Internet Engineering Task Force   M. Lentczner
Internet-Draft   Linden Research, Inc.
Intended status: Informational   M. Hamrick, Ed.
Expires: January 6, 2011   July 5, 2010

Virtual World Region Agent Protocol: Foundation
draft-ietf-vwrap-foundation-00

Abstract

The Virtual World Region Agent Protocol documents define the protocols by which a vast, Internet wide virtual world can operate. This protocol enables different regions of the virtual world to be operated independently, yet interoperate to form a cohesive experience.

This document specifies the foundation upon which various suites of virtual world functionality are built. It describes the basic structure of VWRAP interaction and common methodology and terminology for protocols.

Status of this Memo

Copyright Notice

Table of Contents

1. Structure

1.1. Introduction

The Virtual World Region Agent Protocol (VWRAP) is about a three way interaction between viewer, agent and region in order to facilitate a shared experience between people. While the description here is grounded in a common view of what a virtual world is, the terms are deliberately described so as to be usable in a wider variety of situations. The VWRAP structure is design to support the abstract the notion of persistent user identity interacting over time in a variety of shared experiences in different persistent locations, especially where the users and locations are operated by different administrative domains.

1.1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

1.2. Domains

The viewer is the element that senses and acts on the state of the virtual world. The viewer does so from the vantage point of an agent. An agent is persistent identity and persona that interacts in a virtual world. The agent persists and can be interacted with even when the user controlling it (though a viewer) is off-line. Regions are persistent locations in the virtual world. Multiple agents may be present in a region at the same time, and when they are they have a shared experience.

Groups of regions and agents are managed by domains. A region domain is responsible for a collection of regions. An agent domain manages agent accounts.

This protocol makes few assumptions about how a domain manages its collection of elements. In particular, it does not assume that a region will be entirely managed on a single host, nor that an agent will or won't be managed by a single process.

It is useful to think of the "stance" that each element takes in the three-way protocol:

The viewer is the direct proxy for a human that wants to control an agent. This control can be direct as in the case of an interactive 3D viewer, or indirect as in the case of a web site that the user directs to display their agent's status.

The agent domain is responsible for the agent itself. The persistent state of the agent is held within the agent domain, and requests to interact with the agent, even by the viewer, are mediated by the agent domain.

The region domain runs the live simulations of regions in the virtual world. The region domain manages the persistent state of these regions.

1.3. Basic Flow

The basic flow of the protocol is:

1. The viewer authenticates to an agent domain for the authorized control of a particular agent.

2. The viewer directs the agent domain to to place the agent in a region.

3. The agent domain contacts the region domain for the region, and negotiates placement of the agent.

4. The region grants access to the agent domain, which in turn passes some of that granted access on to the viewer.

At this stage, each entity will have access to many resources in the other entities. For example:

o The viewer has access to region resources that let it move the avatar.

o The region has access to viewer resources that update the state of objects in the region.

o The viewer and agent have access to resources in each other to facilitate text messaging.

1.4. Structure of the Protocol

The protocol is fundamentally composed of individual resources that can be invoked by one entity in the system upon another. Each resource is a member of a resource class that describes the syntax and semantics of invoking the resource.

The resource classes are composed into suites that form logical groupings, though suites do not otherwise play a part in the protocol. Other protocol suites based on this document, when complete, will describe the several hundred resource classes that

make up the virtual world.

In order to facilitate migration from the existing systems, as well as support future extension, some resources could return information that allow entities to continue to communicate using other protocols and structures. These protocols and structures are not part of VWRAP. It is the intention that when this work is complete, virtual world interaction will be entirely VWRAP based, and that VWRAP itself will have enough extensibility for future development.

Agent and region domains have a few resources that are available at well known URLs. All other resources in the agent and region domains are accessed via capabilities obtained from the those few initial resources.

Since viewers are typically behind firewalls that do not allow connection, resources in the viewer are accessed by event queues held in the agent and region for the viewer. The viewer uses the "long poll" technique to efficiently proxy these inward resource invocations.

1.5. Document Structure

VWRAP is a large suite of interrelated protocol suites. Each major protocol suite is described in its own document. For examples, see the VWRAP Authentication and VWRAP Teleport documents. This document describes the base facilities and concepts upon which the other protocols are based. To be compliant with VWRAP, an implementation MUST conform to this document, and may implement any of the other protocol sets that are deemed relevant.

2. Base Protocols

2.1. Resources, HTTP & REST

All interaction between entities is through a client invoking a resource. Resources are invoked either directly via HTTP [RFC2616], or through an event queue.

For each resource class, this protocol defines how the client obtains the URL, the HTTP verb (or verbs) to be used, the request and response bodies (if any), and significant status codes. Resource classes are designed with REST style semantics.

In general, HTTP & REST are used as follows: The URL will be either well-known in advance or returned in a response from another resource. The latter is called a capability. Except for security

reasons, URLs are always treated as opaque. Clients should not modify them. Parameters are never added to them via the query section. Resource handlers must be prepared to ignore query sections.

Resources follow general REST semantics and so respond to one of these HTTP verb sets:

GET for cacheable resources

GET, PUT for cacheable resources that can be modified

GET, PUT, DELETE for cacheable resources that can be modified and deleted

POST for non-cacheable resources

Unless otherwise stated, if a resource accepts PUT, it accepts multiple PUT invocations.

The request and response bodies are transmitted as serialized LLSD data. If a resource has no response defined, then it can return either an undefined value, an empty map, or have a zero length response body.

HTTP status codes should only be used to indicate the status of the HTTP interaction itself. In general, if the resource is reachable, and the request understood, a 2xx code should be returned.

HTTP headers, both for the request and the response are never part of a resource class definition. Headers are handled as per the HTTP standard.

2.2. LLSD & LLIDL

All data in this system is defined by LLSD and protocols specified in LLIDL. LLSD is an abstract way of talking about structured data. It is defined in LLSD & LLIDL [I-D.ietf-vwrap-type-system].

2.2.1. Serialization

When used as part of VWRAP, the XML and JSON serializations of LLSD MUST be supported.

2.3. Capabilities

This protocol makes extensive use of capabilities. A capability is an opaque HTTP (or HTTPS) URL used for accessing a particular

resource. The provider of the resource has three logical parts: the_grantor_, the_capability host_, and the_service_.

The grantor uses the capability host to construct a capability that maps to the service that provides the resource, then returns that capability to the client. At some point in the future, the client invokes the capability which makes a connection to the capability host. The capability host then proxies to the service to provide the resource.

The parts that make up the provider may be separate entities or may be the same.

The client can't invoke the resource without the capability. Typically the capability is a URL with a cryptographically secure path component. Within the capability host, this URL is mapped to the actual internal resource URL.

The client is free to hand the capability to other entities who become clients of the capability as well. Other than for the security considerations below, the client must not rely on any assumed structure of the capability URL.

2.3.1. Obtaining

For each resource a client wants to invoke, the capability must be obtained. In a few cases, the capability will have been expressly returned in the result of some other resource. Usually, the system uses a seed capability (see below) to request the capability for a given resource by name.

2.3.2. Invocation

To invoke a capability, the holder performs an HTTP transaction with the capability as the URL. The resource class the capability represents will dictate which verb (or verbs) can be used, and what the request and response bodies (if any) should be.

2.3.3. Lifetime & Revocation

Capabilities can be either unlimited or one-shot. Unlimited capabilities can be used multiple times, whereas one-shot can be used only once and are automatically revoked on invocation.

Invoking a one-shot with the HTTP verbs HEAD or OPTIONS does not revoke it.

Any capability can be revoked at will by the provider of the

resource. Clients must be prepared to handle revoked capabilities. A revoked capability, when invoked must return a 4xx HTTP status code. The capability host may return a 404, even if the capability had been previously active.

2.3.4. Names

The resource a capability performs is identified by name. When requesting a capability, or when returning a capability, the opaque URL is identified with this name. The names of such resources are intended to be globally unique.

Names are URIs. When a name appears without a scheme component, then it is a relative URL, considered relative to the base: http://xmlns.secondlife.com/capability/name/

While names do exhibit path-link structure, they are to be considered opaque identifiers. For example, while the following three capability names are indeed from the same protocol suite, nothing should be inferred about a capability that starts with their common prefix:

```
inventory/root

inventory/folder_contents

inventory/move_folder
```

While not required, this protocol prefers names that are all lower case ASCII letters, separated by underscores and forward slashes.

2.3.5. Seed Capability (Resource Class)

In many cases, a sub-system will return a_seed capability_from which other capabilities can be requested.

```
%% seed
-> { capabilities: [ string, ... ] }
<- { capabilities: { $: uri } }
```

The request contains an array of all resource names for which capabilities are desired. The response contains a map with an entry for each capability granted. Note: a grantor may grant all, some or none of the requested capabilities. The grantor may also grant additional capabilities that were requested, or none at all. If the grantor grants none, the response map must be empty and the HTTP status code should still be 200.

2.3.6. Security

If an end-point receives a capability from an untrusted source, it is permissible for security reasons to check the following aspects of the URL before use:

- o The scheme should be http: or https:.
- o The authority (in particular, the resolved host name) should not resolve to ports on the local machine that aren't publicly accessible.

2.4. Event Queues

An event queue enables an entity to invoke resources in the viewer, which cannot be directly contacted via HTTP. This is usually the case because the viewer is behind a firewall that doesn't allow incoming TCP (and hence HTTP) connections from the region or agent domains.

In such a situation, the client establishes a queue of invocation requests for resources in the viewer. At the same time, the viewer uses an*event_queue/get*capability to effectively tunnel the requests from the client to itself.

2.4.1. Basic Flow

When the viewer invokes*event_queue/get*, the entity replies with the list of messages that have been queued up. The viewer takes the response, breaks it apart into a series of requests that it processes on itself, as resource invocations that the entity wanted to perform. The next invocation of *event_queue/get* includes the responses to any requests that have completed processing. While it takes two resource invocations of* event_queue/get* to tunnel a set of invocations in the other directions, subsequent transactions are chained, since the acknowledgement of a previous requests is performed in the same invocation that gets the next set.

2.4.2. Restrictions

Resources accessed this way have the following restrictions:

- o Resources are identified by their resource class name. With capabilities, there can be several resources in an entity that all conform to the same resource class. With event queues only one resource can exist for each resource class within the viewer. This is not usually a severe restriction.

o The only verb allowed is POST.

2.4.3. Event Queue Get (Resource Class)

This resource is a capability both in the agent and in the region, for implementing a tunneled series of resource invocations from the entity back to the client:

```
%% event_queue/get
-> { responses: [ &response, ... ], done: bool }
<- { requests: [ &request, ... ] }

&request = { id: int, name: string, body: undef }
&response = { id: int, status: int, body: undef }
```

2.4.4. Requests

Each request contains a name and a body. The name is the resource name to be invoked. The request can then be seen as equivalent to fetching the capability for this named resource from a seed capability, and then invoking that capability. Since the viewer cannot have URLs that point into it, these two steps must be combined into one operation here.

The id field represents a number that is used later to correlate responses with the requests. The number must be considered opaque from the point of view of the viewer. It is up to the entity to choose an allocation regieme that works for itself.

2.4.5. Responses

Each response includes the id number from the request it is the response to. This enables the entity to correate responses with requests. The status value is the same as the HTTP status code for the request. However, the status of 0 (which corresponds to the value that would be seen if the status field were missing in the LLSD), shall be construed as a status of 200. The body is the response body.

Note that requestors need to be prepared to handle the same set of eventualities as any REST request: A response to a request might never come, or might be delayed significantly.

2.4.6. Long Poll

Both viewers and entities must be prepared to handle use the "long poll" technique to keep the flow of requests timely. Viewers must be prepared to handle that invoking *event_queue/get* may take a relatively long time to return, as the entity may choose to delay

responding if there are no requests pending, or if it believes it would be better to wait for more requests to queue. Entities must be prepared to handle viewers that request as soon as they are ready for events with no delay. Both sides must be prepared to handle time outs and retries.

2.4.7. Closing the Queue

When the viewer is ready to terminate the queue, meaning that it wishes to be done accepting requests, it may signal such by including the done flag in the next invocation of* event_queue/get*. This value is purely advisory, but enables entities to cleanly flush remaining events, and release resources. Specifically, setting done to true in the invocation body indicates to the entity that if no requests are returned, the viewer intends to no longer invoke this queue.

3. Security Considerations

The VWRAP protocols described by this document describe mechanism by which other application specific protocols are layered on top. Issues such as authentication and authorization are described in other VWRAP documents, and only pertain to systems that choose to use them. However, as this document's protocols form a base of others, and these are intended to be deployed across the Internet, there are some basic security considerations at this level.

All resources in VWRAP are invoked via HTTP or HTTPS URLs. Where a resource requires any of end-to-end data integrity, protection from man-in-the middle attacks, or authentication of resource provider, that resource should be accessed via HTTPS, with the client checking the validity of the server certificate. If the URL indicates https:, then the security available with HTTPS connections applies to the resource request.

Some resources in VWRAP are accessed via cryptographically strong URLs. That is, a entity decides to authorize a client to access a resource, and does so by handing back a non-guessable URL to the service to the client. When such a URL is returned, it must be over a HTTPS channel, lest the URL be sniffed as it traverses the network. Care must be taken by the entity providing the capability to ensure that the URL is unguessable and unforgeable. Usually, using a 128 bit random key in the URL path is sufficient, assuming the randomness has sufficient cryptographic properties.

Clients must take care to consider such URLs precious - just as they would session cookies in a web browser environment. These URLs are

authorized to invoke some action, and if leaked, give out that ability. Since they are generally limited in scope, it is possible to delegate these URLs to other sub-systems the client may entrust to perform its work, and it is safer to do so than techniques like sharing session cookies or account passwords.

As discussed in Section 2.3.6, clients must take caution that capabilities returned by services don't point to localhost. The primary reason for this is that it is common for hosts to have more ports and services open to localhost than to external entities. A malicous external entity returning a URL pointed at localhost, if it can guess the likely services available, can cause the client to invoke those services on its behalf, even thought it can't directly view the results. Clients should check the resolved IP address for the host in the URL, since it is trivial to have remotely controlled DNS names that resolve to 127.0.0.1. Note: This threat is no different that already exists in web browsing in general.

There are two denial of service attack vectors. As with any web service, entities must be prepared to handle all manner of ill formed requests, requests that take too much time, and requests that come at a high rate. Standard web service techniques can be used to mitigate these risks. In the case of the Event Queue, clients must be prepared to handle unreasonable, or malformed requests from the contacted entity. If a client finds itself overwhelmed by requests from an Event Queue, simply dropping the connection and not replying is completely acceptable mitigation. The long poll technique also allows either side to release the connection at any time that resources are being too heavily consumed.

4. IANA Considerations

This document has no actions for IANA.

5. Normative References

[I-D.ietf-vwrap-type-system]
Brashears, A., Hamrick, M., and M. Lentczner, "VWRAP : Abstract Type System for the Transmission of Dynamic Structured Data", July 2010.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext

Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

Authors' Addresses

Mark Lentczner
Linden Research, Inc.
945 Battery St.
San Francisco, CA 94111
US

Phone: +1 415 243 9000
Email: zero@lindenlab.com

Meadhbh Siobhan Hamrick (editor)
P.O. Box 783
Boulder Creek, CA 95006
US

Phone: +1 650 283 0344
Email: OhMeadhbh@gmail.com