Proceedings of the 2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design

Hermes: On Collaboration across Heterogeneous Collaborative Editing Services in the Cloud

Huanhuan Xia*, Tun Lu*, Bin Shao†, Xianghua Ding* and Ning Gu*
*School of Computer Science, Fudan University, Shanghai, China
Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China
Email: {huanhuanxia, lutun, dingx, ninggu}@fudan.edu.cn
†Microsoft Research, Beijing, China
Email: binshao@microsoft.com

***Abstract*—With the prevalence of cloud services, more and more editing tools are moving from client to cloud for collaboration. Google Docs, Microsoft Office Web Apps, Zoho Office Suite are such representative cloud backed co-editing services. People from different organizations may choose different service providers, which implies the needs of cross-cloud document collaboration. However, it is hard to make different collaborative editing services interoperate with each other. On the one hand, it is not realistic to let all service providers comply to a unified standard and provide interoperation interfaces. On the other hand, the challenge of consistency maintenance in real-time document synchronization still exists. These make interoperability one of the most significant collaboration barriers in online collaborative editing. This paper presents a transparent approach called Hermes to interoperating between heterogeneous collaborative editing services in the cloud. Users are allowed to use their familiar services to participate in the cross-cloud document collaboration. Service providers do not need to put any effort to interoperate with other services. Our approach is validated by a system prototype.**

Keywords*—Collaborative editing, cross-cloud collaboration, consistency maintenance.

I. Introduction

In the trend of economic globalization, we have witnessed an explosion of cloud services for online collaboration [1]. Cloud services are accessible anytime anywhere on any device as long as there is internet connection. Cloud services allow geographically dispersed teams to work simultaneously on a large range of online documents, supporting real-time collaboration and seamless integration with other in-cloud services like cloud storage and data sharing. More and more enterprises and organizations are turning to services in the cloud, a.k.a. SaaS (software as a service) to leverage its high availability, elasticity, and lower infrastructure costs [2]. Google Docs, Microsoft Office Web Apps and Zoho Office Suite are representative cloud collaborative services for document editing.

Due to the pay-as-you-go model of SaaS, enterprises have the freedom to choose their preferable collaboration services (considering the service functionalities, price, and reliability, etc.) to react the changing business needs. On the one hand, users are often unwilling to abandon the user interface they are familiar with, to pay for and learn a new user interface for an ad-hoc collaboration. On the other hand, cloud service users would not like to be bound to a single service provider [3]. Interoperable and portable cloud services are very desirable since they allow users to work collaboratively using their familiar tools. Moreover, lack of a universal set of standards or interfaces leads to a significant risk of vendor lock-in [2].

Standardization appears to be a good solution to address this interoperability issue. However, it is very difficult to make progress on the standardization process if the leading vendors can not reach a consensus [4]. In current stage, cloud service brokerage is a promising way to interoperate between heterogeneous services by protocol conversion. Technically, unlike storage service interoperation or integration, e.g. Jolidriver [5], which can be easily realized by wrapping service APIs to provide a unified user interface, it is much more difficult to make real-time interactive applications like collaborative editing service interoperate with each other.

Firstly, existing collaboration services only provide APIs for document storage and sharing, but no APIs are exposed for manipulating the shared document directly. Secondly, concurrent updates on the shared documents in different cloud inevitably lead to inconsistency. Consistency maintenance is one of the fundamental challenges in collaborative editing services, and it is still there when a brokerage tries to make them interoperate for cross-cloud collaboration. Thirdly, the brokerage service cannot change user experience on original services, e.g. requiring users to install new software.

It is a meaningful yet challenging work to bridge heterogeneous cloud services. In this paper, we propose a transparent approach called Hermes that can enable the interoperability between different cloud collaborative editing services. The brokerage service enables the document synchronization across heterogeneous collaborative editing services with strong data consistency guarantees. It is transparent for both service providers as well as end users. On the one hand, service providers do not need to modify their source code or provide additional APIs to interoperate with other services. On the other hand, users do not need to pay for new services and learn new user interfaces. They can still use their familiar services to participate in the cross-cloud document collaboration. Users are not required to install any additional tools like browser plugins. They can use the brokerage service just as an ordinary cloud service on any device.

The rest of the paper is organized as follows. Section II presents the main idea of our approach and the technical issues that need to be addressed. The Hermes architecture is described in Section III. Section IV elaborates how to transparently adapt heterogeneous collaborative editing services so that they can talk to each other. Section V elaborates a generic collaboration engine for document consistency maintenance. Hermes is evaluated in Section VI. We discuss the related work in Section VII and conclude in Section VIII.

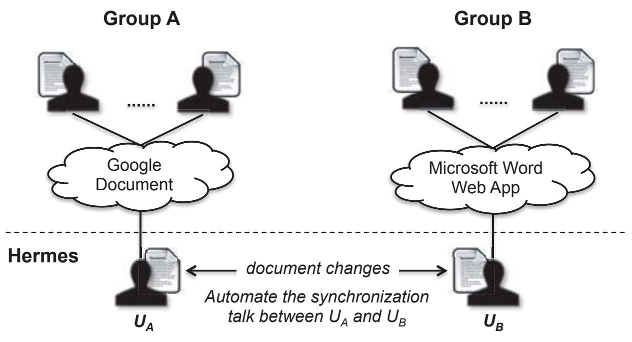

Fig. 1. Illustration of the main idea of Hermes.

II. Main Ideas and Challenges

A. Main ideas

Cloud collaborative editing services allow geographically distributed users to edit a document at the same time. A user's local document can be automatically synchronized with others. Consider two groups of users G_A and G_B as shown in Figure 1. They are independently working on the same document but using two different collaborative editing services. For any user in a group, e.g. $U_A \in G_A$, he can see any change on the document from other users in his own group. Therefore, for two users $U_A \in G_A$ and $U_B \in G_B$, if one can tell the other what changes have been made by his group and apply the remote changes on his own document, the two groups' documents can be synchronized through the "synchronization talk" between U_A and U_B.

Based on this basic idea, we employ robot users in Hermes to automate the "synchronization talk" between heterogeneous cloud collaborative editing services. From a user's perspective, one user only needs to share the document with a robot user in the brokerage service to participate in a cross-cloud collaboration. The robot user is just a member in the same group. Collaborators can still use their familiar collaborative editing services and do not need to worry about the document synchronization across different services. From a service provider's perspective, robot users have no difference with other human users. They do not need to modify their source code or provide additional APIs for interoperating with other similar services.

In Hermes, a robot user is designed such that it works like a human user. Specifically, it uses a web browser to participate in a document collaboration on a collaborative editing service. A document synchronization from a robot user U_A to U_B can be accomplished through following steps:

Step 1: U_A updates the local document by *synchronizing* with the server of the corresponding collaborative editing service, e.g. through saving the current document in a Microsoft Word Web App.

Step 2: U_A *reads* the document rendered in the user interface.

Step 3: U_A compares the current document with the previous one observed.

Step 4: If any change is detected, U_A sends these changes to U_B in the form of editing operations; otherwise, it finishes this document synchronization.

Step 5: U_B *applies* the received document changes on its document.

Step 6: U_B *synchronizes* the updated document with the server to upload the applied changes.

B. Technical Issues

1) Automatic interactions with the collaborative editing service: There are three kinds of interactions between a robot user and its corresponding collaborative editing service, namely *reading* the document state, *applying* document changes received from other robot users, and *synchronizing* the local document with the server. However, most of the existing cloud collaborative editing services do not provide necessary APIs to programmatically realizing these interactions, especially for document editing APIs which are required for a robot user to apply received document changes. Therefore, the first issue that has to be addressed is how to automate the interactions with a collaborative editing service.

2) Consistency maintenance in document synchronization: Another issue in a synchronization talk is the document inconsistency caused by concurrent operations. In a "synchronization talk" between two robot users, since the documents are concurrently updated by individual groups, these changes cannot be applied in another robot user's document directly. For example, the original documents of G_A and G_B contain the same string "abc". Before a "synchronization talk", robot users U_A and U_B see their documents changed to "$axbc$" and "ac", respectively. If they exchange the document changes, i.e. $O_1 = Insert[1, \text{“}x\text{”}]$ and $O_2 = Delete[1, \text{“}b\text{”}]$, and apply the remote changes to their own document directly. U_B's document becomes "axc", but U_A's document becomes "abc" where character "x" is incorrectly deleted. This leads to document inconsistency. Therefore, algorithms of concurrency control and data consistency maintenance have to be implemented for robot users to apply remote changes correctly.

III. Design of Hermes

Figure 2 shows the architecture of Hermes. Each robot user consists of three core components: *CES User Interface* , *CES Adaptor*, and *Collaboration Engine*.

CES User Interface (CUI) is used by a robot user to participate in a document collaboration on the corresponding cloud collaborative editing service. It is the same with the user interface used by human users, i.e. a web user interface opened in a web browser.

CES Adaptor (CA) realizes the automatic interactions with the collaborative editing service, i.e. *reading* the document state, *applying* document changes received from other robot

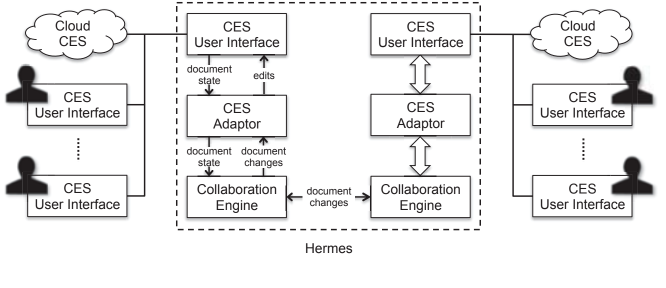

Fig. 2. Hermes architecture.

users, and *synchronizing* the local document with the server. Intuitively, it can be considered as a robot user's "eyes" and "hands". CES Adaptor's implementation is specific to cloud collaborative editing services, however it exposes a common interface specified as follows:

```
interface CESAdaptor {
  ReadDocument() : Document
  Apply(Operation[ ] operations) : void
  Sync() : void
}
```

ReadDocument returns a *Document* object which is currently displayed in the *CUI*. A *Document* object corresponds to one or several HTML code fragments in the web user interface. Robot users have to communicate with each other based on a common document model to synchronize the document states, which hides the heterogeneity of different services.

Apply is invoked to apply remote operations on the local document. These operations are received from other robot users and defined on the common document model. The invocation of *Apply* must ensure the *operations* are defined on the current document displayed in the robot user's *CUI*.

Sync is invoked to synchronize the local document with the server of the corresponding collaborative editing service. Specifically, after a *Sync* invocation, human collaborators who are using the same service will be aware of the document changes made by other groups, and the local document will get updated if the document is modified by human collaborators.

Collaboration Engine (CE) is responsible for detecting document changes and synchronizing the document with other robot users with strong data consistency guarantees. In a cross-cloud document collaboration, a robot user periodically checks the document state, i.e. synchronizing the document with the server and comparing the current document state with the previous document state. If any changes are detected, it will propagates them to Collaboration Engines of other robot users. When remote changes are received, due to consistency issue, these changes cannot be applied on the local document directly. These changes have to be transformed to the right form before applied.

Collaboration Engine is common to all cloud collaborative editing services. It needs to be implemented only once and can be reused when a new collaborative editing service is integrated by Hermes. Note that only two robots are shown in Figure 2. Actually, the number of robot users depends on the number of groups involved in a cross-cloud collaboration. These robot users are connected with each other via collaboration engines to form a Peer-to-Peer network inside Hermes.

IV. Adapting Cloud Collaborative Editing Services

A. Document Model and Operations

To enable the document collaboration across heterogeneous collaborative editing services, a common document model, including the document operations, should be determined first for robot users to communicate with each other on a common basis. What document model to use depends on the type of the document on which the cross-cloud collaboration is conducted, e.g. Office Open XML [6] for collaborations on format text documents, SVG DOM [7] for collaborations on graphic document.

Although documents on Google Document and Microsoft Word Web App may contain tables, headings, footnotes, floating figures besides formatted characters, we simplify the document as text document that only consists of formatted characters for clear explanation of Hermes's essential ideas. As a result, a text document is modeled as a list of formatted characters, and two primitive operations are defined: $Insert(pos, ch)$ (insert a formatted character ch at position pos) and $Delete(pos, ch)$ (delete the character ch at position pos). Changing the format of a character can be realized by a combination of a $Delete$ and an $Insert$ operation. Actually, a more complex text document, e.g. a word file, can also be modeled as list of document objects [8].

B. Adapting Cloud Collaborative Editing Services

APIs provided by cloud collaborative editing services can ease the service adaptation for cloud interoperability to some extent. However, almost all existing cloud collaborative editing services do not provide suitable APIs for document editing. Therefore, we resort to a more generic technique – web browser automation.

Selenium [9] is a set of tools for automating web applications. It interacts with the UI elements in a web page using JavaScript injection, and provides client APIs for Java, C#, Ruby, and Python. Selenium was originally proposed for testing web applications automatically. Now, it supports almost all mainstream web browsers, and some mainstream browser vendors have taken (or are taking) steps to make Selenium a native part of their browsers.

In this section, we elaborate how to adapt Google Document and Microsoft Word Web App by using the technique of web browser automation.

1) Read the document state: Reading the document state is actually a reverse engineering that converts a document from the view representation to the model representation. In a web user interface, a document corresponds to one or several code fragments in the HTML view.

Figure 3 (a) and (b) show the hierarchical structures of web elements for rendering a document in Microsoft Word Web App and Google Document, respectively. Selenium supports finding web elements by their identifiers, tags, CSS selectors etc., and retrieving elements' attributes and inner text. Therefore, the content of a document can be extracted by traversing the hierarchical structure of these related web elements.

2) Apply remote document changes: Like a human user's interactions with the service's UI using a mouse and keyboard, document changes can be applied automatically by sending a series of mouse and keyboard events to the

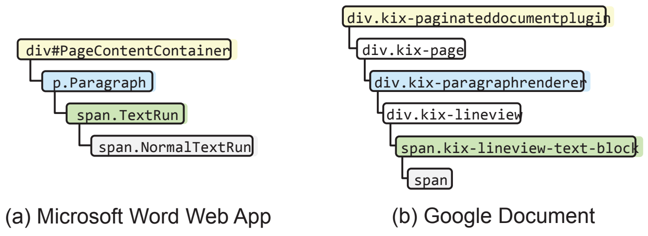

Fig. 3. Hierarchical structures of web elements for rendering a document in Microsoft Word Web App (a) and Google Document (b).

web UI through Selenium (e.g. `WebElement.click` and `WebElement.sendKeys`). Algorithm 1 specifies the procedure of inserting and deleting formatted characters.

Algorithm 1 Apply ($operations$)

```
1: for all $o \in operations$ do
2:     SetCaret(o.pos)
3:     if $o$.type = Insert then
4:         SetFont(o.ch.font)
5:         driver.switchTo().activeElement().sendKeys(o.ch)
6:     else if o.type = Delete then
7:         driver.switchTo().activeElement().sendKeys(
           Keys.DELETE)
8:     end if
9: end for
```

Before applying an operation, the caret needs to be moved to the editing position specified by the parameter pos of the operation (line 2). A straightforward way is moving the caret to the beginning of the document, and then performing pos times of rightward-moving (`sendKeys(Keys.ARROW_RIGHT)`) operations. However, it is not efficient for a long document.

After the caret is moved to the right editing position, a $Delete$ operation can be directly executed by sending a "DELETE" keyboard event (line 7). To insert a formatted character, if the format of the character is the same with the current font setting, sending a keyboard event is enough (line 5); otherwise, the text font needs to be set at first (line 4), which can be realized by interacting with those UI controls for font settings. For example, enabling or disabling the bold font setting can be realized by "clicking" the web element `div#boldButton` and `a#fsbcBold-Small` in Google Document and Microsoft Word Web App, respectively.

3) Synchronize with the collaborative editing service server: The method of *Sync* in the CES Adaptor is provided for Collaboration Engine to programmatically control the document synchronization within a group. Actually, it extends a collaborative editing service to support *asynchronous* document synchronization.

Microsoft Word Web App synchronizes the document asynchronously by default. The document synchronization can only be triggered by explicitly saving the document through clicking the "Save document" button or pressing "Ctrl+S". So, the asynchronous document synchronize can be straightforwardly realized by sending a mouse click event to the element of "Save document" button (`a#qatSave-Small`).

However, documents in Google Document are synchronized in real time, i.e. Google Document saves the document automatically and collaborators can see the document changes stroke by stroke. Fortunately, offline editing is supported by Google Document. Offline operations will be saved locally and synchronized with the server when the network becomes available. So, the asynchronous document synchronization can be realized by programmatically switching the working mode between online and offline.

V. Generic Collaboration Engine

Since documents on different cloud collaborative editing services are concurrently modified by human users in individual groups, directly applying remote changes received from a remote robot user may lead to inconsistent document state. To address the inconsistency issue caused by concurrent operations, a lot of consistency maintenance and concurrency control algorithms have been developed for collaborative editing over past two decades. Operation Transformation (OT) [10], [11] is the most famous one. Its basic idea is to transform the parameters of an operation according to the effects of previously executed concurrent operations so that the transformed operation can achieve the correct effect and make the document consistent. For the example shown in Section II-B2, remote operation O_2 must be transformed against O_1 to become $O_2' = Delete[2, \text{“}b\text{”}]$ before applied on U_A's document. Similarly, O_1 must also be transformed against O_2 before applied on U_B's document, although the resulting form is the same with the original form.

Hermes uses OT to resolve conflicts and maintain data consistency for cross-cloud document synchronization. Robot users communicate with each other via their Collaboration Engines, and form a Peer-to-Peer collaborative editing network in Hermes. For each robot user, its local document corresponds the one in the cloud collaborative editing service this robot user is connected with; human users' operations performed on the document can be considered as the local operations performed by the robot user.

The implementation of the Collaboration Engine is generic to all cloud collaborative editing services. Two core methods *DetectLocalChanges* and *ExecuteRemoteChanges* are implemented in the Collaboration Engine for detecting and propagating local document changes and applying remote changes received from other robot users. After all of human users' operations are propagated through robot users and applied on remote documents, document states will be consistent with each other across all cloud collaborative editing services.

A. Detect and broadcast local document changes

In a typical OT collaborative editing system, a user operation will be propagated to remote sites after executed on local site; meanwhile, the operation will be recorded for transforming later received remote operations. However, a robot user cannot directly get the history operations performed by the human users. Therefore, it uses the algorithm of Myers [12] to derive the incremental operations on the document by diffing the current document state with the one observed previously.

Algorithm 2 (*DetectLocalChanges*) specifies the procedure of detecting and propagating local document changes. Before detecting the document changes, the method *Sync* in the CES Adaptor is called to synchronize with the server of the corresponding collaborative editing service to update the local document (line 1–2). Local document changes are derived by diffing the last document state maintained by the Collaboration Engine with the current document state (line 3). If any changes are detected, the incremental operations are propagated to all other robot users (line 5), and these operations are recorded in the history buffer hb used for transforming later remote document changes (line 7).

When the document changes are propagated to remote robot users, the robot user's identifier rid as well as the state vector sv of the document on which the document changes are generated are propagated (line 5). The state vector sv (line 6) is a vector of N logic clocks, one clock per robot user. For each robot R_{rid}, $sv[rid]$ denotes the number of R_{rid}'s operations that have been executed on the document.

Algorithm 2 DetectLocalChanges()

```
1: $adaptor$.Sync()
2: $newdoc \leftarrow adaptor$.ReadDocument()
3: $opts \leftarrow$ Diff($doc$, $newdoc$)
4: if $opts \neq [\ ]$ then
5:     Broadcast($rid$, $sv$, $opts$)
6:     $sv[rid] \leftarrow sv[rid] + opts$.length
7:     $hb$.append($opts$)
8:     $doc \leftarrow newdoc$
9: end if
```

State vectors can be used to determine the relation of two operations. For any two operations O_1 and O_2, we say (1) O_1 is *happen-before* O_2 ($O_1 \rightarrow O_2$), if and only if $\forall_i (sv_1[i] \leq sv_2[i]) \land \exists_i (sv_1[i] < sv_2[i])$, where sv_1 and sv_2 are the state vectors of the documents on which O_1 and O_2 are generated, respectively; (2) O_1 is *concurrent-with* O_2 ($O_1 \| O_2$), if and only if $O_1 \not\rightarrow O_2 \land O_2 \not\rightarrow O_1$ [11].

Algorithm 3 ExecuteRemoteChanges (rid, $opts$, rsv)

```
1: if operations in $opts$ are not casually ready then
2:     Add $\langle rid, rsv, opts \rangle$ into $q$ to delay the execution of $opts$
3: else
4:     $opts' \leftarrow$Transform($rid, rsv, opts, hb$)
5:     $adaptor$.Apply($opts'$)
6:     $sv[rid] \leftarrow sv[rid] + opts$.length
7:     $hb$.append($opts$)
8:     Check remote operations in $q$ and apply those are casually ready now
9:     $doc \leftarrow adaptor$.ReadDocument()
10:    $adaptor$.Sync()
11: end if
```

B. Execute remote document changes

Given the identifier of the remote robot user rid, the received operations $opts$, and the remote document state rsv on which the operations are generated, Algorithm 3 (*ExecuteRemoteChanges*) specifies the procedure of applying the remote document changes on the local document. A remote operation can only be executed when it is *casually-ready*; otherwise, the operation's execution will be delayed (line 2) until it is *casually-ready*. An operation O is *casually-ready* means all operations that *happen-before* O have been executed on this site. Before applied on the document, remote operations $opts$ have to be transformed against concurrent operations recorded in the history buffer hb (line 4). More details of operation transformation can be found in [10], [11].

The method *Apply* in the CES Adaptor is called to apply the transformed operations on the document rendered in the user interface (line 5). Note that opt's execution may get some delayed operations *casually-ready*. So, the queue q for storing delayed operations is checked until all *casually-ready* operations are applied on the document (line 8). Finally, the document object maintained by the Collaboration Engine is updated (line 9), and the method *Sync* in the CES Adaptor is called so that the collaborator using the same collaborative editing service can be aware of the new document changes (line 10).

VI. Evaluation

Hermes is a cloud brokerage service that enables the real-time document synchronization across heterogeneous collaborative editing services. The *awareness latency* of document changes in a cross-cloud collaboration is crucial to user experience, which means how long it takes for Hermes to get a document change become perceivable for collaborators in other cloud collaborative editing services. In this section, we evaluate Hermes by measuring the awareness latency of document changes and identifying the most important factors that determine the awareness latency.

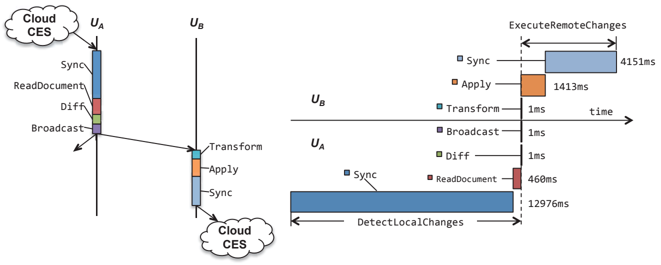

Fig. 4. (a) sub-procedures in a document synchronization. (b) sub-procedures' execution times (ms) in an example document synchronization in the experiment.

A. Experiment setup

A proof-of-concept prototype of Hermes was implemented in Java. Two robot users, namely U_A and U_B, were set up on a server configured with two Inter Xeon 2.80 GHz CPUs and 4G memory. Google Chrome V27.0.1453.94m was used by U_A and U_B to access Google Document and Microsoft Word Web App, respectively. To measure the awareness latency of document changes, for example from Google Document to Microsoft Word Web App, a test client was set up to author a document on Google Document, and the awareness latency was recorded when each document change was successfully synchronized to the Microsoft Word Web App.

The experiment was repeated twice to measure the awareness latencies from Google Document to Microsoft Word Web App ($G \rightarrow M$) and from Microsoft Word Web App to Google Document ($M \rightarrow G$), respectively. To ensure the experiment can be repeated under the same setting, the test client was implemented by using Selenium to automate the document authoring. Before the experiment, an author's process of transcribing a Wikipedia document entitled Hermes was recorded. The generated edit script was used by the test client to author the document by simulating the author's editing behaviors, including the editing operations and the typing speed.

B. Results

As illustrated in Figure 4 (a), The awareness latency of a document change depends on the execution times of two procedures implemented in the Collaboration Engine: *DetectLocalChanges* (Algorithm 2) and *ExecuteRemoteChanges* (Algorithm 3). *DetectLocalChanges* consists of four sub-procedures: *Sync*, *ReadDocument*, *Diff* and *Broadcast*; while *ExecuteRemoteChanges* consists of three sub-procedures: *Transform*, *Apply* and *Sync*. Among them, *Sync* is invoked by *DetectLocalChanges* to update the local document, while *Sync* is invoked by *ExecuteRemoteChanges* to upload the remote changes (received from other robot users) to the server so that human collaborators using the same service can be aware of these changes.

Taking a document synchronization ($G \rightarrow M$) in the experiment for example, Figure 4 (b) shows each sub-procedure's

TABLE I. AWARENESS LATENCY (MS).

Sub-procedures		G→M	M→G
DetectLocalChanges	**Sync**	11536.8	4701.4
	ReadDocument	346	1469.8
	Diff	0.9	0.4
	Broadcast	0.5	0.4
ExecuteRemoteChanges	**Transform**	1.1	0.8
	Apply	2109.2	1563.7
	Sync	4785.6	11971.9
Total		18780.1	19708.5

execution time in details. This example document synchronization took about 19 seconds, i.e. the awareness latency. Two invocations of CES Adaptor's *Sync* method account for the majority of awareness latency (90%, 17.1 seconds in total). *ReadDocument* and *Apply*, another two methods in the CES Adaptor, took 460ms (2.4%) and 1413ms (7.4%), respectively. The execution times of *Diff*, *Broadcast* and *Transform* are negligible compared to that of CES Adaptor.

There were 27 ($G \to M$) and 16 ($M \to G$) document synchronizations happened in the two experiments, respectively. And each document synchronization consisted of 43 and 72 operations on average. Table I shows the results of the two awareness latency experiments. The average awareness latencies of $G \to M$ and $M \to G$ are 18.8 seconds and 19.7 seconds. The execution times of Google Document adaptor's *Sync* and Microsoft Word Web App adaptor's *Sync* are about 12 seconds and 5 seconds, respectively. They account for the majority of awareness latency (85% around).

From the experimental results, we observe that the awareness latency in cross-cloud collaboration mainly depends on the QoS of collaborative editing services, especially for the asynchronous document synchronization. The intermediate brokerage, i.e. Hermes, imposes only a short delay of few seconds (2.5 seconds for $G \to M$ and 3 seconds for $M \to G$ in the experiment).

VII. RELATED WORK

A lot of work has been done on transparently sharing desktop editors or making heterogeneous editors interoperate, e.g. Intelligent Collaboration Transparency (ICT) [13] and Transparent Adaptation (TA) approach[8]. They are able to convert existing applications into multi-user collaborative ones without any modification to the original applications. ICT even allows the shared editors to be heterogeneous.

However, both ICT and TA must be implemented such that local editing events are captured and replayed at remote sites. This mechanism works well for desktop application sharing where both data and applications are replicated on all sites. However, it does not work for services hosted in cloud where applications are deployed in the cloud and users access the data and services using a thin client on any device, e.g. a browser on a mobile phone. Users are often reluctant to install any software or even a browser plug-in when using a cloud service.

Li and Lu [14] proposed a lightweight approach to transparent sharing of heterogeneous single-user editors. It eliminates the needs of capturing and translating editing events; instead it only assumes two simple interfaces that capture and reset the editor state. However, during the document synchronization, users editing operations are blocked. It is quite annoying to the deeply involved collaborators when they are working on a highly interactive tasks where frequent document synchronizations are required.

VIII. CONCLUSIONS

In this paper, we propose a transparent approach to making heterogeneous editing services interoperate in the cloud. The brokerage service Hermes enables real-time cross-cloud document synchronization with strong data consistency guarantees. Users can participate in a cross-cloud document collaboration using their familiar services, and service providers do not need to put any effort to interoperate with similar services. The proof-of-concept prototype validates the feasibility of our approach. The evaluation results show that the brokerage imposes only a short delay on the cross-cloud document synchronization, while the awareness latency mainly depends on the QoS of involved services.

ACKNOWLEDGMENT

The work was supported by the National Natural Science Foundation of China (NSFC) under Grants No.61272533, No.61332008, No.61300201, and Shanghai Science & Technology Committee Project under Grant No.11JC1400800 and No.13ZR1401900.

REFERENCES

[1] "The Forrester Wave: Cloud Strategies Of Online Collaboration Software Vendors, Q3 2012," www.ibm.com/cloud-computing/files/The%5Forrester%5Wave%5Cloud.pdf.

[2] G. A. Lewis, "The role of standards in cloud-computing interoperability (cmu/sei-2012-tn-012)." 2012, http://www.sei.cmu.edu/library/abstracts/reports/12tn012.cfm.

[3] Z. Zhang, C. Wu, and D. W. Cheung, "A survey on cloud interoperability: taxonomies, standards, and practice," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 4, pp. 13–22, Apr. 2013. [Online]. Available: http://doi.acm.org/10.1145/2479942.2479945

[4] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, 2010, pp. 27–33.

[5] "Jolidrive," http://www.jolicloud.com.

[6] "Office Open XML," http://www.ecma-international.org/publications/standards/Ecma-376.htm.

[7] "SVG DOM," http://www.w3.org/TR/SVG/svgdom.html.

[8] S. Xia, D. Sun, C. Sun, D. Chen, and H. Shen, "Leveraging single-user applications for multi-user collaboration: the coword approach," in *Proceedings of the 2004 ACM conference on Computer supported cooperative work*. ACM, 2004, pp. 162–171.

[9] "Selenium," http://docs.seleniumhq.org/.

[10] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1989, pp. 399–407.

[11] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems," *ACM Transactions on Computer-Human Interaction*, vol. 5, no. 1, pp. 63–108, Mar. 1998.

[12] E. W. Myers, "An o(nd) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, 1986.

[13] D. Li and R. Li, "Transparent sharing and interoperation of heterogeneous single-user applications," in *Proceedings of the 2002 ACM conference on Computer supported cooperative work*. ACM, 2002, pp. 246–255.

[14] D. Li and J. Lu, "A lightweight approach to transparent sharing of familiar single-user editors," in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. ACM, 2006, pp. 139–148.