Efficient Coordination of Web Services in Large-scale Multimedia Systems

Mohammed Shatnawi
Microsoft, Redmond, WA, and
Simon Fraser University
Burnaby, BC, Canada

Mohamed Hefeeda
Qatar Computing Research Institute
Hamad Bin Khalifa University
Doha, Qatar

ABSTRACT

Interactive multimedia communication services, such as Skype, are complex and composed of software components typically implemented as web services. Efficient coordination of web services is challenging and expensive, due to the statelessness nature of web services, and because web services change over time. The existing protocols implementing web service transactions are inefficient. They waste resources due to their inability to selectively add/remove individual web services in transactions without incurring high overhead that affects the quality of multimedia sessions. We propose a simple and effective optimization to current web service transaction management protocols that allows individual web services to *selectively* participate in distributed transactions they contribute to. We implement the proposed approach in one of the largest multimedia communication services in the world, and find that it enhances the throughput of multimedia service distributed transactions by 36%, reduces failure rate by 35%, improves multimedia quality (Mean Opinion Score (MOS)) of succeeded transactions by 9%, and reduces the overall time required by all transactions by 35%.

CCS Concepts

•Information systems → Information systems applications; Multimedia information systems; Multimedia streaming;

Keywords

Online multimedia communications; distributed transactions; quality of service;

1. INTRODUCTION

Large-scale multimedia communications services such as Skype and Google Hangout, online eLearning like Coursera, and content distribution and streaming like Amazon CloudFront and YouTube are complex and composed of many software components running on different platforms. For example, an interactive communication service like Skype requires many functions, including user authentication, telemetry, media encoder, dejitter, decoder, storage, and renderer. These functions are implemented using various programming languages, run on different data centers, and could be offered by different, internal or external, organizations. One common approach to handle the complexity of large-scale multimedia systems is to design various functions as *web services* that are accessed via platform-independent and standard protocols and interfaces like RESTful APIs.

To serve a request like create a video conferencing session, the system needs to construct and manage *distributed transactions* involving various web services, while ensuring resources are not wasted nor over-committed, consistency is always maintained, and concurrent transactions do not interfere with each other.

Web services are stateless in nature, which makes handling distributed transactions across multiple web services complex and expensive. This is exacerbated by the dynamic nature and continuous updates of web services [12, 5, 2]. For example, new codecs can be added as new web services, which a multimedia communication service needs to consider without breaking the client code calling the multimedia system.

The current approaches like the protocols in [1, 3, 11], and their implementations in the OASIS projects [13] for managing distributed transactions in web services are not efficient, can lead to substantial waste of resources, and result in reduced multimedia session capacity and quality. The inefficiency is mostly due to a limitation in current protocols that prevents the system from selectively adding and removing individual web services in a distributed transaction without incurring high overhead. To accommodate the dynamic and statelessness nature of web services, current protocols may make the system include unnecessary web services in each distributed transaction. Unnecessary web services for a transaction are those that will not contribute to the successful execution of that transaction. Since web services do not implement rollback [13], the system must issue compensating transactions to reclaim the unused resources and maintain consistency. Compensating transactions are difficult to implement and take long time in real scenarios [13], which result in higher client latency, reduced multimedia session quality, and higher failure rates.

We propose a simple, practical, and effective *optimization* to current distributed transaction management protocols used in web services. It allows individual web services to *selectively* participate in distributed transactions that they contribute to their successful completions, while fully supporting the dynamic updates of web services, and not requiring any significant changes in the implementation of the web services or the systems using them.

Our implementation in a large multimedia system shows that the proposed optimization substantially reduces the number of web services that are defensively included in distributed transactions, reduce the number of compensating transactions, improves the effi-

NOSSDAV'16, May 13 2016, Klagenfurt, Austria

DOI: http://dx.doi.org/10.1145/2910642.2910643

ciency of the systems using the web services, enhances the number of multimedia sessions that can be created between users, and enhances the media quality of these media sessions.

The contributions of this paper can be summarized as follows: we present an efficient approach to dynamically select which web services to include in distributed transactions in multimedia services. We implement and evaluate the effectiveness of the proposed approach in one of the multimedia communication services used in Microsoft, that handles more than one million sessions per second at peak time. We run the experiments for two weeks in a test cluster that gets more than 200 million multimedia requests. The results show that the proposed approach, on average, increases the throughput of transactions by 36%, reduces failure rate by 35%, the multimedia quality (Mean Opinion Score (MOS)) of the succeeded sessions by 9%, and reduces the overall time required by all transactions by 35%.

The rest of the paper is organized as follows. Section 2 summarizes the related works in the literature. Section 3 presents the proposed approach, and Section 4 describes its evaluation in the considered multimedia system. Section 5 concludes the paper.

2. RELATED WORK

Several previous works attempt to address the transaction coordination in distributed multimedia services. For example, Ott et al. [7] address multimedia transaction coordination in distributed services and note the lack of transport-level protocols designed to work independently, as well as inability to share information between multimedia flows without coordinating data transport. They propose an open architecture for sharing network state and transaction information between multimedia flows. Li et al. [4] describe the coordination required for en-route multimedia object caching in transcoding processes for tree networks by requiring service transaction coordination between proxies on a single path or at individual nodes.

Poellabauer et al. [8] argue that real-time and multimedia applications require transaction coordination of event for multimedia delivery mechanism, and note that the observed low Quality of Service in multimedia services is due to lack of effective distributed transaction coordination. Rainer and Timmerer [9] study the impact of geographical distribution of multimedia services and distributed peers, and propose a self-organized distributed transaction synchronization method for multimedia content. They propose a new distributed control scheme that negotiates a reference for the playback time-stamp among participating peers.

Lin et al. [6] study the problems of multimedia distributed transaction latency and their impact on multimedia quality, and introduce compression in graphics streaming. Shatnawi et al. [12] study the use of distributed synthetic transactions to monitor and predict failures in multimedia services. They incorporate a model were all web services are considered part of the transactions, with no ability to dynamically define the atomicity of transactions. Riegen et al. [10] note that scenarios using distributed transactions in online web services are generally controlled by the service client; the service client decides which web services to include in a distributed transaction without any collaboration with the participating web services.

We note that web service distributed transaction management concepts are based on models built for distributed database systems [1, 3]. These models are inefficient for web services. The OASIS projects [13] attempt to define standards for context, coordination, and atomicity between disparate web services. Noting the high cost of transactions in web services, our approach optimizes the selection process of which web services to include in a transaction, builds on top of existing protocols, and adds an optimization communication layer that allows the service client and participating web services to determine, *together*, the web services that need to participate in a given transaction.

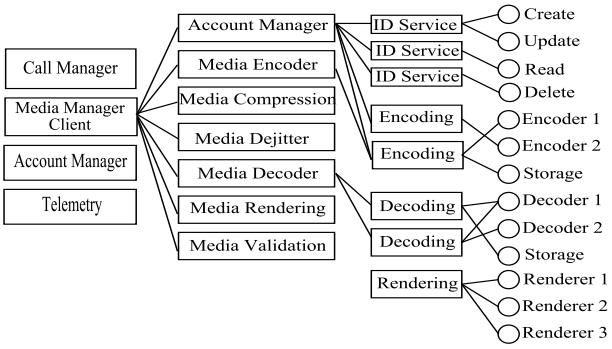

Figure 1: Media Manager client and web services.

3. PROPOSED OPTIMIZATION

We propose the idea of *selective joining and defecting* from distributed web transactions. This improves the efficiency of distributed transaction protocols by allowing them to dynamically include web services in transactions with minimal overhead and no code changes.

3.1 Background

Multimedia services are typically implemented as the collaboration of multiple media web services. The constituent media services may all be part of the same enterprise, or offered through third party online services, such as Amazon Web Services (AWS) [14] for storage. Figure 1 shows a high level diagram of an online multimedia service. Sharing a video between two applications entails getting the user IDs and validating them. The Media Manager Client uses IDAccountManager() that provides create(), read(), update(), delete(), and validate() user accounts. Then the Media Manager Client downloads the video from the source application, performs encoding and size optimization on it, caches the video on the service to enhance the sharing experience and to allow for faster re-attempts in case of delivery interruptions, and finally renders the video for showing at the destination application.

The Media Manager Client uses the following services: MediaEncoding(), MediaDecoding(), MediaStore(), MediaDejitter(), and MediaRender(). The Media Manager Client is the component that controls which web services join transactions. Procedure 1 shows a pseudo code example of how WS Coordination, AtomicTransaction, and BusinessActivity are used in the Media Manager Client to create a transaction context, add web services to it, and conclude the distributed transaction of sharing a video between two applications. To keep the pseudo code simple, we only show how the Media Manager Client creates a transaction context and include web services in it. We show in the following subsections, how the code in Procedure 1 is optimized when we apply the proposed approach.

3.2 Overview

From the example in Section 3.1, there are four steps to implement a transaction. (1) Create a transaction context, and register all required web services in it. (2) Execute all web services in the transaction context. (3) Commit the the transaction. (4) Finalize and close the transaction context. The proposed approach works during the first step; we give the web services, e.g., Create, Update,

Procedure 1 Share a video

```
1: function EncodeAndShareVideo
2:     Create WS-CoordinationContext Context
3:     Create an Activity ShareVideoActivity
4:     Set Coordination Protocol /*e.g. Completion, VolatileTwoPhaseCommit, or DurableTwoPhaseCommit*/
5:     Set ShareVideoWebServices = Encoding, Storage, and Rendering
6:     for each Web Service in ShareVideoWebServices do
7:         Register Service in ShareVideoActivity
8:     Set Compensating Activity for each service in ShareVideoActivity
9:     for each Web Service in Context do
10:        Call Web Service
11:        if Web Service Fails then
12:            Set TransactionFailure = true
13:            for each Web Service that succeeded do
14:                Compensate for Web Service
15:    if TransactionFailure is not true then
16:        for each Web Service in Context do
17:            Commit Web Service
18:            if Web Service Commit Failure then
19:                Set TransactionFailure = true
20:                for each Web Service that succeeded do
21:                    Compensate for Web Service
22:    if TransactionFailure is not true then
23:        Close Transaction Context
```

Read, Delete, Encoder1, Encoder2, Storage1, Decoder1, Decoder2, Storage2, Renderer1, Renderer2, and Renderer3 in the Media Manager example shown in Figure 1, *the choice to join* a transaction. The proposed approach uses the existing web service REST APIs in its communications.

Before we explain the proposed approach, we define the software components that participate in it:

- **Service Client**: the service that applications call to perform the multimedia communication, like the Media Manager Client in Figure 1. It initiates the transaction and controls its lifecycle and closure; either commit or rollback.
- **Master Service**: the service that has the registration information of all participating web services and the data entities they create, update, and/or delete.
- **Web Service**: the participating web service that represents one atomic functionality provided to the Service Client, like encoding a video.

In the proposed approach, the Service Client calls the Master Service to get information about the web services it is going to call. This includes data entities, like user and video, that the web services write, update, or delete. The Service Client uses this information to make an initial assumption to include the web service in the transaction. The Service Client *may allow* some web services to defect from the transaction if the web service does not impact the transaction data entity, or to join the transaction if it does. This eliminates the inclusion of web services in transactions where they are not needed. This results in more media session capacity and better quality, due to less client latency, faster transaction execution, and higher transaction success rate since fewer web services are included and so less failure points. Also, less compensating transactions after roll-back in case of transaction failures.

We propose two extra parameters in the methods of the web services that participate in the proposed protocol: (1) a reference to the type of entities that are impacted by the the transaction; e.g., user, video, and audio. (2) An enumeration value that represents the mode of the transaction. The proposed protocol has three modes that control the web service participation in the transaction:

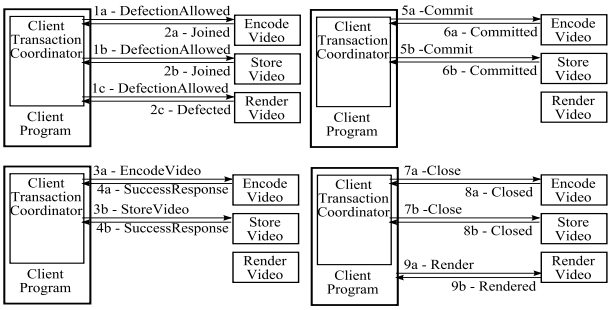

Figure 2: Summary of the proposed approach.

- **All-In**: this is the most common mode in practice in current research and implementation of web service distributed transactions [10, 13]. All participating web services are required to be part of all transactions. If any of them fails, the transaction fails. In the call to the web service, the client passes a reference to the type of entities that will be impacted by the transaction, and a transaction enumeration value of All-In that forces the web service to join the transaction.
- **Defection-Allowed**: the Service Client makes the assumption that the web service is required in the transaction, based on information acquired from the Master Service. The Service Client includes the web service in the transaction in the first step of creating the transaction context, but allows it to defect if it does not impact the data entity at hand, like user. In the call to the web service, the Service Client passes a reference to the type of entities that will be impacted by the transaction, and a transaction enumeration value of Defection-Allowed. If the web service defects from the transaction, it informs the Service Client through its return value.
- **Join-Allowed**: the Service Client makes the assumption that the web service is not required in the transaction, it does not include it in the transaction context in the initial call, but allows the web service to join the transaction if it impacts the entity at hand. In the call to the web service, the client passes a reference to the type of entities that will be impacted by the transaction, and a transaction enumeration value of Join-Allowed. If the web service joins the transaction, it informs the Service Client through its return value.

Figure 2 shows the new sequence of events for the example in Section 3.1 of sharing a video between two applications. Note the impact of our approach on the first and second steps: the Encode(), Store(), and Render() web services are initially included in the transaction context, but are given the chance to defect from the atomic transaction. The Render() web service defects from the transaction, and will be called by the Media Service Client after the Encode() and Store() transaction is successful.

The following sub-sections describe how the Service Client, participating web services, and Master Service implement the proposed protocol.

3.3 Service Client Design

The Service Client maintains a *Transaction Participation Table (TPT)* for the transactions it issues. The TPT has the issued transactions, the participating web services, the entities involved, and the web service transaction inclusion mode. If the Service Client assumes a web service is part of a transaction, but allows

Table 1: Transaction Participation Table (TPT) example.

Transaction	Web Service	Entity	Web Service Inclusion	Inclusion Mode
1	Encode-Video	Video	In	Defection-Allowed
1	Store-Video	Video	In	Defection-Allowed
1	Render-Video	Video	Out	Defection-Allowed

Procedure 2 DTFC Algorithm - Service Client

SERVICE CLIENT ALGORITHMS START A TRANSACTION

```
1: function StartTransaction
2:     Create the Transaction Context
3:     Create Activity, Specify Protocol, and Register Services
4:     Get Web Service Table from Master
5:     Get Entity Table from Master
6:     Include every web service in the transaction that impacts transaction
       entities, as found in the Master Entity Table
7:     Create TPT
8:     Call all Web Services that are included in the transaction
9:     for each Each Web Service Response do
10:        Compare Returned WebService Inclusion Value with Initial
           Assumption in TPT
11:        if Inclusion Value is Different then
12:            Update TPT Inclusion Assumptions for Web Service
13:            Update Transaction Service Registration
14:    Wait for all Web Services to Finish or Timeout
15:    if Any Web Service In Transaction Fails or Timeout then
16:        Roll Back through Compensating Transaction
17:    if All Web Services Succeeded then
18:        Commit Transactions
19:    Conclude Transaction and Close Transaction Context
```

it to defect, and it defects, then the Service Client updates its TPT to indicate the web service is not part of the transaction. After all web services return from the transaction context creation, Step 1, the TPT is updated to reflect the web services inclusion in the transaction. The Service Client then monitors the success and failure of the web services participating in the transaction, and closes the transaction when done, through commit or roll-back, just as it did without the proposed approach. Note that we did not impact, update or change, the used coordination protocol. If it is Completion, VolatileTwoPhaseCommit, or DurableTwoPhaseCommit, it will proceed as it did; only now with just the *right* set of web services that need to be included in the transaction. Table 1, provides an example of an updated TPT that shows the inclusion/exclusion of EncodeVideo(), StoreVideo(), and RenderVideo() in the transaction described in the example in Figure 2.

If the Service Client is in doubt about the need to include a web service in a transaction, due to lack of information at the Master Service, the Service Client includes the web service in the transaction using All-In, or Defection-Allowed modes. The Service Client algorithm is summarized in the Distributed Transaction Federated Control (DTFC) algorithm shown in the "DTFC Algorithm - Service Client" procedure.

3.4 Participating Web Service Design

Each participating web service registers with the Master Service; the registration process is described in the "Master Service Design" section. The web service checks the available entities in the Master Service Entity Table. If it impacts any existing entity, it adds itself to the entity writers. If the web service impacts entities that are not registered with the master, the web service adds these entities to the master Entity Table, and adds itself as an entity writer.

Each web service adds two parameters to its APIs, the first is a reference to the entities impacted by the Service Client call. The other parameter is the transaction control enumeration described above. It is important to note that by requiring participating web services to report the metadata of the entities they impact, we do not change the statelessness nature of the design and implementation of these web services; i.e. there is no execution state maintained. The participating web services implement the Initialization, JoinTransaction, and DefectTransaction functions as shown in the "DTFC Algorithm - Web Service" procedure.

Procedure 3 DTFC Algorithm - Web Service

WEB SERVICE ALGORITHMS INITIALIZE SELF WITH MASTER

```
1: function Initialization
2:     Register Self with Master
3:     Check Available Entities at Master
4:     Add Self to Existing Entity Writers That Web Service Impacts
5:     Add Entities that Web Service Impacts to Master if They Do Not
       Exist
```

JOIN TRANSACTION

```
1: function JoinTransaction
2:     Receive Call from Client to Create/Update/Delete Entity
3:     if Transaction Join Value is not ALLIN then
4:         if Transaction Join Value is JOINALLOWED then
5:             if Web Service Impacts Entities then
6:                 Join Transaction
```

DEFECT TRANSACTION

```
1: function DefectTransaction
2:     Receive Call from Client to Create/Update/Delete Entity
3:     if Transaction Join Value is not ALLIN then
4:         if Transaction Join Value is DEFECTIONALLOWED
       then
5:             if Web Service Does Not Impact ENTITY then
6:                 Defect From Transaction
```

3.5 Master Service Design

The Master Service maintains the following tables:

- **Web Service Table**: has all participating web services names, endpoints, hosting data center, available methods, parameters, authors, readers, writers, and creation date.
- **Entity Table**: has the entity name, ID, description, and web services that update the entity. This table acts as an entity dictionary for the system.

To register, each web service pulls the entity table from the Master Service, updates the table with its entity information, and pushes the updated table back to the Master Service. The Master Service pings the web services regularly to ensure that they are still alive and active. If a web service fails to respond to the Master Service pings after a given threshold, the Master Service removes it from the Web Service Table and from the entity writers in the Entity Table. The Entity Table is an ever-increasing list; there is no need to purge it. Service Clients call the master to get the web service information and the entity lists to allow them to make the right initial assumptions about web services inclusion in their transactions.

4. EVALUATION

We implemented the proposed approach in the multimedia communication service described in the Background section. It is one of the largest services in the world, and deployed in 8 data centers in 3 continents with more than one million transactions per second at peak. We implement the proposed approach on 6 web services in one data center. Service one, Account Management, manages user information and implements Create(), Read(), Update(), and Delete() user. Service two, Encode Multimedia, manages multimedia information and implements EncodeMultimedia() and VerifyMultimedia(). Service three, Compress Multimedia, manages multimedia size before transmitting it on the wire. Service four, Dejitter Multimedia, manages the de-jittering of audio and video content in live communications. Service five, Decode Multimedia, decodes multimedia content after receiving it at the other end of the communication channel. Service six, Multimedia Rendering, renders the video frames before delivering them in a consumable format to the receiving application.

We run the experiments on a test cluster of 10 servers that get 1% of the data center traffic. Each server is a quad-core intel Xeon server with 12 GB RAM. We deploy the current approach on 5 servers, and deploy the proposed approach on the remaining 5 servers. We implement the Service Client and the Master Service on a separate server. We route the same traffic to both sets of services for two weeks, a total of 200 million multimedia requests, and measure the metrics described below. The average number of distributed transactions generated within these requests is 27%, or about 53 million distributed transactions. The remaining traffic is comprised of requests that do not require setting a transaction.

The metrics we use to evaluate the proposed approach are:

- **Throughput**: the number of transactions handled per second.
- **Execution Time**: the total time used by the system to finish all transactions.
- **Efficiency**: the total computation savings, measured by how many web services are excluded from transactions, as a result of the proposed approach. Note that the excluded web services may still be required as part of the service functionality, but not as part of transactions. So failures in such web services result in re-running them, but not in the rollback of other web services that are needed in transactions.
- **Failure Rate**: the transaction failure rate reduction due to only including web services that are needed in each transaction.
- **Media Quality**: the quality of media (audio, video, and image) that is shared between clients, using MOS. We use a proprietary *automated* MOS algorithm. Automated MOS algorithms enable quality measurements in test environments, where sessions are generated programmatically between test clients.
- **Overhead**: the extra calls incurred by querying participating web services about their impact on the given transaction.

4.1 Results

Throughput: 53 million transactions ran over the two weeks of the experiment for a total time of 41 hours and 47 minutes using the proposed approach, and 64 hours and 36 minutes using the current approach. The throughput of the proposed approach is 352 transactions/second, and the current approach is 227 transactions/second. The throughput enhancement is 36%.

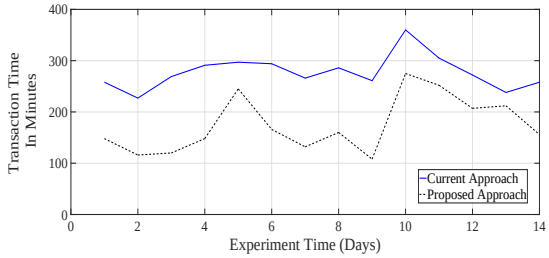

Figure 3: Transaction execution time.

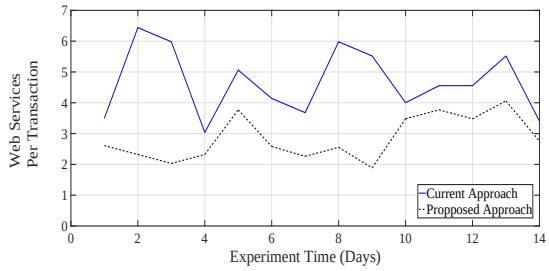

Figure 4: Web services per transaction.

Execution Time: The execution time of the proposed approach (41 hours and 47 minutes) is 65% of the current approach (64 hours and 36 minutes). The reduction in execution time is due to the reduced number of web services required per transaction, which we explain in the efficiency section. If any of the web services that is no longer included in a transaction fails, it will not require rolling back of the web services that were included in any transaction. Figure 3 shows the daily transaction execution time using the current and the proposed approaches. The enhanced execution time of the proposed approach is 36% on average over the time period of the experiment.

Efficiency: The average number of web services included in transactions in the current approach is 4.6, and in the proposed approach is 2.9. The proposed approach is 37% more efficient in using web services in transactions. These web services would have been otherwise included, unnecessarily and incorrectly, in the transactions. Figure 4 shows the daily distribution of web services per transaction, for both approaches.

Failure Rate: From the service logs, the average transaction failure rate due to a web service failure that is not required in the transaction is 17 failures per million transactions. 6 of these, on average, succeeded using the proposed approach since the web service failure did not impact the transaction. The failure rate reduction is 6/17, or 35%, as shown in Figure 5. The remaining 11 failures per million were caused by failures in web services that were required in the transactions. We note that the comparison is not perfect, but as close to fair as we possibly could execute it; we pass the same set of transactions to two identical sets of servers implementing the two approaches. The number of web services that fail on the two systems is very close; the difference is less than 2 failures per million transactions.

The enhancements to Throughput and Execution Time of transactions mean more resources are available to handle more media sessions. The reduction of Failure Rate of transactions means the media sessions that would otherwise have failed and required to be reattempted, are now succeeding.

Media Quality: The succeeded transactions have seen an im-

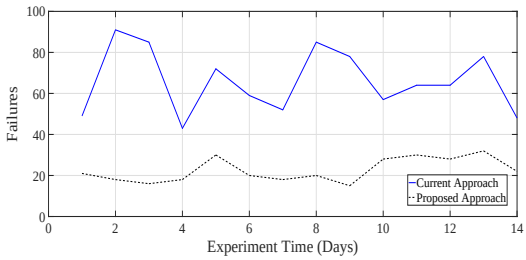

Figure 5: Transaction failure rate.

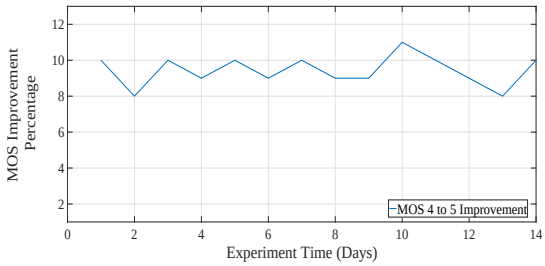

Figure 6: MOS increase from 4 to 5 over 14 days.

provement in their observed media quality (MOS). Figure 6 shows that, on average 9% of the sessions that used to meet SLA with good quality (MOS 4), are now meeting SLA with excellent quality (MOS 5). We attribute this enhancement to less number of web services running as one atomic operation, which means more available resources, less load per transaction, and so higher quality.

Overhead: In Defection-Allowed and Join-Allowed modes, the Service Client makes a call to each participating web service to determine its transaction inclusion. The number of these calls and replies are the Service Client overhead. It is found from the average number of web services per transaction without our approach, which is 4.6 calls and their replies. We find the average overhead cost, from the service logs, to be about 0.2ms per transaction. The highest overhead noted was about 0.8ms. The average transaction execution time reduction due to reducing the number of web services in transactions from 4.6 to 2.9 is about 1.6ms. So the average saving of the proposed approach outweighs the overhead of the Service Client by an order of magnitude.

5. CONCLUSIONS

Current approaches to coordinate web service distributed transactions cause client applications to include all possible web services in distributed transactions.This is especially important in multimedia communication services as any waste, loss, or inefficiency in managing resources result in poor multimedia communication quality, which leads to customer dissatisfaction and loss of business. We presented a novel approach to allow multimedia web services to selectively join or defect from distributed transactions depending on their impact on the transactions. This reduced the number of multimedia services included in each distributed transaction, which led to enhancing the throughput of multimedia distributed transactions by 36%, and resulted in 9% media quality enhancements from MOS 4 to 5.

6. REFERENCES

[1] S. Bhiri, O. Perrin, and C. Godart. Ensuring required failure atomicity of composite web services. In *Proc. of ACM Conference on World Wide Web (WWW'05)*, pages 138–147, Chiba, Japan, May 2005.

[2] B. Gedik and L. Liu. Peercq: A decentralized and self-configuring peer-to-peer information monitoring system. In *Proc. of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, pages 490–499, Providence, Rhode Island, May 2003.

[3] K. Haller, H. Schuldt, and C. Turker. Decentralized coordination of transactional processes in peer to peer environments. In *Proc. of ACM International Conference on Information and Knowledge Management (CIKM'05)*, pages 36–43, Bremen, Germany, May 2005.

[4] K. Li and H. Shen. Coordinated enroute multimedia object caching in transcoding proxies for tree networks. In *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM'05)*, pages 289–314, New York, NY, August 2005.

[5] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y. Wang. Webprophet: Automating performance prediction for web services. In *Proc. of USENIX Symp. on Networked Systems Design and Implementation (NSDI'10)*, pages 143–158, San Jose, CA, April 2010.

[6] L. Lin, X. Liao, G. Tan, H. Jin, X. Yang, W. Zhang, and B. Li. Liverender: A cloud gaming system based on compressed graphics streaming. In *Proc. of ACM Conference on Multimedia (MM'14)*, pages 347 – 356, Orlando, Florida, November 2014.

[7] D. E. Ott and K. Mayer-Patel. An open architecture for transport-level protocol coordination in distributed multimedia applications. In *Proc.of ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM'07)*, New York, NY, August 2007.

[8] C. Poellabauer, K. Schwan, and R. West. Coordinated cpu and event scheduling for distributed multimedia applications. In *Proc. of ACM Conference on Multimedia (MM'01)*, pages 231–240, New York, NY, 2001.

[9] B. Rainer and C. Timmerer. Self-organized inter-destination multimedia synchronization for adaptive media streaming. In *Proc. of ACM Conference on Multimedia (MM'14)*, pages 327–336, Orlando, FL, November 2014.

[10] M. Riegen, M. Husemann, S. Fink, and N. Ritter. Rule-based coordination of distributed web service transactions. In *Proc. of IEEE Transactions on Services Computing (SC'10)*, pages 60–72, 2010.

[11] D. Roman and M. Kifer. Reasoning about the behavior of semantic web services with concurrent transaction logic. In *Proc. of Proceedings of IEEE Conference on Very Large Data Bases (VLDB'07)*, pages 627–638, Vienna, Austria, September 2007.

[12] M. Shatnawi and M. Hefeeda. Real-time failure prediction in online services. In *Proc. of IEEE INFOCOM'15*, Hong Kong, April 2015.

[13] Oasis web services business process execution language (wsbpel) tc and oasis web services coordination (ws-coordination) and oasis web services atomic transaction (ws-atomictransaction). https://www.oasis-open.org/committees and http://docs.oasis-open.org/ws-tx/wscoor/2006/06 and http://docs.oasis-open.org/ws-tx/wsat/2006/06.

[14] Transaction library for dynamodb. http://aws.amazon.com/blogs/aws/dynamodb-transaction-library.