Multimedia streaming platform based on components

Ovidiu Răţoi[1], Haller Piroska[1], Genge Bela[1]
[1] "Petru Maior" University of Tg. Mureş, N. Iorga st, No. 1, Romania, 540088
[1]oratoi@engineering.upm.ro, {phaller, bgenge}@upm.ro

Abstract

We propose a platform for distributed multimedia applications which simplifies the development process and at the same time ensures application portability, flexibility and performance. Also this is a solution for adding streaming components that can change the transfer and presentation rates automatically, relying on a monitoring and control component. The platform is based on Mozilla technologies so it is portable across different platforms. It is implemented using the Netscape Portable Runtime (NSPR) and the Cross-Platform Component Object Model (XPCOM).

1. Introduction

Recent years have shown an increased interest towards multimedia rich applications. Multimedia content ranges from text or simple images to audio and video data or even animations. The increased availability of broadband Internet connections leads the way towards applications that offer high quality multimedia streaming over wide area networks. In this context there is a need for solutions that enable application developers to quickly and effortlessly develop this kind of applications.

In the same time software components technologies and component based software engineering are maturing, in fact the use of components is a sign of maturity in any field of engineering. The usage of software components offers a lot of advantages the most important of them being reusability, a component once developed may be reused in any number of applications, depending of how generic are the services it offers. Also the task of applications developers changes from development of new software to composition of existing pieces. Another characteristic property of software components is encapsulation. This property hides the internal structure and exposes a well defined interface through which the services are accessed. Encapsulation confers high flexibility to component based software as individual components can be easily replaced with improved ones as long as their interface remains identical.

Network communication was always an important issue when handling multimedia content especially when real time streaming was involved. A research team tackled this problem and proposed a multimedia applications middleware which regulated network traffic, optimized resource usage and offered a high degree of portability to applications [1].

Recently some groups are proposing a platform for collaboration systems which integrates mobile devices [2]. It uses the client-server architecture to provide multimedia content adapted to the capabilities of any devices used clients.

There were other attempts to use software components in the context of multimedia applications, so [3] proposes an architecture for-real time video applications based on CORBA. The focus was on the quality of service by monitoring the system entities load and network communication. In time the use of general purpose components was proved not to be well suited for multimedia based systems, and dedicated frameworks were developed.

Our goal was to create an interactive Web application based on components that allows bi-directional, real time communication between the resources and the user. This paper describes the component based platform proposed and implemented by us for multimedia transfer.

The paper is structured as follows. In section 2 we provide a short description of the Mozilla platform. In section 3 we provide a detailed description of the proposed platform and we describe the interface exposed by the platform and used by applications. A case study for the Multimedia Platform is presented in section 4, where beside the test results made for different type of applications the proposed platform provides an adaptive stream control component. We end the paper with a conclusion and future work in section 5.

2. Mozilla Platform architecture

Mozilla is an open source portable platform, developed and maintained by the Mozilla

Foundation, best suited for rapid development of highly interactive visual applications [4]. Its conceptual architecture is presented in *Figure 1*.

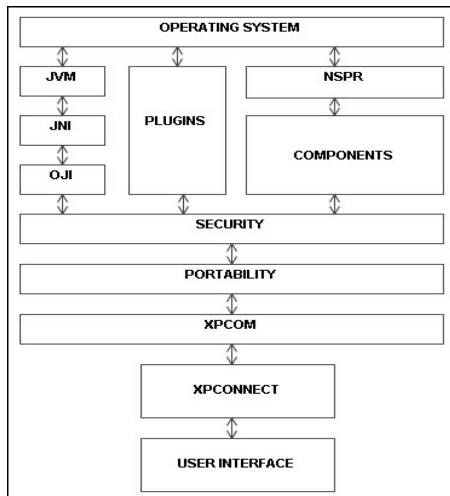

Figure 1. – Mozilla Platform Architecture

Mozilla based applications have three possibilities to access the Operating System: using the JVM (Java Virtual Machine), using plugins or through an API called NSPR (Netscape Portable Runtime). NSPR is a portable API designed to provide operating system level services like threads and synchronization support, file and network I/O, memory management, time management, atomic operations or process creation.

XPCOM (Cross Platform Component Object Model) is Mozilla's object management and discovery system very similar to Microsoft COM and remotely to CORBA (Common Object Request Broker Architecture). It was designed to provide greater flexibility to the platform and to applications developed on top of it. These components can be created in a variety of languages ranging from C, C++ to JavaScript or Python and are accessed through a set of interfaces they implement [5]. In order to provide greater portability and implementation language independence, interfaces are described in a special language called XPIDL (Cross Platform Interface Definition Language), a variant of CORBA IDL. Object lifetime management and interface discovery are implemented in a Microsoft COM style, by reference counting and special methods for querying all implemented interfaces.

XPConnect is the technology used to expose object interfaces to scripting languages, like JavaScript. User interfaces for applications are usually described in XUL (Mozilla's XML based User interface Language) [6] or HTML the well-known markup language.

Using the Mozilla platform, we focused on the development of streaming components, capable of receiving multimedia data from different sources, decode it, and deliver it to the user interface. The modular architecture of the Mozilla platform enables developers to add or remove modules with little effort, fitting the software to the available hardware and adjusting functionality to match product requirements. Our components adjust the transfer rate continuously, monitoring the devices and network capabilities. The need of the self-managing components was recently introduced in Web technologies [7], but not in implementations.

Another novelty of our approach is that the platform supports not only binary streaming (through binary channels), but also a collection of channels communicating through the use of messages specific to web services.

3. A Platform for Distributed Multimedia Applications

3.1. Platform model description

A well known method for providing a high degree of transparency and portability to distributed applications is positioning an intermediate layer called by us *Multimedia Platform* between the operating system and the application. In *Figure 2* the multilayer structure of a multimedia applications platform is presented.

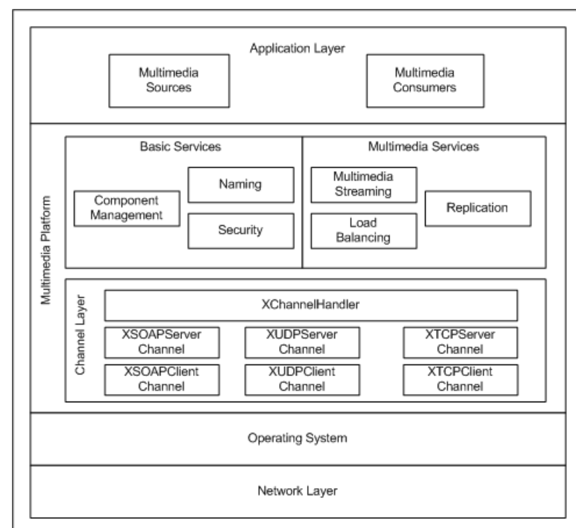

Figure 2. – Multimedia Platform Architecture

The bottom layer in this architecture is represented by the network which provides host computer interconnection and basic data transmission services. On the following level we have the operating system which provides services ranging from process management or memory organization to communication and synchronization. Above the operating system we can find the *Multimedia*

Platform which is divided into two sections: one channel layer (described later in this paper) and a service layer.

On top of the *Multimedia Platform* there is the application layer which contains multimedia sources or multimedia consumers.

The *Multimedia Platform* offers a service for data streaming between multimedia sources and consumers. Whenever a consumer needs data from a source a stream between them has to be established. The communication is based on the concept of channels.

A channel, as mentioned in the previous section, is a logical communication link between two software entities, like presented in *Figure 3*. The communication requires the existence of a connection between each pair of communicating applications. Channels embody communication protocols, while access is provided through one single interface.

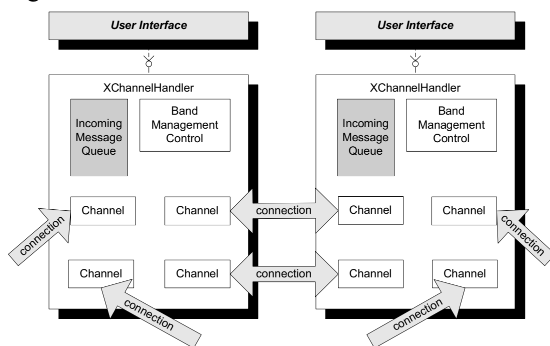

Figure 3. Platform Connection Model

The *Multimedia Platform* provides an interface for using the channels, named *XChannelHandler*. By using this interface we can create or destroy a channel, send messages to one specific channel and get the received messages from opened channels. Each newly created channel is given a unique channel ID. All operations involving channels are done through this ID. This interface exposes four functions used for creating and destroying channels, sending messages to one channel or receiving messages from the opened channels.

The platform was designed for being used in Mozilla web based client application also. In this case some elements, like receiving channel events from the platform, had to be taken into consideration. Due to restrictions imposed by the *XPConnect* technology notifications cannot be sent asynchronously from the component to the user interface thus an alternative solution had to be found. One feasible option would be to implement a waiting queue in the component for stream data or events, and then at regular time intervals the user interface would query the object. To prevent memory allocation overflow this queue had to be limited to a maximum size.

The functions exposed by the interface are:

- *int createChannel(channel_info*, unsigned int&)*. This function is used for creating one new channel. The *channel_info* is a structure which contains information for the channel which has to be created: *channel type*, *host address* and *host port*. Each time a channel is created a new channel ID is generated. This value is returned through the second argument of the function. The function returns *CHANNEL_OK* on success or a negative error code otherwise.

- *int destroyChannel(const unsigned int)*. This function is used for destroying one channel, identified by the channel ID. The function returns *CHANNEL_OK* on success or a negative error code otherwise.

- *int sendToChannel(const XMessage*)*. This function is used for sending one message to a channel. The message is encapsulated into an *XMessage* object which also contains the channel ID. The function returns *CHANNEL_OK* on success or a negative error code otherwise.

- *int getMessage(XMessage* &)*. This function is used for retrieving one message form the internal queue. All the messages received from the active channels are deployed into this internal queue. The source channel ID is contained into the message object. The function returns *CHANNEL_OK* if a message has been received or *CHANNEL_NOMESSAGES* otherwise.

3.2. Platform interfaces description

In the current form the platform provides six types of channels, but the Multimedia platform architecture offers an easy way to add more channels.

Each type of channel represents one component, which is loaded by the platform. The interface implemented by the channels is the same for all of them.

For the moment applications can chose from the following channel types: *XSOAPServerChannel* (channel accepting SOAP client connections), *XSOAPClientChannel* (SOAP client channel), *XUDPServerChannel* (channel accepting UDP client connections), *XUDPClientChannel* (UDP client channel), *XTCPServerChannel* (channel accepting TCP client connections) and *XTCPClientChannel* (TCP client channel).

The reason for introducing the concept of the UDP server and client channels was to create a similarity between the concept of a TCP connection and a UDP one. By using these channel types, applications that make use of TCP channels can be very easily switched to UPD channels, without

having to change any of the application's upper-layers, or the logic on which the application was built. When a datagram is received using UDP channels, the source IP and port are analyzed. If they are not present in the saved internal list of the *XUDPServerChannel* channel, this means it is a "new connection" and a corresponding notification message is constructed for the upper-layers. Also, the newly received IP-port pair is saved in the internal list and a new channel is created which is provided with a new channel ID.

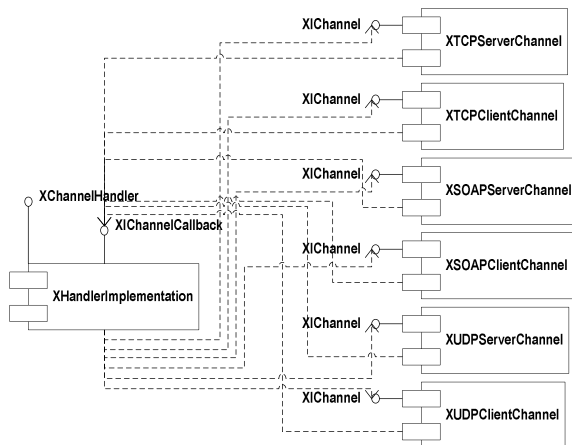

Figure 4. General view of the channel architecture

The SOAP channels are created for developing applications based on WEB Services architecture. It provides SOAP communication protocol for creating both server and client applications. The main problem with today's distributed systems is interoperability. Web Services intend to solve this problem by introducing software components that are "capable of being accessed via standard network protocols such as but not limited to SOAP over HTTP" [8].

SOAP [9] (i.e. Simple Object Access Protocol) provides a simple mechanism for exchanging structured and typed information through the form of XML messages. In order for our platform to support a standard web service interface, we created a SOAP-based channel capable of sending and receiving standard SOAP messages. For the implementation of the SOAP transport we used the open source gSOAP library [10].

The SOAP standard does not only provide a means for exchanging XML data, but also binary data through the use of *base64* or *hex* encodings. Because of this, integrating streaming data into SOAP messages becomes a straight-forward process.

SOAP messages are encoded through the form of envelopes, containing namespace definitions and a SOAP body. The body contains the actual messages. Our platform implements a *rawDataMessage* in the SOAP body for exchanging streaming binary data. An example SOAP message constructed by our channel is the following:


```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV=
"http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC=
"http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi=
http://www.w3.org/1999/XMLSchema-instance
xmlns:xsd=
"http://www.w3.org/1999/XMLSchema"
xmlns:ns= "urn:simple-calc">
<SOAP-ENV:Body
SOAP-ENV:encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/">
<ns:rawDataMessage>
<data xsi:type= "xsd:base64Binary">
QUxBIEJBTEEgUE9SVE9DQUxBÃ□Ã□
</data></ns:rawDataMessage>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```


4. Multimedia platform case study

4.1. Channel traffic load

Once the platform was operational we tested it for maximum traffic load. For this purpose, one client and one server application was designed based on the proposed platform. Both applications ware developed as standalone applications and neither of them had a graphical interface. The purpose of those two applications was to exchange messages at maximum speed. Once the messages ware received, they ware extracted from the platform and erased as quickly as possible, with no other processing made.

The tests were made over the internet using two Windows machines on a period of 160 minutes with a 1 minute sampling time. For testing purposes we used the TCP type of channels. The results are shown in *Figure 5* and *Figure 6*.

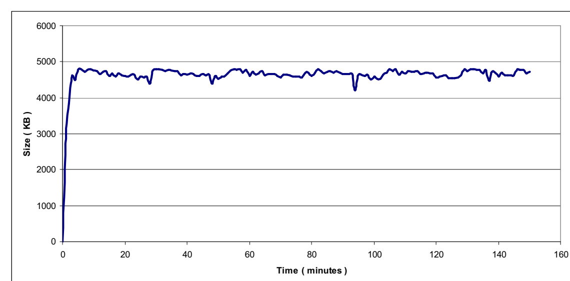

Figure 5. Test results for 1024 B packets with 1 minute time interval

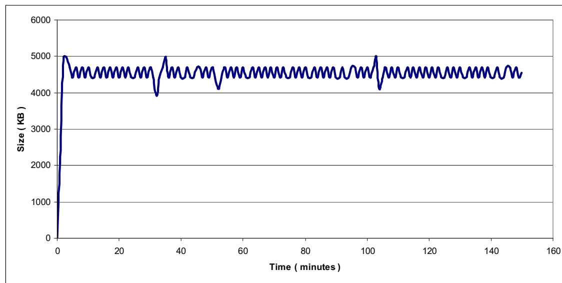

Figure 6. Test results for 10024 B packets with 1 minute time interval

The differences between the two tests are in the size of the packets send through the channels. The first test used 1024 B packets and the second one used packets 10 times bigger. The spikes on the graph appeared when the internal queue size reached the maximum value and, as a failsafe measure, the platform stopped reading data from the channels. In this case, as we can see on the graph also, the channel traffic load showed a decrease until some of the incoming messages were processed. After the internal queue size had dropped under that critical value the channel data reading started again and the measured band has increased again. As we can see from the graphics the maximum traffic load over the internet using the current architecture of the platform is in the range of 4000 to 5000 Kb of data. This value is a satisfying one for a client-site application used in real time multimedia streaming, and especially for a web-based client application.

4.2. Video streaming application test results

For testing the platform in a multimedia environment we had created a stream server running in *XULRunner* and a web-based client application running in "*Mozilla Firefox 2.0.0.20*". Both of them used the proposed multimedia platform. For the client application, the interaction between the browser and the platform was made using Java Script. For testing purposes we opened several instances of the client application that were connected to the stream server. Several sources were also connected to the stream server and the client applications received frames from them.

Using the model presented above, the video streaming application was tested on several platforms with variable number of cameras. Three parameters were measured, ***Incoming Bandwidth*** resulting from data received on the communication channel established with the stream server, ***Outgoing Bandwidth*** resulting from the total size of the video frames transmitted to and displayed by the user interface and ***Queue Size*** representing the number of video frames stored in the object's waiting queue. All tests were conducted with the same application, stream server and cameras on a period of 10 minutes with a 4 seconds sampling interval. Once again the TCP based channels were used again. The results are presented in the following figures.

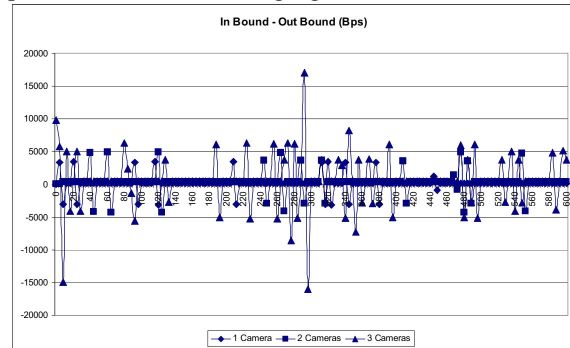

Figure 6. Bandwidth on Windows XP

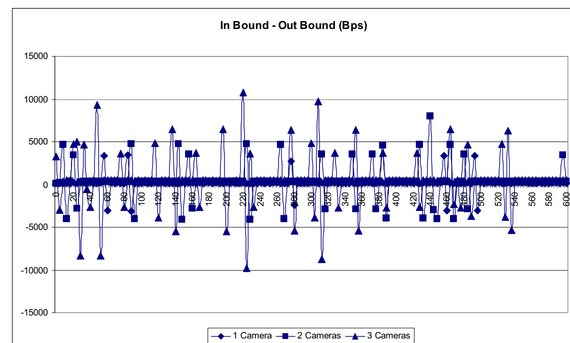

Figure 7. Bandwidth on Mac OS X

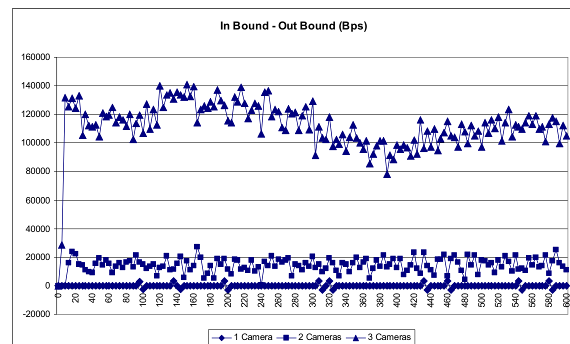

Figure 8. Bandwidth on Linux

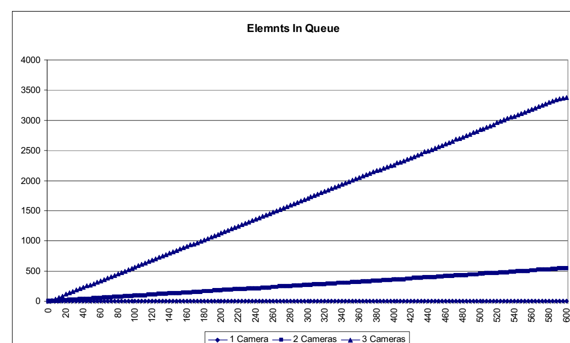

Figure 9. Queue size on Linux

Test results show that the application performs well on both ***Windows*** and ***Mac OS*** environments and although there are oscillations in bandwidth, the waiting queue never grows bigger than two frames which, considering a data rate of 7-10 fps from each camera, translates into a very small delay, even when receiving stream from three different cameras. The performance of the same application is significantly worst in the ***Linux*** environment especially when video stream is received from more than one camera.

The operating systems are not responsible for the different performances of the platform. As a mater of fact, the ***Incoming Bandwidth*** on all of the tested platforms was in the same range. The differences ware because of the ***Outgoing Bandwidth***. Mozilla Firefox has a different behavior on those environments when rendering frames.

Figure 8 shows that when displaying images from three different cameras the difference between incoming and outgoing bandwidth is quite high, which produces an abrupt accumulation of frames in the waiting queue, as it can be seen from *Figure 9*. From this increase of the waiting queue size results an unacceptable delay in the video stream.

4.3. Adaptive stream control

For using an adaptive stream control, the stream servers need to have a mechanism for setting the prescribed value for client bandwidth. Some of them accomplish this by recompensing the multimedia stream accordingly to the prescribed value. Other servers accomplish this by dropping some of the frames that should be sent to the client.

Exploiting this facility could improve application's performance on some platforms by reducing delays in stream, especially when a large number of devices are observed. Because the internet bandwidth can vary in time and the application is an interactive one where the number of devices from which stream is received can vary in time, an adaptive control mechanism has to be implemented. A possible solution would be to introduce a new XPCOM component responsible for gathering parameter values measured by the channels and taking control decisions according to them.

In the proposed model the *Band Management Control* component has a passive role. Every channel reports periodically to it the ***Incoming Bandwidth***. The ***Outgoing Bandwidth*** is also computed periodically.

The adaptive control algorithm is using those two values along with the ***Queue Size*** for computing the maximum ***Incoming Bandwidth*** for each channel.

This value is send to the streaming server, which in turn will set the prescribed value for client bandwidth. Because out stream server accomplishes this by dropping some frames video quality will decrease but there will not be any delays, thus maintaining the real-time quality of the stream. Test results are presented in *Figure 10* and *Figure 11*.

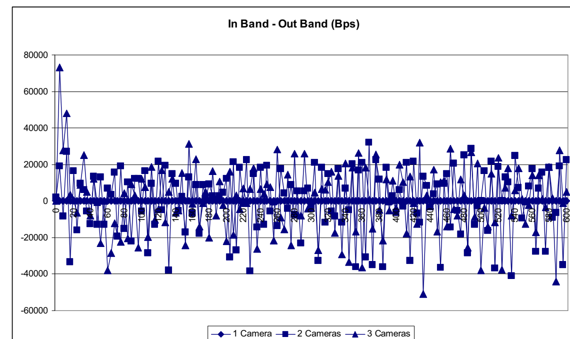

Figure 10. Controlled bandwidth on Linux

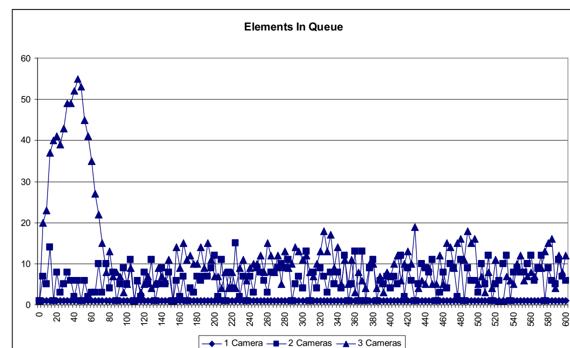

Figure 11. Queue size on Linux with bandwidth control present

Figure 11 clearly shows that when bandwidth control is present, even if images from 3 cameras are received, the size of the waiting queue decreases and then maintains its value in a relatively small interval, under 20 frames. Taking into account that the number of frames captured from a camera in one second ranges between 7 and 10 frames this produces only a small, acceptable, delay in the video stream.

4.4. Scalability issues

Further tests have been made in order to demonstrate that the model we proposed is scalable. Four separate channels were used each handling up to 3 different streams summing up to a total of 12 simultaneous video streams. System's performance can be seen in *Figure 12* and *Figure 13*.

These are the results only for one operating system but the others behave in a very similar manner. They show that the system can handle 12 simultaneous streams and although there is an initial build up of elements in the platform's queue the control mechanism quickly reacts and reduces the

load by limiting the maximum bandwidth. Such a rapid response is obtained by taking into account the current incoming and outgoing values of the bandwidth when adjusting its maximum value.

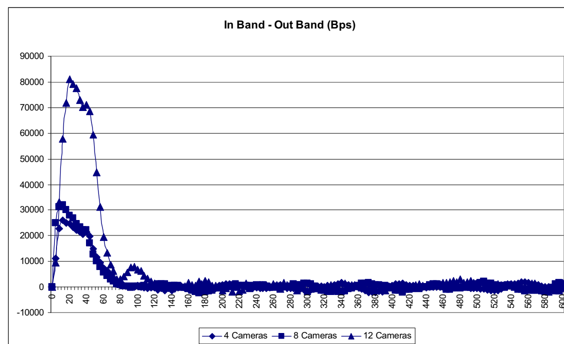

Figure 12. Bandwidth on Windows

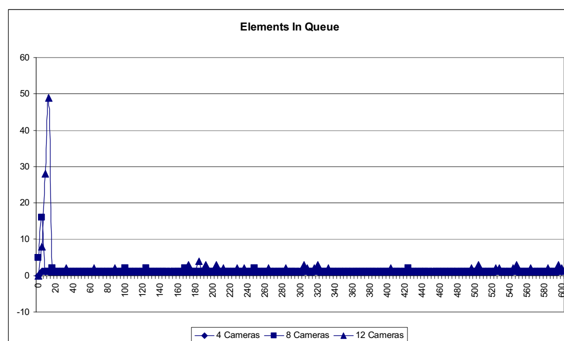

Figure 13. Queue size on Windows

Figure 14 shows the response in case of a variable number of streams. There is a small accumulation of elements in queue whenever the number of cameras is increased but it soon disappears. Also a small spike on the graph can be seen when the number of cameras is decreased, this is because the bandwidth regulator tries to use as much as possible from the available resources thus increasing the maximum bandwidth automatically and in this way causing a small accumulation of elements before it stabilizes.

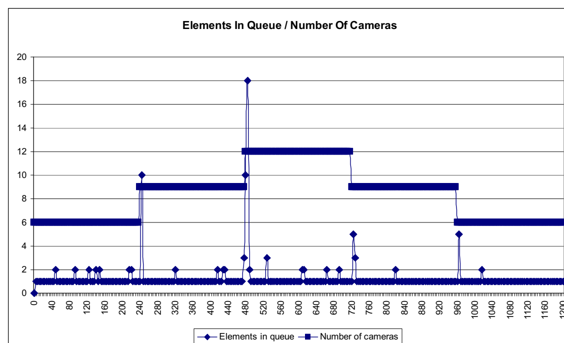

Figure 14. Response to variable number of streams

5. Conclusions and future work

In this paper we proposed a component based, real time streaming, portable, Web platform for distributed multimedia applications, developed on the Mozilla Platform.

Based on this model we implemented a set of components that can be used in any kind of multimedia streaming application. Furthermore a monitoring and control mechanism was presented, which allows the application to dynamically change transfer rates in order to reduce delays in the stream caused by slow presentation rates.

This approach also simplifies the development of multimedia centered applications and ensures their transparency, portability and performance. By providing a unique interface for all supported channel types, application developers can easily change the underlying transport and channel type.

As future work we intend to extend the adaptive stream control mechanism and to prepare the multimedia platform for usage in a dynamic QoS environment. This means that applications could fine tune the adaptive control mechanism in such a way that different types of multimedia content should be treated different. What this means is the fact that audio streams could be preferred over video ones or even the other way around if necessary.

References

[1] M. Lohse, M. Repplinger, P. Slusallek, *"An Open Middleware Architecture for Network-Integrated Multimedia"*, Proceedings of the Joint International Workshops on Interactive Distributed Multimedia Systems and Protocols for Multimedia Systems: Protocols and Systems for Interactive Distributed Multimedia, Portugal, pp. 327-338, 2002.

[2] X. Su, B. S. Prabhu, C. C. Chu, R. Gadh, *"Middleware for Multimedia Mobile Collaborative System"*, Proceedings of IEEE ComSoc Third Annual Wireless Telecommunications Symposium (WTS 2004), USA, pp. 112-119, 2004.

[3] V. Kalogeraki, L. E. Moser, P. M. Melliar-Smith, *"A CORBA Framework for Managing Real-Time Distributed Multimedia Applications"*, Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS 2000), Vol. 8, Hawaii, pp. 8042, 2000.

[4] Alan Grosskurth, Ali Echihabi, *"Concrete Architecture of Mozilla"*.

[5] Doug Turner, Ian Oeschger, *"Creating XPCOM Components"*, Brownhen Publishing, 2003.

[6] Nigel McFarlane, *"Rapid Application Development with Mozilla"*, Prentice Hall, 2003.

[7] H. Liu and M. Parashar, *„Rule-based Monitoring and Steering of Distributed Scientific Applications"*, International Journal of High Performance Computing and Networking (IJHPCN), issue 1, 2005.

[8] http://www.oasis-open.org/committees/wsia/glossary/wsia-draft-glossary-03.htm

[9] http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[10] Robert A. van Engelen and Kyle Gallivan, *"The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks"*, in the proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002), pp. 128-135, May 21-24, 2002.