IWPS Base Specification Draft

Version 0.3

Executive Summary

The Inter-World Portaling System (IWPS) is a foundational API specification that enables seamless interoperability between diverse virtual worlds, applications, and platforms. This document outlines the base specification for teleportation between interconnected worlds, providing a framework for building an “open metaverse” with additional features such as security, identity management, asset transfer, and payments.

The IWPS specification implements a highly flexible protocol, supporting different software architectures such as traditional cloud infrastructure, blockchain-based networks, and combinations of the two. This flexibility ensures IWPS can accommodate Web2 and Web3 implementations depending on the specific needs and capabilities of each virtual world.

The document defines two API calls—*Query* API and *Teleport* API,—which together facilitate the process of setting up and executing a teleportation between clients and their corresponding cloud services.

Key capabilities of IWPS include:

- Dynamic Teleportation: The specification enables users to seamlessly teleport between different worlds, either on the same device or across multiple devices. It supports both direct communication between clients and indirect communication through cloud services, depending on security and technical constraints.
- Secure and Trustworthy Communication: The specification incorporates multiple security profiles, from basic internet security to advanced standards set by OMA3, ensuring trusted interactions between clients and services.
- Flexible Identity and Asset Management: IWPS supports various user identity formats, including traditional usernames, email addresses, and blockchain addresses. It also enables the transfer of digital assets, such as avatars and items, across different worlds. It provides a protocol for negotiating transfers of assets with a different look and feel than what is found in the destination world.
- Extensible Design: IWPS is designed to be extended with additional frameworks, such as user agents, , and payments, making it a versatile platform for evolving virtual world requirements.

Contents

1 Introduction

1.1 Background

For more background on IWPS, the reader is referred to IWPS Position Paper [6]

1.2 General Considerations

This specification meets the requirements as specified in IWPS RFP [7]

1.3 Scope

This is the base API specification for IWPS and is enough to implement a test network of interconnected worlds. Future documents will go into more detail on features that augment the base specifications in areas such as security, user agents, identity, asset transfer, and payments.

2 References

2.1 Normative References

[1] GraphQL, https://spec.graphql.org/October2021/

[2] IETF RFC 3986: "Uniform Resource Identifier (URI): Generic Syntax", January 2005, https://www.rfc-editor.org/rfc/rfc3986.

[3] IETF RFC 8446: "The Transport Layer Security (TLS) Protocol Version 1.3", August 2018, https://www.rfc-editor.org/rfc/rfc8446.txt

[4] IETF RFC 9110: "HTTP Semantics", June 2022, https://www.rfc-editor.org/rfc/rfc9110.txt

2.2 Informative References

[5] W3C "Naming and Addressing: URIs, URLs, ... ", https://www.w3.org/Addressing.

[6] OMA3: “Portals Unleashed- Creating a teleportation standard to connect the metaverse”, https://github.com/oma3dao/portal-position-paper

[7] OMA3: “Inter World Portaling System (IWPS) Request for Proposals”, https://github.com/oma3dao/iwps-rfp

[8] NSA, Eliminating Obsolete Transport Layer Security (TLS) Protocol Configurations, https://media.defense.gov/2021/Jan/05/2002560140/-1/- 10 1/0/ELIMINATING_OBSOLETE_TLS_UOO197443-20.PDF

[9] IETF Deprecation of IKEv1 and obsoleted algorithms. 12 https://datatracker.ietf.org/doc/rfc9395

3 Definitions and Abbreviations

3.1 Modal Verbs Terminology

This section contains the established conventions for the use of dedicated terms, describing normative and informative requirements. These terms have a specific meaning, as defined below. These are based on the definitions in the following organizations:

- IETF, as defined in RFC 2119,
- ETSI, as defined in Section 3.2 of the ETSI Drafting Rules, and
- Bluetooth SIG.

In the present document "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "MAY", "NEED NOT" are used to express normative requirements within the scope of the given specification.

SHALL	The term "SHALL" defines a *mandatory* requirement. The course of action expressed in the requirement is to be strictly followed to conform to the specification. Deviations are not permitted.
SHALL NOT	The term "SHALL NOT" expresses the negative form of "SHALL". The course of action expressed in the requirement is *prohibited*. The requirement is to be followed to conform to the specification. Deviations are not permitted.
SHOULD	The term "SHOULD" defines a *recommended* requirement. While several other possibilities exist, it is strongly recommended to follow the requirement, but not necessarily required. Valid reasons can exist to not follow the requirement, but the full implications are to be understood and carefully weighed before choosing a different course.
SHOULD NOT	The term "SHOULD NOT" expresses the negative form of "SHOULD". While the course of action expressed in the requirement is *deprecated*, it is not forbidden.
MAY	The term "MAY" defines a truly *optional* requirement. The course of action expressed in the requirement is permissible within the scope of the specification. The implementor freely decides whether to implement an optional requirement. An implementation, which does not include an optional requirement, has to be prepared to interoperate with another implementation,

	which includes the option. Similarly, an implementation, which includes an optional requirement, has to be prepared to interoperate with another implementation, which does not include the option.
NEED NOT	The term "NEED NOT" expresses the negative form of "MAY".

In the present document terms like "can", "will", "will not", "is/are", "is/are not" are used to express requirements, which are informative within the scope of the given specification. These terms are not used to describe a normative *mandatory*, *recommended* or *optional* requirement. The following definitions of informative terms are meant to help the reader to better understand the intent of the present document.

Must	The term "must" expresses a *natural consequence* or states an *indisputable statement of fact*.
Can	The term "can" expresses a *possibility* and *capability* when conforming to the specification.
Will	The term "will" expresses an inevitable behavior, which is outside the scope of this specification.
Will not	The term "will not" expresses the negative form of "will".
Is/are	The term "is" or "are" or any other verb in indicative mode and present tense expresses a *statement of fact*.
Is not/are not	The term "is not" or "are not" expresses the negative form of "is" or "are".

When the present specification shows one of many alternatives to satisfy a specification requirement, the alternative shown is not intended to limit other acceptable implementation options, which also satisfy the specification requirement.

Requirements in tables are defined as "Mandatory" (M), "Optional" (O), "Excluded" (X), “Not Applicable” (N/A), or "Conditional" (C.n). Conditional statements (C.n) are listed directly below the table in which they appear.

"Conditional" (C.n). Conditional statements (C.n) are listed directly below the table in which they appear.If conformance to the present specification is claimed, all normative capabilities indicated as mandatory within the scope of the present specification shall be supported in the specified manner. This also applies for all optional and recommended capabilities for which support is indicated. If an optional or recommended capability is implemented, it shall be implemented as specified.

Where a field in a data structure is described as "Reserved for Future Use", the device creating the structure SHALL either exclude it or set its value to zero, unless otherwise

specified. Any device receiving or interpreting the data structure SHALL ignore that field. The receiving device SHALL NOT reject the data structure because of the presence of the field or its value.

3.2 Abbreviations

In the present document, the following abbreviations apply:

DNS	Domain Name System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ID	Identifier
TTL	Time To Live
URI	Uniform Resource Identifier
UX	User Experience
IWPS	Inter World Portaling System

3.3 Definitions

The following definitions are used within the present document.

Term	Definition
Application	A self-contained software application on a Device.
Asset	A digital asset owned by the User, such as an avatar or mount.
Client	A piece of software that resides on a Device that a World uses to display the User and Portal. A Client can be an Application, or it can be software that runs in an Application.
Cloud	Backend infrastructure of a Service.
Deeplink	A mechanism, triggered by a URI, that allows an Application to be automatically launched from another Application: https://en.wikipedia.org/wiki/Deep_linking
Destination Client	The Client the User arrives in at the end of a teleportation.
Destination Cloud	Backend infrastructure that interacts with the Destination Client
Destination Service	Combination of Destination Client and Destination Cloud.

Term	Definition
Destination URI	URI that the Source Portal will use to connect the User to the Destination Service with an *Activate* API call
Device	A unique piece of hardware with at least one operating system that runs Applications
Local Teleport	Teleport between Worlds within the same Application
Remote Teleport	Teleport between Worlds on different Devices.
Service	A provider or host of Worlds.
Source Client	The Client the User starts from within a teleportation.
Source Cloud	Backend infrastructure that interacts with the Source Client
Source Service	Combination of Source Client and Source Cloud.
Portal	UX feature on a Source Client that allows a User to Teleport or Warp a User to a Destination Client.
Portal URI	URI that the User uses to create a Portal, optionally including parameters such as the landing location in the Destination World.
Teleport	A User's move between different Clients.
User	The person who participates in a Service. In some Services the representation of a User in the World is an avatar, which is an Asset for the purposes of this specification.
User Agent	A Client or other Application which acts on behalf of the User.
World	A virtual world (https://en.wikipedia.org/wiki/Virtual_world)

3.4 Taxonomy

The following taxonomy is used within the present document.

Taxonomy	Definition
`[square brackets]`	Indicate a placeholder for a specific argument or option.
`<angle brackets>`	Indicate an optional element.
name	Indicates an element, object or variable name.

4 Introduction

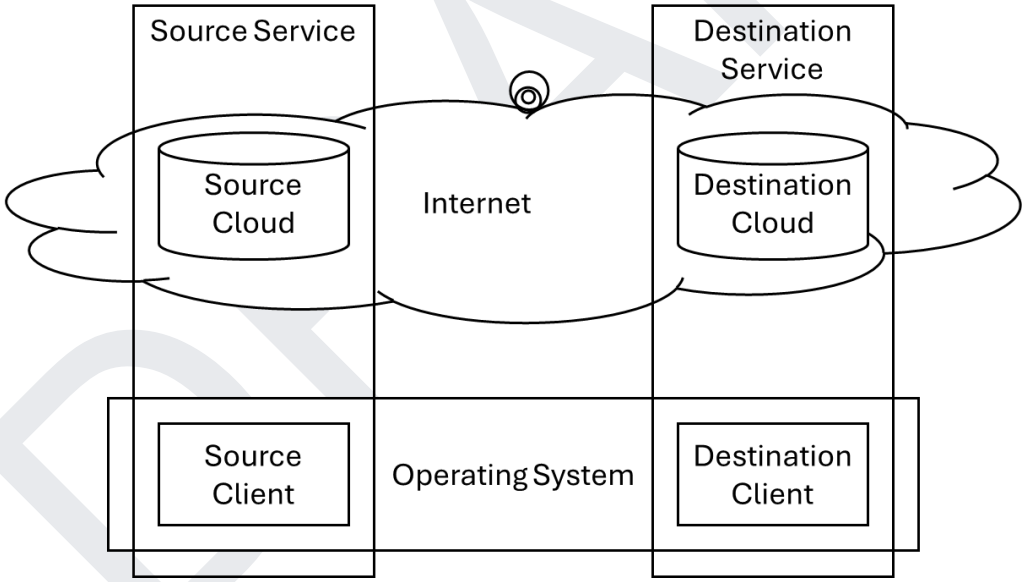
4.1 Actors

Within the present document, a Client is defined as a piece of software that resides on a Device, which is a unique piece of hardware with at least an operating system able to run Applications. A virtual World uses the Client to display the User and a Portal.

The software that controls each World can be described as a Service, and each Service has two main components: the Client, which is software that resides on a User's Device, and the Cloud, which is software that is not on a User's Device.

Figure 1 shows the main software actors that interact to teleport a User from one Client to another. During a Teleport, the User starts from the Source Client and arrives in the Destination Client at the end of the Teleport. The UX feature on the Source Client that allows a User to Teleport is defined as the Portal

Figure 1: Overview of Actors



The Source Cloud is the backend infrastructure that interacts with the Source Client. Together they provide the Source Service. Similarly, the Destination Cloud is the backend infrastructure that interacts with the Destination Client. Together they provide the Destination Service.

Direct secure communications between the Source Client and the Destination Client may not be possible. Communication mechanisms are operating system dependent and

may not be available. Therefore, the Source Service and Destination Cloud need to negotiate the most secure communication path to use for the teleportation. If the communication path between the Source Client and Destination Client is secure and reliable it will be used for the teleportation. If not, the Source Client may need to communicate with the Destination Client through a Cloud backend, to trigger a Teleport between the two.

4.2 Assumptions

4.2.1 Communication Assumptions

The following assumptions are made regarding the communication capabilities between the Source and Destination actors:

- Any Client can initiate communication with any Cloud.
- The Cloud can initiate communication with the Client belonging to the same Service (note- such communications may be enabled by Client polling).
- A Cloud may not be able to initiate communications with the other Cloud (e.g.- Service to Service Cloud communications).
- A Client may not be able to communicate reliably with the other Client.

Messages between the different Services may be HTTP or HTTPS calls. The URI of the receiving actor is known to the sending actor initiating the communication.

The DNS infrastructure to resolve the URI is assumed to be reliable and trustworthy.

4.2.2 Client Implementation Assumptions

The present document does not make any assumptions about the implementation of the Client. Nevertheless, it is assumed the Client has the following information about its implementation:

- The Client is aware of the Operating System it is installed on.
- The Client is aware of the User using the Client to access the World.
- The Client is aware of the type of implementation, like a browser application, a desktop application, or a mobile application.

4.2.3 Cloud Implementation Assumptions

The present document does not make any assumptions about the implementation of the Cloud. However, the following implementations have been considered during the creation of this specification:

- Traditional: The Cloud consists of one or more servers completely controlled by the World and hosted on premises and/or in public cloud infrastructure. The servers can use any number of software stacks to receive and initiate communications, and store data.
- Blockchain: The Cloud is one or more smart contracts running on one or more blockchain networks. The World can control the smart contracts, but not the nodes running the blockchain network, nor the software underlying the smart contracts. Smart contracts can accept communications but the protocol is limited by the node architecture. Smart contracts cannot initiate communications to servers outside of the blockchain network.
- Hybrid: The Cloud is a combination of Traditional and Blockchain infrastructure. One primary advantage of a Hybrid Cloud is it allows a Client to use a Blockchain with smart contracts for most functionality but also use a traditional Cloud to initiate communications or translate incoming API communications for the blockchain network.

4.2 API Overview

The present document specifies a set of base API calls, which are used to request and to execute a Teleport between two Clients. These are namely:

- *Query* API Call (see Section 5.3.2 for details)
- *Teleport* API Call (see Section 5.3.3 for details)

4.3 Extension Frameworks

IWPS also can integrate several additional frameworks to improve the basic user experience of IWPS:

- **Identity**: To allow users to log in to the Destination Client automatically with a consistent User ID across Worlds. See Section 5.5 for details.
- **Asset transfer**: To allow users to transfer their Assets to the new World, such as avatars, items, and even item characteristics. See Section 5.6 for details.

- **Look and feel**: To negotiate discrepancies between Worlds, such as bringing laser guns into medieval times. See Section 5.7 for details.
- **Payment**: See Section 5.8 for details.

5 Specification (Normative)

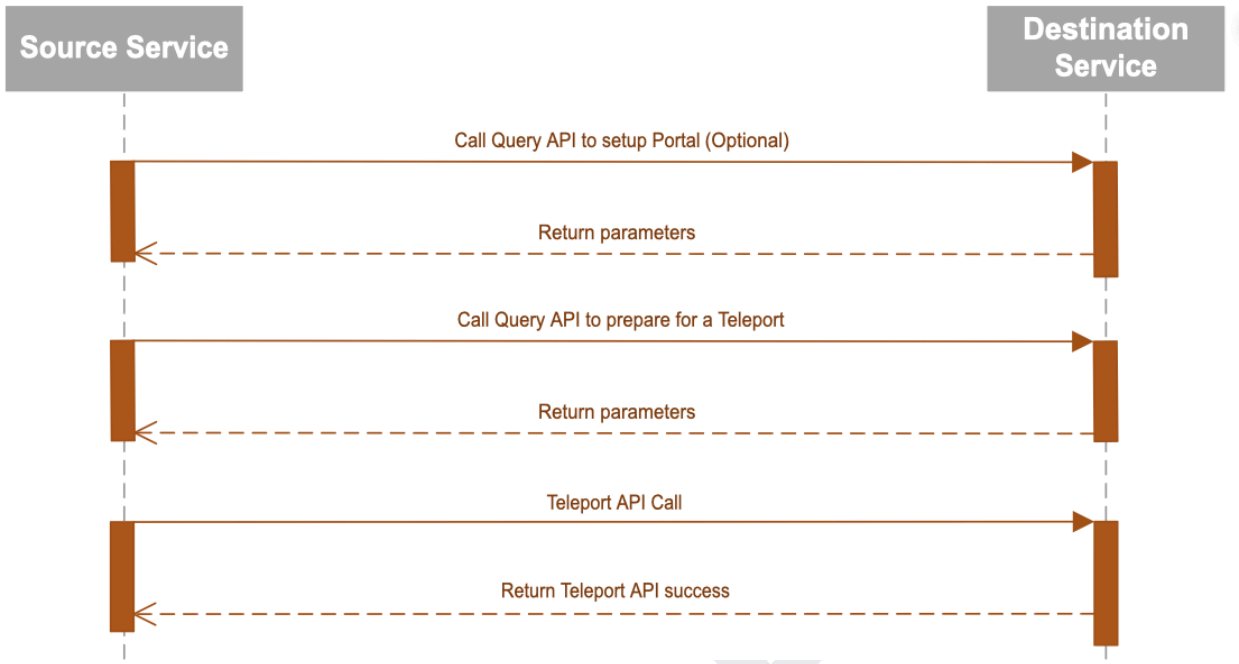
5.1 Mode of Operation

5.1.1 API Flow Overview

During a Teleport, a User moves from the Source Client through a Portal to the Destination Client. A Teleport is executed via two API calls from the Source Service to the Destination Service:

- Step 1: *Query* API: The Source Service (Client or Cloud) SHALL use a *Portal URI* to call the *Query* API on the Destination Cloud to set up or update a Portal's parameters (Section 5.3.2) or to get the status of the Destination World. The parameters returned by the *Query* API include the *Destination URI*, landing *location* in the Destination World, allowed Assets, etc. The *Query* API MAY be called multiple times before the *Teleport* API is called.
- Step 2: *Teleport* API: The Source Service SHALL use a *Destination URI* to call the *Teleport* API on Destination Service (Client or Cloud) to execute the Teleport. The communication path between the Services is determined by the *Query* API. The *Teleport* API launches the Destination Client, moves any included Assets once the User logs into the Destination Service, and confirms completion of the Teleport to the Source Service.

Figure 1: IWPS API Flow

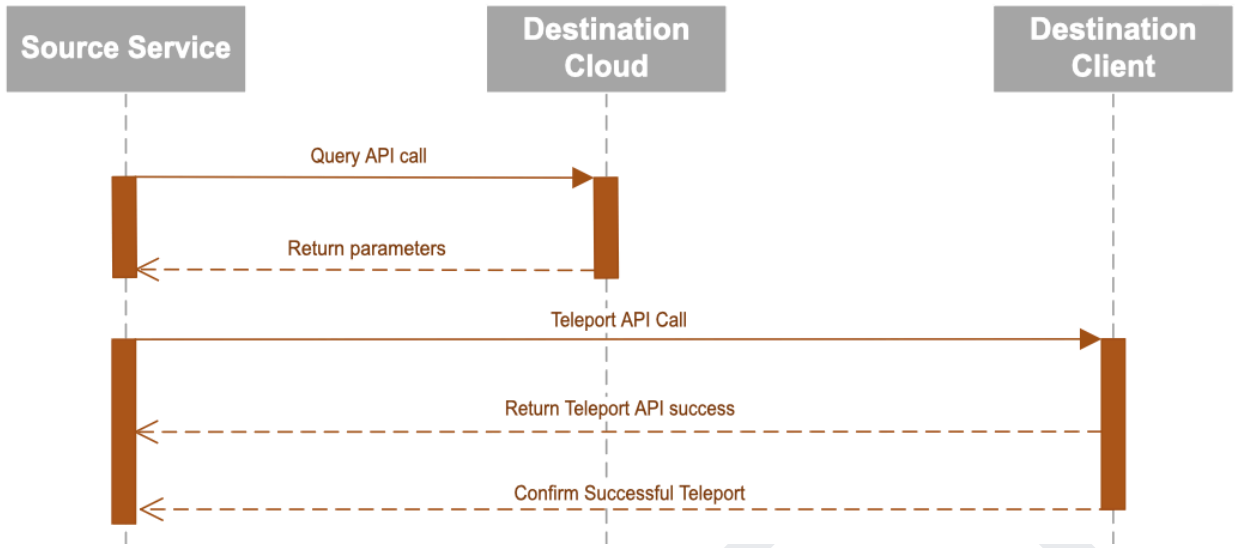


5.1.2 Teleport API Communication Options

Whereas the communication path for a *Query* API always terminates at the Destination Cloud, the communication path between Source and Destination for a *Teleport* API call could use one of the below defined mechanisms:

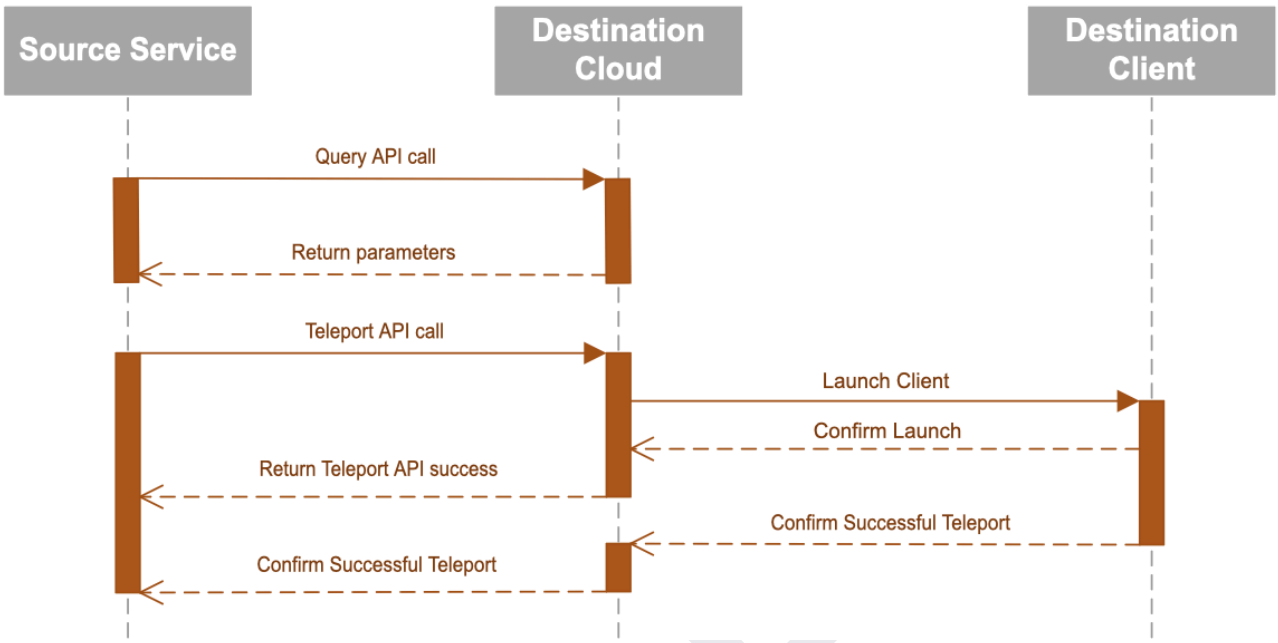
- **Device Communication**: The Source Client sends the *Teleport* API call directly to the Destination Client via a communication path on the same Device. The Destination Client returns a response back to the Source Client using the same communication path. This path must have the correct characteristics (see below) in order for it to be used.

Figure 2: Device Communications



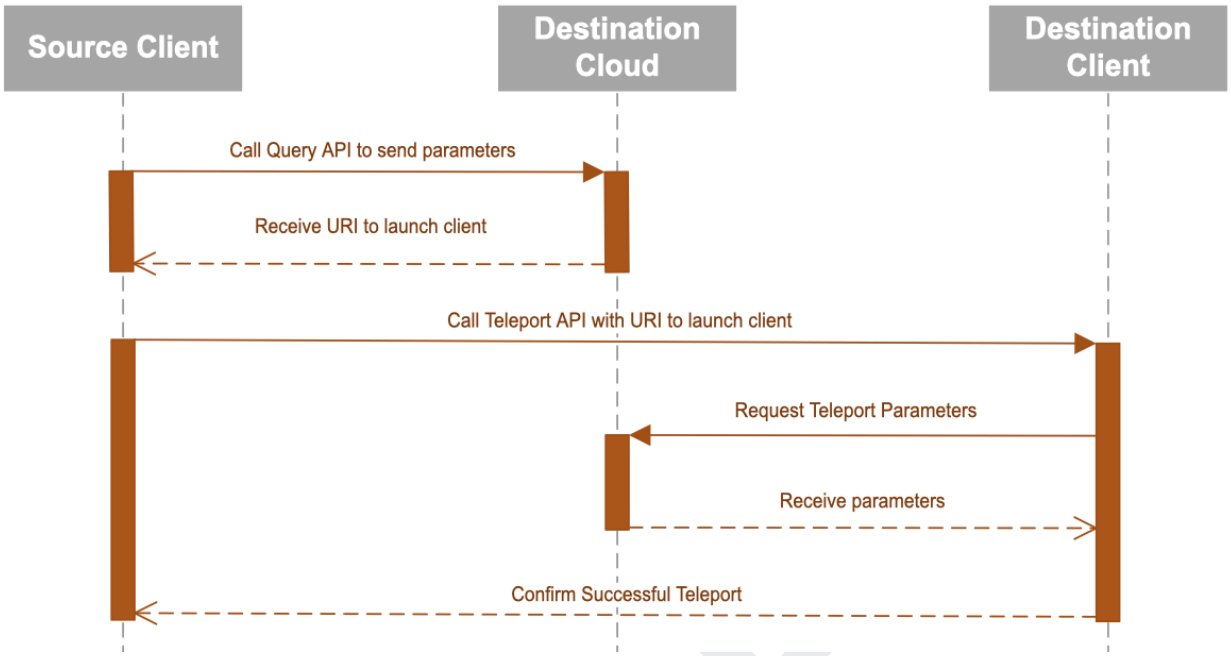
- **Cloud Communication**: This communication path is required when the Source and Destination Clients are on different devices but it can also be used if Clients are on the same Device. The Source Service (Client or Cloud) sends the *Teleport* API Call to the Destination Cloud. The Destination Cloud launches Destination Client and returns a response back to the Source Service. Details of how the Destination Cloud launches the Destination Client are implementation-dependent and outside the scope of the present specification. However, some examples are given in Section 6.1.

Figure 3: Cloud Communications



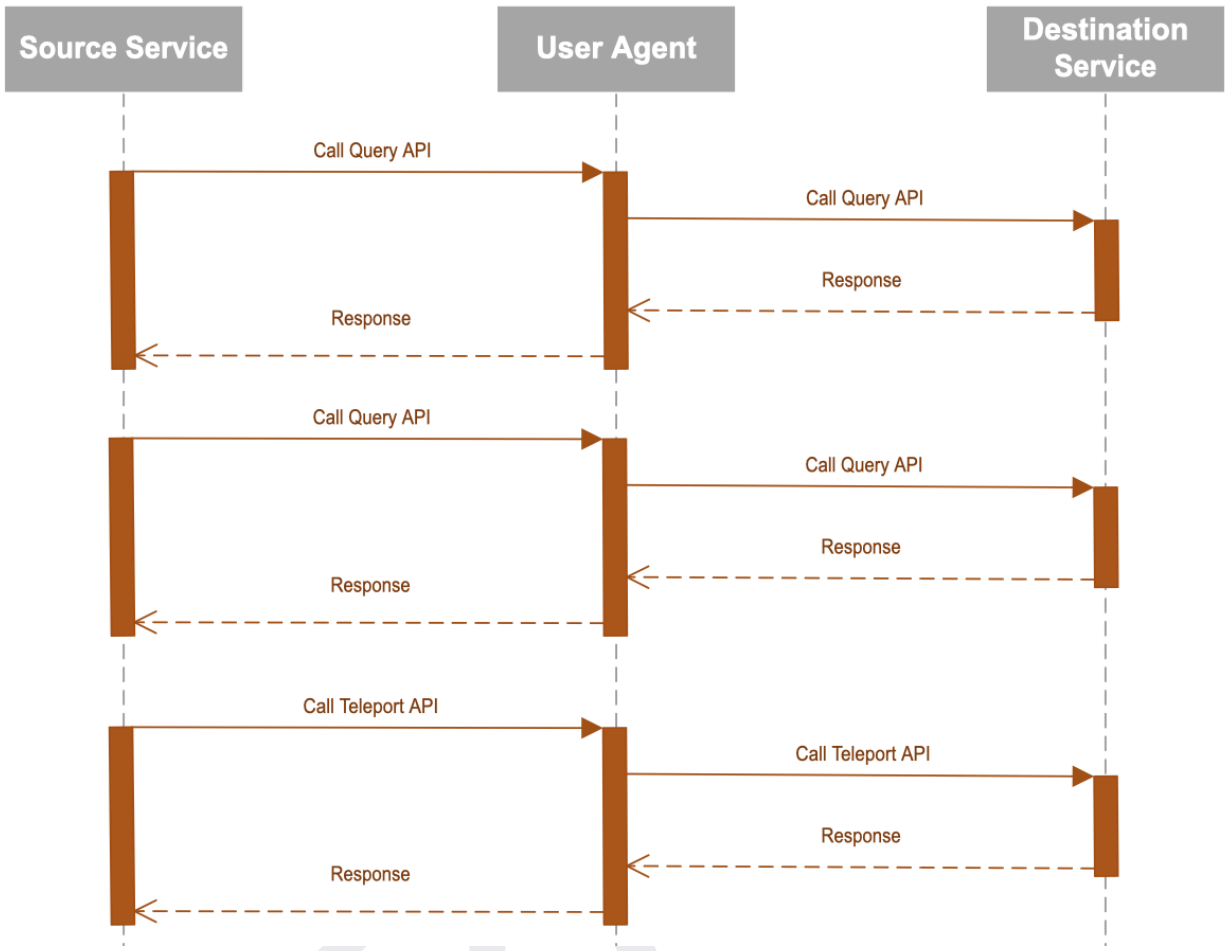
- **Hybrid Communication**: Hybrid Communications is used when the Device Communication path has a limitation that does not allow all the *Teleport* API parameters and response values to be communicated in a reliable manner, but the Destination Cloud still prefers to use the Device's OS communications to launch the Destination Client. In this case, before calling the *Teleport* API the Source Service leverages the *Query* API call to pre-negotiate the Teleport parameters that cannot be exchanged reliably using Device Communications. Once the *Query* API is complete the Source Client sends the *Teleport* API call directly to the Destination Client via device communications.

Figure 3: Hybrid Communications



- **User Agent Communication**: The Source Service sends *Query* and *Teleport* API calls to a User Agent. The API call parameters additionally include the final URI of the Destination Service. The User Agent then sends the API call to the URI of the Destination Service. The Destination Service returns a response back to the User Agent. The User Agent returns a response back to the Source Service. See Section 6.4 for details.

Figure 4: User Agent Communications



The Destination Service determines the communication path based on parameters sent by the Source Service in a *Query* API call. The factors in the decision include:

1 Which path is the most efficient?
2 Which path is trustworthy (e.g.- secure and reliable)?
3 Are the Clients on the same Device?

5.2 Uniform Resource Identifiers

An IWPS Uniform Resource Identifier (URI) is defined as a URI, which is used in the context of IWPS.

5.2.1 URI Formats

Any IWPS URI SHALL conform to RFC 3986 [2].

Note: The W3C standard for URI [4], refers to RFC 3986 as well.

Any IWPS URI SHALL be formatted according to the following as given below:

`[protocol]://[destination]`

The `[destination]` SHALL be of the form:

`[domain]/[location]?<parameters>`

Where `[domain]` is an internet domain, or an IPv4 or IPv6 address, `[location]` is a spatial position in the target world, and `<parameters>` are user, state or time-specific properties.

The `[protocol]` SHALL be of one of the below options:

`http`	HTTP protocol [4]
`https`	Use of the HTTP protocol [4] over TLS [3] .
`ws`	TBD

5.2.2 Portal URI

The *Portal URI* is the marketing IWPS link for a World.

The *Portal URI* SHALL be used to call the *Query* API, defined in Section 5.3.2. A *Portal URI* MAY come from many sources, including but not limited to:

- Directly from the World developer, through a website or a direct message from a representative of the World.
- From a referral, such as a friend sending the User a link to join the World.
- From an advertisement such as a QR code on a billboard or a TV commercial.

5.2.3 Destination URI

The *Destination URI* SHALL be used to call the *Teleport* API, defined in Section 5.3.3. It is returned as a response to a *Query* API call. It can also be updated with repeated *Query* API calls.

The *Destination URI* can come in many forms. It can be the address of a simple GraphQL call to a cloud endpoint, or it can be a deep link for a specific operating system using the operating system's inter-application or inter-process communication service.

If the User desires to utilize a User Agent as an intermediary between the Source and Destination, the *Destination URI* needs to be generated by the User Agent. The is done using these steps:

1. Submit a *Portal URI* to the User Agent and receive an updated *Portal URI*.
2. Use the updated *Portal URI* in a *Query* API call.
3. The *Query* API call will return a *Destination URI* that utilizes the User Agent.

5.3 Application Programming Interfaces

An IWPS Application Programming Interface (API) is defined as an API which is used in the context of IWPS.

5.3.1 API Formats

The content of all IWPS API calls SHALL be sent to the respective URI as a string. All API calls SHALL be formatted according to the *GraphQL* format, as defined in [1]. An API call can have properties, parameters or functions. Either of them can be mandatory or optional.

Therefore, the receiving URI endpoint SHALL be capable of accepting a string.

Note: The maximum acceptable length of a string is not defined

The response to IWPS API call SHALL be provided in JSON format and SHALL contain the supported requested properties from IWPS API call.

The *GraphQL* server behind the receiving URI NEED NOT support all queried fields. In case such a field is not supported, the field SHALL NOT be returned in the response.

5.3.2 Query API

The *Query* API call determines the Teleport parameters before the *Teleport* API is called. If the *Query* API call is made before a *Teleport* API call with a *teleport-id* parameter, the Destination Cloud MAY return *true* for *approval* and a positive *expiration* value to give a pre-approval to the *teleport-id* as long as the *Teleport* API is called within *expiration* milliseconds of calling the *Query* API.

The *Query* API can also be called without a *teleport-id* to pre-negotiate certain parameters of a Portal. For example, when a Portal is created the Source Service MAY call the *Query* API using the *Portal URI* to pre-negotiate aspects of the teleportation for a particular combination of Source Client, Destination Client, and Device, such as URIs, Asset transfers and user agreements. This allows Users to know what can be expected

from a Teleport before initiating the teleportation. In this case the information passed back from the *Query* API call can be cached by the Source Client. The *Query* API call can be initiated by a User, or the World itself upon installation of the Source Client on the Device.

The *Query* API has the following capabilities:

- Negotiate the communication path between Source and Destination.
- Calculate the *Destination URI.*
- Give the User a *Download URI* to download the Destination Client to the User's Device.
- Pre-negotiate look and feel rules, as well as other teleporting rules between two Worlds (such as coordinates of destination location).
- Get pre-approval for a pending Teleport.

A Source Service and a Destination Cloud SHALL support the *Query* API.

The parameters of the *Query* API call are defined in Table 1:

Table 1: Parameters for Query API Call

Parameter	Type	Description	Status
location	Text	Coordinates in a form determined by the Destination World	M
source-characteristics	Object	Reserved for Future Use	M
uri-portal	Text	Portal URI of Destination Cloud	O
source-isa	Enum	enum- [x86, arm]	A.1
source-bits	Enum	enum- [32, 64]	A.1
source-os	Enum	enum- [windows, macos, android, ios, ps, xbox, switch, etc.]	A.1
source-os-version	Float	Operating System version	A.1
source-client-type	Enum	enum- [browser, application]	A.1
teleport-id	Integer	128-bit integer randomly generated by the Source Client when Teleportation is initiated	A.2
user-id	Text	ID of the User in the Originating Service	A.2
teleport-pin	Integer	A 2-digit decimal number randomly generated by the Source Client when Teleportation is initiated	A.2

Parameter	Type	Description	Status
uri-source-ack	Text	URI (called by Destination Service if teleport is successful)	A.2
uri-source-nack	Text	URI (called by Destination Service if teleport is unsuccessful)	A.2
assets	Object	Reserved for Future Use	A.2
error	Text		O

A.1: These parameters SHALL be passed as a bundle.

A.2: These parameters SHALL be passed as a bundle.

The returned fields with the *Query* API Response are defined in Table 2:

Table 2: Parameters for Query API Response

Field	Type	Description	Status
approval	Boolean		M
location	Text	Approved landing location in the Destination.	M
uri-destination	URI	*Destination URI*	M
uri-download	URI	*Download URI*	A.1
portal-limitations	Object	Reserved for Future Use.	O
uri-portal	URI	Updated *Portal URI*	O
expiration	Integer	How many seconds the Portal is active for.	A.2
approved-assets	Object	Reserved for Future Use.	A.2
status	Object	Reserved for Future Use.	A.2

Use of the *Query* API during Portal creation without *teleport-id* is as follows:

1. The Source Service (Client, Cloud, or both) generates the A.1 *Query* parameters (see Table 1) and calls the *Query* API on the *Portal URI*.
2. When the Destination Cloud receives a *Query* API with mandatory and A.1 parameters it uses the *location*, *source-isa*, *source-bits*, *source-os*, *source-os-version*, *source-client-type*, and *source-characteristics* parameters to determine if the Teleport is allowed.
3. If the Teleport is not allowed (e.g.- there is no way to complete a successful Teleport), then the Destination Cloud SHALL return *false* for the *approval* return

value and an optional explanation in the *error* return value. The remaining steps are skipped.

4. The Destination Cloud uses the *source-characteristics* parameter to generate the *portal-limitations* objects.
5. The Destination Cloud uses the *source-isa*, *source-bits*, *source-os*, *source-os-version*, *source-client-type* parameters to determine *uri-destination*, *uri-launch*, and *uri-download*.
6. The Destination Cloud responds to the *Query* call with *true* for approval, along with *uri-destination*, *uri-warp*, *uri-download*, *location*, and *portal-limitations*.
7. If the Destination Cloud returns a different *location* value from what the Source Client requested in the *Query* call, the Source Client SHOULD give the User the option to not create the Portal.
8. If a *Download URI is* returned in *uri-download*, the Source Client MAY give the User the option to download the Destination Client.
9. The Source Client MAY cache *location*, *uri-destination*, *portal-limitations*, and *uri-launch* with the Portal for future use.
10. If the User calls uri-download the Query API SHOULD be called after the download to update the *Destination URI* and *Launch URI*.

Use of the *Query* API with *teleport-id* as a status check right before calling the *Teleport* API:

1. The Source Service (Client, Cloud, or both) generates the mandatory and A.2 *Query* parameters (see Table 1) and calls the *Query* API on the *Portal URI*.
2. When the Destination Cloud receives a *Query* API with A.2 parameters it uses the *teleport-id, user-id, source-characteristics,* and *assets* parameters to determine if the Teleport is allowed.
3. If the Teleport is not allowed (e.g.- the User is blocked), then the Destination Cloud SHALL return *false* for the *approval* return value and an optional explanation in the *error* return value. The remaining steps SHALL be skipped.
4. The Destination Cloud SHALL construct the *approved-assets* parameter from the *source-characteristics* and *assets* parameters and store them, along with other A.2 parameters, with *teleport-id*.

5. The Destination Cloud SHALL return the required parameters to the Source Service.

6. The Source Client calls the *Teleport URI* with *teleport-id* to initiate the Teleport.

A *Portal URI* SHALL point to a Destination Cloud server or a User Agent that will relay parameters to the Destination Cloud.

A Source Service and a Destination Cloud SHALL support the *Query* API.

Example *Query* API Calls and Responses are provided in Section 6.2.1.

5.3.3 Teleport API

The *Teleport* API call executes the Teleport. The call exchanges information between the Source Service and Destination Service, ultimately logging in the User to the Destination Client and closing the Source Client.

The *Destination URI*, obtained from the *Query* API, defines the endpoint the Source Service SHALL call to initiate the Teleport. Depending on the format of the *Destination URI* the call may originate from the Source Client or the Source Cloud.

The *Teleport* API has the following capabilities:

- Launch the Destination Client so that the User can log in.
- Give the User the option to use a User Agent intermediary.
- Transfer User Assets to the Destination Service.
- Return the User back to the Source Client if the Teleport fails.
- Tell the Source Client to close after a successful Teleport.

A Source Service and a Destination Service SHALL support the *Teleport* API.

The parameters of the *Teleport* API Call are defined in Table 3:

Table 3: Parameters for the Teleport API Call

Parameter	Type	Description	Status
teleport-id	Integer	128-bit integer randomly generated by the Source Client when Teleportation is initiated	M
teleport-pin	Integer	A 2-digit decimal number randomly generated by the Source Client when Teleportation is initiated	O
user-id	Text	ID of the User in the Originating Service	O

Parameter	Type	Description	Status
launch-client	Boolean	*true* if Destination Service should launch the Destination Client, *false* if the Source Client will launch the Destination Client. Default is *false*.	O
uri-destination	Text	Destination URI	O
uri-source-ack	Text	URI (called by Destination Service if teleport is successful)	O
uri-source-nack	Text	URI (called by Destination Service if teleport is unsuccessful)	O
assets	Object	Reserved for Future Use	O

The returned fields within the *Teleport* API Response are defined in Table 4:

Table 4: Parameters for Teleport API Response

Field	Type	Description	Status
approval	Boolean	See above	M
error	Text		O

The mode of operation for the *Teleport* API is described below:

1. The Source Service (Client, Cloud, or both) SHALL retrieve *teleport-id* *teleport-pin*.
2. The Source Client SHOULD instruct the User to take note of *teleport-pin*.
3. The Source Service SHALL call the *Teleport* API at the *Destination URI* with relevant parameters (see Table 3). If the *Destination URI* contains parameters (such as *location*), these parameters SHALL be passed with the *Teleport* API call. The Destination Service MAY require a transaction fee for using the API (Section 5.8). Note: Transaction fees discourage DDoS attacks.
4. When the Destination Service receives the *Teleport API* call it retrieves the cached information sent to it via the *Query* API call based on the *teleport-id* and makes sure the *Teleport* API was called within the *expiration* time. If not, it responds to the *Teleport* API call with *false* for the *approval* return parameter and an optional explanation in the *error* return value.
5. If the Teleport is valid, the Destination Service stores *any parameters* with the rest of the parameters stored with *teleport-id*.

6. The Destination Client SHOULD show the *teleport-pin* to the User so the User can confirm the validity of the teleportation before logging in.
7. The Destination Cloud and Source Service SHOULD work with the Asset Transfer Framework, defined in Section 7.8, to complete the teleportation.
8. The Destination Service responds to the *Teleport* call with *true* for approval.
9. The Destination Service SHALL call the *uri-source-ack* if the Teleport is successful or the *uri-source-nack* if the teleportation is unsuccessful.
10. The Source Service SHOULD authenticate the *uri-source-ack* or *uri-source-nack* calls from the Destination Client and then SHALL act appropriately.

Example *Teleport* API Calls and Responses are provided in Section 6.2.

5.4 IWPS Authentication

5.4.1 Overview

As is the case with web hyperlinks, authentication (and encryption) SHOULD be used with IWPS. There are several ways Worlds can interact with IWPS:

- **No Security**: This is the equivalent of building a website that only supports HTTP and does not support HTTPS. These Worlds SHALL NOT be able to connect to Worlds that require internet or OMA3 security.
- **Internet Security**: This is the equivalent of building a website that uses HTTPS. A World that supports internet security would support TLS (preferably 1.2 or greater) and obtain an X.509 certificate from a Certificate Authority that is well supported in modern browsers. Worlds that only support internet security SHALL NOT be able to connect to Worlds that support OMA3 security.
- **OMA3 Security**: OMA3 offers a certification program that significantly hardens standard internet security. It uses the same mechanisms of internet security but adds additional functionality that increases trust and reduces the attack surface of IWPS.

The following requirements are provided for IWPS Clients and IWPS Clouds:

- SHALL support at least one of OMA3 Security, internet security or No Security.
- SHOULD support OMA3 Security defined in Section 5.4.3.
- SHOULD support internet security, defined in Section 5.4.2, if OMA Security is not supported.
- SHOULD NOT support No Security.

In addition to giving API Clients the ability to authenticate an API Server, TLS also allows an API Server to authenticate the API Client that is accessing the API. This is called mutual authentication. Mutual TLS requires each client to have a unique X.509 certificate and put a certificate identifier in a registry trusted by the API server. This is straight-forward for a cloud-based API Client to do since it uses a similar process as web server X.509 certificates. For Device Application clients it is more difficult. To comply with Mutual TLS when connecting to a Cloud, Device Application clients can do one of the following:

- Have the Source Service generate a unique X.509 certificate for each Client.
- Use X.509 certificates to wrap an OAuth2 token obtained from an OAuth Server.
- Rely on the Cloud portion of a Source Service to make the API call and provide the X.509 certificate.

In addition to TLS, which authenticates endpoints at the transport-level, IWPS also supports User-level authentication. This aspect of OMA3 security is described in more detail in Section 5.5.

5.4.2 Internet Security

IWPS Clients supporting internet security SHALL adopt the following practices:

- SHOULD NOT support or downgrade to versions of TLS that are earlier than TLS 1.2.
- SHOULD NOT support algorithms and cipher suites deprecated by NSA [8] or IETF [9].

Cloud servers SHOULD use an SSL/X.509 certificate with Organization Validation, Extended Validation, or equivalent.

5.4.3 OMA3 Security

OMA3 Security adds the following security requirements to internet security:

- SHALL support all recommendations for internet security.
- SHALL support TLS with mutual authentication (both endpoints of a TLS connection authenticate the other) for Cloud endpoints (Client authentication is not required).
- SHALL use X.509 certificates, signed by an OMA3 authorized certificate authority (CA).
- SHALL use at least two factor authentication for User login.
- SHALL complete and maintain OMA3's future cybersecurity certification.

5.4.4 Blockchain Node Security

Blockchain-based Worlds do not use Cloud servers in the traditional sense. Instead, the Client communicates with a blockchain node, sending its signed transactions to record changes in state. Blockchain nodes cannot initiate communications and can only respond to requests.

In IWPS, blockchain nodes SHOULD support TLS 1.2 or higher. The X.509 certificate can store either the URI of the node (if it leverages DNS) or the public IP address of the node. If the IP address is referenced, it should be recorded in both the Subject Alternative Name (SAN) field and the Common Name (CN) field of the X.509 certificate.

IWPS blockchain nodes MAY support Mutual TLS.

5.5 Identity Framework

Services that support IWPS SHOULD support at least one of the following ID formats:

- Email
- Username
- Phone number
- Blockchain address (or a blockchain address name service)
- Passkey

More formats can be added in the future.

The handling of User identity in portaling can be handled in several ways, all supported by the present specification. Four different identity implementations are shown below:

- **Separate Identities, Separate Verification**: In the implementation, the Source and Destination Service SHALL support different IDs for the User. The Source Service SHALL pass the Source User ID to the Destination Service during the portaling process, but the User SHALL log into the Destination Service with the User's Destination Service ID. The Destination Service MAY cache the two IDs together so the next teleportation between the two Services is more seamless.
- **Same Identity, Separate Verification**: In this implementation, the Source Service SHALL pass the Source User ID to the Destination Service, but the Destination Service SHALL verify independently that the User has access to that Identity.
- **Same Identity, Source-verified**: In this implementation, the Source Service SHALL pass the Source User ID to the Destination, along with the information necessary to verify that the User has access to the Identity. The Destination Service NEED NOT perform login authentication. However, there are limits to the usage of

Source-verified ID, e.g. when the session ends, the Identity is no longer available without further authentication.

- **Separate Identities, User Agent Intermediary**: In this implementation, the Source Service SHALL pass the Source User ID to a User Agent. The User Agent SHALL then pass the Destination User ID to the Destination Service. The User Agent MAY cache the two IDs together so the next teleportation between the two Services is more seamless.

5.6 Asset Transfer Framework

TBD.

5.7 Look and Feel Framework

TBD.

5.8 Payments Framework

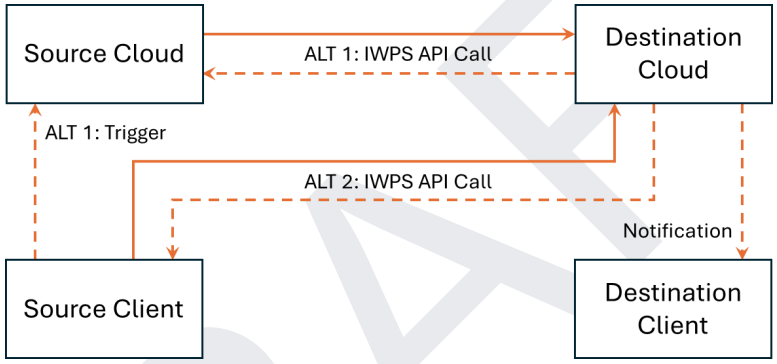
TBD.

6 Examples (Informative)

6.1 Example Hybrid Communication Scenarios

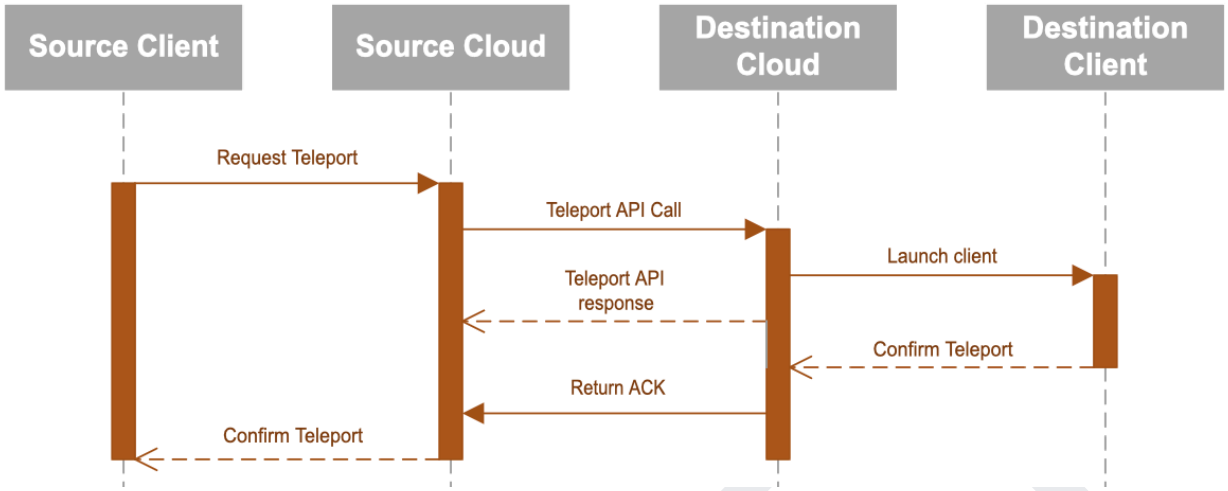
Hybrid Communications as described in Section 5.1.2 deserves more explanation given the myriad of possibilities based on the capabilities of communication between apps on the same device. Here we illustrate two examples of Hybrid Communications when calling the *Teleport* API. The first has the Source Client leveraging the Source Cloud for communications with the Destination Cloud (ALT 1), and the second has the Source Client calling the Destination Cloud directly, bypassing the Source Cloud (ALT 2).

Figure 5: Hybrid Communications Alternatives



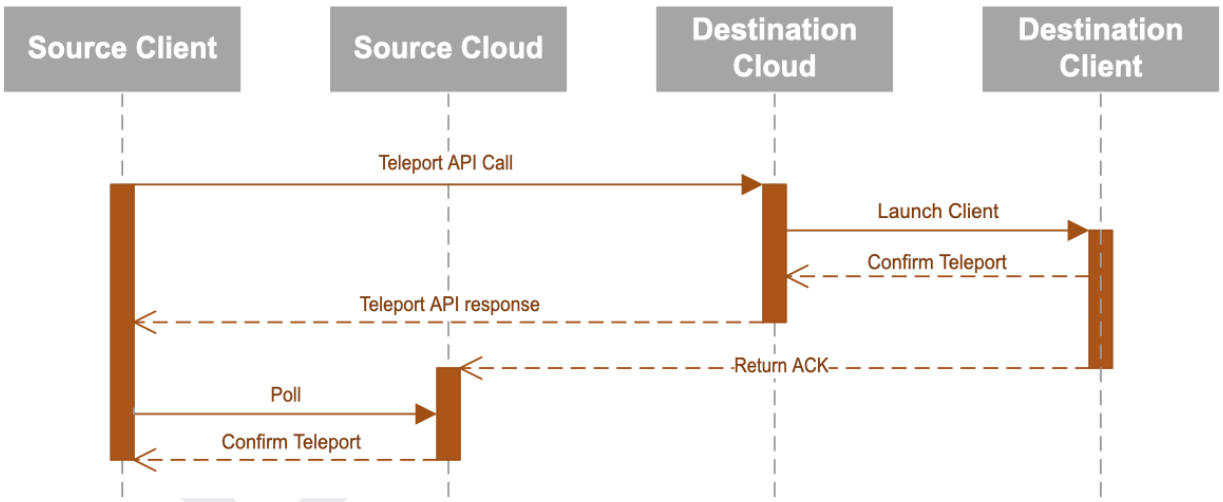
The message sequence diagram for Hybrid Communication (ALT 1) is shown in Figure 6.

Figure 6: Hybrid Communications via Source Cloud (ALT 1)



The message sequence diagram for Hybrid Communication (ALT 2) is shown in Figure 7.

Figure 7: Hybrid Communications via Destination Cloud (ALT 2)



The benefit of leveraging the Source Cloud is to simplify mutual TLS authentication with the Destination Cloud. Without the Source Cloud the Source Client will need to provide its own unique X.509 certificate to the Destination Cloud. However, some blockchain-based Worlds may not have a Source Cloud that can initiate communications with a Destination Cloud. In this case, ALT 2 is used.

6.2 Example Teleport Scenarios

This section shows the flows for different types of virtual Worlds. The main parameters are:

- Is the virtual World a “Traditional” World in that the Cloud software stack is completely controlled by the Service, or is the Cloud a “Blockchain” World where the Service only has control over the smart contract?
- What are the mechanisms of direct communication between the Source Client and Destination Client? Inter-Application? Inter-Device?

6.2.1 Traditional Source, Traditional Destination, Same Device

This example reflects the following parameters:

- Source Service: Traditional
- Destination Service: Traditional
- Client Communication: Hybrid Communications

The following software actors are considered:

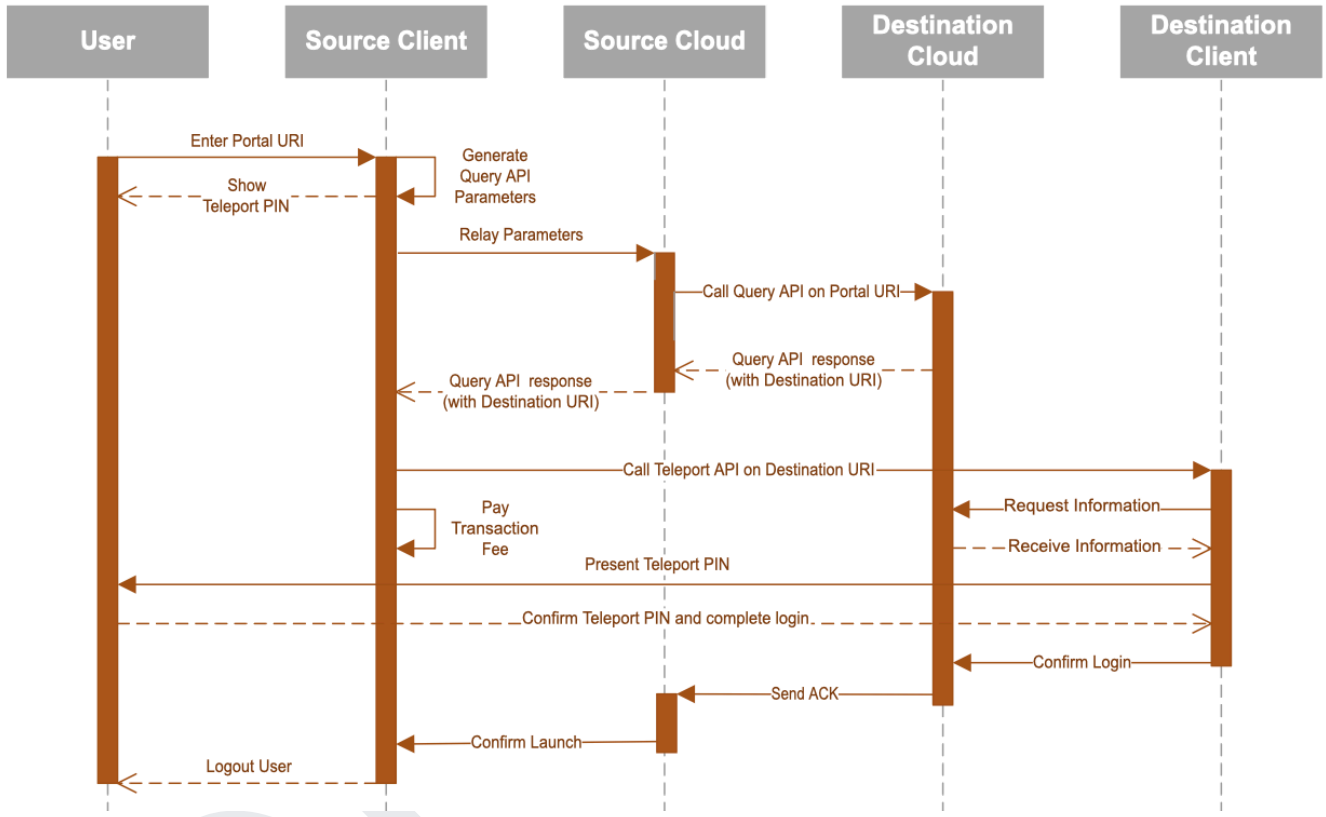
- Source Client: stand-alone application on Android
- Destination Client: stand-alone application on Android
- Source Cloud: public cloud server with standard web stack
- Destination Cloud: public cloud server with standard web stack

Teleport Flow (see Section 5.3 for a more detailed description):

1) User enters the *Portal URI* into a Portal on the Source Client.
2) Source Client generates *teleport-pin* to show User.
3) Source Client sends *Portal URI* and *teleport-pin* to Source Cloud.
4) Source Cloud sends the *Query* API call, with parameters, to *Portal URI*, which is the Destination Cloud.
5) Destination Cloud responds to the *Query* API, approving the Teleport and returning a *Destination URI*.
6) Source Cloud relays the *Destination URI* to the Source Client.
7) Source Client, with User’s permission, calls *Destination URI* (which includes the *teleport-id*) to launch the Destination Client.
8) Destination Client sends *teleport-id* to Destination Cloud to get the Teleport parameters.
9) Destination Client shows *teleport-pin* to User to confirm Teleport.
10) User logs in to the Destination Client, which confirms login to Destination Cloud.

11) Destination Cloud sends *uri-source-ack* to Source Cloud.
12) Source Cloud relays the successful teleportation to Source Client.
13) Source Client logs out User and closes.

Figure 9: Detailed messaging for Clients on the same Device



6.2.2 Traditional Source, Traditional Destination, Different Devices

This example reflects the following parameters:

- Source Service: Traditional
- Destination Service: Traditional
- Client Communication: None

The following software actors are considered:

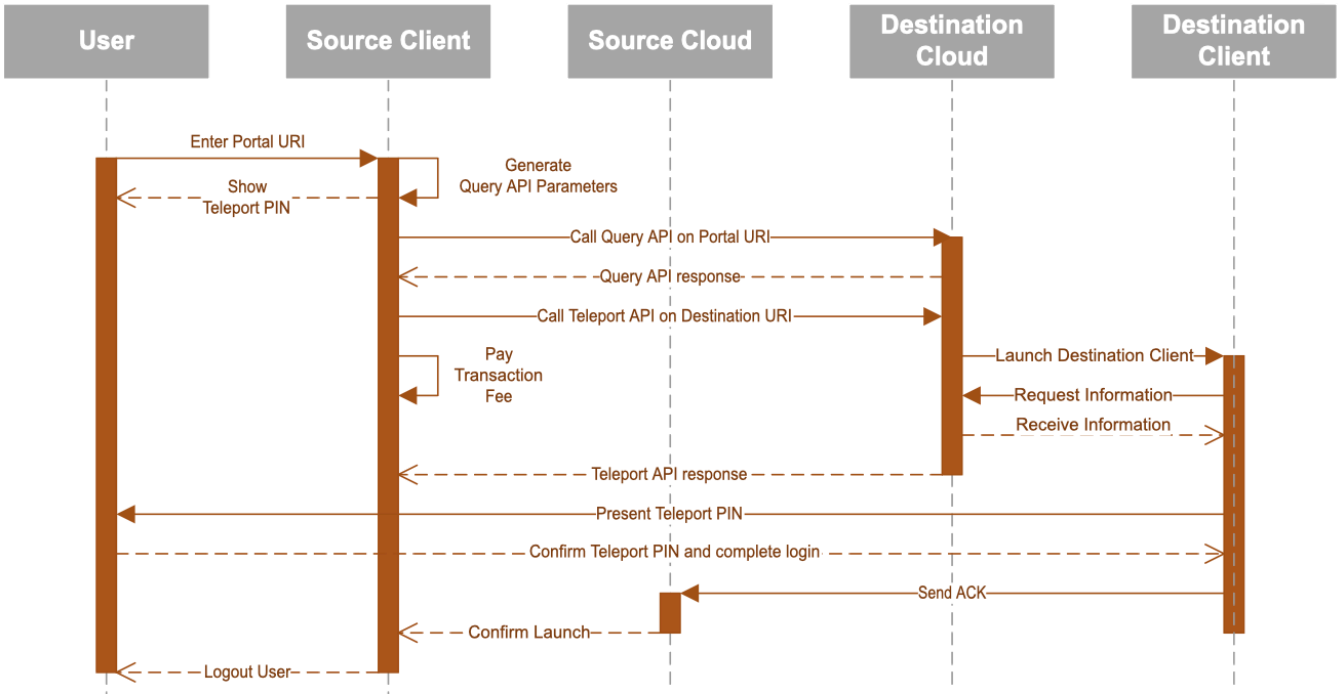
- Source Client: stand-alone application on Android
- Destination Client: stand-alone application on iOS
- Source Cloud: public cloud server with standard web stack
- Destination Cloud: public cloud server with standard web stack

Teleportation Flow (see Section 5.1.2 for a more detailed description):

1) User enters the *Portal URI* into a Portal on the Source Client.
2) Source Client generates *teleport-pin* to show User.
3) Source Client sends *Portal URI* and *teleport-pin* to Source Cloud.
4) Source Cloud calls the *Query* API with parameters at the *Portal URI*, which is the Destination Cloud.
5) Destination Cloud responds to the *Query* API call with *Destination URI*.
6) Source Cloud sees Destination URI has the same base address as Portal URI and calls the *Teleport* API with *teleport-id* on the Destination Cloud.
7) Destination Cloud launches Destination Client, sending it *teleport-id*.
8) If necessary, Destination Client sends *teleport-id* to the Destination Cloud to get the remaining teleportation parameters.
9) After confirming the *teleport-pin*, User logs in to the Destination Client, which confirms login to Destination Cloud.
10) Destination Cloud sends *uri-source-ack* to Source Cloud.
11) Source Cloud relays the successful teleportation to Source Client.
12) Source Client logs out User and closes.

The message sequence chart in Figure 10 shows the flow when the Destination Cloud launches the Destination Client directly.

Figure 10: Detailed messaging with Client in different Devices



6.2.3 Traditional Source, Blockchain Destination, Same Device

This example reflects the following parameters:

- Source Service: Traditional
- Destination Service: Blockchain
- Client Communication: Direct

The following software actors are considered:

- Source Client: browser-based application on PC
- Destination Client: browser-based application on PC
- Source Cloud: public cloud server with standard web stack
- Destination Cloud: smart contracts running on an IWPS-compatible blockchain

Teleportation Flow:

1) User enters the *Portal URI* into a Portal on the Source Client.
2) Source Client generates *teleport-pin* to show User.
3) Source Client sends *Portal URI* and *teleport-pin* to Source Cloud.
4) Source Cloud sends the *Query* API call with parameters to *Portal URI* at the Destination Cloud, which is a blockchain node capable of receiving the API call.
5) Destination Cloud responds to *Query* with a *Destination URI*.
6) Source Cloud relays the *Destination URI* to the Source Client.
7) Source Client, with User's permission, calls *Teleport* API on *Destination URI* with *teleport-id* to launch the Destination Client in a new browser tab.
8) Destination Client sends *teleport-id* to Destination Cloud smart contract to get the remaining teleportation parameters.
9) After confirming *teleport-pin*, User logs in to the Destination Client.
10) Destination Client sends *uri-source-ack* to Source Cloud, bypassing Destination Cloud. Note- Destination Client will need to authenticate itself with Source Cloud. If it can't, then Destination Cloud will need to send *uri-source-ack*.
11) Source Cloud logs User out of the Source Client.

Figure 10 still applies to this scenario.

6.2.4 Blockchain Source, Hybrid Destination, Same Device

This example reflects the following parameters:

- Source Service: Blockchain
- Destination Service: Hybrid
- Client Communication: None

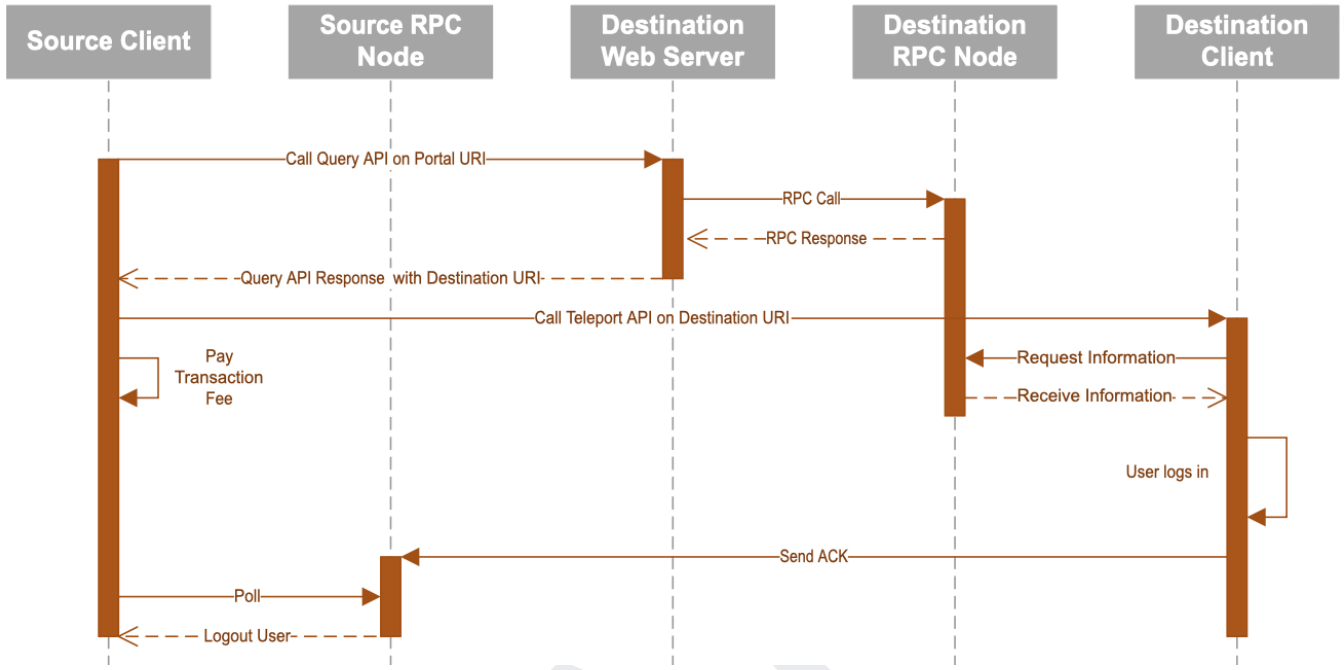
The following software actors are considered:

- Source Client: stand-alone application on PC
- Destination Client: browser-based application on PC
- Source Cloud: smart contract running on Ethereum
- Destination Cloud: public cloud server with standard web stack and a smart contract running on Ethereum

Teleportation Flow:

1) User enters the *Portal URI* into a Portal on the Source Client.
2) Source Client generates *teleport-pin* to show User.
3) Source Client sends the *Query* API call to the Destination Cloud's web server (note- Source Client needs to authenticate itself to Destination Cloud).
4) Destination Cloud's web server sends an Ethereum RPC call with *Query* parameters to an Ethereum node running a Destination Cloud smart contract.
5) Destination Cloud smart contract responds to Destination Cloud's web server with a *Destination URI*.
6) Destination Cloud's web server responds to Source Client with the *Destination URI*.
7) Source Client, with User's permission, calls *Destination URI* with *teleport-id* to launch the Destination Client.
8) Destination Client sends *teleport-id* to Destination Cloud smart contract to get the complete teleportation parameters.
9) After confirming *teleport-pin*, User logs into the Destination Client.
10) Destination Client sends *uri-source-ack* to Source Cloud smart contract.
11) Source Client polls Source Cloud smart contract to get Teleport confirmation.
12) Source Client logs out User and closes.

Figure 11: Detailed messaging with Client in different Devices



6.3 Example IWPS API Calls

6.3.1 Query API

TBD.

6.3.2 Teleport API- Cloud Communications

TBD.

6.3.3 Teleport API- Direct Communications

TBD.

6.4 Example User Agent Flows

The Destination URI can be formatted in such a way that instead of Source Service calling Destination Cloud directly, Source Service can instead call a User Agent that sends the *Teleport Init* API call to the Destination Cloud. This option allows the User to have more control over their privacy as the User Agent disintermediates the Source Service from the Destination Service and makes tracking more difficult.

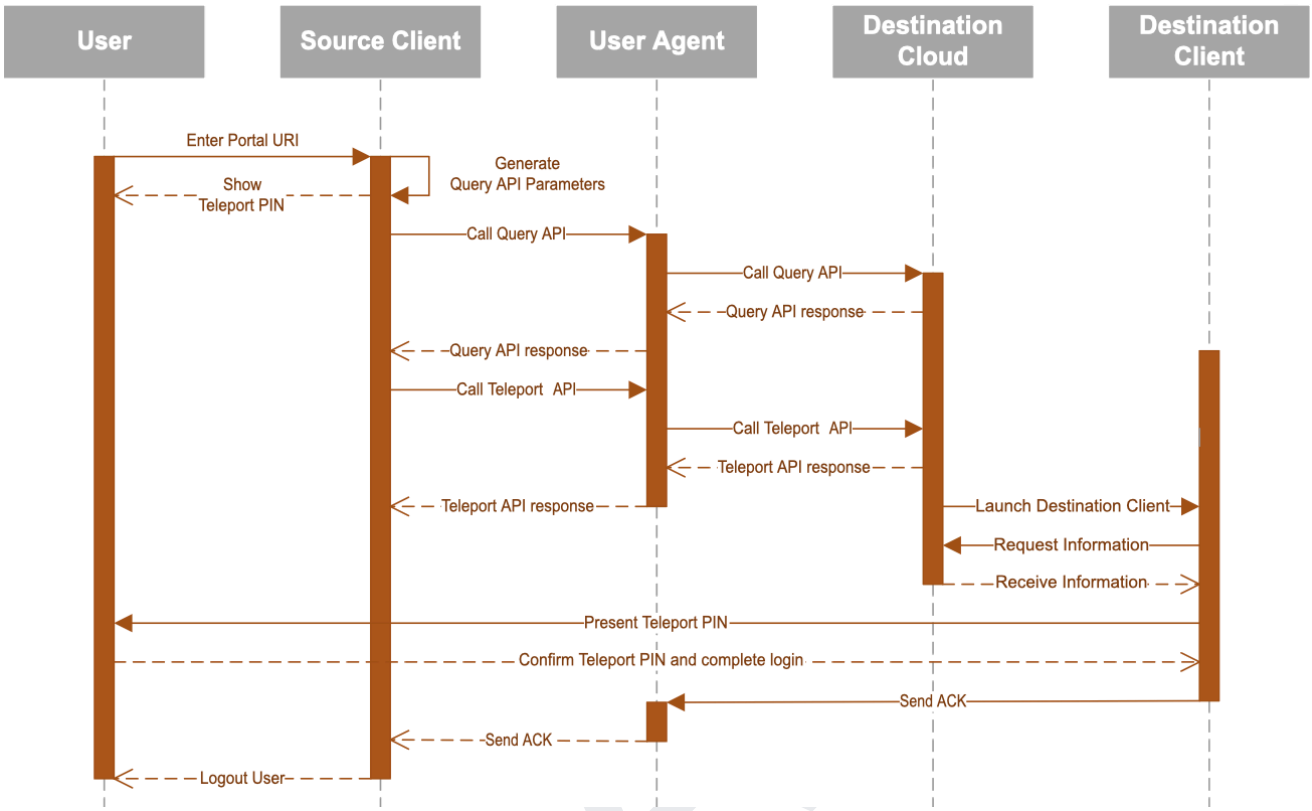
6.4.1 User Login

Here is a possible flow for IWPS using a User Agent.

1) User leverages User Agent to log into Source Client.
2) User gives User Agent a Portal URI and User Agent returns a Modified Portal URI that calls the User Agent instead of the Destination Cloud.
3) User enters the Modified *Portal URI* into a Portal on the Source Client.
4) Source Client generates *teleport-pin* to show User.
5) Source Client calls the *Query* API on the Modfiied *Portal URI*, which points to the User Agent.
6) User Agent stores *uri-source-ack* and *uri-source-nack* and generates its own modified versions to send with Teleport API.
7) Modified *Portal URI* contains the Destination Cloud URI, so the User Agent calls the *Query* API on the Destination Cloud URI. Note- Destination Cloud must trust the User Agent in order to authenticate it.
8) Destination Cloud responds to User Agent with its URI.
9) User Agent responds to Source Client with the User Agent's *Destination URI*, which incorporates the URI of the Destination Cloud.
10) Source Client calls the *Teleport* API on User Agent with *Destination URI*.
11) User Agent calls *Teleport* API on Destination Cloud by extracting the URI from the parameters.
12) Destination Cloud responds to *Teleport* API call.
13) Destination Cloud sends *teleport-id* to the Destination Client to launch it.
14) Destination Client sends *teleport-id* to the Destination Cloud to get the remaining teleportation parameters.
15) Destination Client prompts User for login, showing *teleport-pin*.
16) User leverages User Agent to log into the Destination Client.
17) Destination Client calls modified *uri-source-ack* to User Agent.
18) User Agent retrieves *uri-source-ack* and calls it on the Source Client.
19) Source Client logs out User and closes.

Since Destination Service does not communicate directly with Source Service it cannot track the User's movement in the metaverse. The User Agent, which is trusted by the User, can be enabled to automatically log in User to the Destination Service.

Figure 12: Detailed messaging with a User Agent



6.4.2 Asset Transfer

TBD.