Towards Real-Time Emergency Response using Crowdsourcing

Ioannis Boutsis, Dimitrios Tomaras, and Vana Kalogeraki
Department of Informatics
Athens University of Economics and Business
Athens, Greece
{mpoutsis, tomaras, vana}@aueb.gr

ABSTRACT

Crowdsourcing has emerged as an attractive paradigm in recent years for information collection for disaster response, which utilizes data received from the human crowd, to provide critical information collection and dissemination during emergency situations and visualize this data to generate emergency maps for the human crowd. In this paper we investigate the use of crowdsourcing mechanisms for real-time emergency response and describe our approach for developing a crowdsourcing tool that can be effectively used to formulate questions and seek answers from the human crowd using a MapReduce programming model, and integrate this information into a novel spatiotemporal data structure and create a visual emergency map. Our experimental evaluation shows that our approach is practical, efficient and can be used for applications with real-time demands.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]

Keywords

Distributed Sensor Systems, Crowdsourcing, SpatioTemporal Data, Emergency Response

1. INTRODUCTION

Over the recent years, the prevalence of social networks, the ubiquitous sensing capabilities and the widespread adoption of smartphones, are driving the development of applications and services that are changing the way we interact with the world and each other. Smartphones equipped with various heterogeneous sensors along with the ubiquitous connectivity of these devices have introduced new trends on application development, by providing important real-time geo-located data.

At the same time, crowdsourcing has emerged as an attractive paradigm that can leverage the collective intelligence of these mobile workers quickly and inexpensively, to extract useful information. Typical crowdsourcing systems constitute marketplaces for tasks, such as AMT [1], mCrowd [9] etc.; these allow humans or automated systems to define tasks, while human workers execute them in exchange for a reward. For instance, mobile human workers can easily provide traffic information regarding their location, without needing any expensive infrastructure. Our aim is to exploit mobile users to perform geo-located crowdsourcing tasks to extract useful information.

One fundamental challenge in such a setting is how to store and efficiently retrieve location-based crowdsoucing data provided by mobile users during emergency situations. Finding densely populated geographic regions in real-time is an important capability to city personnel, police departments, etc., especially during rescue and recovery efforts. More specifically, in this paper, we focus on user trajectories, that represent the route that a mobile phone user has followed, along with the respective crowdsourcing data. These can be either low or high sampling trajectories depending on the frequency of the tasks assigned to the user and the corresponding answers. Our goal is to provide an efficient way to index the crowdsourcing data under different levels of spatial and temporal granularity.

In this paper we present T-Crowd our crowdsourcing system that can be effectively used for processing and indexing geo-located crowdsourcing tasks during emergency situations. We summarize our contributions below:

- We develop a crowdsourcing system used to formulate questions and seek answers from the human crowd using a MapReduce programming model.
- We propose a novel spatiotemporal index structure for user mobile trajectory data, to create indexes over different geographic region granularities.
- We provide an experimental evaluation, that illustrates that T-Crowd is practical, efficient and can answer range queries extracted from crowdsourcing tasks in real-time.

2. A MOBILE CROWDSOURCING SYSTEM

Our Crowdsourcing platform has been developed using the Misco system [4, 3, 5] , a MapReduce framework tailored for mobile devices. MapReduce requires that the computational process will be decomposed into two steps, namely map and reduce tasks. Misco adopts a distributed task programming model based on the MapReduce model, which is a extensible and efficient way to program applications. It provides

PETRA 2014, May 27-30, 2014, Island of Rhodes, Greece.

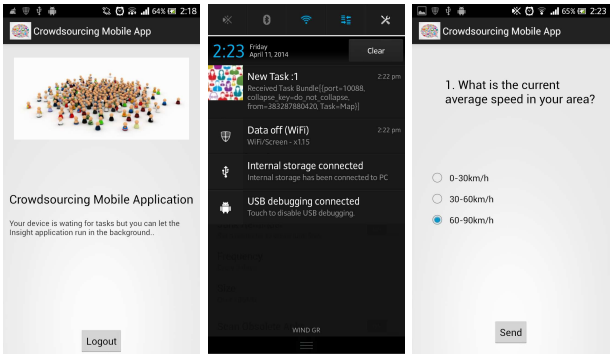

Figure 1: Crowdsourcing Mobile Application (a) Waiting Screen, (b) Push Notification, (c) Map Task

a powerful programming abstraction to ease software development, while hiding many of the distributed computing complexities from the programmer.

2.1 The T-Crowd Mobile App

Our architecture is based on the MapReduce programming model and structured as one logical *Master Server* and a number of *Worker Nodes* (the human contributors in the system take the role of the worker nodes). The Server is in charge of keeping track of query requests submitted by the users and assigning tasks to Workers. The Server is multi-threaded, spawning a new thread to handle incoming worker connections. Queries are instantiated and managed by human requesters, using a browser interface.

The main responsibility of the Worker node is to process map tasks and return the results to the server. Each task is characterized by a unique $TaskId$ for the task to execute, the *location* of the task in terms of latitude and longitude, an estimation of its *urgency* and potentially the task *description*. Crowdsourcing tasks can be typically executed by workers through their personal devices such as mobile devices or cellphones. Nevertheless our system focuses on mobile users since the location plays an important role on the user selection.

2.1.1 Task assignment

Suppose that the crowdsourcing server has selected a subset of the users to execute a crowdsourcing query. We describe the step-by-step sequence followed in each of the Android devices to process the query.

In the implementation described below we show how we utilize the Android SDK that provides the API libraries and developer tools necessary to build, test, and debug apps for Android[1]. Every Android application is typically a separate process that can be composed of the following components: **Activities** for graphical display, **Services** for background tasks, **Content Providers** for accessing persistent data and **Broadcast Receivers** for receiving notifications.

Since users are mobile they might change connections often, such as switching from WiFi to 3g. In order to be able to track the users without being restricted by the type of connection and possible private IP restrictions (NAT) we use Push Notifications to initiate the communication with the users of the crowdsourcing application. Push notification services (PNS) allow users to register for delivery of messages. An application server sends the messages to a connection server, which is operated by the platform owner and pools messages from all application servers a particular user has registered for, and sends the pooled messages to the user. Such push-notification services are supported in every major mobile (Google Cloud Messaging service (GCM) in Android, Apple Push Notification Service in iOS and Microsoft Push Notification Service in Windows Phone).

In order to start receiving tasks, the user would first need to login to our system. During normal operation the T-Crowd app would run in the background, since we cannot expect the users to constantly use the App (Figure 1a). When the server selects a user to execute a crowdsourcing task the crowdsourcing server delivers a push notification to the user devices, selected from the Scheduling policy, through the Push Notification service, to trigger the application. These notifications are implemented as the Broadcast Receivers for Android. Once the `BroadcastReceiver` receives the notification it alerts the user (Figure 1b). If the user opens the notification an Alert Dialog Box pops-up on the screen, so that the user can decide whether to accept the task execution. In the case that the user decides to accept the task the Crowdsourcing application is triggered and the task will be displayed in the user screen (Figure 1c). Finally, the combination of all the answers in the reduce phase will provide the result of the crowdsourcing task. In our system we store the answers of each individual user as described in the following section. This enables us to perform aggregation queries in different ranges than the original query, by exploiting results from several crowdsourcing tasks.

3. THE *DATA STORAGE* STRUCTURE

In the bibliography, one can find different types of data structures that can be used as indices [2, 6]. Each of them has its own purpose, some of them such as R-trees and STR-trees have similar methods of organizing the data into their leaves, similar insertion and splitting strategy, but with different goals on what information can implicitly or explicitly be stored.

3.1 Enhanced Trajectory Bundle Tree

Our approach to the solution makes use of Trajectory Bundle trees [2],[7]. Trajectory Bundle trees are a special kind of R-trees, that are very useful to create indices over geographic regions, using rectangular boundaries, known as mininum round boxes (MRB) or minimum round rectangles(MRR). Trajectory Bundle trees store whole trajectories on their leafs, and to make it more specific, each node stores M segments(fanout) from only one trajectory. In the case that the trajectory is splitted to more than M segments, a new leaf node is created and both leaf nodes are connected through a double-linked list. We developed an enhanced version of the TB-trees and from now on we will refer to it as ETB-tree, that works as follows:

3.1.1 Segments

Trajectory segments can be modelled by using the following information: 1) *SegID* : the ID of the segment which is unique, 2) *TrajectoryID* : the ID of the trajectory to which the segment belongs to, 3) *Starting and End Point* : the starting and the end point of the trajectory segment. We use this format in order to store all segments in a database.

[1]Android platform: http://www.android.com/

However, on our extended tests with two different datasets, we store this kind of info on the node, i.e. each leaf nodes uses two integers for storing *trajectoryID*,*userID* and some arrays in order to keep *SegIDs*, *segment latitude*, *segment longitude* respectively, but we are on the process of removing unnecessary information and to keep what is really necessary.

3.1.2 Leaf Nodes - Internal Nodes

Some characteristics are the same for both types of nodes. Each node has its unique *Node ID*, used for the storage on a hashmap. We use a hashmap that maps <*Node ID*> to <*Node Reference*> in our ETB-tree implementation for keeping all the references of our nodes in order to achieve fast fetching results from the hashmap.

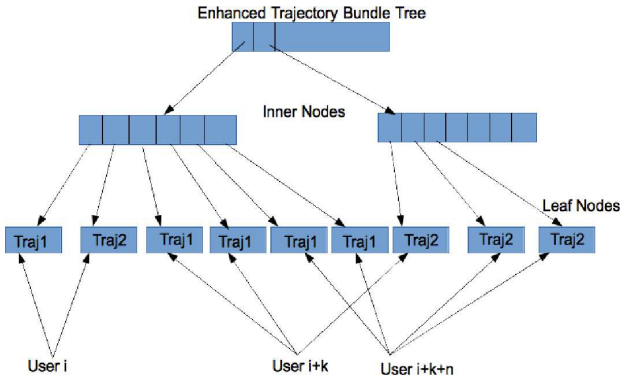

Figure 2: An enhanced Trajectory Bundle Tree.

Internal Nodes. Inner nodes follow the philosophy and the design of those used in R-trees. Inner nodes also follow the same policy with leaves. Thus, they have a maximum capacity for storing IDs of MRRs. Also, they do not just keep the minimum but also the maximum coordinates of IDs inserted to this node and also the minimum and the maximum timestamp. In other words, this describes the coverage area that each node overlaps and also the time range during which this area is covered. Inner nodes also store information about the current number of entries of the node, and also they keep information about the level of the tree they belong. In our implementation, we take advantage of this information, by using the level of inner nodes in order to initiate the creation of leaves. In the case of adding a specific segment that will initiate the process of creation of a new leaf, a "split" process, almost same as that of original TB-trees (where we find the most right free entry to insert the new leaf), will also begin and will update all the necessary info (*MRR limits and timestamp limits*) up to the root of the tree.

Leaf Nodes. In leaf nodes, we need to keep the fanout, denoted as *maxNumKeptSegments*, *i.e.*, the capacity of segments that each leaf can store, which is the same for all trajectories. Traditional TB-trees maintain a double linked list of nodes of the same trajectory on the leaf level, i.e., leaf nodes of the same trajectory are connected with a double-linked list in order to retrieve trajectory segments with trajectory identifier as the key and in such way that will preserve trajectory evolution. On the contrary, we use hashmaps in order to achieve better results over the fetch procedure of same trajectory nodes. We also store the *userID* as one more key element to preserve one of the fundamentals of the traditional TB-trees, which is that different trajectories are stored in different leaf nodes. Finally, we need to store the segments of the trajectory and for this case we use an array initialized to *maxNumKeptSegments*. In the case that there is no available space to store the segment, our method returns a negative number that will initiate the process of adding a new leaf node, with a splitting strategy that is described in the next section. Leaf nodes also maintain a hashmap for mapping segment ids to a pair of timestamps:

SegID,<Segm. Timestamp Start , Segm Timestamp End>

This strategy leads to a very handy way of coping with time window queries, or even queries for a specific time interval, as every leaf node keeps track of the timestamped values of stored segments, and also the minimum and the maximum timestamp value.

3.2 Algorithms of ETB-tree

3.2.1 Insertion - Splitting Algorithm

Our approach inherits the insertion algorithm from traditional TB-trees. We descend the tree until we reach the last level of inner nodes by using MRR limits. This is where we can find the candidate parent node of our leaf, into which we are going to store the segment id. Then, we search for the candidate leaf, by using the combination of *trajectory ID, MRR limits, and user ID*. In the case of a different trajectory, or user, or even if we reach full capacity of the current candidate node, we follow a right-most policy of node insertion. In other words, we find the rightest free child, if it does not exist, we create it and add it as a child to the current parent node. If the maximum capacity of node is reached, then we trigger a splitting-like process that can be transferred up to the root of the tree. Following this policy, one can observe illustratively that our tree expands from left to right, in all levels, as the splitting process is being triggered, same as with the TB-tree.

3.2.2 Retrieval Algorithm

Part of our contribution is over the retrieval algorithms used for support of many kind of queries, especially over navigational queries. We take advantage of hashmap fundamentals, as we know that we have a fast retrieval of any value stored. For example, when we want to execute a range query to the tree, despite running a BFS algorithm, we take advantage of the fact that node references are stored to the hashmap, and for this case, we make a linear search over nodes of last inner level that overlap with the requested MRR limits. For that reason, instead of querying N nodes, we just query $\log_C N$ nodes, where C is node capacity and N the total number of nodes.

4. EXPERIMENTAL EVALUATION

For our experiments we have developed a dataset that contains 23577330 segments, very dense-sampled (every 5 secs), stored on a MySQL database server. Experiments were conducted on a Linux Server provided by GRNET academic program "okeanos", dual core 2.1Ghz, with 6GB Ram.

The following figures provide the results for two different kinds of queries: (i) the average speed into a specific MRRed region within a time window and (ii) the average length of

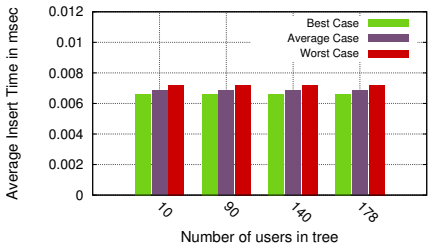

Figure 3: Average Insert Time

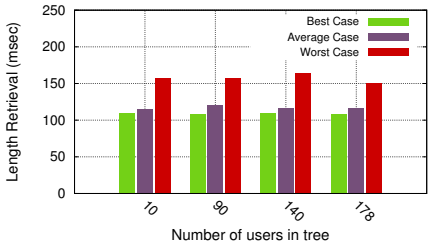

Figure 4: Average Length Retrieval Time

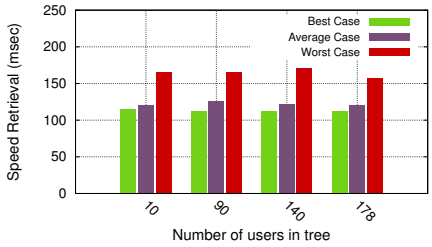

Figure 5: Average Speed Retrieval Time

trajectories in this region. We provide results for (i) the worst case, where we need to search the whole MRR range and time window kept by our tree, (ii) the average case over all queries to fetch the results from our tree, examining a rather large area defined by the MRR range, but not as big as in the worst case, and (iii) the best case, where we just search for a small limited area, to retrieve the results regarding the defined queries.

Figure 3, shows the time needed for the insertion of a segment in our tree. As can be observed the time needed to insert a segment remains almost the same as the users increase, due to our structure. In addition, the insertion time is less than 0.008ms even in the worst case, providing a high speed solution for such systems, where we expect multiple parallel insertions for different workers. Moreover, in figure 4 and 5, we present the time needed to retrieve the aggregated data for the defined queries. As the figures show, the time needed for all cases is almost the same, despite the population of users stored in the tree. Another interesting result is that the time needed for average speed retrieval and for average length of trajectories retrieval in the average case is almost the same with that of the best case.

5. RELATED WORK

Storing and indexing the users' movements has attracted interest in recent years in different application domains, such as traffic monitoring, road and social recommendation systems, ride-sharing applications and personalized driving directions. Several papers have been proposed in the context of indexing and storing trajectory data [2, 7, 8]. However, most of these works either present different kind of methods in order to process trajectory data or take advantage of users repetitive behavior such as traveling back home or going at work, at which they tend to follow similar paths. In [2], there is the presentation of the data structure of TB-tree, that is compared to STR-tree and R-tree, and what types of queries should be supported in general by those kind of indices. Authors in [7] used a traditional TB-tree, as an index, to implement their own FlowScan algorithm for discovering hot routes. In our work, we use this enhanced version of TB-tree to answer different types of queries (mostly combinational queries) such as, time-window and regional queries, while at the same time keeping our data structure, as simple as we can.

6. CONCLUSIONS

In this paper we have presented our crowdsourcing system for efficient processing and indexing of geo-located crowdsourcing tasks during emergency response. Our system comprises the following components: (a) a crowdsourcing system for mobile devices, based on the MapReduce paradigm, that is able to perform geolocated crowdsourcing and crowdsensing tasks and collect user mobile data, and (b) a novel data-structure for preserving the geo-located user answers of the tasks and perform aggregation queries under different levels of location granularity and in real-time. Our experimental study verifies the practicality of our approach and its efficiency for real-time response systems.

Acknowledgment

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thalis - Athens University of Economics and Business - DISFER.

7. REFERENCES

[1] Amazon mechanical turk. http://www.mturk.com/.
[2] Y. T. Dieter Pfoser, Christian S. Jensen. Novel approaches to the indexing of moving object trajectories. *VLDB*, pages 395–406, September 2000.
[3] A. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen, and V. H. Tuulos. Misco: a mapreduce framework for mobile systems. In *PETRA*, June 2010.
[4] A. J. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikinen, V. Tuulos, S. Foley, and C. Yu. Data clustering on a network of mobile smartphones. In *SAINT*, pages 118–127, Munich, Germany, July 2011.
[5] T. Kakantousis, I. Boutsis, V. Kalogeraki, D. Gunopulos, G. Gasparis, and A. Dou. Misco: A system for data analysis applications on networks of smartphones using mapreduce. In *MDM*, pages 356–359, Bengaluru, India, July 2012. IEEE.
[6] K. Z. X. Z. Ke Deng, Kexin Xie. Trajectory indexing and retrieval. *Computing with Spatial Trajectories*, (Chapter 2), 2011.
[7] X. Li, J. Han, J.-G. Lee, and H. Gonzalez. Traffic density-based discovery of hot routes in road networks. In *SSTD*, pages 441–459. Springer, 2007.
[8] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Eng. Bull.*, 26(2):40–49, 2003.
[9] T. Yan, M. Marzilli, R. Holmes, D. Ganesan, and M. Corner. mcrowd: a platform for mobile crowdsourcing. In *SenSys*, pages 347–348, Berkeley, CA, USA, November 2009. ACM.