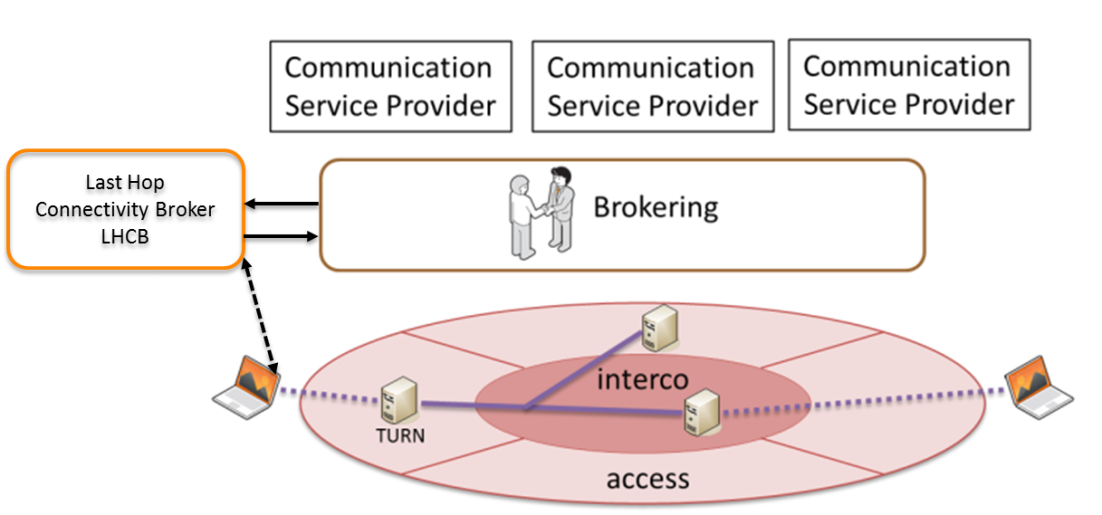
## Quality of Service Control

### Overview

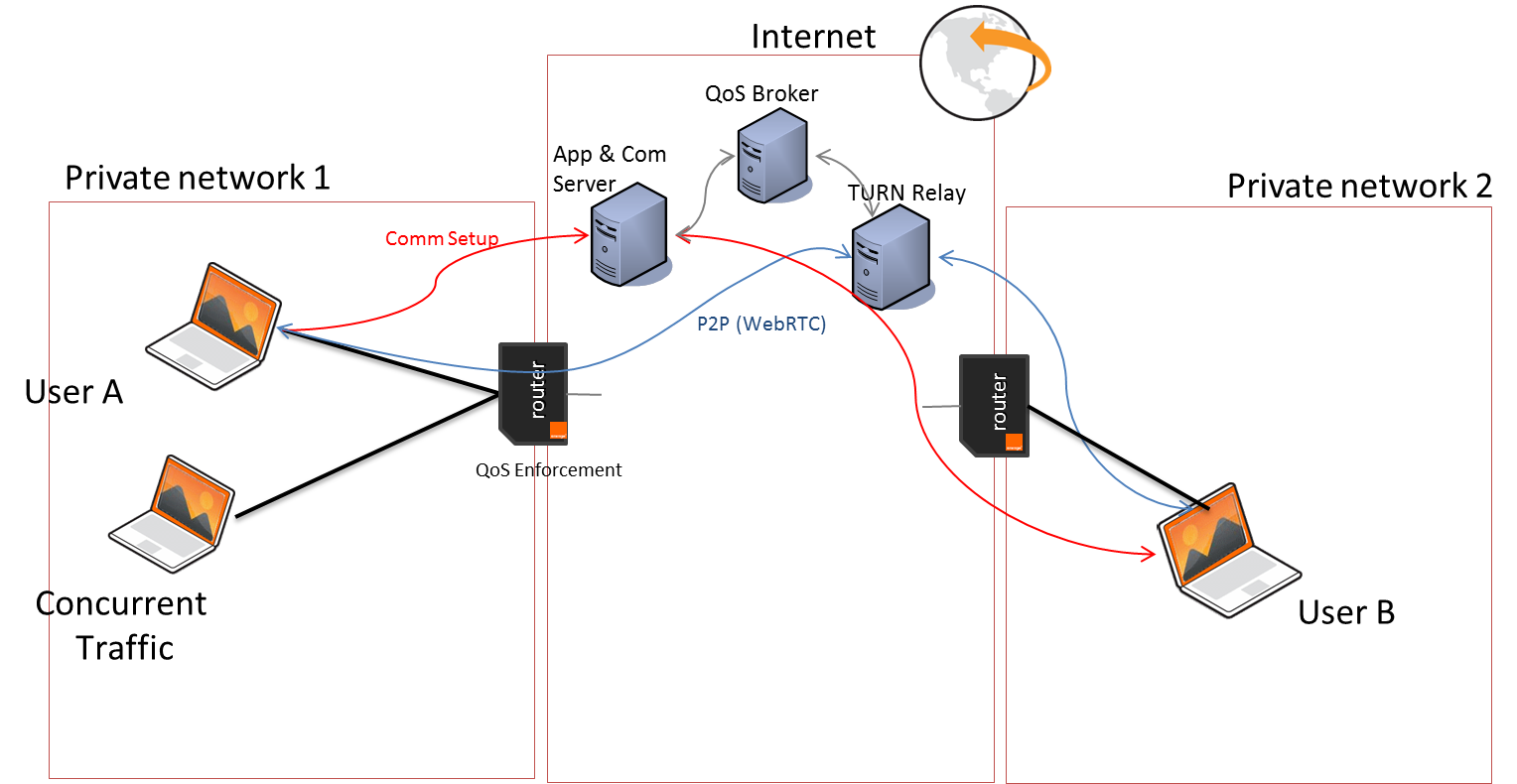
#### QoS in reTHINK

The reTHINK architecture enables activating QoS and policy as selectable options, via APIs to the service providers. While OTT services have no such choice, and Mobile services automatically provide managed QoS over managed packet network, the reTHINK architecture can deliver QoS ‘on-demand’ over the Internet, selected only where necessary, according to network conditions, user preference and service requirements. Several QoS enforcing points and technologies have been envisioned. One of the solutions is based on providing QoS on CPE Broadband and mobile access. The other one is a solution based on network selection (LHCB) in which a client provides information about available "uplinks" (i.e. alternative wired or wireless connections) and associated quality parameters, and in which the client may be requested to switch its connectivity over to an indicated network (interface) providing a certain QoS level.

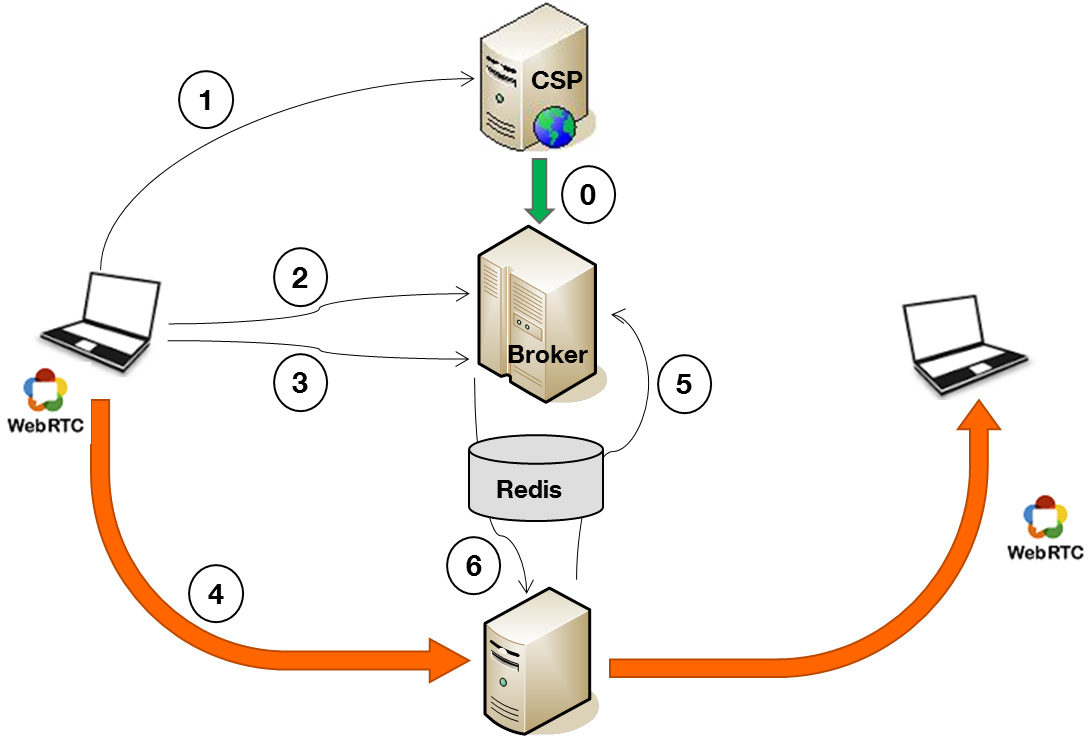


#### reTHINK TURN services

On the first solution, design of network traffic control has been implemented in the CPE. The general mechanism is the following one:



To use the right TURN server, the general flow is the following:



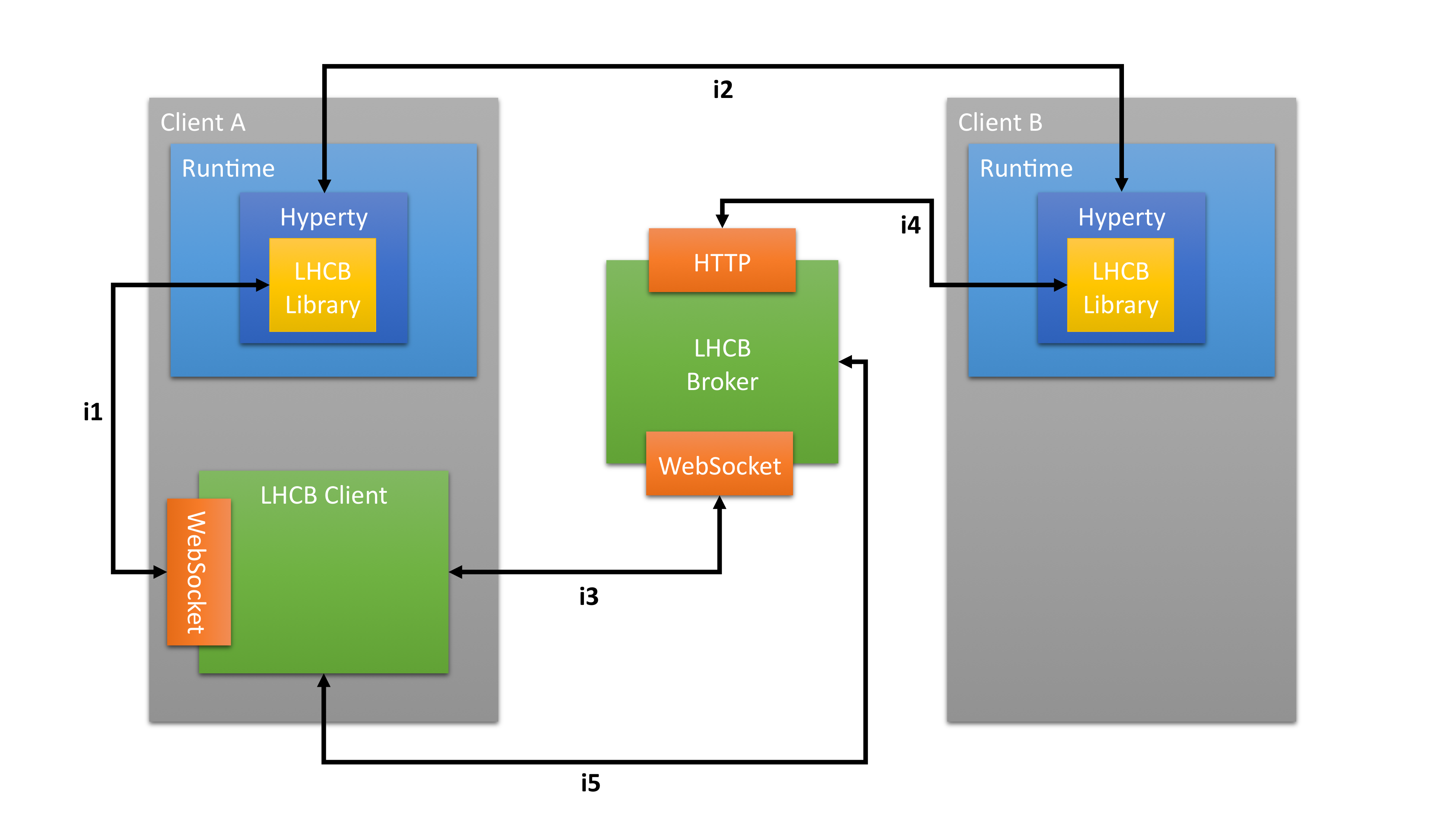
• Step 0: Offline CSP’s provisioning for a certain amount of a data : audio, video, throughput datachannel. The subscriber (CSP) is assigned a unique identifier cspID, it will provide to its clients so they can later be associated to the subscriber by the broker.  
• Step 1: The client retrieves from its CSP a communication Web App (including cspID).  
• Step 2: The Web App asks the broker information about the TURN server to use (IP address). If the data quota reserved is exceeded, the broker returns an error. **clientID = getTurnAddr(cspID, mediaType, clientName);**  
• Step 3: The Web App asks the broker the login information (credentials, username / password ...) to provide to the TURN server. **credentials = getTurnCredentials(clientID);**  
• Step 4: The client initiates the communication WebRTC via the TURN server.  
• Step 5: The TURN server sends periodically to the broker, information on data volume consumed through Redis Database (pub/sub mechanism).  
• Step 6: If the data quota reserved by the subscriber is exceeded, the broker sends the server TURN a command to interrupt the current communication.

The application is using a Quality of Service Broker, that manages a fleet of TURN servers available. The application has to register to the QoS Broker to be able to get the best TURN server dedicated to a user, regarding parameters (this registration is done ofline). Then, when a user is setting up a call, the runtime gets, from the broker, the best TURN candidate, with the good credentials to authorize the usage of TURN. If the operation is ok, it will benefit from quality of service, and the router will manage traffic regarding the traffic management implemented.

#### Last Hop Connectivity Broker

The Last Hop Connectivity Broker (LHCB) aims at providing information about available, alternative (wireless) uplinks of an end-device and the associated current quality of service. Such information is provided within the reTHINK framework to both, the global QoS reTHINK mechenism provided by the TURN services on an end-to-end scale, and to Hyperties and applications running on the client and wishing to access link information about their current connectivity. As such, the LHCB provides a supporting means to ensure and retrieve information about the current quality of service.

Besides providing only information about a single, i.e. the local end device, the LHCB's architecture allows for monitoring QoS indicators for a set of devices. The following figure illustrates the overall LHCB architecture as implemented in reThINK.



The LHCB Client is an entity running at the end-device and providing low-system-level access to network interface drivers. Due to the complexity and close integration with the hosting operating system, the LHCB Client is provided in parallel to the client's runtime environment.

Within the runtime, a LHCB library is provided that allows to transparently access the LHCB Client -- which is itself operating system depending -- via the web-socket-based interface *i1*. The latter interface provides means to retrieve and set information about the QoS status of all available (local) interfaces.

In addition, the LHCB arichitecture allows to provide information about a client's last hop connectivity QoS to other clients (i.e. communication partners using the reTHINK hyperty framework) or components in the network's backbonde, such as the reTHINK TURN service. For that, every LHCB Client registers itself at a LHCB Broker service via interface *i3*, which is also used to regularly update the clients QoS and link information at the broker. Note that the LHCB Broker may manage a set of clients. The LHCB Broker exposes a public http-based interface *i4* that allow for retrieving information about the QoS status about each client registered at the LHCB Broker. In order to guarantee privacy, a randomized globally unique identifier that maps to a client's identity has to be known to retrieve these information from the LHCB Broker; this identifier has to be retireved directly from the client via interface *i2* thus leaving the control at the user about which information about available links and QoS to share with whom. Hence, in the figure above, "Client B" may be exchanges with the reTHINK TURN service if the latter requests LHC QoS information via interfaces *i4* and *i2*.

The auxiliary interface *i3* is only required as generalized deployment of the LHCB service allow to deploy the LHCB Broker "as a service" in data centers which put the running LHCB instance behind firewalls and IP proxies. For such a deployment, the public IP address under which the LHCB Broker is accessible has to be know, e.g. by Client B, to contact the LHCB Broker. This information is provided by the client -- here Client A -- about which Client B want to retrieve link QoS information via interface *i2*. The local LHCB Library retrieves this information via *i1* from the LHCB Client which in turn retrieves it once via *i3* directly from the broker. This generalized approach also supports a set-up in which several, different LHCB broker handle diffent users, i.e., a globally / domain-specific unique LHCB broker is not required.

An example dynamic view on the interaction of components is provided [here](../dynamic-view/qos/readme.md).

### Interface of reTHINK QoS support with the Runtime QoS Agent

*specify the interface of the Runtime QoS Agent here. Currently, only the DIRECT interface of the Runtime QoS Agent with other QoS components is in scope. This section might actually not appear in this md-file but instead* [*here*](../dynamic-view/qos/readme.md) *with MSC diagrams and the specification of messages would be provided* [*here*](../messages/qos-management-messages.md)  
***This is internal use of the QoS, this has to be designed to be integrated in the runtime. The QoS Agent seems to be the place, even if the logic is rather thin in this runtime.***

The Quality of service depends of the ICE candidate setup. During the buildPeerConnection, the application has to call two methods, and provide its application identifier. This can be implemented in the QoS Agent.

https://broker\_URL/getAppropriateTurn  
{  
 cspId: applicationclientID,  
 clientName:"RealTimeVideoCall"   
}

that returns:

["TURN\_IP:PORT","identifier"]

if the application still benefits of the TURN service, otherwise it returns “Unable to get a turn server : data consumption exceeded”

Then second call should be:

https://broker\_URL/getCredentials  
{  
 clientId : “identifier”  
}

that returns:

{"clientId":"identifier","password":"xxxx"}

The resulting object should be:

RTCPeerConfiguration = {  
 iceServers: [  
 {  
 urls: 'turn:'+ TURN\_IP:PORT  
 }  
 ],  
 iceTransportPolicy: "relay"  
 }

This object is then used to build the peerconnection with the constraint {'googDscp': true}.

Example of code:

function buildPeerConstraints(useQos) {  
 var options = [];  
 options.push({'DtlsSrtpKeyAgreement': 'true'});  
 // We add DSCP support in case of QoS needs  
 if (useQos)  
 {  
 options.push({'googDscp': true}); // for QoS  
 }  
 return {optional: options};  
 }

To build the PeerConnection:

if (useQoS)  
 {  
 var turnServer;  
 var RTCPeerConfiguration;  
 ///  
 $.ajax({  
 url:settingsJS.brokerGetTurnURL,  
 type:'get',  
 async:false,  
 data:  
 {  
 cspId: settingsJS.applicationclientID,  
 clientName:"RealTimeVideoCall"   
 },  
 success:function(data, status){  
 console.log('done - buildPeerConnection: ');  
 console.log(data);  
 turnServer = {  
 urls: 'turn:'+data[0],  
 };  
 $.ajax({  
 url:settingsJS.brokerGetCredsURL,  
 type:'get',  
 async:false,  
 data:{   
 clientId : data[1]  
 },  
 success:function(data,status){  
 turnServer.username = data.clientId;  
 turnServer.credential = data.password;  
 console.log(turnServer);  
 RTCPeerConfiguration = {  
 iceServers: [  
 turnServer  
 ],  
 iceTransportPolicy: "relay"  
 };  
 console.log('turnServer :');  
 console.log(turnServer);  
 console.log("RTCPeerConfiguration"+RTCPeerConfiguration);  
  
 //installICE(RTCPeerConfiguration);  
 iceConfig = RTCPeerConfiguration;  
 console.log('iceConfig :');  
 console.log(iceConfig);  
 }  
 })},  
 error:function(err){  
 //Only 404 supported now  
 console.error("Unable to get appropriate turn:" + err);  
 }  
 });  
 }   
  
 console.log(iceConfig);  
 if (self.debugPrinter) {  
 self.debugPrinter("building peer connection to " + otherUser);  
 }  
  
 try {  
 pc = self.createRTCPeerConnection(iceConfig, buildPeerConstraints());

**Note: the final spec should be included (refined) in the dedicated QoS deliverable**