POLITECNICO DI TORINO
Collegio di Ingegneria Informatica

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Estensione delle funzionalità della piattaforma di videoconferenza BigBlueButton

Relatore:
Prof. Antonio Servetti

Candidato:
Daniele Viscuso (M. 256907)

OTTOBRE 2021

Keywords:

Multimedia, stream, BigBlueButton

Indice

Indice delle figure

1 Introduzione

I sistemi di videoconferenza hanno assunto un ruolo di primo piano nella vita aziendale di tutti i giorni. La possibilità di comunicare da remoto con persone sparse per tutto il mondo ha agevolato il fenomeno dello smart working, permettendo -dove possibile- ai lavoratori di continuare a svolgere i loro compiti da casa quando impossibilitati a raggiungere la sede di lavoro. Un discorso analogo ha cominciato presto a farsi strada anche all'interno dei sistemi universitari: alunni e docenti che non hanno la possibilità di incontrarsi fisicamente per organizzare delle lezioni, possono comodamente creare delle classi virtuali, permettendo in questo modo il normale svolgimento delle lezioni.
Negli ultimi anni si è avuta una rapida crescita nell'utilizzo dei sistemi di videoconferenza, dovuta anche all'emergere della pandemia del COVID-19 a livello globale. Si è stimato che il traffico medio generato quotidianamente sia salito del 535% nel 2020 [1], a testimonianza del fatto che la presenza di questi strumenti sia stata fondamentale nel permettere a determinate attività di continuare a funzionare.
Ovviamente esistono tantissimi software diversi in grado di creare e gestire una videoconferenza (si pensi ad esempio a Microsoft Teams, Zoom, eccetera), ognuno con le proprie caratteristiche e funzionalità aggiuntive.
In questa tesi verrà analizzato in particolar modo uno specifico sistema di videoconferenza, ossia BigBlueButton, una piattaforma usata dal Politecnico di Torino per permettere la creazione di classi virtuali duranti gli ultimi anni didattici.
L'obiettivo di questo lavoro di tesi è quello di analizzare questa piattaforma e implementarne nuove funzionalità, volte principalmente a migliorare le prestazioni del sistema e ridurre l'effettivo uso di banda da parte degli utenti connessi.
In particolare, le funzionalità aggiunte in questo progetto sono le seguenti:

- Possibilità di generare un flusso video a frame-rate ridotto della webcam
- Rimozione e re-inserimento degli stream webcam di altri utenti
- Inserimento degli stream webcam in una finestra esterna

2 BigBlueButton

Prima di addentrarsi nella descrizione delle funzionalità implementate verrà fornita una breve introduzione sul sistema BigBlueButton, volta soprattutto a illustrare i componenti principali dell'architettura e le loro interconnessioni.

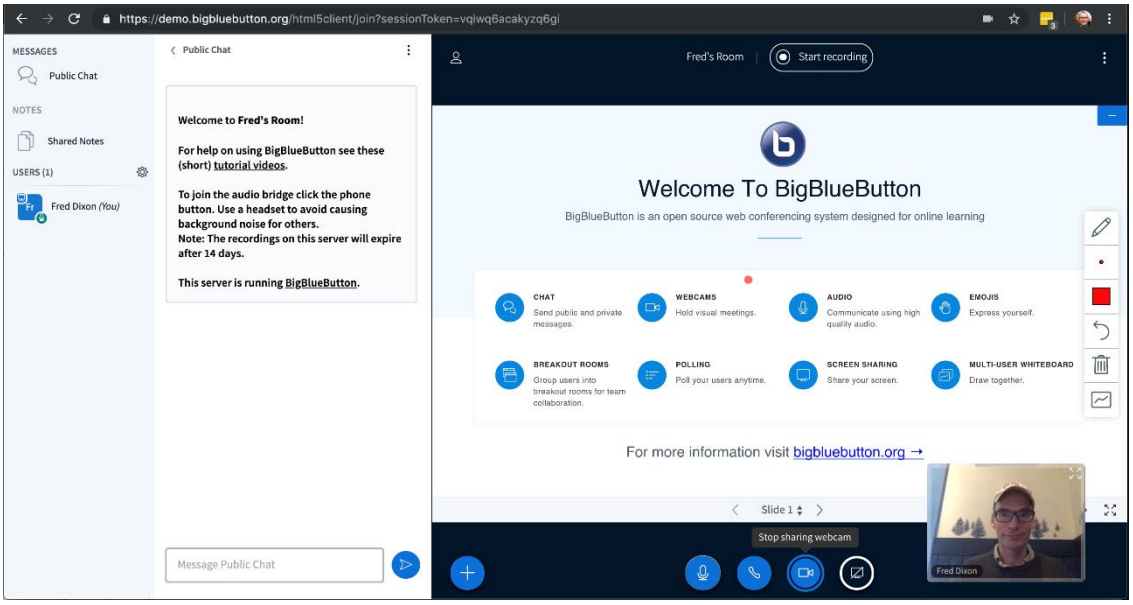


Figura 1 – Pagina principale di BigBlueButton

2.1 Architettura

BigBlueButton si compone di più componenti, che comunicando tra loro permettono il corretto funzionamento dell'applicazione.[2] In questa sezione verranno analizzati quelli principali, e le interazioni che tra di essi intercorrono.

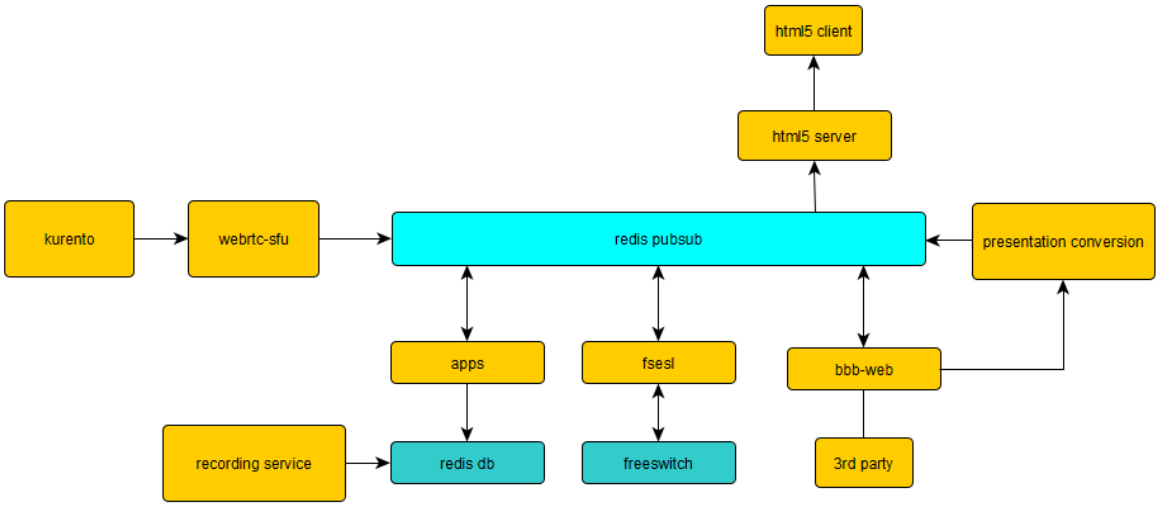


Figura 2- Componenti di BigBlueButton

2.1.1 Apps-Akka

Questo modulo rappresenta il cuore dell'intera architettura, ed è forse il più importante fra tutti.
Apps-Akka (in figura 2 indicato come Apps) si occupa di coordinare tutti gli altri componenti, dialogandoci attraverso l'utilizzo di messaggi inviati tramite un canale creato da Redis PubSub (un altro modulo di BigBlueButton, analizzato in seguito).
Per svolgere questo compito si avvale dell'utilizzo di RedisDB, un database che viene usato per memorizzare gli eventi interni alla classroom.
Ogni volta che viene generato un nuovo evento all'interno del meeting, questo viene intercettato da Apps-Akka, che lo registra all'interno di RedisDB, esegue le necessarie azioni a supporto ed eventualmente redirige il messaggio verso altri moduli.
E' di fatto lui il modulo che consente la collaborazione real time tra utenti diversi, che permette la corretta gestione della lavagna virtuale, della lista delle chat e anche delle registrazioni della videoconferenza.

E' sempre lui a moderare l'invio di messaggi verso altri peer del meeting. A titolo di esempio viene mostrato un estratto di codice preso dal modulo, chiamato al momento dell'attivazione della webcam da parte di un utente.

```
def broadcastEvent(msg: UserBroadcastCamStartMsg): Unit = {
  val routing = Routing.addMsgToClientRouting(MessageTypes.BROADCAST_TO_MEETING,
props.meetingProp.intId, msg.header.userId)
  val envelope = BbbCoreEnvelope(UserBroadcastCamStartedEvtMsg.NAME, routing)
```

```
  val header = BbbClientMsgHeader(UserBroadcastCamStartedEvtMsg.NAME,
props.meetingProp.intId, msg.header.userId)

  val body = UserBroadcastCamStartedEvtMsgBody(msg.header.userId, msg.body.stream)
  val event = UserBroadcastCamStartedEvtMsg(header, body)
  val msgEvent = BbbCommonEnvCoreMsg(envelope, event)
  outGW.send(msgEvent)
}
```

L'oggetto *outGW* rappresenta un canale di comunicazione tra il componente e Redis PubSub. In questo caso sul canale viene inviato un messaggio identificato dal type *BROADCAST_TO_MEETING*, che identifica l'invio verso tutti i membri interni alla videoconferenza, e il messaggio viene così rediretto da PubSub agli altri peer.

Apps-Akka dialoga anche internamente col servizio di video recording interno a BigBlueButton.
Quando in una sessione l'opzione di registrazione è disponibile, il server BigBlueButton salva i contenuti del meeting tramite i suoi sottomoduli: le slide, le chat, la lavagna virtuale, i flussi webcam, di screensharing e audio vengono così catturati e memorizzati. La memorizzazione di questi contenuti avviene secondo diverse modalità: le chat e gli eventi legati al cursore o alla lavagna sono gestiti per esempio direttamente da Apps -che li salva su RedisDB-, mentre i flussi video sono gestiti da Kurento (di cui si parlerà in seguito). Quello che è importante ricordare è che all'interno del server sono presenti delle directory predefinite, che contengono i vari stream (audio e video) registrati, dividendoli in sottodirectory identificate dall'id del meeting a cui fanno riferimento. E' facile così, in fase di post-processing, trovare le registrazioni relative a uno specifico meeting.

Quando un utente fa partire la registrazione -cliccando sull'apposito pulsante- viene generato un evento che viene intercettato da Apps-Akka e salvato all'interno del db. Vengono così generati dei marker, che identificano l'inizio e la fine della registrazione della videoconferenza da parte dell'utente.

Al termine del meeting viene quindi verificata la presenza di questi marker, che se presenti fanno scattare un meccanismo interno di post-processing (gestito da un apposito script) per l'elaborazione della registrazione da rendere disponibile agli utenti.

Questo processo consiste essenzialmente nel recuperare tutti gli eventi e i media precedentemente salvati, e "montarli" insieme sulla base del timestamp a cui sono associati i suddetti marker. Se la registrazione comincia, ad esempio, a 3 minuti dall'inizio del meeting, allora solo i contenuti effettivamente visibili da quel momento in poi saranno resi disponibili. Il risultato finale di questa operazione è un file multimediale memorizzato sul server BigBlueButton e accessibile tramite apposite API.

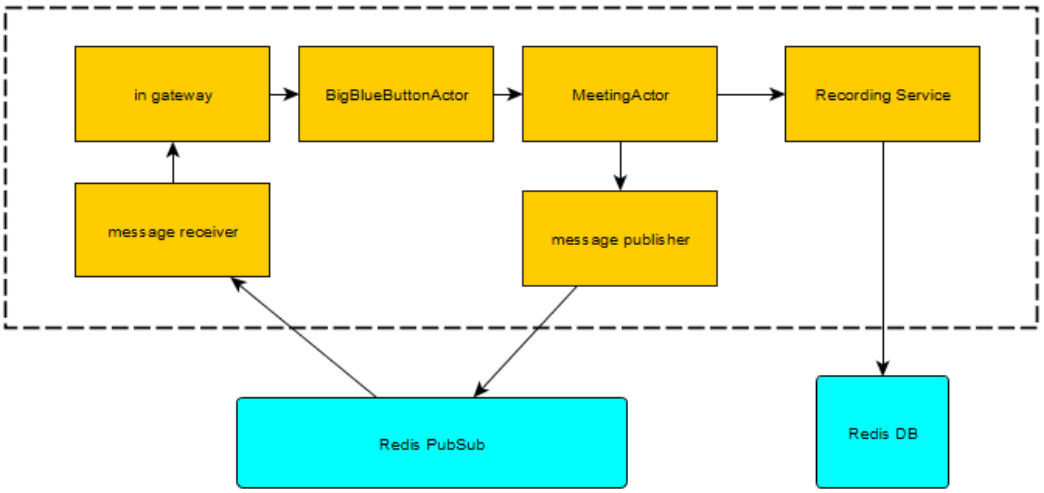


Figura 3- Apps-Akka, principali componenti interni

2.1.2 Redis PubSub

Redis PubSub è il modulo che si occupa di collegare tutti gli altri componenti di BigBlueButton, permettendo loro di scambiarsi messaggi. Può essere pensato come il bus di sistema dell'applicazione: apre un canale di comunicazione tra le varie applicazioni attive sul server BigBlueButton, ricevendo e inviando messaggi verso porte predefinite, sulle quali gli altri componenti rimangono in ascolto.

2.1.3 BigBlueButton-Web

Questo modulo, scritto in Scala, implementa le API di BigBlueButton. Non analizzeremo nel dettaglio questo componente, ci basti sapere che da qui passano le chiamate per la creazione, gestione e chiusura dei meeting, così come quelle per altre funzionalità.

2.1.4 BigBlueButton-HTML5

Questo modulo è il client dell'applicazione, ed è ovviamente il componente con il quale si interfaccia principalmente l'utente finale: una responsive single page web application che viene caricata direttamente dal server e fatta girare all'interno del browser.

Il modulo si divide in frontend e backend, indicati in figura 2 rispettivamente col nome di html5-client e html5-server.
Il modulo, nella sua interezza, è costruito tramite l'utilizzo di Meteor, un web framework open source per la creazione di applicazioni cross-platform basato su Node.js. Meteor fornisce integrazione sia con i principali framework per lo sviluppo di applicazioni frontend (quali React, Vue, Angular, ecc.) sia con MongoDB, permettendo quindi di interfacciarsi facilmente anche con una base dati No-SQL.
Nel caso di BigBlueButton il frontend è stato sviluppato tramite ReactJS.
Il backend si interfaccia con MongoDB per mantenere coerente lo stato del client con quello del server. Dal database vengono infatti recuperate le informazioni relative ai meeting esistenti e agli utenti che vi sono connessi: stato del meeting, chat pubbliche e private degli utenti, sono tutte contenute al suo interno. Ogni client ha ovviamente visibilità solamente sulla porzione di dati relativa a se stesso e al meeting di cui fa parte.
Il funzionamento del modulo è basato principalmente su una meccanica "ad eventi": ogni volta che un partecipante al meeting compie un'azione di rilievo (ad esempio apre un flusso video, 'muta' un altro utente, muove il cursore del mouse durante lo screen sharing) viene generato un evento che dal frontend viene passato al backend, e da questo al componente Apps-Akka. Apps si occupa di aggiornare il database degli eventi legati al meeting ed eventualmente di redirigere l'evento verso altri componenti interni al server e/o ad altri peer del meeting. L'invio verso altri peer avviene appoggiandosi a un server PubSub. Lato ricezione, il backend del client verrà notificato dell'arrivo di un nuovo evento e aggiornerà - dove necessario- la base dati in maniera congruente all'evento ricevuto.
Si sottolinea ancora una volta che la comunicazione con altri moduli non è un procedimento gestito interamente dal client, che deve essere visto solo come l'ultimo nodo di una catena che coinvolge altri componenti di BigBlueButton. Il client si limita a generare un messaggio e a trasmetterlo verso un altro componente del sistema, Redis PubSub: è poi quest'ultimo a occuparsi del collegamento con gli altri moduli.

Il client è stato costruito basandosi su HTML5, linguaggio di markup standard per la costruzione di pagine web. [3]
Originariamente il client non era però stato ideato su HTML5, ma bensì sull'utilizzo di Adobe Flash.
Flash è una piattaforma software per la produzione e la rappresentazione di contenuti multimediali interattivi, pensata principalmente per lo sviluppo web. [4] Inizialmente utilizzato per creare giochi o interi siti web, nel tempo è divenuto un potente strumento per la creazione di *Rich Internet Application* e piattaforme di streaming audio/video, anche in virtù della sua capacità di catturare input da mouse, tastiera, microfono e webcam.

I motivi che hanno spinto BigBlueButton a riprogettare il componente passando a una nuova tecnologia sono molteplici. [5] HTML5 è uno standard web libero utilizzabile senza l'aggiunta

di plug-in esterni, che invece Flash -di proprietà di Adobe- richiede. Flash ha presentato inoltre nel corso degli anni diverse criticità in ambito di sicurezza, che hanno spinto i principali browser a smettere progressivamente di fornirne supporto. A tutto questo si aggiunge il fatto che il software di Adobe non era utilizzabile sui dispositivi mobile, ai quali invece il progetto BigBlueButton mirava di espandersi. Tutto questo ha spinto alla migrazione verso la nuova tecnologia e alla creazione di questo nuovo componente. Il server BigBlueButton dalla versione 2.0 in poi supporta connessioni sia tramite client Flash che HTML5. [6]

2.2 Comunicazione tra i componenti

Come accennato in precedenza, i moduli di cui si compone BigBlueButton sono connessi tra di loro da un canale di comunicazione che utilizzano per scambiarsi messaggi relativi ad eventi che si sono verificati all'interno del sistema.
Redis PubSub è il principale fautore di questo meccanismo, creando il canale attraverso il quale il flusso di informazioni diventa possibile.
Analizzandolo più nel dettaglio, si vede come le comunicazioni tra i vari moduli siano realizzate attraverso l'apertura di socket su porte IP predefinite. Ad esempio, Redis PubSub è raggiungibile di default alla porta 6379 del server.
Il bus di sistema mette a disposizione una serie di canali preimpostati per raggiungere altri componenti (si conoscono infatti le porte su cui sono raggiungibili), e una serie di funzioni che permettono sia di inviare nuovi messaggi su questi canali, sia di mettersi in ascolto per intercettarne di nuovi.
Si fornisce di seguito un estratto del codice preso dal modulo *bigbluebutton-html5*, al fine di chiarire ulteriormente la meccanica.

```
export default function sendAnnotationHelper(annotation, meetingId,
requesterUserId) {
  const REDIS_CONFIG = Meteor.settings.private.redis;
  const CHANNEL = REDIS_CONFIG.channels.toAkkaApps;
  const EVENT_NAME = 'SendWhiteboardAnnotationPubMsg';

  const whiteboardId = annotation.wbId;
  //...
  const payload = {
    annotation,
  };

  return RedisPubSub.publishUserMessage(CHANNEL, EVENT_NAME, meetingId,
requesterUserId, payload);
}
```

In questa specifica situazione, il client sta inviando un evento relativo alla creazione di una nuova annotazione sulla lavagna virtuale.

Si notano tre cose:

1. L'utilizzo della funzione *publishUserMessage()* messa a disposizione da PubSub per l'invio di messaggi sul canale;
2. La comunicazione avviene su un canale predefinito (*toAkkaApps*) ricavato dalla configurazione di Redis;
3. La configurazione della comunicazione del componente con Redis è settata staticamente all'interno di un file presente all'interno del modulo stesso (settings.yml).

Prendendo un estratto di tale file si può notare come vengano dichiarati sia l'indirizzo a cui raggiungere il modulo, sia i canali messi a disposizione del componente client (inclusi quelli su cui mettersi in ascolto).

```
redis:
  host: 127.0.0.1
  port: '6379'
  timeout: 5000
  password: null
  debug: false
  channels:
    toAkkaApps: to-akka-apps-redis-channel
    toThirdParty: to-third-party-redis-channel
  subscribeTo:
  - to-html5-redis-channel
  - from-akka-apps-*
  - from-third-party-redis-channel
  - from-etherpad-redis-channel
```

Lato ricezione, viene fornita un'analoga funzione che permette di specificare la funzione da chiamare all'arrivo di un determinato evento.

```
RedisPubSub.on('SendWhiteboardAnnotationEvtMsg', handleWhiteboardSend);
```

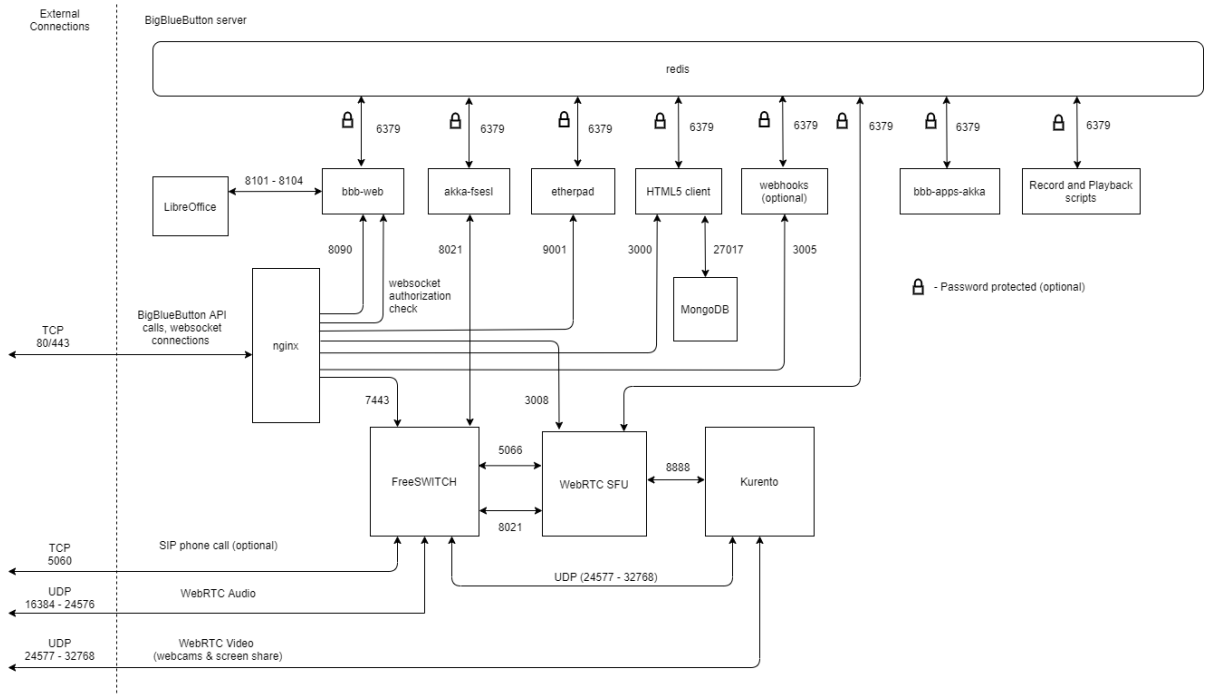


Figura 4 - Canali tra i vari componenti e porte su cui raggiungerli

2.3 Kurento KMS e WebRTC

Una menzione speciale la meritano sicuramente questi due componenti, che costituiscono le fondamenta della comunicazione real time tra end users all'interno di BigBlueButton.
WebRTC è una tecnologia open source per l'aggiunta di comunicazione in tempo reale su applicazioni web e mobile.[7] Permette di creare canali di comunicazione tra peer per il trasporto di traffico sia audio che video ed è molto usata tra i sistemi di videoconferenza, tra cui BigBlueButton stesso. Le sue funzioni sono fruibili tramite API Javascript accessibili ormai da tutti i principali browser.
Kurento Media Server (KMS) è un server multimediale basato su WebRTC, che permette di implementare sia architetture di tipo SFU (Selective Forwarding Unit) che MCU (Multipoint Conferencing Unit). [8]
Nonostante WebRTC nasca infatti come tecnologia peer to peer, col tempo si è cominciato a declinarla verso altre possibilità, che prevedevano per l'appunto l'utilizzo di un server per la ricerca dei peer e la redirezione dei flussi tra i vari client.
La differenza tra l'architettura SFU e la MCU risiede essenzialmente nel ruolo assegnato al server: in un'architettura SFU il server riceve un flusso multimediale da ogni peer e lo redirige verso ogni altro client che soddisfi la configurazione adottata; in un'architettura MCU il server riceve ancora un singolo flusso da ogni peer, ma invece di redirigere subito ciò che riceve effettua prima un mixing di tutte le tracce, per inviare poi ai client un singolo stream. [9]

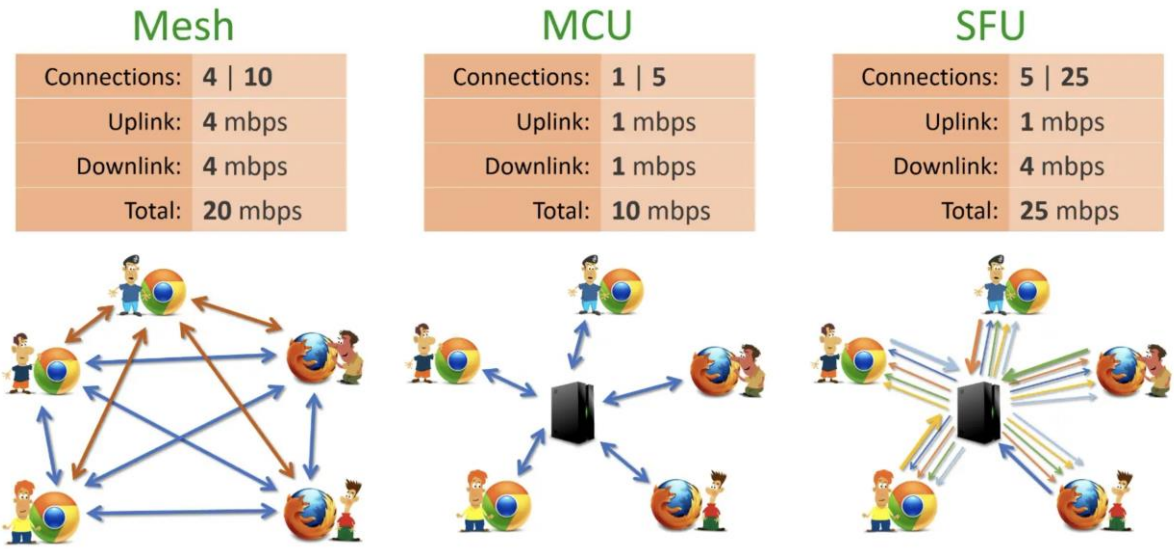


Figura 5 - Mesh vs MCU vs SFU: modelli a confronto

Come si può intuire dall'immagine, rispetto al tradizionale schema adottato dalle comunicazione peer to peer, sia l'architettura SFU che la MCU offrono vantaggi sul numero di connessioni da far aprire ai client.
BigBlueButton si appoggia su un server Kurento impostato con architettura SFU. La preferenza rispetto al MCU è dovuta probabilmente al minor carico di lavoro richiesto in questo caso al media server, che potrebbe causare rallentamenti in caso di un ingente numero di connessioni.
Kurento viene usato per gestire sia i flussi delle webcam, sia i flussi di screen sharing, sia i flussi audio-only.
BigBlueButton fa girare Kurento all'interno di uno specifico componente, bbb-webrtc-sfu, che gestisce le negoziazioni tra i client e i flussi multimediali.
All'interno del client (modulo bigbluebutton-html5) viene integrata la libreria kurento-utils.js, che fornisce le API per creare facilmente dei nuovi canali di comunicazione basati su WebRTC. [10]
Aprendo il file settings.yml del modulo client, è possibile visualizzare la configurazione adottata dal server Kurento.

```
kurento:
  wsUrl: wss://danyvbbbex1.it/bbb-webrtc-sfu
  # Valid for video-provider. Time (ms) before its WS connection
  # times out and tries to reconnect.
  wsConnectionTimeout: 4000
  cameraTimeouts:
    # Base camera timeout: used as the camera *sharing* timeout and
    # as the minimum camera subscribe reconnection timeout
```

```
    baseTimeout: 15000
    # Max timeout: used as the max camera subscribe reconnection
    # timeout. Each subscribe reattempt increases the reconnection
    # timer up to this
    maxTimeout: 60000
  screenshare:
    constraints:
      video:
        frameRate:
          ideal: 5
          max: 10
        width:
          max: 2560
        height:
          max: 1600
      audio: false
    chromeScreenshareSources:
      - window
      - screen
    firefoxScreenshareSource: window
```

3 Creazione dell'ambiente di lavoro

In questa sezione verrà descritto brevemente come creare un ambiente di sviluppo funzionante, partendo dalla creazione di un server fino ad arrivare ai comandi da lanciare per avviare i singoli componenti.

3.1 Avvio di un server BigBlueButton

La prima cosa da fare per poter lavorare al progetto BigBlueButton è avere un server funzionante a cui collegarsi.
La creazione di un server non risulta essere particolarmente complessa, essendo già presenti degli script messi a disposizione dalla comunità di sviluppatori della stessa BigBlueButton. [11]
Si specifica che la versione di BigBlueButton su cui questo progetto di tesi è stato incentrato è la 2.2.
Come ambiente di sviluppo si è scelta una macchina con sistema operativo Linux Ubuntu 16.04 -versione consigliata dagli sviluppatori di BigBlueButton- con 500GB di spazio su disco e 8 GB di RAM.
Viene consigliato di lasciare la macchina su cui è fatto girare il server quanto più "vuota" possibile, evitando di installare altre applicazioni che potrebbero andare a occupare le porte di rete utilizzate dal server e compromettere il corretto funzionamento della web application.
Prima di procedere con l'installazione del server è necessario avere a disposizione un hostname e un certificato SSL. Se, infatti, sarebbe possibile far partire il server anche utilizzando unicamente un indirizzo IP valido, non sarebbe possibile per gli utenti condividere il loro audio o video, compromettendo quindi l'utilizzo della piattaforma.
Tutti i browser richiedono infatti la verifica di un valido certificato SSL a ogni server che cerchi di accedere alla webcam o al microfono di un client, e naturalmente i certificati sono legati all'hostname del server di appartenenza.

Poiché il tema principale di questo progetto era incentrato sul lavoro con i flussi multimediali dell'applicazione, si è optato per l'acquisto di un dominio.
A tal scopo la scelta del domain provider è ricaduta su GoDaddy.com, che permette di gestire con facilità anche i servizi DNS legati al nome scelto. [12]
Per quanto riguarda il certificato SSL si è utilizzato invece Let's Encrypt, una Certificate Authority che permette di registrare certificati validi gratuitamente. [13]
Una volta ottenuto un hostname è possibile seguire la guida presente sulla pagina di BigBlueButton per l'installazione del server. Dopo aver completato alcuni passaggi preliminari, ben spiegati sulla pagina del progetto, è sufficiente -sulla macchina su cui si vuole installare il server- lanciare lo script fornito tramite il comando:

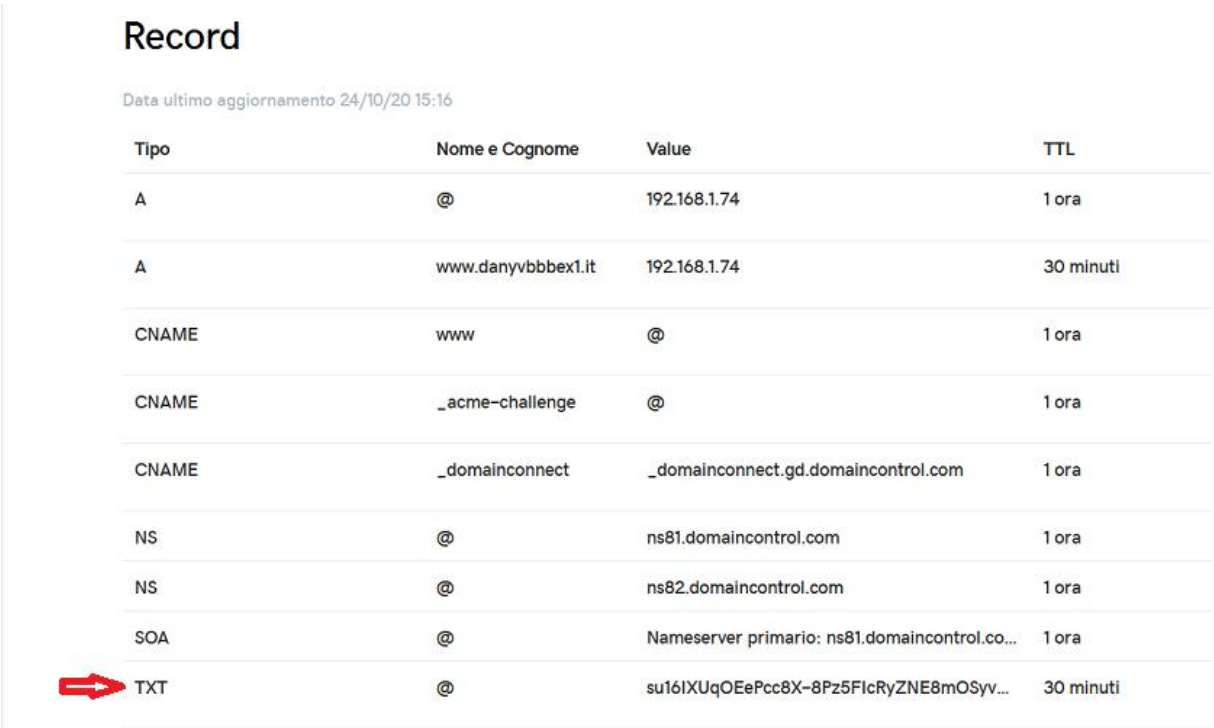
```
wget -qO- https://ubuntu.bigbluebutton.org/bbb-install.sh |
bash -s -- -w -a -v versionName -s domainName -e emailAddress
```

Il flag *-s domainName* permette di settare l'hostname del server a quello del dominio scelto; -v *versionName* consente di scegliere la versione del server da scaricare; *-e emailAddress* consente di specificare un indirizzo email da associare al certificato SSL ottenuto tramite Let's Encrypt.

Il certificato rilasciato ha una validità di 3 mesi. Superato questo tempo limite, può essere rinnovato lanciando il seguente comando:

```
certbot -d domainName --manual --preferred-challenges dns certonly
```

La generazione di un nuovo certificato passa per motivi di sicurezza da una verifica di colui che la richiede. In questo caso usando il flag *--manual* stiamo specificando che la sfida deve essere svolta manualmente. Essa consiste nell'inserimento di una stringa di testo -fornita durante l'esecuzione dello script- da associare al campo TXT del nostro dominio all'interno del servizio DNS.
Qui di seguito viene mostrata la configurazione DNS che assume il nostro hostname.



Record

Data ultimo aggiornamento 24/10/20 15:16

Tipo	Nome e Cognome	Value	TTL
A	@	192.168.1.74	1 ora
A	www.danyvbbbex1.it	192.168.1.74	30 minuti
CNAME	www	@	1 ora
CNAME	_acme-challenge	@	1 ora
CNAME	_domainconnect	_domainconnect.gd.domaincontrol.com	1 ora
NS	@	ns81.domaincontrol.com	1 ora
NS	@	ns82.domaincontrol.com	1 ora
SOA	@	Nameserver primario: ns81.domaincontrol.co...	1 ora
TXT	@	su16IXUqOEePcc8X-8Pz5FIcRyZNE8mOSyv...	30 minuti

Figura 6 - Configurazione DNS del dominio

Al termine di queste operazioni, che dureranno dai 30 ai 60 minuti, si avrà a disposizione un server BigBlueButton funzionante e dotato di certificato digitale.

3.2 Settaggio ambiente di sviluppo

Una volta che il server risulta essere operativo si può procedere con la modifica del sorgente. Come nel paragrafo precedente, anche gli step necessari per configurare gli ambienti di sviluppo sono ampiamente descritti all'interno del sito di BigBlueButton, con liste dettagliate dei comandi da eseguire per installare tutti i tool necessari. [14]

Il sorgente dell'applicazione è reso disponibile su un repository github ad accesso libero. [15] E' sufficiente eseguire un clone del progetto in locale, aprire un branch separato e lavorare sul modulo di interesse. I comandi utilizzabili per far ciò sono di seguito riportati:

```
cd /home/ubuntu/dev/
git clone repositoryLink .
cd /home/ubuntu/dev/bigbluebutton

git remote add upstream https://github.com/bigbluebutton/bigbluebutt
on.git

git fetch upstream

git checkout -b my-changes-branch upstream/v2.3.x-release
```

Fatto ciò, si avrà il sorgente della versione più recente dell'applicazione all'interno della cartella /home/ubuntu/dev/bigbluebutton.

3.3 Avviare il custom client

Come già descritto in precedenza, il sorgente dell'applicazione è composto da sottomoduli, ognuno relativo a uno specifico componente.
Alcuni moduli, una volta che vi sono state apportate delle modifiche, necessitano di essere configurati prima di poter essere correttamente eseguiti dal server.

E' il caso del modulo bigbluebutton-html5 (il client dell'applicazione), su cui verterà l'attenzione di questo paragrafo, in quanto fulcro del lavoro svolto in questa tesi.

L'unica modifica di cui necessita questo modulo è all'interno del file settings.yml, ed è volta al consentire il grabbing della webcam e dello schermo da parte del client.

Il primo passo è salvare il valore ritornato dall'esecuzione di questo comando:

```
grep "wsUrl" /usr/share/meteor/bundle/programs/server/assets/app/config/settings.yml
```

Dopo di che aprendo il file */private/config/settings.yml* all'interno della directory bigbluebutton-html5, cambiare il valore alla voce **wsUrl** al risultato dell'operazione precedente.

Fatto questo, per eseguire la propria versione del client è sufficiente lanciare i seguenti comandi da terminale:

```
sudo systemctl bbb-html5 stop
npm start
```

Il primo terminerà l'esecuzione del modulo client già presente all'interno del server, il secondo metterà in esecuzione il modulo customizzato dell'utente.

4 Flusso video a frame-rate ridotto

Ora che si è visto come creare un ambiente funzionante e che sono state analizzate le componenti principali del sistema e il loro funzionamento, si può passare alla descrizione delle funzionalità implementate all'interno di questo lavoro di tesi.
Lo scopo principale che si è cercato di raggiungere è stato quello di migliorare -per quanto possibile- le prestazioni dell'applicazione, soprattutto per quanto concerne l'utilizzo della banda di rete.
La presenza di troppi utenti all'interno del meeting può infatti costituire un problema per chi non ha risorse di rete sufficienti a garantire l'acquisizione di così tanti flussi di dati. La prima funzionalità implementata, in quest'ottica, è stata quella di un flusso video a frame-rate ridotto.

4.1 Descrizione

L'idea nasce inizialmente dalla necessità di fornire al docente un contatto più stretto con la classe virtuale con cui si sta svolgendo la lezione.
Quando si svolge una lezione online, utilizzando un sistema di videoconferenza, è raro che le videocamere degli studenti restino accese: la maggior parte delle persone preferisce rimanere "invisibile" agli occhi del professore ed è quindi restia ad avviare la webcam.
Un altro motivo che spinge gli studenti a non farsi vedere però, è anche il notevole consumo di banda che questa operazione comporta, che potrebbe compromettere il corretto ascolto della lezione in corso.
Il risultato finale è che il professore si ritrova la maggior parte delle volte a tenere una lezione davanti a una classe che di fatto gli risulta essere inesistente; non c'è la possibilità di associare un volto a una determinata voce, il rapporto studente-docente diventa molto più impersonale, e probabilmente anche tenere una lezione in queste condizioni diventa a lungo andare più tedioso per lo stesso professore.
E' così che l'idea è scaturita: dare la possibilità ai partecipanti del meeting di condividere il flusso video della propria webcam prendendo solamente dei singoli frame campionati a intervalli regolari, come se fosse un susseguirsi di foto scattate all'utente.
In questo modo il docente può avere la possibilità di avere visione della classe a cui sta facendo lezione, senza però che gli studenti debbano portarsi dietro tutti i costi che la tradizionale accensione della webcam comporta.

4.2 Implementazione

Verrà ora analizzata la concreta implementazione di questa funzionalità, mostrando i punti salienti ed estratti del codice usato.

Il modulo di BigBlueButton modificato per tale scopo è stato **bigbluebutton-html5**, ovvero il client.
Per permettere all'utente di scegliere di azionare il flusso video "a basso costo" è stata aggiunta un'opzione all'interno della lista dei comandi in dotazione a ogni utente.

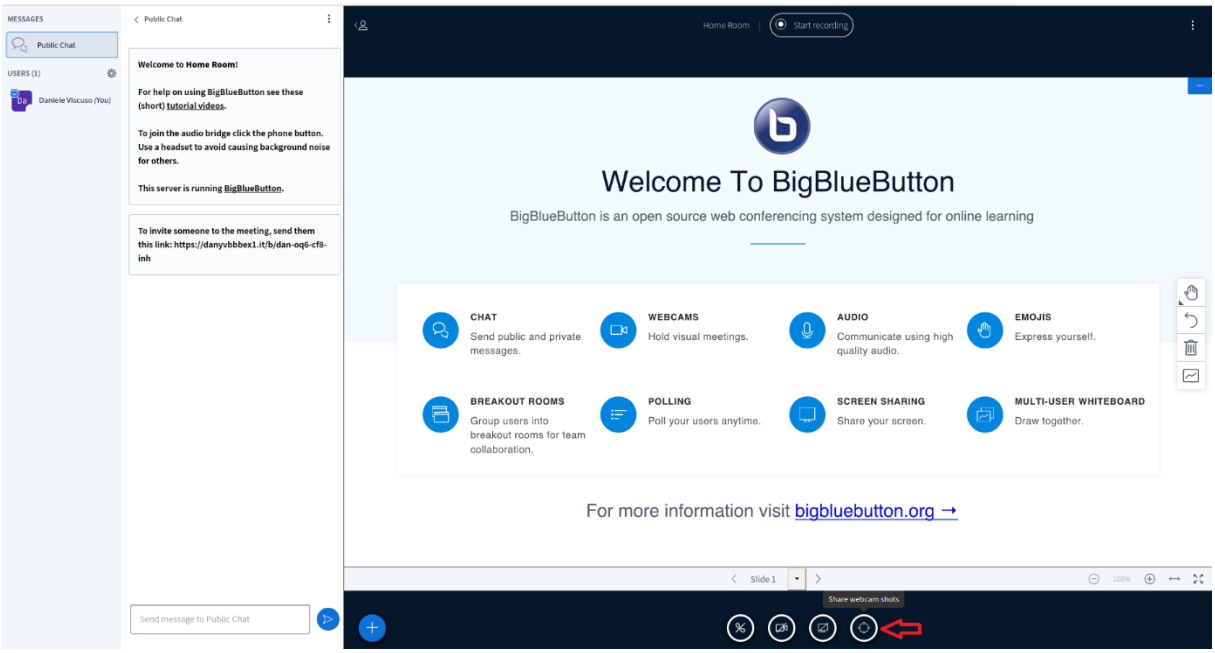


Figura 7 - Home page BigBlueButton, pulsante per flusso video a frame rate ridotto

Oltre ai classici comandi di condivisione video, schermo, e unmuting, si può notare l'aggiunta di una nuova icona, associata per l'appunto alla condivisione della webcam con frame-rate ridotto.
Passando sopra l'icona con il mouse viene mostrato un suggerimento all'utente sull'azione associata a questo nuovo componente, in modo da rendere più gradevole la user experience.
Alla pressione del pulsante ci si aggancia al flusso di codice già precedentemente implementato per l'accensione della webcam: viene mostrato un popup tramite il quale è possibile scegliere quale webcam utilizzare come sorgente video e regolare la qualità dell'immagine. Viene anche mostrata un'anteprima del video acquisito dalla camera.
In questa parte di flusso, è stato aggiunto un pezzo di codice volto essenzialmente a differenziare il tipo di condivisione webcam scelto: se tradizionale o a frame rate ridotto.

```
setCameraModeAsFrame() {
  const {
    webcamDeviceId
  } = this.props;
  Session.set(webcamDeviceId, "frame")
}
```

Questo breve frammento di codice usa l'oggetto *Session* per salvare la preferenza scelta dall'utente, associando all'id della webcam (univoco) una stringa che identifica la scelta dell'utente di voler mostrare un flusso a basso frame rate.
Session è un comodo oggetto messo a disposizione dall'API di Meteor, che permette di salvare a livello globale all'interno del client delle coppie chiave-valore di qualsiasi genere. [16]
Tramite i metodi *Session.set()* e *Session.get()* è possibile rispettivamente scrivere e leggere un valore all'interno della Session.
Così facendo, è stato possibile differenziare il tipo di flusso video generato semplicemente andando a leggere la variabile precedentemente settata all'interno della Session dentro al blocco di codice predisposto alla generazione e invio dello stream.
Di seguito viene proposto parte del codice usato per la creazione e invio del flusso video. Per semplicità di spiegazione, è stato diviso in 2 blocchi.

```
//Blocco 1
const cameraMode = Session.get(cameraIdentifier)
//…
let realCameraVideo= document.getElementById('preview')
let resultingVideo= document.getElementById('finalVideo')

navigator.mediaDevices.getUserMedia({ video: true, audio: false })
    .then(function(stream) {
      realCameraVideo.srcObject = stream;
      realCameraVideo.play();
    })

if((cameraMode && cameraMode === 'frame')){
  this.takePicture()
  let canvasStream = canvas.captureStream()
  resultingVideo.srcObject = canvasStream
}
else{
  resultingVideo.srcObject = realCameraVideo.srcObject
}
```

```
//Blocco 2
var peerOptions = {
  videoStream: resultingVideo.srcObject,
  onicecandidate: this._getOnIceCandidateCallback(cameraId, isLocal),
};
//…
if (isLocal) {
  WebRtcPeerObj = window.kurentoUtils.WebRtcPeer.WebRtcPeerSendonly;
} else {
  peerOptions = {
    onicecandidate: this._getOnIceCandidateCallback(cameraId, isLocal),
  };
  WebRtcPeerObj = window.kurentoUtils.WebRtcPeer.WebRtcPeerRecvonly;
}
```

```
this.webRtcPeers[cameraId] = new WebRtcPeerObj(peerOptions, (error) => {
  const peer = this.webRtcPeers[cameraId];
}
//…
peer.generateOffer(errorGenOffer, offerSdp);
```

Il blocco 1 mostra come viene generato lo stream. Ci si è avvalsi dell'uso di tre oggetti messi a disposizione dallo standard HTML5: un *Canvas* e due *Video*.
Il flusso realmente uscente dalla webcam viene iniettato all'interno del primo oggetto *Video* (id: 'preview'). L'acquisizione del flusso avviene attraverso la chiamata alla funzione *navigator.mediaDevices.getUserMedia({ video: true, audio: false })*, messa a disposizione dalla Web API della MDN (Mozilla Development Network) e supportata da tutti i principali browser. [17]

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet
getUserMedia	53 ★ ▼	12	36 ★ ▼	No	40 ★ ▼	11	53	53 ★ ▼	36 ★ ▼	41 ★ ▼	11	6.0
Secure context required	53	79	68	No	40	11	53	53	68	41	11	6.0

Figura 8 - Compatibilità tra getUserMedia e i moderni browser

Tra le altre cose, questa funzione permette di specificare -alla sua chiamata- quali flussi catturare tra audio e video. In questo caso, è necessario il solo video.
Sulla base del valore trovato all'interno della *Session*, usando come chiave l'id della webcam dell'utente, avviene quindi la differenziazione tra flusso video classico e ridotto.
In entrambi i casi, il risultato da inviare agli altri client viene iniettato all'interno del secondo oggetto *Video* (id: "finalVideo"), che viene usato per parametrizzare l'oggetto *WebRtcPeerObj (della classe WebRtcPeer).*
Il blocco 2 mostra l'utilizzo della libreria kurento-utils per la creazione di un canale su cui inviare lo stream risultante.
WebRtcPeer è una classe messa a disposizione da tale libreria, che funge da wrapper per l'entità peer e offre un set di metodi per la creazione del canale di comunicazione e l'invio di dati al suo interno.
Dopo aver parametrizzato l'oggetto, ne viene chiamato il metodo *generateOffer()* che avvia la negoziazione col server Kurento. [18]

La creazione del flusso a basso frame-rate è gestita dalla funzione *takePicture(),* qui di seguito analizzata.

```
takePicture(){
  let video= document.getElementById('preview')
  let context = canvas.getContext('2d');
  let width = 480;
  let height = 320;
  canvas.width = width;
  canvas.height = height;
  context.drawImage(video, 0, 0, width, height);

  let self = this;
  setTimeout(function() {
    self.takePicture();
  }, 10000);
}
```

Si tratta di una semplice funzione ricorsiva, che -finchè il flusso video è attivo- richiama se stessa ogni 10 secondi. La funzione che svolge è quella di prendere l'oggetto *Video* su cui viene riprodotto il reale stream di dati emesso dalla webcam, estrarne un frame, e disegnarlo all'interno dell'oggetto *Canvas*.
Questo è possibile tramite il metodo *drawImage()* messo a disposizione dalla DMA WebAPI, che preso un oggetto video come parametro, è in grado di estrapolarne un singolo frame per poi riprodurlo sul *Canvas* stesso. [19]
In questo modo il canvas diventa a tutti gli effetti un collettore di screenshots scattati all'utente.
La stessa API mette poi a disposizione il metodo *captureStream()* utilizzato all'interno del Blocco 1. Quest'ultimo metodo consente di generare un flusso video creato a partire dal contenuto di un oggetto *Canvas*. In questo modo si riesce a creare uno stream video contenente solo ed esclusivamente gli screenshots scattati all'utente, a intervalli molto inferiori rispetto alla norma.

Si fa notare che la funzione *getUserMedia()* (utilizzata per catturare il flusso video della webcam) permette di specificare tramite i suoi parametri dei vincoli da applicare all'acquisizione dello stream. Tra questi rientra il frame rate, del quale è possibile specificarne sia un valore massimo che uno minimo. Da prove sperimentali emerge però che browser differenti adottano comportamenti differenti al raggiungimento di determinati valori critici: nel momento in cui si imposta un valore per il frame rate che non riesce a essere supportato dalla webcam, a seconda del browser utilizzato si verificano comportamenti diversi. Se Chrome riesce infatti a modificare internamente lo stream fino a ottenere il risultato desiderato, Firefox lancia invece un'eccezione (di tipo *OverConstrainedError*) che comporta il blocco a monte dell'acquisizione dello stream dalla webcam.
Per ovviare a questo inconveniente, che minerebbe alla base il procedimento di generazione del flusso video, si è deciso quindi di adottare la soluzione precedentemente illustrata.

4.3 Prestazioni

Ora che si è analizzata nel dettaglio questa prima funzionalità introdotta e si è descritto come la si è implementata, è tempo di analizzare l'impatto sulle prestazioni effettivamente apportato.
Per monitorare l'utilizzo della banda si è utilizzato il tool Nload [20], un semplice strumento a linea di comando disponibile su Linux.
Nload permette di visualizzare in real-time l'effettivo utilizzo della banda di rete su ogni interfaccia, mostrando il traffico sia in ingresso che in uscita.

Analizzando il traffico in uscita, durante una normale riproduzione del flusso video, i risultati ottenuti sono i seguenti.

Figura 9 - Traffico misurato da Nload per flusso video tradizionale, screenshot da terminale

Si noti come la larghezza di banda mediamente utilizzata sia di circa 300 kb/s, una stima presa su un intervallo temporale di circa 60 secondi.
Eseguendo la stessa misurazione sul flusso a frame-rate ridotto invece, questo è quello che si ottiene.

Figura 10 - Traffico misurato da Nload per flusso video a frame-rate ridotto, screenshot da terminale

In questo caso la larghezza di banda occupata risulta essere di circa 90 kb/s, più di 3 volte inferiore rispetto al precedente caso.

Si può concludere quindi l'analisi di questa prima funzionalità affermando che l'implementazione permette effettivamente di ridurre la quantità di banda utilizzata dagli utenti, garantendo loro di riuscire ad attivare la webcam senza il consueto consumo in termini di traffico di rete che questo normalmente comporta.

5 Rimozione degli stream

Questa sezione è dedicata all'analisi della seconda funzionalità implementata all'interno di questo progetto, ovvero la possibilità di rimuovere i flussi video degli altri partecipanti al meeting dalla propria user interface.

Come nel caso precedente si procederà dapprima descrivendo il tipo di lavoro svolto e la relativa implementazione e in seguito analizzando gli effettivi impatti della soluzione sul sistema in uso.

5.1 Descrizione

L'implementazione dei flussi video a frame rate ridotto, analizzata nel capitolo precedente, si poneva come obiettivo non solo quello di fornire a studenti e docenti un' esperienza di lezione virtuale meno alienante, ma anche quello di ridurre gli impatti sulla rete domestica causati dalla condivisione della webcam.

Una volta realizzata questa prima soluzione ci si è chiesti: quale può essere il passo successivo? E' possibile trovare una via alternativa per ottenere risultati altrettanto validi? In che altro modo si può ottimizzare l'utilizzo della rete da parte dell'applicazione per fornire agli utenti un'esperienza migliore?

La risposta a cui si è arrivati è stata la seguente: dare la possibilità ai vari client di scegliere loro stessi quali webcam visualizzare tra quelle in condivisione.
Nel momento in cui il meeting raggiunge dimensioni considerevoli, con 20 o anche più persone che contemporaneamente inviano il proprio flusso webcam, il carico sulla rete degli utenti riceventi diventa gravoso.

Si ricordi infatti che BigBlueButton lavora tramite un'architettura di tipo SFU, in cui ogni stream video viene rediretto a tutti gli altri utenti nella conference room dal server centrale.
Questo vuol dire che se l'utente A è in un meeting con 10 persone, e tutte loro hanno la webcam attiva, A riceverà e dovrà gestire 10 stream diversi. Questo ovviamente comporta un carico prestazionale considerevole non solo sulla banda di rete di A, ma anche sulle prestazioni del client stesso.
L'idea quindi è dare la possibilità ad A di scegliere quali flussi effettivamente gestire.
Supponiamo di essere nel mezzo di una lezione e che tutti gli studenti abbiano avviato la condivisione della webcam. Dal punto di vista di A, ai fini della lezione, non serve che vengano effettivamente visualizzati i video degli altri studenti, quanto piuttosto quello del professore che sta spiegando.

L'utente A può così scegliere, in caso di rallentamenti del sistema dovuti a un sovraccarico della rete, di rimuovere gli stream degli altri studenti, mantenendo solo quello del docente.
Questo non vuol dire semplicemente non visualizzare i flussi su schermo, ma bensì rifiutarsi di riceverli, andandoli quindi ad eliminare dagli stream ad esso inviati.
In caso di ripensamenti, viene data ad A anche la possibilità di ripristinare i flussi eliminati, tornando alle condizioni di partenza.

5.2 Implementazione

Anche in questo caso, il modulo di BigBlueButton modificato per l'implementazione della nuova funzionalità è bigbluebutton-html5. Si è già visto nel precedente capitolo infatti, che è proprio questo componente a creare gli oggetti *WebRtcPeer* per ricevere e inviare nuovi flussi di dati.

Le modifiche sull'interfaccia grafica preesistente sono state ridotte al minimo, per far sì che l'utente non percepisse differenze a livello di user experience con l'applicazione.
Ci si è limitati all'aggiunta di un'opzione all'interno del menù associato a ogni utente presente all'interno della lista dei partecipanti al meeting.

Per semplicità di spiegazione l'analisi di questa implementazione verrà divisa in due parti: gestione della voce nel menù e gestione della comunicazione WebRTC.

5.2.1 Gestione dell'opzione nel menù utenti

La lista degli utenti presente sul lato sinistro della schermata principale di BigBlueButton permette, cliccando sul nome di un utente, di aprire un menù di opzioni applicabili alla persona scelta. Alcune delle opzioni disponibili sono ad esempio l'inizio di una chat privata, o l'aggiunta di privilegi all'interno della chat room (opzione, quest'ultima, accessibile solo ai moderatori del meeting).
All'interno di questo menù è stata aggiunta la voce 'Remove user stream', associata alla nuova funzionalità.

A livello di codice, per l'aggiunta dell'opzione, si è agito all'interno del componente *UserDropdown* (user-dropdown/component.jsx).

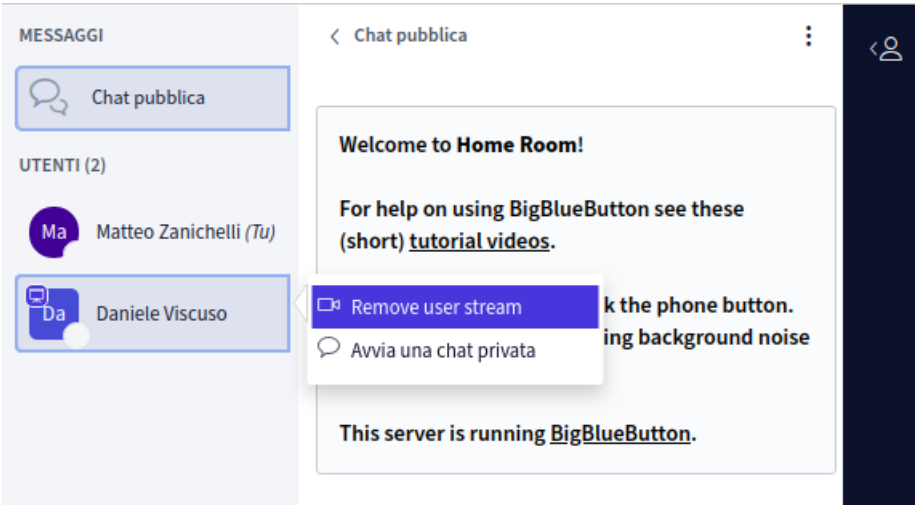


Figura 11 - Menù utenti: opzione di rimozione video; screenshot dal browser

```
const { showNestedOptions, removedStreams, shownStreams } = this.state;
const actions = [];
//…
let streamIsRemoved = false
let streamIsShown = false

removedStreams.forEach(s => {
  if(s.toString().startsWith(user.userId.toString())){
    streamIsRemoved = true
  }
})

if(!streamIsRemoved){
  shownStreams.forEach(s => {
    if(s.toString().startsWith(user.userId.toString())){
      streamIsShown = true
    }
  })
}
//...
if(streamIsShown || streamIsRemoved){
  actions.push(this.makeDropdownItem(
      'removeStreamBtn',
      (!streamIsRemoved && !streamIsShown) ? "Remove user stream" :
streamIsRemoved ? "Show user stream" : "Remove user stream",
      () => this.handleStream(),
      (!streamIsRemoved && !streamIsShown) ? 'video' : streamIsRemoved ?
'video_off' : 'video',
  ));
}
```

Siamo all'interno della funzione *getUserActions()* del componente, che gestisce il caricamento delle opzioni disponibili per uno specifico utente nella lista dei partecipanti al meeting.
Per ogni voce da aggiungere al menù viene creato un oggetto -tramite la funzione *makeDropdownItem()*- che viene poi aggiunto nell'array *actions[]*.

Si può vedere come l'aggiunta della nuova voce nel menù sia condizionata da due booleani: *streamIsShown* e *streamIsRemoved*. Tali variabili indicano la presenza di uno stream video (generato dall'utente in questione), rispettivamente all'interno della lista degli stream visibili o di quelli già rimossi. Questo sta a significare che se entrambe le variabili sono false allora l'utente non sta inviando alcun flusso proveniente dalla webcam. In tal caso, e solo in tal caso, l'opzione non viene inserita.
In caso contrario viene invece aggiunto il nuovo elemento all'interno del menù, con icona e testo cambiati dinamicamente a seconda che il video risulti visibile o rimosso.

Tale gestione dello 'stato' di un flusso webcam avviene tramite l'utilizzo dei concetti di *state* e *CustomEvent.*

Lo state è un concetto introdotto da React, che rappresenta essenzialmente lo stato di un componente. [21]

Ogni componente di un'applicazione React può dichiarare uno state composto da coppie chiave-valore, che può modificare all'occorrenza attraverso un'apposita funzione. Ogni volta che lo state di un componente cambia, il componente viene ricreato, mantenendo all'interno dello state l'ultimo valore ad esso associato.

In questo specifico caso all'interno dello state del componente UserDropdown sono state aggiunte due nuove chiavi: 'removedStreams' e 'shownStreams'. Ad esse sono associati come valori degli array -inizialmente vuoti- che conterranno i seguenti dati:
- shownStreams conterrà gli id dei flussi webcam ricevuti dall'utente e visibili;
- removedStreams conterrà gli id dei flussi webcam presenti nel meeting ma rimossi dall'utente.

Per determinare se un utente stia generando un flusso video e se questo sia visibile o meno viene quindi semplicemente eseguito un ciclo for all'interno di tali valori dello state del componente, cercando uno stream con id associato allo userId dell'utente in questione.

Si noti infatti che gli id relativi agli stream sono delle stringhe che cominciano con lo userId che identifica l'utente nel meeting.

userId	w_6nbn1y3ucbwz_t
steamId	w_6nbn1y3ucbwz_t/QmbdkEIgrZP9gYtZWQ0kdK8f5LGuZkWnbqTciP1w=

Figura 12 - Esempio di userId e streamId associato

Come viene però aggiornato lo state? Qui entra in gioco il concetto di *CustomEvent*. La Web API di Mozilla mette a disposizione un altro costrutto particolarmente utile, ossia il *CustomEvent*, un'interfaccia che permette di rappresentare un generico evento interno all'applicazione. [22]

Tali *CustomEvent* possono essere 'inviati' e 'ricevuti' da oggetti che implementano l'interfaccia *EventTarget*, un'interfaccia DOM implementata da classi piuttosto comuni nella programmazione web, come *Element* o *Window*. [23]
Un *EventTarget* ha a disposizione il metodo *dispatchEvent()* per inviare dei *CustomEvent* [24] e i metodi *addEventListener()* / *removeEventListener()* per registrarsi e rimuoversi dalla ricezione di un determinato evento. [25][26]

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet
CustomEvent	15	12	6	9	11	5.1	37	18	6	11	6	1.0
CustomEvent() constructor	15	12	11	No	11.6	6.1	37	18	14	12	6.1	1.0
detail	15	12	11	No	11.6	6.1	37	Yes	14	Yes	6.1	Yes
initCustomEvent	Yes	12	6	9	11	5.1	Yes	Yes	6	Yes	Yes	Yes
Available in workers	Yes	12	48	Yes	Yes	Yes	Yes	Yes	48	Yes	Yes	Yes

Figura 13 – Compatibilità tra CustomEvent e i moderni browser

L'uso di questi costrutti è stato utilizzato come base per mantenere aggiornati gli state sopra indicati.
Abbiamo già visto, nel precedente capitolo, che ogni volta che un utente accende la webcam viene generato un nuovo oggetto *WebRtcPeer* all'interno della classe *VideoProvider* (video-provider/component.jsx).

Scendendo più nel dettaglio si vede come il componente *VideoProvider* mantiene un elenco degli id relativi ai flussi webcam attivi, all'interno della props *streams*. Questa props viene aggiornata tramite la ricezione di messaggi che arrivano dal componente Apps-Akka, da cui passano tutti gli eventi relativi all'apertura e chiusura di ogni flusso dati del meeting, sia video che audio. Aggiornando la variabile *streams* si attiva il processo di creazione degli oggetti

WebRtcPeer per l'instaurazione dei canali di comunicazione; questo avviene basandosi sugli id presenti all'interno della props, che identificano univocamente lo stream all'interno dell'intero meeting, e permettono quindi di ritrovarlo dal server Kurento.

Si è pensato di aggiungere e inviare, nel punto in cui viene creato il peer WebRTC, un evento che identifica l'aggiunta del nuovo flusso webcam tra quelli visibili, in modo da tenere aggiornati gli state dei componenti.
Il seguente frammento di codice descrive meglio quanto detto finora.

```
connectStreams(streamsToConnect) {
  streamsToConnect.filter(cId =>
!this.state.hiddenStreams.includes(cId)).forEach((cameraId) => {
    const isLocal = VideoService.isLocalStream(cameraId);
    this.createWebRTCPeer(cameraId, isLocal);

    const handleStreamEvent = new CustomEvent('streamShown', { detail: {
mediaElement: cameraId } });
    window.dispatchEvent(handleStreamEvent);
  });
}
```

Questa funzione viene chiamato ogni volta che la props *streams* del componente *VideoProvider* viene aggiornata.

Partendo dagli id degli stream attualmente presenti (memorizzati in *streamsToConnect*) per ognuno di loro viene:
1- generata una comunicazione WebRTC tramite la funzione *createWebRTCPeer()*;
2- inviato un evento identificato come '*streamShown'*, che memorizza al suo interno l'id del nuovo flusso.

Questo evento viene fatto catturare dal componente *UserDropdown*, che conseguentemente aggiornerà il suo state e mostrerà l'opzione per la rimozione del flusso video nel menù dell'utente con userId contenuto nell'id dello stream.

Di seguito viene mostrato il codice, inserito in *UserDropdown*, che esegue tali azioni.

```
constructor(props) {
  super(props);
  this.handleStreamRemoved = throttle(this.handleStreamRemoved).bind(this);
  this.handleStreamShown = throttle(this.handleStreamShown).bind(this);
}

componentDidMount() {
  window.addEventListener('streamRemoved', this.handleStreamRemoved);
  window.addEventListener('streamShown', this.handleStreamShown);
}
```

```
componentWillUnmount() {
  window.removeEventListener('resize', this.handleStreamRemoved, false);
  window.removeEventListener('resize', this.handleStreamShown, false);
}

handleStreamRemoved(e) {
  const { mediaElement } = e.detail;
  const { removedStreams, shownStreams } = this.state

  e.stopPropagation();

  let sStreams = shownStreams
  let index = shownStreams.indexOf(mediaElement);
  if (index > -1) {
    sStreams.splice(index, 1);
  }

  let rStreams = removedStreams
  let index2 = removedStreams.indexOf(mediaElement)
  if (index2 > -1) {
    rStreams.splice(index2, 1);
  }

  this.setState({
    removedStreams: rStreams,
    shownStreams: sStreams
  })
}

handleStreamShown(e){
  const { mediaElement } = e.detail;
  const { removedStreams, shownStreams } = this.state

  e.stopPropagation();

  let sStreams = shownStreams
  sStreams.push(mediaElement)

  let rStreams = removedStreams
  let index = rStreams.indexOf(mediaElement)
  if (index > -1) {
    rStreams.splice(index, 1);
  }

  this.setState({
    removedStreams: rStreams,
    shownStreams: sStreams
  })
}
```

Si può vedere come il componente si registri all'evento '*streamShown'* subito dopo la fase di mount e rimuova il listener prima della fase di unmount. Alla ricezione dell'evento viene chiamata la funzione *handleStreamShown()* che:
1- estrae lo streamId dall'evento;
2- aggiunge lo streamId all'interno dello state *shownStreams*;
3- rimuove lo streamId, se presente, dallo state *removedStreams*.

La funzione *handleRemovedStream()* è la controparte di quanto visto sopra. Viene chiamata a seguito della ricezione dell'evento 'streamRemoved', che identifica la chiusura di una connessione WebRTC. L'evento viene quindi generato dal componente *VideoProvider* quando l'utente remoto smette di condividere la webcam.

Questo termina l'analisi della gestione del menù utenti. I concetti di *state* e *CustomEvents* saranno alla base anche del prossimo paragrafo, incentrato sulla gestione interna dei flussi video ricevuti.

5.2.1 Gestione interna dei flussi video

La rimozione di un flusso video dall'insieme di quelli ricevuti ha inizio nel momento in cui l'utente clicca sull'opzione 'Remove user stream' descritta nel paragrafo precedente.

In tal caso viene invocata la seguente funzione all'interno del componente UserDropdown:

```
handleStream(){
  const { removedStreams, shownStreams } = this.state;
  const { user } = this.props;

  let finalStream = user.userId

  let rStreams = removedStreams
  let sStreams = shownStreams
  let foundInRemoved = false

  rStreams.forEach(s => {
    if(s.toString().startsWith(user.userId.toString())){
      let index = rStreams.indexOf(s)
      rStreams.splice(index,1)
      sStreams.push(s)
      foundInRemoved= true
    }
  })
```

```
  if(!foundInRemoved){
    let foundInShown = false
    sStreams.forEach(s => {
      if(s.toString().startsWith(user.userId.toString())){
        foundInShown = true
        let index = sStreams.indexOf(s)
        sStreams.splice(index,1)
        finalStream = s
        rStreams.push(s)
      }
    })
  }

  this.setState({
    removedStreams: rStreams,
    shownStreams: sStreams
  }, () =>{
    if(foundInRemoved){
      const restartStreamEvent = new CustomEvent('restartStream', { detail: {
mediaElement: finalStream } });
      window.dispatchEvent(restartStreamEvent);
    } else {
      const blockStreamEvent = new CustomEvent('blockStream', { detail: {
mediaElement: finalStream } });
      window.dispatchEvent(blockStreamEvent);
    }
  })
}
```

Come si può osservare la funzione recupera inizialmente lo userId dell'utente dalle props e i valori di *removedStreams* e *shownStreams* dallo state (analogamente al caso precedente).

Lo userId viene inizialmente usato per cercare un flusso associato allo stesso utente all'interno della lista degli stream nascosti (*removedStreams*). Se c'è un riscontro -il che significa che il video era nascosto- l'id viene spostato da *removedStreams* a *shownStreams*, per annotare che il video deve tornare a essere visibile.

Se questa prima ricerca non porta a nulla, ne viene avviata una seconda all'interno di *shownStreams*. Anche qui in caso di riscontro positivo -indicante che il flusso video era tra quelli visibili- l'id viene spostato nell'array opposto.

A ogni chiamata della funzione lo streamId viene quindi spostato tra removedStreams e shownStreams: l'array destinazione indicherà come gestire lo stream associato.

Una volta fatto ciò, a seconda del risultato, vengono generati due eventi distinti:
1- Se il flusso risulta marcato come 'da rimuovere', viene generato l'evento '*blockStream*';

2- Se invece lo stream deve tornare a essere visibile viene generato l'evento '*restartStream'*. In entrambi i casi, l'evento generato conterrà al suo interno lo streamId del flusso in questione.

Evento	*Descrizione*
streamShown	Notifica l'inizio di un flusso video da parte di un altro utente
streamRemoved	Notifica la terminazione di un flusso video da parte di un altro utente
blockStream	Segnala la volontà di non ricevere lo stream video di un altro utente
restartStrem	Segnala la volontà di voler riprendere lo stream video di un altro utente

Figura 14- Lista degli eventi per la gestione interna degli stream , con descrizione

I due *CustomEvent* sono stati fatti intercettare dal componente *VideoProvider*, aggiungendo al suo interno dei listener in maniera analoga a quanto visto in precedenza. Le funzioni chiamate alla ricezione dei due eventi sono rispettivamente *blockStream()* e *restartStream().* Verrà analizzata solo *blockStream(),* in quanto la sua controparte si comporta in maniera analoga.

```
blockStream(e){
  const { mediaElement } = e.detail;
  const { streams } = this.props

  e.stopPropagation();

  let correspondingStream = null
  const streamsCameraId = streams.map(s=> s.cameraId)
  streamsCameraId.forEach( ss => {
    if(ss.toString().startsWith(mediaElement.toString())){
      correspondingStream = ss
    }
  })

  if(correspondingStream != null){
    this.handleReceivedStream(correspondingStream, false)
  }
}
```

Come già visto in precedenza, il componente *VideoProvider* mantiene all'interno delle sue props la variabile *streams*, contenente l'elenco dei flussi video attivi all'interno del meeting. La funzione creata si limita a cercare al suo interno un flusso con id combaciante con quello incapsulato all'interno dell'evento. Se c'è un riscontro, viene chiamata la funzione

handleReceivedStream(), che accetta come parametri uno streamId e un booleano indicante lo stato a cui si vuole settare il flusso (visibile o nascosto).

```
handleReceivedStream(cameraId, enabled){
  const {hiddenStreams} = this.state

  if(!enabled){
    this.stopWebRTCPeer(cameraId, false, false)
    delete this.videoTags[cameraId]

    let currentHiddenStreams = hiddenStreams
    if(!currentHiddenStreams.includes(cameraId)){
      currentHiddenStreams.push(cameraId)
    }
    this.setState({ hiddenStreams: currentHiddenStreams });
  } else{
    this.createWebRTCPeer(cameraId, false).then(
        (value => {
          let currentHiddenStreams = hiddenStreams
          let index = currentHiddenStreams.indexOf(cameraId)
          currentHiddenStreams.splice(index,1)
          this.setState({ hiddenStreams: currentHiddenStreams });
        })
    )
  }
}
```

Se il video deve essere fatto ripartire viene chiamata la funzione *createWebRTCPeer(),* già vista nel capitolo 4.2. Essa crea l'oggetto *WebRtcPeer* e ne chiama il metodo *generateOffer()* che comincia la creazione del canale di comunicazione col server Kurento.

Se invece il video deve essere interrotto si invoca la funzione *stopWebRTCPeer().* In sintesi, questa funzione recupera il peer object associato a quello streamId e ne chiama il metodo *dispose():* esso libera le risorse utilizzate dall'oggetto chiudendo il canale di comunicazione ad esso associato. [27]

In questo modo viene gestita l'apertura e chiusura dei collegamenti con Kurento. Chiamando *dispose()* il client dichiara di non essere più interessato a ricevere dati dal canale associato a quello specifico streamId, e la comunicazione si interrompe. Per essere nuovamente raggiunto dallo stream è sufficiente ricreare un oggetto *WebRtcPeer* che fa richiesta a Kurento per lo stesso streamId. Essendo gli streamId disponibili memorizzati all'interno delle props di *VideoProvider*, l'informazione non viene mai persa fino a che l'utente remoto non termina effettivamente la condivisione della webcam.

Per terminare la discussione di questa implementazione si sottolinea infine un ultimo dettaglio: all'interno del componente *VideoProvider* è stata aggiunta una nuova chiave -*hiddenStreams*- all'interno dello state.

Come nel caso precedente, ad essa è associato un array contenente gli streamId dei flussi nascosti. Questa variabile viene usata per due motivi:

1. come si è visto in precedenza ogni volta che un nuovo utente accende la webcam viene aggiornata la props *streams* del componente, e questo comporta la creazione di un peer per aprire il canale *WebRtcPeer* verso quell'utente. Lo state viene usato per non creare oggetti peer verso utenti già rimossi;
2. la logica di gestione dell'interfaccia grafica prevede che venga creato un riquadro (in cui mostrare la webcam dell'utente) per ogni valore presente nella props *streams*. Anche in questo caso, lo state permette di rimuovere i riquadri sprovvisti di flusso.

5.3 Prestazioni

Rimane da analizzare, per la funzionalità descritta in questo capitolo, l'impatto ottenuto sulla banda di rete del client.
Anche in questo caso è stato utilizzato il tool Nload per misurare il traffico in ingresso sulla scheda di rete dell'utente.

L'analisi si è svolta sotto le seguenti condizioni:

- meeting con due persone all'interno (utente A ed utente B);
- B con webcam attiva, A con webcam spenta;
- misurazioni effettuate sulla scheda di rete di A, su un arco temporale di 60 secondi.

Le misurazioni effettuate sono state due: nella prima il flusso video di B è stato mantenuto visibile, nella seconda è stato rimosso dalla schermata dell'utente A.

Analizzando il primo scenario (stream della webcam visibile all'utente A) i risultati ottenuti sono stati i seguenti.

Figura 15 - Traffico misurato da Nload con stream di B visibile, screenshot da terminale

Come si può osservare i risultati sono molto simili a quelli analizzati nel capitolo 4.3 per la condivisione di un flusso video tradizionale, con una media che si aggira intorno ai 290 kb/s.

Andando a rimuovere lo stream dall'elenco dei flussi ricevuti da A la misurazione dà invece i seguenti risultati.

Figura 16 - Traffico misurato da Nload con stream di B rimosso, screenshot da terminale

I dati dimostrano che la banda occupata cala sensibilmente nel secondo caso, attestandosi su una media di poco superiore ai 20 kb/s.
Il valore raggiunto è al livello del traffico dati normalmente registrato da BigBlueButton in assenza di flussi webcam in ingresso, dimostrando quindi l'effettiva chiusura della connessione WebRTC e dell'invio di dati verso il client A.

6 Stream webcam in finestra pop-up

In questa sezione verrà analizzata l'ultima funzionalità implementata in questo lavoro di tesi, che verte sulla creazione di una finestra pop-up separata da quella principale dell'applicazione, in cui far confluire i flussi webcam attivi all'interno del meeting.
Come sempre verrà dapprima data una panoramica sulla nuova aggiunta, per poi scendere nel dettaglio della sua implementazione pratica.

6.1 Descrizione

Tutte le implementazioni viste fino a questo momento erano volte principalmente al miglioramento delle prestazioni dell'applicazione, con particolare enfasi sulla riduzione dell'utilizzo della banda di rete da parte del client.
Quest'ultimo lavoro invece nasce per soddisfare un'esigenza diversa, ovvero quella di rendere accessibile la videoconferenza a coloro che non vi possono partecipare. Si immagini il seguente scenario: viene creata una classroom per lo svolgimento di una lezione universitaria, a cui gli studenti iscritti al corso hanno accesso libero. Essendo una lezione molto importante si decide di renderla fruibile in tempo reale anche a persone non iscritte al corso in questione, in modo che tutti quelli interessati all'argomento possano trarne beneficio. La soluzione semplice per realizzare quanto descritto sopra sarebbe quella di rendere pubblico il link alla videoconferenza, in modo che chiunque voglia possa accedervi con un semplice click. Questo però potrebbe portare a dei risvolti negativi: se il numero di persone collegate alla classroom fosse eccessivamente elevato le prestazioni del server BigBlueButton potrebbero risentirne, andando a inficiare la qualità della lezione stessa. Allo stesso tempo, potendo tutti i partecipanti attivare il proprio microfono, si rischierebbe di creare una situazione caotica in cui la voce del professore è sovrastata dal rumore.
Una possibile soluzione al problema potrebbe essere quella di rendere disponibile la videoconferenza sotto forma di stream, e dare quindi la possibilità di accedervi su un link a parte. Agendo in questo modo gli utenti "esterni" alla classe avrebbero sì modo di seguire la lezione in tempo reale, ma al tempo stesso potrebbero solo parteciparvi passivamente, come se fossero spettatori di una diretta.

Al giorno d'oggi sono disponibili vari software che permettono di gestire una diretta streaming.
Uno dei più usati è ad esempio OBS Studio, un software open source disponibile per tutti i principali sistemi operativi, che permette di effettuare operazioni di video-recording su finestre o porzioni dello schermo del PC per trasmetterle poi in streaming accessibili tramite URL.

Catturando con OBS il meeting creato tramite BigBlueButton e rendendolo disponibile sotto forma di streaming il problema viene così a risolversi. E' però a volte necessario filtrare i contenuti resi disponibili all'esterno della classroom: all'interno del meeting potrebbe essere attiva non solo la webcam del professore, ma anche quelle di alcuni studenti. Catturare l'intera pagina del client BigBlueButton sarebbe di scarsa utilità, o potrebbe anche causare fastidio agli studenti, che magari non gradiscono che il loro volto venga reso disponibile in una diretta streaming.

Per questo motivo si è pensato di implementare una soluzione che permette di spostare i flussi webcam degli utenti su una finestra separata: nella finestra secondaria potranno essere inseriti i soli flussi di reale interesse -come quello del professore- e la finestra potrà poi essere usata come sorgente dello streaming da OBS. In questo modo verranno resi disponibili agli spettatori esterni alla classroom solo i contenuti essenziali.

La soluzione attualmente implementata permette di inserire nel pop-up solo i flussi delle webcam degli utenti. Lo streaming della nuova finestra non è attualmente implementato. Una possibile espansione, conservata per lavori di tesi futuri, potrebbe essere quella di inserire altri elementi all'interno della nuova finestra, come le slide della lezione o la lavagna virtuale.

6.2 Implementazione

Per abilitare l'aggiunta dei video all'interno della finestra pop-up si è scelto di aggiungere un pulsante nell'angolo in alto a sinistra dei riquadri video stessi: alla pressione del pulsante lo stream mostrato all'interno del riquadro viene inserito o rimosso dalla finestra pop-up. La creazione e rimozione della suddetta finestra avviene dinamicamente: viene creata all'inserimento del primo flusso video e distrutta alla rimozione dell'ultimo.

Figura 17 - Button per inserimento e rimozione del riquadro webcam nella finestra pop-up

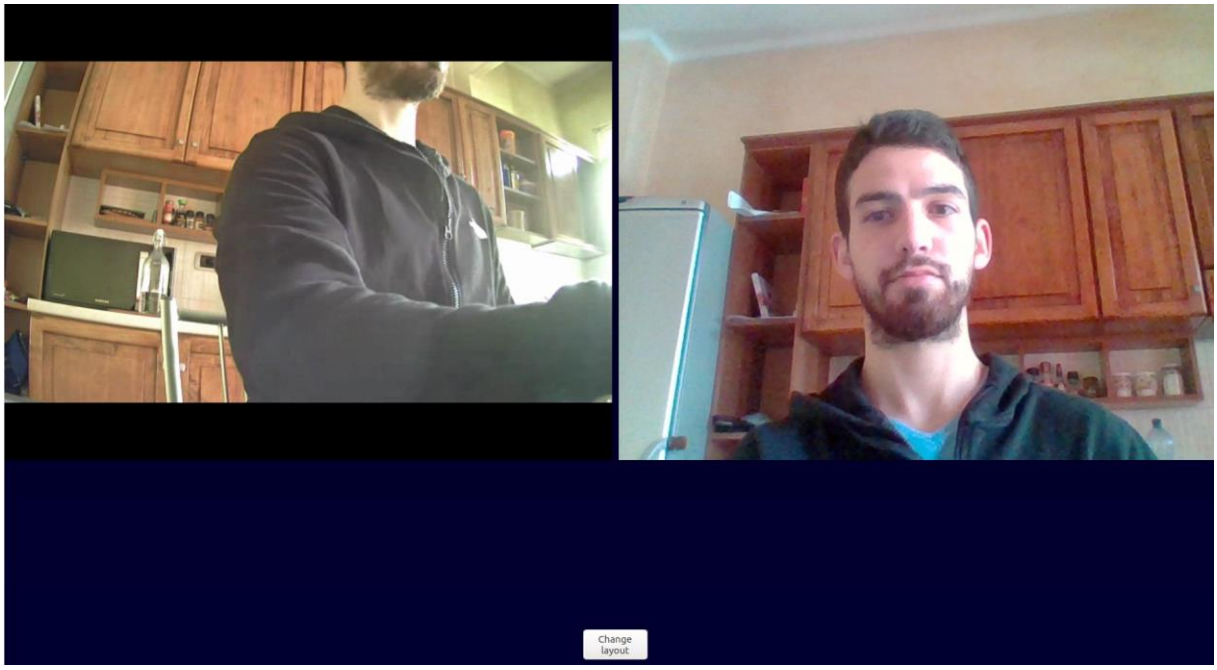


Figura 18 - Finestra pop-up: visualizzazione a griglia

Come si può vedere dalla precedente immagine la finestra è stata dotata di una doppia modalità di visualizzazione degli stream, modificabile tramite un apposito pulsante: nella prima, tradizionale, i flussi vengono mostrati tutti in una semplice disposizione a griglia; nella seconda invece viene mostrato il flusso relativo a un singolo utente, con la possibilità di passare all'utente successivo tramite l'utilizzo di un secondo pulsante.

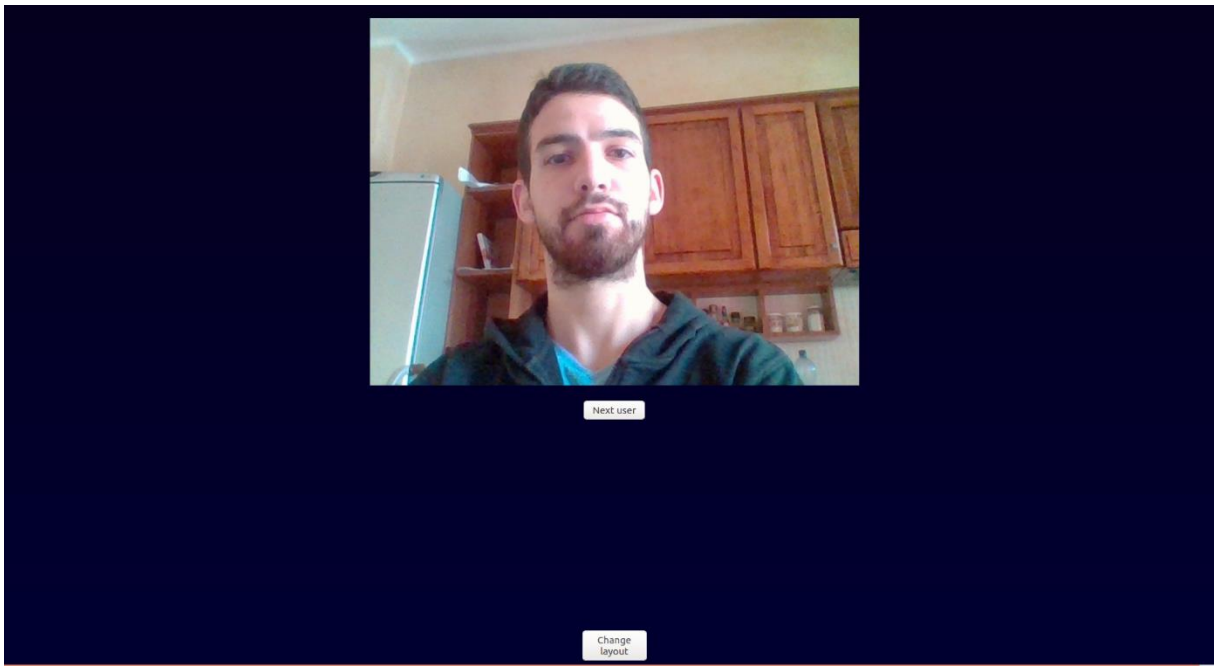


Figura 19 - Finestra pop-up: visualizzazione singola

A livello di codice si è agito principalmente all'interno di due classi:

- *VideoListItem* (video-list-item/component.jsx) che gestisce il singolo riquadro in cui mostrare il flusso video di un utente;
- *VideoProvider*, già vista in precedenza.

All'interno della classe VideoListItem è stato aggiunto il pulsante per l'inserimento/rimozione del flusso video all'interno della finestra secondaria. Alla pressione del pulsante viene eseguita la seguente funzione:

```
addStreamToPopup(){
  const { cameraId } = this.props
  const addStreamToPopupWindow = new CustomEvent('addStreamToPopupWindow', {
detail: { mediaElement: cameraId } });
  window.dispatchEvent(addStreamToPopupWindow);
}
```

Il compito del metodo è piuttosto semplice: recupera lo *streamId* del flusso video dell'utente e lo inserisce all'interno di un CustomEvent di cui viene fatto il *dispatch,* seguendo una linea di pensiero molto simile a quella utilizzata nelle precedenti implementazioni.

L'evento viene fatto intercettare dalla classe *VideoProvider* tramite l'utilizzo di un listener. All'arrivo del suddetto evento viene chiamato il metodo *handleNewVideoInPopup()*. Si ricordi che la classe *VideoProvider* mantiene una lista degli stream attualmente in esecuzione nel meeting all'interno della props *streams*.

```
handleNewVideoInPopup(e){
  const { mediaElement } = e.detail;
  const { swapLayout, currentVideoPageIndex, streams } = this.props;
  const { popupStreams } = this.state;

  let streamToAdd = streams.filter((s) => {
    return s.cameraId.toString() === mediaElement.toString()
  })[0]

  if(streamToAdd == null){
    return;
   }
  var currentPopupStreams = popupStreams;

  if(popupStreams.length === 0){
    currentPopupStreams.push(streamToAdd)

    this.setState({ popupStreams: currentPopupStreams })
    this.externalWindow = window.open('', "popupWindow",
"width="+screen.availWidth+",height="+screen.availHeight)

    let streamCameraids = [streamToAdd.cameraId.toString()]
```

```
    ReactDOM.render(
        <IntlProvider>
          <PopupVideoListContainer
              streams={streamCameraids}
              onMount={this.createVideoTag}
              swapLayout={swapLayout}
              currentVideoPageIndex={currentVideoPageIndex}
          />
        </IntlProvider>,

this.externalWindow.document.body.appendChild(document.createElement("DIV"))
    )
  } else {
    var streamAlreadyAdded = false

    currentPopupStreams.forEach(s => {
      if(s.cameraId.toString().startsWith(mediaElement.toString())){
        let index = currentPopupStreams.indexOf(s)
        currentPopupStreams.splice(index,1)
        streamAlreadyAdded = true
      }
    })

     if(!streamAlreadyAdded){
        currentPopupStreams.push(streamToAdd)
     }

    this.setState({ popupStreams: currentPopupStreams }, () => {
      if(streamAlreadyAdded){
        const removeVideoEvent = new CustomEvent('removePopupVideo', { detail: {
mediaElement: mediaElement.toString() } });
        window.dispatchEvent(removeVideoEvent);
        if(currentPopupStreams.length === 0){
          this.externalWindow.close()
          this.externalWindow = null
        }
      } else{
        const addStreamToPopupWindow = new CustomEvent('addPopupVideo', { detail:
{ mediaElement: mediaElement.toString() } });
        window.dispatchEvent(addStreamToPopupWindow);
        currentPopupStreams.push(streamToAdd)
      }
    })
  }
}
```

Come si può notare all'interno dello state della classe *VideoProvider* è stata aggiunta la chiave *popupStreams*, che conterrà gli id degli stream correntemente inseriti all'interno della finestra pop-up.
La funzione recupera lo streamId relativo all'utente selezionato dalla props *streams* e verifica la presenza di elementi all'interno di *popupStreams*.

Per semplicità verranno analizzati separatamente i due possibili scenari: quello in cui non vi sia ancora nessun flusso nella finestra secondaria e quello in cui ve ne sia almeno uno.

Nel primo caso la finestra pop-up non è ancora stata creata. La funzione quindi aggiunge lo stream all'interno dello state *popupStreams*, e apre una nuova finestra. All'interno della classe VideoProvider è stata aggiunta la variabile *externalWindow*, che conterrà una reference alla nuova finestra. L'apertura avviene attraverso la chiamata alla funzione *window.open()*, messa a disposizione dalla WebAPI di MDN, che permette di caricare una risorsa all'interno di una nuova pagina del browser. [28]

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet
open	1	12	1	4	3	1	1	18	4	10.1	1	1.0

Figura 20 - Compatibilità tra window.open() e i moderni browser

All'interno della finestra viene renderizzato il componente *PopupVideoListContainer*, che si occupa della gestione dei flussi video all'interno della finestra pop-up. Il flusso da visualizzare è ovviamente unico alla creazione della finestra, e viene passato tramite props al componente.
PopupVideoListContainer fa parte di una serie di componenti creati appositamente per gestire i flussi video all'interno della finestra di pop-up. Essi sono:

- *PopupVideoListContainer*: funge da tramite tra l'effettivo componente e il resto dell'applicazione, gestendo il passaggio delle props che arrivano dall'esterno;

- *PopupVideoListComponent*: si occupa della gestione dei flussi video nella loro totalità, utilizzando props e state. All'interno del suo state sono state inserite le chiavi *shownStreams* e *removedStreams:* la prima è associata a un array contenente gli id dei flussi inseriti nel pop-up, la seconda a un array contenente gli id dei flussi resi non visibili (dalla funzionalità descritta nel capitolo precedente);

- *PopupVideoListItemComponent*: si occupa del rendering del singolo flusso video all'interno della finestra, gestendo il riquadro che lo andrà a contenere;

- *PopupVideoListItemContainer:* ha una funzione di collegamento del relativo component con il resto dell'app, analogamente a *PopupVideoListContainer*.

Quando la finestra secondaria viene aperta, al suo interno viene caricato *PopupVideoListComponent,* e lo state *shownStreams* viene inizializzato con lo streamId dell'unico flusso da gestire.
Il componente si occupa quindi di recuperare lo stream associato a quello specifico id e visualizzarlo su schermo. Di seguito viene mostrato un estratto di codice che mostra il lavoro svolto dal componente:

```
renderVideoList() {
  const {
    intl,
    onMount,
    swapLayout,
  } = this.props;
  const { focusedId, removedStreams, shownStreams } = this.state;
  const numOfStreams = shownStreams.length;

  return shownStreams.filter(s=> {
    let shouldBeSeen = true
    removedStreams.forEach(ss=>{
      if(s.toString() === ss.toString()){
        shouldBeSeen = false
      }
    })
    return shouldBeSeen
  }).map((stream) => {
    const { streams, totalNumberOfStreams } = VideoService.getVideoStreams()
    let selectedStream = streams.filter((s) => {
      return s.cameraId.toString().startsWith(stream.toString())
    })[0]

    const { cameraId, userId, name } = selectedStream;
    //...
    return (
        <div
            key={cameraId}
            style={{display: `grid-item`, backgroundColor: `black` }}
        >
          <PopupVideoListItemContainer
              numOfStreams={numOfStreams}
              cameraId={cameraId}
              userId={userId}
              name={name}
              actions={actions}
              onMount={(videoRef) => {
                this.handleCanvasResize();
                onMount(cameraId, videoRef);
              }}
              swapLayout={swapLayout}
```

```
          />
        </div>
    );
  });
}
```

Come si può vedere, gli streamId contenuti all'interno di *shownStreams* (quelli inseriti nel popup) vengono scremati degli id indicati come "nascosti": gli id che superano questo primo controllo vengono effettivamente visualizzati su schermo. Gli sviluppatori di BigBlueButton mettono a disposizione la funzione *getVideoStreams()* della classe *VideoService*, che recupera tutti i flussi webcam disponibili sul client. Cercando per streamId viene trovato il flusso corrispondente, che viene poi visualizzato dalla classe *PopupVideoListItemContainer*.

La finestra secondaria viene così riempita con lo stream della webcam del primo utente selezionato. Tornando nella funzione *handleNewVideoInPopup()* è possibile ora analizzare il secondo scenario, quello in cui risultano essere già presenti dei video nella finestra pop-up. In questo caso la finestra non viene ricreata, ma si aggiorna lo state *popupStreams* di VideoProvider, aggiungendo o rimuovendo lo streamId. Per notificare la finesta pop-up delle modifiche apportate vengono quindi usati nuovamente dei CustomEvent: l'evento identificato con '*addPopupVideo*' segnala l'aggiunta di una nuova webcam nel pop-up; l'evento identificato con '*removePopupVideo'* segnala invece una rimozione. In entrambi i casi il *CustomEvent* ingloba al suo interno lo streamId contrassegnante il flusso in questione.

Gli eventi sono quindi intercettati e gestiti dalla classe *PopupVideoListComponent* (che si ricorda gestisce la totalità dei flussi nella finestra esterna e regola la loro visualizzazione basandosi sul contenuto del suo state).
Le funzioni eseguite per l'aggiunta e la rimozione di un flusso dalla finestra sono rispettivamente *addPopupVideo()* e *removePopupVideo()*.

```
addPopupVideo(e){
  const { mediaElement } = e.detail;
  const { shownStreams } = this.state;

  var currentStreams = shownStreams;
  currentStreams.push(mediaElement.toString())

  this.setState({ shownStreams: currentStreams })
}

removePopupVideo(e){
  const { mediaElement } = e.detail;
  const { shownStreams } = this.state;

  var currentStreams = shownStreams;
  currentStreams.forEach(s => {
    if(s.toString().startsWith(mediaElement.toString())){
      let index = currentStreams.indexOf(s)
```

```
      currentStreams.splice(index,1)
    }
  })
  this.setState({ shownStreams: currentStreams })
```

Entrambe le funzioni sono pensate per aggiornare semplicemente lo state *shownStreams* del componente, aggiungendo o rimuovendo lo streamId corrispondente alla webcam selezionata. Come si è visto in precedenza, i flussi visualizzati dal componente all'interno del pop-up sono unicamente quelli contrassegnati da id contenuti all'interno dello state *shownStreams*. La semplice modifica dello state permette quindi di ottenere il risultato desiderato.

7 Conclusioni

Quest'ultima funzionalità conclude la descrizione del lavoro svolto in questo progetto di tesi. Ricapitolando il lavoro svolto, i risultati ottenuti sono stati i seguenti:

- Si è analizzato il sistema di videoconferenza BigBlueButton, studiandone l'architettura, i componenti principali e la comunicazione tra gli stessi;
- Si è mostrato come creare un server BigBlueButton funzionante, passando anche dall'ottenimento di un certificato SSL valido;
- Si è implementata una modalità di condivisione della webcam con un flusso dati generato da immagini campionate ogni 10 secondi dal flusso originale; questo ha permesso di ridurre sensibilmente il traffico dati in uscita generato dall'accensione della webcam stessa, e migliorare quindi le performance in rete del client;
- E' stata aggiunta la possibilità di interrompere la ricezione di flussi video generati dalle webcam di altri partecipanti al meeting; anche in questo caso, a beneficiarne è l'impatto del client sulla banda di rete, questa volta nel traffico in ingresso;
- E' stata introdotta la possibilità di inserire i riquadri della webcam in una finestra pop-up separata, dotata di doppio layout; tale aggiunta pone le basi per la realizzazione di un servizio di streaming dell'intero meeting, realizzabile tramite l'utilizzo di un software di video-recording esterno, come OBS Studio;

Per quanto già BigBlueButton sia un sistema complesso e pieno di funzionalità, sarebbe possibile aggiungere molte altre caratteristiche all'applicazione oltre a quelle implementate in questo progetto. Essendo BigBlueButton una piattaforma open source con una comunità aperta di sviluppatori, nuove idee vengono concepite e implementate ogni giorno, rendendo il sistema nel suo complesso un prodotto in continua evoluzione.
In un'epoca in cui lo smart working e in generale i contatti da remoto hanno raggiunto una tale importanza, l'evoluzione dei software per il lavoro a distanza non può che essere accolto positivamente.

Bibliografia

[1] DigitalInTheRound, «20 Astonishing Videoconferencing Statistics for 2021 » [Online]. Available: https://digitalintheround.com/video-conferencing-statistics/.

[2] BigBlueButton, «Architecture,» [Online]. Available: https://docs.bigbluebutton.org/2.2/architecture.html.

[3] Wikipedia, «HTML5,» [Online]. Available: https://it.wikipedia.org/wiki/HTML5.

[4] Wikipedia, «Adobe Flash,» [Online]. Available: https://it.wikipedia.org/wiki/Adobe_Flash.

[5] Ionos, «HTML5 vs Adobe Flash,» [Online]. Available: https://www.ionos.it/digitalguide/siti-web/programmazione-del-sito-web/perche-abbandonare-flash-per-passare-allhtml5/.

[6] BigBlueButton, «Legacy,» [Online]. Available: https://docs.bigbluebutton.org/legacy/html5-overview.html.

[7] WebRTC, «Introduction,» [Online]. Available: https://webrtc.org/.

[8] Kurento Media Server, «Documentation,» [Online], Available: https://doc-kurento.readthedocs.io/en/latest/user/intro.html#what-is-kurento.

[9] CallStats, «webRTC architectures,» [Online]. Available: https://www.callstats.io/blog/webrtc-architectures-explained-in-5-minutes-or-less.

[10] Kurento Media Server, «kurento-utils,» [Online]. Available: https://doc-kurento.readthedocs.io/en/stable/features/kurento_utils_js.html#overview.

[11] BigBlueButton, «Install» [Online]. Available: https://docs.bigbluebutton.org/2.2/install.html#install

[12] GoDaddy, «Home,» [Online]. Available: https://it.godaddy.com.

[13] Let's Encrypt, «Session,» [Online]. Available: https://letsencrypt.org.

[14] BigBlueButton, «Development,» [Online]. Available: https://docs.bigbluebutton.org/2.3/dev.html.

[15] GitHub, «BigBlueButton,» [Online]. Available: https://github.com/bigbluebutton.

[16] Meteor, «Session,» [Online]. Available: https://docs.meteor.com/api/session.html.

[17] Mozilla, «getUserMedia,» [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia.

[18] Kurento Media Server, «generateOffer,» [Online]. Available: https://doc-kurento.readthedocs.io/en/stable/features/kurento_utils_js.html#generateoffer.

[19] Mozilla, «canvas API,» [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API.

[20] Tecmint, «nload,» [Online]. Available: https://www.tecmint.com/nload-monitor-linux-network-traffic-

bandwidth-usage/.

[21] ReactJs, «State,» [Online]. Available: https://it.reactjs.org/docs/state-and-lifecycle.html.

[22] Mozilla, «CustomEvent,» [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent.

[23] Mozilla, «EventTarget,» [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Event/target.

[24] Mozilla, «dispatchEvent,» [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/dispatchEvent.

[25] Mozilla, «addEventListener,» [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener.

[26] Mozilla, «addEventListener,» [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/removeEventListener.

[27] Kurento Media Server, «dispose,» [Online]. Available: https://doc-kurento.readthedocs.io/en/stable/features/kurento_utils_js.html#dispose.

[28] Mozilla, «window.open,» [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/open.