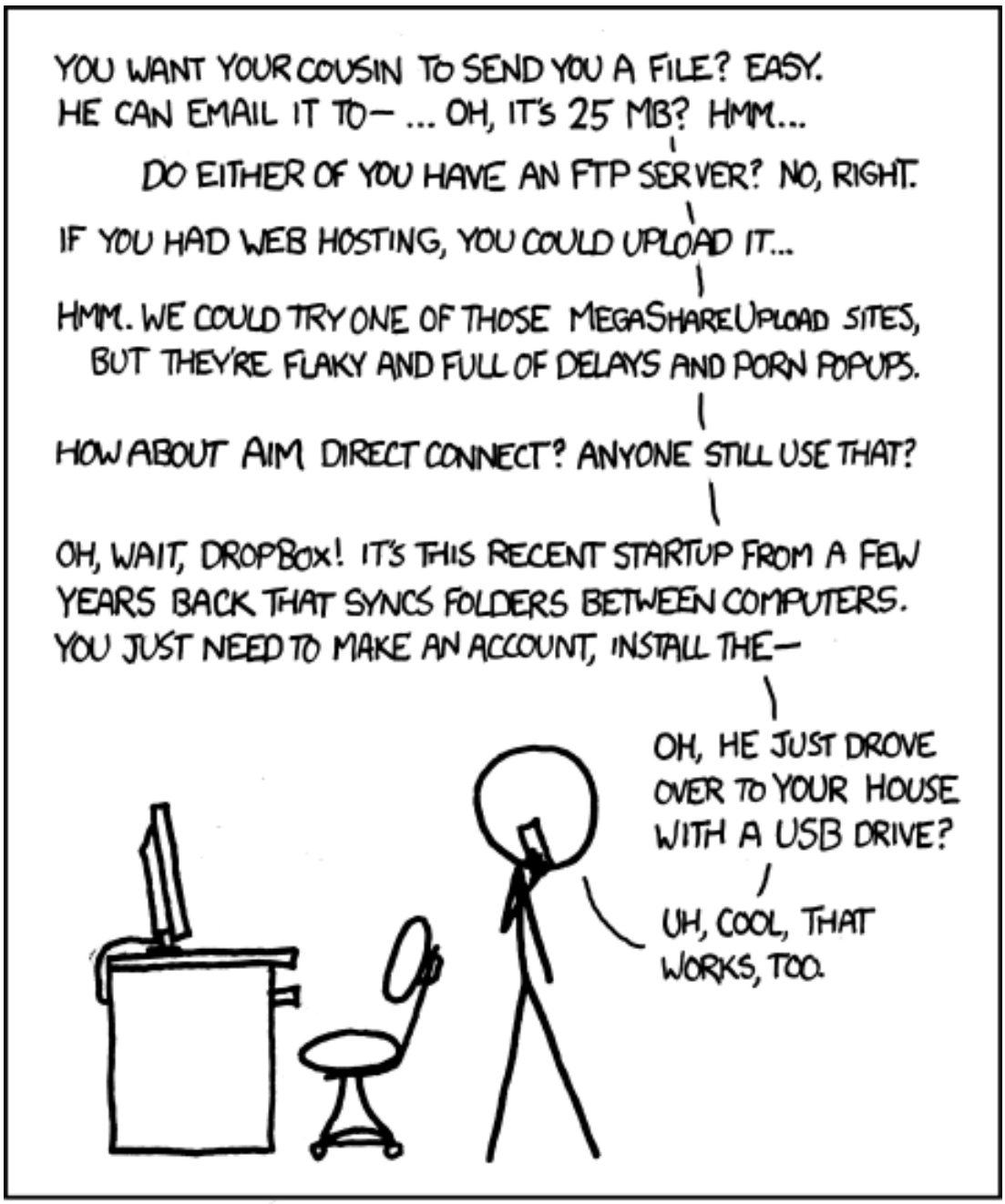
Webmission

Funcionamiento

Webmission es una aplicación web p2p que permite el envío de ficheros entre dos usuarios, ofreciendo una solución sencilla a este caso de uso:



I LIKE HOW WE'VE HAD THE INTERNET FOR DECADES, YET "SENDING FILES" IS SOMETHING EARLY ADOPTERS ARE STILL FIGURING OUT HOW TO DO.

Nuestra aplicación presenta las siguientes ventajas con respecto a otras herramientas de envío de ficheros (correo electrónico, ftp, almacenamiento en

servidores ajenos -dropbox, google drive -, aplicaciones p2p -torrent, emule-) :

- Al ser una aplicación web, no precisa de instalación en el cliente. Tampoco se necesita instalar plugin alguno.
- No existe límite en cuanto al tamaño del fichero a enviar: únicamente la limitación física en el espacio de almacenamiento del usuario receptor.
- No existe un servidor centralizado que guarde los datos enviados ni los referentes a los usuarios.
- No se requieren cuentas de usuario ni login: cualquiera puede usar Webmission.

Existen múltiples y populares aplicaciones P2P y protocolos, como eMule, BitTorrent o Ares. Sin embargo, es necesario instalar el programa en el sistema o bien en el navegador mediante un plugin. Con Webmission, el usuario lo único que tiene que hacer es abrir una página web. Otra ventaja de esto es que, en caso de lanzar una mejora o actualización en la aplicación, esta se realizaría de manera inmediata al volver a utilizar el servicio.

Para ello, la aplicación hace uso de unas tecnologías de reciente aparición: **node.js** para la creación del servidor y la señalización previa a la conexión de los usuarios; y las APIs de Javascript **WebRTC** y **Filesystem**, encargadas del envío de datos y el guardado de estos en disco duro, respectivamente.

Comencemos por el principio:

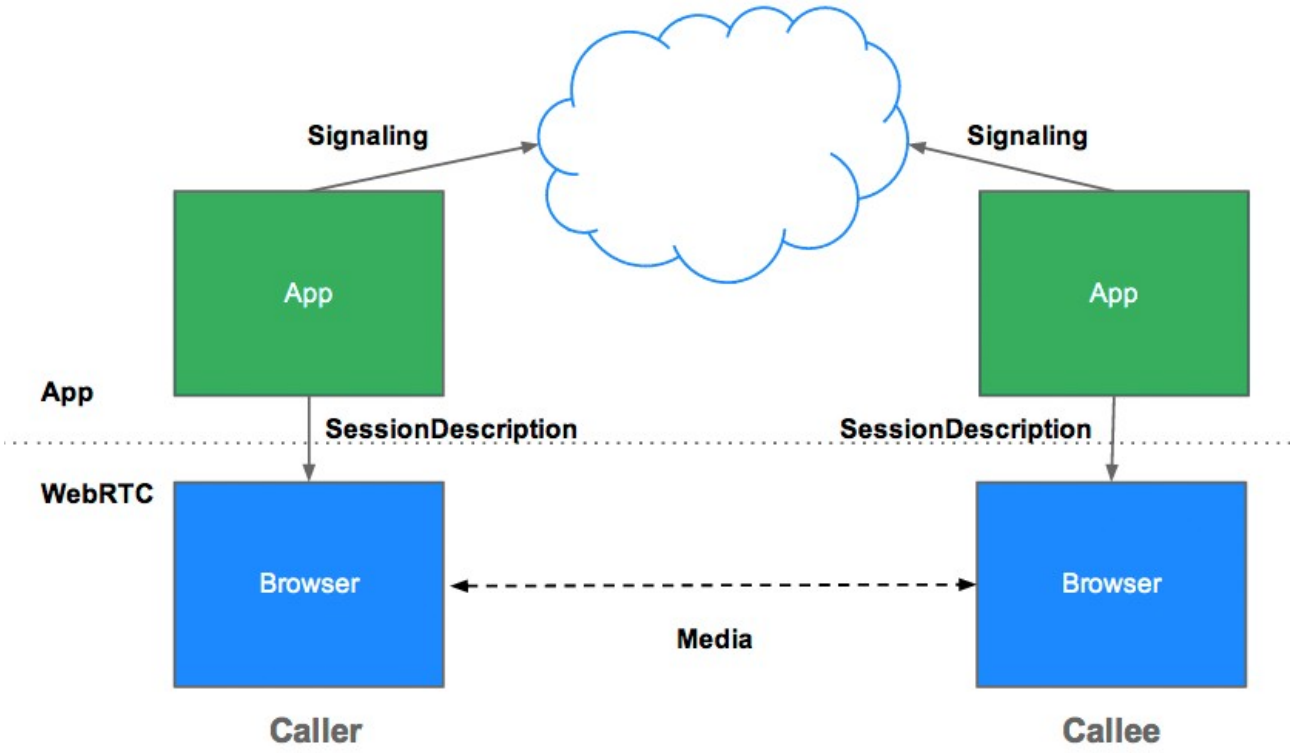
WebRTC (Web Real Time Communication) es una especificación de estándares abiertos para comunicación en tiempo real de todo tipo de datos. Está compuesta en realidad de 3 APIs :

- **MediaStream** (para video y audio).
- **RTCPeerConnection** (establece y mantiene una conexión estable).
- **RTCDataChannel** (permite la comunicación bidireccional de todo tipo de datos).

Nuestra aplicación utiliza específicamente estas dos últimas APIs. Pero, además, precisa de un mecanismo de señalización que permita establecer el contacto entre los usuarios previamente al establecimiento de la conexión. WebRTC no incluye este mecanismo, pudiéndose realizar de diversas maneras. En Webmission se ha optado por utilizar **Socket.io** corriendo sobre un servidor **Node**.

Este mecanismo de señalización se usa para intercambiar tres tipos de información:

- *Mensajes de control de sesión*: inicio o cierre de la comunicación.
- *Configuración de red*: reconocimiento de la IP pública y puerto que se está usando, a través de un servidor STUN.
- *Mensajes de tipo ICE candidate* (ip pública y puertos disponibles) *y SDP* (Session Description Protocol).



El intercambio de información debe realizarse (y completarse) antes de que se pueda establecer la conexión propiamente dicha con las APIs WebRTC, estando todo el conjunto interconectado en el proceso. Sin embargo, para mayor claridad en la explicación, vamos a explicar por separado el proceso por el cual dos navegadores realizan su interconexión en Webmission:

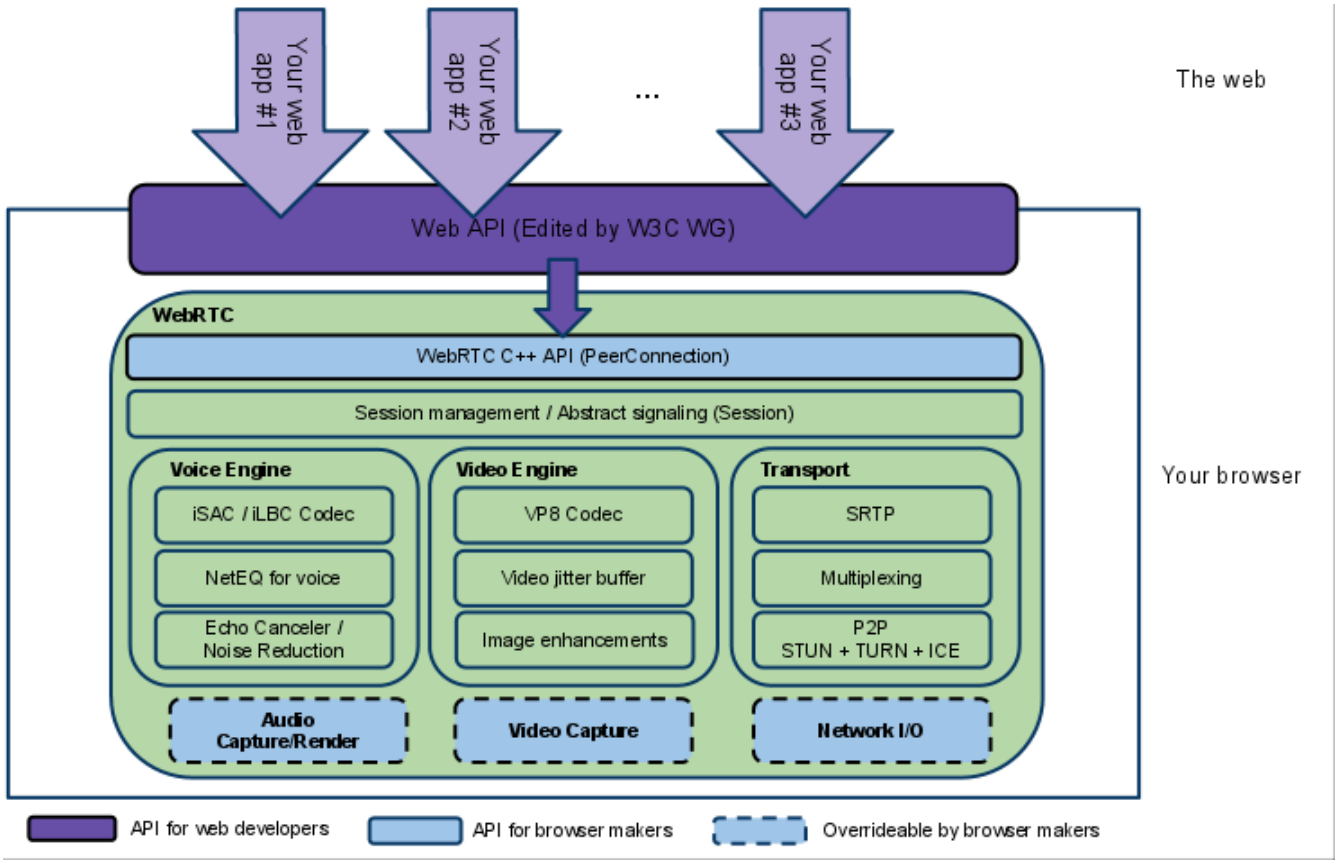
1. A crea una instancia de RTCPeerConnection, con un manejador onIceCandidate
2. El manejador es invocado cuando hay candidatos disponibles (una ip y un puerto).
3. A envía su información de red a B, a través del mecanismo de señalización.
4. B recibe el mensaje de información de A.
5. B llama a addIceCandidate. Se vuelve a repetir el punto 3, pero intercambiando los actores.

1. A crea su SDP (local) y la envía a B a través del mecanismo de señalización.

2. B recoge la SDP enviada por B y la guarda (remota) crea la suya propia (local), que es enviada a A.
3. A recibe la SDP de B, la guarda como remota
4. PeerConnection ya tiene los elementos que necesita para establecer la conexión.
5. Sobre esta base, abrimos DataChannel.
6. ¡La conexión y el canal de datos están disponibles!

Como hemos dicho, estos procesos se realizan simultáneamente y deben finalizar con éxito antes de establecer la conexión, la cual es labor de RTCPeerConnection.

RTCPeerConnection es el componente de WebRTC encargado de manejar una conexión directa, estable y lo más eficiente posible entre pares.

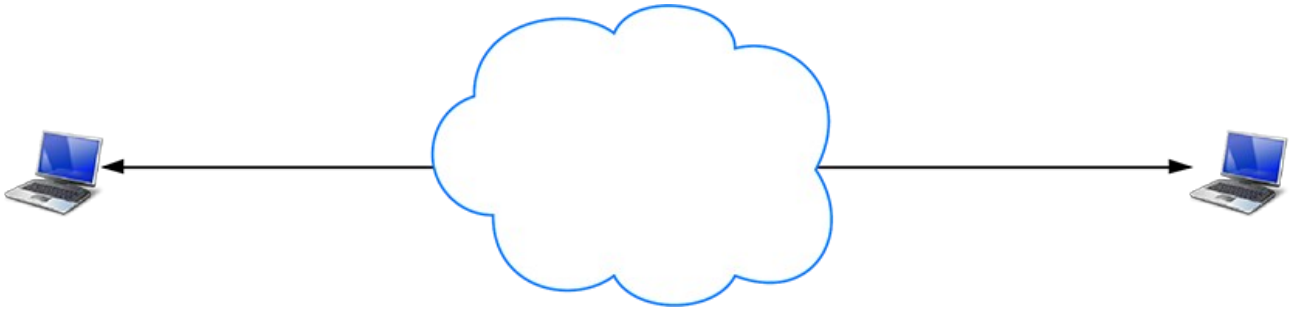


Internamente realiza una labor muy compleja (manejo de pérdida de paquetes, de

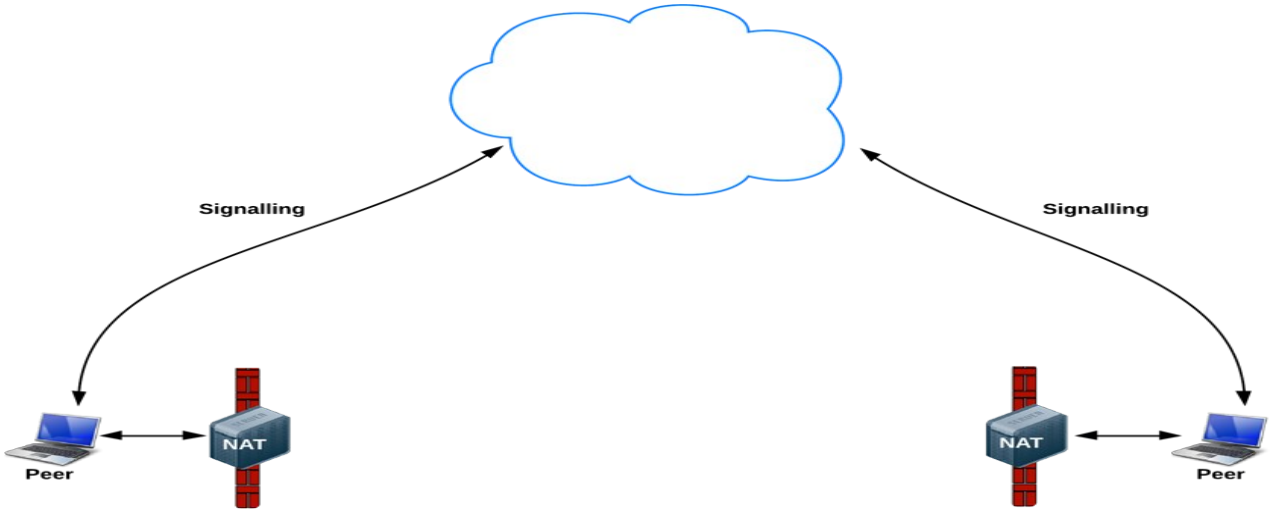
eco, adaptabilidad del ancho de banda...), ofreciendo al mismo tiempo al desarrollador una interfaz muy clara y simple.

Con el fin de ofrecer una ligera pincelada, hemos de indicar que RTCPeerConnection utiliza el framework ICE para poder lograr la conexión a través de NATs y cortafuegos.

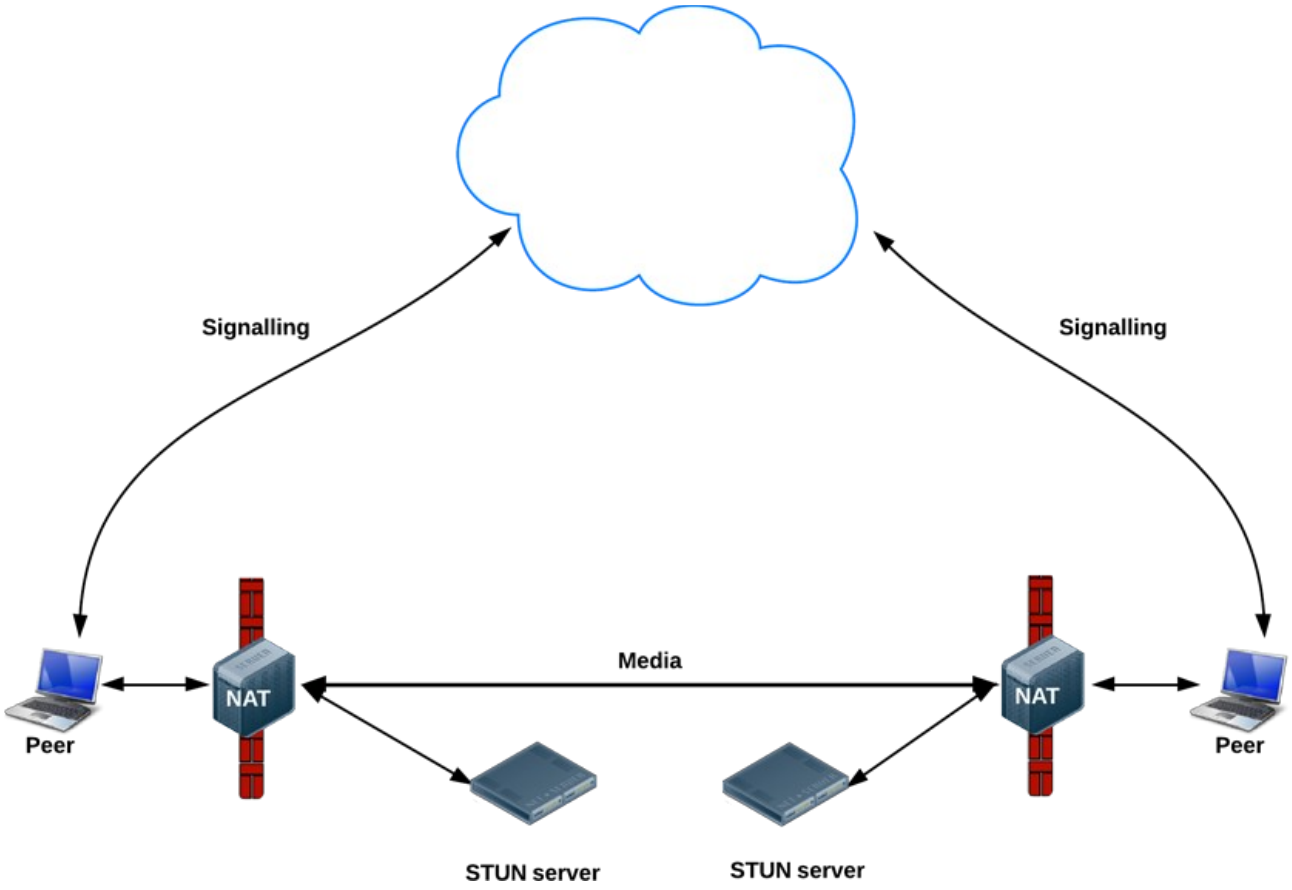
Esto no ocurre:



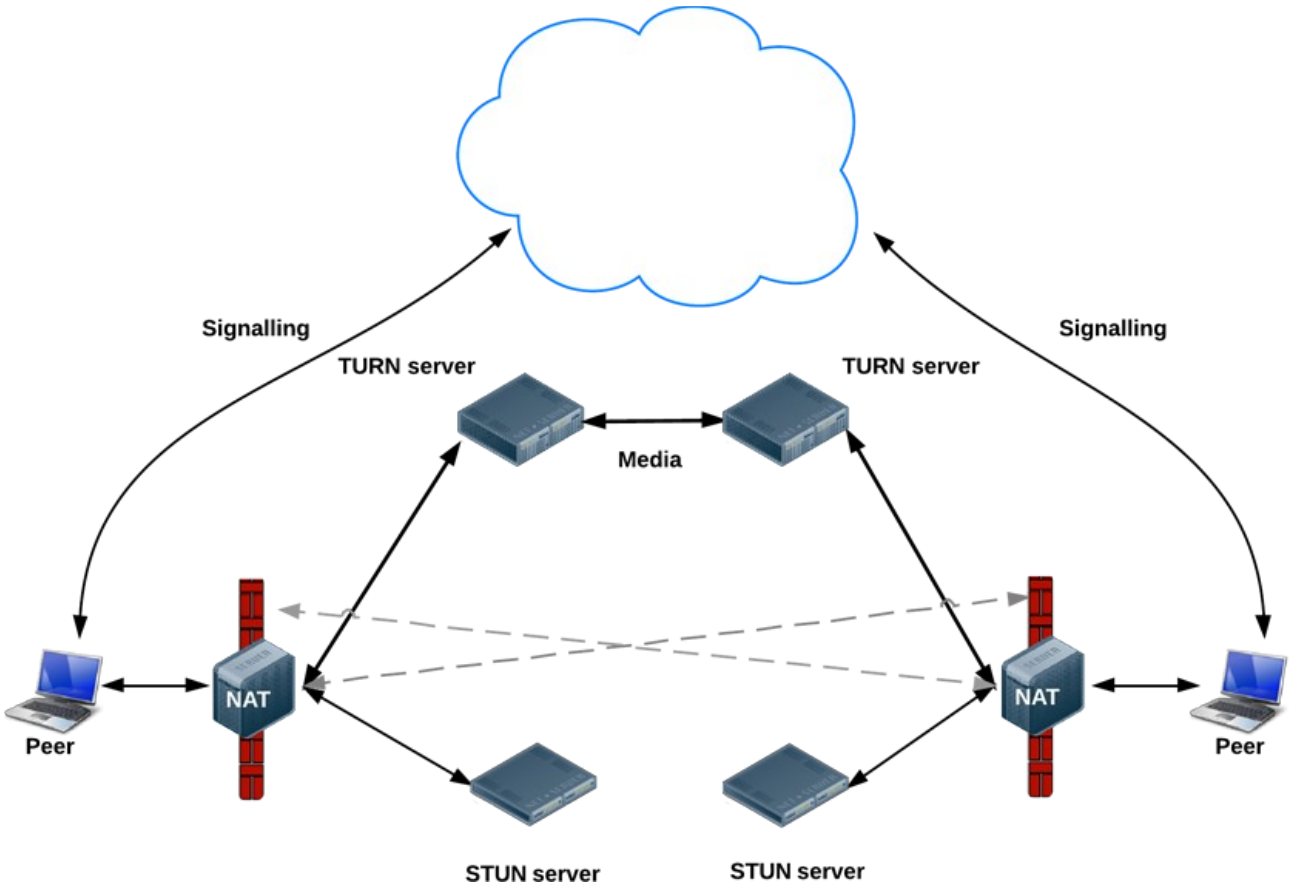
Nos encontramos con esto:



ICE (Interactive Connectivity Establishment) trata de realizar la conexión de la siguiente manera:



1. En primer lugar se intenta la conexión directamente, con la más baja latencia posible, mediante UDP. En este proceso, el servidor STUN tiene como único objetivo permitir que una máquina detrás de un NAT averigüe su propia IP pública (en Webmission utilizamos un servidor STUN público ofrecido por Google).
2. Si UDP no funciona, ICE lo intenta dos veces más a través de TCP: primero HTTP, y luego HTTPS.



3. Si hasta aquí no se ha logrado la conexión, es bastante probable que sea debido a cortafuegos o configuraciones del NAT. Se intentaría entonces mediante un servidor TURN (un derivado de un servidor TURN que permite el paso de información) y, dependiendo de la severidad de las restricciones encontradas, no se establecería una conexión P2P, sino que toda la información a través de este servidor. Webmission utiliza uno de pruebas ofrecido gratuitamente por la empresa canadiense de telecomunicaciones Viagenie.

Una vez establecida la conexión hacemos uso de **RTCDataChannel**, una API muy interesante del conjunto WebRTC que permite el envío de cualquier tipo de datos entre navegadores.

Nos atrevemos a decir que, aunque la API MediaStream sea la más renombrada a la hora de tratar sobre WebRTC por su uso en populares aplicaciones de videochat

(hangouts de google, firefox hello, implementaciones beta en skype...) , a nuestro juicio RTCDataChannel ofrece unas posibilidades infinitas que pueden llegar a abrir una nueva etapa en la web (redes descentralizadas, CDN, aplicaciones de escritorio remoto, transferencias de datos...).

El canal de datos queda abierto entonces, encargándose ya exclusivamente de la comunicación bidireccional entre los distintos navegadores y terminando aquí la labor del mecanismo de señalización y de los servidores externos (únicamente es necesario el servidor node a efectos exclusivamente de mostrar el documento html).

Webmission hace uso de este canal de datos para ofrecer un sencillo chat de texto, así como para su principal labor: el envío de cualquier tipo de archivo.

Veamos cuál es el proceso seguido para enviar un archivo:

1. Cuando un usuario se conecta, la aplicación comprueba la url. Si únicamente contiene el nombre del dominio, se ofrece la posibilidad de crear una sala. Al ser creada esta, se añadirá a la url un código aleatorio alfanumérico que la identifica unívocamente.
2. El usuario indica por sus propios medios esta url completa a la persona con quien quiera compartir un archivo.
3. Este otro usuario, B, entra en esa dirección y la aplicación inmediatamente inicia el proceso de señalización y conexión que hemos comentado anteriormente.
4. A ofrece un fichero a B, al cual le llega información acerca del nombre, tipo de fichero y peso.
5. Si acepta, la aplicación en A hace uso de la API **Filereader** para leer el archivo en trozos/chunks mediante el siguiente procedimiento:
   - El archivo es cargado en memoria en trozos de 1MB, con el fin de no sobrecargar la memoria RAM del usuario que envía.
   - A su vez, estos trozos son divididos en fragmentos de 13 KB. La razón de

esto es que Chrome permite envíos de hasta 16 KB, pero la experiencia durante el desarrollo nos ha demostrado que 13KB es un tamaño razonable para evitar posibles sorpresas y fallos en el transporte.
   - Estos trozos, que se encuentran codificados en base64, se envían uno a uno a través del canal de datos.
   - Cuando se llega al final de un chunk, es decir, se termina de enviar 1MB del fichero original, se envía una señal para indicárselo al receptor. En caso de no haberse terminado de enviar el fichero (sea mayor de 1MB), se carga el chunk correspondiente a > 1MB y <= 2MB, volviendo a realizarse el proceso de subdivisión y envío de fragmentos. En su caso, se envía la señal de fin de fichero.
6. El receptor, B, actúa de la manera siguiente:
   - Recoge estos fragmentos de chunk en un array.
   - Cuando llega la señal de final de chunk, mediante la API **Filesystem** se vuelca el array a un archivo en el disco duro, previa conversión de los datos en base 64 a base 8, necesario para poder crear el blob.
   - Prosigue la recogida de fragmentos y sobreescritura en el disco duro hasta que llega la señal de fin de fichero.
   - Se crea un enlace para la conversión y descarga de ese fichero, con el nombre y el tipo de archivo originales.

La API **Filesystem** permite la lectura / escritura de directorios y ficheros en una zona acotada del disco duro, una *sandbox*.

Las opciones de uso de esta API son las siguientes:

- Temporal.
  - El navegador puede borrar los datos almacenados de manera discrecional.
  - Existe un límite de capacidad: la mitad del espacio libre en disco y, de ese 50%, cada aplicación no puede superar un máximo del 20%. (Si el espacio libre es de 50 GB, cada aplicación no puede usar más de 5GB).
  - Disponible en Chrome, Opera y Firefox (este último mediante un *polyfill*

que hace uso de InnoDB: Filesystem no está disponible oficialmente para Firefox).

- Persistente
  - Los datos permanecen en disco hasta que el usuario, o la aplicación, decida eliminarlos.
  - No existe límite de capacidad más allá de la disponibilidad de espacio libre en disco.
  - Únicamente disponible en Chrome y Opera.

En aras de seguir la filosofía P2P de la aplicación y no poner trabas en cuanto al tipo y tamaño de los ficheros a compartir, hemos decidido utilizar el tipo de almacenamiento persistente aunque ello conlleve el precio de que, de momento, sólo pueda ser usada la aplicación en navegadores con motor webkit.

Como conclusión, podemos decir que Webmission posee las siguientes características:

- Se trata de una aplicación P2P cuyo funcionamiento no requiere de un servidor central: sólo hace falta un servidor node para alojar la aplicación (que puede ser en la máquina misma de uno de los usuarios), un servidor STUN que permita el handshake y, eventualmente, un servidor TURN en caso de que no se pueda establecer una conexión directa. Estas dos últimas necesidades podrían ser obviadas en un futuro, en función del desarrollo de las APIs de WebRTC.
- Garantiza el anonimato: no se requieren logins y los datos enviados mediante RTCDataChannel se encuentran cifrados mediante los protocolos DTLS y TLS .
- No se recopilan datos en ningún servidor.
- Al tratarse de una aplicación web, en caso de actualización no habría problema de incompatibilidades entre usuarios con distintas versiones o tecnologías.

En el desarrollo no se ha seguido un diseño estricto POO (clases, herencias...), sino que se han creado y utilizado objetos de tipo JSON conteniendo cada uno de ellos las variables y funciones necesarias.
Se ha adoptado un patrón singleton, de acceso único mediante la función initMain.
Los objetos utilizados más importantes son los siguientes:

FileSystem

-fs
-root
-flagInicio
-nombreDirectorio

+disponibleFileSystem()
+initFileSystem()
+crearDirectorio()
+guardarArchivo()
+crearEnlaceDescarga()
#manejaError()

GestorSala

+compruebaEnSala()
+escuchaMensajesSala()

GestorConexion

+conexP2P
+dataChannel
+socket

+enviarMensajeConexion()
+procesarMensajeConexion()
+crearConexionP2P()
+crearSesionLocal()
#manejaError()

GestionaContenidos

+navegadorNoCompatible()
+iniciador()
+vaciarElemento()
+vaciarElemento()
+annadeChat()
+annadeBarraProgreso()
+ofreciendo()
+enviando()
+recibiendo()

CargaEnvia

+CHUNK_LENGTH
-chunkActual
-numChunks
+envioDetenido
-archivo

+enviarFichero()
-cargaSiguienteChunk()
-procesaChunk()
-enviaChunk()

InitMain

-sala
-socket

-compruebaUrl()
-annadeTurn()
-conexionServidorSennalizacion()

ControlTransmision

-dataChannel

+cancelarEnvio()
+cancelarRecepcion()
+pausaReanudaEnvio()
+pausaReanudaRecepcion()

DragAndDrop

+archivosDnD

+cargaFicheroDnD()
+seleccionadoEnInput()
+preventDragOver()
+onClickDropZone()

RecibeDataChannel

-dataChannel

-onOpen()
-onClose()
-onMessage()
-procesaRecibido()

Chat

-mensaje
-rol
-dataChannel

+enviaMensaje()
+recibeMensaje()