GRADESTA

level 0 cap‘n proto schema

27.10.2021

hobbs.cz
gradesta.org

Level 0

This is level 0 of the Gradesta protocol

The purpose of level 0 is to provide the absolute minimal required functionality.

If you want to add new functionality, add it to level 1.

While this source file is licensed under the LGPLv3, You should consider this document to be read only. Changes to level 0 of the protocol are not welcome.

The reason for this is not animosity towards others, but due to a desire to maintain compatibility as widely as possible.

Message	
ForClient Vertex VertexState UpdateStatus	**ForService** Address
Shared DataUpdate PortUpdate EncryptionUpdate VertexMessage Time	

```
@0xa838a0f012aecc79;


struct Message {
  forClient         @0 :ForClient;
  forService        @1 :ForService;
}




struct ForClient {
  vertexMessages    @0 :List(VertexMessage);
  vertexes          @1 :List(Vertex);
  vertexStates      @2 :List(VertexState);
  updateStatuses    @3 :List(UpdateStatus);
  portUpdates       @4 :List(PortUpdate);
  dataUpdates       @5 :List(DataUpdate);
  encryptionUpdates @6 :List(EncryptionUpdate);
  timestamp         @7 :List(Time);
 Timestamp is sent as a list, but it is really just an optional
value.
}

struct ForService {
  vertexMessages    @0 :List(VertexMessage);
  portUpdates       @1 :List(PortUpdate);
  dataUpdates       @2 :List(DataUpdate);
  encryptionUpdates @3 :List(EncryptionUpdate);
  select            @4 :List(Address);
    Instance ids
  deselect          @5 :List(Int64);
  timestamp         @6 :List(Time);
}
```

struct Address

Addresses are strings with no maximum length. They can include utf-8 emoji's. They have five or six segments. There form is:

*gradesta://<**host**>(:**port**)/<**locale**>/<**service name**>/<**service specific vertex address**>?<**query**>#<**state to be passed to view**>*

OR

<path to unix socket>:<locale>/<service name>/<service specific vertex addresss>?<query>#<state to be passed to view>

the anchor after the # is cut off by the client and not actually sent to the service. GUIs should default to url DECODING the strings so instead of showing URL encoded text. Each segment may contain any valid utf-8 character except newline, `/` and `:`. The service specific address may contain `/`. If it does contain `/` then prefix substrings of the address when using `/` as a separator must also be valid addresses. That means if
`gradesta://example.com/en-us/foo/bar/baz/baf`
is a valid address:
`gradesta://example.com/en-us/foo/bar/baz` *and*
`gradesta://example.com/en-us/foo/bar`
must also be valid addresses.
Addresses should be urlencoded when copied to the clipboard but should not be urlencoded on the wire.

PS: Of course the host/path segment must be a valid hostname or path and hostnames typically don't contain emojis ;)

PPS: The service name `meta` is reserved for level1 of the protocol.

```
struct Address {
  socket        @0 :Text;
 socket is  either gradesta://example.com:<port-number>
  or     gradesta://example.com
  or     /path/to/unix/socket
  it can also be `^` if the socket is a standard local gradesta
manager service
  locale            @1 :Text
  serviceName       @2 :Text;
  vertexPath        @3 :List(Text);
    Names of quargs
  qargs             @4 :List(Text);
    Values of quargs
    qargs and qvals can be zipped together to get the
pairs you want.
  qvals             @5 :List(Text);
   The id of the identity. Identities are gnupg derived and are
specified at level1 of the protocol
  identity          @6 :UInt64;
}
```

Shared messages

Messages are passed between the client and the vertex. These are arbitrary data and are not specified at this level of the protocol. This level of the protocol only specifies the routing of messages to vertexes.

```
struct VertexMessage {
  vertexId      @0 :UInt64;
  data          @1 :Data;
}
```

Heartbeating

We copy, and use, websocket's PingPong method for heart beating. This means that the service sends a ping message when it starts to feal like the client might no longer be active. If it does not get a Pong or other response within a given short amount of time the service will free the resources associated with that client.In our case a Ping using websockets is a literall ping. Otherwise, any time we get an otherwise empty ForClient message that has a timestamp set, this is considered a Ping end the client should respond immediately with a Pong (an empty ForService message with the timestamp set)

```
struct Time {
```
Actual number of seconds since 00:00:00 1.1.1970 None of that leap second nonsense, we're interested in linear time.
```
  time_tai_secs    @2 :Int64;
```
Nanosecond part of time. Note, it can be quite a problem to find the correct TAI time.
```
  time_tai_ns      @3 :Int32;
```
These fields are somewhat optional. The protocol functions without them set but it is better to set them for better debugging and profiling.

Prefer in this order:

1. TAI time.

2. Linear UNIX time (no leapseconds while the program is running)

3. Non-linear UNIX time

```
}
```

Updates

```
Updates originating from the service have negative ids,
updates origionating from the client have positive ids
struct DataUpdate {
  updateId      @0 :Int64;
  vertexId      @1 :UInt64;
  mime          @2 :Text;
  data          @3 :Data;
}

Vertexes have ports. These ports can either be connected or
disconnected.
struct PortUpdate {
  updateId      @0 :Int64;
Vertex from wence the port connects
  vertexId      @1 :UInt64;
Each port has a direction. Directions are important for walk trees
which are defined in a higher level of the API.
  direction     @2 :Int64;
Ordinarilly there are only 4 directions -1, 1 (up/down), -2, and 2
(left, right), however other directions are allowed by the protocal
and may be used, for example by a version control system to link
to a previous version of a cell or by a citation system to link to a
source. In these cases it is typical to refer direction 3 to a context
menu with these choices rather than creating new ports for every
single purpose.
  connectedVertex :union {
    disconnected @3 :Void;
    closed       @4 :Void;
    vertex       @5 :Address;
    symlink      @6 :Address;
  }
}
```

```
struct EncryptionUpdate {
  updateId      @0 :Int64;
  vertexId      @1 :UInt64;
```
Blank string for unencrypted, otherwise a list of GNUPG public keys signed by a trusted key. Each public key is then used to encrypt a shared private key (specific to this vertex) which is used to encrypt all message data and vertex data. A vertex's data and messages are considered to be end to end encrypted if the keys Text is signed by a trusted key and the messages and data are correctly encrypted by the secret key.
```
  keys          @2 :Text;
}
```

For Client

```
struct Vertex {
  address              @0 :Address;
  instanceId           @1 :UInt64;
```
View is an IPFS link to javascript used for viewing and intracting with data. This is an IPFS directory. It can also contain documentation for the vertex's messaging API. In the future other types of frontends besides javascript may be supported. The entry to the javascript should be found in the path: webview/js/index.js
```
  view                 @2 :Text;
}

struct VertexState {
  instanceId    @0 :UInt64;
```
Status is similart to in HTTP.
200 is OK
404 is not found
There is one special response code 222 which referes to 'phantom' cells. Phantom cells work just like normal cells, only they don't exist. There is no way for clients to create cells in gradesta but a service can provide phantom cells, which can become real when the client interacts with them.
```
  status        @1 :UInt64;
  reaped        @2 :Bool;
}

struct UpdateStatus {
  updateId    @0 :UInt64;
```
Status is similart to in HTTP. 200 is OK
```
  status      @1 :UInt64;
```
The address of a vertex providing an explanation
```
  explanation @2 :Address;
}
```