Understand WebSocket Connection for Finesse

Contents

Introduction

This document describes the WebSocket connection completely so that, during troubleshooting, the underlying processes are thoroughly understood.

Prerequisites

Requirements

There are no specific requirements for this document.

Components Used

The information in this document is based on these software and hardware versions:

- Cisco Finesse
- UCCX

The information in this document was created from the devices in a specific lab environment. All of the devices used in this document started with a cleared (default) configuration. If your network is live, ensure that you understand the potential impact of any command.

Background Information

Web Socket is a persistent connection between the client and the server.

Web Socket

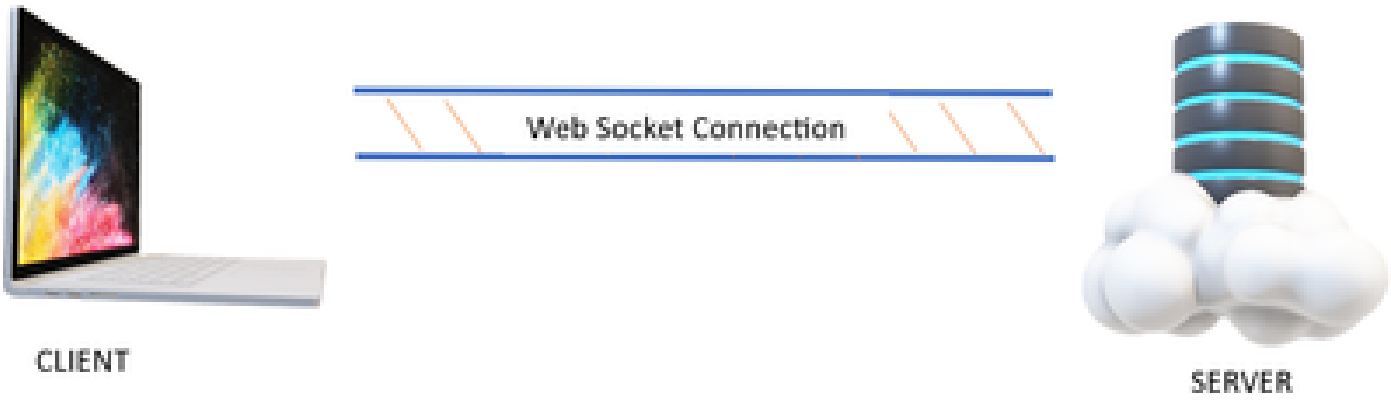
What is meant by the term persistent connection?

It means that once the connection between the client and the server is established, then client and server can send and/or receive data at any time.

This is a bi-directional full duplex connection.

The server does not have to wait for the client request to push back any data.

Similarly, the client also does not have to create a new connection every time to send any new data to the server.

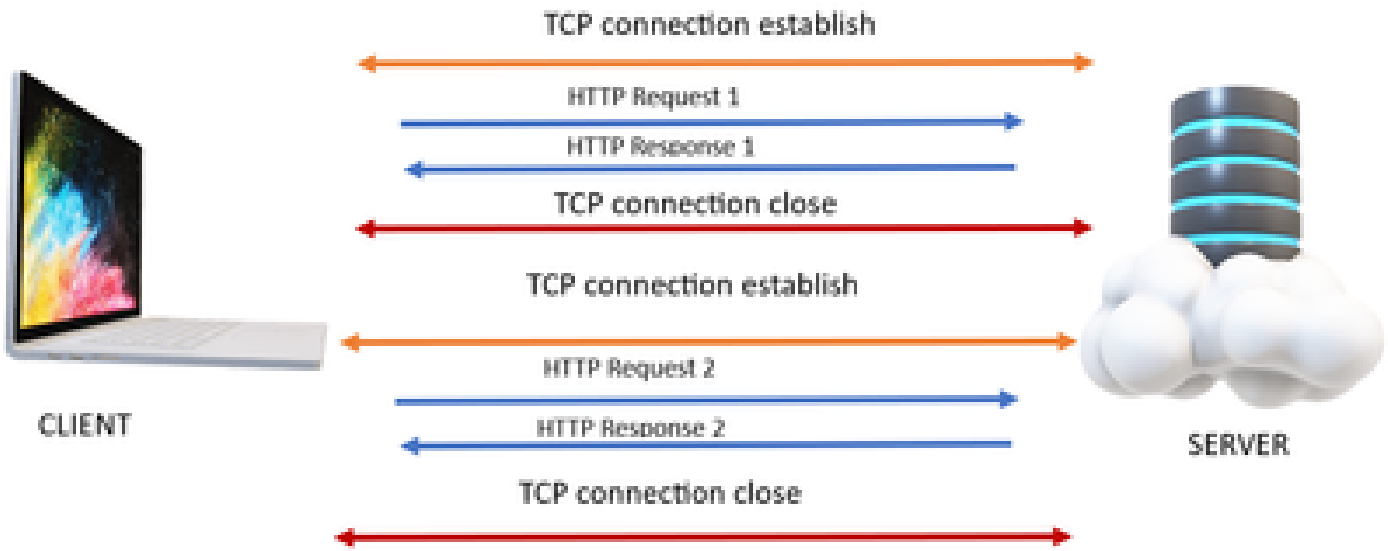


Web Socket connection is mainly used in the applications where real time data updates are required.

For example, Stock Trading apps, Messaging apps, and in our case, **Cisco Finesse**.

How do WebSockets Work?

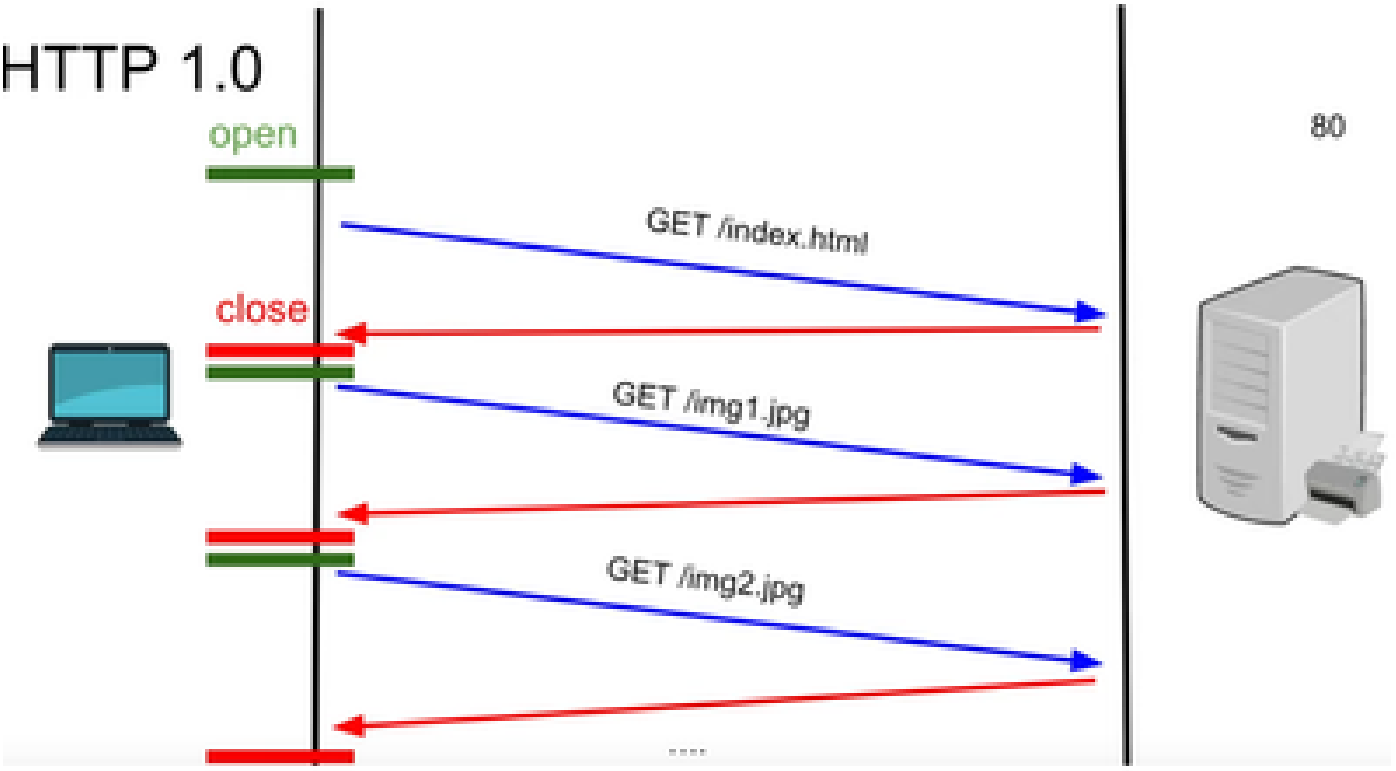
Consider:



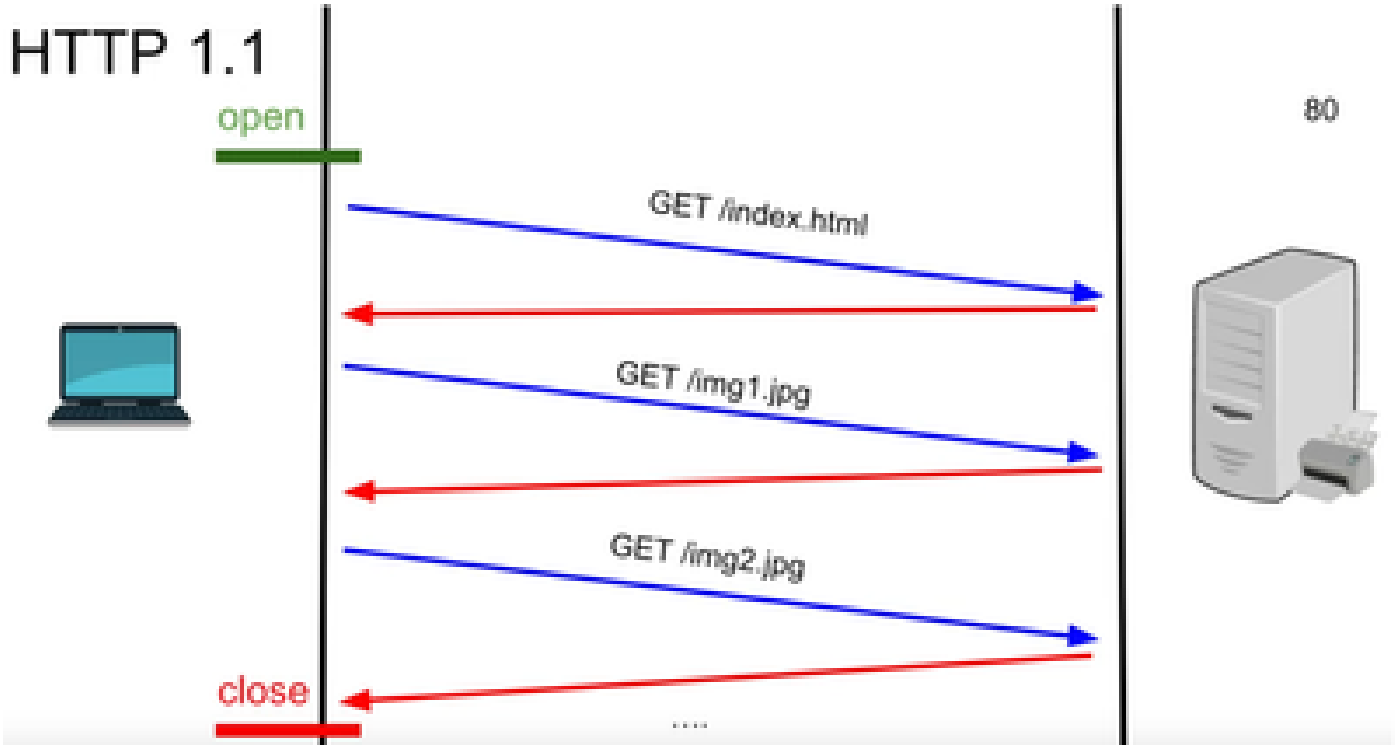
HTTP

1. The TCP connection (3-way handshake) takes place.
2. Then client sends HTTP request.
3. Server sends HTTP response.

4. After one request response cycle, the TCP connection closes.
5. For a new HTTP request, again, TCP connection establishes first.



HTTP 1.0 - After every request response, the TCP handshake starts again for another HTTP request response.



HTTP 1.1 - This connection worked because you could send and receive data and then close the connection.

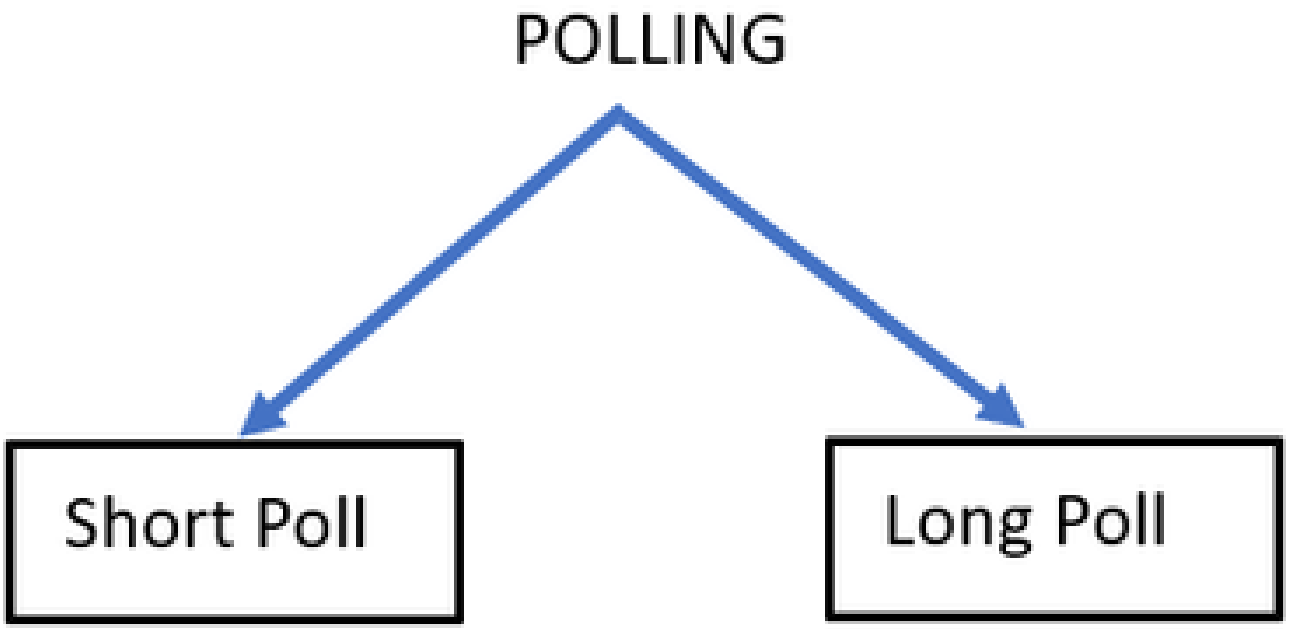
Again, this was not suitable for real time apps because the server can send some data even when the client is not requesting it. Therefore, this model is not effective.

Problem with HTTP

The problem starts with the real time systems.

For a website which requires real time updates, it is very difficult to send HTTP requests every time to get an update from the server and it uses a lot of bandwidth and causes overload.

To solve this, a mechanism of HTTP called Polling is used.



Short Poll – The is implemented when a short fixed timer is set for the requests and responses. For example, 05 seconds or 1 second depending upon the implementation.

If there is no update from the other side, then you can get empty responses in that timeframe that can waste resources.

Long poll – It somehow overcomes the short poll, but still has a fixed time to wait for a response.

If there is no response in that timeframe which is relatively longer than short poll but still it is fixed, then again request timeouts.

Therefore, Polling is not the best way to overcome this problem.

For this, the other method to use is called SSE.

SSE

Server Sent Events

In this, there is a unidirectional connection between the server and the client by which the server can send the data to the client at any point.

The thing to note here is that it is a unidirectional connection which means that only the server can send the data to the client and not the other way around.

An example of a use case is: Bulk notifications or updates from a server to client. For example, news live updates, Instagram live, and so on.

This is not very effective for applications involving real time updates and messaging.

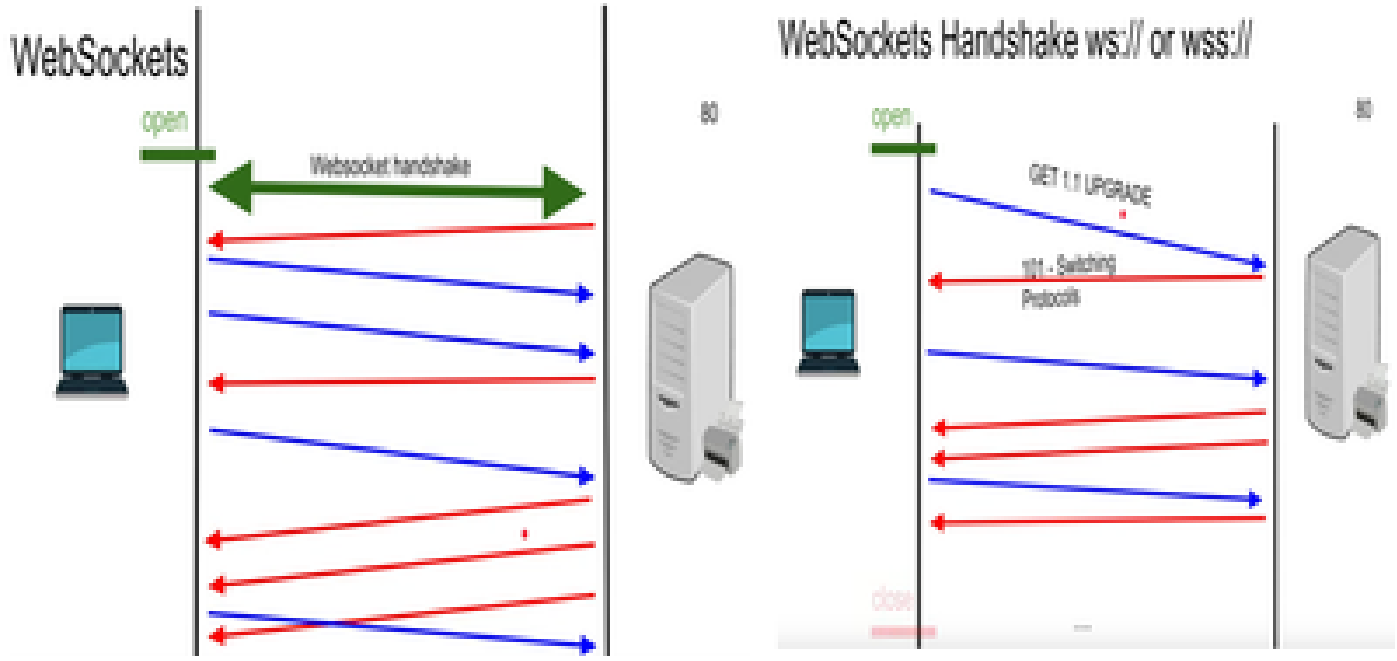
The Web Socket Connection is a persistent bi-directional full duplex connection.

This could be a phone call between a server and a client in which any party can talk to the other party at any time.

WebSocket Actions

1. To establish a websocket connection, the client sends an HTTP handshake request with an upgraded or updated header.
    1. This means that the client is saying to the server that right now, this is over HTTP but from now on, it moves onto the websocket connection.
2. The server then responds with HTTP 101 response which means sit is witching the protocol response.
3. After this, the websocket connection is established.

Now the server and client can use that same connection to transfer data to each other at any time.



WebSocket Debugs

If at this point you log into the Finesse client and see the network debugs, it displays as:

METHOD – GET
Domain – SERVER NAME

FILE – /WS/

INITIATOR – Openfire.js – websocket

Examining request and response:

Request

GET

Scheme: wss

host : uccxpub.prabhat.com:8445

filename : /ws/

Address: IP of the uccx server

Status: 101

Switching Protocols

VersionHTTP/1.1

RESPONSE HEADER

Connection: upgrade

Upgrade: WebSocket

Request – open <open xmlns='urn:ietf:params:xml:ns:xmpp-framing' to='uccxpub.prabhat.com' version='1.0'/> Response <open from='uccxpub.prabhat.com' id='5wfpqp1aly' xmlns='urn:ietf:params:xml:ns:xmpp-framing' xml:lang='en' version='1.0'/> <stream:features xmlns:stream='http://etherx.jabber.org/streams'><mechanisms xmlns="urn:ietf:params:xml:ns:xmpp-sasl"><mechanism>PLAIN</mechanism></mechanisms><c xmlns="http://jabber.org/protocol/caps" hash="sha-1" node="https://www.igniterealtime.org/projects/openfire/" ver="k3mOuil8afx3OTZxYy6yxLmFsok="/></stream:features> Request - auth <auth xmlns='urn:ietf:params:xml:ns:xmpp-sasl' mechanism='PLAIN'>YWRtaW5pc3RyYXRvckB1Y2N4cHViLnByYWJoYXQuY29tAGFkbWluaXN0cmF0b3IAMTIzNA==</auth> Response <success xmlns="urn:ietf:params:xml:ns:xmpp-sasl"/> Request – XMPP Bind Bind request to Bind the resource which in this case is desktop with a jabber id <iq type='set' id='_bind_auth_2' xmlns='jabber:client'><bind xmlns='urn:ietf:params:xml:ns:xmpp-bind'><resource>desktop</resource></bind></iq> Response – XMPP Bind where User ID is given a jabber id <bind xmlns="urn:ietf:params:xml:ns:xmpp bind"><jid>administrator@uccxpub.prabhat.com/desktop</jid></bind></ <iq xmlns="jabber:client" type="result" id="_bind_auth_2" to="uccxpub.prabhat.com/5wfpqp1aly"><bind xmlns="urn:ietf:params:xml:ns:xmpp-bind"><jid>administrator@uccxpub.prabhat.com/desktop</jid></bind></iq> Presence request <presence xmlns='jabber:client'/> Presence response <presence xmlns="jabber:client" from="administrator@uccxpub.prabhat.com/desktop" to="administrator@uccxpub.prabhat.com/desktop"/> <presence xmlns="jabber:client" id="liU1q-3" from="admin@uccxpub.prabhat.com/desktop" to="administrator@uccxpub.prabhat.com/desktop"/> <presence xmlns="jabber:client" id="P0Xxn-608" from="presencelistener@uccxpub.prabhat.com/web_framework" to="administrator@uccxpub.prabhat.com/desktop"><c xmlns="http://jabber.org/protocol/caps" hash="sha-1" node="http://www.igniterealtime.org/projects/smack" ver="NfJ3flI83zSdUDzCEICtbypursw="></c></presence> <presence xmlns="jabber:client" id="WhTJB-406" from="finesse@uccxpub.prabhat.com/web_framework" to="administrator@uccxpub.prabhat.com/desktop"><c xmlns="http://jabber.org/protocol/caps" hash="sha-1" node="http://www.igniterealtime.org/projects/smack" ver="NfJ3flI83zSdUDzCEICtbypursw="></c></presence> PUBSUB request – Requesting to subscribe the user to the pubsub node so that all the events on the user are monitored. <iq from='administrator@uccxpub.prabhat.com/desktop' to='pubsub.uccxpub.prabhat.com' type='set' id='3937d4d7-0046-4ad0-9d50-eebb3504e11e:subscribenode' xmlns='jabber:client'><pubsub xmlns='http://jabber.org/protocol/pubsub'><subscribe node='/finesse/api/User/Administrator/ClientLog' jid='administrator@uccxpub.prabhat.com/desktop'/></pubsub></iq> Response – user subscribed. <iq xmlns="jabber:client" type="result"

id="3937d4d7-0046-4ad0-9d50-eebb3504e11e:subscribenode" from="pubsub.uccxpub.prabhat.com" to="administrator@uccxpub.prabhat.com/desktop"><pubsub xmlns="http://jabber.org/protocol/pubsub"><subscription node="/finesse/api/User/Administrator/ClientLog" jid="administrator@uccxpub.prabhat.com/desktop" subscription="subscribed"><subscribe-options/></subscription></pubsub></iq> <iq xmlns="jabber:client" type="get" id="669-625" to="administrator@uccxpub.prabhat.com/desktop" from="uccxpub.prabhat.com"><query xmlns="jabber:iq:version"/></iq> PUBSUB request – Requesting to subscribe the Team to the pubsub node so that all the events on the team are monitored. <iq from='administrator@uccxpub.prabhat.com/desktop' to='pubsub.uccxpub.prabhat.com' type='set' id='14c3623e-d8cd-4272-8243-76bbcfcf193f:subscribenode' xmlns='jabber:client'><pubsub xmlns='http://jabber.org/protocol/pubsub'><subscribe node='/finesse/api/Team/1/Users' jid='administrator@uccxpub.prabhat.com/desktop'/></pubsub></iq> Response – Team subscribed <iq xmlns="jabber:client" type="result" id="14c3623e-d8cd-4272-8243-76bbcfcf193f:subscribenode" from="pubsub.uccxpub.prabhat.com" to="administrator@uccxpub.prabhat.com/desktop"><pubsub xmlns="http://jabber.org/protocol/pubsub"><subscription node="/finesse/api/Team/1/Users" jid="administrator@uccxpub.prabhat.com/desktop" subscription="subscribed"><subscribe-options/></subscription></pubsub></iq>

Related Information

- Cisco Technical Support & Downloads