SPIAR: An architectural style for single page internet applications

A. Mesbah, K. Broenink, A. van Deursen

REPORT SEN-R0603 APRIL 2006

ISSN 1386-369X

SPIAR: An architectural style for single page internet applications

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. In this paper, we examine the architectural properties of such applications and introduce the SPIAR architectural style. We describe the guiding software engineering principles and the constraints chosen to induce the desired properties. The style emphasizes single page interfaces, user interface component development, and intermediary delta-communication between client/server components, to improve user interactivity and reduce user-perceived latency. In addition, SPIAR is used as an abstract model to present and discuss the architecture of Backbase, one of the most mature single page Internet application development frameworks available to date.

2000 Mathematics Subject Classification: 2000
1998 ACM Computing Classification System: C.2.4 D.2.11
Keywords and Phrases: web engineering; software architecture; architectural style; single page interaction; web application
Note: This work was carried out under project SEN1 - Single Page Computer Interaction

SPIAR: An Architectural Style for Single Page Internet Applications

Ali Mesbah
Delft University of Technology
& CWI
The Netherlands
A.Mesbah@ewi.tudelft.nl

Kees Broenink
Backbase
The Netherlands
Kees.Broenink@backbase.com

Arie van Deursen
Delft University of Technology
& CWI
The Netherlands
Arie.van.Deursen@cwi.nl

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. In this paper, we examine the architectural properties of such applications and introduce the SPIAR architectural style. We describe the guiding software engineering principles and the constraints chosen to induce the desired properties. The style emphasizes single page interfaces, user interface component development, and intermediary delta-communication between client/server components, to improve user interactivity and reduce user-perceived latency. In addition, SPIAR is used as an abstract model to present and discuss the architecture of Backbase, one of the most mature single page Internet application development frameworks available to date.

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 the user interactivity has been lost. Classical web applications are based on a multi page interface (MPI) model, in which interactions are based on a page-sequence paradigm. While simple and elegant in design for exchanging documents, the MPI model has many limitations for developing modern web applications with desirable human-computer interaction.

Recently, there has been a shift in the direction of web development. A new breed of web application, dubbed AJAX (Asynchronous JavaScript And XML) [11], is emerging in response to the limited degree of interactivity in large-grain stateless Web interactions. At the heart of this new approach lies a single page interface (SPI) model that facilitates rich interactivity. In SPI, changes are made to individual components that compose the interface, as opposed to refreshing the entire page.

The current efforts in this field are mainly focusing on the richness of the client. Implications to the design and implementation within this model is a research area that needs more attention.

Software architecture research investigates how system components are identified, how information is communicated, how elements of a system can evolve independently, and how all of the above can be described in comprehensive notations in order to understand and evaluate the impact of design choices [9]. A set of architectural constraints inducing the architectural properties desired of a system, when given a name, becomes an architectural style.

The objective of this paper is to come up with an architectural style for single page Internet applications. Such a style should make it possible to describe the design of particular single page frameworks concisely, to assess the impact of modifications to these frameworks, and should help to compare properties of these frameworks.

To arrive at such a style, this paper first of all answers the question what architectural properties a single page Internet application should comply with (Section 2). Secondly, the paper investigates which architectural constraints, some new, some taken from existing styles, induce these properties (Section 3). The resulting properties and constraints lead to SPIAR, our Single Page Internet Application aRchitecture.

To evaluate the resulting style, we use SPIAR to present and discuss the architecture of the Backbase[1] development framework, one of today's leading commercial AJAX products. Furthermore, we use the style to discuss various tradeoffs involved in the design of single page applications in Section 5.

The paper concludes with a brief summary of related work, a list of key contributions, and an outlook to future research.

2. SINGLE PAGE APPLICATIONS

Generally, web applications have client, application server and back-end server components, which suggests a three tiered architecture [23]. Our work focuses on the client tier and the portion of the middle tier that communicates with the client.

We define a *Single Page Internet Application* (SPIA) as a web application based on the single page interface model consisting of a single page web client and a server application. The client code runs in a universal client, providing the user with an interactive standard-based single page user interface. This universal client is the web browser and the various standards that allow browsers to run on almost any computing device. Contrary to a classic server application, a single page server application does not process a whole page for responding to each request. The interaction is based merely on state changes.

SPIA can be seen as a hybrid of web- and desktop-applications, inheriting characteristics from both worlds.

The architectural properties of a software architecture include

[1] `www.backbase.com`

both the functional properties achieved by the system and non-functional properties, often referred to as quality attributes [2]. We will only focus on the architectural properties which we believe are of great importance for SPIA. Some of these properties are related; for instance, user interactivity is influenced by user-perceived latency, which in turn is affected by network and server performance.

User Interactivity
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* [10], the term used in software architecture literature. Teo *et al.* [22] 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.

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 and second, by allowing the user to interact asynchronously with the system.

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 granularity of transmitted data.

Server Performance
Server performance represents the elapsed time for the server to respond to a user request, with the exclusion of network delay. It is the period between the moment a server receives the request till the moment a response can be sent back to the user. One way to improve server performance is by reducing the amount of needed processing (workload) for user requests on the server. Another approach is decreasing the number of requests from the clients which results in an increase in the amount of resources on the server to handle the remaining requests.

Development Effort
Development effort is defined as the total number of hours that is needed to design, implement, analyze, and repair a web application. Development effort is an important factor for the usage and acceptability of any new approach.

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

3. SPIAR

Fielding [9] defines a software architecture as a configuration of architectural elements - components, connectors, and data - constrained in their relationships in order to achieve a desired set of architectural properties. An architectural style is in turn defined as 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.

3.1 Motivation

Many different network-based architectural styles, such as client-server [20] and n-tier [23], exist but perhaps the most complete and appropriate style for the Web is the REpresentational State Transfer (REST) [8]. Single page architectures, however, are not so easily captured in REST, due to the following differences:

- While REST is suited for large-grain hypermedia data transfer, it is not optimal for small data interactions required in single page applications.
- REST focuses on a hyper-linked resource-based interaction in which the client requests a specific resource. In contrast, in single page 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. Single page applications, however, require a model for asynchronous communication. REST explicitly constrains the server to be stateless. While this constraint can improve scalability, the tradeoffs with respect to network performance and user interactivity are of greater importance when designing a single page architecture.

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

3.2 Architectural Style

In this paper, we use the framework and terminology of Fielding [9] which is based on the work of Perry and Wolf [19]. We believe the framework is an appropriate approach for understanding and describing the architecture through architectural styles. This choice also makes a comparison of our architectural style with other styles such as REST more convenient.

The SPIAR (Single Page Internet Application aRchitecture) architectural style is an abstraction of the architectural elements within single page Internet applications. The style consists of a set of architectural constraints some of which are derived from common architectural styles. Table 1 presents an overview of the constraints and induced properties.

Single Page Interface
SPIAR is based on the *client-server* [20] 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 the SPIAR style is its emphasis on a *single page interface*. 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.

Asynchronous Interaction
The client-server interaction can be realized in both a push- or pull-based style. In a push-based style [12], the server broadcasts the state changes to the clients asynchronously every time its state changes. Event-based Integration [1] and Asynchronous REST [13] are event-based styles allowing asynchronous notification of state changes by the server. The current HTTP protocol [7], however, does not allow the implementation of this style and it is only supported in peer-to-peer architectural environments.

In a pull-based style, client components actively request state changes. Event-driven [18] architecture is a pull-based style. Event

Table 1: Constraints and induced properties

	User Interactivity	User-perceived Latency	Network Performance	Server Performance	Development Effort	Portability
Single Page Interface	✓					
Asynchronous Interaction	✓	✓				
Delta Communication	✓	✓	✓	✓		
Client-side processing	✓	✓		✓		
UI Component-based	✓				✓	
Web standard-based					✓	✓

connectors 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.

The SPIAR interaction is designed to have a high user interactivity and a low user-perceived latency. In an *asynchronous interaction* the user can, 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.

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 [3], have been adopted in order to reduce data redundancy. Delta-encoding [16] uses caching techniques to reduce network traffic, however, it does not reduce the computational load since the server generates the entire page for each request [17].

SPIAR uses a *delta-communication* style of interaction, in which 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 uses delta-encoding architectural principles but is different i.e., delta-communication does not rely on caching and as a result, the server and client only need to process the deltas.

This constraint induces the properties of network and server 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. Server performance is improved because the load per request is reduced.

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.

User Interface Component-based

SPIAR relies on a user interface (UI) component model 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. SPIAR's component model is similar to C2 [21], a component-based style that relies on asynchronous notification of state changes and request messages that handle component interactions.

UI component programming improves development effort because developers can use reusable components to assemble a Web page. User interactivity is improved because the user can interact with the application on a component level, similar to desktop applications.

Web standard-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.

3.3 Architectural Elements

Inspired by [9, 19] the key architectural elements of SPIAR are divided into three categories namely processing, data, and connecting elements.

Processing Elements

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

The *Client Browser* offers a set of standards such as HTTP, HTML, Cascading Style Sheets (CSS), JavaScript and Document Object Model (DOM)[2]. It processes the representational model to produce the user interface. The user interface is called *Single Page User Interface (*SPUI*)* which is the point of presentation to and interaction with the user. 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 immediately without requiring a page refresh.

The *Single Page Engine (*SPENGINE*)* is a zero-install engine that loads and runs in the client browser. The engine is responsible for the initialization and manipulation of the representational model. It 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 origin server and operates by accepting HTTP-based requests from the network, and providing responses to the requester. All server-side functionality resides in server application.

The *Server Processor* 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 processor's function is invoked by event listeners, attached to server-side 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. Although, the protocol might be specific, the implementation details are hidden behind this interface.

Data Elements

The data elements contain the information that is used and transformed by the processing elements. The key abstraction in SPIAR is a UI component. Each component has a unique persistent identifier on both the client and server while having a different representation

[2] `http://www.w3.org/DOM/`

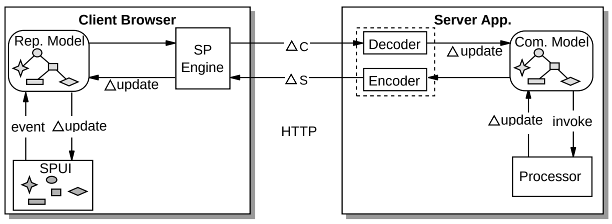

Figure 1: Processing View of an SPIAR-based architecture.

on each side. SPIAR allows the client and server to synchronize state using the unique identifier of the components.

The *Single Page Language (*SPL*)* is a domain specific language that can be utilized in a declarative way to define the structure of a single page interface. SPL can be seen as a library of identifiable widgets. The server can send its state changes enclosed in this language which will be interpreted by a rendering engine on the client. Note that the components created in the language form the data elements and not the language itself. XUL[3], XAML[4] and BXML[5] are all examples of declarative user interface languages. This element decreases development effort, once the architecture is in place, which leads to a shorter time to market. It also improves user interactivity by providing widgets on a single page interface.

The *Representational Model* is a run-time abstraction of how a SPUI is represented on the client. Manipulating this representation results in a direct (visible) change on the user interface by the client browser.

The *Component Model* consists of a set of server-side UI components. The component model on the server conceptually resembles the representational model on the client, although the implementations will be different. Each server-side component contains the data and behavior of that part of the corresponding client-side widget which is relevant for state changes; client-side functionality and visual effects that do not affect the state on the server are not part of the server-side component model.

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 can be in any desired format e.g., XML, JSON, XOXO.

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.

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.

[3] XML User Interface Language (XUL) 1.0, http://www.mozilla.org/projects/xul/

[4] Extensible Application Markup Language (XAML), http://www.xaml.net

[5] Backbase Extensible Markup Language (BXML), http://www.backbase.com

Table 2: Processing Elements

SPIAR	Backbase
SPUI	Single page with widgets
Client browser	Mozilla, Firefox, IE, Opera, Safari
SPENGINE	Backbase Presentation Client (BPC)
Server application	Backbase Java Server (BJS)
Server processor	JavaServer Faces lifecycle, Backing Beans
Delta encoder/decoder	Client: request encoder/response decoder Server: component renderer/request decoder

3.4 Architectural View

Given the processing, data, and connecting elements, we can use different architectural views to describe how the elements work together to form an architecture. An architecture can be viewed from various perspectives. A processing view, for instance, concentrates on the data flow and some aspects of the connections among the processing elements with respect to the data [9].

Figure 1 depicts the processing view of an SPIAR-based architecture. 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 SPUI widgets fires off an event to indicate some kind of component-defined action which is delegated to the 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, identify and notify the relevant components in the component model. The changed components will ultimately invoke the event listeners of the processor. The processor, 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 SPUI. The engine has also the ability to update the representational model directly after an event, if no round-trip to the server is required.

4. BACKBASE

Backbase is an Amsterdam based company specialized in rich Internet applications. It was established in 2003, more than two years before the term "AJAX" was coined by Garret [11]. Backbase presently employs approximately 40 people, half of which work on research and development. The company has won several innovation awards, been profitable since inception, and its clients include numerous worldwide companies.

Backbase offers an AJAX development framework, developer tools, and server side support. A key element of the framework is the Backbase Presentation Client, a standards-based, ultralight engine that can be programmed via a declarative user interface language.

At the time of writing, Backbase is in the process of releasing its new Backbase Java Server, which together with the presentation client covers the full SPIA chain. In this section, we will use SPIAR to discuss the key design decisions in the Backbase architecture, and assess which tradeoffs were made in its design.

4.1 Processing Elements

The main processing elements of the Backbase framework are presented in Table 2 against the processing elements of the SPIAR style. Backbase provides a cross-browser and platform-neutral clie-

Table 3: Data Elements

SPIAR	Backbase
Single Page Language (SPL)	BXML
Representational model	BXML and DOM
Component model	BJS components
Delta communicating data	DELTA-CLIENT (POST properties) DELTA-SERVER (BXML message)

nt with a SPUI, supporting the majority of modern *client browsers* (e.g., Firefox 1+, IE 5+) by utilizing Web standards.

It has an engine called *Backbase Presentation Client (*BPC*)*, written in JavaScript, corresponding to SPIAR's SPENGINE element. BPC's main functionality can be summarized as creating a *single page interface* with widgets by parsing and interpreting the user interface language (BXML), delta-communication with the server and asynchronous interaction with the user through the manipulation of the representational model. It also includes many features such as XSLT and XPath utilization for dynamic data interchange and data manipulation.

The server application comes in two variants, namely .Net and Java, both capable of interacting with BPC. We will focus on Backbase Java Server (BJS) which is built on top of JavaServer Faces (JSF)[6], the new J2EE presentation architecture. JSF provides a user interface component-based framework following the model-view-controller pattern. The interaction in JSF is based on the classic page-sequence model.

BJS provides its own set of UI components and extends the JSF framework to provide a single page interface implementation. BJS utilizes all standard JSF mechanisms such as validation, conversion and event processing through the JSF life-cycle phases. Any Java class that offers getters and setters for its properties can be directly assigned to a UI component property. Classes bound to UI component properties and events are called Backing Beans. Furthermore, the method that should be invoked on a certain event can be defined using the J2EE Expression Language (EL). The EL makes it possible to easily access application data stored in objects from server-side scripting languages. Within a Backing Bean the developer has full control over all available UI components. BJS tracks all changes and sends them as DELTA-SERVER back to the BPC.

Backbase has defined a client/server protocol that allows for synchronization of components (e.g., input field values, attributes, style and structural changes) through decoding and encoding elements. Every component has the ability to tell the BPC which changes should be tracked and reported to the server. The encoded DELTA-CLIENT will be posted to the server on certain defined events. These can be action events like clicking a button, or value change events such as checking a radio button. The server-side decoder translates the DELTA-CLIENT and identifies the corresponding component(s) in the BJS component tree. The encoder renders a DELTA-SERVER of the changes to be responded to BPC.

4.2 Data Elements

Table 3 presents the Backbase data elements. Backbase has developed a declarative *single page language* called Backbase Extensible Markup Language (BXML) which relieves developers from having to code rich effects in client-side scripting languages. BXML is an extension to HTML and is interpreted on the client by the BPC. It offers support for user interface declaration, such as widgets (e.g., windows, menus, decks, tabs) and their behaviors (e.g., open, close, collapse, drag-drop) and can be used in combination with other W3C standards such as CSS.

The representational model of BPC is constructed by utilizing *DOM*, which is a platform and language neutral standard model, with a set of objects for representing HTML and XML documents, a model of how these objects can be combined, and an interface for accessing and manipulating them. It is used by the engine to dynamically access and update content, structure and style of the SPUI; BXML is parsed and transformed to DOM.

On the server, BJS *Components* are a set of well-defined UI components. Each component resembles a client-side widget and is capable of rendering the corresponding BXML code; i.e., there is a one-to-one relation between the component on the server and the one on the client for components that need state synchronization. The client can also consist of visual widgets that do not take part in the state management of the application. For such widgets there is no need for server-side components.

Backbase delta communicating data are the synchronizing medium between the client and server. Backbase uses HTTP POST properties for the DELTA-CLIENT:

```
clientDelta:[evt=COMPONENT-ID|EVENT-NAME|EVENT-TYPE]?
  |[att=COMPONENT-ID|ATTRIBUTE-NAME|ATTRIBUTE-VALUE]+
```

where clientDelta is the name of the HTTP parameter with its value being a list of items. There are two types of items: the event item captures the data about the event triggered on the component, where as, an attribute item encapsulates information about the state changes of the client component itself. Each attribute item has three parts, namely, the unique identifier of the component (`COMPONENT-ID`), the name (`ATTRIBUTE-NAME`) and the new value (`ATTRIBUTE-VALUE`) of the affected attribute (e.g., `[att=city1|value|amsterdam]`). Similarly, the event item contains the identifier of the component, the event name and the type of the event.

The DELTA-SERVER is in BXML format. All the BXML elements and functions can be accessed through the `b` and `s` namespaces.

4.3 Connecting Elements

The connecting elements provide an interface for interaction, enhancing simplicity and clean separation of resources and communication mechanisms. The connecting elements can be seen in Table 4.

Backbase uses *DOM events* to delegate user actions to the BPC 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 *delta connectors* connect BPC to BJS through a HTTP request and response in BXML format as presented in the previous subsection. Each request contains the necessary information (about the components changed on the client) for the server to carry out the actions on the server-side components and respond. The delta connectors serve to reduce user-perceived latency by increasing the network performance.

Delta updates are local connectors that can be seen as procedural invocations updating the state of components. BPC is connected to DOM, for instance, through JavaScript functions and on the server, the components are updated by the decoder and processor through method calls.

4.4 Processing View

The Backbase Java Server Edition ships with a demo application to manage contacts, similar to e.g., Outlook contacts functionality by implementing the Versit Vcard[7] specification. For all practical

[6] JavaServer Faces Specifi cation v1.1, http://java.sun.com/j2ee/javaserverfaces/

[7] The Electronic Business Card, version 2.1, http://www.imc.org/pdi/vcard-21.rtf

Table 4: Connecting Elements

SPIAR	Backbase
Events	DOM events
Delta connectors	DELTA-CLIENT, DELTA-SERVER
Delta updates	procedural invocation passing parameters

purposes, we focus on a simple input component (email) to present the processing view of this demo application. Note that we are only presenting the code concerning the input component which is only a small part of the whole application.

On the server, Backbase provides the `bjs` tag library to create the needed components for a user to provide the email address of a contact:

```
<bjs:inputText id="email" required="true"
 value="#{contact.email}">
 <f:validator validatorId="Email"/>
 <f:valueChangeListener
  type="DefaultValueChangeListener"/>
</bjs:inputText>
<bjs:message id="message-email" for="email"
 errorClass="error"/>
```

On the initial request of the user, this code initializes the component tree for the two components (inputText and message) each having a unique identifier. The value of the input component is *required* not to be empty. There is also an email validator connected to the component to check the email address. Because we want the email to be checked separately as a component (while it is being filled in), a `valueChangeListener` is also registered on the component. After initialization, these two components are rendered to generate the following code to be transmitted to the client:

```
<input id="email" b:attsync="value">
 <s:event b:on="change">
  <s:task b:action="settext"
   b:target="id('message-email')" b:value=""/>
  <s:task b:action="serversync" b:event="change" />
 </s:event>
</input>
<span id="message-email"/>
```

The client code represents an input widget with a placeholder for eventual server-side notification messages. The `b:attsync` attribute indicates that the server-side component is interested in any *value* change of this widget for synchronization. It is also possible for the server component to register its interest for other attribute changes (e.g., style) as well. The `s:event` element commands the engine to reset the value of the message widget on any value change of the input. The `serversync` action orders the BPC to synchronize the changes with the server.

Now imagine that the user inserts an invalid email address e.g., `sara@nl`. The engine then automatically creates the following DELTA-CLIENT code to be submitted as a HTTP parameter to the server:

```
[att=email|value|sara@nl][evt=email|b:event|change]
```

As it can be seen, the DELTA-CLIENT indicates that the value of the email component has changed. Upon arrival at the server, the decoder will translate the message and notify the corresponding server-side component with the new value. The change in the value of the component goes through the validation process and since the value is not a valid email address, the violation causes a modification of the message component with an explanatory message. Since the message component has changed, the encoder, renders it and sends the following DELTA-SERVER to the client engine:

```
<b:serverdelta
 xmlns:s='http://www.backbase.com/s'
 xmlns:b='http://www.backbase.com/b'>
 <s:execute>
  <s:task b:action="settext"
   b:target="id('message-email')"
   b:value="The given value (sara@nl) is
   not a correct email-address."/>
  <s:with b:target="id('message-email')">
   <s:setatt class="error"/>
  </s:with>
 </s:execute>
</b:serverdelta>
```

Back on the client, this DELTA-SERVER is executed by the BPC to make an in-place update of the message-widget's value and style sheet class.

4.5 Evaluation

Backbase uses an event-driven, component-based programming model which played an important role in success and acceptability of desktop applications. This model is believed to be easy to learn and develop, and the most convenient way to handle interactive user interfaces to date.

The Backbase architecture uses a well defined protocol for small interactions among known components. Data needed to be transferred over the network is significantly reduced in the architecture using delta-communication. This can result in faster response data transfers. On the other hand, number of requests sent to the server can increase if the system is not designed well. Backbase takes advantage of client-side processing by utilizing its engine and SPL functionality. While client-side processing reduces roundtrips, client performance can become an issue if too many widgets are used.

The server application does not need to render whole pages, hence number of CPU cycles needed for each request can be reduced. Note, however, that because of the way the component tree is handled in JSF, the full component tree is traversed multiple times for each request.

The client is portable because of the Web standards-based approach. Note that, although the BXML language is not a standard itself, it is parsed to DOM (a standard) and processed by the engine using JavaScript, a language supported by all modern browsers.

5. DISCUSSION

In this section we discuss the various decisions and tradeoffs made in SPIAR by placing it in relation to other approaches and evaluate the overall architecture.

Resource-based versus Component-based
The architecture of the World Wide Web [25] 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 [3]. 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.

Generally, client/server interactions in a Web application can be divided into two categories of *Safe* and *Unsafe* interactions [24]. A safe interaction is one where the user is not to be held accountable for the result of the interaction e.g., simple queries. An unsafe interaction is one where a user request has the potential to change the state of the resource.

In single page Internet applications, where interaction becomes more and more desktop-like, where eventually Undo/Redo replaces Back/Forward, the safe interactions remain using URIs while the unsafe ones can be *safely* carried out at the background using delta-communication in which neither the data transmitted nor the data received in the response necessarily correspond to any resource identified by a URI. This implies the engine should also provide the means of linking to safe operations as well as hyper-linked documents. The URI's *fragment identifier* can be used for this purpose. Interpretation of the fragment identifier is then performed by the engine that dereferences a URI to identify and represent a state of the application.

Stateless versus 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 [8]. 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.

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.

Asynchronous Synchronization

The asynchronous interaction constraint of SPIAR 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.

Separation of Concerns

SPIAR emphasizes the possibility of client-side processing to improve responsiveness and latency. This could result in a tendency to replicate business logic to the client, which in a distributed client/server application would increase the cost of development and maintenance.

In SPIAR, however, the user interface can be deployed from the server to the client browser, removing the need to install the client separately. This typically entails defining the user interface in one location, e.g., in a server-side scripting language, and generating the logic for both the client and server from a single definition.

6. 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. As far as we know, the closest appears to be the work by Mac-Vicar *et al.* [14], who present a simple model based on the model-view-controller pattern for extensibility of rich clients using a plugin-based approach.

The SPIAR style itself draws from many existing styles [20, 18, 21] and software fields [19, 8, 16], discussed and referenced in the paper. Our work relates closely to the software engineering principles of the REST style [9] which focuses on the architecture of the Web [25] as a whole.

Fragment-based Approach

The page-sequence model of the Web makes it difficult to treat portions of Web pages (fragments), independently. Fragment-based research [5, 4, 3] 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-side 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.

Applicability

Lately, a number of frameworks have been emerging that aim at making the development of single page applications easier. In this section, we briefly discuss three of these frameworks. The SPIAR style can be applied to such frameworks not only for their evaluation, but also for analzying the presence or absence of desirable properties.

ICEFaces[8] is based on JSF, which is also used in the Backbase framework. ICEFaces utilizes the JSF component model but lacks a *Single Page Language* (SPL) and client-side components. Instead, any state change on the server is directly rendered to DOM and transmitted as DELTA-SERVER to its SPENGINE. As mentioned before, because of the way the component tree is handled in JSF (and its natural page-sequence character) not much server performance improvement can be gained when relying on JSF. Furthermore, following the SPIAR principles, it can be seen that ICEFaces does not support client-side processing as a result of its Direct-to-DOM approach. Looking at Table 1, we can conjecture that ICEFaces does not benefit from the advantages of improvements in user interactivity, user-perceived latency and server performance induced by client-side processing.

Echo2[9] has an event-driven server application which monitors the state of the component model. The component model consists of independent components which complies with the UI component constraint of SPIAR. It uses delta connectors for the client/server communication. Echo2 does not provide a SPL; state changes on the server-side components are rendered to HTML elements which are processed by its SPENGINE to update the interface. Further, all the visual effects are implemented in the components and hidden from the developer.

Another interesting framework to look at is ZK[10]. ZK provides a set of XUL-based components and a markup language called ZUML for describing the SPUI. It includes a SPENGINE, and relies on delta connectors. ZK does not, however, produce client-side

[8] ICEFaces 0.3.0, http://www.icesoft.com

[9] Echo2 2.0.0, http://www.nextapp.com/platform/echo2/echo/

[10] ZK 1.0.0, http://zk1.sourceforge.net

XUL widgets which means that this framework, like the other two, lacks a client-side SPL.

It is worth noting that the three frameworks conform to the single page interface, delta-communication, asynchronous interaction, UI component, and Web standards-based constraints of the SPIAR style. Client-side processing is the only constraint not met by the frameworks. The main differences between the frameworks remain in the set of architectural elements they provide; nevertheless, they can all be described and evaluated by SPIAR, because of their common architectural interaction characteristics.

7. CONCLUDING REMARKS

We consider the following as our main contributions:

1. We offer a definition of single page Internet applications and an analysis of their key architectural properties.
2. We propose SPIAR, an architectural style consisting of constraints that induce these properties.
3. We present and evaluate the architecture of the Backbase development framework, one of today's leading AJAX products, and illustrate how the concepts of SPIAR recur in the Backbase architecture.
4. We demonstrate how SPIAR can be used to discuss design tradeoffs in single page architectures, and use SPIAR to compare different existing frameworks.

Future work encompasses the in-depth application of SPIAR to other frameworks, such as the .NET variant of the Backbase architecture, or the open source Echo2 framework. Furthermore, we consider SPIAR as a starting point for enriching existing web applications with the single page user experience. In particular, we intend to use our Symphony software architecture reconstruction approach [6] to recast legacy web architectures in terms of SPIAR concepts. Finally, we will use SPIAR to analyze how domain-specific languages (such as BXML) and code generation techniques can be used to deal with crosscutting concerns in (single page) web applications [15].

Acknowledgments Partial support was received from SenterNovem, project Single Page Computer Interaction (SPCI). We thank Bas Graaf (TU Delft), Tijs van der Storm (CWI) and Mark Schiefelbein (Backbase) for their feedback on our paper.

8. REFERENCES

[1] 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.

[2] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice, 2nd ed.* Addison-Wesley, 2003.

[3] 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.

[4] 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.

[5] 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.

[6] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 122–134. IEEE Computer Society, 2004.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 1999.

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

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

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

[11] J. Garrett. AJAX: A new approach to web applications. Adaptive path, 2005.

[12] 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.

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

[14] D. Mac-Vicar and J. Navon. Web applications: A simple pluggable architecture for business rich clients. In *5th International Conference on Web Engineering (ICWE)*, pages 500–505. Springer-Verlag, 2005.

[15] A. Mesbah and A. van Deursen. Crosscutting concerns in J2EE applications. In *7th Int. Symp. on Web Site Evolution (WSE)*, pages 14–21. IEEE Computer Society, 2005.

[16] 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.

[17] 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.

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

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

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

[21] 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.

[22] 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.

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

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

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