Quadrilateral and Triangle Transfer Protocol – QATTP /1.0

By Vaughan Hilts and Brandon Smith

Status of Memo

This document specifies an Internet application protocol for storing data about polygon shape's. The current version is a working draft subject to protocol changes; distribution of this memo is unlimited.

Contents

Introduction

Purpose

The **Quadrilateral and Triangle Transfer Protocol (QATTP)** is an application level protocol that describes an easy to use standard for storing shape data with nothing but vertex information for querying later based on a simple request and response system. It is designed to be simple, extensible, and stateless.

Distributed systems that require querying of large data-sets coming from multiple sources can benefit greatly from this protocol. The protocol allows for multiple clients to feed in data while others can query in real time. This document will outline a specification that can be used for creating compatible clients and server applications.

QATTP adopts a lot of policies from HTTP where it is useful and simplifies things to make the protocol lighter and minimal for the needs of the application.

Terminology

Shape

A shape is described as either a ***Triangle*** or ***Quadrilateral*** by the definition of this specification.

Triangle

Any polygon with three points is considered to be a triangle. The specification does not enforce constraints.

Quadrilateral

Any polygon with four points is considered to be a quadrilateral. The specification does not enforce constraints.

Client

For this specification, a client is a single user that wants to submit or look up data about shapes.

Occurrence

A simple count of how often a particular shape might have been counted. This is defined more in detail later.

Query

A query is an initial statement issued to a server.

Filter

A filter is a set of rules used to aid in the querying of GET shape data. See later sections for more specifics.

Header

A header is defined as an auxiliary piece of data attached to a request after the main query request has been constructed. These are borrowed from the HTTP specification but are *TAB* delimited rather than newline delimited like the HTTP specification.

CRLF

This is the line ending format that has been adopted for use in this specification. It is denoted by "\r\n"

Verb

Similar to HTTP verbs, the only valid options are GET or POST. They allow retrieval and uploading of data, respectively.

Protocol Constraints and Parameters

Sockets

As with most applications, **QATTP** is a server and client model application.

QATTP is designed as a request and response system running on a TCP socket system, with a persistent connection. Generally, a client will open one connection for all communication with the server. Both requests and responses will be serviced over the same connection, without any additional services required by the platform. The specification tries not to specify anything that requires TCP exclusively – but reliability is key when sending data to be stored, so if TCP is not used some other verification method that is not described in this specification will be required.

When a client is done communicating with the server, the client will close the connection by itself. The specification does not specify when communications are required to be terminated.

Generally speaking, there is no handshake procedure for the protocol as it is an *unauthenticated* protocol.

Encoding

All messages are transported in plain ASCII encoding text, without exception. All transmitted bytes can be treated as text according to the specification, similar to the HTTP specification.

Notation

A brief word on notation is required before specifying the format of messages. The format is relatively simple and thus will be outlined briefly.

`|`

The pipe is a logical OR. When this is spotted, it means either the left hand side or the right hand side. For example, “Yellow | Blue” reads as “Yellow or Blue”.

`<REQUIRED>`

A portion of a message that is shown in angled brackets such as these indicate a mandatory field. A mandatory field is always before any optional fields, outlined below.

`[OPTIONAL]`

A portion of a message that is shown in right brackets such as these indicate an optional field in a message. An optional field is always followed by optional fields, and nothing else.

`CRLF`

As described in terminology, this is a new line. This marks the end of a message.

`\T HEADER:`

The “\T” can be read as the standard escape sequence for the ASCII character “tab”. A portion of a message that is prefixed by the ASCII tab and suffixed with the ASCII “:”is known as a header. A header should be caps insensitive – an implementation server should normalize the input. All excess white space is trimmed from around the headers.

Protocol and Message Format

The basic protocol is described below, with emphasis on the main skeleton implementation. With no exceptions, all messages can be considered to be terminated when a blank (read: new line; CRLF) is encountered.

In general, a request from a client begins with what we call the *query statement.* The query statement follows a form similar to HTTP, in a form of:

```
<VERB> [SHAPE QUALIFIER/VERTICES] [\t Header1:]...[\t HeaderN:] CRLF
```

A **verb** is either **GET** or **POST** which perform operations similar to their HTTP counterparts. These are described in more detail in the client section.

In general, a response from an implementation server responds with something similar to:

```
<CODE>     [MESSAGE]  [\t   Data:]     CRLF
```

Client

When a client wants to perform an action on the server, it must choose between two verbs. These are **GET** and **POST.** Depending on the verb, the format of the message will look different.

Note: If a mandatory field is omitted, then the server will generate an error of type **400 Bad Request** (the standard HTTP error) See server documentation for details.

GET

When a client sends a GET request, it is making a request to fetch shape data from the server. All the fields on this request make up the constraints and narrow the scope of the search. The exact request will look like the following. (Note: Even though line breaks are inserted for readability here, they do not exist unless a CRLF is specified.)

```
GET   <SHAPE QUALIFIER>

[\t Occurrences: n]
[\t Type:]
[\t Shares: x_1 y_1 .. x_n y_n ]

CRLF
```

And then, where:

- Shape Qualifier is either “T”, “Q”, or “B”. This specifies the class of shapes to search for. In the event that this is missing, an error will be back. This applies to every mandatory field, as noted previously.
- **Headers**. The headers in a **GET** context only server a single purpose: they are used to apply filters on the server side. The client specifies the filters they want applied on the data sets on the request. The filter header’s that are supported are as follows and described:

    - **Occurrences**: This filter header asks the server to only return shapes from the data set that have been sent exactly *n* times before.

      Exception: If *n* is 0, then all shapes should be sent up.

      Errors: If *n* is absent, **Bad Request** will be sent by the server with an implementation specific message indicating this.

    - **Type:** This filter header asks the server to only return shapes from the data set that have a specific property type. These properties are generally properties of the shapes, so they are not all detailed. For example, **Square** would return only those that could be classified as a **Square.** Whereas **Right** would only return those triangles that could be counted as right angled triangles. Properties that are supported are:

        - **For triangles…**
            - ACUTE
            - RIGHT
            - OBTUSE
            - SCALENE
            - EQUILATERAL
            - ISOSCELES
        - **For quadrilaterals…**
            - SQUARE
            - RECTANGLE
            - RHOMBUS
            - PARALLELOGRAM
            - TRAPEZOID

      If a property is not supported, a **Bad Request** will be sent by the server in place of the standard reply.

    - **Shares**: This filter header asks the server to only return shapes from the data set that have points in common with the ordered pair vertices `x_1 y_1 .. x_n`

`y_n.` For each pair of points, `(X_i, Y_i)` the point must be included on a shape for it to be returned in the data set.

Note: If the points are not paired properly, (for instance; not numbers or there is an odd amount of numbers) then **Bad Request** will be sent instead of the standard reply.

This is all that is necessary to have a server reply with requested shape data. The **Server** section of this **RFC** will outline other cases where non-standard replies may be sent. If no errors are encountered, then the server will reply with **OK.**

POST

When a client sends a POST request, it is making a request to store shape data on the server. The format is very simple and has very little constraints:

```
POST <X_1 Y_1 ... X_N Y_N >
CRLF
```

Notes on the formatting of the points:

- If the amount of numbers is odd, then **Bad Request** is sent.
- If the parameters following POST are not numbers, then **Bad Request** will also be sent.
- If duplicate vertices are sent, the server will filter these. Clients should avoid sending them but they will not generate any errors. They will be silently removed; so the client will not be notified.
- If more than 8 numbers (that is, 4 points) or less than 6 numbers (that is, 3 points) are specified, the server will reject it and send **Bad Request** as a response.
- If occurrence has a negative value the server will reject it and send **Bad Request** as a response.

In the event that no errors occur, then the server will reply with an **OK** to **POST** messages. See the server documentation.

Server

The server will never send data unless it a *response* to a *request* from a client. The server can reply to both **GET** and **POST** requests.

These are described below as they are separate, although format is similar. There is a single response which is sent in both cases in case of internal failure; it is documented later.

Invalid Request

If a client sends a message that is malformed, bad or otherwise not usable then a request in the form described below will be sent as a response. The client should not repeat this request, as it will return the same result.

```
400 Bad Request
<\t Reason: >
CRLF
```

`Reason` will be populated with an implementation specific reason as to why the message could not be interpreted. The specification does not enforce the possible error strings that could be returned.

GET

When a client sends a **GET** request, there are two possibilities: the request is valid or it is not. Thus, this section is broken up into two sections.

When the request from the client has not violated any of the structured rules, the request is valid. The response will be in the format of:

```
200 OK [Shape1, ... ShapeN]
CRLF
```

Shape1, Shape2 are formatted objects with a schema that is defined below. If there are no shapes to return, the line will simply be empty. For each **Shape,** an object will exist in such that the object looks like…

```
<X_1,Y-2..X_i,Y_i>:<properties>:[occurrenceCount]
```

If more than one shaped is returned by the call, they will be separated by ampersands. Such as the following example:

```
<X_1,Y-2..X_i,Y_i>:<properties>:[occurrenceCount]&<X_1,Y-
2..X_i,Y_i>:<properties>:[occurrenceCount]
```

Notes: Properties can assume any of the supported **Type** filters provided by the client, and contain multiple values. Properties will be comma separated like the vertex declarations. The type of the shape being returned can be inferred from the number of points that are provided from the server in the response. This will always be 3 vertices or 4. The various lists of items are returned are separated by colons as a list terminator.

POST

If the client sends a valid **POST,** then a message will be returned in the following format

```
200   OK
<\t Type:>
CRLF
```

The **Type** is set to the identifying shape type. This will either be “T” or “Q”.

Otherwise, “**Invalid Request**” as previously defined kicks in and a **Bad Request** will be sent.

Other Responses

If for some reason, the server cannot reliably respond for a reason that is not at the fault of the client it will generate a **500 Internal Server Error.** The format of the response will look like:

```
500   Internal Server Error
CRLF
```

To avoid leaking details, the server is not responsible for providing any other information to the client as to why. The client is free to keep making requests as the server believes it is a one off.

In case of fatal errors, the users socket will be disconnected and the client application implementation will handle this itself.

Synchronization

Although synchronization itself can be an implementation detail in many cases, this section will outline the important aspects a multi-threaded implementation will need to obey and what clients can expect.

Queries about shape data are made at processing time; not request time

This means a server is not responsible for keep tracking of state snapshots to reply to requests when a client had initially requested them. This is important as the typical implementation of a server will be multi-threaded. This means that between the time a **GET** request is issued and the time it is processed another client may have already issued a **POST** and added additional data to the server.

Typical implementations will be multi-threaded; data must be protected

An implementation of this protocol is expected to be multi-threaded with one thread per client that connects. Our implementation of the storage backing will be using a simple **`ArrayList`** provided by Java while using the **`Collections.synchronizedList`** wrapper allowing `.add` to be protected by a mutex. This allows us to not worry about the specific details of the implementation.

Increments on the counters themselves for occurrences should be synchronized as the increment operator on the entry is not atomic.

Clients are expected to request fresh data when they want it; the specification provides no notification mechanism

The client will only have information that is accurate up to when the client's GET request was fully processed. In other words, the server is not responsible for synchronizing the client's copy of the local data it stores. Between application launches the client should not persist or use any caching mechanisms with the expectation that the server will inform it when fresh data is available.