of the World Wide Web, Volume One, Dec. 15, 2004. W3C Recommendation.

TUD-SERG-2008-013
ISSN 1872-5392