Map Marker Clustering Strategies for the Maps API for Java ME

This article explains several strategies for map marker clustering. This means that only a some of the data points associated with an application will have a `MapMarker`, other data points will be replaced by a summary instead.

Introduction

The number of `MapMarkers` which can be displayed on screen on a typical series 40 device is severely limited. This is due both to the small screen size of the phone and due to the limited capabilities of the device itself. If too many `MapMarkers` are displayed close together, the resulting map is difficult to see as the `MapMarkers` tend to both overlap and obscure the underlying map. It may be worth considering **clustering** the data points, this means that only a some of the data points associated with an application will have their own `MapMarker`, other data points will be replaced by a summary instead.

The images below show an example of `MapMarker` clustering at three different zoom levels, over one hundred data points (shown in blue) are reduced to half-a-dozen clusters of various sizes. When the zoom level is increased, the clusters split and more noise points can be seen.

Simple client side clustering

Using simple client side clustering should not be considered as a serious solution for achieving marker clustering in Java ME, but helps to illustrate some of the issues involved, and can be read as an introductory example. The technique used here for clustering is also deliberately kept simple and is not optimal. Anyone looking seriously at a more efficient algorithm should consider using a quadtree

The basis of any clustering algorithm is relatively simple, each data point is processed in turn, if it is **near** to an existing `Cluster` it is added to that `Cluster`, otherwise a new `Cluster`is created and the datapoint is added to that instead. This process is repeated for every datapoint. This is achieved in the `clusterObjects()` method as shown below:

```
private synchronized void clusterObjects(GeoBoundingBox box,
   double clusterPhysicalDistance) {

  MapObject[] mo = container.getAllMapObjects();
  for (int i = 0; i < mo.length; i++) {
   GeoCoordinate center = mapObjectToGeo(mo[i]);
   if (box.contains(center)) {
    if (!isClustered.contains(mo[i])) {
     isClustered.addElement(mo[i]);
     Cluster closest = getClosestCluster(center);
     if (closest != null
       && center.distanceTo(closest.getCenter()) < clusterPhysicalDistance) {
      closest.add(mo[i]);
     } else {
      clusterContainer.addElement(new Cluster(mo[i]));
     }
    }
   }
  }
 }
```

Passing in a `GeoBoundingBox` holding the edges of the current map is a simple method to achieve **viewport marker management** - only markers which are visibile are kept for further processing.

As well as holding an array of associated markers, any `Cluster` needs two further properties - a center point and a radius. The simplest way to set a centre point would be to use the location of the first marker in the cluster. The `clusterPhysicalDistance` should be related to the viewport size e.g. the distance between the top and bottom of the map. Finding the nearest cluster can be obtained by iterating through the existing clusters as shown. the distances between objects are obtained using the `Geocordinate.distanceTo()`, which implements the Haversine Formula.

(C) Copyright Nokia 2013. All rights reserved.

```
private synchronized Cluster getClosestCluster(GeoCoordinate coord) {
  Cluster closest = null;
  double curDist = -1;
  Enumeration e = clusterContainer.elements();

  while (e.hasMoreElements()) {
   Cluster cluster = (Cluster) e.nextElement();
   if (curDist == -1
     || (coord.distanceTo(cluster.getCenter()) < curDist)) {
    closest = cluster;
    curDist = coord.distanceTo(closest.getCenter());

   }
  }
  return closest;
 }
```

All of the code above is encapsulated into a custom `MapComponent`, the component doesn't respond to events (and therefore does not require an event listener) nor does is it visible (and therefore the `paint()` method can be left blank . However it does need to override the `attach()` and `mapUpdated()` methods as shown:

```
public void attach(MapDisplay map) {
  this.map = map;
  coord = map.getCenter();
  thresholdDistance = map.getCenter().distanceTo(
    map.pixelToGeo(new Point(map.getWidth() / 4,
      map.getHeight() / 2)));
 }

public void mapUpdated(boolean zoomChanged) {

  if (zoomChanged) {
   clearClusteredObjects();
  }

  if (zoomChanged
    || map.getCenter().distanceTo(coord) > thresholdDistance) {
   coord = map.getCenter();
   thresholdDistance = map.getCenter().distanceTo(
     map.pixelToGeo(new Point(map.getWidth() / 4, map
       .getHeight() / 2)));
   if (container != null && clustering == false) {
    clustering = true;
   }
  }
 }
```

The `thresholdDistance` is used as a trigger to update the clusters if the map view has moved too far. Obviously if the zoom level has changed, the `MapMarkers` will also need re-clustering. This is not done immediately (since the user will not want to wait for the map to render fully whilst panning) but a flag is set to note that re-clustering will be required.

A good indication that panning has stopped, is the fact that the map has completely refreshed. This is indicated by the `onMapContentComplete()` event on the `MapCanvas`. Therefore the call to cluster the data is made there:

```
public void onMapContentComplete() {
   clusterer.cluster();
  }
```

The actual clustering is made on a separate thread as shown:

```
public void cluster() {
  if (clustering) {
   clustering = false;
   Thread t = new Thread(new Runnable() {

    public void run() {
     GeoCoordinate d1 = map.pixelToGeo(new Point(0,
       clusterPixelDistance));
     GeoCoordinate d2 = map.pixelToGeo(new Point(0, 0));
     clusterObjects(
       new GeoBoundingBox(
         map.pixelToGeo(new Point(
           -clusterPixelDistance,
           -clusterPixelDistance)),
         map.pixelToGeo(new Point(map.getWidth()
           + clusterPixelDistance, map
           .getHeight() + clusterPixelDistance))),
       d1.distanceTo(d2));
     addClusteredObjects();
    }
   });
   t.start();
  }

 }
```

Problems with client side clustering

The simple client side clustering example loads the locations of over one hundred table tennis tables in the Berlin area and clusters the result. The example itself helps to demonstrate some of the problems of using large data sets in Java ME. Firstly even though the data set is relatively small, it takes the device a long time to process the data and the lead in before something is displayed on the map is unacceptably long. A delay also occurs when the map is moved since **all** the markers need to be reprocessed. This has been mitigated where possible by checking that `Markers` are both within the viewport and not unclustered and re-clustered necessarily, but even this is insufficient to make the example usable in a real application. Furthermore because each point needs to find the nearest `Cluster` as a candidate to join, the solution will not scale since the number of operations to find the candidate will rise exponentially.

 (C) Copyright Nokia 2013. All rights reserved.

Resource file pre-clustering

The problem with loading a large data set can be reduced by pre-processing the data into clusters and noise points, this removes the need calculate clusters each time, but serves to illustrate a different problem associated with large data sets.

A series of resources files have been created, one for each zoom level as shown:

```
<?xml version="1.0"?>
    <markers>
        <cluster lat="52.4907" lng="13.4726" size="57" />
        <cluster lat="52.4172" lng="13.3538" size="18" />
        <cluster lat="52.5953" lng="13.292" size="14" />
        <cluster lat="52.5118" lng="13.1878" size="7" />
        <marker lat="52.4455" lng="13.6763" text="2 tables in Strandbad Mueggelsee" />
        <cluster lat="52.6249" lng="13.4826" size="2" />
        <cluster lat="52.3938" lng="13.0289" size="3" />
    </markers>
```

Again a custom `MapComponent` is created, this time the `attach()` and `mapUpdated()` set a pair of flags - to either refresh the view port, or to load a new resource file.

```
     public void attach(MapDisplay map) {
  this.map = map;
  coord = map.getCenter();
  thresholdDistance = map.getCenter().distanceTo(
    map.pixelToGeo(new Point(map.getWidth() / 4,
      map.getHeight() / 2)));
 }

 public void mapUpdated(boolean zoomChanged) {

  if (zoomChanged) {
   readXMLthenRecluster = true;
   thresholdDistance = map.getCenter().distanceTo(
     map.pixelToGeo(new Point(map.getWidth() / 4, map
       .getHeight() / 2)));
  }
  if (map.getCenter().distanceTo(coord) > thresholdDistance) {
   viewportChanged = true;
  }
 }
```

The `cluster()` method itself reloads data from a file using the method described in the Reading XML Map Marker data with the Maps API for Java ME article

```
public void cluster() {
  if (readXMLthenRecluster) {

   // Clear the map.
   map.removeAllMapObjects();

   readXMLthenRecluster = false;
   viewportChanged = false;

   int zoom = map.getZoomLevel();
   zoom = (zoom < 7) ? 7 : (zoom > 16) ? 16 : zoom;
   mapDataProcessor.getMarkers().removeAllMapObjects();
   parser.parse(
     getClass().getResourceAsStream("/zoom/" + zoom + ".xml"),
     // Pass in data of a known format.
     mapDataProcessor, // Include a processing delegate which
          // will
     // handle this format.
     this);

  } else if (viewportChanged) {
   viewportChanged = false;
   addObjectsInViewPort();
   removeObjectsNotInViewPort();
  }

 }
```

The callback method from this `onParseComplete()` then adds the processed `Markers` and `Clusters` onto the map.

Problems with resource pre-processing on Device

Even though pre-clustering has significantly reduced the latency of the app, Resource file pre-clustering is not considered as a serious contender for displaying large data sets. As can be seen from the code resource files, the gain of speed comes at a price - the size and number of resource files required has increased tenfold. Given the highly restricted memory environment of a typical Java ME phone, it should be obvious that the phone itself should not be pre-processing or holding the data of processed clusters, rather that these operations should be done **server-side** and the API should merely load and display the processed data.

Server side-clustering

Tip: The code behind this server-side clustering example has been included the latest 1.3 release of the HERE Maps API for Java ME - see HERE Map Code Examples. The code has been designed to be fully backwards compatible for older phones . The API has been integrated as a plug-in into the Asha SDK 1.0.

Since most Java ME phones support JSR-172, the **XML** data format would be an obvious candidate to use to upload marker and cluster data. The choice of back-end server technologies to use is up to you, and it is not proposed to go in to details of how to produce **XML** output for any particular server stack, but a couple of example links should give an indication of how to create your data:

- PHP and MySQL to XML

 (C) Copyright Nokia 2013. All rights reserved.

- PHP and MySQL to XML
- PERL and simple text file to XML

Note: When a datapoint is added to the database, it could be automatically added as a marker at the lowest level and clustered at each zoom level above, so a request like `.../cluster.xml/?lat1=56.891&lng1=8.129&lat2=47.754&lng2=18.676&zoom=7` would just make a SQL query such as:

```
SELECT * FROM markers WHERE lat> 47.75 AND lat < 56.891
AND lng > 8.129 AND lng < 18.676
AND zoom =  7
```

This is a simple case of view port marker management since only **XML** data within the bounds will be returned.

The custom `MapComponent` for reading server side clustered data onto the device is very similar to the previous resource based code, again `attach()` and `mapUpdated()` are implemented to set a boolean flag which is consumed by the `cluster()` method

```
public void attach(MapDisplay map) {
  this.map = map;
  coord = map.getCenter();
  thresholdDistance = map.getCenter().distanceTo(
    map.pixelToGeo(new Point(map.getWidth() / 4,
      map.getHeight() / 2)));
 }

public void mapUpdated(boolean zoomChanged) {

  if (zoomChanged) {
   thresholdDistance = map.getCenter().distanceTo(
     map.pixelToGeo(new Point(map.getWidth() / 4, map
       .getHeight() / 2)));
  }
  if (zoomChanged || map.getCenter().distanceTo(coord) > thresholdDistance) {
   // The XML should only hold the items in the viewport.
   readXMLthenRecluster = true;
  }
 }
```

```
public void cluster() {
  if (readXMLthenRecluster) {

   readXMLthenRecluster = false;
   GeoBoundingBox viewPort = new GeoBoundingBox(map.pixelToGeo(new Point(
     0, 0)), map.pixelToGeo(new Point(map.getWidth(), map
     .getHeight())));

   try {
    parseXMLFromServer( viewPort);
   } catch (IOException ioe){
    System.out.println(ioe.getMessage());
   }

  }
```

The `parseXMLFromServer()` method, creates the server URL and obtains the data from the server.

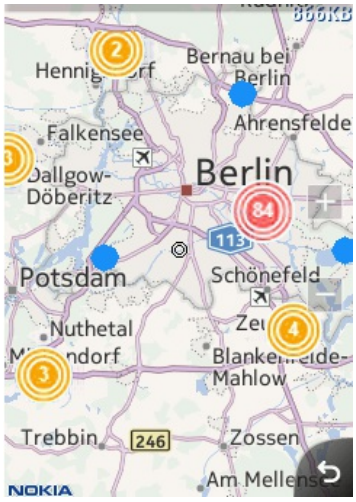
```
private void parseXMLFromServer(GeoBoundingBox viewPort) throws IOException {

  HttpConnection http = null;
  InputStream in = null;

  int zoom = map.getZoomLevel();
  zoom = (zoom < 7) ? 7 : (zoom > 16) ? 16 : zoom;
  mapDataProcessor.getMarkers().removeAllMapObjects();
  StringBuffer buf = new StringBuffer(server);
  buf.append(zoom);
  buf.append(".xml");

  buf.append("?lat1=");
  buf.append(viewPort.getTopLeft().getLatitude());
  buf.append("&lng1=");
  buf.append((viewPort.getTopLeft().getLongitude()));
  buf.append("&lat2=");
  buf.append(viewPort.getBottomRight().getLatitude());
  buf.append("&lng2=");
  buf.append((viewPort.getBottomRight().getLongitude()));

  try {

   http = (HttpConnection) Connector.open(buf.toString());
   http.setRequestMethod(HttpConnection.GET);
   if (http.getResponseCode() == HttpConnection.HTTP_OK) {

    int length  = (int) http.getLength();

             in = http.openDataInputStream();
             if (length > 0)
             {
               byte serverData[] = new byte[length];
               in.read(serverData);
               parser.parse(new ByteArrayInputStream(serverData),// Pass in data of a known format.
       mapDataProcessor, // Include a processing delegate which
            // will handle this format.
       this);

             }
   }
  } finally {
   // Clean up
   if (in != null)
    in.close();
   if (http != null)
    http.close();
  }

 }
```

`MapMarkers` are created when the XML Parser has returned in the same manner as the previous example.

(C) Copyright Nokia 2013. All rights reserved.

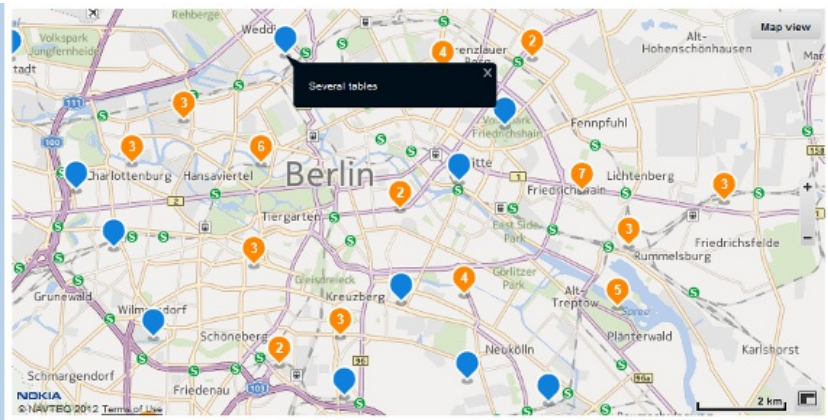
Use of Overlays

It should be noted that the amount of processing on the client side could be reduced even further by creating a `MapUrlProvider` which returns the cluster points as an overlay from the server side. This would mean that only the unclustered markers needed to be returned in the **XML** query and processed. Whilst this reduces the **XML** data required (and associated marker processing), it increases the data traffic for each map tile as it would need to request an overlay tile as well. A `MapUrlProvider` could be used to make the cluster markers prettier if `MapMarkers` are not sufficient.



Summary

The Java ME environment and Series 40 phones are not well suited for processing and displaying large numbers of `MapMarkers` simultaneously. For the best results, when working with large data sets, pre-processing should be done **server side** far as possible, and the API should be used as a presentation layer using a minimal amount of data points. Through the use of server side view port marker management and the addition of overlays, massive data sets can be reduced to a manageable amount. of data. Of course **XML** and **server side clustering** may also be used to offer the same data set to other platforms such as in this demonstration of a simple HTML Webpage using AJAX shown below:



The location of over one hundred Table Tennis tables in Berlin.

Click on the blue markers for details, the orange markers are clusters.

 (C) Copyright Nokia 2013. All rights reserved.

 (C) Copyright Nokia 2013. All rights reserved.