CompoWeb: A Component-Oriented Web Architecture

Rui Guo
Beihang University
Beijing, China
imguorui@gmail.com

Bin B. Zhu, Min Feng, Aimin Pan
Microsoft Research Asia
Beijing, China
{binzhu, minfeng, aiminp}@microsoft.com

Bosheng Zhou
Beihang University
Beijing, China
bszhou@acm.org

ABSTRACT
In this paper, client-site Web mashups are studied from component-oriented perspective, and CompoWeb, a component-oriented Web architecture, is proposed. In CompoWeb, a Web application is decomposed into Web components called gadgets. A gadget is an abstraction of functional or logical Web component. It is isolated from other gadgets for security and reliability. Contract-based channels are the only way to interact with each other. An abstraction of contract-based channels supported or required by a gadget is also presented. It enables binding of gadgets at deployment, and promotes interchangeable gadgets. Unlike the model of a normal function call where the function logic is executed in caller's context, CompoWeb ensures that the function logic is executed in callee's context so that both the caller and callee are protected. Implementation of a prototype CompoWeb system and its performance are also presented.

Categories and Subject Descriptors
D.1.5 [Object-oriented Programming], D.2 [Software Engineering]: D.2.12 Interoperability – distributed objects, D.2.13 Reusable Software – reuse models; D.3.3 [Programming Languages]: Language Constructs and Features – classes and objects, frameworks, Inheritance. D.4.6 [Operation System]: Security and Protection.

General Terms
Security, Standardization, Languages.

Keywords
Mashup, Web, browser, component, same-origin policy, security, protection, isolation, encapsulation, reuse, delayed-binding, interface.

1. INTRODUCTION
We have witnessed dramatic progresses in Web applications in the past decade. Web pages have evolved from static HTML documents to dynamical content using client-side scripting, from creating content from a single site to integrating contents from different Web sites seamless to offer an enriched Web experience. For example, *housingmaps.com* uses Web mashups to link the craigslist housing database to the Google Maps, creating a new Web service that was not originally envisaged by either source. Another mashup example is iGoogle [1] and Windows Live [2] where gadgets from different sources can be aggregated into a personally customized portal page. A gadget is a component containing both HTML content and scripting code. Due to its tremendous power and flexibility, Web mashups will soon be widely adopted and prevail in Web applications.

Contact author: Bin B. Zhu (binzhu@microsoft.com).

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

In a Web mashup application, contents from different sources are integrated together to achieve the desirable functionality. This can be compared to a desktop application built on top of binary components from different vendors. A component is a unit of program structure that encapsulates its implementation behind an interface used to communicate across the components. The explicit declaration of a component's requirements increases reuse by decoupling components from their operating environment. Component-oriented programming has established itself as the predominant software development methodology over the last decade. It breaks a system down into binary components for greater reusability, extensibility, and maintainability. Several component technologies, such as COM/DCOM, CORBA, Java Beans, and .NET, have been used widely to allow an application with interchangeable code modules, and promote "black box reuse", which allows using an existing component without caring about its internals, as long as the component complies with some predefined set of interfaces.

In this paper, we examine Web mashup applications from component perspective. Component-oriented paradigm is introduced and applied to Web applications for programming efficiency, manageability, functionality, and security. A new Web component called gadget[1] is proposed in this paper. A gadget plays the same role in Web applications as a component in component-oriented programming paradigm. A gadget provides an abstraction to a functional Web component isolated from others except contract-based channels used to interact with others. An abstraction of contract-based channels that a gadget can implement or query is also introduced. The actual implementation of a gadget is encapsulated. Gadgets can be nested: a gadget contains another gadget. With gadgets, a complex Web application can be decomposed into gadgets. Those gadgets, possible distributed or hosted by other Web sites, can be easily glued to deliver a designated functionality, which is exactly a mashup application. Due to its efficiency in developing an application, reusability, and ease in management, we believe that more and more Web applications will be built with gadgets.

In our project CompoWeb, we aim to design and build a component-oriented gadget system for rapid development of rich Web applications. We focus on specifications and execution of a gadget-level abstraction with contract-based interactions, and protection of running environment from attacks and interference by others.

1.1 Design Requirements
To achieve the goal of this project, a gadget should meet the following requirements.

[1] Note that gadget defined in this paper is different from the gadget used by iGoogle or Windows Live, as it will become apparent later in the paper.

- *Encapsulation*: The implementation detail of a gadget should be encapsulated. The actual data and code inside a gadget are hidden from others. A gadget behaves like a black box to others except the contract-based services it provides to others. Separation of implementation and contract-based services are highly desirable in Web applications.
- *Delayed Binding*: When a gadget is implemented, the developer does not have to bind it to another gadget. An abstraction of supported and required contract channels is supported so that binding two gadgets together can be delayed until running time. This delayed binding offers a great flexibility in writing gadgets and gluing them together for a Web application.
- *Isolation of running environment:* A gadget is an abstraction in Web applications that no running state is shared between two gadgets. Each gadget runs in its own execution environment isolated from others. The only communications between two gadgets are through the contract-based channels. This guarantees the security of a gadget at running time even if some gadgets come from untrusted sources, and avoids any interference from other lousily implemented and crash-prone gadgets. Therefore security and reliability of the Web application are ensured.
- *Easy adoption and incremental deployment.* The design of gadget-level abstraction should ensure easy adoption and allow incremental deployment. Our system should be built on top of the existing Web standards and browser implementations with minimized changes. There should be an easy way to provide a fallback mechanism for legacy browsers which do not support our gadget-level abstraction without undesirable consequences.

By meeting the above requirements, gadget-oriented Web development attains much greater reusability, extensibility, and maintainability, and greatly improves security and reliability. These benefits can, in turn, lead to shorter time to market, more robust and highly scalable applications, and lower development and long-term maintenance costs.

1.2 Similarity with Singularity

Although our approach to client-side Web mashups is from the software component perspective, it is also possible to look at it from the Operating System (OS) perspective. For a client mashup application, a browser resembles a multi-user OS: mutually distrusting Web sites interact programmatically in a single page on the client side and share the underlying browser resources for the browser, while mutually distrusting users share the host resources for the OS. Such an OS approach has been adopted by Wang et al. in their MashupOS [3][4] to build a browser-based multi-principal operation system for client-side mashups. Looking from the OS perspective, the system presented in this paper resembles Singularity [5], a research OS with a more reliable and flexible OS architecture, and offering the following three key features as compared to a traditional OS:

- Software-Isolated Processes (SIP) for protection of programs and system services.
- Contract-based channels for communications between two SIPs.
- Manifest-based programs for verification of system properties.

Our CompoWeb has much more in common with Singularity than with a traditional OS:

- A gadget resembles a SIP in Singularity: a gadget runs in an environment isolated from other gadgets by a browser.
- Contract-based channels are the only way to communicate between two gadgets. This resembles contract-based channels to communicate between two SIPs
- A gadget can describe what contract-based channels it requires and supports, verifiable by a machine. This property is used in CompoWeb to delay binding of a gadget with other gadgets until its deployment. A gadget is interchangeable with another one with the same required and supported contract-based channels. This resembles the Singularity's manifest which describes the program's dependencies and desired capabilities, and is machine-verifiable.

1.3 Organization of the Paper

This paper is organized as follows: Section 2 introduces the background for the paper, including the method that current Web applications are developed, the binary trust model that modern browsers have adopted, and Web mashups. In Section 3 gadgets and detailed specifications and design of CompoWeb are presented. The implementation details for a prototype of CompoWeb are provided in Section 4, and experimental results with the implemented prototype are reported in Section 5. Related work is presented in Section 6, and future work is described in Section 7. The paper concludes in Section 8.

2. BACKGROUND

Rapid advance of Web technologies has completely changed the Web from static, single-source HTML documents in the early days to dynamic, interactive, and multiple-source services nowadays. Applications delivered by the Web appear more and more like desktop applications, and will rival even finally replace desktop applications we are using today. Compared to the technologies used in desktop applications, Web applications still lags far behind. Although a Web page can mashup content from different sources, the Web is still a monolithic architecture that does not support component-level abstraction. A binary trust model is used in access control of contents from different sources. In this section, relevant Web technologies are briefly summarized.

2.1 Monolithic Architecture

Web applications are still implemented with a monolithic architecture: each functional part is glued statically at implementation time. With today's Web standards and browsers, scripting from other sources can be used and contents from different sources can be aggregated, but the implementation is not separated from the contract-based services that the implementation provides. Delayed binding and module interchangeability are not supported. Those unsupported features are widely used in component-oriented software development.

2.2 Binary Trust Model

The binary trust model, either no trust or full trust, is used by today's Web standards and browsers, governed by the *Same-Origin Policy* (SOP) which prohibits documents or scripts of one origin from accessing documents or scripts of a different origin [6]. SOP is needed to protect against Cross Site Scripting (XSS) attacks. An origin consists of the domain name, protocol, and port. Two Web pages have the same origin if and only of their domain

names, protocols, and the ports are all the same. Each browser window, <frame> or <iframe>, is a separate document. Each document is associated with an origin. A HTML document is accessed through the platform- and language-neutral interface Document Object Model (DOM). Programs and scripts can use DOM to dynamically access and update the content, structure, and style of documents [7]. Scripts enclosed by <script> in a document are treated as libraries that can be downloaded from different domains, but run as the document's origin rather than the origin from which they are downloaded. With SOP, a binary trust model, either full trust or no trust at all, is used for today's Web applications. A site *a.com* either does not trust another site *b.com*'s content at all by enclosing *b.com*'s content inside a frame, thus a separate document, or trusts *b.com*'s scripts entirely by embedding *b.com*'s scripts to grant them full access to *a.com*'s resources.

2.3 Web Mashups

A Web mashup is defined as a Web page containing documents from different sources. SOP prevents these documents from interacting with each other, thus restricts the functionality that a mashup page can possibly deliver. To work around SOP, a proxy server can be used to aggregate the contents from different sources before sending to the client so that the mashup contents appear to be the same origin to the browser. Drawbacks of this approach include that the proxy server can be a bottleneck and unnecessary round trips are required.

AJAX (Asynchronous JavaScript and XML) has been widely used to provide interactivity through client-side code with minimized impact on network and server performance. AJAX makes client-side mashups popular since client-side mashups reduce latency and bandwidth as compared to the proxy approach described above. A client-side mashup includes documents from various sites and makes them interact with each other at the client side. To circumvent SOP, a document in a client-side mashup embeds scripts from the target sites in order to achieve cross-domain interactions, which again requires full trust of those sites. SOP's binary trust model forces Web programmers to make tradeoffs between security and functionality. Security is frequently sacrificed for functionality.

Figure 1: Three gadgets aggregated into a page.

Web gadget aggregators are used by iGoogle [1] and Windows Live [2] to enable a user to customize his or her portal page by selecting multiple third-party contents. Each content manifests as a gadget. A gadget in these applications is a separate frame. SOP isolates one gadget from another as well as from the gadget aggregator. This has restricted the functionality of a Web mashup. For example, a Web page shown in Figure 1 contains three gadgets from different origins: the top left one is a people gadget which lists people, the bottom left is a weather gadget which shows a city's weather, and the right one is a map gadget which shows a map. SOP prevents the weather and map gadgets from responding to a click on a person in the people gadget to show his home on the map gadget and the weather of his home on the weather gadget. To support this desired functionality, scripts from a different source need to be embedded with a full trust being granted. With CompoWeb, the described functionality can be delivered with a few lines of code, as given in Section 3.3, without sacrificing security.

2.4 Cross-Domain Communications

New technologies have been proposed to offer client-side cross-domain communication mechanisms without sacrificing security. These technologies include the <module> tag [11], Subspace [18], URL fragment identifier [19][14], MashupOS [3][4], etc. More can be found in Section 6.2.

Schemes for secure cross-domain communications from browser to server have also been proposed [16][17][21][13]. Crockford [21] proposed using JSONRequest with the following features that allow it to be exempted from the Same Origin Policy: don't send or receive cookies [8] or passwords in HTTP headers; transport only JSON text, drop responses from legacy server. This scheme has also been adopted in CompoWeb.

3. COMPOWEB

3.1 Overview

3.1.1 New Concepts

CompoWeb applies the component-oriented software programming paradigm to Web applications. Two key concepts are introduced in CompoWeb: gadget and interface. A *gadget* is an abstraction of a functional or logical Web component supporting contract-based channels to communicate with others. It is equivalent to a component in the component-oriented programming paradigm. An *interface* is an abstraction of pre-defined, machine-queryable contract-based channels through which a gadget can communicate with others in a controllable manner.

3.1.2 Key Features

CompoWeb meets the requirements described in Section 1.1 with the following key features:

- **Browser-Isolated Gadget**: Each gadget runs under a private environment isolated from others by the browser. This ensures the integrity and guarantees the confidentiality of the internal state of a gadget. Reliability is also improved since the running status of one gadget does not affect other gadgets. A gadget resembles a SIP in Singularity [5] which runs under a software isolated environment.
- **Safe Invocation:** A gadget can invoke another gadget in the same way as if invoking a normal JavaScript object, i.e., through latter gadget's exposed member properties, methods and events, the so-called PME model [20]. Unlike invoking a normal function call that the invoked function logic runs in caller's context, the invoked member method of a gadget runs in its own context without interfering caller's context. The input arguments and the return values are exchanged between the caller and callee as pure data.
- **Delayed Binding Mechanism**: This mechanism allows a gadget developer to declare dependencies on an abstraction of contract-based channels (i.e. gadget interfaces) and write logic to collaborate with these channels, without statically

binding to actual gadget instances. Binding with actual gadgets can be delayed until deployment, i.e., CompoWeb supports dynamic binding of gadgets.

3.1.3 Extension to HTML and Scripts

Gadgets in CompoWeb are defined with HTML and JavaScript. Our goals are:

1. Minimal modifications to convert a current Web page into a gadget-based Web page.
2. Majority of a gadget's content can be rendered by legacy browsers which do not support CompoWeb.

CompoWeb extends the current Web standards to achieve the design goals. A new HTML tag named <gadget> is added to define a gadget, and three new HTML meta types, i.e., implementedInterfaces, internalUse, and usage, are added. Several global JavaScript objects and functions shown in Figure 2 are added in CompoWeb. Usage of these added terms will be explained in the subsequent sections.

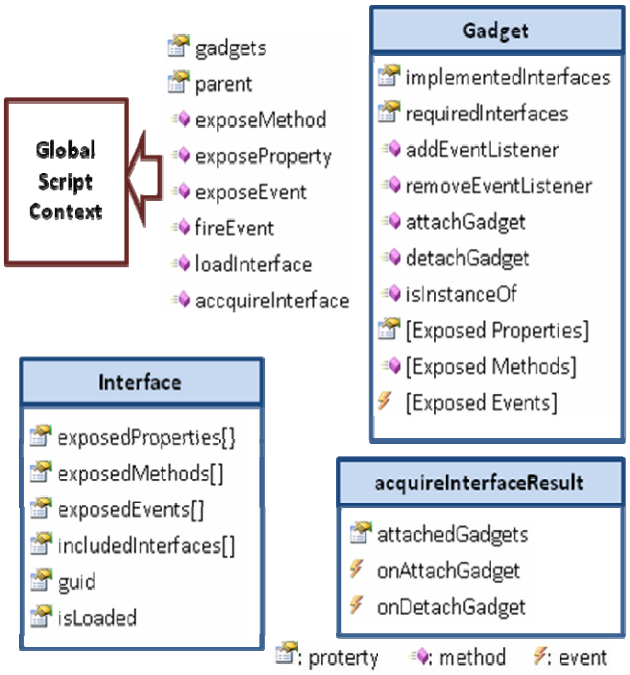

Figure 2: Added JavaScript objects and functions.

3.2 Gadgets

A gadget with an ID of "alice_news" is defined as follows:

```
<gadget src="http://alice.com/news.htm"
id="alice_news" width="400" height="300">
```

This definition is similar to that of a frame. Like a frame, each gadget is associated with an origin. Gadgets can be nested. When the above gadget is instantiated, the browser creates an isolated running environment, fetches the gadget content from the specified source address http://alice.com/news.htm, processes the DOM objects, and runs the script objects inside the gadget in a private space isolated from other gadgets or frames.

For each source HTML file that implements a gadget, the following <meta> tag is used to explicitly declare that the content is a gadget:

```
<meta name="usage" content="gadget" />
```

This statement tells a browser that the source HTML file intends to be only a gadget. When a gadget file is embedded in an <iframe> or <frame> tag, a browser still ensures that it behaves as a gadget rather than a frame (i.e. only the exposed members can be accessed by others, even if the access is from the same origin).

Persistent state of a gadget is stored in cookies, which are currently handled in the same way as existing browsers. A cookie is accessible by the Web pages of the same directory as or subdirectories of the Web page which created the cookie. Therefore two <gadget>s can share the persistent data in a cookie if and only if their sources share the same domain and path. Our current design that isolates the running environment of a gadget except its persistent state does not compromise any security in practice since the additional access specification of a cookie, i.e., the path, can be used to isolate the persistent state of a gadget if necessary: placing the creating Web page of the cookie, typically the gadget itself, in a unique directory that no other Web page or gadget resides in that directory or its subdirectories except the gadget itself.

It is informational to compare a gadget with a frame. Full or no trust governs the accessibility of a frame: Its internal document trees and scripts are fully accessible by other frames or gadgets from the same origin, or not accessible at all if from different origins. SOP is not applicable to govern accessibility to a gadget. A gadget is not accessible by other gadgets or frames except through its exposed contract-based channels, no matter those gadgets or frames are from the same origin or not. Therefore a gadget has a much finer access control.

3.3 Encapsulation

As we have mentioned, a gadget appears as a black box to others except the contract-based channels it supports. A contract-based channel is an exposed member method, property, or event. The three extended global JavaScript functions, i.e., exposeMethod, exposeProperty, and exposeEvent, can be used to define a communication contract with a member method, property, and event, respectively. For example, the source of a map gadget may contain the following code to expose a method named "setLocation" for other gadgets to show a specific location on the map gadget:

```
function setLocation(loc) {
    innerMapControl.goto(loc);
    return innerMapControl.getCenter();
}
exposeMethod('setLocation');
```

Another gadget can manipulate the map gadget through the exposed member method to set the map gadget to display a location such as Beijing:

```
// NOTE: 'map1' is the id of the map gadget.
var newLocation = map1.setLocation('Beijing');
```

These few lines of code can fulfill the function to let the map gadget to show on the map the home of the person clicked in the people gadget in Figure 1, as we desired in Section 2.3.

Although the above code looks exactly the same as a normal JavaScript function call in syntax, they have a fundamental difference with implication in security. In the above call, the caller gadget marshals the input arguments (i.e. 'Beijing') to the map gadget. The function logic is then executed in *callee's* context. At the end of the execution, the result is marshaled back to the caller. The "exposeMethod" exposes only the name of the method, rather than the method handle. This is very different from calling a JavaScript function where the called function logic is executed in the *caller's* context.

This difference of running in different contexts has a great impact on security. As we mentioned previously, calling a JavaScript function from an untrusted origin has security implication. Many XSS attacks have exploited this method to launch successful attacks. On the contrary, the same syntax is basically secure to use when it is applied to gadgets in CompoWeb. When gadget A calls a method exposed by gadget B, A is secure since the called function logic is executed in B's context. Of course, A should be cautious about the returned result, which should be checked and validated before using. B is also secure when its member function is called by another, possibly untrusted, gadget since the function logic of its member function is executed in its own context, and B's internal state keeps isolated during the function call.

A careful reader may notice that we have not specified the arguments in exposing a member method. This is because that every function in JavaScript has essentially a variable length of argument list no matter how many arguments appear in the function declaration. This makes specification of arguments for an exposed method meaningless. As a result, only the method name is exposed when exposing a member method.

A gadget can also expose its property. A property is exposed as follows:

```
function get_Name() {
    return …;
}
function set_Name(value) {
    …
}
exposeProperty('Name', 'get_Name', 'set_Name');
```

The first argument in exposeProperty is the name of the exposed property, the second and third arguments of exposeProperty are the getter method and setter method, respectively. The last two arguments are optional. When they are omitted, the default names are used. The default names for the getter and setter methods are the property name prefixed with "get_" and "set_", respectively.

Reading or writing a gadget's property is just like reading or writing an object's field, e.g., gadgetId.Name ='Alice', which is then translated to calling getter or setter method, resulting in a higher level of encapsulation.

Gadgets can also provide notifications about an occurrence of a specific event, such as a successful completion of a method, to other objects. Events are exposed with "exposeEvent" and triggered with "fireEvent":

```
exposeEvent('CalcCompleted');
function calc() {
    …
    fireEvent('CalcCompleted', result);
}
```

fireEvent is also a global function in CompoWeb. Its usage is self-explained.

A gadget can register or unregister handlers to another gadget's event notifications with "addEventListener" and "removeEventListener". These two methods are fixed member methods of every gadget. For example, gadget A can register or unregister a handler "someScriptMethod" to gadget B as follows:

```
B.addListener('CalcCompleted',
someScriptMethod);
B.removeListener('CalcCompleted',
someScriptMethod);
```

It is possible that multiple gadgets respond to a single event. This is easily done by registering their handlers to the event. When an event fires, the associated handlers are called. CompoWeb guarantees that a handler registered to respond to an event of a gadget cannot be accessed by the gadget which fires the event. A browser maintains a list of handlers responding to an event. When an event is fired, the browser executes all the handlers registered to respond to the event. As a result, a gadget can register private member method as a handler to respond to an event of another gadget without sacrificing security. Like method calls, a handler runs in the context of the gadget which registers the handler to respond to an event. A handler does not run in the context of the gadget which fires the event.

The event mechanism described above can be used to deliver rich Web experience easily. For example, if we would like the map gadget and the weather gadget shown in Figure 1 to respond to a click of a person in the people gadget to show the location and the weather of the home of the person being clicked, we can simply write the following two lines of code to realize the functionality:

```
list.addEventListener('locationChanged',
map.setLocation);
list.addEventListener('locationChanged',
                weather.queryByLocation);
```

3.4 Scope of Exposed Members

By default, an exposed member is visible and callable by any gadget. Such an exposed member is said to be of global scope. Global scope may be undesirable in some cases. A gadget may want to restrict an exposed member to be viewable and accessible by a specific gadget or group of gadgets. This is supported by CompoWeb but only at the granularity of an origin. Two levels of scopes are supported by CompoWeb: the global scope and the Same Origin Scope (SOS). When a gadget is of SOS, its exposed members can be viewed and called only by the gadgets of the same origin. The gadget seems to have exposed nothing to gadgets of a different origin.

There are two ways to specify the scope of gadget. The first method is to specify in the source file of a gadget. The syntax is:

```
<meta name="internalUse" content="true | false"/>
```

When internalUse is set to true, the gadget's scope is SOS. Otherwise the scope is global. If internalUse is not specified, the default scope is applied. In CompoWeb, the default scope is the global scope.

The other method to specify a gadget's scope is within the <gadget> tag:

```
<gadget … internalUse="true | false"/>
```

Like the first case, the default value is false, i.e., the global scope, if not specified.

The two methods in specifying a gadget's scope have different effects. If the scope is specified inside the source code of a gadget, any instance of the gadget is of the same origin scope. If the scope is specified inside <gadget>, only the instance of the gadget specified by the tag <gadget> is of that scope. A same gadget specified by another <gadget> may have a different scope.

If the scope of a gadget is specified more than once, for example, once inside the source code, and another inside the <gadget> tag, the narrower scope prevails.

3.5 Interfaces

An interface is an abstraction of contract-based channels. CompoWeb utilizes an XML-based file format to define an interface. An interface defines a set of names for exposed properties, methods, and events. For example, we can define an IMap interface as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<interface GUID="FCB5ED82-F243-44fc-974D-
A56248DB20AA">
  <exposedProperty name="Center" />
  <exposedMethod name="SearchLocation" />
  <exposedEvent name="LocationChanged" />
</interface>
```

Every interface has associated with a unique GUID to prevent conflict of exposed names in difference interfaces.

A gadget is said to have implemented an interface if both of the following two conditions are met:

1) The gadget has implemented *all* the members declared in the interface definition;
2) The gadget has declared that it has implemented the interface. Declaring implementation of an interface has implicitly exposed all the members specified in the interface. Therefore a gadget does not need to declare each exposed member already defined in an interface declared to be implemented by the gadget.

A gadget uses the following syntax to declare that it has implemented an interface:

```
<meta name="implementedInterfaces" content=
interfaceList />
```

Each element in the interfaceList declares an implemented interface, which must specify a URL where the interface is defined, and may optionally specify a hash value of the interface definition file calculated at the implementation phase. This calculation can be done with a developing tool. The hash value helps a browser to verify whether the interface definition has been modified after the gadget is deployed. The hash value should be provided when the gadget developer does not fully trust the host of the interface definition. Otherwise a malicious host may be able to expose a private member of a gadget by adding the name of the private member in the interface definition after the gadget is deployed. Since the gadget has declared that it has implemented the interface, which has implicitly exposed all the members specified in the interface. Such a modification would result in exposing the private function that the gadget developer has no intention to expose, a potential security loophole. A browser would produce an error message if an interface does not match its hash value included in a gadget which implements the interface.

A user can verify if a specific gadget has implemented a specific interface by using the gadget member method named "isInstanceOf", which is a fixed member method of every gadget as shown in Figure 2:

```
var flag = gadgetName.isInstanceOf(interfaceURL);
```

CompoWeb allows users to define an interface by reusing and extending an existing interface:

```
<?xml version="1.0" encoding="utf-8"?>
<interface GUID="AAE65ED4-1152-4050-8F45-
CEDEC50D3ABB">
    <!--including IMap interface-->
    <include src="http://interfaces.com/imap.xml"
/>
    …
</interface>
```

If a gadget has implemented the outer interface, it should have also implemented the included interfaces, such as the IMap interface in the above example. When the hash value of the outer interface is calculated at the implementation phase of a gadget, the included interfaces are expanded and then the hash value of the expanded file is calculated. Similarly, declaring implementation of the outer interface implicitly declares implementation of the included interface.

3.6 Delayed Binding

In CompoWeb, a gadget, say gadget A, can explicitly declare its interest to communicate with other gadgets which have implemented a certain interface (such as the IMap interface). This can be accomplished by calling an extended global function named "acquireInterface" inside the gadget:

```
var imap =
acquireInterface('http://interfaces.com/imap.xml')
;
```

This function declares that the gadget depends on some contract-based channels specified by the interface. It implies that the gadget hopes to find another gadget which has implemented the interface to complete its logic.

The declared requirement of dependency is met when a suitable gadget, say gadget B, is attached to the requirement submitter:

```
//B has implemented the interface imap.xml
A.attachGadget(B);
```

"Suitable" means that gadget B has implemented the interface acquired by gadget A earlier, and "attach" means that gadget A can collaborate with gadget B through the acquired interface.

Such declarations enables an aggregator to bind gadgets, such as binding gadget A which wants the IMap interface to complete its logic with gadget B which has implemented the IMap interface, without knowing the meaning the IMap interface or functionaries of the gadgets. Such a binding does not require any modification of a gadget, and can be done when the gadgets have been published. CompoWeb also allows a gadget to decide through scripts and configure files whether and how to connect with the gadgets which are recommended by the aggregator and have implemented the interface that the gadget requires.

The function of acquireInterface always returns an acquireInterfaceResult object which has three members, as shown in Figure 2. The first member, attachedGadgets, is a gadget array storing the "suitable" and attached gadgets; the second member is an event which is fired after attachedGadgets inserts a new element, and the third member, also an event, will get fired after attachedGadgets removes an existing element.

A gadget may communicate with its attached partners by accessing the attachedGadgets member of the acquireInterface result:

```
/*below code will call setLocation on all attached
IMap instances */
if (imap.attachGadgets.length > 0) {
      for (i = 0; i< map.attachGadgets.length; i++)

      map.attachGadgets[i].setLocation('Beijing');
}
```

One of a gadget's fixed members, the requiredInterfaces property, stores all the interface requirements of the gadget, and gets updated after every invocation of acquireInterface. Two gadgets are able to examine each other's dependency requirements via their requiredInterface members.

For an aggregator gadget, it is possible to "auto-connect" its children gadgets by inspecting and mapping their requiredInterfaces and implementedInterfaces:

```
//Note: assume 'g1' and 'g2' are two gadget
objects
If (g1.implementedInterfaces intersects with
g2.requiredInterfaces)
{
      g2.attachGadget(g1);
};
```

Such an auto-connection script helps the aggregator bind matched gadgets together. When there is more than one possible way to bind, scripts and configure files can be used to choose a binding, as explained previously in this section.

3.7 Incremental Deployment

Incremental deployment is critical in adopting a new technology since it is impossible to replace overnight the existing browsers with those that support CompoWeb. We must ensure that there will be no undesirable effect or interaction between a CompoWeb-enabled Web application and a legacy browser which does not support CompoWeb. Web developers should have a safe fallback mechanism to deal with the case that CompoWeb extended HTML tags and JavaScript functions are not recognized or supported by a legacy browser. A safe fallback can be implemented as follows:

Firstly, a "Not Supported" notification should be added as the inner text to every <gadget> tag:

```
<gadget id="..." src="...">
      CompoWeb is not supported by your browser.
</gadget>
```

This message is ignored by CompoWeb-enabled browsers but rendered as plain text by legacy browsers. We have exploited the fact that a legacy browser ignores any unrecognized tags.

Secondly, we can examine our script blocks and embrace every occurrence of CompoWeb extended functions and objects into a conditional statement block, where the conditional statement checks whether CompoWeb is currently supported or not by examining whether some of the CompoWeb extended functions are defined:

```
if (acquireInterface && exposeMethod)
{
      //code using CompoWeb functions and objects
}
```

Therefore the script engine in a legacy browser will not be interrupted by CompoWeb functions and objects.

4. IMPLEMENTATION

We have implemented a prototype CompoWeb system to verify our proposed concepts and evaluate its performance. Our prototype is based on Internet Explorer 7. The prototype was tested on both Windows XP SP2 and Windows Server 2003 SP1. Although we have not tested yet, our methodology and techniques can also be applied or extended to other browsers such as Firefox and Opera.

Instead of modifying IE's source code directly, we leveraged the browser extensions and public interfaces exported by IE to implement the prototype without touching IE's code base. Nevertheless we expect that CompoWeb will be implemented directly inside modern browsers instead of implemented as add-ons when CompoWeb is widely adopted.

Our system consists of two major extensions to the IE architecture [10]. The first extension is an ActiveX control. The second extension is the CompoWeb MIME filter, which is responsible for supporting our HTML language syntax extensions. In addition, we have implemented a set of COM objects to help implement various features in CompoWeb.

Figure 3 shows our implementation to support gadgets at run-time. Each gadget is associated with an ActiveX instance, which processes the gadget and provides an "isolated" running environment for a gadget. The outmost part is an ActiveX control which wraps the native Web browser control (ShDocvw). Most extensions of CompoWeb to the current Web standards are implemented in the ActiveX. When a gadget loads its source page, ActiveX enumerates all the <meta> tags in the loaded document, and examines the interfaces that the gadget has implemented. Then a set of functions and objects are attached to the window object as its members to become global script functions and objects (see Figure 2 for the whole list of these functions and objects).

Among these global script extensions, the three "expose" methods are the most frequently used functions. Every invocation of them results in a name entry added into the corresponding list of the exposed members. Three separated lists are used to record the exposed properties, methods, and events, respectively.

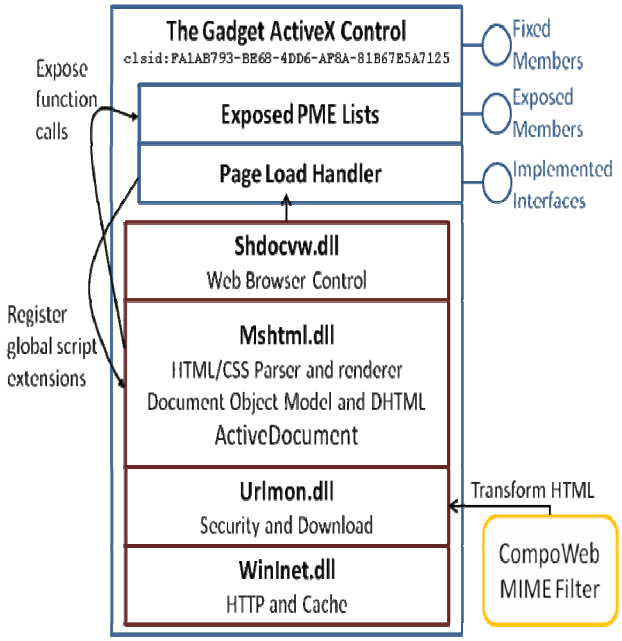

Figure 3. Implementation for run-time support of gadgets.

The gadget ActiveX control has implemented the IDispatch COM interface, and interacts with the browser script engine through this interface.

The gadget ActiveX control exposes both dynamically exposed members and a set of fixed members (see Figure 4) through the IDispatch COM interface, which can be recognized and invoked by IE's script engine.

When it invokes an exposed method, the script engine first queries the IDispatch interface to check whether the target gadget has such a method. Then the IDispatch interface looks up the method in the list of exposed methods. If a corresponding entry is found, the gadget control will try to invoke the script function resides in its embedded Web browser control through a set of public interfaces provided by IE. All invocations in our implementation are made by value: All input arguments are copied with non-data fields of each argument discarded, and the copied arguments are then passed to the target gadget. The above processing steps are shown in Figure 4.

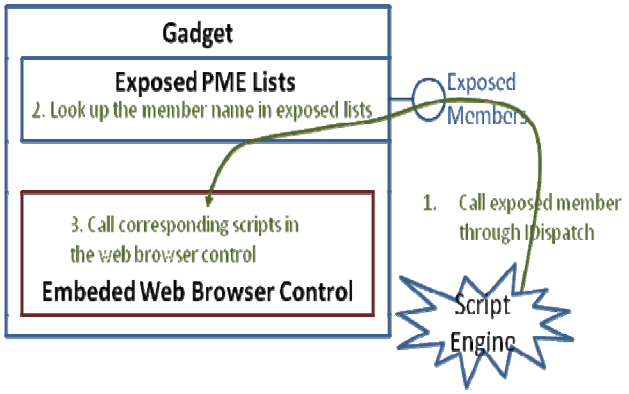

Figure 4. Processing an exposed method.

The second extension, i.e., the CompoWeb MIME filter, is an asynchronous pluggable protocol handler at the software layer of URLMon.dll, where various content (MIME) types are handled. The filter takes as input an HTML stream and transforms new tags into existing tags. For example, as we described above, each gadget is processed by an ActiveX control. Therefore the filter should transform all the <gadget> tags into <object> tags such as the gadget shown below

```
<gadget
src="http://localhost/GadgetTest/timeline.htm"
width="450" height="380" id="timeline"></ gadget >
```

is translated by our MIME filter to

```
<object classid="clsid:FA1AB793-BE68-4DD6-AF8A-
81B67E5A7125" width="450" height="380"
id="timeline">
<param name="Src"
value="http://localhost/GadgetTest/timeline.htm"
/>
</object>
```

The current implementation of the prototype system was programmed in C#/.Net since .Net framework has provided rich libraries that make the implementation much easier. That said, the performance of our prototype system might have been sacrificed. The time cost overhead in our performance tests reported in Section 5, although acceptable as a prototype, might be caused by a complex interoperation between .Net runtime and the COM objects. We are currently re-implementing some critical components in C++ to achieve a better performance.

5. EVALUATION

The prototype implementation of CompoWeb described in Section 4 has been tested on a Dell Optiplex GX 620 PC with 3.0 GHz Pentium-4 PC and 1 GB of RAM for performance evaluation. The PC ran on Windows XP SP2 with the Web browser Internet Explorer (IE) 7.

The first test was the standard JavaScript speed test with Orendorff's JavaScript benchmark [9]. This benchmark contains 77 test cases in 13 categories, ranging from text processing to object handling. For each test case, the tester creates a function that executes the test case code for N times in a tight loop. Then it calls the function repeatedly, with N=1, then 2, then 5, 10, 20, 50, and so on until the loop actually takes a significant amount of time to execute (at least 200ms). It does this 5 times, throws out the worst time, and averages the other four as the test result.

We ran benchmark either with or without our browser extension. From the test results, we have observed negligible differences for both cases. This is expected since there is no change to the script engine in our implementation. Only a few methods are added into the global script scope.

To have a better evaluation of the gadget wrapper's overhead spent to process exposed functions, a set of benchmarks was designed as follows:

1. We designed a gadget which defines 4 JavaScript methods, all of them exposed. These four methods were "**Empty** loop", "Create nonempty **function**", "Populate 500 numbered properties with **object**" and "Populate 500-element **array** of numbers using push()", copied from Orendorff's JavaScript benchmark. These four methods are referred to as "**Empty**", "**Function**", "**Object**", and "**Array**", respectively, in Table 1.
2. We tested and recorded the time cost in invocating a **cross-frame** (with the same domain) function (i.e., calling a function with a script like "frameName.document.funcName();"). This case is referred to as "**cross-frame**" in Table 1 and in the remaining part of this section.
3. We tested and recorded the time cost in invocating a function exposed by the **gadget**, (i.e., calling the function with a script like "gadgetName.funcName();"). This case is referred to as "**gadget**" in Table 1 and in the remaining part of this section.
4. We reused some code of Orendorff's JavaScript benchmark to build our own test, and followed the same test procedure.

The average time costs for both "**cross-domain**" and "**gadget**" are reported in Table 1. We can see from the table that the difference between the two cases for each member function remains roughly flat while the actual time cost rises over 10 times. The results show that CompoWeb incurs almost constant overhead. This near-constant overhead can be attributed to:

- The look-up time to find the designated exposed member in the target gadget.
- The constant overhead in calling the IDispatch interface (calling once for "**cross-frame**" and twice for "**gadget**", see Steps 1 and 3 in Figure 4)

Table 1. Time cost (μs) for each of the four member methods, "Empty", "Function", "Object", and "Array", with both "cross-frame" calls and gadget calls.

	Empty	Function	Object	Array
Cross-frame	111	118	594	1456
Gadget	187	203	686	1561
Difference	76	85	92	105

Since the number of exposed methods would significantly affect the efficiency in looking up an exposed function from the list of exposed methods and calling IDispatch interfaces, it would be interesting to study the relationship between the time cost in invocating a gadget's exposed function and the number of exposed methods of a gadget. The "**Empty** loop" method was used for this test. The test results are shown in Figure 5. The data in this figure confirms that the time cost overhead incurred by CompoWeb depends linearly on the number of exposed methods in our current implementation.

As we described in Section 4, the performance overhead might be mainly due to a complex interoperation between .Net runtime and the COM objects since the current implementation of the prototype was in C# and .Net. Better performance is expected when the prototype is implemented in C++.

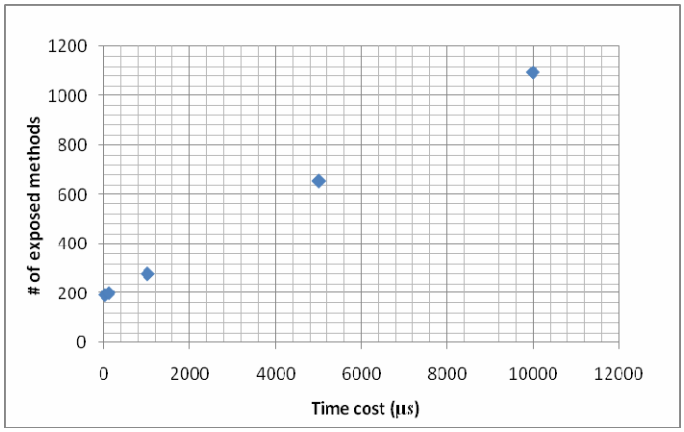

Figure 5: Time costs (μs) under various numbers of exposed functions.

6. RELATED WORK

6.1 Component-Oriented Software Development

Component-oriented software development provides a high level of abstraction in software development. It separates specifications from actual implementation and promotes reuse of components. Many modern software technologies have used component-oriented approach, such as COM/DCOM, Java Beans, and .NET, to develop desktop, server-client and distributed applications. The component-oriented program paradigm, however, has not been used in Web applications and mashup systems.

6.2 Cross-Domain Communications for Web Mashups

The new <module> tag was proposed by Crockford [11] to partition a Web page into a collection of modules. A module is isolated except that JSON [12] formatted messages are allowed to communicate between a module and its parent document. By simply defining and exposing the send and receive member functions, we can have our gadget to mimic a module.

A similar scheme has been proposed for HTML 5 [13] to provide cross-document communications, no matter if the documents belong to the same domain or not. Since documents are arranged in hierarchy structure, this proposal leverages the current abstraction of a document instead of proposing a new isolation abstraction like the <module>. Though cross-domain communications are supported in this HTML 5 proposal, the communication receiver has to decide the trustiness of the sender by itself. This requires every component has its own access control system. Furthermore, DOM and JavaScript resources are shared based on the same origin policy. Therefore, a separate DNS domain per component would still be required.

Flash Player framework uses cross-domain policy files [16] to configure and give the Flash Player permission to access data from a given domain without displaying a security dialog. Although this approach provides more flexibility and controls than standard SOP communication model, it depends on a configuration outside a browser, and the service provider cannot distinguish whether the requests originator comes from the same domain as the provider or not.

Subspace [18] provides a cross-domain communication mechanism without any browser plug-ins or client-side changes. Subspace splits a site into sub-domains, using one of them to evaluate scripts from other domains, and another page to hold a notification object. Then the two sub-domain pages relax their domain to a common value to exchange information, and send information back via the held notification object. Subspace is complex to use, esp. for complex mashups, and may not work for certain domains. For example it is impossible to relax a domain such as "a.com" or "192.168.0.1" to create a parallel domain to receive partially trusted information. Therefore Subspace does not work in these cases.

Approaches to communicate between <iframe>s by using the fragment identifier [19] of the frame URL have been proposed. Modification of the URL fragment identifier dose not reload the page, and can be observed by frames from different domain, thus can be used to transport messages between frames. However, such communication is limited to the size of fragment identifiers (the maximum length of a URL in Internet Explorer is 2,083 characters), and can be overheard by other frames.

In DOMLAC [15] a browser plug-in provides a fine-grained access control on read, write, and traverse actions of the DOM tree of a Web application. In order to safely isolate the DOM sub-tree of each component, policies are associated with parts of the DOM tree inside a Web page, such as defining a policy that only the component and the event hub can access and modify a communication zone between them. Therefore it prevents innocent parts from accessing potentially malicious parts of the DOM tree.

MashupOS [3] proposes to add several new elements to HTML. Among them, <Sandbox> and <OpenSandbox> tags are designed to consume unauthorized content without liability and over trusting. The <ServiceInstance> tag creates an isolated region to hold related memory and network resources. A <ServiceInstance> may also hold

multiple display area resources by possessing some <Friv> nodes in the HTML document tree. MashupOS also provides browser-side communication across domains. <ServiceInstance>s may declare ports to listen to communication requests. Such a request can be sent from any script block by using a CommRequest object provided by MashupOS. Cross-gadget communications in CompoWeb are through the PME model [20], which is more convenient than the sending and receiving message model used in MashupOS as well as in the <module> approach [11] and the HTML 5 proposal [13]. CompoWeb also supports an abstraction of contract-based channels to promote interchangeability among gadgets and separation of a gadget's implementation from its actual deployment. MashupOS lacks these features.

7. FUTURE WORK

There are several possible ways to further study and extend the current work. Although we have isolated executable environment of a gadget, the persistent state is stored in cookies which are still handled in a traditional manner. That may give two gadgets an opportunity to share their persistent states. We may want to modify the current cookie handling mechanism to provide a further isolation of a gadget: the persistent state is also isolated.

The current access control to exposed members in CompoWeb is very coarse. Only two scopes are supported. A gadget cannot specify an arbitrary set of gadgets to see and access its exposed members while disallowing other gadgets from knowing or accessing the members it has exposed.

Like other proposals, the current scheme of CompoWeb lacks a sophisticated mechanism to handle page refreshing and navigations that occur in a gadget. These actions may pose some new challenges such as unloading resources and dealing with potential attacks. These issues will be addressed in the next phase of CompoWeb.

8. CONCLUSION

In this paper, we examined Web applications, esp. client-side Web mashups, from component-oriented perspective, and proposed a component-oriented Web architecture, CompoWeb, in which gadgets are building blocks. A gadget offers an abstraction at a functional or logical Web component level. Each gadget is isolated from others for security and reliability, and communicates with others through contract-based connections. Binding of a gadget with others can be delayed until deployment to separate implementation from the actual deployment. CompoWeb promotes component-level abstraction, encapsulation, and isolation as well as interchangeability and reuse.

9. ACKNOWLEDGMENTS

We would like to thank Sunava DUTTA, Xiaofeng FAN, Helen WANG, and Zhenbin XU for their valuable helps, discussions, and feedbacks to this project.

10. REFERENCES

[1] Google Inc. Google Gadgets API Developer Guide. http://www.google.com/apis/gadgets/docs-home.html.

[2] Microsoft. Windows Live Gadget Developer's Guide. http://microsoftgadgets.com/livesdk/docs/default.htm.

[3] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating System Abstractions for Client Mashups. In 11th Workshop on Hot Topics in Operating Systems (HotOS XI), San Diego, CA, May 7-9, 2007.

[4] H. J. Wang, X. Fan, C. Jackson, and J. Howell. Protection and Communication Abstractions for Web Browsers in MashupOS. In 21st ACM Symposium on Operating Systems Principles (SOSP), Stevenson, WA, October 2007.

[5] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. Operating Systems Review (ACM SIGOPS), Vol. 41, No. 2, pp. 37-49, April 2007.

[6] J. Ruderman. The Same Origin Policy. http://www.mozilla.org/projects/security/components/same-origin.html.

[7] Document Object Model. http://www.w3.org/DOM.

[8] D. Kristol and L. Montulli. HTTP State Management Mechanism. IETF RFC 2965, Oct. 2000.

[9] Jason Orendorff. JavaScript speed test. http://www.jorendorff.com/articles/javascript/speed-test.html.

[10] Internet Explorer Architecture. http://msdn.microsoft.com/workshop/browser/overview/ie_arch.asp.

[11] D. Crockford. The Module Tag: A Proposed Solution to the Mashup Security Problem. http://www.json.org/module.html.

[12] D. Crockford. RFC 4627 The application/json Media Type for JavaScript Object Notation (JSON).

[13] Web Hypertext Application Technology Working Group. HTML 5 - Cross-document messaging. http://www.whatwg.org/specs/web-apps/current-work/#crossDocumentMessages.

[14] F. De Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. SMash: Secure Cross-Domain Mashups on Unmodified Browsers. http://domino.research.ibm.com/library/cyberdig.nsf/1e4115aea78b6e7c85256b360066f0d4/0ee2d79f8be461ce8525731b0009404d?OpenDocument.

[15] N. Uramoto, S. Yoshihama, and F. De Keukelaere. OpenAjax Security Work Session. http://www.openajax.org/member/wiki/images/0/0c/2007_March_Members_Meeting_Ajax_Security_Threats.pdf.

[16] Adobe, Adobe Flash Player 9 security white paper, July, 2006. http://www.adobe.com/devnet/flashplayer/articles/flash_player_9_security.pdf.

[17] J. Couvreur. FlashXMLHttpRequest: cross-domain requests. http://blog.monstuff.com/FlashXMLHttpRequest.

[18] C. Jackson and H. J. Wang: Subspace: Secure Cross-Domain Communication for Web Mashups, WWW 2007, pp. 611-619, Canada, May 2007.

[19] James Burke: Cross Domain Frame Communication with Fragment Identifiers. http://tagneto.blogspot.com/2006/06/cross-domain-frame-communication-with.html.

[20] Borland Software Corp. PME: Properties, Methods and Events. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1384.pdf.

[21] D. Crockford. JSONRequest. http://www.json.org/jsonrequest.html.