Towards Service Composition Middleware Embedded in Web Browser

Gang Huang[1,2], Qi Zhao[1,2], Jiyu Huang[1,2], Xuanzhe Liu[1,2], Teng Teng[3], Yong Zhang[3], Honggang Yuan[3]
[1]Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
[2]School of Electronics Engineering and Computer Science, Peking University, Beijing, China
[3]Kingdee Middleware Company Ltd., Shenzhen, China
{huanggang, zhaoqi06, huangjy07, liuxzh}@sei.pku.edu.cn, {tengteng, zhangyong, danielyuan}@apusic.com

***Abstract*—Due to the rich user experience and Internet-wide scalability, more and more Web-delivered services are assembled in web browsers and the resulted service composition itself is also running in the browsers. Today's popular service composition environments embedded in web browsers mainly focus on the experiences of end-users or non-professional users. The solutions for other composition issues, e.g. service access and interaction enablement, are private and tightly coupled with the user interfaces. In this paper, we propose a new type of middleware, which is embedded in web browsers and encapsulates reusable solutions for common problems to the composition of Web-delivered services, including a container for component instances, a set of communication mechanisms for interactions within the browser, between the browser and server, between the browser and local resources. Based on iCM, different service composition environments preferred by different users can be constructed easily with high quality. In the evaluation, we implement a prototype of the browser middleware, called Internetware Client Middleware (iCM), construct a new service composition environment, called iMashup, with iCM and compare iMashup with some popular environments. The evaluation results demonstrates that iMashup has richer composition capabilities, supports more types of web browsers, consumes smaller memory and gains practical scalability. These observations show the feasibility and effectiveness of the proposed middleware.**

Service-composition; middleware; mashup; Internetware

I. INTRODUCTION

Nowadays, many web sites and applications, such as Google, Amazon and Facebook, expose either data feeds or more advanced web-delivered services (e.g., SOAP and RESTful web services). Developers are now assembling various services to create a large number of composition applications (e.g., those so-called mashups) to solve all types of problems [1]. ProgrammableWeb.com is a mashups statistics and classification web site, as of April 2009, it listed 4,000 mashups (composition applications) and more than 1200 different web-delivered services.

Composition of web-delivered services is not trivial and there are some supporting environments for their development [2][3]. These environments always include graphic tools for composition design and analysis as well as runtime frameworks for composition deployment and operation. The tools can run either in a web browser or as a standalone program and the frameworks can run at the client side or the server side. Then, the environments can be divided into four types, as shown in Table I:

TABLE I WEB-DELIVERED SERVICE COMPOSITION ENVIRONMENTS

	Client-side Frameworks	Server-side Frameworks
Browser-based Tools	Microsoft Popfly, Intel Mash Maker, QEDWiki	Yahoo! Pipes, Sharable Code, Damia
Standalone Tools	Adobe Durango, Proto, Openkapow	ActiveBPEL, BPEL4J, Bite

In practice, comparing with other types, the environments running both tools and frameworks in web browsers, called browser-based composition environments in this paper, have significant advantages. First of all, although the web browser initially was designed as a pure thin client to display web pages, it has already become much "richer" and offers a set of powerful built-in and plug-in mechanisms, such as dynamic HTML and JavaScript engines. The browser is capable of hosting the whole service composition environment. Second, the browser-based environments have all advantages of web applications. For example, developers are able to use these environments in web browsers without any downloads, installations and upgrades. The environment itself and service compositions have real portability of "write once run anywhere". Third, the browser-based environments distribute the work load of development and service compositions operation from a central server to every user's browser. Hence, they never suffer from the scalability problems caused by mass users. Last but not the least, composition of web-delivered services always requires a lot of fine-gained and user-unperceivable communications between the tools and frameworks for rich user experience, e.g. WYSIWYG (What You See Is What You Get). Locating the tools and frameworks in the same browser can get better performance and reliability.

User experience is the dominant rationale for browser-based service composition environments. Since different environment vendors have different understanding of users and their experience, they provide very different environments, e.g. different look-and-feel, different service components and different composition styles. Besides these GUI (Graphical User Interface) features, environment vendors have to handle many common problems for service composition, including service access, component management, interaction enablement and browser compatibility. Poor solutions for these common problems definitely put negative impacts on user experience. Since browser-based service composition environments are just in a very early stage, they pay more attention to bringing service composition capability to end users and offering end user-friendly interface. The solutions for common problems are private to the environments and tightly coupled with their GUI. Obviously, it is hard and even impossible for a single vendor to produce optimal or best-of-the-breed solutions. But the monolithic and private design and implementation of each environment prevent vendors from sharing and collaborating on the private solutions. On the other hand, building up new environments has to implement these common solutions again and again. It increases the cost of improving user experience by new GUI features.

In this paper, we propose a new type of middleware, which is embedded in the browser and encapsulates the solutions of common problems in service composition development and operation. It provides an open way for producing the optimal solutions in most cases, i.e. different service composition environments preferred by different users can be constructed easily in high qualities. The prototype of the middleware, called Internetware Client Middleware:

- provides a component model and a container. The components invoke web-delivered services and construct user interface (UI). A well-defined interface provided by the component model ensures that the components can be used in more than one composition easily. All components are managed by the container;
- offers a composition model and enablement mechanisms. The composition model is event-based, since the event-based style is well suited to service composition in the browser [2]. To implement this composition model, the middleware provides several mechanisms, including a unified event model, an light-weight event bus and two types of connectors;
- encapsulates a set of mechanisms for web-delivered services access. The mechanisms solve the common problems of interactions between the browser and web-delivered services. They include the web-delivered service handler, cache handler, cross-domain handler, HTTP-push handler and OAuth authentication handler;
- can be executed in most modern web browsers, including IE, Firefox, Opera and Safari. The middleware handles many differences across browsers, so that developers can achieve the browser compatibility with little effort.

The middleware exposes all functionalities through a set of easy-to-use APIs. We implement a browser-based service composition environment, iMashup[1], based on these APIs, and compare it with some other environments. The comparison and evaluation results demonstrate the values of this middleware, i.e., better and reusable solutions for common problems in different environments.

The rest of the paper is organized as follows. Section 2 provides the overview of the browser middleware. Section 3 and Section 4 respectively discuss the implementation of the component container and the communication mechanisms. Section 5 presents evaluation results. Finally, we provide some discussions in Section 6 and conclude this paper in Section 7.

II. BROWSER MIDDLEWARE OVERVIEW

In this section, we provide a general overview of the proposed browser middleware, as shown in Figure 1. The implementation details are elaborated in section 3 and 4.

The browser provides the hosted mechanisms, such as HTML engine, JavaScript engine and HTTP protocol handler. Browser plug-ins (e.g. Flash and Google Gear) offer a number of useful mechanisms not included in the hosted mechanisms, such as local data storage. All of these mechanisms form the basis of our middleware.

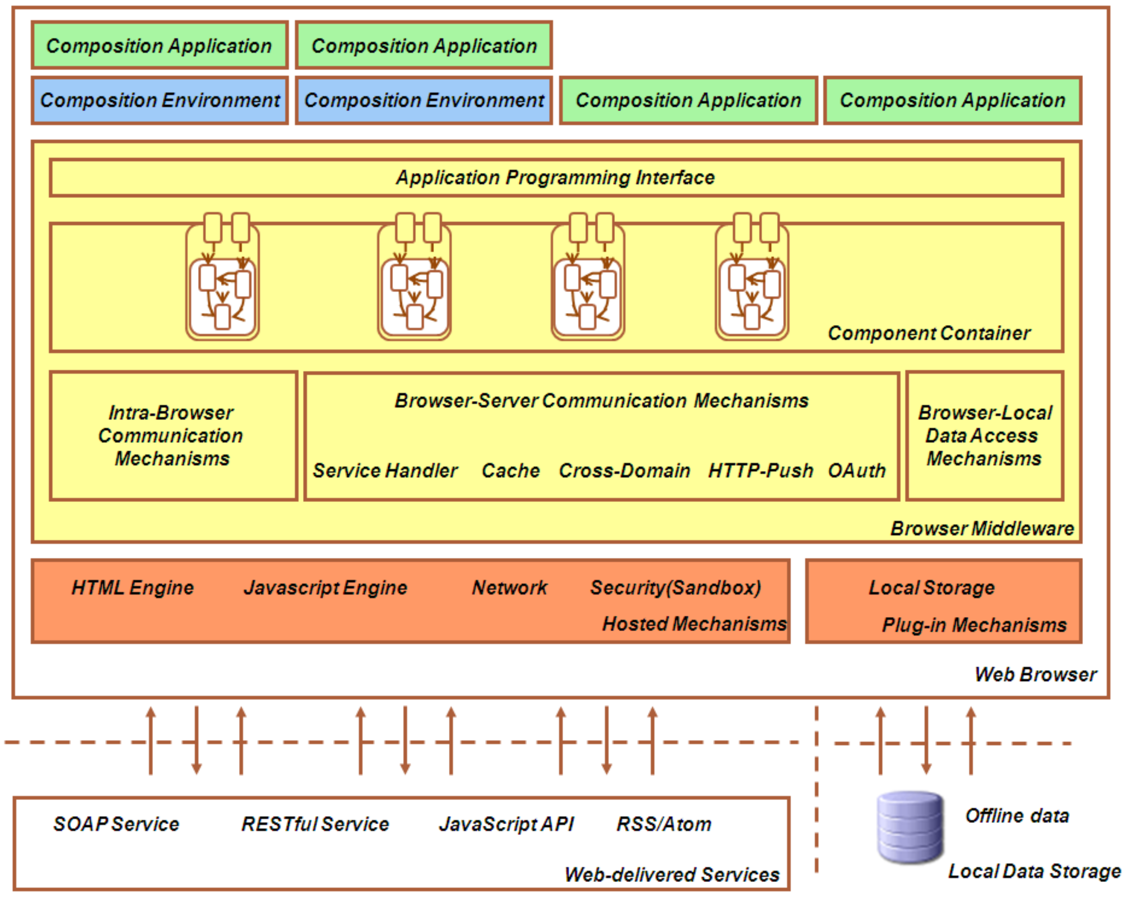
The browser middleware is built on the top of hosted and plug-in mechanisms. It is implemented with JavaScript which is the most used programming language in web browsers. It is also based on Dojo JavaScript framework, since the framework enhances JavaScript language with powerful object-oriented support. Dojo also handles many differences across browsers, and hence we can achieve the browser compatibility with less effort hence.

The container is in charge of component management. Its responsibility is to manage the definitions and the instances of component. The container need not consider the concurrency control of component instances, since the JavaScript engine in the browser runs in single-thread.

The communication mechanisms assist components to communicate with other components inside web browser, web-delivered services on servers and local resources, because the middleware not only manages the components but is also in charge of the communication both inside and outside the browser. Depending on the different communication types, these mechanisms can be divided into three parts:

- **The intra-browser communication mechanisms** offer some mechanisms to implement an event-based composition model which makes components communicate with others in the same web browser.
- **The browser-server communication mechanisms** provide the solutions of common problems for communication between the browser and web-

[1] It can be downloaded from http://code.google.com/p/imashup/



delivered services on servers. The mechanisms offer a series of handlers supporting the common capabilities for web-delivered services access.

- **The browser-local data access mechanisms** seek to provide some mechanisms to assist web applications to take advantages of local data storage.

The middleware provides a set of Application Programming Interface (API). Developers can build service composition applications directly based on these APIs. The time and effort of development can be saved, since developers can resolve many common problems by using the solutions encapsulated in the middleware and pay more attention to the business logic.

Furthermore, it is also easy to construct service composition environments with the middleware. In fact, the composition environments are a particular type of composition application, which facilitates developers or non-professional users to create their own applications. Since the middleware provides most functions an environment required, the main work of environment implementation is building GUI.

III. Implementation of Component Container

A. Component Model

When assembling services in the browser, developers use the data or logic of services and create corresponding UI. A well-designed component model, which encapsulates the application logic of service access and UI, can facilitate reusability and ensures extensibility.

The components in the browser middleware encapsulate both the application logic and UI. These components are similar to traditional ones and consist of the interface and the implementation. Yet, in contrast with traditional ones, the interface of components comprises the UI (user interface) and the programming interface. The programming interface exposes application logic. The components interact with others through their programming interface. The UI responds to users' actions and invokes the corresponding functions in the implementation. The component model is shown in Figure 2.

The implementation of components adopts the Model-View-Controller pattern. The model implements application logic by invoking web-delivered services. The view is a fragment of HTML which is rendered and displayed during component instantiation. The controller manages the interaction logic between model and view.

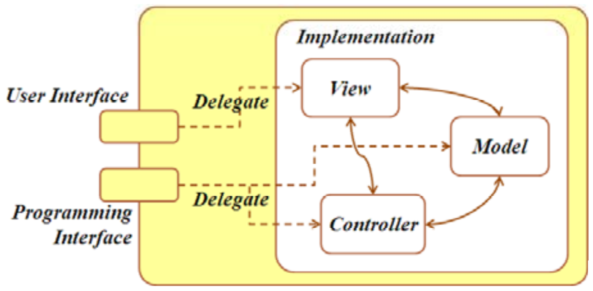


Figure 2. Component Model

The programming interface exposes the application logic of components. It consists of methods and events. Methods can invoke services and query and modify the component state. Events notify changes of the component state and can be published into the event bus. The UI is implemented by the view part of component. When developers assemble the components, they can determine whether the UI of components should be visible or not.

B. Component Management

The size of middleware determines the startup speed of applications and should be carefully controlled, since all files of middleware will be downloaded into the browser when users visit the applications built on top of the middleware. Hence, instead of downloading all component definitions at the start, the container downloads them on demand, i.e. downloads a definition only when applications use this component. The on demand definition download is carried out in the following steps:

- A request to instance creation API is arrived at the container;
- The container checks whether the definition of required component is downloaded or not;
- If the definition is already downloaded, the container creates a new instance;
- If not, the container will first construct a callback method to create an instance and subscribe the ComponentDefinitionDownloaded event with this method;
- And then the container converts the full name (with namespace) of this component into the path of its definition on the server and downloads it;
- After downloaded, the ComponentDefinitionDownloaded event will be triggered;
- Then the callback method will create a new instance. Moreover the container will maintain the downloaded definition for future use.

The container also offers the APIs for component instance retrieval, modification and deletion. Developers can manage instances through these APIs.

IV. Implementation of Communication Mechanisms

A. Intra-Browser Communication Mechanisms

Our browser middleware provide an event-based publish-subscribe composition model. The event-based composition model is well suited to browser-based composition environments. In web-delivered service composition, there are two major styles for component orchestration, flow-based and event-based: the flow-based style defines the orchestration in sequencing or partial order among components, while the event-based style uses publish-subscribe models [2]. The event-based style is suited to browser-based service composition, due to the nature of the browser is strongly event-based. The components in the browser encapsulate a lot of events. The start, change and finish stage of many user interactions and asynchronous operations, such as buttons clicked and Ajax called, are all notified by events. When developers assemble components, a typical composition scenario is that a component invokes a function to respond to an event published by another component.

To implement the event-based composition model, the intra-browser communication mechanisms offer a unified event model for all component events and an event bus.

Event Model

The browser has its own DOM (Document Object Model) event model. Each DOM node includes numerous events, such as click, mouse-over, and so forth. When an event is triggered, an event instance will be created and passed to corresponding callback functions. It should be noted that many other asynchronous operations exist in the browser, such as Ajax http requests and timers, which do not use DOM event model.

To facilitate composition, the intra-browser communication mechanisms provide a unified event model for all events of components, regardless of user interactions or asynchronous operations. The event model comprises the name of event, the reference of source instance triggering this event and a hash map containing event parameters.

Event Bus

The intra-browser communication mechanisms provide an event bus which supports publish-subscribe event binding. To guarantee its performance, the event bus is implemented in a very light-weight manner.

The event bus has some channels which are set by developers. Each channel is actually a string whose value is the unique name of the channel. The events of components can be published to these channels. The components will record the mapping between their own events and channel names. Components can subscribe channels, i.e. binding the methods of components to the names of channels. When receiving a binding request, the event bus will record the subscribe method in a hash map with the name of the given channel as its key.

When an event is triggered, the component will check its event-channel mappings to determine which channel the event should be sent to, and then send it to the event bus with the given channel name. When an event arrives, the event bus will find all subscribe methods with the given channel name. After that the bus traverses the found methods and invokes them with the event as input parameter. Event flows through the event bus are shown in Figure. 3.

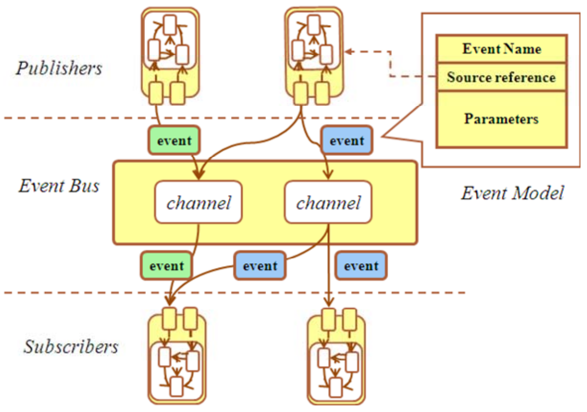


Figure. 3. Event Bus and Event Model

B. Browser-Server Communication Mechanisms

Web browsers do not support the communication with web-delivered services natively. The browser-server communication mechanisms include service handler, cache handler, cross-domain handler, HTTP-push handler and OAuth authentication handler.

Web-delivered Service Handler

Currently many heterogeneous web-delivered services (SOAP web service, RESTful service) exist over the Internet. Each of them has its own protocols. Nevertheless, the browser does not natively support all of these protocols.

The web-delivered service handler encapsulates the details of protocols and makes these details transparent to developers. Generally speaking, given a service interface description, the web-delivered service handler dynamically generates a JavaScript class to encode the request message, send it to the appropriate end-point and decode the response.

Service Cache Handler

Web-delivered service composition allows developers to create more complicated service composition applications in the browser. Due to the increasing complexity of application logic, the data schema and data manipulation become more complex as well. As a result, developers begin to build complicated data models of applications in the browser to simplify the development.

Although the browser already supports a simple page-based cache, composition applications require more complex cache due to the complexity of data model. The service cache handler allows developers to indicate which data the application should cache, and which caching policy it should use.

Cross-Domain Handler

A major limitation of web-delivered service composition in the browser is the cross-domain limitation. It prevents scripts running on pages from accessing services on other sites in different domains [9]. Fortunately, there are already some feasible solutions, such as JSONP, server proxy and plugin-based approach.

The cross-domain handler supports common cross-domain approaches. When developers need access cross-domain services, they should decide which solution to use and make some corresponding configurations. And then they can access cross-domain services as if the services were in the same domain, with all implementation details of solutions encapsulated by the cross-domain handler.

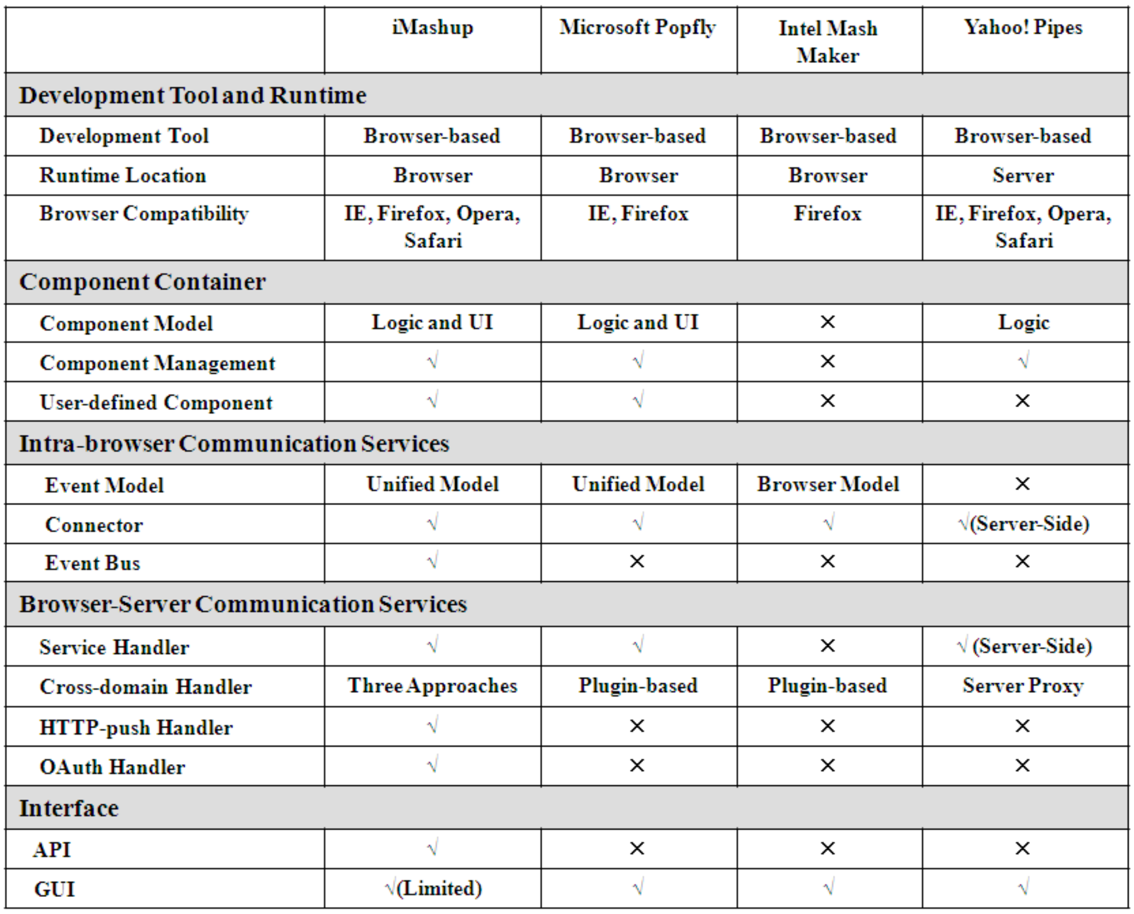
HTTP-Push Handler

In some applications, real-time dynamic web data such as chat update, stock tickets and auction updates need to be propagated to users as soon as possible. Therefore, these applications require delivery of asynchronous messages from the server to the browser. The technique is often described as "HTTP-push". However, the browser’s request/response architecture prevents servers from pushing real-time data [7].

Bayeux protocol [8] supports HTTP-push and has been implemented by many web servers, such as WebSphere and Glassfish. The HTTP-push handler offers a Bayeux protocol implementation in the browser and assists developers to create push-style service composition.

OAuth Authentication Handler

Though web-delivered service composition empowers developers to create applications by assembling existing data from different web sites [3], many data in the web sites are protected through username-password and cannot be visited without authentication. OAuth is a protocol which focuses on publishing and interacting with protected data.



	iMashup	Microsoft Popfly	Intel Mash Maker	Yahoo! Pipes
Development Tool and Runtime				
Development Tool	Browser-based	Browser-based	Browser-based	Browser-based
Runtime Location	Browser	Browser	Browser	Server
Browser Compatibility	IE, Firefox, Opera, Safari	IE, Firefox	Firefox	IE, Firefox, Opera, Safari
Component Container				
Component Model	Logic and UI	Logic and UI	×	Logic
Component Management	√	√	×	√
User-defined Component	√	√	×	×
Intra-browser Communication Services				
Event Model	Unified Model	Unified Model	Browser Model	×
Connector	√	√	√	√(Server-Side)
Event Bus	√	×	×	×
Browser-Server Communication Services				
Service Handler	√	√	×	√ (Server-Side)
Cross-domain Handler	Three Approaches	Plugin-based	Plugin-based	Server Proxy
HTTP-push Handler	√	×	×	×
OAuth Handler	√	×	×	×
Interface				
API	√	×	×	×
GUI	√(Limited)	√	√	√

The OAuth authentication handler encapsulates the authentication process of OAuth, since the process of OAuth is a bit complex and confusing to be implemented manually. Developers only need to provide OAuth request URLs, consumer key and secret, the handler will carry out OAuth authentication automatically. If users grant access, the composition application can use these protected data for further composition.

V. EVALUATION

To evaluate our browser middleware, we implement iMashup, a web-delivered service composition environment based on the middleware, and compare it with other environments. iMashup is built on the top of the browser middleware. The size of iMashup is 603 KB while the browser middleware is 566 KB. To our experience, developing iMashup is a relative easy and simple work.

The evaluation platform consists of an Intel Core 2 Duo CPU at 1.80 GHz with 2GB of RAM and running the Windows XP SP3. All experiments are run in four popular browsers: Internet Explorer 8.0.6001, Firefox 3.0.6, Opera 9.62 and Safari 3.1.2. We focus on three questions: How does the capability of iMashup compare with other service composition environments? What is the overhead of iMashup? What is the scalability of iMashup, which determines how complex composition the environment can handle?

A. Capability of iMashup

We compare the capability of iMashup with Microsoft Popfly, Intel Mash Maker and Yahoo! Pipes, as shown in Table II. From Table II, we find the functionalities of iMashup are richer than other three mashup environments, since iMashup benefits from the capability of the browser middleware.

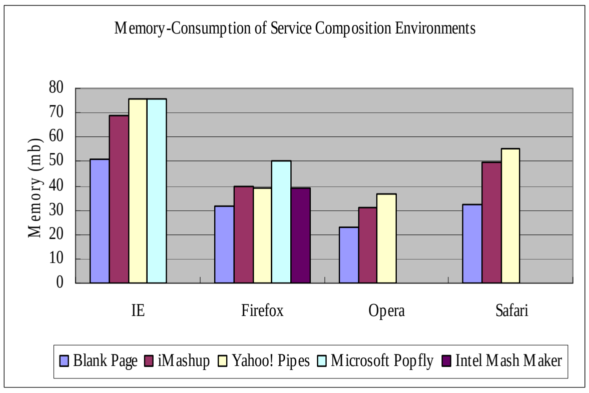
B. Overhead of iMashup

For our second question, the overhead of iMashup, we measured the size of files downloaded and the memory-consumption. We also compare the result with the overhead of other three composition environments.

Figure. 4 a) shows the download size of iMashup and other environments. iMashup is the biggest mainly because the browser middleware is relatively big. It is a typical result when we compare applications with and without middleware. There are two main reasons: 1) the capability comparison in the previous section indicates that iMashup provides richer functionalities than the other three.; 2) the other three environments listed are all real products, and then their sizes are carefully compressed and minimized. Comparing with them, iMashup is still a prototype without thorough optimization.

Composition Environment	Download Size (kb)
iMashup (Middleware)	603 kb (566kb)
Microsoft Popfly	347 kb
Intel Mash Maker	259 kb
Yahoo! Pipes	112 kb

a) Download Size



b) Memory-Consumption

Figure. 4. The Overhead of Browser-based Service Composition Environments.

The size of our middleware is a bit large, which means iMashup may start up slower than other environments. Yet Figure. 4 b) indicates the memory-consumption of iMashup is even smaller than the others. We cannot accurately analyze the causes because the source code of the other environments is not opened. However, we speculate on some of the reasons: 1) the middleware provides several communication handlers, such as cross-domain handler, which can be shared by many component instances. While the other environments pay more attention to end user service composition and their GUI, therefore, they may not well abstract and encapsulate these handlers. For example, maybe each instance has its own cross-domain handler, which consumes more memory. 2) the other environments use some powerful graphical technologies, such as SVG and VML, for more beautiful GUI. These technologies may also cause more memory-consumption.

C. Scalability of iMashup

The scalability problems of a service composition environment relate to the number of users, the number of instances and the complexity the composition [2]. As we mentioned above, iMashup, being a browser-based environment, never suffers from scalability problems caused by a large number of concurrent requests of users, since it is executed on the client side. Therefore we consider scalability of iMashup from two perspectives:

- What is the memory-consumption of component instances in the container?
- What is the performance of the event bus?

First, we measure the changing memory-consumption of iMashup with increasing numbers of component instances. We test a typical component, a Google Weather component calling RESTful service.

As Figure. 5 shows, at worst (with IE) the memory-consumption of 200 instances is still lower than the consumption of Gmail, which is a widely used web application with complex logic executed in the browser.

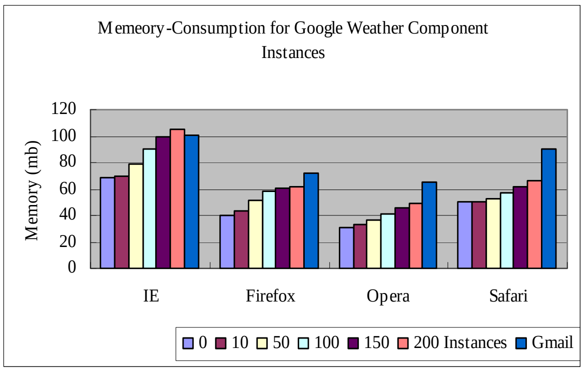


Figure. 5. The Memory-Consumption with an Increasing Number of Component Instances. The rightmost items in the charts are the memory-consumptions of Gmail.com as a reference.

Second, we measure the performance of the event bus. For this measurement, we set a one-to-many event binding: one event publisher and multiple event subscribers. And then we trigger an event and measure the time spent from the event triggered until the last subscriber receives it.

Figure. 6 shows the time spent of blank subscription methods invocations when an event is triggered. In the worst case (with IE), our event bus still routes the event extremely fast, i.e. handling 25,000 subscribers within 350 milliseconds.

The scalability of iMashup still can be optimized. However, from our experience and the statistics data from ProgrammableWeb.com, even the most complicated web-delivered service composition includes far less than 200 components and 500 subscribers. Thus, we believe the current scalability of iMashup is well enough.

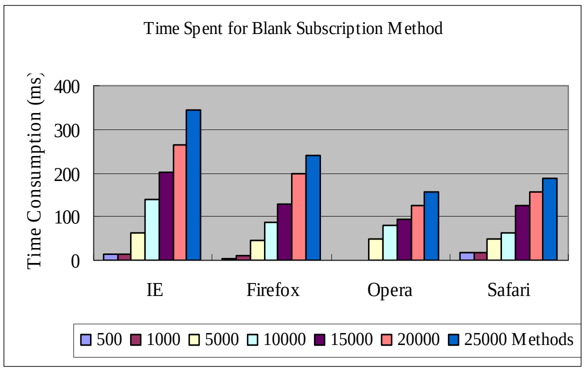


Figure. 6. The Time Spent with an Increasing Number of Subscription Methods.

VI. DISCUSSION

Though we have observed that the middleware does benefit service composition in the browser, there are still several open issues to further address.

First, we will further minify the size of browser middleware. In fact, such size is trivial for web browsers in PC but a bit large for web browsers in mobile phones. We are trying to separate the core of middleware into several independent parts and make them be able to be downloaded on demand.

The evaluation indicates the performance of the event bus is highly reliant on the complexity of subscription methods, i.e. complicated methods may cause poor performance. As we discussed, it is mainly because all subscription methods are executed within a single thread. Fortunately, some browser plug-ins support multi-threads JavaScript. We will try to make subscription methods run in different threads and believe it can significantly improve the performance of the event bus.

The communication mechanisms are composed of three parts. However, the third part, the browser-local data access mechanisms, is still under development. These mechanisms aid developers to bring the offline capabilities into composition applications. They include a simple object-relation mapping framework, which helps developers to store data objects into local databases, and some handlers which can detect and resolve conflicts between server and local data.

Last but not the least, although iMashup has richer capabilities, up to now, it only includes a few built-in components. Consequently, the adoption of iMashup is still limited, since developers should build required components by themselves. Furthermore, iMashup does not provide user-friendly GUI for all features of the middleware yet. Developers must still code a little to use some features, such as service cache and HTTP-push handler. Therefore, an important next step involves enriching the built-in components and making iMashup support the missing features.

VII. CONCLUSION

The web-delivered service composition environments embedded in web browsers are becoming popular since their advantages such as in user experience and scalability. These browser-based environments have to handle many common problems for service composition.

In this paper, we present the common problems for web-delivered service composition in the browser. We also propose a new type of middleware embedded in the browser, which encapsulates the solutions of these common problems and facilitates the development of composition applications in the browser. Finally, we implement a service composition environment, iMashup, based on the middleware. We evaluate iMashup by comparing it with some other popular environments.

Acknowledgement. This work is sponsored by the National Key Basic Research and Development Program of China (973) under Grant No. 2009CB320703; the National Natural Science Foundation of China under Grant No. 60821003 and 60873060; and the High-Tech Research and Development Program of China (863) under Grant No. 2009AA01Z16.

REFERENCES

[1] E. Michael Maximilien, Ajith Ranabahu, Karthik Gomadam, An Online Platform for Web APIs and Service Mashups. IEEE Internet Computing, 2008

[2] Jin Yu, Boualem Benatallah, Fabio Casati, Florian Daniel, Understanding Mashup Development. IEEE Internet Computing, 2008

[3] Volker Hoyer and Marco Fischer, Market Overview of Enterprise Mashup Tools. International Conference of Service Oriented Computing, 2008.

[4] Qi Zhao, Gang Huang, Jiyu Huang, Xuanzhe Liu, Hong Mei, Ying Li, Ying Chen. An On-the-fly Approach to Web-based Service Composition. Proceedings of IEEE Service Congress and International Conference on Web Services, 2008.

[5] Qi Zhao, Gang Huang, Xuanzhe Liu, Jiyu Huang, Towards a Component Model for Web-based Service Composition. Journal of Frontiers of Computer Science and Technology, 2008.

[6] David E. Simmen, Mehmet Altinel, Volker Markl, Sriram Padmanabhan, Ashutosh Singh, Damia: Data Mashups for Intranet Applications. ACM's Special Interest Group on Management Of Data, 2008

[7] Engin Bozdag, A. Mesbah, A. Deursen. Performance Testing of Data Delivery Techniques, for AJAX Applications. Journal of Web Engineering, 2008.

[8] Bayeux, http://svn.cometd.com/trunk/bayeux/

[9] Cross Domain Ajax: a Quick Summary. http://snook.ca/archives/javascript/ cross_domain_aj/

[10] Intel Mash Maker, http://mashmaker.intel.com/

[11] Microsoft Popfly, http://www.popfly.com/

[12] Yahoo! Pipes, http://pipes.yahoo.com/