Live Entity State Stream (LESS) protocol description
draft-jwatte-less-protocol-01

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on September 5, 2009.

Copyright Notice

Copyright (c) 2009 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents in effect on the date of publication of this document (http://trustee.ietf.org/license-info). Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Abstract

Virtual worlds, typically implemented as multi-user shared simulations, are becoming increasingly used for serious work in addition to the traditional uses of research and entertainment.
Whereas previous distributed simulation protocols have been designed with narrow, time-definite scope, the LESS (Live Entity State Stream) protocol is designed to allow open-ended join and leave for a multitude of simulation peers. The LESS protocol specifies how peers of a simulation collaborate and share state to achieve a mutually agreed "collective hallucination," leading to a user-perceivable shared state of a simulated worlds.

Table of Contents

1. Introduction ....................................................... 3
   1.1. Overview .................................................. 3
   1.2. Simulation Model ......................................... 3
   1.3. Justification ............................................. 4
2. LESS Sessions ..................................................... 4
3. Protocol Specification ........................................... 6
4. UDP Framing discussion ........................................... 13
5. Security Considerations ......................................... 14
6. IANA Considerations ............................................. 14
7. Previous Versions ............................................... 14
8. Informational References ....................................... 15
Author’s Address ..................................................... 15
1. Introduction

1.1. Overview

This protocol (the LESS protocol) describes presenting virtual world object properties between virtual world peers (typically, servers of virtual world instances). It allows peers to introduce the types of objects that they will provide, and allows the responding peers to decide what types, and what properties within those types, they actually understand and are interested in. This protocol is part of a suite of protocols that together make up an advanced virtual world interoperability standard; when all the protocols are implemented by two or more separate virtual world systems, those systems can interoperate at a high level, and provide a congruent shared experience to participants from all the participating world implementations and peer hosts. Additional use cases include traditional data loggers, a standardized set of analysis tools, and others.

1.2. Simulation Model

The basic model is that individual objects are simulated on a host that has affinity for that object, and a certain set of typically well-known properties (identified by id in LESS, and by well-known name mapping to id in schema) are presented to observing peers. Those properties are intended to provide a reasonable level of detail of presentation for the object in question as it updates itself, but are not intended to provide a perfect replica of the internal workings of the object. Thus, only properties that are of interest to presentation (to users or other simulators), or user interaction are intended to be included in the schema and data stream.

Because multiple hosts participate in the shared simulation, it is expected that each host simulates some number of objects, presenting them to the other host peers, while simultaneously receiving presentations of the objects simulated on those peers. This makes a typical session using LESS bi-directional. When objects simulated on one host interact with objects simulated on another host, interactions defined in object schema are forwarded from the host detecting the interaction to the host simulating the interactee. Through means outside the scope of this protocol, the servers agree on the basic environment of the simulation, including properties like gravity, up and North vectors, center of coordinate reference frame if referenced to some external frame (such as WGS-84) and static, immutable terrain geometry.
1.3. Justification

The reason for this new protocol, as opposed to various existing protocols such as Sun-RPC, SOAP or SNMP is that virtual world objects update their properties at typically a much higher rate of change than other kind of objects, and the timing of those updates is important. Additionally, efficiency of encoding is fairly important, leading to an implicit-type system that puts type description out of line with the data update stream. The motivation for this is similar to the motivation to use separately negotiated binary codecs for VoIP traffic or streaming video traffic.

The model of interoperability as viewing the presentation of simulated objects through simulation host peer connections was carefully chosen to allow the greatest amount of interoperability possible while requiring the least amount of re-work within existing virtual world implementations. Server-side peering to construct a single, shared space opens up many exciting applications, that alternative approaches to interoperability, such as a forced client/server protocol, or a forced object execution model, would not enable. Additionally, it is expected that the work necessary to implement LESS in existing virtual worlds is less than that required to achieve similar interoperability through other means. A model with the same justification, but a more anachronistic implementation, has been used successfully within the Department of Defense for a long time (IEEE 1278; Distributed Interactive Simulation). Meanwhile, a model that requires all participants to share a given execution model, (like IEEE 1516; the High-Level Architecture) or a common interaction description and client protocol (like VRML; Virtual Reality Mark-up Language and its successor X3D) has not had the same success in enabling broad interoperability across different vendor technologies.

2. LESS Sessions

A connection is established through means outside the scope of this protocol. The connection includes each peer referencing a schema for the connection (which is separate from a schema for entities and types). That schema describes some properties about the protocol implementation (such as what properties are allowed in the connection control messages). Authentication is also assumed to take place during the connection set-up, before the LESS protocol is in effect. One possible implementation is to use HTTP Upgrade: to switch from HTTP/HTTPS to the LESS protocol.

Live entity state updates are streamed through a sequence of packets. Packets are the physical grouping mechanism for transmission over a
network or other medium. A packet contains zero or more messages. Each message is assumed to take effect at the timestamp defined in the packet header. Timestamps are integral representations of time at the source end, and are relative to a clock that only the source end knows. The duration of a single quantum of the time stamp is approximately described in the schema for the connection.

The physical serialization of data types (ints, floats, strings and binary) is described in addition to the logical description of how the data is organized. The logical description assumes the use of object type schemas, as described in the LODS (Live Object Description Schema) document. The schema for the type of an object must be introduced into the state stream, and the receiving peer must subscribe to properties of that type, before objects (entities) of that type can be introduced over the wire. Also note that the name spaces and schema spaces are not generally the same in both directions -- the connection is really two separate connections shared over a bi-directional stream. Object id "123" may mean something totally different when sent from peer A to peer B, as opposed to when sent from peer B to peer A. To further confuse the matter, each message has an implicit direction, which determines which set of identities and schemas is used. For example, "update entity" means that properties on the source system changed, and thus the object id references objects from the source system. "tweak entity" messages are requests from the source to change the state of the destination system, so the object id references objects on the destination system.

Because transport may be TCP or UDP, some ordering rules need to be enforced: Property updates are defined to not be ordered. If a property update goes out in a packet that is dropped, but the sending side has since sent a new update for that same property, the old property set will not be part of the re-send of the dropped packet. Similarly, properties know which packet last set them, and a received older packet with a property set will not have any new effect on the property in question.

Meanwhile, method calls are ordered per entity. A method call may not be effected while there are outstanding dropped, not re-sent messages in the direction of the pending method call message. Method calls are also effected in the order they are sent in a packet. A corollary is that any packet sequence number must be re-sent if dropped over a lossy connection, even if such re-send results in a packet with no messages, to avoid later interactions to be un-queued and effected.

How to handle three or more peers is not defined by this protocol. The connection negotiation may tell each connecting peer about all
other peers it knows about, and only send state updates for entities that it is the master for, or the negotiation may include a forwarding arrangement, where one end takes it upon itself to forward entity data for some other number of peers. For that to work, the forwarding end needs to re-number entities that come from other systems as part of this protocol, which means that the forwarding system has to understand all the data types that include object ids. Hence, trying to "escape" object id into binary blobs in the protocol is not recommended.

3. Protocol Specification

(simplified BNF form, alternatives on separate lines)

packet:
   header message-count (message)*

header:
   // TCP and UDP physical framing is separate.
   // Over TCP, a packet-length field is needed.
   // Over UDP, a minimum of packet-sequence,
   // last-received-sequence and sequence-ack-bitmask
   // is needed.
signature timestamp
   // The signature is calculated on all data in the packet,
   // including leading framing data for TCP or UDP, while
   // treating the signature field as 8 bytes of 0. This means
   // that packet acknowledge fields in UDP and packet length
   // fields in TCP are included in the signature. Because the
   // encoding is endian neutral (or, in some cases, uses
   // defined endian-ness), calculating the signature on the
   // serialized data in the packet is not subject to endian-
   // ness variations.
signature:
   UINT64-LITTLE-ENDIAN
   // The signature is a hash of the contents of the packet
   // (see above) and a key that is provided as part of session
   // negotiation. This allows the system to detect packets
   // that have been altered in flight, as well as possible
   // replay attacks. 64 bits is not cryptographically strong,
   // but good enough for realtime data. View it as an
   // expanded version of a crc-32 of a packet, with a
   // shared key thrown in for good measure. Exchanged
   // that need cryptographically strong security should
   // be done over SSL with appropriate cyphers.
timestamp:
INTEGER

message-count:
INTEGER

message:
  introduce-type-message
    // A data type (with uri for schema) must be introduced
    // from source to destination before objects can be
    // introduced. ID space is source of message.
  subscribe-type-message
    // A destination for object updates will subscribe to certain
    // properties from objects of certain types. The source will
    // only send the properties that are subscribed to. ID space
    // is destination of message, which is source of objects.
  unsubscribe-type-message
    // Stop receiving updates for objects of this type. ID space
    // is destination of message (source of objects).
  introduce-entity-message
    // When subscribing to a type, or when new objects come along
    // on the source, the source introduces the objects to the
    // destination, with initial values. ID space is source of
    // message, which is source of objects.
  remove-entity-message
    // When an object goes away, or when the destination unsubscribes
    // to a type, the source sends remove-entity-message for each
    // entity that goes away. ID space is source of message (which
    // is source of objects).
  update-entity-message
    // As properties change, the source will send occasional updates
    // for each property that changes. Exactly when this is done is
    // up to the source. ID space is source of message (which is source
    // of objects).
  request-entity-message
    // If for some reason the destination wants the latest state of
    // a given object, in a consistent "snapshot" manner, it can send
    // request-entity-message. The result is a new introduce-entity-message
    // from the object source for the identified object (which has already
    // been introduced before). This request is expected to not be common,
    // but helpful in certain scenarios. ID space is destination of message
    // (which is source of objects).
  method-invocation-message
    // Invoke a declared method on an object, in response to some desire
    // on the destination system. ID space is destination of message,
    // which is source of objects.
  method-result-message
    // After an invocation has started (but not necessarily completed, in
// the case of unknown-duration requests like a "sit" animation), the
// result and possible return value is returned to the requester. There
// will be exactly one method-result-message sent per incoming
// method-invocation-message. ID space is source of message, which is
// source of objects.
connection-control-message
// The connection control message is used for meta requests, such as
// requesting more or less frequent data updates, or other commonly
// agreed connection parameters. The ID space is that of the connection
// schema of the destination of the message.
interaction-message
// When an object detects an interaction (such as collision), it sends
// an interaction for peers to pay attention to. ID space is the
// source of the message, which is the source of the object.
tweak-entity-message
// Certain objects may declare UI for tweakable properties. For example,
// a "timer" object may have a "duration" property that can be tweaked
// by any user. Such tweaks are provided UI and expressed not as RPC,
// but as property change requests (for UI presentation reasons). There
// is no explicit response for whether the tweak succeeded or not, but if
// the tweak resulted in a property change, the property change update
// will be sent during regular course of operations. ID space is the
// destination of the message, which is the source of the objects.

introduce-type-message:
  1 typeid uri

typeid:
  INTEGER

uri:
  STRING

subscribe-type-message:
  2 typeid component-property-id-list-list

cOMPONENT-PROPERTY-ID-LIST-LIST:
  component-count (component-property-id-list)*

cOMPONENT-PROPERTY-ID-LIST:
  // Because components may have sub-components, I need to identify
  // them using a path through the object.
  component-id-path property-id-list

cOMPONENT-ID-PATH:
  path-length (component-id)*
component-id:
   INTEGER

property-id-list:
   property-id-count (property-id)*

property-id-count:
   INTEGER

property-id:
   INTEGER

unsubscribe-type-message:
   3 typeid

introduce-entity-message:
   4 typeid entity-id component-property-value-list-list

entity-id:
   OBJECT-ID

cOMPONENT-PROPERTY-VALUE-LIST-LIST:
   // The "value" for a property of type component is in turn
   // a property-value-list, so here I do not need to introduce
   // an entire path. It takes care of itself in the data representation,
   // recursively.
   component-count (component-property-value-list)*

cOMPONENT-PROPERTY-VALUE-LIST:
   component-id property-value-list

property-value-list:
   property-count (property-value)*

property-count:
   INTEGER

property-value:
   property-id value

property-id:
   INTEGER

value:
   CONTEXT-DEPENDENT-VARIANT
   // context-dependent-variant marshals only data, because
   // type/size is known to both sides
remove-entity-message:
  5 entity-id

update-entity-packet:
  6 entity-id component-property-value-list-list

request-entity-message:
  7 entity-id

method-invocation-message:
  8 request-id entity-id component-id-path property-id (argument-list)*

request-id:
  INTEGER

argument-list:
  argument-count (argument-value)*

argument-count:
  INTEGER

argument-value:
  STANDALONE-VARIANT
  // Standalone-variant marshals type, size and data

method-result-message:
  9 request-id status return-value

status:
  INTEGER
  // 0 means success
  // Non-0 means error, which means the return-value will be
  // a string describing the error.
  // HTTP status codes can be used for the error numbers.

return-value:
  STANDALONE-VARIANT

connection-control-message:
  property-value-list
  // These properties depend on the schema for the connection.
  // A peer will only send properties that the other peer have
  // declared in their schema, and that the peer knows how to
  // produce.

interaction-message:
  10 interaction-id property-value-list
interaction-id:
    INTEGER

tweak-entity-message:
    11 entity-id component-id-path property-id value

Serialization of base data types are as follows (C++-like pseudo-code):

INTEGER:
    flag = 0x0;
    if (value > 127) {
        flag = 0x80;
    }
    else if (value < 0) {
        value = ~value;
        flag = 0xC0;
    }
    if (flag != 0) {
        emit_byte(flag | (value & 0x3f));
        value >>= 6;
    }
    while (value > 0x7f) {
        emit_type(0x80 | (value & 0x7f));
        value = value >> 7;
    }
    emit_byte(value);

OBJECT-ID:
    INTEGER

STRING:
    emit_INTEGER(strlen(str));
    while (*str) {
        emit_INTEGER(*str);
        ++str;
    }

FLOAT16:
    emit_little_endian_raw(&value, 2);

FLOAT32:
    emit_little_endian_raw(&value, 4);

FLOAT64:
    emit_little_endian_raw(&value, 8);

LIST(type):
emit_INTEGER(value.count());
for (list<type>::iterator ptr = value.begin(),
   end = value.end(); ptr != end; ++ptr) {
  emit(*ptr);
}

VECTOR(count, type):
  for (int i = 0; i != count; ++i)
    emit<type>(value[i]);

STANDALONE-VARIANT:
  emit_INTEGER(typeof(value)::ID)
  emit_INTEGER(sizeof(emit(value)))
  emit(value)

VARIABLE-BINARY:
  emit_INTEGER(value.size());
  emit_bytes(value.begin(), value.end());

FIXED-BINARY:
  emit_bytes(value.begin(), value.begin() + FIELD_LENGTH);

UUID:
  16 bytes, big-endian format, reading hex values from left to right

Type codes (serialized as INTEGER). Some types have variable-length encoding of their data; other types have an implicit (FLOAT32) or explicit (FIXED-BINARY) length encoding.

0    NULL
1    OBJECT-ID (serialized as integer, but separately called out for routing translation purposes)
2    INTEGER
3    STRING
4    FLOAT16
5    FLOAT32
6    FLOAT64
7    <Type> LIST
8    <Count> VECTOR (followed by count as INTEGER)
9    STANDALONE-VARIANT
10   VARIABLE-BINARY
11   <Count> FIXED-BINARY
12   UUID
The specific physical framing of packets over TCP is:

packet-length: INTEGER

The specific physical framing of packets over UDP is:

sequence-number ack-last-sequence-number ack-previous-bitmask packet

sequence-number: BYTE

ack-last-sequence-number: BYTE

ack-previous-bitmask: INTEGER

4. UDP Framing discussion

When using UDP, the peer should send packets in sequence order, starting at a sequence number described in the connection negotiation. Sequence numbers increase by 1 for each packet sent, and wrap over. The receiving peer should in turn send the latest received sequence number as ack-last-sequence-number (with the initial value being one less than the negotiated first sequence number in that direction). The ack-previous-bitmask is constructed such that, to send an ack for the packet with the sequence number one before the last received packet, the lowest bit is set. For the sequence number before that, the second lowest bit is set. Repeat, up to 64 bits’ worth of acknowledgement. This means that the maximum outstanding window size is 64 packets. With a simulation rate of 30 Hz, this means over two seconds of window size, which should cover any moderately interactive scenario. If you want to support higher latency connections without running out of bits in the 64-packet outstanding window, just pack more into each packet, and send at a lower rate.

The UDP peer can stop putting in ack bits when it receives an ack from the other end for some packet that it knows that the packet in question was acked in. Also see the discussion on which bits of data to include in a packet re-send above. Finally, there is no risk of treating a much delayed packet with the same sequence number as the current packet as an accidentally correct packet, because the timestamp in the header must be monotonically increasing.
5. Security Considerations

This protocol intends to enable interoperability across different hosts using different underlying virtual world technologies. Additionally, trust of interoperating hosts is established using means external to this protocol. Thus, a parser for the protocol should be conservative in range and value checking, and it is recommended that a session is terminated (using means outside this protocol) if a framing error or logical error is discovered. For example, a remote node may attempt to introduce an object of a type for which the schema has not been introduced, or may attempt to present a property that has not been subscribed. A remote node may also have implementation defects that do not properly follow the marshaling rules for data encoding. Any such deficiency should be detected, and lead to immediate disconnection of the remote connection.

Additionally, the simulation consistency of each world host is the responsibility of that host. Whether a given interaction, method call or property tweak is allowed at a particular point in time must be verified by the receiving host. By contrast to logical or framing errors, such permission errors may be caused by user interface shortcomings, timing/race conditions, etc, and thus should not generally result in a dropped connection, but instead a failure result in the cases where results are reported (such as method calls).

A minimal protection against replay, out-of-order and source-spoofing attacks is provided through the 64-bit MAC checksum preceding each packet. If the session is initially established through secure means (such as HTTPS), and a hard-to-guess MAC key is used to generate this checksum, an open transmission of packets over UDP will be moderately hard to spoof using man-in-the-middle attacks or packet alteration. However, if full encryption-grade security is required, the LESS protocol should be used over a fully trusted channel, such as HTTPS with a high-grade cypher and a public-key infrastructure based host trust model.

6. IANA Considerations

This documents does not require any IANA action.

7. Previous Versions

This version supersedes the document draft-jwatte-less-protocol-00.
This version adds a Security Considerations section; clarifies the underlying interoperability model; fixes some spelling errors and clarifies the use of the MAC signature.

8. Informational References

[IEEE1278]
"Distributed Interactive Simulation", IEEE 1278.

[IEEE1516]

[VocabSchemaWIP]
"Session Vocabulary Schema Example (work in progress)", February 2009.

<TypeSchemaWIP>
"Object Type Schema Example (work in progress)", February 2009.

Author’s Address

Jon Watte
Forterra Systems

Email: jwatte@gmail.com