Scalable Framework for Live Data Sharing Through 802.11

Eduardo Soares, Pedro Brandão and Rui Prior

Abstract We propose a multi-platform framework for easy development of applications that share live or recorded data of any type in a classroom. It is especially aimed at training in the medical area, where it can make the learning process much more interactive and enriching, but is equally well suited for use in any type of workshop, tutorial, or other learning environment. The framework is browser-based, for better portability. In order to scale well to a large audience, the framework uses multicast for communication. It provides configurable reliability that is adaptable to data flows with different requirements, real time (RT) or not. It also provides security, privacy and access control features that are necessary in medical training environments. Finally, it allows session discovery and management, and multi-sender support.

Keywords Multicast · Network · Reliable multicast

1 Introduction

Technology has been reshaping our concept of teaching; the educational content became richer with the usage of elements like ultrasounds, electrocardiograms, auscultations or even 3D models [2]. These materials need to be shared between teachers and students and in the process cannot be transformed in any way that

E. Soares (✉)
Faculdade de Engenharia da Universidade do Porto,
Instituto de Telecomunicações, Porto, Portugal
e-mail: easoares@fe.up.pt

P. Brandão · R. Prior
Faculdade de Ciêcncias da Universidade do Porto,
Instituto de Telecomunicações, Porto, Portugal
e-mail: pbrandao@dcc.fc.up.pt

R. Prior
e-mail: rprior@dcc.fc.up.pt

A. El Oualkadi et al. (eds.), *Proceedings of the Mediterranean Conference on Information & Communication Technologies 2015*, Lecture Notes in Electrical Engineering 381, DOI 10.1007/978-3-319-30298-0_19

makes major changes irreversible. Compression with too much loss, to reduce transmission size, is not an option, making it difficult to send to many users.

Current options for solving this problem use third party services, such as cloud storage or a content management system. In education, it is common to use Moodle [5]. These options are not the best solution; they mandate a connection to a server, possibly outside the local network, for all users. This places unnecessary load on the network nodes, as in a classroom scenario origin (lecturer) and destination (audience), are under the same access point (AP), and thus on the same physical link.

Some solutions are also device dependent. Applications for a single platform create barriers, and make it hard to use in the real world, where multiple platforms abound. To avoid this, solutions should be device independent and open. A good option is using the web browser as a multi-platform enabler. All the modern operating systems have a variety of web browsers to choose from, and most of them support the latest standards of HTML, CSS and JavaScript.

1.1 Scenario

Our working scenario focuses on a classroom or conference, where all the participants are in a room over the same Wi-Fi 802.11 link of a single AP. We assume that all the participants should be able to access the transmitted content. In addition, no one without physical access to the room should be able to access the shared content, even if the wireless link is accessible. The accessed content can be a live stream or previously recorded. This last point creates a special case, as it needs to look like a real time (RT) stream.

1.2 Objectives

As per the scenario and its requirements, the work in this paper aims to accomplish the following objectives: (i) **Reliable delivery**: all packets sent should be received by all users that are the intended recipients; (ii) **Scalability**: the framework should accommodate from a small number of users up to a few dozens; (iii) **Security**: it is crucial that only authorized users receive the shared content and that receivers can validate the senders; (iv) **Work without Internet**: there should be no need for a reliable Internet connection. An example could be in a conference, to access the Internet some login and credentials could be needed but it can be easy to access the local Wi-Fi 802.11 network; (v) **RT content**: even if the platform does not capture content, it is important to adapt when the content needs to appear as RT to the receivers; (vi) **Multi-platform**: not everyone has the same device or uses the same platform. The solution needs to work within the maximum number of platforms in order to cover the maximum number of users.

2 Proposed Solution

At the time of this writing, no web browser Application Programming Interface (API) allows to send data in a multicast connection. Furthermore, only a small sub-set of them is prepared to communicate directly between each other with little usage of an intermediate web server. As such, the creation of a pure, in-browser solution is currently impossible. To overcome these problems, we developed the communications part of the solution in pure Java. This part offers a connection to the web browser so anyone can use HTML tools to create the user interface.

Figure 1 shows a simplified structure of the architecture developed. The components are a library that implements a service to create sessions for multicast communication in a secure and reliable way; a web server to expose the data received and the library API for the browser where the application runs.

The service creates a representation of a session for sharing data; a group of channels forms each session. A user owns each channel, and each user can have multiple channels. A central coordinator, called session manager, manages and controls the communication. The session manager is also a fundamental part of the search protocol and authentication described respectively in Sects. 2.1 and 2.3.

2.1 Search and Discovery

For the search protocol, the session manager periodically sends a packet with the sessions' information to a predefined multicast channel. Each user that wants to know what sessions are available listens to that channel. To avoid too much noise its session manager sends this every 20 s.

Users that are listening and waiting to receive can send a packet requesting this information, so as to speed up the process. When a session manager receives this request, it waits a short, random time, from 0 to 10 s before sending it. This strategy tries to avoid collisions of packets from multiple session managers, each with its own session.

All the waiting time values used in this protocol are based on the protocols studied that are presented in Sect. 3.

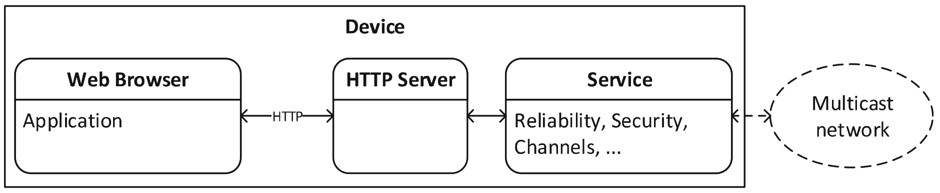

Fig. 1 Architecture of the solution

2.2 Reliable Multicast

To provide reliability on the multicast connection, we chose a NACK strategy. With this approach, a receiver issues a request when it determines that it lost a packet. These requests are sent unicast to the original sender.

In order to avoid request collisions from multiple receivers reporting a missing packet, each one waits a random time between 0 and 10 s.

To optimize request sending, we can ask for several lost packets in the same NACK message. This avoids extra packets for each lost packet in a receiver, and cooperates with the random waiting for sending the NACK.

When the sender receives a NACK it resends the packets that were lost. A possible optimization is using network coding [4] to minimize the number of packets sent. This provides the opportunity to send a network-coded packet that contains information for multiple receivers.

Given the nature of a live stream environment, waiting for a lost packet can break its liveness. To improve the user experience, we have an option to forget about a missing packet after a chosen time has elapsed.

The application that creates the sending channel defines the value for forgetting the packets, because it can create a buffer at a higher level appropriate to the content and adjust the waiting time for missing packets.

2.3 Security

Security is one of the key points given the nature of the data. Only users that are supposed to get the data should be able to see it. It should not be possible for an outsider, which is not allowed in the room where the session is happening, to obtain any important data, even if he can capture all the transmitted packets. For this, we need authentication of the users and secure data transmission.

Authentication. We need to block users that should not have access to the data. For this, and given the environment described in Sect. 1.1, we use a protocol based on public key (PK) cryptography. It allows users to prove they are in the room and have access to a token given by the session creator. It work as follows:

1. The creator of the session generates a public/private key pair;
2. The PK is advertised in the session announcement as described in Sect. 2.1;
3. The PK fingerprint and an authentication token to authenticate the users to the session creator is transmitted by the user who created the session through analogue means (e.g. writing on the classroom board);
4. Users validate the PK and perform authentication to obtain the session information and cipher keys. Defining the client as the user that wants to connect to the session, and the server as the session creator, we have:

(a) Client opens a connection to the Authentication Server (AS);
(b) AS replies with the PK that must match the one advertised;
(c) Client generates a symmetric key (SK) and sends it to the server ciphered with the PK received. From this point onwards, all the messages are encrypted with this SK;
(d) Server challenges the client with a nonce;
(e) Client answers the challenge with a hash of the nonce concatenated with the authentication token;
(f) If the client correctly answers the challenge, the server sends information about the session (SK to cipher the stream data, known channels of data and session management channel). Otherwise an error is replied.

Secure Data Transmission, Key and User Management. From the authentication protocol, users receive a key that ciphers all the data shared in all the channels. This causes a security problem, as an authenticated user could introduce fake packets inducing corruption of the shared data. We assume that this is not a problem, and authenticated users are not inside attackers.

If a user leaves the session, the SK does not change. We use this approach for two reasons. The first is that in a connectionless environment it may be hard to detect if a user leaves. The second reason would be that even if the user notified he was leaving, the session would not be more secure from knowing that and changing the key. As per defined protocol for authentication, the user could redo it and get the new key. We follow a security policy similar to the one used in classrooms. Whoever is attending can access the session. Of course, if the channel to transmit the authentication token is more private we can restrict more the allowed users.

2.4 Session Management

To keep a coherent vision of the session between users, and inform them of creation deletion of data channels, we developed a session management protocol.

To create a channel, a user issues a request to the session manager. Then, the library that is running the protocol asks who is using it if this user should be authorized. The answer is transmitted to the user and if affirmative, the session manager sends this information to the management channel.

The management channel is a channel that all users must be listening in order to receive information of created and deleted channels. To avoid fake packets in this channel, all packets are signed with the private/public key of the session manager (also used in the authentication protocol).

2.5 *Web Server and HTTP API*

As described in the start of this section, we need a web server to load an application built in HTML, JavaScript and CSS and to provide an HTTP API of the created library.

Given the multitude of devices and the simple requirements, we built a small, custom solution. The most complex requirement was to fully support the HTTP API. This API exposes the session feature and reliability from the created library. It has three main components: **Session**: Information about surrounding and connected sessions. Ability to search for sessions and/or create new ones; **Channels**: Information about existing channels in the currently connected session. It also allows to create new channels, remove existing ones, and send to or receive from a channel; **Notifications**: Built over Server-Sent Events (SSE) [10], with the option to reply to a notification that needs an answer.

In addition to the described API, we developed a solution to surpass the web browsers' limitations. Nowadays, web browsers have rich options in order to capture video or audio and use files. However, they cannot go much further to talk with other types of hardware, like Bluetooth or USB devices. This causes limitations in the described scenario, where many acquisition devices interface with the operating system over Bluetooth and with custom protocols on top.

The data generated by these types of hardware has to be pipelined into the multicast channel. A possible solution could be the usage of PhoneGap [1]. This framework allows developing an application in HTML, CSS and JavaScript and then porting it to different OSes. It provides a JavaScript API to access lower capabilities of the hardware that are normally blocked in the browser environment. However, in our scenario, this would mean that the information would flow from the Java Virtual Machine (JVM) (adaptation library from PhoneGap), to the web viewer (that PhoneGap uses to host the HTML, CSS and JavaScript code) and back to the JVM (our library for the multicast sessions). This would create a round trip time that could become noticeable, an extra overhead in the application in order to include the PhoneGap wrapper and creating a dependency, thus making it impossible for a native application to reuse our library.

To avoid this, we introduce the plugins component. Plugins are modules that have a single function and can connect to other plugins in order to form a graph. They can be a source of data, which means that they connect to an external device, collect its data and send it to another plugin. In order to send the data, plugins can be of the output type, where they send from one or more data sources (multiplexing) to an output. This destination can be a channel of the multicast library or other location, like a local file or a remote server. To operate over the collected data, we use a processing plugin. This makes it possible to create filters or encoders for the collected data before sending it to an output.

The graph with input, processing and output plugins operates within the JVM, thus avoids transporting the data through different layers.

Viewers follow the same idea of reusable components across applications, similar to plugins, to allow data visualization. This component runs in the web browser and is self-contained without external requirements. Each viewer displays a particular type of data. Given the passive nature of this component, we can add new viewers uploading them via an HTTP API. The HTTP API also exposes the loading of a viewer, listing existing viewers and their information (data type supported, parameters accepted, name and description) and removing viewers.

3 Related Work

In this section, we discuss some solutions to part of the problems that we enumerated previously.

3.1 Reliable Multicast

Through the years, there have been some proposals to achieve reliability over multicast. They can be divided in two categories The first tries to copy TCP, having the receivers acknowledge the sender every packet received. This causes much overhead in the communication, and does not scale well. Protocols that do this usually have optimizations, like using intermediate nodes to receive acknowledgements and do repairs in case of failure. These nodes can be other receivers or specialized network equipment. The other option is to assume that every node receives the packets, and when some do not, they inform about the failure and try to have it resent. This mechanism is called NACK.

Both solutions add some extra information to the packets sent to identify them. The minimum data that needs to be added is a sequence number.

The Tree-Based Reliable Multicast (TRAM) [3] protocol was built to send data from a source to multiple receivers at more than a hop of distance. This protocol builds a tree of receivers where the sender is the root, and works like the TCP adaptation mentioned previously. The parent of a node in the tree is the one that receives the acknowledgement and replies when packets are missing. The protocol defines extra packets, to inform when a session is still running but the source is not sending data.

The Lightweight Reliable Multicast Protocol (LRMP) [8] was designed for environments that can handle package delay, as long as there are guarantees that all of them arrive. It recovers missing packets by sending NACK messages to the multicast group. An element of the group that has the lost packet sends it to the sub-group of the requesting node. The protocol uses NACK suppression to avoid NACK messages collision. It also employs a strategy of trying to recover the packet in a close node. To achieve this, it sends the NACK with a small Time To Live (TTL). After a time out, it resends the NACK after increasing its value. This mandates that each node must keep a queue of received packets to answer requests. The protocol also has some optimizations to avoid congestion, adapting the transmission rate. Moreover, it gives the possibility to use Forward Error Correction (FEC) packets [9].

The Pragmatic General Multicast (PGM) [6] protocol uses some of the previous techniques to achieve reliable multicast. It uses NACK sent via unicast to the sender, and has optional usage of FEC packets, among others. It also has the option of using specialized network equipment to improve the packet repair on reception failure. In this protocol, routers can suppress NACKs that other receivers already asked for and are waiting for replies.

3.2 Multi-platform Development

We chose to start by testing in Microsoft Windows and Android because they are the most used platforms on computers and mobile devices.

While there currently exist options that allow for the development of a single solution for the targeted platforms, most of these are based on developing a web application. This implies an application coded in JavaScript, along with the usage of some platform-specific wrappers added at compile-time. Examples of this can be found in [7]. Our approach, however, overcomes the limitations of the web browser APIs for communication.

4 Conclusion

The work presented aims to achieve a solution to enable easy development of multi-platform applications that can share data using multicast communications without the need of a remote service.

Multicast allows achieving the best scalability possible in an environment where the network equipment cannot scale at the same rate as that of the number of devices. While we are still limited by the AP that connects the users, the fact that we do not go outside the local network alleviates congestion problems that would otherwise affect other network nodes. Multicast also avoids duplication of data per receiver, removing overload of packages at the AP.

The objectives set forth in the introduction were achieved, although with some constraints.

The reliable delivery cannot be assured in RT. Waiting for a lost packet can take a variable time, and if the request for it is also lost, the process starts again. Given the nature of a wireless environment with several users, this is prone to happen. So we trade-off to support soft RT.

Scalability by default: we built the solution to always use multicast to send data. With a small number of users, it could be a better option simply to use TCP; the overhead of the communication would not be a problem. TCP would give reliability but it could affect the RT aspect of the stream.

Multi-platform is only tested on Android and MS Windows: but all the software should run without problems in any other platform that supports the Oracle JVM and a web browser (including but not limited to Linux and Mac OS). The web browser requirements are for the SSE and XMLHttpRequest API to be present, which is the case in all the modern web browsers (e.g. Mozilla Firefox or Google Chrome). Stricter requirements only depend on the applications created using this framework.

An important point that was achieved was the capability to function without an Internet connection. This required extra steps for establishing a connection in a secure way.

In the future, we should support easier authentication via previous exchange of PKs between the users involved in the session. This removes the need to share key fingerprints and authentication tokens, making it possible to authenticate specific recipients. Also notably missing and currently under development are tests against real-live scenarios and comparison with TCP based solutions.

Acknowlegments This work was funded by Project I-CITY—ICT for Future Health, NORTE-07-0124-FEDER-000068, funded by the Fundo Europeu de Desenvolvimento Regional (FEDER) through the Programa Operacional do Norte (ON2) and by national funds through FCT/MEC (PIDDAC).

References

1. Adobe: PhoneGap|about: http://phonegap.com/about/. Accessed 29 Aug 2014
2. Body, V.: Visible body|3D human anatomy: http://www.visiblebody.com/index.html (2014). Accessed 03 Sept 2014
3. Chiu, D.M., Hurst, S., Kadansky, M., Wesley, J.: TRAM: a tree-based reliable multicast protocol (1998)
4. Costa, R.A., Ferreira, D., Barros, J.: FEBER: feedback based erasure recovery for real-time multicast over 802.11 networks. CoRR abs/1109.1265 (2011)
5. Dougiamas, M., Taylor, P.: Moodle: Using learning communities to create an open source course management system. In: World Conference on Educational Multimedia, Hypermedia and Telecommunications. vol. 2003, pp. 171–178 (2003)
6. Gemmell, J., Montgomery, T., Speakman, T., Crowcroft, J.: The PGM reliable multicast protocol. IEEE Netw. **17**(1), 16–22 (2003)
7. LeRoux, B.: PhoneGap|PhoneGap, Cordova, and what's in a name? http://phonegap.com/2012/03/19/phonegap-cordova-and-what%E2%80%99s-in-a-name/ (March 2012). Accessed 29 Aug 2014
8. Liao, T.: Light-weight reliable multicast protocol specification. Internet-Draf: draft-liao-lrmp-00.txt 13 (1998)

9. Luby, M., Vicisano, L., Gemmell, J., Rizzo, L., Handley, M., Crowcroft, J.: The use of forward error correction (FEC) in reliable multicast. Technical report RFC 3453, December (2002)
10. W3C: Server-sent events. http://ww1w.w3.org/TR/eventsource/ (December 2012). Accessed 29 Aug 2014