A Real-Time Browser-Based Collaboration System for Synchronized Online Multimedia Sharing

Cristian Gadea, Bogdan Solomon, Bogdan Ionescu, Dan Ionescu
NCCT Lab, University of Ottawa, Ottawa, Canada
{cgadea, bsolomon, bogdan, dan}@ncct.uottawa.ca

Gabriela Prostean
Politehnica University of Timisoara, Romania
gabriela.prostean@mpt.upt.ro

***Abstract*—The amount of multimedia content on the web has been growing at a staggering rate, and users are increasingly looking to share it with colleagues and friends in real-time. Several commercial, open source and academic solutions have attempted to make it easier to share a large variety of online content with others but they are generally limited to sending links. Existing products have not been able to provide a browser-based system that synchronizes disparate web content among many users in real-time. Additionally, they have lacked a platform with a modular architecture that can be easily extended by developers to support new sources of online media. In this paper, we propose a next-generation software architecture for a real-time web-based collaboration platform. The platform allows users to collaborate over webcam chat while viewing videos, photos, maps, documents, listening to music, and playing games, all in real-time. As examples, we show how three versions of a cloud-based environment called Watch Together were deployed to live users, including within Facebook and an e-learning environment. Finally, we present usage data from the deployments and reflect on how users share and consume real-time multimedia content.**

***Index Terms*—real-time web collaboration, distributed systems, cloud-based digital content delivery, online multimedia sharing**

I. Introduction

With the advent of Web 2.0 technologies, new services have appeared where web users are no longer satisfied with just viewing simple HTTP pages but expect the ability to share and collaborate with other people, such as friends or colleagues, online and in real-time. This can be seen with services like Google Docs [1], where multiple users can work on the same document at the same time, and the document is stored on Google's remote servers in the cloud. Other popular websites which offer the ability for online content sharing between users include Facebook and Twitter. However, the services offered by both of these systems are not truly real-time. In both Facebook and Twitter, a message posted by a user containing links to photos or videos is later viewed by one or more users separately. In many ways, this is in no way different than sending an e-mail with either links or attachments.

Currently, if users wish to share some photos with their friends, they can upload them to a cloud storage service like Flickr [2]. They can then use a service like Twitter or send an email to their friends in order to let them know that the photos are available. Their friends, some of which may already be online, may receive the links and choose to view them, which will be done on their own. The users who shared photos with their friends may receive a response but they do not know the true reactions of their friends to the photos as they are not able to view the photos as a group and receive instant feedback. This is true as well for sharing videos, maps and even documents in a corporate setting.

The goal of the system presented in this paper is to achieve real-time collaboration between users who, as a group, share a collaborative session. This implies that all users in the session will see the exact same state of the system - be it the same video at the same moment in time, the same image or the same page in a document or slide. Actions performed by a user - changing the image, fast forwarding in the video, changing the page - are replicated across all the users in order to ensure that the same state is maintained. At the same time, text and video/audio chat are integrated in the system so that users can see and hear each other as they collaborate over the online content.

The system described has several notable constraints that are not present in simpler sharing systems such as the one used by Facebook. First of all, the requirement for the system to be real-time implies that the latency between users has to be very low to prevent discrepancies between the state of the system seen by the different users. At the same time, the existence of video chat implies that each user will consume a considerable amount of bandwidth. Our system will attempt to minimize the amount of bandwidth used by each user, while at the same time ensure that the quality of the video chat does not degrade excessively. For this, the system will provide the user with visual feedback of their measured latency time and bandwidth usage.

Furthermore, the system has to be itself a platform on top of which new synchronized applications can be developed and deployed. Through this capability, developers can extend the functionality of the system by either adding entirely new synchronized application types or extending existing types with new data sources. For example, developers could either add an entirely new collaborative application such as a game of Scrabble, or they can customize the existing image sharing application (which is already used for sources like Flickr and Facebook Photos) to create a collaborative application for a new source such as PhotoBucket. This way, as new online services become available, the platform can be extended to support them and make their content collaborative with relative ease. Since the system has to be a platform, it must provide easy-to-use APIs for developers. Additionally, the

synchronization between users has to be done transparently such that developers do not have to worry about the necessary synchronization messages reaching all users of a session.

Finally, the system must be as accessible as possible for users. As such, it must make use of the latest web-based technologies and standards. In order to achieve this, the system must not require the download and installation of proprietary browser plugins and must run on various types of devices, including emerging smartphones and tablet PCs. This is achieved through the use of the Flex 4 framework for Adobe Flash [3] on the client side and a Real Time Messaging Protocol (RTMP) based [4] server. While Adobe Flash is a browser plugin, it can be found on more than 98.9% of client machines [5]. Currently, this represents better penetration than HTML5, which is still in the course of development and can be found only in experimental states in all major web browsers.

The organization of the remainder of this paper is as follows: Section II introduces work related to the system presented here. Section III then discusses the requirements and architecture of the proposed collaboration system. Section IV offers a closer look at the implementation details of the system and its API. Some results of the system, including screenshots of the system in action and usage data from three separate live deployments, are presented in section V. Finally, section VI reflects on the contributions of this paper and proposes topics for future research.

II. Related Work

Several commercial and academic web-based collaboration solutions have existed for some time [6]. Many collaboration solutions, such as the ones mentioned in [7], rely on screen-sharing techniques where a user decides to share their desktop view with other users. Since this technique is essentially taking multiple screenshots of the user's desktop per second and streaming them to the other users, it requires all users to have a large amount of bandwidth, especially when video chat is also present. In addition, the framerate and video/audio streaming quality is not sufficient for a proper video viewing experience, especially for the larger high-resolution videos commonly found online today. Finally, all solutions require the users to download and install new client or server components on their computers (and ensure that all users have compatible and up-to-date versions) in order to gain access to share their desktop.

One of the first companies to release the ability to view synchronized videos to the public was Yahoo! with a product named Zync [8]. Developed as an add-on to the popular Yahoo! Messenger client, Zync would detect when a user taking part in a two-user IM conversation pasted a link to a video on YouTube or Google Videos, which would cause a synchronized video player window to appear. They found that "the synchronicity and social co-presence would promote online conversation, engagement with shared media", with 31% of users returning to reuse the service after their first session with it. They also found that music, entertainment and comedy clips made up most of the types of videos being shared, and that users generally performed a skipping action at the start of the video while chatting more near the end of the video. Users would need to download and install the Yahoo! Messenger application as well as the Zync addon, and the system was only for Microsoft Windows. In addition, the Zync client had to first download the entire Flash-based video for both users before it was available for co-viewing.

In the system we propose, users are able to collaborate on video data while having the video rendered on each user's local machine within their own instance of that application. Videos are not re-encoded as part of a remote screen update; they therefore run at full speed for all users and synchronization is ensured through event-based signals. Our system is able to make use of the latest web technologies and service-specific APIs to allow for instant streaming and skipping within videos, as well as support for additional online content such as photos, maps, live video streams and documents. The system is also designed to support a nearly-unlimited number of users per session through a cloud-based architecture. Finally, we plan to augment the social viewing experience with browser-based video and audio chat in addition to regular text chat.

III. Requirements and Architecture

This section looks at the overall architecture design considerations. As mentioned in the introduction, the system presented in this paper must achieve real-time collaboration between the different clients in the same session, while at the same time be extendable and accessible to various devices. The requirements for the above architecture are first introduced, and the architecture is then developed.

A. Requirements

The central idea of the system presented in this paper is its real-time browser-based collaborative nature. A number of functional software requirements result directly from this:

1) Clients in the same session must see the exact same thing in the collaborative part of the application.
2) Clients must be able to communicate with each other through text, audio and video.
3) Clients must be able to search through various data sources for content.
4) Clients must be able to see which of their uploaded content is online.
5) Clients must be able to invite their contacts to a session.
6) Clients must be able to accept or reject an invitation received from another user.
7) Clients which join the session after the session has started must be synchronized to the state of the session.

A number of non-functional requirements can also be determined:

1) Clients must be able to access the system via a web browser without the need for downloading extra proprietary addons or plugins.
2) The system must be extendable with new data sources and even new application types.

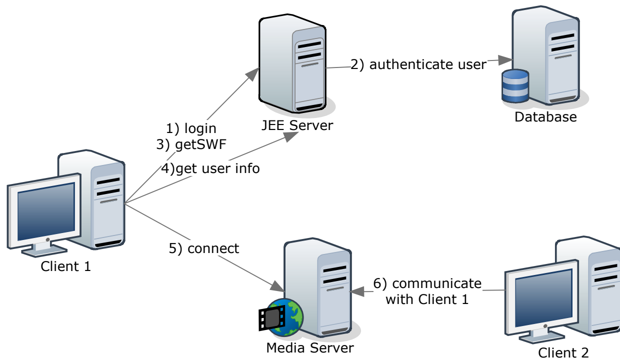

Fig. 1. Server-client Architecture.

3) Data sources and application types are loaded on demand by clients.
4) The system must examine the user's latency and bandwidth in order to provide a balance between video chat quality, latency and bandwidth usage.
5) The system supports the ability for different data sources and application types to be loaded depending on the deployment type.

B. Architecture

In order to achieve the functional and non-functional requirements of the system, the architecture was developed as a client-server application, where the Media Server has three responsibilities: providing the audio/video streaming between clients, passing synchronization messages between clients in the same session, and passing session setup messages between clients. Figure 1 shows the high-level client-server architecture. In this figure, the Java Platform, Enterprise Edition (JEE) server is responsible for authenticating and authorizing the users who wish to use the system. If the system uses an external authenticating/authorization system, such as when it is embedded as a Facebook application, then the JEE server would not be used. However, if no JEE server is used, it would be replaced by a web server whose sole responsibility is to provide the HTML page and the embedded ShockWave Flash (SWF) files necessary for the client to run the application. Once the client is authorized and the SWF files are loaded client side, the client connects via RTMP to the Media Server. After a connection is established, the client can start collaborating with other users.

1) Server Architecture: The server side is structured in three classes - a class which handles client connections and messages from clients, a class which represents a session and is responsible for holding session state and broadcasting session messages to clients, and finally a class which represents the clients themselves. Figure 2 shows the server side structure.

The ServerModule class, which is a singleton in a server, allows clients to join and leave the application, as well as create sessions, invite other users to sessions and allows invitees to accept or reject invitations. The server has zero or more clients connected at any time and has zero or more

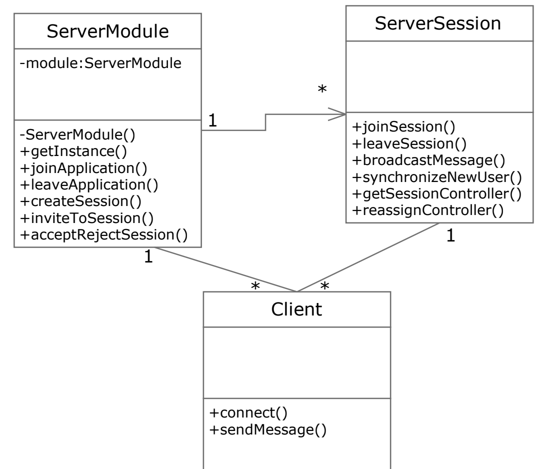

Fig. 2. Server Class Structure.

sessions running at any time. Clients can connect to the server and send messages to other clients. Clients can also join sessions by either creating a new session or accepting an invitation from another client. Clients can also leave sessions - either explicitly or by disconnecting from the server while in a session. When a new client joins a session, the new client is synchronized to the state of the session. Clients can also use a session to broadcast messages to other clients. Finally, every session has a session controller. The controller can allow or deny other users from controlling what is seen in a session. Initially the controller is the user who creates the session. The controller can also reassign the session control either explicitly by selecting another user to pass control to or by leaving the session. If the controller leaves the session, then another user in the session is randomly chosen by the server as a session controller, and all the users in the session are notified.

2) Client Architecture: The client side has to be easily extendable and be capable to load its components on demand, which requires the architecture of the client to be modular. The client modules are split into two categories:

- Domain modules - these modules are domain specific. A domain represents a deployment instance of the system and each domain has a separate login approach. For example, the login approach for the client integrated as a Facebook application is different than when the deployment uses a local JEE server. There are two domain modules which are developed for each domain - a login module, which performs the login logic and retrieves the local user's information and the user's contacts, and a user list module, which displays the user's contacts. The reason for using different user list modules is that different domains can have different contact categorizations. For example, Facebook contacts are called friends; a user has a number of friends and there are no subcategories. If the system is deployed inside an organization however, the organization will have groups for various tasks. As such, the contact display list has to be capable of showing

the contacts inside a group, and to allow the user to select a different group. In order to maintain a consistent Look and Feel, the user list modules extend an existing component.

- Media modules - these modules are for collaborative *applications* and data sources. Each media module is actually separated into four components: a search component, a viewer component, a control component, and an information component.
  - The search component allows the user to search for media to share with other people. For example, for YouTube videos the search component mimics the standard YouTube search options and display a list of thumbnails for the videos. The user can select the video to view from this list. For games, the search component displays the game lobby from which the users can select what game to join.
  - The viewer component displays the media content selected from the search list. For YouTube, this is a video. For documents, this is the selected document. For games, this the actual game screen. The viewer supports a maximized full-screen mode and is responsible for resizing the media content.
  - The user control component displays the various controls that the user has over the media content. Not all actions done via this bar are synchronized. For videos this is a bar with play/pause, volume and video timeline seeking controls. However, the volume is not synchronized across the session, as various users might prefer different volume settings. For documents, this includes zoom levels, as well as controls for switching to the next/previous page and for skipping to a specific page. The zoom level is not synchronized, as various users can be using different screen resolutions.
  - The information component displays information related to the currently selected media content. This is the title of the video, image or document.

  Each of the four media module components implements the Model-View-Controller (MVC) software pattern. The model is shared between all four components as it represents the shared state of the media content.

Figure 3 shows the state chart diagram of the client. Initially, the client's browser loads the SWF file from the server. Once the main application is loaded, it determines the domain under which it is deployed. Based on the domain, it loads a configuration file from the server. This XML configuration file describes the domain modules to load, as well as the media modules which are available to the application. The client code then loads the login module and the user list module, and performs a login for the user. As part of the login process, the user's contacts are loaded. Once the user is logged in and the contact list is loaded, the client opens the connection to the Media Server and the search modules become available. When the user selects a search module from the list of available

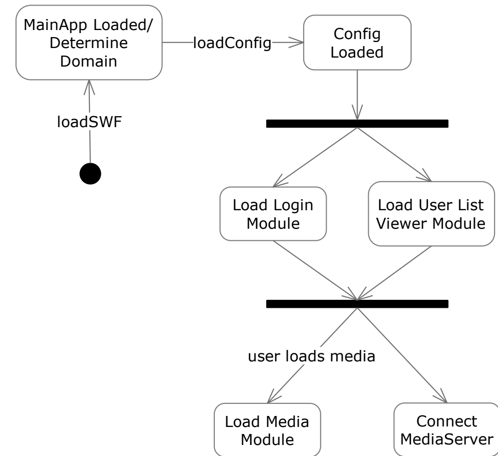

Fig. 3. Client State Chart.

modules, the appropriate search module is loaded from the server. Finally, if the user selects a specific media item to view, the corresponding viewer and media control modules are loaded from the server.

3) Server Client Communication Architecture: A very important part of the architecture deals with the server-client communication. Due to the modular and extendable nature of the client, the communication between the client and the server must be able to support messages that were not considered at design time. Figure 4 shows the sequence diagram for establishing the connection between the client and the server, and then disconnecting. The diagram assumes that there is already a client connected to the server (client2). The sequence is initiated when a different client, client1, connects to the server. Upon connection, the media server performs access control and determines if the user should be allowed to connect. If the user is allowed to connect, a *connect success* message is sent back to the client. The client then sends a *notify online* message, which contains a list of the unique IDs of the client's contacts. Assuming client2 is a contact of client1, the server posts messages to both client1 and client2 that the other client is online. Following a collaborative session, the sequence diagram for which is shown in Figure 5, client1 disconnects from the Media Server. The server notifies client2 that client1 has gone offline.

The session sequence diagrams assumes that the three clients are all contacts of each other. In order to begin a session, a client - client1 in this case - invites another client, client2, to a collaborative session. Upon reception of the request, the server creates a new session, adds client1 as the controller of the session and forwards the invitation to client2. At the same time, it notifies client3 that the other two clients are busy, as well as notifies client1 and client2 that client2 and client1 are busy respectively. The goal of the busy state for clients is to prevent a client from receiving invitations while in a session or while an invitation is pending. After the invitation is received, client2 decides to accept the invitation and the invite reply message is sent to the server. The server adds

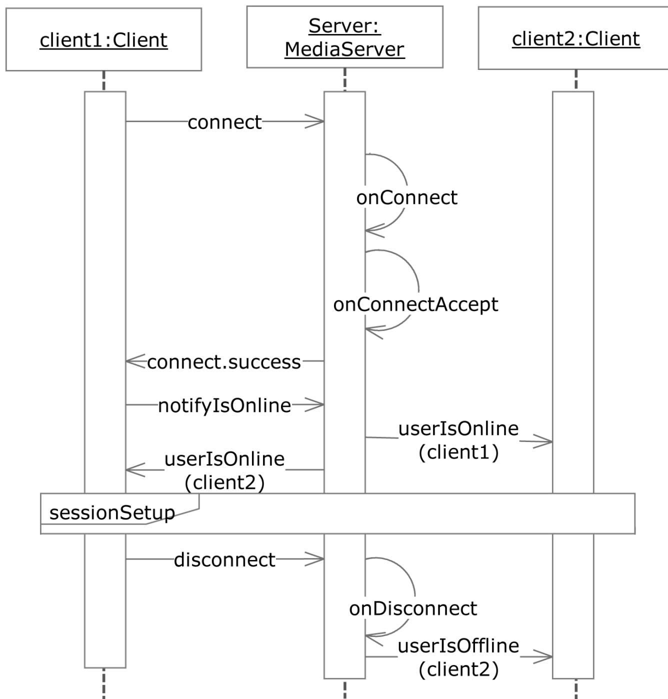

Fig. 4. Server-Client Connect Sequence Diagram.

the new client, client2, to the session. When a new client is added to a collaborative session, the session synchronizes the new client to the state of the session. In order to determine the state of the session, the session asks the controller what the controller's state is. The controller determines its state and sends it back to the server-side session. The server session then sends a *media API* message to the client or clients that are to be synchronized. *Media API* messages are the same messages that are used in order to synchronize new actions done by the users. Also note that if multiple users join a session while the session is waiting for a synchronize state reply from the session controller, the clients are added to a waiting queue and multiple messages are not sent to the controller. Once the reply arrives from the controller, the message is broadcasted to all the clients waiting for it. This is done in order to minimize the messaging between server and clients as much as possible. Finally, when a client decides to leave a session, a message is sent to the server. Upon reception, the client is removed from the session and the other clients receive notifications that the client is online, indicating that the client has become available again.

IV. Implementation

In order to implement the architecture described above, Adobe Flex was used for the client side and Red5 [9] was used as a server. The reason for using Red5 for the server is that it is an open source implementation of the RTMP specification. In order to achieve the desired modularity of the system, Flex Modules are used. By using Flex Modules, we can allow for the downloading at run time of only the required modules. This section focuses on the client-side implementation and APIs provided for the extension of the client as the server implementation is fairly simple.

As mentioned in the architecture section, each media module implements the MVC software pattern. In order to allow for the extendability of the platform, each of the controllers for the modules must implement one of the provided interfaces. This is done in order to ensure that the modules, which are loaded at runtime, can communicate with each other. The model, which is common between all four modules, is defined for each media type by the developer as it is media dependent. The viewer for each of the modules is also media dependent, and as such is defined by the developer. Figure 6 shows the structure of the client code interfaces.

The SearchController performs two actions - *search*, which uses the underlying search mechanism for the data source in order to find media content and is dependent on the data source, and *loadMedia*, which loads selected media content in the viewer. In order to actually load media, a *MediaCommandQueue* is used, which is a singleton. The role of the *MediaCommandQueue* is to store commands until the viewer module is loaded and then play the commands in the order received. All commands, be they *loadMedia* or control commands, pass through the *MediaCommandQueue*. This is necessary due to the fact that modules are loaded on demand. Consider what would happen if a new user joins a session, receives a synchronization command, and while the client side loads the correct viewer, new commands are received from other clients. In such a case, the newer commands would be lost, which would lead to the desynchronization of the sessions. A second role for the *MediaCommandQueue* is that of ensuring that the correct modules are loaded. When a new command is received, the *MediaCommandQueue* determines if the currently loaded modules can perform the command. If they can not, then the correct modules are loaded, the queue is emptied and the command is added to the queue. If, while modules are being loaded, a new command comes which requires different modules than the ones being loaded, then similarly the queue is emptied and the command is added to the queue. The *MediaCommandQueue* is also responsible for broadcasting messages to other members of the session through the connection to the server. The *MediaCommandQueue* uses a data structure to hold the commands which has three fields: *command*, which represents a unique ID for the command, for example loadVideo, play, pause, seek for video media content; *data*, which stores the data for the command, such as for example the new time position for the video seek command; and *description*, which holds the description of the media type used by the information module. The *ViewerController* performs five actions: *initComplete*, which notifies the command queue that the viewer has finished initializing and that the command queue should start playing back commands; *command*, which performs a command coming from outside (either from another user or from the user control module); *setSize*, which resizes the viewer; *synch*, which synchronizes the viewer to the current state of the session (this is what new users execute); and *getSynchState*, which retrieves

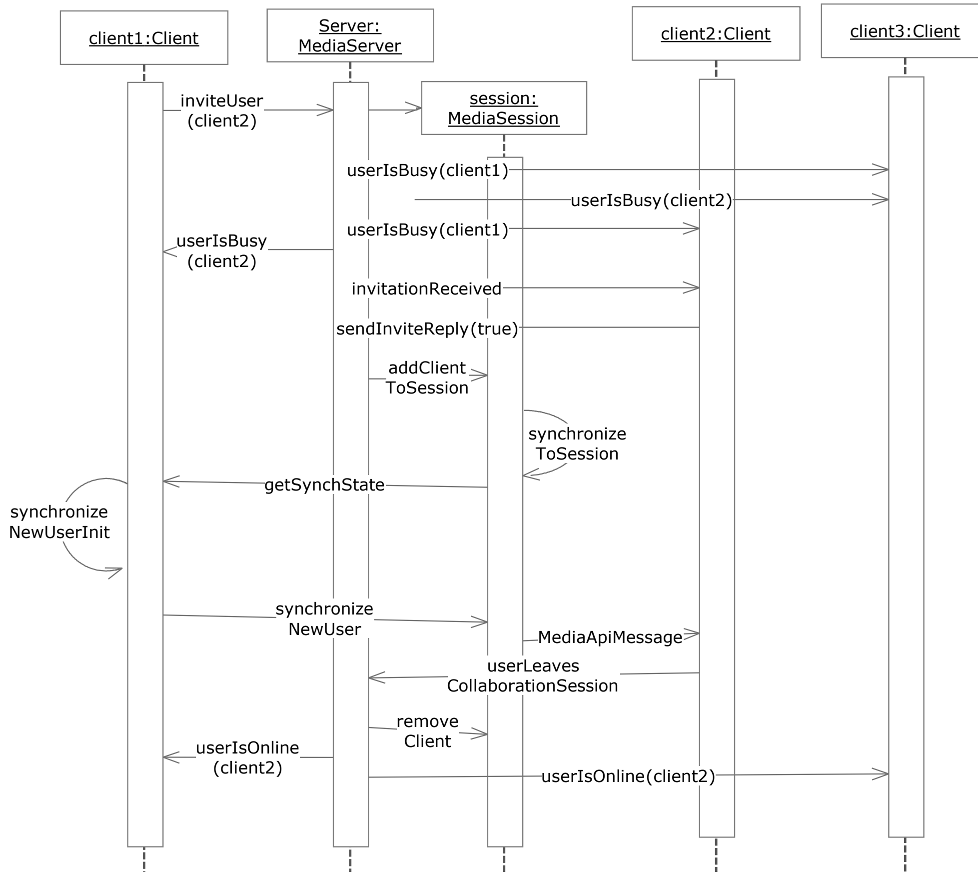

Fig. 5. Server-Client Session Sequence Diagram.

the current state of the session (this is executed by the user who is the session controller). The *UserCommandController* performs two actions: *sendCommand*, which sends a command to the *MediaCommandQueue* to be executed by the viewer, and *maximizeMinimize*, which is sent to the viewer to resize. Finally, the *InformationController* has only one method, *setDescription*, which is called from the *MediaCommandQueue* when new media content is loaded.

Through the use of the above interfaces, developers can easily add new applications and data sources to the system without needing to worry about the synchronization mechanism. The modular approach also allows different media modules to be distributed across different locations, thus behaving like a cloud application. Developers can host their own modules either on their own servers or on clouds like Amazon's Elastic Computing Cloud (EC2), and the system, through a simple configuration file, can find and load them. This can be easily extended to use a registry instead of a configuration file in order to discover and load modules.

Currently, the system supports YouTube recorded videos, Flickr and Facebook images, Twitter text messages, local documents where users can upload documents to the system and share them, UStream live video, and Google Maps. Similar services such as Vimeo videos can be added with relatively

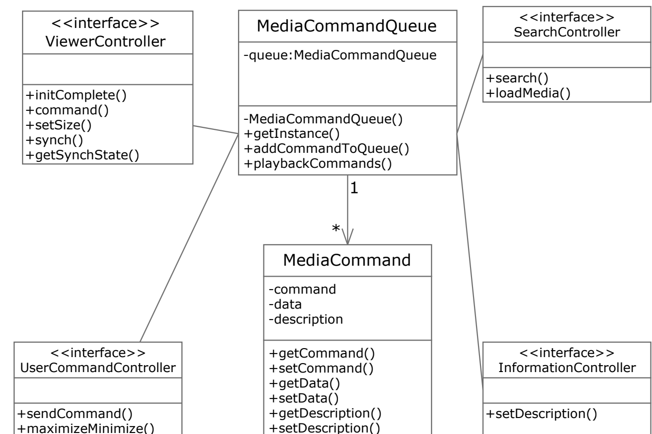

Fig. 6. Client Interfaces.

little effort. The only requirements are for developers to create the integration with the external data source and use the provided interfaces.

V. RESULTS

The implemented system was called *Watch Together* to highlight its collaborative nature. Three variations of Watch

Fig. 7. Flickr Photo Sharing and Video Chat on a Test Deployment.

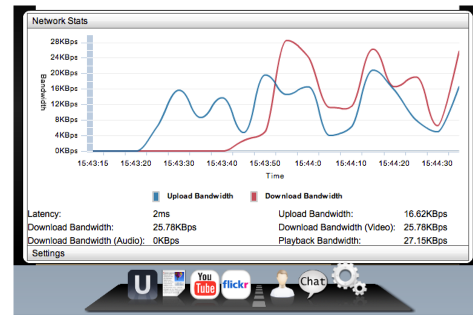

Fig. 8. Monitoring Upload and Download Bandwidth in Watch Together.

Age Group	Percentage of Users (%)
< 18	2
18-21	11
22-25	54
26-29	28
30-33	4
> 33	1

TABLE I
WATCH TOGETHER USERS BY AGE GROUP.

Together were deployed to different groups of users and their usage of the deployments was observed.

A. Internal Test Deployment

Our initial deployment of the system was connected to a test user database with a simple login system developed using Java Server Faces (JSF). An internal webpage presented the user with a username and password login screen, and a successful login would load a page that had the compiled Watch Together SWF embedded within it. Figure 7 shows three users participating in a video chat session on this JEE-based test deployment. The viewer module described in the architecture section appears in the top half of Watch Together and is always synchronized between the users in the session. The users currently in the session are shown along the bottom of the Watch Together interface using either their profile images (retrieved from the user database) or a live video stream from the user's webcam.

Near the middle of the Watch Together interface is a menu bar with clickable icons. The left side of the menu bar contains the list of applications developed for the Watch Together platform (by using the API described in section IV) to support various sources of popular online multimedia content. Clicking one of these icons brings up a search module containing thumbnails of the search results. The thumbnails can be clicked on to change what is displayed within the viewer module. The right side of the menu contains icons for the contacts module (which shows a list of available online users that can be invited to a session), text chat module, and the settings module, all of which appear on top of the viewer module but do no affect the synchronized content. The settings module provides real-time incoming and outgoing video bandwidth details of the video streams, as shown in Figure 8. In addition, the user's latency relative to the RTMP server is displayed beside their name in the bottom left corner.

The initial testing among small groups of friends and colleagues proved to be a success, with users particularly drawn to the YouTube application for sharing their favourite video clips. The system properly kept all content synchronized among the users, and, at the same time, it allowed them to use video chat to discuss the content. A common request from our test users was to add the ability to change the volume of the audio coming from each user's video chat stream. We therefore added a volume slider and mute button over each user's video stream that appeared whenever the user moved their mouse over the video chat area.

B. Facebook Deployment

Once internal testing was completed, the system was deployed to the public as a Facebook Application [10]. Users can access the system by logging in with their Facebook account and adding Watch Together to their application bookmarks list. Facebook's API [11] was used to retrieve the information about the currently logged in user (such as the user's name, their list of friends, their profile image, etc.) and to populate the user interface.

Based on over a thousand users who opted in to share annonymized usage data, 65% of users of Watch Together were found to be male and 35% were female. We were pleased to find that 24% of users enabled their webcam when using Watch Together, which is a strong indicator that users enjoy sharing and discussing online content in this collaborative fashion. The YouTube application was again the most popular, with an average of 11.4 videos viewed per user. The distribution of users by age can be seen in Table I, with most users falling into the 22-25 age group.

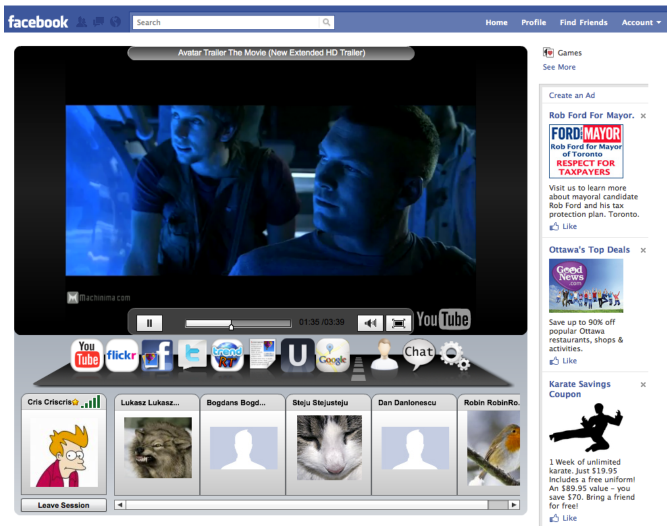

Fig. 9. Six Facebook Users Collaboratively Watching a YouTube Video.

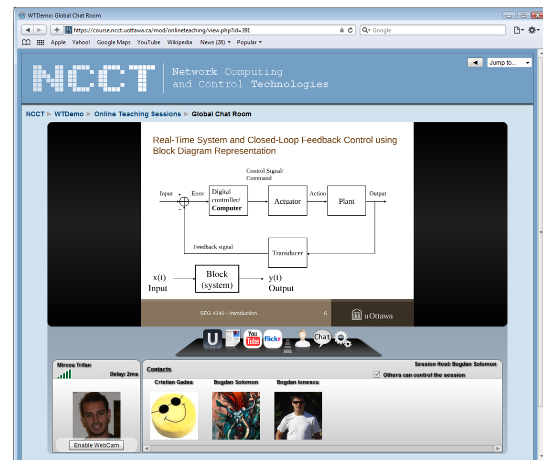

Fig. 10. Students Collaborating Over Slides Within the Moodle Platform.

C. E-Learning Deployment

A third version of Watch Together was customized as a module for the Moodle e-learning software platform [12]. Moodle is a free and open source course management system used by Professor Ionescu for his classes at the University of Ottawa. Each student is given an account in the system that they can use to upload assignments, check their grades, download the latest course slides and more. The flexibility of the design allowed Watch Together to be intergrated using the Moodle module API [13] such that students can collaborate with each other and with the professor over course-related material. A feature was added so that the professor is able to prevent students from changing the content of the viewer module of Watch Together. In addition, all users are always joined into the same session rather than having to invite each other separately. While the system is mostly used for its document sharing feature during the professor's online "office hours", YouTube videos and other content related to the class are also made available for collaboration. The use of Watch Together in this way reveals its potential beyond entertainment and more towards enterprise-oriented collaboration scenarios.

VI. Conclusion

In this paper, we presented the design and implementation of a collaboration platform for experiencing synchronized online media from a web browser. We showed how users can collaborate over video chat while viewing videos, photos, maps, documents and more in real-time. This differs from existing collaboration solutions which may require cumbersome installations and large amounts of bandwidth. As a platform, developers can easily add new media sources and applications to the system so that all popular online digital media can be made available for instant sharing. Additionally, through a product called Watch Together, we have shown how the system can be integrated into other platforms, such as Facebook and Moodle, to enable new ways of meeting online. Through these live deployments, we observed that users enjoy collaborating on multiple types of online media in real-time with respect to one another. The usage data we obtained provided us with a better understanding of the users of synchronized online media collaboration tools and their requirements. This allows us to continue to improve all aspects of the system and to add features such as real-time document editing, drawing and games.

References

[1] (2010) Google Docs - Online Documents, Spreadsheets, Presentations. Google Inc. [Accessed: September 2010]. [Online]. Available: http://docs.google.com/

[2] (2010) Welcome to Flickr - Photo Sharing. Yahoo! Inc. [Accessed: September 2010]. [Online]. Available: http://www.flickr.com/

[3] (2010) Flex Open-Source Framework. Adobe Systems Inc. [Accessed: September 2010]. [Online]. Available: http://www.adobe.com/products/flex/

[4] (2010) Real-Time Messaging Protocol (RTMP) Specification. Adobe Systems Inc. [Accessed: September 2010]. [Online]. Available: http://www.adobe.com/devnet/rtmp.html

[5] (2010, June) Flash Player Version Penetration. Adobe Systems Inc. [Accessed: September 2010]. [Online]. Available: http://www.adobe.com/products/player_census/flashplayer/version_penetration.html

[6] W. Wang, "Powermeeting: GWT-Based Synchronous Groupware," in *HT '08: Proc. of 19th ACM Conf. on Hypertext and Hypermedia*. New York, NY, USA: ACM, 2008, pp. 251–252.

[7] M. R. Thissen, J. M. Page, M. C. Bharathi, and T. L. Austin, "Communication Tools for Distributed Software Development Teams," in *SIGMIS-CPR '07: Proc. of ACM SIGMIS CPR Conf. on Computer Personnel Research*. New York, NY, USA: ACM, 2007, pp. 28–35.

[8] Y. Liu, P. Shafton, D. A. Shamma, and J. Yang, "Zync: The design of synchronized video sharing," in *DUX '07: Proc. of 2007 Conf. on Designing for User eXperiences*. New York, NY, USA: ACM, 2007, pp. 1–8.

[9] (2010) Red5. The Red5 Project. [Accessed: September 2010]. [Online]. Available: http://red5.org/

[10] (2009, June) Watch Together. [Accessed: September 2010]. [Online]. Available: http://www.watch-together.com/

[11] (2010) Facebook Developers. Facebook Inc. [Accessed: September 2010]. [Online]. Available: http://developers.facebook.com/

[12] (2010) Moodle.org: Open-Source Community-Based Tools for Learning. Moodle Trust. [Accessed: September 2010]. [Online]. Available: http://www.moodle.org/

[13] (2010) Moodle Modules and Plugins. Moodle Trust. [Accessed: September 2010]. [Online]. Available: http://moodle.org/mod/data/view.php?d=13&rid=679