Ranged Filtering of Streaming Numeric Data, or Geolocation Filtering of Streaming GPS Data, using Topic-Based Pub/Sub Messaging

Aaron W. Lee
© Solace Systems Inc., Ottawa, Canada

Abstract— *This paper describes an innovative method to implement location-based filtering of streaming GPS point data without the use of a geospatial database or GIS engine. The filtering algorithm takes a set of polygons as input, and generates an approximating shape of a specified relative accuracy as output, consisting of a union of rectangles. These rectangles can be used to perform point-in-polygon comparisons quickly using text-based pattern matching in a pub/sub messaging system, which excels at handling streaming data at scale. This method has applicability in transportation, logistics, smart cities, security, and surveillance, and can be generalized to one or three dimensions.*

Keywords: geolocation, GPS, filtering, messaging, pub/sub, MQTT

1. Introduction

The number of connected devices and associated generated data continues to rapidly increase. Finding efficiencies in handling this growing volume of data, and deriving value and utility from information in real-time is becoming essential. Messaging systems are emerging as a central technology in the domain of the Internet of Things for connectivity at scale, as well as being able to efficiently route the torrents of information in the Big Data realm.

To filter streaming data based on location, or to provide geo-fencing capabilities (detecting an entity entering or exiting a defined area), one may consider using a Geographic Information System (GIS) or some form of geospatial database to comprehend location-based semantics. Retrieving data from these systems is often poll-based, running queries at set intervals, and may not scale to large numbers of devices or modern real-time, event-based requirements.

As an example, a transportation/delivery company may have tens of thousands of vehicles, hundreds of depots, and hundreds of thousands of potential delivery sites; or a municipal transit service with 5,000 buses and many times more passengers. If every vehicle was generating GPS updates every second, and if every depot and every passenger wanted to be able to receive the live streaming location of the vehicles within a set distance from their location, consider the sheer number of updates and queries these information systems would have to be able to support.

I present a method by which to filter streaming coordinate data to a specified area using a publish-subscribe messaging system that utilizes text comparison rules for routing of information. This paper shall:

- Define messaging systems, pub/sub, and nomenclature around routing with topics
- Demonstrate how topics can be used to encode coordinate data, and how topics can be used to match a rectangular geographic area
- Provide an algorithm to generate a filtering mechanism to approximate a given area of interest

The method presented here is in the context of 2-dimensional latitude/longitude GPS coordinates, but could be extended to any n-dimensional bounded real number range. Additional applications could include: streaming 1-dimensional numeric readings from pressure, temperature, or vibration sensors (e.g. pipeline or bridge monitoring); 2-dimensional Military Grid Reference System (MGRS) coordinates (e.g. ground Moving Target Indication (MTI) radar data); or 3-dimensional coordinates (e.g. aircraft position).

2. Topic-Based Pub/Sub Messaging

Message-Oriented Middleware (MOM) infrastructure allows the sending and receiving of data, in the form of *messages*, between distributed components using the concept of a shared bus [1]. Many MOM systems use a centralized message *broker* to route and filter the data between components, which architecturally decouples producers and consumers of the message data [2]. While there are several ways to exchange message data, one typical pattern used is the publish-subscribe pattern, or *pub/sub* [3].

As illustrated in Figure 1, when data is published onto the message bus for distribution in a topic-based pub/sub system, it is associated with a *topic* to help describe the data, which provides a mechanism by which to help route it. Components, or *clients*, that wish to receive data about a particular subject will register one or more *subscriptions* with the message broker to indicate their interest. When the data, or message, is published onto the bus, the broker compares the topic of the message with the known subscription list of every connected client, and for every client that has at least

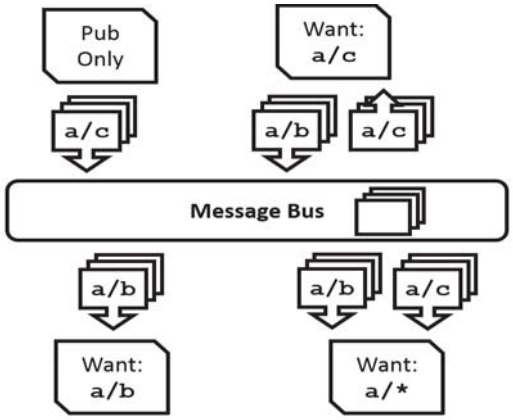

Fig. 1: Simplified diagram of pub/sub architecture

one matching subscription, the message will be delivered to them. This message delivery is typically done in a push-based asynchronous manner without the need to poll the message broker.

2.1 About Topics

In some topic-based publish-subscribe messaging systems, topics are simply text labels, and routing based on subscriptions is essentially a straight string comparison. Other pub/sub systems use hierarchical topics and subscriptions to route the data to interested parties [4]. The exact implementation and syntax of the topics can vary from one messaging system to another, but many of them share similar features. For the purposes of this paper, the following definitions will be used:

Topic string: defined by the publishing client, this is the entire topic belonging to the published message, consisting of one or more levels. E.g. `animal/dog/husky`

Topic delimiter, or *topic level separator*: a (typically) single character used to separate the topic string into multiple levels, giving rise to a hierarchy. In the above example, the topic delimiter is "/" or slash.

Topic level: a portion of the topic string contained within the topic hierarchical delimiters. E.g. "`animal`" or "`dog`"

Topic subscriptions are sent to the message broker by consuming applications to indicate the particular data they are interested in receiving. When a message arrives on the bus that matches a particular subscription, that data will be sent to the corresponding client. A topic subscription can contain one or more wildcards.

Wildcards allow a single topic subscription to match more than one topic string, using rules similar to very basic pattern-matching regular expressions. Although different pub/sub system can implement wildcard matching differently, for the purposes of this paper, define the following topic subscription wildcard:

> * (star): a single-level wildcard, matching zero or more text characters within a single level (between topic delimiters). Only one * wildcard can appear within each topic level. This wildcard can have characters preceding it within the level, thereby providing a prefix match at that level.

```
Subscription:    animal/dog/*
Matches topic:   animal/dog/husky
                 animal/dog/poodle

Subscription:    animal/do*/h*
Matches topic:   animal/dog/husky
                 animal/dolphin/hector
Doesn't match:   animal/dog/poodle
                 animal/dragon/horntail
```

The outcome of the topic subscription match operation is entirely determined by successive text string comparisons: for the match to be successful, each topic level is considered individually and must match the corresponding level of the subscription exactly (if no wildcard), or match the prefix characters exactly (with a wildcard).

Architects and users of a hierarchical topic-based messaging system can specify a standard format for the topics published onto the bus for applications to use. Below are some real-world examples of topic formats:

- `APP_ID/PUB_ID/PRI/DEST_SYSTEM/SUBJECT`
  e.g. `TALON/053B/3/RISK/EOD_REPORT`
- `VEH_TYPE/NUM/LAT/LON/MSG_TYPE`
  e.g. `CAR/0034/_45.382/-075.751/UDPATE`

Building descriptive topic string formats using relevant parts of the accompanying data allows complex and powerful routing rules to be defined. For example:

```
TALON/*/3/*/EOD*
```

A client application subscribing to this would receive all Priority 3 End-of-Day messages published by any client in the Talon application, regardless of the intended destination.

2.2 Encoding Geolocation in a Topic String

I will now define a format to encode location data into a topic string. The following assumes coordinate data will be in decimal degrees [5], but a similar approach could be used to route based on Universal Transverse Mercator (UTM) or MGRS coordinates, Eastings and Northings within a specific grid.

Define two levels within the message topic string to denote the latitude and longitude of the data in decimal degrees format. The range of possible values are therefore:

- Latitude: -90.0 to +90.0
- Longitude: -180.0 to +180.0

Hence, for the latitude coordinate, we need 2 digits before the decimal point, and one character for a plus or minus sign; similarly, 3 digits before the decimal point are required for longitude. If we only consider these two levels of the topic hierarchy, we can provide some examples of encoding:

- Topic: `_45.38/-075.75/`
  Location: 45.38°N, 75.75°W, in Ottawa, Canada
  2 decimal places of accuracy (approx. 1km resolution)
- Topic: `_51.5160/-000.0707/`
  Location: 51.516°N, 0.0707°W, in London, UK
  4 decimal places of accuracy (approx. 10m resolution)

The use of the underscore symbol "_" is used to differentiate non-negative coordinates; the plus symbol "+" could be used, however some messaging standards have special meaning for this character [6]. Note that the coordinates must be zero-padded to prevent accidental subscription matches; this will be explained more fully in the next section.

2.3 Geolocation Topic Subscriptions

After defining a method to describe the data's geographic location in the topic string, it remains to provide a method to subscribe to the data. Consider the two levels for latitude and longitude individually, as the following applies to both equivalently.

As the message broker treats the topic string and topic levels as text, the geographic coordinates must be matched using the text-based comparison in the topic subscription. This is not a numeric comparison: a topic "45.38000" would not match a subscription "45.38" although they are numerically equal. However, using a single-level wildcard, a continuous range of numbers can be matched to the given topic subscription. Consider the following examples for latitude:

- Subscription: `_45.38*/` would match any number that starts with these characters, such as: 45.38, 45.38000, 45.387312, or any number in the range 45.38 to $45.38\overline{9}$; or in interval notation: $[45.38, 45.39)$
- `_45.3*/` would match anything in $[45.3, 45.4)$
- `_45.*/` would match anything between 45°N inclusive and 46°N exclusive
- `_4*/` would match anything in $[40, 50)$, but due to the zero-padding mentioned at the end of Section 2.2 above, this will correctly *not* match numbers between 4 and 5, which is handled by: `_04.*/`
- `-45.3*/` would match anything between $-45.3\overline{9}$ and -45.3, or $(-45.4, -45.3]$
- `-45.*/` would match anything in $(-46, -45]$

Even though this is a text-based comparison, this method using wildcards permits the matching of a range of real numbers using a topic subscription.

By using two levels within the topic hierarchy to represent latitude and longitude, a range of two-dimensional coordinates can be matched using topic subscriptions with wildcards. That is, each subscription corresponds to an area.

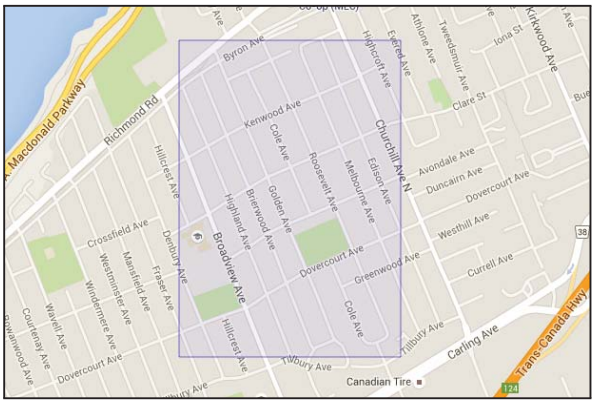

Fig. 2: Axis-aligned rectangle for `_45.38*/-075.75*/`
Background Map data ©2016 Google

Referring to Figure 2, a subscription defined as `_45.38*/-075.75*/` would match any coordinate inside the square area bounded by 45.38°N to 45.39°N, and 75.75°W to 75.76°W. Data published onto the message bus with a coordinate that intersects this box would be sent to the recipient via this subscription. Note that the displayed area in Figure 2 is not perfectly square due to longitude degrees becoming compressed towards the poles; this area would be square at the equator.

Fig. 3: Rectangle for `_45.38*/-075.7*/`
Background Map data ©2016 Google

Consider Figure 3. Subscription `_45.38*/-75.7*/` would match any coordinate in the long, wide rectangular area bounded by 45.38°N to 45.39°N, and 75.7°W to 75.8°W. As the wildcard on the second level is placed one decimal place to the left, this rectangle's longitude range is 10 times larger than its latitude, or 10 timers "wider" if drawing with north up. By changing where the wildcard is placed within the subscription string, it allows the creation of different shaped areas as follows:

- Square rectangle: decimal accuracy for both latitude and longitude portion of the subscription is the same.
  E.g. `_45.38*/-075.75*/`

- Horizontal rectangle: spans a larger range of longitude coordinates; the decimal precision for the longitude is at least one decimal places less.
  E.g. `_45.38*/-075.7*/`
- Vertical rectangle: spans a larger range of latitude coordinates; the decimal precision for the latitude is at least one decimal place less. E.g. `_45.3*/-075.751*/`

Observe that the square rectangle illustrated in Figure 2 is contained wholly within the horizontal rectangle in Figure 3, and is in fact exactly $1/10$ of the larger rectangle; this occurs as the corresponding topic subscription has one extra decimal degree of accuracy. More generally, it is possible to split any rectangle corresponding to a geolocation topic subscription described above, either horizontally or vertically, into ten equally sized sub-rectangles, or *child* rectangles, by replacing the larger topic subscription with ten smaller subscriptions that have the wildcard character moved one position to the right in either the latitude or longitude level respectively.

From these examples, it should be clear that there exists a 1-to-1 mapping between the defined wildcard topic subscriptions and axis-aligned rectangles that line up with decimal boundaries of the degree coordinates.

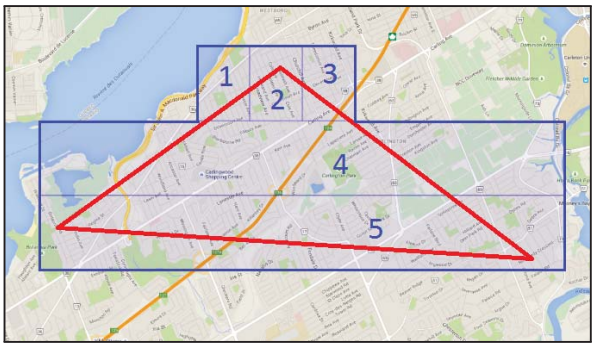

Fig. 4: Composite geometry of five rectangles
Background Map data ©2016 Google

Finally, by subscribing to multiple topic subscriptions simultaneously that cover different areas, it is possible to produce a composite geometry of boxes. Figure 4 shows a union of five rectangles that approximate a triangle. A message whose coordinates lie within the union of all the rectangular shapes will match one of the corresponding geolocation topic subscriptions.

3. Iterative Algorithm for Generating Subscriptions

I now present an algorithm for generating a set of rectangles to approximate a given input geometry or geometries. As demonstrated in the previous section, a geolocation topic subscription can be represented by specific a rectangular axis-aligned shape, whose edges line up with the decimal boundaries. Hence, by providing an algorithm to generate a set of these rectangles which can be converted back to corresponding topic subscriptions, the solution to the problem of filtering streaming point-location data to an input geometry will ultimately be shown.

Due to using decimal degrees as the coordinate system, it effectively divides the surface of the earth into four quadrants, defined by the location of the origin at $(0, 0)$ and the respective signs of the latitude and longitude coordinates. Namely:

- North-East quadrant: origin in lower-left corner
  Coordinates: $(+lat, +lon)$
  Topic strings of the form: `_yy.yyy/_xxx.xxx/`
- North-West quadrant: origin in lower-right corner
  Coordinates: $(+lat, -lon)$
  Topic strings of the form: `_yy.yyy/-xxx.xxx/`
- South-East quadrant: origin in upper-left corner
  Coordinates: $(-lat, +lon)$
  Topic strings of the form: `-yy.yyy/_xxx.xxx/`
- North-East quadrant: origin in lower-left corner
  Coordinates: $(-lat, -lon)$
  Topic strings of the form: `-yy.yyy/-xxx.xxx/`

As the format of the topic strings varies in each quadrant, the starting state of the algorithm will be four large rectangles to cover the earth, with coordinates of the form:

$$(lat_1, lon_1), (lat_2, lon_2) = (0, 0), (\pm 100, \pm 1000)$$

The basic premise of the algorithm is as follows:

- Start with 4 quadrant rectangles that cover everything
- If the approximation is not accurate enough (based on the percent of coverage of the input geometry, or some other measure), pick a rectangle and subdivide into 10 equally-sized rectangles, splitting either vertically or horizontally as appropriate. This translates into replacing one topic subscription with 10 more-accurate subscriptions.
- Discard any rectangles that do not intersect the input
- Repeat, splitting larger rectangles into smaller ones, until arriving at the desired result

3.1 The Algorithm

Define:

- *Coverage Ratio*: the percentage of how much the inputted target shape geometry covers a particular rectangle, or also how much the target covers the union of all rectangles. This will be the accuracy of the approximation, and determines how many false positives will match; an 80% coverage ratio, or 0.8 will result in approximately 20% false positives, or 1 out of 5.
- *Child*: when a rectangle is split into 10 smaller rectangles, they are the children

- *Split*: the act of subdividing a larger rectangle into 10 equally sized rectangles, either vertically or horizontally

Constraints:

- *Subscriptions*: the number of subscriptions a topic-based pub/sub messaging system can maintain is not infinite. Having more subscriptions requires more work to be done by the system to match data. Hence, it may be necessary to limit the number of subscriptions (rectangles) generated as a result of the output, even if the desired coverage ratio is not achieved.

Inputs:

- T = target shape geometry that we are trying to approximate. The target is a multi-polygon: it can have holes and multiple pieces, but cannot be complex (i.e. no crossings)
- k = maximum number of rectangles allowed in the output, a terminating condition; an integer ≥ 4
- z = minimum coverage ratio desired for the output, i.e. the accuracy, a terminating condition; a scalar $[0, 1)$

For example: given a geometry T, compute an approximation to T, where T covers at least $z = 80\%$ of the approximation or consists of no more than $k = 50$ rectangles.

Computed Values and Outputs:

- $L = \{R_1, \ldots, R_n\}$ = list of all rectangles (the algorithm output)
- $n = |L|$ = number of rectangles in the list L
- R_i = the rectangle at position i in L; a geometry
- $S = \bigcup_{i=1}^{n} R_i$ = union of all rectangles; a geometry
- $A(T)$ = area of the target shape; a scalar
- $A(R_i)$ = area of rectangle at position i in L; a scalar
- $A(S)$ = area of the union of rectangles; a scalar
- $c(R_i)$ = coverage ratio of a rectangle; a scalar $[0, 1]$, given by $(A(R_i) \cap A(T))/A(R_i)$. That is, what percentage of the area of the rectangle is covered by the target:
  0 = no intersection
  1 = the rectangle is completely contained by the target
- $c(S)$ = coverage ratio of the union of rectangles; a scalar $[0, 1]$, given by $(A(S) \cap A(T))/A(S)$
- $R'_{i,j}$ = the j^{th} child rectangle of R_i after splitting, with j in $[1, 10]$; a geometry

Pseudocode:

```
L = [R1,R2,R3,R4]
while (c(S) < z && n < k) do {
  sort(L)
  split(R1) → [R'1,1..R'1,10]
  for each R'1,j (j in 1..10) {
    if (R'1,j ∩ T != ∅) {
      L = L + R'1,j
    }
  }
}
```

That is:

- Start with four (large, completely covering) rectangles
- While the coverage area of the union of rectangles is less than the specified accuracy, and more rectangles are allowed...
- *Sort* the current list of rectangles by some metric
- *Split* the "worst" rectangle into 10 smaller child rectangles (either horizontally or vertically, as specified in Section 3.3)
- Add each one that intersects the target area to the list of candidate rectangles, and recalculate

3.2 *sort*() Function

A sorting function must be derived in order to determine the next rectangle to split. The exact function used can vary, and is left to the discretion of the implementer: different functions will produce results of varying quality. Below are examples of factors to consider when deciding to split a given rectangle R_i, and these can be combined and weighted together to produce a sorting function:

- Coverage ratio of R_i
- Number of potential child rectangles that would intersect the target shape after a split. Define this as $|d(R_i)|$. More formally:

$$d(R_i) = \left\{x \in R'_{i,j} | x \cap T \neq \varnothing\right\} \tag{1}$$

  This metric is important as each rectangle corresponds to a topic subscription, and generally these are considered a "scarce" resource.
- Coverage ratio of each of the child rectangles. E.g. how many children have a coverage ratio of 1?
- Size of the rectangle: $A(R_i)$. E.g. prefer to split larger rectangles before smaller ones.
- Range between largest and smallest rectangles in the union. E.g. only want to have one order of magnitude difference between largest and smallest rectangles.

One example of a weighting for the sort function could be defined as (bigger numbers are more likely to split):

$$(1 - c(R_i)) \cdot \frac{A(R_i)}{1.2^{|d(R_i)|-1}} \tag{2}$$

That is, the "ratio inverse" of the coverage ratio (rectangles that are less covered are more likely to get split), times the area of the rectangle (split bigger ones first), divided by a weighting between 1 and 5.16 (1.2^0 to 1.2^9) which favours splitting rectangles with less children. The constant value of 1.2 was chosen through experimentation.

The reasons for defining the algorithm as above:

- Ratio: when comparing two rectangles of the same size and with the same number of possible resulting children, it is preferable to split the rectangle that is only 10% covered rather than one that is 60% covered,

as this will improve the total coverage ratio more, while resulting in the same number of children (i.e. subscriptions)
- Area: if both the coverage ratios and the resulting number of children are the same, then splitting a larger sized rectangle will improve the total coverage ratio more, while resulting in the same number of children
- Number of Children: if the rectangles' sizes and coverage ratios are the same, it is preferable to choose to split the rectangle that has less children, as this corresponds to less topic subscriptions on the pub/sub system. Something more complex could have been used, such as calculating the coverage area of each child as well and incorporating that into the sort function.

Note that it is often possible to get a split "for free": when considering a rectangle to split, and the target area intersects only one of the resulting 10 children rectangles, that is $|d(R_i)| = 1$, then the rectangle can be split immediately, and replaced by the single child rectangle. This follows as the total number of rectangles doesn't change, and the coverage ratio improves.

3.3 *split*() Function

Similar to the sort function, different split functions can impact the final result of the algorithm. When it is decided to split a particular rectangle into 10 smaller child rectangles, the orientation in which to split the rectangle depends on a few factors. For this implementation this paper is based on, the split function is defined to follow the rules below:

- If the rectangle is horizontal / wide as in Figure 3, split it with vertical cuts into 10 square-shaped grids
- If the rectangle is vertical / tall, split it with horizontal cuts into 10 square-shaped grids
- If the rectangle is square-shaped, then the shape of the area defined by the intersection of the target and rectangle must be considered:

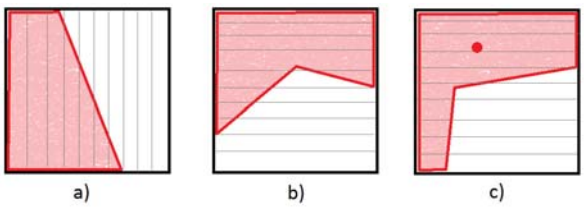

Fig. 5: Choosing a vertical or horizontal split orientation

a) If the intersecting shape has more of a vertical tendency (i.e. height > width), the square should be split with vertical cuts into 10 vertical rectangles due to the likelihood of being able to discard child rectangles from future consideration if they do not intersect the target.
b) Similarly, if the target shape is more horizontal, split the square with horizontal cuts into 10 horizontal rectangles.
c) Otherwise, another metric is employed, such as using the centroid of the intersection, or possibly splitting the square into 100 smaller squares (double split). Again, the exact implementation on how to choose which way to cut is left to the implementer.

After the split has occurred, the coverage areas for each of the 10 new children rectangles are calculated:

- If the child's coverage ratio is 0 (meaning the child doesn't intersect the target T at all), it therefore does not need to be considered, can be discarded, and this child is not added to the list L of rectangles
- If the child's coverage ratio is 1 (meaning it is completely covered), then it would make no improvement to the overall coverage ratio to split this child further. The sort function should disregard any such rectangles with coverage ratio = 1, yet it is still added to L.
- Else, the rectangle is partially covered by the target shape, and is added to the list of potential split candidates, with a weighting defined by the sorting function.

Iteratively perform the steps outlined in Section 3.1 until one of the two terminating conditions are met: either the number of subscriptions is reached, or the desired accuracy of the output is achieved.

3.4 Further Implementation Details

This section discusses some insights gleaned during the implementation of the algorithm.

As mentioned at the end of Section 2.2, the GPS coordinates within the topic string must be zero-padded, both before and after the decimal point. The amount of padding after the decimal must be at least the maximum number of digits in the topic subscription. That is, if the subscription had two decimal places (e.g. `_45.30*`), the topic must have at least two as well (e.g. otherwise `_45.3` would not match, as it requires the 0). Civilian GPS systems have accuracy around 3-4 metres, so using five or six decimal places of accuracy should be sufficient for most applications; this yields a resolution of approximately 1 metre or 10 cm respectively [5].

At the termination of the algorithm, the various rectangles that are generated as output must be converted into topic subscriptions for the filtering to occur in the message bus. As stated at the beginning of Section 3, the topic subscription format has different signs, whether $+$ or $-$, depending on which quadrant (NE, NW, SE, or SW) the rectangle is in. The quadrant will determine the "inner corner" of the rectangle, or corner of the rectangle closest to the earth's origin $(0, 0)$, which is very useful when converting to the topic subscription.

The algorithm does not specify how to calculate if a particular rectangular area intersects the given input target geometry. A simple approach would be to treat the latitude/longitude coordinates as planar coordinates, and

perform planar geometry intersection calculations. This approach would provide sufficient accuracy at small scales (e.g. urban centres), but very large polygon edges would not follow the great circle route. A more advanced option would be to use spherical geometry, or ensure the various areas are represented by geodesic polygons.

Care must be taken for polygons that traverse the International Date Line, and for polar contained areas.

4. Conclusion and Discussion

The use of this algorithm and technique has broad applicability across industry and government. It allows the filtering of location data to a specified area within the message bus layer, thereby reducing data bandwidth and CPU load on clients by not having to filter locally.

As mentioned in Section 1, this technique can be extended to cover ranged queries in other domains, as well as to more dimensions. For example, it would be easy to add a 3rd topic level to describe elevation, and could therefore be used for aviation geo-fencing.

Algorithm runtime complexity was not evaluated as part of this paper due to the rather small (e.g. low thousands) number of iterations through the main loop to produce very accurate approximations to the input geometry, even with a very complex shape with many vertices and edges.

This algorithm seems to work well for "boxy" or convex shapes, especially those that are more axis-aligned. The algorithm tends to require a larger number of subscriptions to approximate star-shaped polygons, or polygons that have long diagonal edges, as these require many "staircase steps" to approximate sufficiently.

One disadvantage to this approach is that it only works for matching point coordinates to polygons: it cannot do line-to-line intersection, or polygon-to-polygon matching.

For a working demonstration of this algorithm, please visit: http://london.solacesystems.com/aaron/geo/

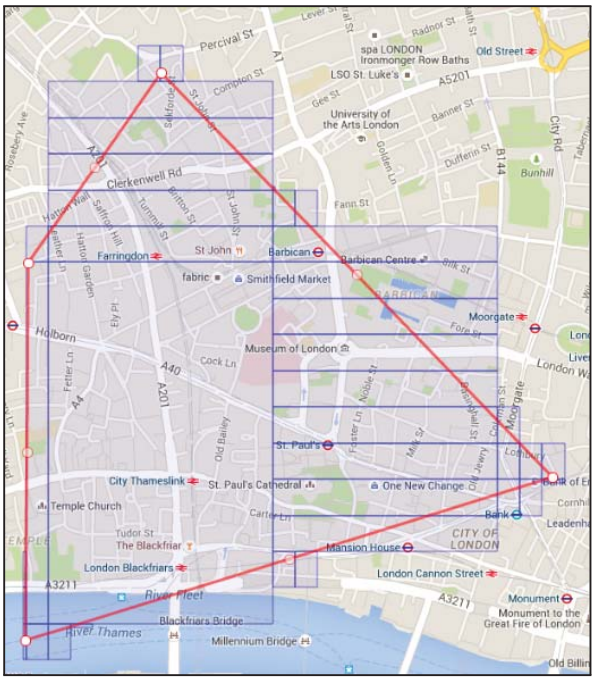

Fig. 6: 37 rectangles, corresponding to 37 subscriptions, approximating a target polygon with 75% accuracy
Background Map data ©2016 Google

References

[1] Wikipedia: "Message-oriented middleware", retrieved March 12, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Message-oriented_middleware
[2] Wikipedia: "Message broker", retrieved March 12, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Message_broker
[3] Wikipedia: "Publish-subscribe pattern", retrieved March 12, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Publish-subscribe_pattern
[4] Solace Systems: "Controlling Information Flow with Topics", May 2012. [Online]. Available: http://www.solacesystems.com/techblog/controlling-info-flow-with-topics
[5] Wikipedia: "Decimal degrees", retrieved March 12, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Decimal_degrees
[6] MQTT version 3.1.1, OASIS Standard, December 10, 2015. [Online]. Section 3.3.2.1, Topics: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html