Web Service Mashup Middleware with Partitioning of XML Pipelines

Eric Wohlstadter, Peng Li, and Brett Cannon
University of British Columbia
{wohlstad, lipeng, drifty}@cs.ubc.ca

Abstract

Traditionally, the composition of Web services to create mashups has been achieved by using an application server as a mediator between a client browser and services. To avoid this bottleneck, mashups are sometimes implemented so that Web service composition takes place directly from the end user's browser. Creating such implementations is difficult because developers must manage the separation of software into various distributable pieces, in different languages, and coordinate their communication. In this paper we describe a middleware for managing Web service mashups in a disciplined, and flexible way. We build upon the established abstraction of XML pipelines, but describe a new approach for selectively partitioning pipeline components between a browser client and application server. We provide a performance evaluation for a common mashup application scenario.

1. Introduction

One of the uses of Web services is in the construction of Web applications. The term *mashup* [14, 15] is used to describe applications where Web services are composed with traditional Web content and presented to the user. Traditionally, the composition of Web services to create mashups was primarily achieved on an *application server*, the same host which provides user interface elements in the form of HTML and JavaScript. This application server acts as a mediator between the client and third-party Web services: composing resources from services and then presenting the information to the end-user's Web browser. Figure 1 provides an illustration.

Unfortunately, the use of the application server as a network intermediary can become a significant performance bottleneck. This is because many Web applications are designed to support a large number of concurrent users. Adding the cost of communication and processing between the application server and a Web service – for each client request – can result in a large increase in processing latency. As described in [7], "The drawbacks of this approach are that the content makes several unnecessary round trips, reducing performance; the proxy can become a choke point, limiting scalability".

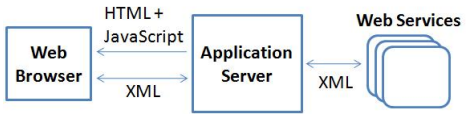

Figure 1. Mediator Architecture for Mashups

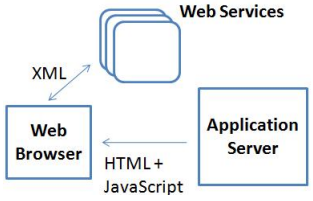

Figure 2. Client-Side Mashup Architecture

Recently, there is a growing trend where some or all of the Web service composition takes place directly from the end user's browser [3, 4]. Here the application server acts as a logical intermediary, by providing scripts and stylesheets to the client for the purpose of "mashing-up" services. The browser will contact Web services directly as in Figure 2.

Although the client-side architecture helps mitigate a performance problem, it can introduce its own problems in the design and maintenance of mashup applications.

First, developers must manage the separation of mashup processing into various distributable pieces, in different languages (e.g. XSLT, XQuery, object-oriented, etc.), and coordinate their communication. This can result in an implementation with many components "glued" together in an ad-hoc fashion. These kinds of implementations are difficult to debug and evolve. In the traditional mediator architecture these problems can be solved by architecting an implementation as a collection of XML processing pipelines.

Pipeline languages such as XProc [11] and Cocoon [1] serve as a unified architectural description for managing components in different languages. However, these pipelines do not directly support the case where components are executed on both the browser and application server. In

this paper we describe an approach to leverage a pipeline architecture even when some processing logic is executed on a client's browser. This is achieved through a middleware which partitions the pipeline processing across the browser and application server.

The second problem with deploying a client-side architecture is caused by the high degree of heterogeneity exhibited by client devices and platforms. So, although performance can be improved by executing components on the client, it must be guaranteed that each client can execute such components. Furthermore, other non-functional requirements such as the security preferences of a client need to be taken into consideration. For this reason, we have worked to provide *client-specific partitioning* in the middleware. This allows the placement of components to depend on the capabilities of specific clients using the application.

Support for distributing pipelines across hosts has been researched in other domains [8]. However existing work has not considered distributing pipelines between the Web browser client and Web application server in an end-user Web service mashup application. This new setting introduces new technical hurdles which must be overcome to realize this scenario. This is because clients and servers in a mashup must be considered asymmetrically in terms of their capabilities and requirements. Our technical challenge then is to devise a middleware to mediate the requirements of clients and servers and to layer the abstraction of pipeline data-flow on top of the traditional client/server interactions.

In this paper we describe a middleware framework called Metro for constructing Web service mashup applications, where part of the pipeline is executing on an application server and part of the pipeline is executing in the client's Web browser. We provide a performance evaluation of Metro for a common mashup application use-case scenario.

The rest of the paper is organized as follows: Section 2 presents a motivating example, Section 3 presents related work and background, Section 4 presents details of Metro, Section 5 presents a performance evaluation, and Section 6 presents future work and conclusion.

2. Motivating Example

Consider a mashup application which is provided by an application server and makes use of some third-party Web service. In this example, the application server loads some client-specific information, uses it to formulate a request to a third-party Web service, transforms the retrieved response through some number of processing steps, and then returns the resulting output to the client.

An example pipeline is shown in Figure 3. We use the syntax of XProc[1] because it is the language supported by Metro. Similar to the UNIX pipe-and-filter architecture, an

[1] XProc is a recent W3C Candidate Recommendation.

```
1 <pipeline>
2   <load name=favs
3         href= http://myapp.com/... />
4     <output port=fOut/>
5   </load>
6
7   <http-request name=eBay>
8     <input port=fOut/>
9     <output port=eOut/>
10  </http-request>
11
12  <load name=style>
13        href= http://myapp.com/... />
14    <output port=sOut />
15  </load>
16
17  <xslt name=xslt>
18    <input port=eOut/>
19    <input port=sOut/>
20    <output port=stdout />
21  </xslt>
22 </pipeline>
```

Figure 3. Example XProc Pipeline: XML elements and attribute names shown here are XProc keywords. The attribute values and the configuration of elements are specific to the example.

XProc `pipeline` consists of a set of processing components connected by pipes. Each `pipe` connects an `output` from one component to the `input` of another. Different from UNIX, an XProc pipeline can manage multiple inputs and outputs for each component.

The example `pipeline` (line 1) is made up of four component instances of three types: `load`, `http-request` and `xslt`. The `load` component type is used to load XML data from a statically specified URL. In the example there are two instances. The first (line 2) is used to load information regarding some user's account information. This "favorites" information is used to determine a query string sent to a Web service (in this example eBay). The second (line 12) is used to load the XSLT stylesheet which will be used to transform the response from eBay. The `http-request` (line 7) component takes input from the `favs` component and contacts eBay through a REST request. Finally, the `xslt` component (line 17), takes input from both of the previous components and outputs the final transformed message.

Although this pipeline is not complex, it can be used to highlight important design decisions with respect to the partitioning of components. For our example, we consider benefits and tradeoffs of potentially executing the `eBay` and

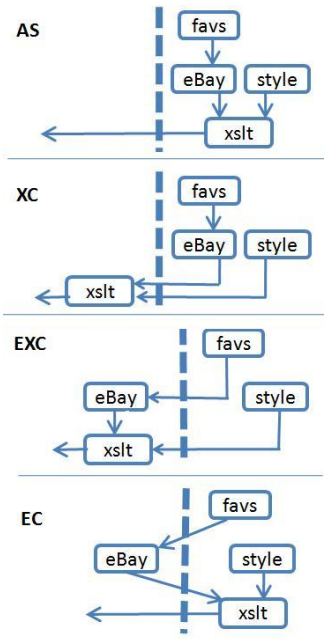

Figure 4. Example Partitioning Configurations: Used in Section 2 for motivating client-specific partitioning and again in Section 5 for performance evaluation.

`xslt` components on the client.

We call the decision of where to execute each component a *partitioning configuration* (or just configuration). Figure 4 demonstrates four possible configurations for the example. In the figure the dashed line represents the division between client and server for each configuration.

Considering interoperability, a developer may prefer configuration `AS` over the others. `AS` requires no assumptions about the client's platform. It corresponds to the traditional mediator architecture, so the scalability of the server could be an issue.

To enhance the scalability of a Web application, a developer may prefer clients to contact services directly. This decision justifies the use of either configuration `EXC` or `EC`. Considering latency of pipeline processing, `EXC` appears like the clear "winner", because the client does not need to make an additional round-trip and route the Web service response back to the server. However, `EXC` requires the client to support XSLT (and the correct version), while `EC` does not. So, `EC` may be a useful tradeoff between performance and interoperability.

Still, a client might not be capable of contacting Web services directly for security reasons. Currently the only way, supported by all popular browsers, to contact multiple third-party XML-based Web services is by leveraging a plugin (such as Adobe Flash). This is because of a security policy imposed by browsers called the *same-host restriction* [7]. Flash (and some other plugins) provide an extended security model which is safe [5] and provides for third-party Web service access[2]. However, while some plugins like Flash are popular, they might not be supported by all clients.

In that case, the configurations `AS` and `XC` could be used. If the client supports XSLT, the server can off-load the burden of message transformation to the client using `XC`, while still serving as an intermediary for security reasons.

This example demonstrates the kinds of decisions that need to be made when partitioning components in a mashup application. The answers are not "black and white" but depend on the capabilities supported by different clients. Managing the execution of different configurations of components, for different clients, serves then as our motivation to explore a disciplined, yet flexible, approach to construction of XML pipelines for mashup applications.

3. Related Work and Background

Composable Network Services Previous work on active networks has inspired research on frameworks for automated composition of network intermediaries. For example, Active pipes [8] or CANS [6] can be used to stream data across components on disparate network hosts; the binding of components to hosts can be determined dynamically based on network conditions. This is so that pipeline data can be routed along the most efficient path. However, this work did not consider integrating pipelines into a Web application architecture where clients and servers must be treated asymmetrically in terms of both capabilities and requirements.

Context-Aware and Adaptive Services Research in context-aware [2] computing provides for the adaptation of delivered Web content. This is especially important for the case of mobile [13] clients, where the operating environment of the client can change over time. The research in this paper does not directly address context-awareness and adaptivity because we only consider the client's platform and not temporally sensitive information such as location. Furthermore, we consider client-specific partitioning for the purpose of migrating processing components only and not for adapting content.

Automatic Application Partitioning Closely related to our research is the work on automatic application partitioning. Projects such as J-Orchestra [10] provide solutions for automatically distributing software components across network hosts. Middleware support for managing communication between components which have been partitioned on different hosts is handled automatically. In this way some

[2] Another way to deal with this problem is to rely on JavaScript Object Notation (JSON). However this requires each third-party service to support JSON as an alternative to XML

existing applications which were written originally for a single host can be transformed into distributed applications. This work did not address coordination of message-oriented pipelines and assumed that components communicated using synchronous procedure calls.

Business Process Orchestration Web services can be implemented using orchestration languages such as BPEL. While these languages are not directly comparable to the XML pipeline languages discussed in this paper, it is useful to place both in perspective. BPEL is generally used to coordinate multiple activities of stateful, long-running, business processes. BPEL uses a model where concurrency is explicit through the use of `flow` activities. Compared to BPEL, XML pipelines are used to implement short-lived message processing steps. Frequently, components are stateless and used specifically to transform message content.

XML Pipeline Background Here we provide a brief description of the implementation details of a typical XML pipeline processor. This is useful to understand some of technical details of Metro.

XML pipelines can be implemented by instantiating the pipe abstraction as a FIFO queue of XML messages. Whenever at least one of the inputs for a component is empty, the component is considered *blocked* and cannot execute. Whenever all of the input pipes for a component contains at least one message, the component can be scheduled to execute.

When a component executes, one message from each of its input pipes will be dequeued and provided to the component implementation. The component can perform message processing and place messages on its output pipes. This may trigger the execution of some other component and then execution will continue.

The topology of the pipeline forms a directed, acyclic, graph (DAG). This can be viewed as a dependency graph (i.e. task graph) where a component d depends on c if there is a directed edge (c, d). Components with no dependencies are called *sources*; components with no dependents are called *sinks*.

We use two metrics of components in a pipeline to help us schedule execution in Metro (in Section 4.3). Assuming that components can freely run concurrently and each component has unit execution cost, we can determine the *earliest start time* of a component by finding the length of the longest path from any source to the component. Obviously, components do not always take a uniform time to execute, so this is useful to find relative orderings.

Second, if one component, d, is assigned a start deadline (i.e. time before it must start executing), we can determine the *latest start time* of some other component, c, that d depends on, by finding the length of the shortest path from c to d. Now, c must start at a time earlier than the start deadline for d minus the length of the path. Both this metric and the previous one can be determined in linear time using standard algorithms [9].

4. Middleware Details

Developers can make use of Metro to partition the execution of components between the client and server. Our server middleware is implemented in Java and the client middleware in JavaScript. No special support for Metro is needed by browsers, they only need to support JavaScript. Both client and server middleware include a custom XProc interpreter which extends XProc with our support for partitioning pipelines. The use of Metro by a mashup is transparent to the end-user.

4.1 Overview

A Metro developer will begin with a standard XProc pipeline, such as the one in Figure 3, which will serve as the basis for different configurations. Developers use Metro-specific XML elements to annotate the pipeline with instructions for partitioning. The developer statically maps a pipeline to a URL, but the partitioning configuration is determined when a client contacts the URL at run-time.

During execution we use a *batching* approach to reduce the number of requests made by a client. First, the client may initiate execution of some source components. If a client-side component places a message on a pipe destined for the server, this message is placed in a queue. When no more client-side components can execute (i.e. they are blocked), the messages in the queue are batched into an HTTP request.

When the server receives a batch request, it will unpack all the messages and dispatch them to the corresponding pipes on the server. The server will then execute components as normal. When a server-side component places a message on a pipe destined for the client, this message is placed in a queue. When no more server-side components can execute, the messages in the queue are batched and returned as the response to the HTTP request.

Now when the client receives a batch response, it will dispatch the messages similarly as described for the server. Execution continues on the client, which may either result in additional requests for the server or the execution of sink components.

We refer to the transport of each message batch, either in a request or response, as a *hop*. Metro schedules execution of components according to the number of hops which should occur before a component executes. We call this the *hop-time*. For example, consider the pipeline in the left-hand side of Figure 6. Here we see that a will execute after 0 hops (i.e. has a hop-time of 0), b and c have a hop-time

of 1, d and e have a hop-time of 2, etc. Note that although b takes no input from the client, it still requires one hop for the client to initiate pipeline execution on the server.

It may seem odd at first that d and e have the same hop-time since e depends on d. The first reason is that Metro is not a real-time platform for scheduling precise timing of distributed computations. We are only concerned with when components execute, in order to control the number of hops between client and server. Second, the way that intra-host components (e.g. d and e) coordinate with each other can be handled as in a traditional pipeline, so we ignore this detail.

Metro is designed to help developers manage (sometimes competing) non-functional requirements of mashup applications, where client and server may be asymmetric in terms of their goals and capabilities. In Section 4.2, we describe a mechanism to balance use of the server with client interoperability requirements. In Section 4.3, we describe a mechanism to balance pipeline processing latency specifically with server memory usage requirements. In Section 4.4, we describe a mechanism to balance use of the server with pipeline processing latency.

4.2 Client-Specific Partitioning: Managing server usage vs. interoperability

As motivated in Section 2, developers may want mashups to bypass usage of the server as an intermediary. However, this requirement must be balanced with the interoperability requirements of clients to be able to execute each component.

Developers can add a `client` element nested under any component declaration. This element declares that the component may be executed on the client. Execution is conditional upon two attributes: `requires` (described here) and `fallback` (described in Section 4.4). The `requires` annotation is used to express a predicate which determines if a component can execute on the client. We make use of XPath for expressing predicates. Predicates are evaluated at run-time when a client makes a request to the URL for a pipeline. Since we allow developers to annotate individual components with the `requires` predicate, a developer does not need to actually enumerate individual configurations (e.g. as we enumerated four of them in Figure 4).

These predicates can make use of parameter values which are collected by a script that runs in the client's browser. The parameter name/value pairs are exposed to the evaluation of predicates through a special XPath variable named `request`, as in Figure 5. Any information which is exposed by the client's browser or HTTP headers can be collected. Our middleware comes with a default script which collects the values of commonly useful parameters.

Some examples of parameters we currently collect are: `userAgent`, the identifier of the client's browser;

```
<client
  requires='request/flashVersion >= 8'
  fallback='false'
/>
```

Figure 5. `client` element for `eBay` component from Figure 4

`xsltVersion`, the version of XSLT supported by the client; `flashVersion`, the version of Flash support[3] (or none); `cookiesEnabled`, client's preference to allow for cookie storage; `domStorage` client's preference to allow client-side persistent document storage.

This collection of information can be extended by a developer, by adding some simple JavaScript. Using this client-specific information allows the specific configuration of a single pipeline to vary according to the capabilities of each client's platform.

4.3 Scheduling: Managing latency vs. server memory usage

In any pipeline, the order of component execution is constrained by the topology of dependencies. However, in general, this is an *under-constrained* problem: some pipelines will have a degree of freedom (i.e. slack) which we can take advantage of when scheduling the hop-time of component execution.

Our scheduling policy deals with client and server components differently to satisfy different requirements. Client components are scheduled to execute *as soon as possible*. This promotes a smaller latency for the pipeline to finish. Server components are scheduled to execute *as late as possible* without increasing the total number of hops. This promotes a decrease in the amount of time the server is required to store pipe queue messages in memory.

For example, consider the execution of component f on the left-side of Figure 6. It is scheduled for a hop-time of 3 but clearly we could have scheduled it to execute at hop-time 1. This is based on our policy which minimizes server memory usage at the expense of latency. If we execute f early, the output will need to be held by the server (as part of the user's session state), waiting for the rest of the pipeline to catch up. Consider that f might be producing a large database result set which is going to be filtered into a small set by g. In that case, we certainly would not want to hold on to it in RAM across multiple requests from a client.

On the other hand, consider the scheduling of component a. This component is scheduled to execute immediately, but we could have waited until hop-time 2. This is based on our policy that a client is likely to prefer components to execute early even if it means saving some session state

[3] Our current implementation uses Flash for cross-domain scripting

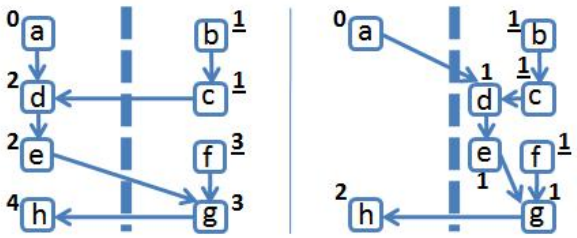

Figure 6. Pipeline Scheduling and Fallback: Figure is divided into two sides illustrating differences caused by choice of execution for d and e. Components are labelled with letter names. Integer label indicates the hop-time assigned by Metro. Underlined labels indicate components which might have slack for their hop-time.

in the browser. Scheduling components early makes their output available sooner. Since clients only maintain state for themselves, they are more likely to be concerned with speed in a browser application than memory consumption.

In the first step of our scheduling procedure, components which are reachable from a client-side source component (e.g. a, d, e, g, h) are scheduled to execute as soon as possible. These are the components in the figure whose integer label is not underlined. These components have no slack because they are either client-side components or directly in the flow of client-side components.

To compute this metric, we use the standard approach for determining an earliest start time (i.e. longest path from any source), but only count pipes which cross the network in measuring the path length. Essentially, these cross-network pipes are given a weight of 1 and all others a weight of 0. We call this the *hop-length*.

Any hop-length computed from a server-side source starts at a hop-length of 1, because the client always needs to at least contact the server for it to start pipeline execution. For example, d is given a hop-length of 2 in the first step of our procedure because: it is reachable from a, and the longest hop-length from any source node is 2 (the path which starts at b). Notice that d could note have executed any earlier.

In the second step of our procedure we consider the hop-times for the components that are not reachable from a client-side source (i.e. the components with an underlined label, b, c, f). The execution of these components is not pushed forward by any client-side components, so they might have slack in their hop-time.

This is done by considering the hop-times for components from step 1 as a deadline. Then, we use the standard approach for determining the latest start time for the components in step 2.

For example (left-side of the figure), g would be scheduled to a time of 3 in the step one. The hop-length from f to g is 0, so f is scheduled to a hop-time of 3. This is the latest hop-time it could execute without increasing the total number of hops for the pipeline.

Allowing different scheduling policies for different hosts in this way makes it possible to accomodate the varying requirements of clients and servers in terms of their memory usage profiles and processing latency demands.

4.4 Fallback: Managing latency vs. server usage

Executing components on the client is good for controlling usage of the server. However, a developer may also need to consider the topology of a configuration, to balance the benefit of client-side execution with the cost of the number of round-trips needed to execute the configuration.

Consider the execution of components d and e on the left-side of Figure 6. Their location causes an additional round-trip between the client and server. This can be seen by comparing to the right-side, where those two components are partitioned on the server.

A `fallback` attribute is a boolean value that specifies if a component with a `client` tag should fallback to executing on the server in the case that executing it on the client would add an additional round-trip. Essentially, it specifies which concern is more important to the developer: executing the component on the client or saving a round-trip.

This determination needs to be made at configuration setup time, after the `requires` attributes have been evaluated. This is because whether or not a component's placement saves a round-trip depends not only on the partitioning of the component but also of the components it is connected to. Consider a case where d was not declared `fallback` (hence justifying its placement as in the left-side of the figure). Suppose e was declared `fallback`. Here, e would still be executed on the client because changing the execution of e from client to server, in that particular case, will not actually save any round-trips.

For each group of client-side components scheduled for the same hop-time (e.g. d and e in the left-side of Figure 6), we determine if all of them are `fallback`. If all of them are, then they are executed on the server, essentially overridding the attempt to execute them on the client. For example, if both d and e were `fallback`, they would be moved to the server, as in the right-side of the figure.

In Figure 5, we see that the `eBay` component is not `fallback`. This will cause it to execute on the client as in Figure 4, configuration `EC`, even though it causes an additional round-trip in that configuration.

Using the `fallback` attribute ensures that developers have control of balancing the latency caused by round-trips with the advantage of client-side execution, even while determination of configurations is automated at run-time.

5. Performance Case-Study

To evaluate Metro's implementation, we examine the performance of the four example configurations from Figure 4. First, we describe our experimental setup. Second, we describe an experiment to compare the perceived end-user latency of pipeline execution for a single user. This experiment reflects the performance in the ideal situation where the server is only under the load of a single client. Finally, we describe an experiment to compare each configuration's affect on the scalability of the server under increasing client load. We hope to demonstrate that since the ideal situation cannot always be achieved, the flexibility of partitioning offered by Metro is useful.

5.1 Experimental Setup

For each experiment the server[4] is a 2.8GHz (Intel Pentium 4) machine with 2GB RAM. The client[5] is a 1.6 GHz (Intel Atom) machine with 1 GB RAM. The two machines were connected over a LAN. Thus there is no significant network latency between the client and server. This does not affect the experiment analysis since we will only focus on latency caused by processing on each machine. The connection to eBay was made from the University of British Columbia. Each entry for both tables of results shows an average[6] over 100 trials in milliseconds.

Each Web service request uses eBay's `FindItems` REST operation to search for 100 auction results matching the query keyword "ipod". The stylesheet used in the tests was obtained[7] from the eBay Web service online help.

For each configuration, we implemented a "hand-coded" version for performance comparison. By hand-coded we mean that the implementation does not make use of Metro middleware, but achieves the same functionality and partitioning of program logic. We made use of the same off-the-shelf components that we used to build Metro, but all of the "glue code" to hook components together was written by hand. We made sure this glue code did not do unnecessary buffering or parsing of XML message streams to ensure high performance. The hand-coded versions are represented in the results by using an `h` subscript.

5.2 Perceived End-User Latency

Figure 7 shows the latency which is perceived by the user when requesting a URL whose processing is implemented by one of the pipeline configurations (in each row of the table). By comparing the Metro results to the hand-coded results for each case (e.g. comparing `AS` to $\texttt{AS}_h$), we see the middleware introduces some small but most likely unnoticeable overhead.

	Latency (ms)
`AS`	1,075
$\texttt{AS}_h$	1,023
`XC`	1,030
$\texttt{XC}_h$	944
`EXC`	1,343
$\texttt{EXC}_h$	1,267
`EC`	1,546
$\texttt{EC}_h$	1,525

Figure 7. End-User Perceived Latency: Single client to server.

Requests/Sec.	10	20	40	80
`AS`	1,174	1,235	$6,631^{\dagger}$	$19,402^{\dagger}$
$\texttt{AS}_h$	945	993	$8,439^{\dagger}$	$18,234^{\dagger}$
`XC`	948	932	$4,334^{\dagger}$	$19,132^{\dagger}$
$\texttt{XC}_h$	845	989	$5,567^{\dagger}$	$19,834^{\dagger}$
`EXC`	63	67	75	93
$\texttt{EXC}_h$	22	44	40	76
`EC`	102	125	142	184
$\texttt{EC}_h$	56	76	149	155

Figure 8. Server Scalability: Multiple simulated clients to server. (†) For these results the response time was continually increasing over time.

The performance of the configurations which act more like a traditional mediator architecture (`AS` and `XC`) perform better than those which use the client-side architecture, in this experiment. This difference is important but must also be weighed against the affect that each configuration has on the scalability of the server hosting the pipeline. Keeping this in mind, we now explore this affect on scalability in more detail.

5.3 Server Scalability

In these experiments the server was put under the load of a varying number of client requests per second. Here the clients are simulated by making the requests from the same client machine using multiple threads. The simulated clients make requests to the server but do not use the response. Using this simulation rather than unique client machines does not adversely affect the results because we are

[4] Server software: Tomcat 6 (App Server), HTTPClient 3.1 (REST), Saxon-B 9 (XSLT), MySQL 5.1 (DB for `favs` information)

[5] Client software: Firefox 3 (Browser), flXHR (Cross-Domain Ajax)

[6] Standard deviation was no more than 26% of the average for all cases except those marked with the dagger superscript. However, deviation should be taken into account when comparing results which are close and small differences may not be meaningful given the deviation.

[7] ebay.custhelp.com/cgi-bin/ebay.cfg/php/enduser/std_adp.php ?p_faqid=1197

only focusing on the server performance in this experiment; we are not including processing time on the client as in the previous experiment.

Comparing the hand-coded and Metro versions, we can see there is not much overhead. The next point to consider are the results for the higher request rates in both AS and XC. These results can be considered as a failure for the server to manage this level of workload[8].

Since AS_h is essentially the traditional mediator architecture, this shows the kind of problems that could arise in that architecture. To deal with this problem, a developer could make use of Metro to partition part of the pipeline processing to clients, without creating an entirely different version of the application or investing in a higher performance (and cost) cluster of servers.

In some cases like EC and EXC each individual client might perceive some slowdown, as demonstrated in Figure 7. However, as shown in Figure 8, these configurations will also allow the server to maintain high throughput for larger client workloads. So, in the end clients may perceive better response time.

Still, as motivated previously, not all clients may be able to support configurations where the clients take on some part of the work. Web application providers may not want to simply reject service to these client "out of hand". Using Metro, Web application developers could take advantage of those clients who can support aggressive client-side partitioning, while still supporting clients who cannot. This is possible in a disciplined way building on the XProc description language with our support for client-specific partitioning; rather than managing different versions for different clients in an ad-hoc way.

6. Future Work and Conclusion

Supporting different clients in different ways motivates the consideration for supporting different service-level agreements for different pipeline configurations. For example, in the case that off-loading some functionality is beneficial to the server, the server could provide some incentive for clients who are able to support this. Incentives might come in the form of a lower-cost (for pay-per-use services) or in the form of relaxed usage quotas (for usage limiting services). In the future, we intend to investigate the management infrastructure necessary for coordinating service-level enforcement with Metro's pipeline partitioning.

In this paper we presented a middleware for managing XML pipelines distributed between a browser client and a Web application server. This was motivated by the emerging practice of composing (i.e. "mashing-up") third-party Web services directly from a client. Without middleware support this practice requires manual integration of components which may be written in different languages and which communicate between different hosts. XML pipeline languages, such as XProc, provided an initial solution, but we had to account for the technical mismatch between pipeline architectures and the traditional (i.e. HTTP client/server) Web application architecture. We demonstrated through a case-study of 4 partitioning configurations, the utility of our approach to flexibly adapt to the heterogeneities of different clients. Also, we demonstrated that we were able to manage these details without inducing much performance overhead.

[8] We have ensured that this was not caused by the eBay Web service. i.e. eBay itself was not affected by the increase in request rate and did not throttle our requests

References

[1] *Cocoon 2.2*. Apache Software Foundation.

[2] F. Daniel and M. Matera. Mashing up context-aware Web applications: A component-based development approach. In *Web Information Systems Engineering*, 2008.

[3] Duane Merrill. *Mashups: The new breed of Web app*. IBM Developer Works, 2006.

[4] Ed Ort and Sean Brydon and Mark Basler. *Mashup Styles*. Sun Developer Network, 2007.

[5] Flash 8 Documentation. *Allowing cross-domain data loading*. http://livedocs.adobe.com/flash/8/main/00001621.html.

[6] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. Composable, adaptive network services infrastructure. In *Proc. of Symposium on Internet Technologies and Systems*, 2001.

[7] Howell et al. Protection and communication abstractions for Web browsers in MashupOS. In *Proc. of the Symposium on Operating System Principles*, 2007.

[8] R. Keller, J. Ramamirtham, T. Wolf, and B. Plattner. Active pipes: Service composition for programmable networks. In *Military Communications Conference*, 2001.

[9] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4), 1999.

[10] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *European Conference on Object Oriented Programming*, 2002.

[11] N. Walsh, A. Milowski, and H. Thompson. *XProc: An XML Pipeline Language*. W3C Candidate Recommendation, 2008.

[12] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned scalable Internet services. In *Symposium on Operating Systems Principles*, 2002.

[13] S. Yang, J. Zhang, A. Huang, J. Tsai, and P. Yu. A context-driven content adaptation planner for improving mobile Internet accessibility. In *IEEE International Conference on Web Services*, 2008.

[14] J. Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding mashup development. *IEEE Internet Computing*, 12(5), 2008.

[15] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. A framework for rapid integration of presentation components. In *International World Wide Web Conference*, 2007.