IT 08 038
Examensarbete 30 hp
Oktober 2008

Searching Through Mobile Geo-Tagged Content

Vladimir Katardjiev

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Searching Through Mobile Geo-Tagged Content

Vladimir Katardjiev

With the rise of Web 2.0, and a more interactive Internet, a greater emphasis has been placed on making services better tailored for the end user; "local" searches for places near the user, or "profiled" searches for music the user likes. At the same time, people have embraced the possibility of geo-tagging their posts and images: associating geographic information with them to convey a meaning or opinion about that specific place. While it is already possible to search through the posts and images organized such, there has been a distinct lack of a Web 2.0-style personalized search, capable of finding the geo-tagged posts and images that a specific user is interested in.

This thesis investigates how to identify a user's interests, using a combination of traditional techniques such as folksonomy tagging and profiling, as well as making use of the geo-location data now available in mobile phones. This data is then used to locate and present to the user the posts that they want to view, both based on their interests, the current location, and the combination of interests at the current location.

Along with the proposed solution, a prototype of it has been developed, testing the feasibility of the solution from an implementation perspective. Furthermore, this prototype has permitted the evaluation of the solution's ability to power mashups – combining two or more existing services, in turn creating a new or unique way of presenting the services' input or output.

Handledare: Göran Eriksson
Ämnesgranskare: Justin Pearson
Examinator: Anders Jansson
IT 08 038
Tryckt av: Reprocentralen ITC

Sammanfattning

Att söka efter information på Internet är något som många personer är bekanta med. Man går till en sökmotor och skriver in ett antal nyckelord som den sedan letar efter. Detta kanske leder till det man letar efter, eller så misslyckas det och man får söka igen, men en sökning går snabbt och man får många resultat samtidigt. Med lite tålamod så fixar det sig till slut och man hittar det man letade efter. Detta fungerar för en stationär eller bärbar dator, men det lämpar sig inte lika bra på en mobiltelefon, där skärmen är liten, knappsatsen är liten och krångligare än ett tangentbord, och det tar lite längre för sökresultaten att komma till telefonen.

Denna uppsats granskar hur man kan göra sökning ifrån en mobiltelefon till ett mer effektivt verktyg som snabbare hittar de resultaten man letar efter. Det görs genom att ta i hänsyn den ytterligare information som finns tillgänglig på en mobiltelefon, såsom vart användaren befinner sig, kunskapen om att endast en person använder en mobiltelefon och metoder man kan använda för att koppla den informationen till s.k. användarprofiler, vars jobb det är att lära systemet vad en användare tycker om och vill se.

Det intressanta för oss är att granska ”geotaggade” inlägg, alltså saker som andra användare har skrivit (”bloggat”) och markerat med vissa beskrivande ord (”taggar”), samt satt en plats på t.ex. en karta där inlägget ska finnas. Denna geografiska information är ”geo”-delen i ”geotaggning”. Vi intresserar os för hur man kan söka igenom inlägg från användare inte med hjälp av nyckelord i en sökmotor utan taggar och platser.

Vi föreslår ett system där varje användare söker efter information från sin mobil om saker som ligger nära, t.ex. turistattraktioner, byggnader, eller s.k. bloggar där man skriver det man vill. Genom att också använda ”taggar” – enstaka eller flera ord som beskriver ett inlägg – kan man utföra sökningar genom att välja en redan förinställd ”tagg” istället för att behöva skriva in ett ord själv. Systemet kan därtill lära sig om man läser inlägg markerade med en viss tagg och välja att självt visa dessa inlägg eller taggar utan att behöva vara tillfrågat. Det tar också till hänsyn vart man brukar läsa dessa, så om en användare läser mycket sport hemma men också använder samma mobil på jobbet där han endast läser om bilar så kommer inga inlägg om sport visas på arbetet.

Vi har också byggt ett sådant system, kallat ”Marco”, som vi har använt för att testa dessa teorier och prestandan. Både en server och klient har skapats, och vi visar att servern kan mycket väl fungera som en sökmotor för geotaggad information, med de föreslagna algoritmerna samt optimerad kod. Det framgår också att de system tillgängliga idag är inte tillräckliga för att utveckla en klient som stödjer alla de möjligheter och krav som vårat system kommer med.

Acknowledgements

This thesis has been contributed to by Jose Lucea in addition to myself. Together, we have worked on some of the ideas discussed in this thesis, as well as the Marco prototype and feedback on ideas for our own thesis papers. His thesis, *A Comparison of Software Environments for Mobile Web Publishing Clients,* can be considered a companion piece describing the Marco prototype and architecture from the perspective of an author as opposed to a reader. I also thank him for the many distractions during office hours, delaying both our papers. Kudos!

The thesis was conducted with the help of Ericsson Research, using sites, equipment and intellectual property of Ericsson AB. I am very grateful for all the resources Ericsson has allowed me to make use of for this thesis, and in particular for the help and input of Göran Eriksson on all matters of the thesis, and the ERGO team for help on the engine that powers large parts of the prototype client.

Table of Contents

1 Introduction

Mobile phones, and similar networked devices, have made great strides towards being accepted for an information platform. Having progressed past the early stages of SMS, WAP and walled gardens by cellular operators, these handsets are poised to become a capable content delivery mechanism. This trend has only been underscored by the emergence of handsets without traditional input mechanisms; touchscreen devices such as the HTC Touch, Samsung Instinct or Apple iPhone have been proliferating headlines and market sales figures demonstrating peoples' desire for mobile media. However, while the phones' capability of displaying more and richer media has increased, there has been relatively little effort put into methods of finding new information by means of the mobile phone. We propose a system by which a mobile phone can automatically search for content to present to the user, utilising a combination of traditional recommendation methods and new inputs available on the mobile phone.

Traditional recommendation algorithms have been based on classification of objects into categories for similar items recommendations, examples of which can be seen on Amazon and similar online stores, and popularity based toplists such as practically any "top X" list available, e.g. Billboard Top 40. The former process is fairly difficult to implement outright in a system that aims to contain objects written by an uncoordinated mass of authors, as it requires the items ordered in a taxonomy to be able to determine what is related and what is not, while the latter is a fairly simplistic measure that has already been implemented and caters to fewer people than the former. Due to the nature of taxonomies, it is impractical to expect that a single coherent taxonomy could evolve out of it, nor is there any example of such. The closest available is a "folksonomy" – the application of labels – used to great effect by popular Web 2.0 sites such as del.icio.us and Digg. The usage of folksonomy in and of itself has been studied in detail elsewhere; the usage here is intended to be combined with data available by using a mobile phone's additional capabilities.

A mobile phone has more information available than a computer. GPS functionality is being included in certain phones, to allow users to locate themselves in the world, although even without GPS it is possible for a phone to use the cellular network to locate itself within 25 metres in many cases (Ahonen och Eskelinen 2003). With this information in hand, we suggest it should be possible to associate folksonomy entries with coordinate information, allowing the system to learn where users' interests lie, and use that data to present a user with information they either find interesting at the current location or relevant to their immediate surroundings.

1.1 Taxonomy and Folksonomy

There are two main schools of tagging widely used on the Internet. The first is called "taxonomy," and is derived from the usage of the word as "the science of classification" (Encyclopædia Britannica 2008, "taxonomy"). The other use is colloquially referred to as "folksonomy", and refers to the social use of tags as "user-generated metadata" in sites such as del.icio.us and Flickr (Mathes 2004).

Taxonomy could be considered representative of an expert author. Under taxonomy, objects are assigned labels by a limited number of editors, and intended for end-user consumption. Each object can then be accessed by the labels, and usually also has

links to the index of objects with the same labels. For example, all objects labelled as "politics" on the CNN.com website, by their editors, can be found by the URI http://topics.edition.cnn.com/topics/politics.

Items in a taxonomy are organised by a strict and pre-determined structure, with hierarchical relationships to other related concepts. Such a system requires very real effort on part of the editor, in order to ensure that any entry in the taxonomy upholds the relationships within, and is appropriately categorised with regards to parents and children. Having this information, on the other hand, allows the computer to infer that "terrier" is a subset of "dog", and potentially display other dog breeds as related keywords. The system can also differentiate between "turkey" the animal or country. Finally, by working within a specified language of possible keywords, redundant categories are not applied to any one object[1]. (Weinberger 2005, pp 9-11)

The main drawback of a taxonomy is the need for a user to conform to, and be familiar with, the vocabulary selected by a small group of editors. For an unfamiliar reader, navigating a taxonomy is cumbersome, and often leads him down the wrong path. For example, to find information about the Computer RPG *Baldur's Gate*, a reader would need to browse a strict taxonomy starting from "Recreation" as opposed to "Computers"[2]. A synonym-aware taxonomy, such as the Yahoo! Directory, is useful for search as it can recognise a user's search terms as an entry point in the taxonomy and provide related topics from the taxonomy. This eliminates the need to browse the taxonomy to reap benefits from it (Weinberger 2005, pp 8-9).

Folksonomy is defined as "social tagging"; that is, the process of any person being able to apply tags to any object, regardless of whether they are author, editor or just consumer. While an author in a folksonomy usually does apply tags to an object, in order for it to be discoverable by other users, the authors' tags are, in a broad folksonomy, no more or less important than the ones of the readers (Wal 2005).

The main differences introduced by folksonomy are that any person can tag any object, the number of tags that can be applied to the same object are not limited, and the vocabulary is not restricted. This allows each person to use their already existing cognitive process to quickly establish tags for an object (Sinha 2005). By allowing any person to easily apply tags to any given object, the set of descriptions of that object should, with sufficient number of people, approach the full set of possible descriptions. This, in turn, means people will be able to find it by searching for the term they associate with the object, as opposed to honing their Google keyword skills.

The unrestricted vocabulary, however, leads to tag divergence on apparent synonyms. Checking if a tag exists or not before posting an entry with that tag is a cumbersome task, and diverges from the folksonomy ideal, so each person contributes their own

---

[1] Redundant categories are still *searchable* and accessible through the use of synonyms, but while the colloquial term "flick" might refer to a movie, all movies could be tagged "movie" with "flick" a synonym of "movie".

[2] User-centric taxonomies have already deduced the need for synonyms, of course, and the Yahoo! Directory (dir.yahoo.com) entry for "Computers -> Games" leads to "Recreation -> Games -> Video Games", as an example of this. Nevertheless, a *strict* taxonomy would not classify a single object under multiple categories, also exemplifying one of the problems in general of taxonomies.

associations. For example, a list of "related tags" on http://del.icio.us/tag/apple[3] reveals both "macintosh" and "mac" as related to Apple. To a human, the two tags are effectively the same. Consequently, one would expect them to be related tags to each other; however, visiting the http://del.icio.us/tag/mac and http://del.icio.us/tag/macintosh[4] does not show them as related tags. This means that an audience browsing the "mac" tag is disjointed from the content tagged with "macintosh", unless that content has been tagged with both tags. Furthermore, since folksonomy generally treats tags irrespective of context, the chances of finding information about Granny Smith apples when browsing the "apple" tag is fairly low, due to the popularity of Apple Inc.'s products.

1.2 Folksonomy and Geodata

There has been much effort invested in attempting to discern the semantic meaning of tags through analysis of the tags themselves, by means of dictionary meaning, or by using graphs to find which tags are related. We propose that it should also be possible to find related tags by means of geographical location.

With the advent of more precise location capabilities for locating mobile phones, the concept of geotagging has gained traction. Geotagging is simply applying a "tag" consisting of the user's current geographic location to an object. Thus far, geotagging has had its greatest success in images[5], with some handsets now shipping with the built-in ability to apply geotags, albeit crippled[6]. To avoid confusion with the textual tags, geotags will be referred to as "geodata" henceforth.

Geodata has the potential to diminish some of the confusion of folksonomy discussed earlier, while at the same time opening up new avenues by which it is possible to find related tags. Folksonomy tags with multiple meanings in a global scale need not have multiple meanings on a local scale. To continue an earlier example, while one might not want to eat the country Turkey, the "turkey" tag as located in a North American town will most likely refer to the poultry. Furthermore, if the same tag has been used repeatedly in a geographically close area, it may refer to the same kind of use. It stands to reason, thus, that associating geodata with a tag, and the specific uses of that tag, has the potential to improve the relevance on tag search results.

1.3 Profiling and Geodata

User profiles, which gather information about a user's habits, interests and history, are much coveted by service providers for their ability to supply more relevant suggestions and search results to the user. By analysing what actions a user has taken in the past, profiles can deduce interests, as opposed to asking the user for them, and presenting results with similar properties are likely to pique a user's interests. Such techniques have been used to arguable success by a number of websites to produce results end-users consider relevant (Sinha och Swearingen, Comparing

---

[3] http://del.icio.us/tag/apple accessed on March 25, 2008. Related tags may be subject to change due to the volatile nature of social bookmarking.
[4] Both accessed on March 25, 2008
[5] It is worth noting that contemporary marketing would have people believe that geotagging is strictly referring to images, which it is not. Images are only one of the objects that can be geotagged.
[6] For example, the SonyEricsson C702 is capable of embedding geodata into an image, but only using a SonyEricsson-specific XML extension and, even then, specifies only mast positioning and not geographical coordinates.

Recommendations Made by Online Systems and Friends 2001). These systems are successful because the results highlight not that which a user has already seen but items they have not seen though are likely to enjoy.

Profiling systems suffer from a number of flaws. Among these flaws is the inability to distinguish between a user's different interests. While a user's interests can be recorded on a global basis, there is no guarantee that a user has the same interests at all times; rather, a user's interests may vary depending on the time of day, his whereabouts, current mood or other factors. Whereas it is not possible to automatically derive a user's mood, as of yet, the possibility to factor in geographical location is now available.

Geodata-aware profiles have the ability to separate a user's interest at various points. At the most basic level, this could be the "home" or "private" interests as opposed to the "not at home" or "public (places)" interests, which separate what a person is interested in privately with what they are willing to display when surrounded by others. A workplace profile might comprise of even fewer, mainly focusing on elements that are "safe for work" or relevant to the workplace.

Another benefit of geodata profiling is the ability to apply semantic categories to a user's location and activities. The above categories are fairly straightforward to define for most locales; e.g. "work" could be defined as the locale from 9am to 5pm (minus eventual lunch). Combinations could be created such as "abroad" being combined with the country of residence to make "tourist from Sweden". These categories could then further be used to automatically make more detailed profiles of users, categorise the posts they make and read, and share data from user profiles in the same categories or locations.

For the purposes of this paper, a geodata-aware profile will only concern itself with what items (tags, posts) a user is interested at a certain location, and not in determining semantic categories. The data sets will presumably arrange themselves in patterns that are similar to the categories outlined in the previous paragraph, albeit without any form of logic to make use of that information. This level of analysis should be sufficient to establish the merits of this approach when it comes to providing ranking and suggestions.

1.4 The Marco Prototype

"Marco" is a system rooted in the principles outlined above, and implemented in order to test the basic theories. Due to time constraints, tests on Marco will only be performed by a very limited amount of testers, and will thus most likely not represent a real-world test. Furthermore, this prototype will not concern itself with data or code efficiency, as these areas have been thoroughly examined otherwise. Finally, Marco itself will not contain or test all elements described in this document. These limitations of the prototype itself are focused mostly on reducing the difficulty and time required to implement it, and are documented in 7.1.

2 Related Work

Ever since the emergence of the "Web 2.0" phenomenon, much research has focused on how to further develop the phenomenon of user publishing. Services such as Twitter, Plazes and Jaiku[7] have allowed users to make small posts, so-called "microblogging", from their mobile phones in order to make short statements on where they are or what they are doing. Flickr and del.icio.us also have mobile versions to upload your photos and manage bookmarks respectively. In true Web 2.0 fashion, all of these services also allow users to categorise their entries with the help of folksonomies.

While much work has been done on making mobile publishing of Web 2.0 content, the area of searching and browsing content, especially geo-tagged such, has not enjoyed as much attention. There are a couple of main schools of thought on how to accomplish it. One is to literally transfer the desktop folksonomies to the mobile, and attempt to use tag disambiguation to present relevant information. Another approach involves printing two-dimensional barcodes ("Mobile Codes"), and their sibling RFID senders, while yet another school is to use positioning systems, e.g. GPS, Mast Positioning, and WLAN identification, in order to make location-aware search.

2.1 Folksonomy Tagging

Folksonomy tagging on the mobile contains the same elements as it does on the desktop. Services such as mobilicio.us, a mobile version of del.icio.us, allow the user to filter on tags by typing in a tag. More advanced services, such as Yahoo! ZoneTag[8], are capable of making tag suggestions as well, depending on a user's location. These tags, however, are a generic collection of all (Flickr) tags used at a location.

Marco should prove superior to such services based on its ability to consider context information when presenting the user with search opportunities. In addition, Marco aims to consider not only data of the user's location, but also his profile, and his contacts' profiles, in order to attempt to infer the user's intent at a location, as opposed to merely the location.

2.2 Mobile Codes

Mobile codes, or 2D barcodes, are images printed or otherwise associated with a physical object. These codes can then be scanned in by a compatible reader, usually housed in a mobile phone application, and will lead the user to a URI appropriate to the object scanned (Holmquist 2006). Some services also allow the encoding of arbitrary text in the code, instead of just offering a URI[9].

The immediate benefits of Mobile Codes are evident; the code scanned by users is almost certainly relevant to what users are currently interested in, so no guesswork is needed by the mobile phone. This gives quick access to information desired by the user, which is always a desired outcome. In addition, the steps to access the

---

[7] http://www.twitter.com, http://www.plazes.com/ and http://www.jaiku.com/ respectively

[8] Currently developed by Yahoo! Research, a preview of which is available from http://developer.yahoo.com/yrb/zonetag/index.html as of May 12, 2008

[9] One such service is http://qrcode.kaywa.com/ (accessed March 26, 2008), allowing the encoding of URLs, text, phone numbers and pre-packaged SMS messages into QR Codes, a type of 2D barcode.

information are always the same: take a picture of the encoded data in question and the link will be accessed. Finally, it is easily recognisable for the user where they can find metadata available to the phone.

The drawbacks are numerous, however. A multitude of barcodes clearly visible in the surroundings would not be aesthetically pleasing. Further, the fact that they are physical visible labels puts them at risk of vandalism, which could destroy or corrupt the data stored, rendering it unavailable for a reader. The physical property is also a drawback in the case of a URI being changed: all the physical tags need to be changed as well. There is little interaction with the tag itself too. Once published, a tag is immutable, giving it a publisher to reader interaction as opposed to a social community.

2.3 Location-Aware Tagging

Location-aware services are quickly gaining traction among researchers. The concept revolves around posting a virtual note at a certain position, which can then be retrieved by being on the same position, or by browsing a map displaying notes based on their geographic location. López-de-Ipiña, et al., describe this kind of service in-depth, and also propose the Sentient Grafitti system, which will be discussed henceforth as the representative of current location-aware systems.

The strength of location-aware systems is their ability to highlight entries to a user based on criteria other than entering keywords. As discussed previously, a person is much more apt to recognise the concept desired based on a selection of choice, as opposed to entering the correct keyword needed. Anecdotally, this could be compared to knowing the name of a particular neighborhood of a particular town, as opposed to picking out one's general area on a map; the latter should be an easier task for a user who is not too familiar with his location.

The focus on Location-Aware tagging is, however, still closely tied to the concept of an object's location being of paramount importance to the user, thus ranking tagged objects primarily on proximity. Marco aims to take this a step further, by leveraging the location and other contextual information to determine a user's likely interests at a point, both in nearby and slightly more distant objects, and to rank all of these objects by methods more than simple date and time.

3 Considerations and Limitations in Mobile Clients

This paper concerns itself with how to efficiently search and browse text or image entries on a mobile phone. As such, it should take into account a number of major elements required to make a functional application on a handheld device. There is no attempt made to completely solve these issues; rather, they are acknowledged here as a necessity when working with a handheld device, and have been allowed to influence the architecture of the Marco prototype system.

When considering the subsequent limitations, a few assumptions have been made to focus on creating a distributed architecture, as opposed to mainly working around handheld device limitations. First off, the device is assumed to have full network coverage for data transfer, and such data transfer never fails. Next, the device is considered to have so much RAM that memory conservation is not a concern. Finally, there is no risk for data loss[10] at any time, so data protection and session saving measures are not necessary. Even with these assumptions, however, a number of concessions have to be made for the basic difference between handheld devices and computers.

3.1 Limited Input Capabilities

By its very nature, a handheld device does not have ample input opportunities. At best, one hand has to be used to hold the device and, at worst, both are necessary. This limits input to at most six fingers and can be as low as one or two thumbs, depending on the device. These limitations manifest in slower or more difficult typing, as using nine buttons for typing is inherently slower than separate keys, and more cumbersome selection of screen widgets, owing to the lack of mouse and touchscreen on most phones.

There have, of course, been attempts to avert these limitations. The popular T9 input system allows much faster input of text, although this limits input to words in a pre-defined dictionary. Touchscreens have also been gaining traction, with devices such as the iPhone even eschewing other buttons in favour of the screen. However, textual input on touchscreens also has presented several flaws, most notably a lack of button ridges and finger-based input means a user can easily inadvertently input the wrong character.

Despite the drawbacks of touchscreens, a prudent assumption is that phones will in greater numbers support such means of input, which lends itself very well for interface with maps or other displays of widgets in geometrical relation to each other. The text input limitations, on the other hand, appear to be remaining.

3.2 Network Delay and Power Usage

A wireless network behaves with different characteristics than a wired network. One of the main differences is the round trip time of packets. While the average rtt, for local services, will hover below 200ms, there is significant variance upward 500ms or

[10] "Data loss" could be from an application crashing, or, more in line with the assumption, the device itself shutting down the application to free up RAM, or running out of battery.

even 1 second (Chan and Ramachandrian 2005). This unpredictable change makes it unsuitable to repeatedly querying a server; such an action could create sporadic delays that, in turn, would cause needless and jarring delays on the client side, especially for display of data. On the other hand, Chan notes that it is possible to saturate the bandwidth of a wireless link by modifying the TCP parameters. This means it is more feasible, in terms of performance, to download a large data set as opposed to requesting it in pieces.

Another major concern with network access on the mobile phone is the power drawn by the circuitry responsible for the sending or receiving of data, especially on a UMTS network. When connected to a network, the phone can either passively listen on the network for traffic information, or power up its radio to send and receive data. The former requires little power, but provides little functionality, whereas the latter requires a medium to high power consumption, although it is now possible to send and receive data[11]. The high-power mode will, after a certain short interval, power down to a more energy-efficient mode. This behaviour implies that sending or receiving a coherent burst of data is more efficient, with regards to power consumption, than several short, sporadic transmissions with a lower overall bandwidth requirement.

Both of these limitations support the same conclusion: data should be delivered to the mobile in a coherent burst, then lie dormant for as long as possible. This behaviour will provide the best possible battery life, at least as far as network interactions go. More data sent to the device means a higher computational requirement is placed upon it, which, if taken too far, could cancel or even exceed the power saved from the data transfer.

3.3 Programming Support for Rich Media Hardware

Although not a theoretical challenge but a technical one, mobile phone firmware is not sufficiently advanced yet to expose its functions in practical ways. Most notably, for this application, it is impossible (or very difficult) to, on a standard handset, access camera or GPS functionality from a JavaScript runtime. Access to such functionality is limited to JME on many mobile phones[12], wherein it is not possible to create the very popular mashup functionality of Web 2.0.

3.4 Computational Limitations

Another highly important limitation is that of the mobile phone's ability to perform advanced computations. While the phones are fairly programmable, they lack very many of the data structures and conveniences afforded desktop programming. JME, for example, discards concepts such as sorting of arrays, Base64 encoding of data to transfer it safely over the network. Furthermore, JME applications on the mobile phone are also limited by the amount of classes they are allowed to contain and execute. JavaScript, on the other hand, is severely limited by the computational power

---

[11] This is, unsurprisingly, a slight paraphrase from a document by Nokia, describing best practices when designing "always on" applications. See Nokia Corporation, "Recommendations for Reducing Power Consumption of Always-on Applications," *Forum Nokia*, 26 Sep 2007, http://www.forum.nokia.com/info/sw.nokia.com/id/a6d789aa-9321-444e-a3c5-27cc7faa9ccb/Recommendations_for_Reducing_Power_Consumption.html (accessed Aug 08, 2008).

[12] Although even this is not possible on some Symbian-based phones, oddly enough.

of the handheld devices, a drawback especially noticeable when performing computations in JavaScript. As such, it is much preferred to use JME to perform calculations, with toll-free bridging to the JavaScript execution engine.

4 The Marco Client-Server Cooperative Architecture

With the above points in mind, we have settled on a client-server architecture for Marco, which makes use of the server's computational capabilities and creating mash-ups on the mobile phone. The overall distribution of tasks is visible in Figure 1.

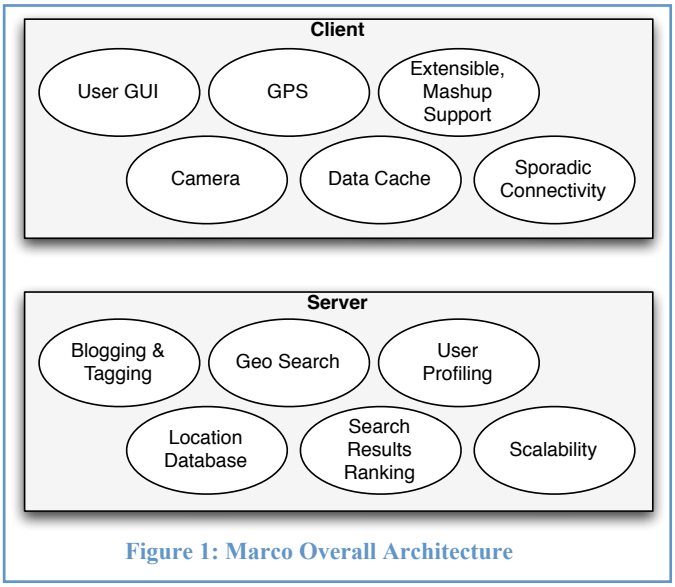

Figure 1: Marco Overall Architecture

The primary task for the mobile client is to provide the interface for the user to interact with the system. This interface should be able to display both Marco-specific information, as well as mashups with different sources, most notably a map overlay using Google Maps. The concept of using and facilitating mashups has been emphasized for its growing necessity by both López-de-Ipiña and Mathes, and we agree with the its usage. Marco is thus designed to facilitate it. The client needs furthermore to contain logic for the caching of data, to offset radio power costs of sending and receiving, to access the mobile phone's hardware, and to delay the sending of data for as long as possible.

The server, in contrast, must concern itself with data analysis and storage. All the intensive searching – narrowing down of location, tag isolation, date cutoffs – must be done on the server. It is also responsible for creating a profile of the user, based on profiling logs it receives from the client. This profile is then used to rank entries on the server and order them before they are sent to the client. Finally, in order to ensure that the Marco prototype refers to a feasible system, the server may not limit itself to only one concurrent instance, but should be a scalable entity.

4.1 Mobile Client

There are three possible avenues of deployment on a mobile client. The first is to create an application compiled for the mobile phone in its native bytecode, and using an SDK or language specific to the mobile phone model(s) targeted. It is also possible to create a JME application, which is able to run on multiple phone models in the same archive. Finally, it is becoming possible to write client applications in Ajax even for mobile phones. Out of these three methods, we opted for the Ajax-based solution, with a twist to allow us to access the phone's hardware.

The most compatible way to develop software, in regards to access to the device's hardware, is to code in the native API. This gives the programmer full access to the model's hardware functionality, at the expense of portability. In this project, the aim is to create a prototype based on several different models of accessing data. This will almost certainly necessitate the usage of several different phone models, with porting between the two being undesirable. Writing a native mobile client is therefore inappropriate.

JME applications attempt to implement Java's "write once – run anywhere" motto on the mobile phones. They introduce a trade-off of performance and features in order to allow an application to run on more phone models without having to be specifically coded and tested for each of those models. The limits imposed are by the usage, and support on the phone side, of the JSR additional APIs. Some phones may not give full or any access to hardware from JME, while the same hardware is accessible from a native application. Nevertheless, JME represents the broadest possible software compatibility with the minimum possible effort.

Ajax applications are written in XHTML and JavaScript, to be interpreted by a web browser or similar. Like JME, this makes Ajax applications platform agnostic, although they may still function differently depending on which web browser is used to interpret them. With the advent of widgets, Ajax has gained popularity for client-side applications, with Nokia announcing S60 support for widgets. Ajax was also for a long time the only method to develop third party applications for the Apple iPhone. The main drawback of Ajax-based applications is their inability to access phone hardware such as camera, GPS, microphone or other sensors. A client application would thus be forced to completely ignore these innovative, and, in the context of this paper, interesting, data sources. Alternatively, an Ajax client would need to make use of platform- or vendor-specific extensions to JavaScript, which allow access to hardware functions such as GPS location and the phone camera. Such solutions are available from, or are under development by, Microsoft, Opera, Vodafone and Aplix (W3C 2007).

For the Marco prototype, we have chosen to use a hybrid Ajax and Hosted Objects solution, as portrayed in Figure 2. This combines the benefits of Ajax mashups with the hardware support of JME, meaning that the client itself is a mix of the GUI logic and calculations made in JavaScript, and the back-end support provided by the JME Hosted Objects. The JME objects themselves are prepared and treated like an API to the JavaScript runtime.

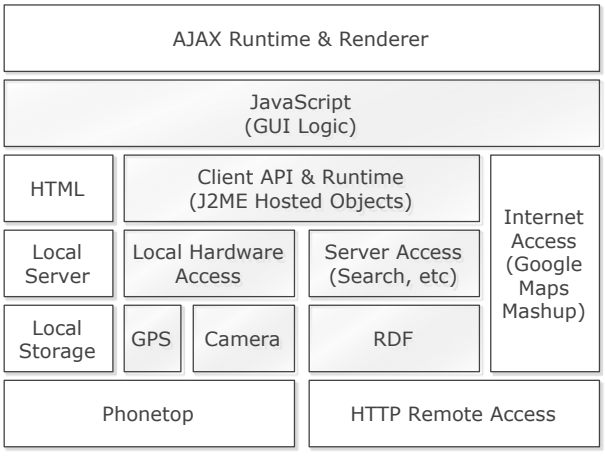

Figure 2: Mobile Client Internal Architecture

4.1.1 AJAX GUI

The Ajax GUI layer is intended to perform as few calculations as possible, to offset the cost of performing calculations in JavaScript. There are some calculations it cannot avoid, however. In order to not need a server processing and sending

unnecessary data to the client (the HTML to render each page), the HTML source code, including JavaScript source, are static documents stored on the mobile phone's memory. When accessed, the Ajax calls the JME API, requests the appropriate data that should be filled in on the page, and generates the HTML code required to fully display the page. The pages are thus generated in full by the Ajax runtime.

The other main feature that befalls the GUI layer is the mashup with Google Maps. This mashup is done directly in JavaScript, using only regular post data as exposed to JavaScript by the JME API. It requests an appropriate image from Google and calculates where to render post images on the Google map.

4.1.2 JME API

The JME API provides several services vital for the functionality of the AJAX GUI as well as system support. It provides the JavaScript code with object persistence between page loads, gives it access to native hardware devices such as Camera and GPS, and acts as a bridge between the JavaScript code and the server, providing caching, pre-caching and profiling opportunities.

All JavaScript objects are cleared between page loads. On the mobile client, faced with the prospect of composing one monolithic page with the ability to perform all possible tasks, or multiple smaller pages, each with a specific task, the answer was dictated by performance. To compensate, some other element must give JavaScript persistence between page loads. The Marco API allows JavaScript to "pass an argument[13]" to itself between page loads, which provides the necessary persistence.

The major reason to have a JME API is access to hardware otherwise inaccessible from JavaScript. Marco gives access to the phone's filesystem, for already taken photos, and the phone's camera, to take new photos and upload them to a blog directly. This functionality is desirable for use from JavaScript, as it permits the GUI to use those features without switching from mashup mode with Google Maps.

4.2 Protocol

The interaction between a client and a server takes place by means of a RESTful interface, using data mainly encoded in RDF format. (Fielding 2002) presents REST as a method to issue commands to, and retrieve data from, a server. REST mandates a stateless connection form, using the HTTP protocol, which makes it compatible with existing infrastructure such as proxies, load balancers, and other devices designed to enhance performance for HTTP accesses. Since its proposal, REST has been gaining traction among web services, including Yahoo![14], Amazon[15] and other large companies. The usage of REST in the context of Marco is for the prototype to inherit the properties of stateless connections and replicable servers, adding restrictions on how both the client and the server may be designed.

REST is, by definition, a method to transfer data. It serves an envelope for the data, but makes no claim on how that data is represented. With the exception of media data, such as images, the Marco prototype makes use of the Resource Description

---

[13] Although only Java Hosted Objects themselves, or primitive types, can be passed as arguments.
[14] All of Yahoo!'s web services are REST services, as described in http://developer.yahoo.com/search/rest.html
[15] Amazon AWS is sometimes used as a type-example of how to design a REST web service

Framework format (RDF) to transfer data. RDF is a W3C Recommendation from February 2004[16]. The main benefit of RDF is its ability to use descriptive "predicates" to label data. This allows a properly written RDF reader to recognise the *title* predicate defined in *http://purl.org/dc/elements/1.1/* and deduce that the data attached represents the title of the document, even if it is unable to recognise the document definition itself.

The usage of RDF is mainly aimed at forcing the design of the prototype and system so that it is possible to create mashups on top of it. Mashups are a phenomenon wherein one web service retrieves data from two or more separate web services, combines them and presents the combination, providing a superior presentation than was possible if the services were separate. (Lathem 2007) describes the concept of mashups further, and illustrates how services annotating their data with RDF provide a framework to create mashups, and mashup creation services.[17]

4.3 Server

The server design follows logically from the requirements based on client and an open protocol. By using REST/RDF, the server must effectively act as a stateless data repository[18]. Internally, the server must act as a coordinator for many different clients and aggregate the data input in useful form. The server is also responsible for collecting and calculating the profile data used for its search ranking, as well as conducting actual data searches. Both profiling and searching are described more in-depth in section 5 below.

For the server implementation, we have chosen to implement a Java 1.6 servlet running on top of the Glassfish server, using the Jersey REST-style library. These choices are quite logically following from both client and protocol; a Java servlet can easily act as resources, especially with a REST library. Furthermore, using the same language on both client and server significantly decreases the amount of work needed on data structures and protocol, which is a non-negligible factor when prototyping. The server is backed by a MySQL database server, for data storage and retrieval.

The server itself has three main tasks. First, it needs to be the simple storage and retrieval mechanism for entries. Second, it must take these store and retrieve messages, along with other data supplied by the client, in order to build a profile for users, posts and tags. Finally, it must be able to conduct comprehensive searches on the data stored, taking into account the user profiles. All of these operations are fairly unchanged between different languages, at least as far as any significant advantage that would offset the loss of code-compatibility between client and server. Thus, JEE/Glassfish became the chosen platform for the server.

---

[16] The date is used for the most-recent RDF Recommendation, as taken from the W3C's Semantic Web Activity page (http://www.w3.org/2001/sw/) as of May 13, 2008.

[17] While the Marco prototype is designed to be accessible for the purposes of mashups, this paper does not attempt to further the field within the area of mashups. Rather, they are included in recognition of their emerging force, and that any design for community-oriented services should keep that in mind.

[18] Using a quite liberal definition of "data repository". While much of the server's actions are calculated results, each such calculated result can be considered stored data, from the perspective of the client (and intermediate proxies), at least insofar as the HTTP GET operation is concerned.

5 Profiling Entities and their Geographical Data

The concept of posting notes at certain locations, and then being able to retrieve them by realising that the user's terminal is at or near that location, was thoroughly discussed by López-de-Ipiña et al. However, their *Sentient Grafitti*, much like Flickr, and other services that take into account location, limit themselves to stating that there are specific types of elements on that location, be it notes or images or tags. We propose a system wherein each of those types of elements, and any other form of object that may be attributed a location, considered an equal element, henceforth called "entity".

5.1 Defining Entities

An entity is a generic descriptor for any uniquely identifiable object. This allows objects on the Internet, as well as physical object, to be referenced and assigned properties. Each entity must have a single, unique URI within the system, as this is used to identify when two different objects reference the same entity. Consequently, each unique URI, even when referencing the same object, is considered a separate entity. Each entity also has a unique author attached, considered by the system as the origin of the entity.

Entities may have one or more properties attached, and each of those properties may in turn be applied several times with different values. For the purposes of this work, only one property is defined: the "location" property. Furthermore, the location property may only be applied once for each entity.

5.1.1 The "location" Property

A location, when applied as a property of an entity, is defined as a combination of both longitude and latitude, but also the time that particular location property was applied, and the author – the user responsible for applying the location property. This defines a location as a point in space and time, as opposed to just space. Each location should also include a type annotation, determining what type of location it is referring to. These types are dependent on later interpretation, therefore can be defined separately. For the purposes of data manipulation, "location" may have further properties attached, such as source of the positioning data, or the estimated accuracy of the geographical location.

5.1.2 Rationale for Entities

Entities are the logical representation of an arbitrary object, allowing us to assign it properties without regard for where it is stored, what it represents or even having access to the object itself. This abstract representation makes it possible to profile, search and rank both tags and posts using the same algorithms.

5.2 Entity Relationships and Tag Entities

Individual entities, without any connections to other entities, are of no use when trying to create a system capable of mapping interest. For the system to be able to draw any conclusions, all entities in the system need to be connected to each other by some form of relationship that can be evaluated. Entities already have two of these – author and location – but an exploratory network requires more.

Entity relationships are defined as a connection between two entities comprising of the type of connection and a numerical value denoting that connection's weight. This results in a closeness graph between the entities. We only required one form of connection, which represented the relation between an entity and another entity that is considered its "tag", and each such connection is given an absolute value of one. This means all tags are defined as absolute relationships, and is done due to the complexity of working with more complex entity relationships during the allotted time.

While the above design implies that any entity can be considered a tag of another entity, in anticipation of unorthodox tags such as images or sounds (e.g. a photo or spoken word), the definition of tag used in the system refers exclusively to a tag object, containing a string representation of the tag and nothing else.

5.3 Entity Profiles

A comprehensive profiling system pervades the entire architecture of Marco. This system records a user's interaction with various entities, as well as tracking the location of the user during the course of those actions, creating a full profile over what a user has done and where. The profile can then be used to power all the functions necessary to the search facility; by leveraging profile information the search engine can extrapolate a user's desires, assuming a sufficiently developed profile. In order to do that, though, the profiling must be very thorough in identifying what the user wants and, crucially, where.

There are four types of events tracked by the profiling engine are "expose", "view", "update location" and "post". Of those four, two are generated on the server and the other two are done on the client.

Expose is triggered when the user can potentially view an entity. Thus, when downloading a list of ten entities, even though only five are displayed on the mobile phone, all ten entities will be considered exposed, since, by scrolling, the user exposes the other five. The expose event is triggered on

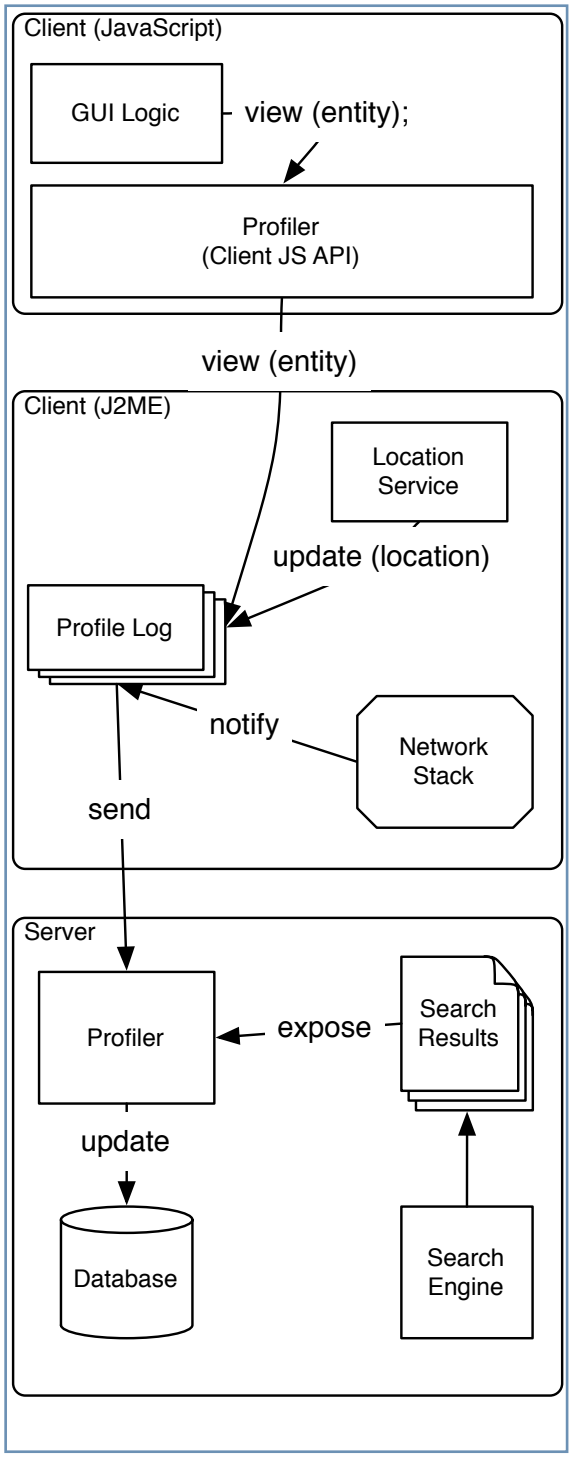

Figure 3: Profile Events Flow

the server before it sends a collection of entities to the client. The server emits the Post event when the client successfully makes the procedure of creating a new entry, and the server has accepted said entry.

A view event occurs when the user chooses to display the full contents of an entity, e.g. the contents of a blog entry or a non-thumbnail image. The client GUI generates this event when an item is actually displayed, an action necessitated by the precaching in 6.3[19]. Views are considered a fairly important metric, as they indicate actual user interest in a particular object.

The Update Location event is emitted by the JME client internals whenever a location is detected that differs from the last known location of the user. This occurs either at user request to give an accurate location or a passive trigger that does the same. Either way, the most recent user location is added to the event log, and used for further updates to the profile data on the server.

The actual profiling part takes place solely on the server. Thus, profile events that occur on the server are immediately profiled and parsed into the database. Events on the mobile, however, are gathered in an event log. This event log is not sent to the server unless some other network traffic occurs. Such behaviour is necessary to avoid draining battery life, as sending profile data immediately would result in much more frequent data traffic that, in turn, leads to the radio being in the active state for longer, as discussed in 3.2. Once sent to the server, the event log is processed, incrementing the appropriate counters on both entity and user profiles as described below.

It should be noted that profile events are recursive. That is, if the event "read" has occurred on a given post, it is to be treated as a read not only for the post itself but also for the post's author(s) and each of the tags applied to the post. Each of those is profiled separately on the server, although sending all that data from the client would use too much bandwidth. The semantics are defined to follow that if a post is read that author's read count goes up, and likewise for tags, making this assumption valid.

Profiles are tracked on two levels. One level of profiles the data of an entity on a global scale. Every user's actions update such profiles for any affected entity, allowing these profiles to approximate the "world's" opinion of an entity. The other level is one profile created for each user for each entity. This tracks how a user is concerned with any given entity (author, tag, blog, post, etc), and allows for a much more personalised identification and ranking.

5.3.1 Profile Data and Entity Value

The entity profile itself simply tracks the numerical values for the number of exposures, views and times someone has referenced it – that is, has made a post that includes the URI of said entity. For each expose and view event these counters are incremented, and each post event on an entity that references a specific entity that entity's reference counter is incremented. These counters are then used to calculate a numerical value.

---

[19] It should be noted that this event is emitted in the JavaScript code due to the convenience. In a deployment scenario, this design would be fairly insecure and require either the trigger to be placed elsewhere, or checks to block abuse. Security of the system has been of no concern for this thesis.

An entity's value is the relation between exposures (e) and references (r) plus views (v) as in

$$Value(Entity) = \frac{e + c}{ar + bv + c} \qquad \text{where} \begin{cases} a,b \rightarrow \text{relative value} \\ \text{c} \rightarrow \text{dampening constant} \end{cases}$$

The value for an entity thus becomes larger the more exposures it receives. This is interpreted as the system showing the entity to users but them not showing any interest in it. If users show interest for an entity, by either viewing it or referencing it, the entity's value is decreased. Since referencing an entity requires more user effort than merely reading it, the two counters are valued differently. To stabilise the entity's value from large fluctuations during the first few exposures and views, the dampening constant *c* is added to the calculation. The value of c depends on the size of the community and how fast a change such as an entity being unpopular should be reflected in the system.

The entity's value is only intended to convey a relative value, which signifies a user's like or dislike for an entity. The values themselves should be tuned so that a value of 1 is fairly neutral to the subject matter, with values approaching 0 identifying a clear interest, and values approaching ∞ indicating disinterest. These values are necessary to facilitate the subsequent calculation on part of complex entities, e.g. tagged entities. An object with many tags should not be unduly penalised compared to an object with a single (or no) tags.

Complex entities can have sub-entities[20], which changes the value of the entity itself. What constitutes a sub-entity differs depending on what complex entity is being considered. For example, a blog post has the sub-entities of author and all the tags the post is tagged with. The product of the sub-entities' values is multiplied with the entity's value to produce the total value of an entity.

$$Value(Entity) = \frac{e + c}{ar + bv + c} \times \prod Value(SubEntity)$$

This value must, and does, hold true to the size conventions mentioned above, as it may itself be referenced by another entity. The entity's value is computed each time it's required by the ranking algorithm, which, given the number of recursion levels it may have, seems a more prudent option than updating a database all the time. However, this also means it cannot be an index value, and that a different such is needed.

5.3.2 Relevance Time

Since an entity's value is a proportional value representing user interest versus system exposure, another value is needed to index the relative popularity of entities. Such an index is necessary for the search algorithm of entities. Relevance time attempts to solve this issue by making a statement about how stale an entity is. An object is considered to become stale in Marco when it is no longer read by anyone, or

---

[20] A related entity as per 5.2 does not imply it being a sub-entity, or vice-versa, although this may be the case. They are two independent concepts. For example, an author is not a related entity of a post, but is a sub-entity.

referenced by anyone. This means, for example, that Shakespeare's *Romeo and Juliet* would be considered "young" by the system because it is continually read and referenced; a measure of popularity.

Relevance Time (*RT*) is updated as a part of entity profiling. The first time an entity is inserted into the system it is assigned a RT equal to the current UNIX timestamp (*CT*). The "view" and "reference" actions as present in the entity profile also affect the relevance time of an entity. Each of those actions performs the transformation $RT_n = (CT - RT_{n-1}) \times k + RT_{n-1}$ where $0 < k < 1$. The value of k differs depending on whether the action is "view" or "reference", with $k_{view} < k_{reference}$. The values of k determine how quickly the system picks up on the renewed popularity of an entity, or how many actions are required to keep an entity from becoming stale. A low k will require many interactions before an entity is considered relevant, while a high k ensures that stale entities are those who are not viewed at all.

Entities are indexed on their Relevance Time value, which is used as a main index for the geographic search. It is therefore important that Relevance Time accurately represents the relative popularity of entities. This also forces new entities to have a high relevance time. If they do not, they are simply never chosen for display, and thus cannot increment their relevance time. Relevance Time is also used for the ranking algorithm, as a weight to the profile values.

5.4 User Profiles

In order to fully utilise the geodata available on a mobile terminal, generic user profiles are insufficient. Given a sufficient number of users, generic profiles will end up identifying all interests everywhere. As a single counter-example, there could be a person living in Uppsala and working in Stockholm, and a person from Stockholm working in Uppsala. A generic profile would identify commuters to both cities, and identify them both as workplace and home, allowing none of the advantages with geographic interest zones. Instead, each user has their own profile that tracks their interests at different geographic location.

5.4.1 User Entity Profiles

A user profile is composed of so-called User-Entity Profiles. These profiles represent a user's disposition towards an entity. This could identify items like favourite posts, images, authors or tags. User Entity Profiles also detail where this interest manifests itself. This could be a very limited area, such as one's own home for very private matters, or essentially anywhere for items such as general news.

UEPs contain both the same data as in a generic Entity Profile – that is exposures, views and references, although these refer to how many times it has been exposed, viewed or referenced by the particular user and not all users – but also include two locations: view location and post location. These locations represent the approximate locations where the interests are centred, with respect to when the user prefers to view and post (or reference) the entity.

Each time a UEP is updated with a *view* or *post* event, a location should be provided describing where the event took place. Said location should, assuming a device that

updates its location with reasonable intervals, correspond to the Update Location event preceding the *view* or *post* event[21]. This information is used to improve the accuracy of the two locations included in the UEP. When an event with a location is recorded, the UEP locations are told to "edge" towards that location. It stands to reason that, although a user will likely not perform the same action in the exact same location every time, the actions should converge on a general area for interests that are associated with that area.

The UEP location starts out in the same location as each event is recorded on for the first time. Once there, it moves towards other locations by means of simulated travelling. When it is asked to edge towards a given location, which is a new place that a user has taken that action, the location accelerates towards it, at a rate of $1/10^{th}$ of the distance between the current location and the new location. If the acceleration vector is more than 35° off the path of the current travel vector, the location's velocity is reduced to zero, and the location begins accelerating anew in the new direction. This slows down the location's travel between wildly different points, as well as prevents the location from overshooting the target.

Ideally, the UEP location should either convey an interest density map of how much interest a user has shown for a subject on all possible points in the world. This is a fairly complex notion, however, so we instead work on the assumption that each interest can be pinned down to a specific point or, at least within one degree longitude and latitude of a specific point.

Calculating the value for the UEP is done similarly to the Entity Profile calculation, including sub-entities (if applicable). The main difference comes from the addition of another factor in the calculations: the distance factor. The distance factor starts at 2/3 for being right on top of the entity and grows as the distance from the view/postloc increases. The effect of this is a boost in disinterest the farther away a user is from the UEP location in question. The distance factor used is the *view* location for a regular search (i.e. reading)[22]. If there is no location on which to base d, it is assumed to be 1. With the default numbers for the equation, this leads to a factor of 1, not modifying the UEP's value.

$$Value(UEP) = \frac{e+c}{ar+bv+c} \times \prod Value(SubUEP) \times \frac{|d|+t}{kt} \qquad \begin{cases} d \rightarrow \text{Distance to view/postloc} \\ k \rightarrow 1.5 \text{ (default)} \\ t \rightarrow 2 \text{ (default)} \end{cases}$$

The UEP provides the main guidance on a user's interest or non-interest in an entity, not because of the entity itself but its sub-entities. For most entities, the initial weight value of references and views will be fairly insignificant, as a user will likely read it once or twice, maybe reference it once and be done with it. The sub-entities, such as

---

[21] Logically, in the user's event log. The event log always begins with Update Location, so a location will always precede a *view* event from the client. While it would be semantically equivalent to simply use the user's current location, this method allows for batch processing of events without necessarily tagging each event with a location.

[22] The *post* location, although tracked by the profiling, is not used by the prototype. The *post* location is intended to be used to search for autocomplete suggestions when composing a post. An automated search for what a user posts about (other posts, authors, tags, etc) at a given location would provide entities useful for an auto-complete feature, although this is a tangent for the thesis.

tags and author, however, will have far more significant values determining a user's interest in the top-level entity being considered. This is what allows the system to make a judgement call on whether or not a user is interested or not in a particular entity that he has not seen so far.

6 Geo-Profile Search Facility

The concepts of "browsing" and "searching" have been used fairly interchangeably so far. This is because while the goal of the paper has been to establish a method that allows the user to browse entries in the system, browsing is merely a search using different search queries. Whereas a search in Google would start with the user typing in a keyword representing their current interest, a search using Marco starts with the user showing an interest for searching, and the system performing a default search. The user can then conduct further searches.

A default search simply means a search with none of the limits otherwise applicable. These limits are things such as the entities have to be tagged with certain tags, come from a certain author, be within a specified geographic area (e.g. "must be close" or "must be in the USA") or contain a certain word. These limits are fairly trivial requirements, insofar as they can be expressed with a simple SQL query and thus make for non-interesting discussion material. The subsequent discussion will assume no such limits, although they are elementarily applied in the first and most basic step of fetching data from the database.

Searching a pool of entities sorted in a geographic space is a three-dimensional search. There are the two physical dimensions, longitude and latitude, but also a third dimension in the form of "relevance". In a traditional search engine, "relevance" could be tailored to a static constant based on the evaluation of an algorithm, such as Google PageRank, and choose the selection on the basis of a keyword. In Marco, one of the aims has been to facilitate a search based on no manual user inputs, which is facilitated by a two-step process. First a selection of entries is made based on simple, generic parameters. This selection is then refined in the second step by taking into account a user profile.

As discussed earlier, recommendation algorithms are successful because of their ability to provide the user with both what they already like, but also to highlight slightly different ideas or concepts. This exploration is very important to the concept of Marco; however, this also means that each user will have different value for each object in a ranked database. Indexing and updating every entry in the database for every user in the database, and their changing and evolving interests, is simply not a feasible approach. Instead, for each search, a select number of entities are chosen, as described in 6.1. These entities, called "candidates" form the pool of objects that will be ranked.

Ranking is the process that takes into account a user's profile data, such as tag interests, author interests and location interests, and uses this data to assign a relative numeric value to the entities, detailed in 6.2. Once ranked, the candidates are further reduced to the number of entities to be sent to the client, and the search is considered complete. Before being sent off to the client, all entities' profiles are updated, as per 5.3.

6.1 Selecting a pool of candidates

Accurately selecting the pool of candidates is a prerequisite to the ranking algorithm being useful. Even if we assume that the ranking algorithm is perfect in every way possible, it must work on a selection of entities, as opposed to the entire set thereof. This means that it is possible for an article, which would have a very high rank in this specific user's profile, to not be present in the candidate pool fetched, and thus cannot be ranked. Increasing the size of the pool decreases the chance of this occurring, with the major drawback of requiring more processing time on the server.

Marco makes use of a pool selection algorithm that tries to balance popular articles with user location. The underlying assumption is that a user's interests are most likely to be reflected geographically; that is, the interests are either close by or far away, that ranking will reveal which is more interesting and, consequently, neither category must overshadow the other. The weakness of this approach is that, regardless on geographical scope, unpopular articles are quickly removed from the field of view. This is desirable in the general case, to filter away uninteresting articles, although it skews the rankings, as items with a niche interest will likely have a lower popularity than other articles. This is due to the way entities are indexed.

Global indexing is based on the location parameters, longitude and latitude, as well as a derived value of popularity dubbed "Relevance Time". This value, detailed in 5.3.2, is used to calculate how "old" an entry can be considered. These three values can then be searched in the database without using a particularly complex calculation.

The pool selection itself uses a growing window. It starts with a small window based around the user's current location and attempts to fetch up to 10 entities[23]. These entities are the one with the highest value of Relevance Time. This window is then expanded by a factor of two, and another 10 entities are fetched. The process continues until the maximum window factor is reached (±180° long, ±90° lat) and the set of entities collected during the process becomes the pool of candidates.

The candidate pool itself is a set of objects, ensuring each entity can only be present once. This is important, as an entity with the most recent Relevance Time can be present in several of the search windows, and each search window is independent of the rest. An entity should not be represented several times in the pool of candidates; such a vote of relative popularity is already expressed by means of Relevance Time, and expressing it twice is not necessary. The search windows are independent of each other out of coding and computational ease. If one was interested in making a map-based view, with fairly consistent coverage of the map, the search windows would need to exclude entities found previously in order to fill up the map better; however, this is not the desired outcome in this case.

The collection method on the pool guarantees that entities from the immediate location of the user will always be candidates for the ranking algorithm, allowing exploration of the nearest surroundings. There is no such guarantee, however, for any of the window's expansions. This means that, assuming the 10 most popular articles worldwide are located within the first window, the entire candidate pool will consist

[23] The exact number is, like the other numbers in this section are, subject to tweaking depending on whether one wants a greater pool for the ranking algorithm or faster performance on part of the search.

of 10 articles. As a corollary, it follows that the 10 most popular entities worldwide will always be present in the candidate pool.

6.2 Ranking the candidates

Once an appropriate pool has been selected, the ranking algorithm has to choose which of the potentially hundreds of entries are of interest to the user. Due to the selection process, the more limited amount of entities in the candidate pool can be analysed with greater scrutiny. Ranking occurs based on the entity's profile, the user's profile and, in the case of entities that reference other entities, also consider the referenced entities' profile values when calculating the rank.

6.2.1 Generic Entity Rank

To create a ranking of a set of entities, each entity is assigned a numeric value. This value represents how disinteresting an entity from the set is. The least possible disinterest value is *-0* and the highest possible is -∞. Once all the entries in the set are ranked by disinterest, it follows that the least disinteresting ones – that is, the ones with values closest to zero – are the ones most probably interesting.

Disinterest was chosen for the ranking due to the exploratory notion in recommendations, as well as the inability to apply an algorithm based on interest suggestions on an uncategorised data set. Suggestion algorithms, such as the one used by Pandora, depend on assigning each object in the data set a distinct value in a number of different categories. These values can then be used to derive similarity between two objects, thus recommending an object similar to a user's interests (Glaser, et al. 2005). A user-provided data set lacks this ability, however, as there is not a distinctive authority able to rate all entities in the data set, nor any series of defined criteria to evaluate them by. Tags applied to objects can help out in this respect, but provide a much slower rate of exploration, as interest has to be shown in neighbouring areas even though there might be interest in both, just not actively displayed to the system.

Ranking by disinterest assumes that there are fewer topics that a user might be uninterested in than potentially interested in. Consequently, those topics a user does not pay any attention to are ranked much lower; leaving entities a user might potentially be interested in the top rankings. Disinterest is interpreted as not viewing or referencing a certain topic. Thus a user is considered interested in topics s/he dis*likes* if those topics are posted about often.

The definition of general disinterest is the product of two factors. The first factor is the proportion of exposures of an entity to the amount of views and references of that entity. This ratio distinguishes between entities with approximately similar relevance time, such as entities that are very popular or entries added fairly recently.

The second factor is Relative Age, which expresses staleness. The Relevance Age is the difference $RT - CT$ [24], which grows more negative if an entity is never viewed and decreases if it's viewed a lot, depending on the values of k in 5.3.2. Relevance Age is what ensures that users will view new entries in the system. When a new entry is inserted, Relevance Age is exactly zero, a value it will essentially never be able to

[24] Relevance Time minus Current Time

achieve anew. This ranks new entries fairly highly to allow them the ability to build up an exposure to view/reference ratio representative of the interest of that entity.

The two factors are combined to create a rating for an entity as per

$$ER_g = Value(Entity) \times -\sqrt{CT - RT}$$

The difference $RT - CT$ would grow too large if allowed to be a pure factor in the Entity Rating. To dampen the growth, especially for old entities, the square root of the difference is taken, causing a growth that, while initially fairly rapid, slows down as the entries age, preventing entities from being too severely penalised for age. The decreased rate of growth also reduces the size of the calculated numbers, which could be very high otherwise and risk numerical overflow. To avoid imaginary numbers, the root of the positive difference $CT - RT$ is taken, and then made negative[25]. This prevents the resulting Entity Rating from spiralling into a gigantic number and causing computational errors.

It should be noted that, both for this above current time, and the reference of it below, the current time is a static value that is recorded once at the beginning of the evaluation. It stands to reason that the current time will change during the course of the computation; however, if different CT values were used during the computation that would unduly influence some entities ratings, especially during the later part of the computation. Thus, the CT value is the one of right before the computation.

This rating can then be used to sort entities in a global basis, without profiling data on a specific user. It gives a general measure on how relevant or stale an entity is, in a manner not unlike Digg. This value is useful in attempting to make a rank for users that have no profile data at all, as it still gives some form of ranking. It is also quite useful as it processes the relevance time value, which ensures the disappearance of stale entries.

6.2.2 Profiled Entity Rank

The key part of the entity ranking is taking into account both the generic popularity profile of an entity and a user's individual profiling data. The global profiling data and the user's specific entity profile data, in their respective recursive forms, are combined using a weighted sum as follows to form the entity rank.

$$ER = \left(m \times Value(Entity) + (1-m) \times Value(UEP)\right) \times -\sqrt{CT - RT}$$

In this equation, *m* is the weighted value that specifies how important the global profile should be versus the user's specific one. Optimally, *m* would decrease as a user's profile grows to become more complete, signifying that the system knows enough about the user to not have to judge him based on the average user. However, a static value, although sub-optimal, should easily suffice to make the profiling work. Thus, *m* was defined as 0.45; this number places slightly more weight on the User Entity Profile specific to the user, as opposed to the generic entity profile. As a result, the user's tag, author and blog preferences have slightly more precedence than the generic popularity description.

---

[25] Since $RT - CT = -(CT - RT)$

Each entity in the search is assigned an Entity Rank, a value only relevant for the time of the computation. All the entities are then sorted according to descending Entity Rank, to determine which are most probably interesting for the user. A selection of the highest ranked ones can then be sent to the client, completing the process of finding which entities to send.

6.3 Pre-caching & Mobile Processing

When an appropriate pool has been selected on the server, the mobile phone downloads the data. The data included from the server includes the entities, their geographical location, the entity ratings as calculated by the server, as well as all the prerequisite details, such as author, tags, and entity body, except for image entities. Since the only entities supported are tags, posts and images, they will be referred to as such henceforth.

The mobile client will cache all the text data available from the above list, as this data can be fairly quickly transferred to the client in one large burst. Furthermore, this information compresses well, so a compressed transfer would be fast enough to not be noticeable, while conveying more information. Said information can be used to present the search results in more ways than one without having to consult the server. Such behaviour is highly desirable, as it may take up to several seconds from a user action until the client receives the new data from the server, caused by both network delay and the potential need for the phone to power up network circuits. When the data can be pre-cached, the operation runs locally and thus much faster.

Images, and other media, however, do not compress well and take fairly long time to transfer. In addition, the data transfer may incur further costs for the end user, something that is clearly not intended. For the time being, this means that image pre-caching is not a viable option, and thus the delay in this case is inevitable. However, by pre-caching all metadata about the image instead, which is textual, only one request is required when loading an entry containing an image.

While pre-caching works fairly effectively to cut down on loading times within the search, it does not help if the user needs to perform more searches, such as if they have read some or all of the entities found in the search results. Pre-caching can be used to combat this behaviour as well, by downloading more search results than can be displayed in the client. Not only is this useful for presenting multiple “pages” of search results, but it can also remove entries from the first results page. Once the user has viewed an entry from the search results, it can be removed from the first page – or prominent view on the map – and make room to show a new entry there instead, without having to conduct a new search. Note that it is not necessary to pre-cache data for map actions, as these will likely require network access, and thus power up the network circuits regardless.

It is possible to download too much data via pre-caching. It is doubtful that the exact amount of data that would constitute “too much” can be specified, although an approximate definition would be when the pre-caching affects the responsiveness of the search itself by a significant amount. Given that the server already processes many more entries than are sent to the client, the additional processing requirement on the server is not significant, but two other factors appear: data transfer time and processing time on the client. The data transfer itself, even on a GPRS network, is not very long for plain text, especially if it is compressed. However, compressed data

requires processing time on the client to extract, and all the data needs to be parsed, and this processing can require more time than the data transfer itself.

In order to properly manage the data from the server, it needs to be unflattened, so that all the appropriate data structures are created. This entire process needs to be completed for all data before even a part of it can be displayed to the user. Arguably, threading could account for displaying partial data while further processing is going on, but doing so only leads to unnecessary complications on how to deal with thread and data synchronisation between the JME and JavaScript runtimes, a prospect that can fairly easily be done without.

7 Marco Prototype Implementation and Results

The goal for implementing the Marco prototype was to create a test bed on which the theoretical algorithms could be tested and proven working but, more importantly, the performance of such a system to be evaluated. The proposed system works over three different platforms; two on the mobile phone and one on the server. It uses a fairly verbose, text-based protocol to talk between the two. Finally, a fairly elaborate and unproven algorithm runs on the server, attempting to provide "relevant" search results.

The last goal is fairly difficult to verify, given the limited scope of both time and resources allotted to the thesis work. Verifying what is relevant would require a set of test subjects to use the system for a period of several weeks, and then answer a questionnaire about it, and this would be required for just minimal testing. However, it is possible and desirable to ensure that the system is able to deliver fast responses to the client, confirm that the ranking algorithms can correctly pick out items with user interest by small-scale, artificial, testing and ensure that none of the design choices create a system unsuitable for long-term development of the system.

7.1 Limitations of the Prototype Implementation

In this paper, a lot of ground has been covered on different possibilities of how the algorithms should work, how the server should handle them, what should be sent to the client, and what the client should be doing once that information arrives. Fully implementing all of these mechanics has never been a goal, and thus the "prototype" designation. However, even within the prototype, there's a fair amount of simplifications, limitations or even omissions of the concepts outlined above. Most of these have been done by means of assumptions that simplify the possible input values, although some omissions have been necessitated due to time concerns. This section will detail what, exactly, is simplified for the prototype, and what is included or not.

In general, the prototype has been designed to err on the side of caution as opposed to performance. Many blocks of code utilise Java's *synchronized* attribute to prevent them from being accessed by two threads, lest they leave the database in a corrupted state. While this happens a lot on the server, the most glaring part of this is the client platform, which, for example, cannot execute JavaScript while displaying the results on-screen[26]. This is generally not a concern, however, since the prototype is not about

[26] This matter is not helped any by the platform used by the client itself (see Appendix B: for an evaluation of the platform used)

making a seamless graphical user interface, nor has there been a significant effort in creating any such.

7.1.1 Location Management and Positioning

The prototype includes no form of accurate geographical positioning. While both the client and the server have fully functional data structures for location management, the location services – actual providers for geo-positioned data – do not connect to any device or input capable of determining a location of the mobile phone. Instead, the mobile phone's location service makes a request to the server for the mobile phone's location, a make-believe implementation of asking the mobile operator for the network location. The server returns either a random location somewhere on the planet, or a static location that has been manually input.

While geographic location of the mobile phone is a key concept, the physical location of our device for development and testing does not need to correlate with the value the system is testing. Furthermore, this functionality actually eases testing as it allows evaluation at several different physical locations without the need to actually move there. There are several drawbacks of this setup, which impact the platform.

First off, these tests will always result in absolute locations, making no statement on positioning errors and how these would affect both reading and writing of entries. Given the complexities involved, there was not enough time to even speculate on this, let alone devise a method to cope with errors, and then test them. Another side effect is that the time it takes to make a location lookup is effectively static, apart from network delay. This can be fairly simply simulated, or might be ignored entirely, merely stating that we know of its existence, or lack thereof. Finally, and perhaps the most important issue, is that this adds more network traffic, powering the radio for a longer time. This cost could be considered roughly equivalent to the GPS' power requirements, and is thus ignored.

7.1.2 Ranking Algorithm

While the ranking algorithm is fairly complete, there is a fairly significant part of it missing: author ratings, and personalised author ratings. This omission is strictly due to the time allotted not sufficing to make a fairly complete analysis and implementation of the author ratings. Instead, adding an additional tag to each element suffices: the "postedby*author*" tag. The effect is roughly similar, in that another sub-entity is added to the profiling stage, although the statistics and weights may not be entirely suitable for authors. Effectively, the author ratings' profiling has been eliminated entirely; reducing the amount of code required for the profiling and profile calculations parts.

As is befitting a prototype, the ranking algorithm isn't optimised either, on quite a large scale. While the algorithm correctly caches entity values for sub-entities shared between several entities, it makes no effort to pre-load them, even though it knows which entities will be used. Furthermore, the easiest place to place the ranking algorithm is in the sorting algorithm of the search results. This is out of a design necessity, as opposed to a time saving measure, as doing this allows modular replacement or adaptation of the sorting algorithm as necessary.

7.1.3 Energy Saving and Mobile Processing

During the course of the system's design, a recurring theme has been to try to not use excessive network resources, conserve processing time on the client and otherwise mind our usage of battery power on a mobile phone. Given the time limitations, some elements of these theories have not been implemented, primarily the user-driven ones.

The delayed profile sending *has* been implemented in full; the network notifications have not. When profile events are gathered in the client, they are stored in a temporary buffer that awaits a notification from the client's network layer that traffic is going on. The "client" in this case refers to our JME client libraries, and not the platform itself. A result of this is that it is possible to have network traffic when loading an image, which is handled by the browser following an `<img>` tag, without sending the user's profile data up to that point. Instead, the profile is sent when either a location is requested from the server, or a new search query is made.

7.2 Evaluation Procedure

Having created a functional prototype for the system, testing it is a very important step in ensuring it works correctly. The main goal of this project, to provide more relevant search results, is fairly difficult to test given the limitations on both time and resources within a thesis work. To ascertain whether a person would find the results more relevant would require a solid group of people to use the system for an extended period of time, creating a full profile of them. This, in turn, would require a non-trivial amount of content in the system, content which is to be assumed user-provided in the intended scenario, but would have to be generated for a test scenario. With this in mind, making a comprehensive evaluation on whether or not the Marco prototype provides more relevant results would become the work for another six months. Instead, the evaluation will be a "sanity check" to ensure that the proposed methods are actually feasible to use.

The feasibility testing is aimed at showing that the algorithm is not unusable. It stands to reason that, even if the search results were the most relevant ones there could possibly be, if the time taken to generate the results is prohibitive that means the system is unusable. Likewise, if the scaling of the system were inefficient, even if future hardware reduces the execution time, the need to search more user-generated data would offset the gains. Inversely, should the system be able to provide sufficiently fast responses as it is, and the scaling acceptable, then, while it does not prove success, it shows the procedure has a chance.

The first and foremost thing to consider when conducting the testing is that the prototype implementation has, at no point, actually been developed with performance testing in mind, as the concepts of rapid development and excellent performance rarely go hand in hand. It is safe to say that the performance exhibited by the prototype should be abysmal at places; however, it should be possible to identify how sub-optimal the prototype is compared to best practices. Furthermore, the key parts in the system, that is the search algorithm and delivery method, should be fairly indicative of their real-world performance. For the rest, the testing method chosen should suffice to highlight how much time is required to perform the different searches.

The system evaluation itself consists of two parts: server and client. The server side evaluation aims to identify the applicability of the proposed search and ranking

algorithm; that is, if the search algorithm's performance is acceptable on the server side, which quite corresponds to the above. The client side evaluation will match whether or not the platform, chosen for its affinity towards mashups on the desktop, is conductive for mashups on the client as well. Again, performance will determine whether or not that is the case.

7.3 Server Evaluation

The key element in the server, and that which has defined its testing, is the search algorithm. While the server contains additional functionality for posts, blogs, multimedia and users, the non-trivial elements that concern this thesis pertain to making searches.

- **Total Time:** The total amount of time from a request to a finished response.
- **Queries:** The amount of queries sent to the database. An exorbitant amount of queries implies the code is inefficient.
- **Database Time:** The amount of time spent in the database. This is useful to try and estimate how much time could be saved by better queries.
- **Entities:** The number of entities loaded. An entity can be loaded several times, by different parts of the code, to ensure data correctness.
- **Posts:** The number of posts loaded from the database. This number implies how many posts have been found by the pool selection algorithm, as each of those posts is fully loaded before being added to the pool set[27].
- **PostsTime:** The amount of time spent loading posts. This time can be effectively subtracted from the execution time, as posts should not be loaded until the search algorithm is finished.
- **Locations:** The number of locations loaded as part of the search. This is included as yet another measure of the inefficiency in the prototype, as there's a fair amount of redundant locations being loaded. At the most basic level, there are only a few hundred unique locations in the system.
- **Profiles:** How many profiles have been loaded to make the ranking. This is the minimum amount of profiles, which is approximately three times the number of unique posts being ranked.
- **Pool:** The time taken to make the selection pool.
- **Ranking:** The time taken to rank the objects in the pool.

Table 1: Total Search Time

PPI	Total Time
10	422ms
20	735ms
30	1078ms
40	1422ms
50	1734ms
60	2093ms
80	2797ms
100	3578ms
200	6938ms
500	18812ms

The tests were made by an automatic widened search, centred on longitude 20, latitude 50, with an unlimited pool size[28]. In order to test the algorithm's performance, searches were made by increasing the number of posts

[27] A quite needless operation that occurs due to how the searches work. The entries are always fully instantiated by the database model, as opposed to returning objects with only stub values. Again, this is to ensure *data* correctness, to allow a faster prototyping.

[28] Mainly to not have to raise the pool size for entities per iteration. Obviously, loading more entities than there's room in the pool isn't a good idea.

per iteration of the selection algorithm. Due to there only being one viewer, the entities should be considered in the same order for each search, as Relevance Times are not modified for merely searching. This implies that the PPI number will effectively represent the number of posts in the ranking pool. Each subsequent search thus comprised of more posts, and took longer to process. The test data comprised of 737 posts with geodata attached, located in the Kista suburb of Stockholm[29]. Several searches were conducted using varying amounts of entities per iteration, with the results recorded in Appendix A: Table 4, although an excerpt is included to the right. PPI stands for Posts Per Iteration.

Unsurprisingly, the time it takes to provide search results exceeds the time a user would be willing to wait. Searching with 100 PPI and ranking 100 posts would take a total time of over 3.5 seconds, a noticeable delay for even the most patient user. Some of this can be attributed to the prototype being inefficient, and removed as such. For example, the ranking algorithm's execution time was improved by a factor of 6 for these tests by caching redundant computations. While there will be no further attempt to optimise the ranking for this testing, we can infer that the post loading time in Table 4, which is caused by the database engine, can safely be removed from the search time, since contents of a post are not relevant for the pool selection or ranking. By doing this and summing the values, we arrive at the "compensated" execution time, graphed below and detailed in Appendix A: Table 5.

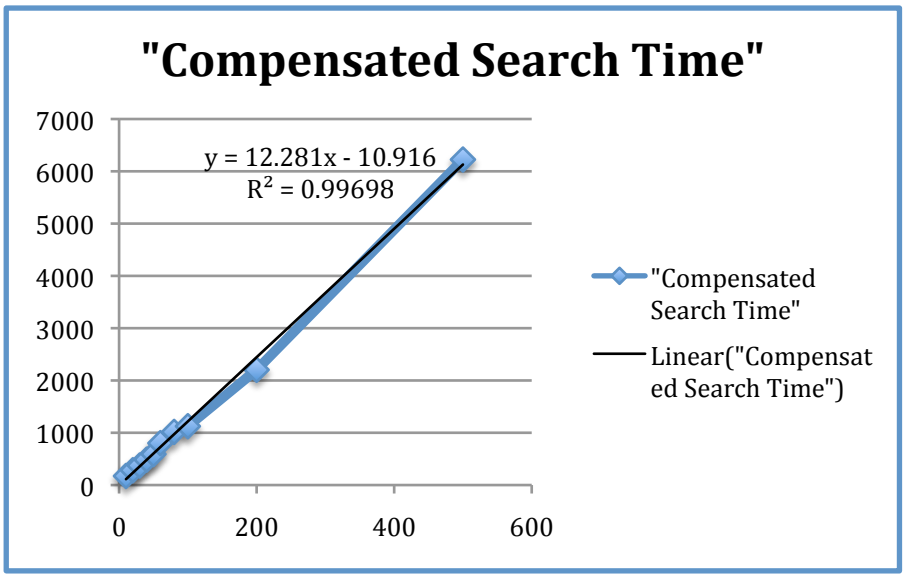

Figure 4: Search time, compensated for inefficient programming

The compensated search time, while greatly reduced from the initial search time, still exceeds one second for 100 PPI search. This situation is mitigated by knowing that both the pool selection and profile loading algorithms are inefficient, although determining how much more efficient they could be is outside the scope of this paper. Furthermore, the absolute value of the time taken depends in great part on what hardware it is run on, and testing on a server-class computer with a database-class SCSI hard-drive would definitely provide different results.

[29] For details on the test data, see Appendix B: Section 1.

Nevertheless, it is possible to infer from the graph that the search algorithm does not take excessive time to produce results. More precisely, as the number of posts increases, the time it takes to process them does not appear to increase exponentially for this test data set. From the fact that the ranking algorithm applies a sorting algorithm, it stands to reason that the sorting algorithm would imply an $O(n \log n)$ execution time, although it is likely it has a lower bound of n.

7.4 Client Evaluation

Similarly to the server, the goal of the client-side execution was to provide the search results in a speedy manner. This is hamstrung by the desire to make the client platform as open as possible, and conductive for Web 2.0-style mashups. Such a setup necessitated the use of a client environment that was more loosely based than could be done otherwise. Among other things, the client itself has elements in HTML and JavaScript, with both having to be interpreted when run, and the protocol is not only text-based, as opposed to binary, but also fairly verbose. This has negative consequences when it comes to client-side performance.

For the performance testing, three core areas were targeted, on two different platforms. The testing was conducted using SonyEricsson's SDK emulator, configured to emulate a JP8 phone with no performance restrictions. This would provide an environment computing faster than the mobile phone but, more importantly, having essentially no network delay. In addition, the server's processing time is effectively constant as long as the PPI is not increased – serving 10 results from the same pool takes about as much time as serving 100, as the inefficient loading has already loaded all the data. The mobile phone tested was a SonyEricsson K850i, over Telia's UMTS network. The testing on the phone was aimed at identifying how large an effect the data transfer over UMTS had, as well as what performance the parsing, JavaScript APIs and JavaScript processing had on a handset. The results are shown in full in Table 6 and Table 7. The values tracked are as follows:

- **NumPosts:** The number of posts sent from the server to the client. It is assumed that this number does not influence the time it takes the server to provide a response.
- **Total Time:** The total time it takes from a user input action to the search results being shown.
- **Network Time:** The time spent on the network. This time includes the server's search time.
- **µJena Time:** µJena takes a certain time to parse the rdf file into its internal data structures.
- **Object Parse Time:** The time it takes to create Java data objects, by fetching the data from µJena
- **Client Library Time:** Total amount of time spent in the J2ME client code

With that in mind, the results from the emulator paint a fairly grim picture. The total execution time, even for 10 posts, is abysmal and, as the amount of processing increases, skyrockets to unacceptable amounts even for the emulator.

Table 2: Selected Emulator Metrics

NumPosts	Total Time	Network Time	object parse time
10	4469	3640	438

20	6016	3828	1392
40	12001	3906	6001
80	34048	4516	24282

The network time is fairly interesting, as it implies that either there is some fairly heavy duty processing going on in the mobile phone's network layer or that the emulator itself imposes some form of transfer limitation. Since this feature is optional and disabled during testing, we can only assume that there is quite some inefficiency in receiving data on the mobile phone. The other interesting element, and quite alarming, is the parsing time for objects, and how it scales. From an unassuming, albeit large, 438 milliseconds for 10 objects, this balloons to 24 seconds for 80 objects.

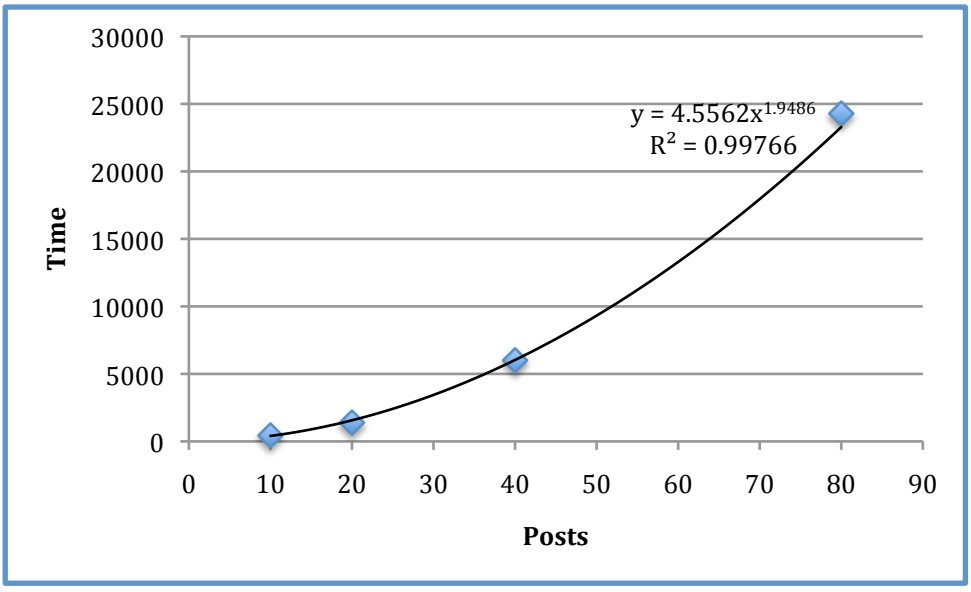

Figure 5: Object Parsing Execution Time

From the data points given, this part of the process scales around the order of $O(n^2)$, which is clearly unacceptable for any application. Even assuming the transfer of a fairly low number of posts, this performance would unnecessarily delay the parsing and, more importantly, prevent the pre-caching of data, or requiring roundabout solutions for doing such. Perhaps more importantly, when placed on an actual mobile handset, the performance becomes intolerable.

Table 3: Selected Handset Performance Metrics

NumPosts	Total Time	Network Time	object parse time
10	10942	4447	1645
20	14607	4642	4614
40	32423	5114	17713
80	108146	6761	78891

As can be noted here, the performance on the actual handset spirals out of any reasonable proportion. Parse time for even ten posts balloons to the order of seconds, and the total execution time for 80 posts takes almost two minutes. Clearly, for any sort of performance on the handset, the prototype client is woefully inadequate, especially while using the µJena library.

The other interesting aspect to compare between the mobile client and the emulator client is the time spent by both on the network. We know the response time for the server when searching through 100PPI is ~3580 millseconds, with a small variance. This is effectively true as long as the number of posts requested is less than 100 because, as noted above, all posts are already loaded into the server's memory. The encoding time itself (including profiling) ranges from 130 to 170ms for 10 to 80 posts respectively, so given the numbers shown on the mobile is negligible. Additionally, we can also assume that data transfer time over the 100mbps network to the emulator is effectively constant. With that in mind, the adjusted graph below subtracts 3580 from the network time to estimate the actual time for network between server and client, including the mobile terminal's network stack.

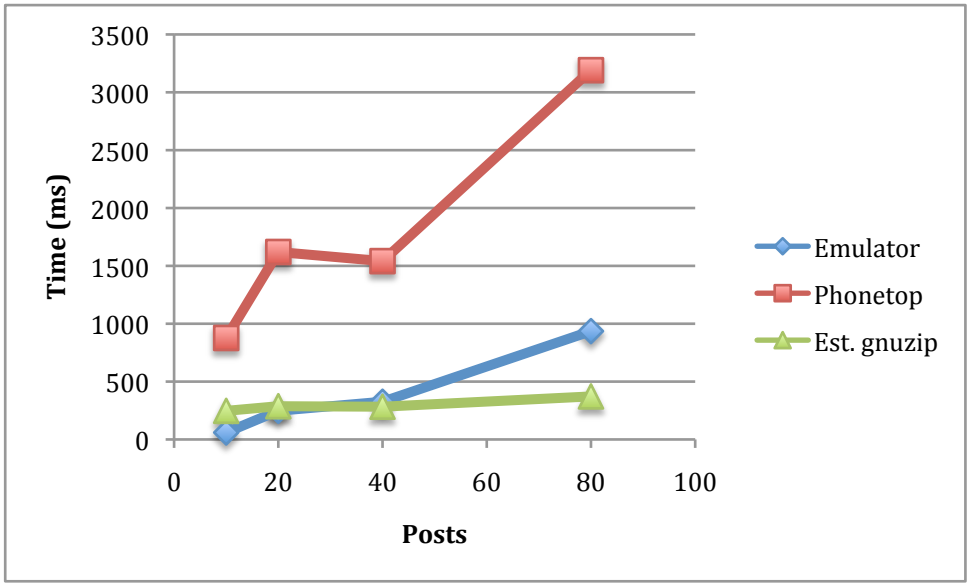

Figure 6: Time spent on network transfer and stack

As this graph shows almost painfully obvious, precaching data, at least before rendering the initial screen, is a very large time sink on uncompressed data. The data could be compressed to significantly reduce the network transfer time, as the protocol data has been compressed almost twentyfold: *gnuzip* compression of 159807 bytes of protocol data – the equivalent of 80 posts – results in 8610 bytes of compressed data. This would reduce the transfer time to less than one second for most, if not all, scenarios transferring text-only data. Such a scenario is estimated above as "Est. gnuzip" by reducing the phonetop estimate with the factor between uncompressed and compressed, and further adding 200ms as the average rtt established for mobile networking in 3.2. This should establish that precaching of compressed data is a viable option, although this would most likely increase the already prohibitive processing time.

The time to create the user representation, including JavaScript execution time, have not been considered. In any event, the presentation is of little to no relevance as we have not made any attempt to improve either the performance or quality of the actual interface when compared to the already available offerings and have, instead, focused on what data is presented. Even so, any benchmarks would be even more flawed as the client runtime itself was a prototype, causing us to run a prototype implementation within another prototype implementation.

8 Discussion and Conclusions

During the evaluation and testing, we saw that the main complications arise from the system design itself, the fairly complicated algorithm on the server side, and the processing limitations on the client side. In this section, we'll examine the effects of them on the prototype.

8.1 System Design

Overall, the system design seems to support the time data gathered above. Processing on the mobile phone seems fairly expensive business, and keeping as much as possible of it away from the phone is the correct approach. Unfortunately, the RDF parsing would appear to be computationally more expensive than expected; this is discussed in SECTION below.

A side effect of the power-saving design for the mobile phone is the impact it has on the profiling. Due to the profiling data on the mobile phone (see 5.3) being sent only after there has been other network activity, a request for a new search can be dispatched before the profile events are sent to the server[30]. The search itself is the network activity that triggers the profile data to arrive, thus the profile data is unavailable for the search. This exact order means that a user can make a search without the current profile data being taken into account. Such an occasion should generally not be noticeable, as, on a mature user profile, most of the entities will have profile values that are not affected much by single variations. It would show an impact on new users not receiving personalised results quick enough, or a "second search" showing more relevant results than the first. The situation can be mitigated somewhat by being able to trigger on other network activity, such as the activity from the web runtime loading an image, another page or even network activity of a separate process; these mechanics does not seem to be available for a JME application.

Another possible remedy would be some level of client-side processing of the search results. One such would be to apply some modification to the rankings, based on new profiling data from the client. A second could be to change the ranking order of items if the current user position changes slightly, thus avoiding an entire request to the server. Both of these would significantly increase complexity on the client, however, and probably require some or all of the profiling data on the server. Given the already limited client-side resources, moving profiling onto it does not appear to be a feasible solution. Rather, the discrepancy between profiling data being generated and it being taken into account is too expensive to avoid.[31]

Client development was switched fairly early on in the process from a complete JME client to the current iteration. While this transition did not sacrifice any of the already completed JME code, it did create some additional effort to make the client function. As was mentioned earlier, the state of the tools we had available to build an AJAX GUI backed by Java objects hosted in JavaScript had limitations of their own. This mainly concerned the GUI work, however, so the rudimentary level of the tools

---

[30] This never happens in the prototype, due to the location service always making a network location request beforehand, thus triggering sending of profiling data. This is a behavioural error on part of the client, as opposed to intended functionality. The intended functionality of the client should be able to produce the described behaviour, when using local GPS functionality to make a request.

[31] In fact, in an optimized and/or deployed system, it could be desirable to not update profile data when received but instead make a batch update once an hour or during lower times of load.

themselves has not necessarily impacted the work in a negative manner, as the GUI was never a priority to begin with.

By using JEE and JME, we shared a great deal of code between the server and client. This was a great boon for the speedy implementation of the system, but likely had severe performance impacts on several levels. The client implementation is in many places essentially the server implementation adapted for JME. As a consequence, some parts of the client, especially the data structures, are effectively designed for a desktop. In addition, many of the features in JEE could not be used in JME; instead of reproducing the functionality on the mobile end the affected API calls were replaced with *functionally* equivalent calls. As one example, effectively all of the *LinkedList* structures from JEE were translated to *Vector*s on the JME client. While they could be used in the same manner, modifying a *Vector* has different time consumption than a *LinkedList*.

8.2 Server & Search Results

As has already been established, there is insufficient time and resources to evaluate whether or not the relevance of the results has been improved. There are not many conclusions that can be drawn from that, making the focus of the server on its efficiency, or lack thereof. As was seen earlier, the server takes a fair amount of time to generate results. Ideally, this time should be as small as possible, although on the server this is as much as over 3 seconds even for 100 PPI. This is a prototype issue however; the search itself, if implemented in an optimised manner, would be feasible.

The first thing to note about the current search procedure is the time it takes to amass a pool of posts: 3 seconds at 100PPI for a pool of 100 posts. Much of this time is spent loading and discarding the same data, namely loading the post data for posts already found in the pool. The weakness lies in that entities are searched in a window that increases its size, which in turn can load entities already loaded from the previous window. Had the pool selection algorithm been implemented differently, such as having a low resolution “grid” to load distant posts from, and only needing to make queries on the local end, not only would the amount of queries needed to create the pool go down, but the wide area low resolution queries could already be cached. Such a setup should dramatically reduce the pool selection times from the ones displayed in the prototype while providing approximately equal functionality.

The second part of the search results, the ranking, has a much better performance than the pool gathering. Initially expected to take more time, the ranking of 100 posts only takes 547ms, or approximately 1ms per post. There is not much optimisation possible here, with the exception of consolidating the profile data. At the moment, each of these 100 posts requires the fetching of 2 profiles from the database, each of which is sent as an individual query, resulting in 200 queries into the database. If nothing else, having loaded all the profiles into memory beforehand should improve the time taken for this stage. This improvement could not be estimated, as the database queries for profiles were not benchmarked, since the benchmarking code here could very well have added a not insignificant amount to the execution time.

Before the scaling is discussed, it should be noted that the test data used was very favourable. There was only one level of related entities – that is, tags were not tagged – leading to only one level of recursion. If the tags themselves were tagged with other tags, as a suggested element of the ranking entails, this would increase the time it

takes for the ranking stage. Along this track, it is also possible to achieve an infinite recursion loop. If a tag “mac” is tagged with “macintosh” and “macintosh” itself tagged with “mac”, the profile for the first tag will look up the profile for the second, which looks up the first and so on. Not only is there no cycle breaking logic included in the ranking, but the situation where a tag depends on a closed cycle to evaluate its own value is not defined. The former flaw can have some fairly stern consequences for the performance of the ranking, while the latter will affect to some degree the relevance of the results, making its consequences harder to gauge.

The scaling of the server itself seems to be good enough, even considering friendly test data. With the entire operation being fairly linear, that makes it probable that the highest cost involved is the $O(n \log n)$ time of the sorting algorithm. Consequently, it stands to reason that the server could be feasible with today’s technology pending optimisation or, failing that, future hardware capable of the initially higher cost of processing.

8.3 Usage of RDF

RDF was chosen as the protocol for the system due to its ability to connect two separate data sources, and infer similarities. These properties, as we already established, make it highly suitable for mashups, which we wanted to facilitate. It has had quite a severe impact on both size of data transfer and the speed of processing, however.

The main issue with using RDF for the data format is its size. In the implementation we used, almost 2 kilobytes of data were needed for each post, even though each post only contained 20-50 bytes of text, 1-3 tags, one image, author and geographic location data. While this could potentially be tolerable in a desktop scenario, it is unacceptably efficient when transferring data to a mobile handset. Fortunately, the vast majority of the data in the RDF stream is duplicated due to the notation style chosen. The RDF standard defines several different notation styles for the data, and we had to choose the most verbose one, called N-triple, in order to be able to use the JME RDF parsing library we had available. Had there been support for it, using N3 or RDF/XML notation would have reduced the bandwidth usage, at the expense of additional processing time.

The CPU time required for accessing data with RDF was also an issue. On the server-side, the relatively fast CPU allowed it to encode the data in 100-200ms. It should be noted, however, that this included encoding all data, every time. It stands to reason that static data, such as a post’s contents, or a tag’s definition, could be pre-encoded RDF, included in the main stream by reference. This property would allow for a reduction in the RDF processing time on the server, useful especially in context of the prospect of generating more complex RDF/XML rather than N-triple. In any event, the encoding time on the server is not an issue, and could likely be optimised even more if it were.

The client decoding time, however, is another issue entirely. As stated above, RDF parsing on the client and, especially, accessing the data within takes a prohibitively long time. This is simply unacceptable, but does not necessarily reflect a weakness in the format as much as it could be the library. While the library’s code was not analysed in detail, the most likely scenario is that it does not create a graph of the

RDF file, for quick data access, but instead performs a search for each data request. This would explain the n^2 execution time for the data access.

In the end, the usage of RDF has, with the current implementation, not assisted our particular system. It would have been more effective to use a shorter text-based protocol, or even a binary format, from a resource usage point-of-view. That would, however, not have been facilitating mashup usage of the data in the system.

9 Future Work

We have presented merely the beginnings of a system for the organisation of geo-located posts. There are still a fair number of avenues to explore down this path, given both time and resources. Apart from the straightforward testing and reviewing whether or not the basic premise improves search results, there are improvements that could be done on the profiling data and the ranking itself.

Profiling data currently holds one interest location per user, but a more useful measure would be an interest "map", which could be likened to an elevation map of the world. This data profile could track a user's relative interest for a tag in any part of the world, and make use of that to present results. Another aspect of profiling that would be useful when travelling is profile type classification. If the system could group similar user profiles and draw on their collective data for places where the current user's profile is inadequate, it could significantly reduce the learning time for a user's preferences.

The ranking has some severe flaws making it unsuitable for deployment. The biggest is its inability to correctly handle loops in the related objects. Consider a tag "apple" tagged with "fruit", and the inverse. This would either lead to infinite recursion or one of the recursions would need to be discarded. An efficient way of identifying this situation and assigning a reasonably correct resulting value for the two tags would go towards making the ranking feasible to implement.

The additional work done, in our opinion, highlights the magnitude of the task to create this system, which we thought would be simpler going into it. The end result, we believe, is nevertheless a step in the right direction towards location, and hopefully intention, aware devices.

10 Works Cited

1. Ahonen, Suvi, and Pekka Eskelinen. "Mobile Terminal Location for UMTS." *Aerospace and Electric Systems Magazine, IEEE* 18, no. 2 (02 2003): 23-27.

2. AUGUST P., MICHAUD J., LABASH C., and SMITH C. "GPS for environmental applications: accuracy and precision of locational data." Edited by Bethesda, MD, ETATS-UNIS American Society for Photogrammetry and Remote Sensing. *Photogrammetric engineering and remote sensing* 60 (1994): 41-45.

3. Chan, Mun Choon, and Ramjee Ramachandrian. "TCP/IP Performance over 3G Wireless Links with Rate and Delay Variation." *Wireless Networks* (Springer Netherlands) 11 (Jan 2005): 81-97.

4. Encyclopædia Britannica. “taxonomy.” *Encyclopædia Britannica.* 2008. http://search.eb.com/eb/article-9110579 (accessed 03 25, 2008).

5. Fielding, Roy T. “Principled Design of the Modern Web Architecture.” *ACM Transactions on Internet Technology*, 05 2002: 115-50.

6. Glaser, William T., Timothy B. Westergren, Etienne F. Handman, and Thomas J. Conrad. Playlist Generating Methods. US Patent APP11/295,339. 06 12 2005.

7. Holmquist, Lars Erik. “Tagging the World.” *interactions*, 07 2006: 51,63.

8. Lathem, Jonathan Douglas. “SA-REST: Adding Semantics to REST-Based Web Services.” Master's Thesis, University of Georgia, Athens, Georgia, 2007.

9. López-de-Ipiña, D., J.I. Vasquez, and J. Abaitua. “A context-aware mobile mash-up platform for ubiquitous web.” *Intelligent Environments, 2007. IE 07. 3rd IET International Conference on.* Ulm, Germany, 2007. 116-123.

10. Mathes, Adam. *Folksonomies - Cooperative Classification and Communication Through Shared Metadata.* 12 2004. http://www.adammathes.com/academic/computer-mediated-communication/folksonomies.html (accessed 03 25, 2008).

11. Nokia Corporation. “Recommendations for Reducing Power Consumption of Always-on Applications.” *Forum Nokia.* 26 Sep 2007. http://www.forum.nokia.com/info/sw.nokia.com/id/a6d789aa-9321-444e-a3c5-27cc7faa9ccb/Recommendations_for_Reducing_Power_Consumption.html (accessed Aug 08, 2008).

12. Sinha, Rashmi. *A cognitive analysis of tagging (or how the lower cognitive cost of tagging makes it popular).* 27 09 2005. http://www.rashmisinha.com/archives/05_09/tagging-cognitive.html (accessed 03 25, 2008).

13. Sinha, Rashmi, and Kirsten Swearingen. “Comparing Recommendations Made by Online Systems and Friends .” SIMS, University of California . 2001. http://www.ercim.org/publication/ws-proceedings/DelNoe02/RashmiSinha.pdf.

14. W3C. “OpenAjax Workshop on Mobile Ajax.” *World Wide Web Consortium.* 28 09 2007. http://www.w3.org/2007/09/28-mobile-ajax-minutes.html (accessed 03 27, 2008).

15. Wal, Thomas Vander. “Folksonomy Explanations.” *Off The Top - Vanderwal.net.* 15 01 2005. http://vanderwal.net/random/entrysel.php?blog=1635 (accessed 03 25, 2008).

16. Weinberger, David. “Taxonomies to Tags: From Trees to Piles of Leaves.” *Esther Dyson's Monthly Report: Release 1.0*, 2 2005.

Appendix A: Testing Data

All times in milliseconds

Table 4: Server Performance Test Data

PPI	Total Time	Queries	DBTime	Entities	Posts	PTime	Loc's	Prof's	Pool	Ranking
10	422	281	359	120	30	233	60	30	343	63
20	735	541	454	230	60	421	120	60	594	110
30	1078	801	700	340	90	674	180	90	891	156
40	1422	1061	1032	450	120	922	240	120	1188	219
50	1734	1321	1061	560	150	1126	300	150	1453	265
60	2093	1581	1438	670	180	1261	360	180	1703	359
80	2797	2101	1671	890	230	1749	480	240	2297	468
100	3578	2621	2437	1110	300	2422	600	300	3000	547
200	6938	5289	4485	2250	600	4718	1200	613	5703	1219
500	18812	14285	12020	6262	1500	12556	3000	1730	15109	3672
1000[32]	30250	23081	18615	10798	2076	20164	4152	2777	23859	6360

Table 5: Adjusted Server Performance Data

PPI	Compensated Pool	Compensated Search Time
10	110	173
20	173	283
30	217	373
40	266	485
50	327	592
60	442	801
80	548	1016
100	578	1125
200	985	2204
500	2553	6225
1000[32]	3695	10055

Table 6: Emulator Performance for Client

NumPosts	Total Time	Network Time	µjena Time	10.1 object parse time	Client Library Time
10	4469	3640	250	438	4328
20	6016	3828	609	1392	5829
40	12001	3906	1890	6001	11797
80	34048	4516	5016	24282	33814

[32] Note that there are not 1000 unique posts in the database, so this test is skewed to take less time than it should.

Table 7: K850i Performance for Client[33]

NumPosts	Total Time	Network Time	µjena Time	object parse time	Client Library Time
10[34]	10942	4447	1241	1645	7332
20[35]	19629	5190	4218	4930	14338
40[36]	28704	5111	5559	14148	24818
80[37]	108146	6761	18017	78891	103669

[33] Raw testing data is not included in any appendix due to the significant volume of it and that it's not very interesting.
[34] Average over 10 tests
[35] Average over 10 tests
[36] Average over 5 tests
[37] Note that, unlike the other tests, the 80 posts one only has one value. This is because tests on 80 posts either failed outright due to the phone running out of memory or succeeded in absurd amounts of time, such as 6 minutes.

Appendix B: Ericsson Material

Some of the material in this thesis references internal materials, sources or other intellectual property of Telefonaktiebolaget LM Ericsson. Their inclusion or reference in this paper is in no way to be regarded as a relinquishing of rights, or other permission for their usage or dissemination.

B.1 Test Data Set

The data set used for the testing of the algorithm consisted of approximately 1100 data points in Kista, Stockholm, Sweden. These data points included a short description and a geographic location, although the data was incomplete for some. For inclusion in the database, they were also assigned tags in a spontaneous manner, between 1 and 3 tags for each data point. When the data was parsed into the system, 737 data points were successfully translated into posts with geodata. The remainder of the data points were for various reasons – such as incomplete data, character incompatibility or parsing failure – not included into the database. This loss was considered acceptable, since enough data points remained to make sufficient tests.