Cloud-TM

Specific Targeted Research Project (STReP)

Contract no. 257784

D4.1: Initial Prototype of Pilot Application 1.

Date of preparation: 31 January 2012
Start date of project: 1 June 2010
Duration: 36 Months

Contributors

Vittorio Amos Ziparo, Algorithmica
Fabio Cottefoglie, Algorithmica
Marco Zaratti, Algorithmica
Francesca Giannone, Algorithmica
Paolo Romano, INESC-ID
Sebastiano Peluso, INESC-ID
Joao Cachopo, INESC-ID
Sérgio Miguel Fernandes, INESC-ID
Sanne Grinovero, RHAT
Manik Surtani, RHAT

Table of Contents

1 Introduction

Apps for smart-phones are rapidly gaining attention as they are considered to be the core of the next generation of applications for the Internet [1]. A key distinguishing feature of smart-phone apps is that they can be used from almost any location, providing information on the position of its user through a GPS device or by triangulation on WI-FI access points and Cellular towers. Thus, it is not surprising that many smart-phone apps have Location-based Mobile Social Networking (LMSN), aka geo-social networking, facilities. The basic idea behind LMSN is to provide a second generation of Social Networks (a killer application for the web) that can take advantage of the position of the users in order to provide innovative services. Notably, LMSN are supported by a significant market share, as ABI Research estimates LMSN will generate global revenues of $3.3 billion by 2013 [2]. The LMSN market is already very rich and features a wide spectrum of products developed both by well known corporations and by independent players (see [3] for an exhaustive list of products).

For example, both Google and Microsoft developed their own LMSNs, called Latitude [4] and Vine [5], respectively. Nokia, instead, recently bought a similar product from an independent player, called Plazes [6]. Pairwise, there are many independent companies which are developing their own innovative products [7–12].

Pinning down mobile social networks to their core, one can think of such systems as client-server architectures, where:

- clients are smart-phones that periodically send location data and that can retrieve data on the state of the social network;
- server side applications are in charge of maintaining a consistent representation of the data and are required to provide up to date information on the relations among users based on their location. For example, a location mobile social network may require to maintain a data structure that dynamically keeps track of all the friends of each user within a given distance.

In this document, we describe the initial prototype of GeoGraph: the first pilot of the Cloud-TM framework. GeoGraph is an open-source framework (distributed under the terms of the GNU Lesser General Public License) for tracking location data and dynamically computing relationships among users. In particular, GeoGraph maintains a graph (called GeoGraph) where nodes are users and where edges connect users who are within a given distance from each other.

GeoGraph may experience sudden peaks of the load, due to the exponential growth phenomenon exhibited by social networks. Moreover, LMSN may experience flash crowds triggered by social events, like concerts or conferences that cause hot-spots in specific geographical regions. Indeed, flash crowds often overload web systems to a point when their services are degraded or disrupted entirely [13]. Being a generic framework, GeoGraph supports the development of a wide range of LMSN applications characterized by highly heterogeneous and dynamic workloads exhibiting diverse data access patterns and conflict rates. This will provide Cloud-TM with realistic use cases to assess the effectiveness of its self-tuning mechanisms

In fact, on one hand, GeoGraph's clients frequently update the state of the graph nodes (i.e., position of users). On the other one hand, the server side part of the application is responsible for using this information to dynamically compute the topology of the graph, i.e.: adding edges to nodes (users) that get close to each other, delete edges among nodes that move away from each other. In order to enhance scalability and ensure a timely update of the graph, these updates are performed in parallel, by multiple threads possibly residing on different nodes. Thus, the data access pattern of GeoGraph (or, more precisely, of the LMSN applications developed using the GeoGraph's framework) appears prone to generate high contention levels.

GeoGraph is a generic framework that will support the development of a wide range of LMSN applications characterized by highly heterogeneous and dynamic workloads. This will provide realistic use cases to assess the effectiveness of the Cloud-TM platform. In the following, we describe the workload profiles generated by some of the existing LMSN applications:

1. **- low traffic, low conflicts, read-dominated -** Example: Plazes [6], Latitude [4] and Vine [5] implement variants of micro-blogging (Twitter-style) systems where posts are geo-localized. Notice that, in this case there is no conflict on data because there is no need to compute the GeoGraph. The system must only associate locations to posts. In this type of applications, the frequency of writes (posts) is generally lower than the frequency of reads (people reading the posts). When first launched on the market, systems like Plazes or Latitude will probably have a low traffic because the frequency of posting is low and the user base is small.

2. **- hi traffic, low conflict, read dominated -** Example: When applications like Plazes and Latitude increase their popularity, and thus their user base grows, they may incur in high traffic peaks. Nevertheless, their workload is dominated by read operations.

3. **- hi traffic, low conflict, write dominated -** Example: Ipoki [11] and Bilin [12] allow you to share your location in real time. In this case, clients send to the server with a relatively high frequency (say one 1Hz) their location. The location is then made available to other users on a map. In this case, also with a limited amount of users, clients can generate a high traffic on the server, which is mainly write dominated. Nevertheless, there are no conflicts because there is no shared data accessed concurrently. Notice that, in this case there is no need to compute the GeoGraph because users can see users close to them simply by inspecting the map.

4. **- hi traffic, hi conflict, write dominated -** Example: Avego [14] and Carticipate [15] allow for real-time ride-sharing. In this type of applications, such as for Ipoki and Bilin, clients update their location with a high frequency thus generating a workload that is write dominated. In this case, the GeoGraph is needed to cluster users in complex ways. Nevertheless, when updating edges, different threads may conflict while executing update transactions and, due to the high frequency of the updates of the location of nodes, other conflicts may arise when the data is fetched from the threads in charge of updating the GeoGraph topology (in a similar way to the Bayesian Network Benchmark described in [16]).

2 MADMASS

The MAssively Distributed Multi Agent System Simulator (MADMASS) is an open-source framework (distributed under the terms of the GNU Lesser General Public License) for developing rich-client web applications that require scalability and feature (real-time) interactions among users. Target applications include, but are not limited to, Multi-Player Online Games, Transaction Processing Systems, Location-based Mobile Social Networks (or geo-social networks) and cooperative systems (e.g., crowd-sourcing apps). MADMASS has been developed in the context of the Cloud-TM project and it constitutes the core of the two Cloud-TM pilots. MADMASS is available at `https://github.com/algorithmica/madmass`.

MAssively Distributed. MADMASS is at the core of the Cloud-TM pilots, and as such, it is designed for the Cloud. MADMASS relies on the best practices for developing apps for the Cloud and integrates seamlessly with the Cloud-TM platform.

Multi Agent System Simulator. MADMASS has its roots in Artificial Intelligence and Multi-Agent Systems research. These disciplines provide the foundations of the MADMASS programming paradigm, making it a versatile and intuitive framework for developing complex applications that feature a high degree of interaction among users.

The MADMASS project stands on the shoulders of a number of existing open-source projects. MADMASS is a Ruby On Rails[1] Engine. Thus whatever you can do with rails, you can do it with MADMASS too. It supports opensource (Socky, Stilts) and commercial (Pusher) WebSockets[2] implementations for enabling real-time interactions. Rich browser GUIs are possible thanks to HTML5[3] and to javascript frameworks such as MooTools[4] and JQuery[5].

We leverage on the JBoss technology to deliver MADMASS apps as enterprise apps. We rely on TorqueBox[6] for extending the footprint of Rails application and enabling functionalities such as clustering, load-balancing and high-availability out-of-the-box. TorqueBox provides an all-in-one environment, built upon the latest, most powerful JBoss AS Java application server [17].

A MADMASS app is composed of many (intelligent) agents that offer one or more services. When a user contacts a MADMASS app, an agent is assigned to him. The role of this agent is to give the user access to a virtual environment (e.g., a virtual world, a data repository, a social network, ...). Some examples of such apps include, but are not limited to:

- Massively multi-player online games, where agents provide access to the virtual world and ensure that the game is played according to the rules.
- Geo-social apps, where agents can promote social behavior, while ensuring safety and privacy.

[1] `http://rubyonrails.org/`
[2] `http://en.wikipedia.org/wiki/WebSocket`
[3] `http://en.wikipedia.org/wiki/HTML5`
[4] `http://mootools.net/`
[5] `http://jquery.com/`
[6] `http://torquebox.org/`

- Personal assistant services, where agents can help to optimize schedules, manage shared agendas and calendars, or coordinate industrial processes.
- Mechanisms design, where agents implement distributed protocols for reputation, auctions, trading, crowd-sourcing and, in general, for both cooperative and competitive systems.
- Data management and mining, where agents can help to access and manage knowledge, as, for example, virtual guides in online museums or libraries.

MADMASS makes your life easier when writing MAS apps, as it automatically manages the whole Multi-Agent System execution (technically called, simulation). To this end, developing a MAS app, amounts to developing GUIs, agents and environments. Environments are the data tier of a MAS app and can be coded with any suitable technology (e.g., ActiveRecord [18] or the Cloud-TM data platform [19]). Agents encode the business logic of your application, and must be defined by using a specific Action Pattern that greatly simplifies the development of complex applications. The development of GUIs is supported by a powerful javascript framework that allows for rich interfaces in the browser.

A MADMASS application is composed of three distinct communities of Agents (see Figure 1):

- **Domain Agents**, that are short-lived agents that execute actions on behalf of other agents while enforcing the business logics of the application.
- **Human Agents**, that are the end-users of the system typically accessing the system through browsers or mobile clients.
- **Autonomous Agents**, that are long living agents serving as system daemons, personal assistants, game bots or user simulators. Within each community, agents can have different roles and functions depending on the application domain.

Domain agents expose their services through a JMS interface. Specifically, there is a queue for incoming commands (i.e., the actions queue) and a topic to which clients can subscribe for updates on the domain model (i.e., percepts topic). One of the advantages of using JMS is that agents can be implemented as message processors that are automatically load balanced and clustered by the JMS server (i.e., HornetQ in this case). Moreover, JMS selectors allow us to enforce visibility (and thus privacy) rules on the domain. In order to provide access to the JMS queue and topic over the Internet (e.g., to browsers) we use Stomplets [7] that provide a bridge to WebSockets.

For example, in GeoGraph, the domain model is a graph where nodes are geo-localized entities and edges represent proximity relationships. The graph is persisted on the DSTM and the Domain agents have the responsibility of maintaining this data structure on behalf of the users. A GeoGraph client may, for example, want to post a new geo-localized comment at some location. In order to do so, it will send a "micro-blog post" request to the action queue. When the request is received an agent will be put in charge of handling such request. In particular, he will

[7] `http://stilts.projectodd.org/stilts-stomplet/`

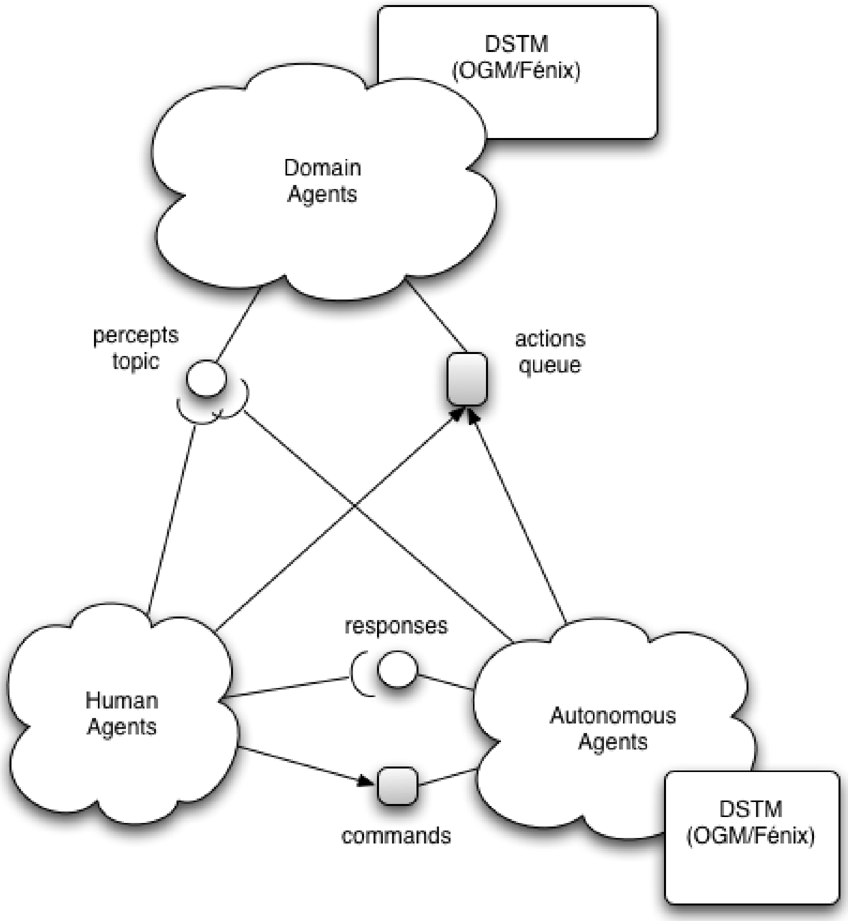


Figure 1: MADMASS Architecture

1. authenticate the user;
2. verify the applicability of the action (e.g., is the user close to where he wants to post?);
3. make the requested post;
4. notify the post to all interested users (i.e., using the percepts topic).

MADMASS also provides a Javascript library for developing user interfaces for Human Agents that simplify both the communication process and the development of rich interfaces that support real-time updates, complex event oriented interfaces, animations and sound.

The third community of agents, Autonomous Agents, can be used to perform several tasks. For the purpose of GeoGraph, Autonomous Agents are used to simulate users. For example, simulators can be used to benchmark the application or, in our case, the underlying Cloud Platform. The Autonomous Agents interact with the domain in the same way human agents do: they send commands to the actions queue and get updates on the environment state through the percepts topic. Moreover, Autonomous Agents offer an interface to Human Agents for managing the Autonomous Agent Community. In particular, authorized users can create groups of agents, possibly of different types. They can also set a simulation speed, pause, start, stop and destroy each group. A more concrete example will be provided when describing the GeoGraph workload generator in Section 3.2.

2.1 Why Transactions?

The MADMASS architecture allows for naturally scaling and well serves for delivering Software as a Service (SaaS) in a Cloud. Nevertheless, the presence of many concurrent agents may lead to conflicts that can put at risk the correctness of the process. To avoid this problem, all actions are performed within a transactional context.

The basic idea is that any operation of the data model must be defined in terms of actions. Actions have a simple, yet powerful, interface that is composed of the following three methods:

1. $boolean\ applicable?()$ This method is in charge of defining when an action can be executed. For example, a buy item action is applicable if the user has enough credit for buying that particular item. This type of method requires only to read data in the domain model.
2. $void\ execute()$ This method describes how the domain state changes upon the execution of an action. For example, in a buy item action, the execution subtracts the cost from the buyer's account, adds the cost to the seller's account and moves the item from the buyer to the seller. This type of method usually requires both reads and writes on the domain model.
3. $percepts\ build\_perception()$ is in charge of returning the changes in the environment produced by the execution of the action. In order to be transmitted over the web, such changes are represented as a Hash of strings.

The following pseudo code shows the execution of an action act:

Listing 1: Pseudo-code for Action Execution

```
transaction do{
 if act.applicable?
    act.execute
 percepts = act.build_percepts
}
send(percepts)
```

Thus, depending on the type of action and the state of the environment we can have very different type of transactions:

- *read/write transactions.* This is the case of a successfully executed action. The action reads the domain model to verify if it is applicable, and then, it writes the domain model to enforce its effects, and finally, reads the domain model again to build the percepts to be sent through the network.
- *read-only transactions.* This case can happen in two different scenarios. The first, and most common, is the case of sensing actions. These actions have no effects and their only purpose is to perceive the current state of the system. In GeoGraph, this can happen when a user scrolls a map or clicks on a geo-localized post to read it. The second case, is when an action fails. Indeed, such an action verifies that the action is not applicable and builds an error percept. Both operations are clearly read-only.
- *write-only transactions.* This is the case of blind actions, i.e., actions with no preconditions. An example of blind actions, is the move action where agents update the user's position without needing to verify any precondition.

3 The GeoGraph Pilot

GeoGraph is a geo-social MADMASS app. Being developed on top of MADMASS, GeoGraph is extremely flexible and can be used as a basis for the development of any geo-social app as it implements a set of services, in terms of actions, that are commonly used in many geo-social apps (e.g., position tracking and micro-blogging). To this end, implementing a new geo-social just amounts to developing a new client with the MADMASS GUI. Also extending the services provided by the Domain Agents is fairly simple, as it is enough to define new actions. As a key feature, GeoGraph comes with a load generator that simulates users and that can be used to benchmark applications.

GeoGraph is composed of two components:

1. The Geograph Domain, that implements the Domain Agents[8] and the Domain Model[9] by using the Object Grid Mapper of the Cloud-TM Data Platform Programming API.

2. Geograph Agent Farm, a workload generator that implements a community of autonomous agents simulating GeoGraph users[10] and the Agent Farm Domain Model[11].

3.1 GeoGraph Domain Agents

The GeoGraph domain is a graph where nodes are GeoObjects and where edges represent proximity relations. GeoObjects can be of several types. For the time being, we have two main types of GeoObjects: moving objects (such as pedestrians, bikers and drivers) and still objects (such as micro-blog posts). GeoObjects are associated to Users, and as such a User can have many GeoObjects.

The domain model has been described by using the Cloud-TM DML [19] that allows us to model the domain model once, and generate two different implementations (i.e., Hibernate OGM and Fénix) that can then be benchmarked one against the other.

GeoGraph Domain Agents offer the following set of geo-social services:

1. **Create a Movable GeoObject.** Users can be associated to movable geo-objects that represent their position. This action is usually performed when a user enables a position tracking feature.

2. **Destroy a Movable GeoObject.** Users can destroy their associated geo-object. This action is usually performed when a user disables the position tracking feature.

3. **Move a Movable GeoObject.** Users can change the position of their associated geo-object. This action is performed when a user that enabled position tracking moves. Notice that this action, depending on the speed of the users, can be performed very frequently generating a high write-intensive workload.

[8] `https://github.com/algorithmica/geograph`

[9] `https://github.com/algorithmica/geograph-domain`

[10] `https://github.com/algorithmica/geograph-agent-farm`

[11] `https://github.com/algorithmica/geograph-agent-farm-domain`

4. **Create Micro-blog post.** Users can post a comment in a specific geographical location. Such comments can be, for example, reviews of some event, location, restaurant or a sight , or simply notes from a vacation log. These operations are mainly writes, but are not as frequent (on a per user basis) as move actions.

5. **Destroy/Edit Micro-blog post.** Users can remove or revise existing posts. This type of operation can be also performed by administrators that may want to remove or edit offending posts.

6. **Read Micro-blog post.** Users can read existing posts close to their location or by inspecting a map. Clearly, this type of operation is read-only.

7. **Update GeoGraph.** Domain Agents are in charge of maintaining the geo-social graph by updating nodes and edges. This is the core operation of GeoGraph as this graph structure is fundamental for data mining activities (such as clustering of users) that are fundamental to any social network. In the current prototype we have implemented two separate versions of the graph maintenance service:

   (a) A first version is implemented as a periodic task that at given intervals iterates over the entire graph and updates its topology.

   (b) A second version is implemented as a side-effect of performing actions. Every time a GeoObject is created or moved, the action iterates over the nodes to update its connections.

The GeoGraph Domain Agents app offers an interface (see Figure 2) to monitor the evolution of the system. By using the Google Maps API, we show the current set of GeoObjects as Markers on a map. By using the MADMASS architecture, updates on the map are directly pushed on the client. As a result, the Map elements are animated and move on the map. The monitoring interface also shows the graph structure by depicting the edges that connect the GeoObjects. Finally, it is possible to inspect the GeoObjects by clicking on the markers. This operation opens the info window with all the relevant information (e.g., text of a post -if a post-, coordinates, type of object).

3.2 GeoGraph Agent Farm

The GeoGraph Agent Farm is a set of MADMASS Autonomous Agents used to simulate users in order to benchmark GeoGraph and the underlying Cloud-TM platform. Figure 3 shows an overview of the web interface to the GeoGraph Agent Farm. Figure 4 a) depicts the interface for selecting the strategy to update the edges within the GeoGraph. Figure 4 b), instead, shows the interface for creating new groups of simulated users (new group button). Upon the creation of a new group, one can specify the following parameters:

- name of the group;
- type of simulated user (i.e., agents' type);
- number of agents within the group;

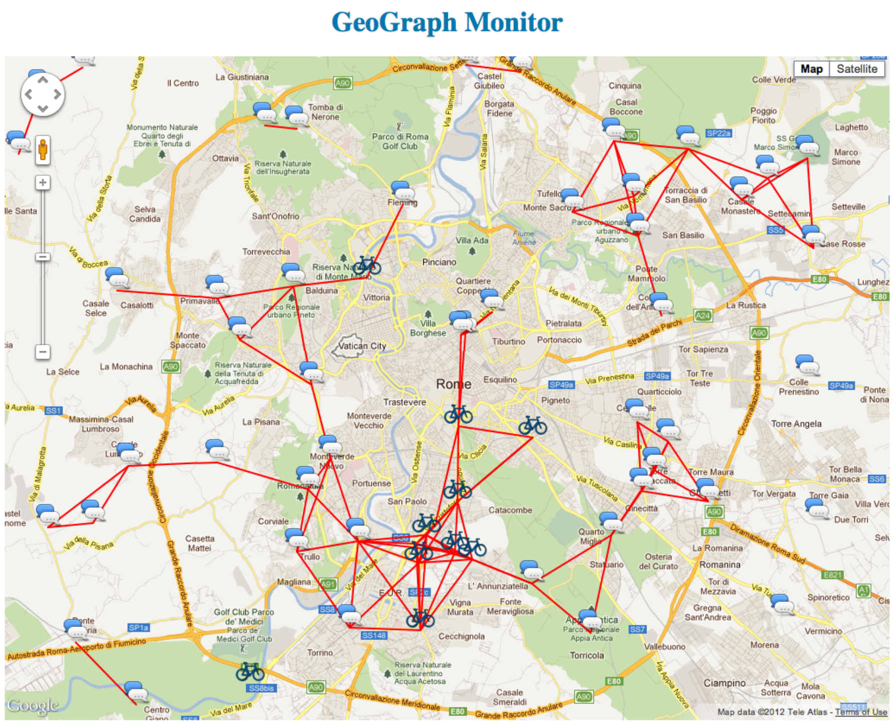
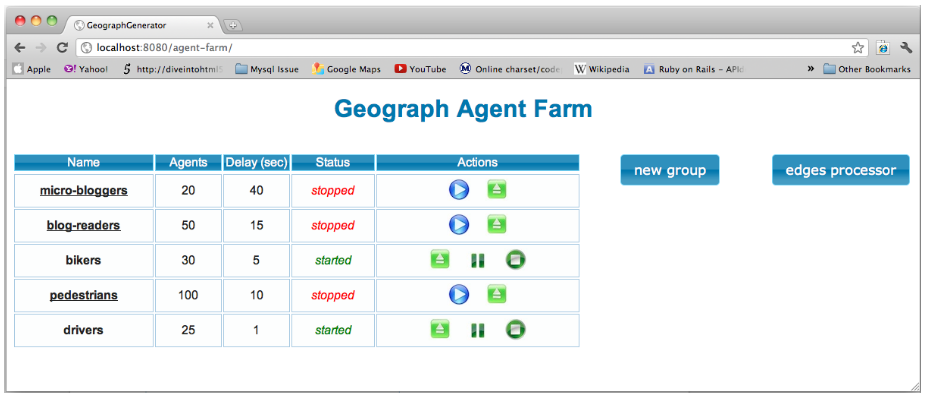
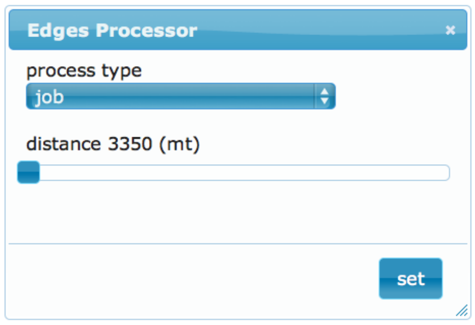


Figure 2: GeoGraph Domain Interface



Name	Agents	Delay (sec)	Status	Actions
micro-bloggers	20	40	stopped	▶ ⏏
blog-readers	50	15	stopped	▶ ⏏
bikers	30	5	started	⏏ ⏸ ⏹
pedestrians	100	10	stopped	▶ ⏏
drivers	25	1	started	⏏ ⏸ ⏹

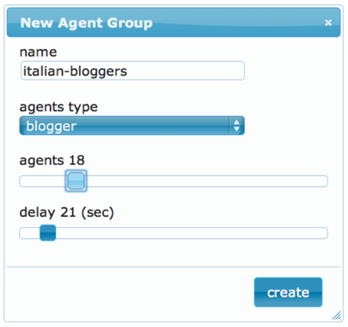
Figure 3: Agent Farm Interface



process type: job

distance 3350 (mt)

(a)



name: italian-bloggers

agents type: blogger

agents 18

delay 21 (sec)

(b)

Figure 4: a) GeoGraph Update strategy selection b) Creation of a new Agent Group

- delay between two subsequent actions.

The properties of an agent group can also be edited at run-time (i.e., after the creation). The web interface (see Figure 3) shows the list of all agent groups created and allows to control each group independently. In particular, a user can start one or more agent groups (play button) to test the desired configuration of the user community and therefore the resulting workload profile. All agent groups can also be destroyed (eject button), paused (pause button) and stopped (stop button). Notice that the stop behavior resets the behavior of the agent from the beginning, while the pause just stores the current state of execution. Thus when playing a stopped agent, the agent will start over (e.g., from his initial position), while when playing a paused agent, the agent will start from where he was paused (e.g., his last position).

We have currently implemented three types of simulated users:

- **Bloggers:** These type of simulated users randomly post comments. They simulate users posting comments on a map, such as travel log books.
- **Readers:** These type of simulated users randomly read blog posts. They simulate users browsing a map and reading related posts.
- **Movers:** These type of users move on the map while having enabled the tracing option. The agents select paths randomly among a path database. The path database consists of real paths available from the Internet and encoded in GPX, i.e., GPS eXchange Format[12]. GPX is an XML schema designed as a common GPS data format for software applications. It can be used to describe waypoints, tracks, and routes.

[12]`http://en.wikipedia.org/wiki/GPS_eXchange_Format`

4 Data Contention

In GeoGraph, the data structure maintaining locations and relations among users (i.e., a graph) is stored server-side, in the Distributed Software Transactional Memory platform at the core of Cloud-TM. This data structure will then be concurrently updated by a variable number of processing threads (physically distributed across a dynamically variable number of machines) to reflect the alteration of the geographical position of the users, and accordingly update the graph data structure.

As already mentioned previously, there are currently two methods for updating the edge data structure. In the following, we describe one of these approaches to highlight data contention issues. Nevertheless, similar considerations apply also to the other method.

Using simplified pseudo-code, an example of transaction used in GeoGraph to alter the graph topology could be the following:

Listing 2: Example pseudo code for Graph Update

```
1.  #upon reception of  a new position of some user "u"
    on_update {
         #atomic transactions
2.       transaction do {
            #retrieve graph node associated with user u
3.          myNode=Graph.get_node_of_user(u);
            #update (i.e. write) position of myNode
4.          myNode.update_position();
            #remove edges with current neighbour nodes
            #that are now farther away than some threshold K
5.          for_each n in neighbors(myNode){
               #read position of node "n"
               #read list of neighbor nodes of "myNode"
6.             if(distance(n,myNode) >= someThresholdK)
                   #update (i.e. write) list of neighbor nodes of
                   #"myNode" and "n"
7.                remove_edge(n,myNode)
8.          }
            #add edges with graph nodes that are
            #within some threshold K
9.          for_each_other_node n in Graph {
              #read position of node "n"
10.           if(distance(n,myNode) < someThresholdK)
                 # update (i.e. write) list of
                 #neighbor nodes of "n" and "myNode"
11.              add_edge(n,myNode)
12.         }
13.      }
14. }
```

This code block will be executed whenever a client updates his position, with a frequency that depends on the actual mobility patterns of users, ranging from very slow (e.g with users strolling around the city) to very fast (e.g. users traveling by car or train). As a consequence of the parallel manipulation of the graph, conflicts will arise on the data structures (e.g. lists) maintaining the set of edges between each pair of nodes.

For instance, assume that a transaction T executes line 5 and determines that node n is currently a neighbor of $myNode$. Now, if, before T is committed, n moves away and a transaction $T^{\star}$ removes n from the list of neighbors of $myNode$, the transaction T will have to be aborted since it has executed on a stale snapshot.

Other read/write conflicts may arise between lines 3, 6 and 10 of two concurrent transactions, where the former one updates the position of a node and the latter ones read this position to determine whether the graph topology should be altered.

As a final remark, note that line 9 of the pseudo-code adopts a naif approach that will be analyzed and improved during the following months. For example, in GeoGraph only a subset of the graph's nodes will be considered into this "for" cycle. To this end, GeoGraph could adopt heuristics that will restrict the analysis only to the nodes that are in the same "geographic area" to $myNode$ and/or exploit the indexing provided by the Cloud-TM search API. This will contribute to enhance the scalability of the algorithm.

5 Conclusions and Future Work

The current prototype of GeoGraph already includes all the core functionalities of many geo-social applications (i.e., position tracking, micro-blogging, social tracking). The prototype has been integrated with the Object Grid Mapper of the Cloud-TM Data Platform Programming API and can be used for preliminary benchmarking of the Cloud-TM Platform.

The GeoGraph Agent farm allows for simulating several synthetic workloads that span over the spectrum of profiles of typical geo-social apps. At this stage, we have already implemented a set of Autonomous Agents in the GeoGraph Agent Farm that can exhibit either a read-dominated or write-dominated workload profile, depending on their type.

The agent farm allows for dynamically varying the profile of the workload along two distinct dimensions: 1) read/write ratio and 2) intensity. Indeed, by deploying multiple groups with different numbers and types of agents, we can vary the read/write ration. Moreover, by changing the number of agents and the speeds at which they perform actions, we can change the intensity of the workload. Notice that, as agent groups can be edited at run-time, we can evolve the workload profile dynamically during the execution of experiments.

Let us consider the scenarios and workload profiles described in Section 1, at page 5, and how they can be replicated by using the current implementation of the GeoGraph Agent Farm. Consistently, with the description provided above, in the first three scenarios the Graph Update services will be switched off and thus there will be low contention on data:

1. **- low traffic, low conflicts, read-dominated -** We launch a small number of Bloggers that start posting at a very low frequency and larger, yet small, number of Readers that read at a medium frequency.

2. **- hi traffic, low conflict, read dominated -** The previous scenario, can be modified to generate higher traffic, for example by increasing the number of agents or the frequency at which they produce.

3. **- hi traffic, low conflict, write dominated -** In this scenario we will deploy many Movers that will require position tracking. Several types of users can be simulated by adjusting the speed at which agents move (i.e., the frequency at which they send updates). For example, pedestrians could update their position every 10 seconds, bikers every 5 and drivers every 2.

4. **- hi traffic, hi conflict, write dominated -** If we enable the Graph Update in the previous scenario, a big number of conflicts will arise as described in Section 4, but the workload profile will not vary its characteristics.

In the near future, we will investigate if there is the need to provide other workload profiles as a result of the preliminary benchmarks. Besides this, we are already planning to improve the current version in several ways. The current handling of the Graph Update is rather naif, and we predict that there may be scalability issues. One way of addressing the problem is to integrate with the Cloud-TM Search API that will deliver an efficient implementation of geographical queries. In case this is not enough, we will design more scalable algorithms for Graph Update, some of which are already under discussion within the consortium.

There can be potentially a huge number of computational tasks that run in parallel both in the GeoGraph Agent Farm and in GeoGraph itself. To date, these tasks are clustered and load balanced by using HornetQ. However, this approach does not take into account locality of data (that is of crucial importance in geographical applications) and it can introduce performance overheads as the number of nodes grows. To address this issue, we plan to integrate with the Cloud-TM Distributed Execution Framework (DEF) as it allows for placing computational tasks close to the data that will be accessed by this task. Finally, DEF will allow us to synchronize the execution of the tasks (through Joins and Forks), features that would be greatly beneficial for a more accurate control of the GeoGraph Agent Farm.

A Getting Started

In the following we provide a brief quick start guide for running GeoGraph and GeoGraph Agent Farm. It is highly recommended to follow the latest instructions available online at the bottom of the following pages:

- `https://github.com/algorithmica/geograph`
- `https://github.com/algorithmica/geograph-agent-farm`

At first run GeoGraph, by performing the following steps:

1. Install TorqueBox v2.0.0.beta3[13].
2. Clone the project from the git repository:

   ```
   git clone git://github.com/algorithmica/geograph.git
   ```

3. Install the needed gem libraries: open a shell, cd to the project folder and run

   ```
   jruby -S bundle install
   ```

   Note: if you are on a Linux machine you must add two gems to the Gemfile before executing the bundle install open the Gemfile (in the root of the application) and add

   ```
   gem 'execjs'
   gem 'therubyracer'
   ```

4. Setup the database (make sure sqlite3 is installed):

   ```
   jruby -S rake db:setup
   ```

   Note: The Sqlite3 database is used exclusively for the authentication of the admin user.

5. Deploy the application into TorqueBox by executing this command in the project folder:

   ```
   jruby -S rake torquebox:deploy
   ```

6. Run TorqueBox:

   ```
   jruby -S rake torquebox:run
   ```

7. Run the Socky Websockets server by executing this command in the project folder:

   ```
   jruby -S socky -c socky_server.yml
   ```

[13]download it (`http://torquebox.org/release/org/torquebox/torquebox-dist/2.0.0.beta3/torquebox-dist-2.0.0.beta3-bin.zip`) and follow the installation instructions (`http://torquebox.org/documentation/current/installation.html`)

8. Open the browser at `localhost:8080`, signup and you will see the GeoGraph map.

Then, to run the GepGraph Agent Farm do as follows:

1. Clone the project from the git repository:

   ```
   git clone
   git://github.com/algorithmica/geograph-agent-farm.git
   ```

2. Install the needed gem libraries: open a shell, cd to the project folder and run

   ```
   jruby -S bundle install
   ```

   Note: if you are on a linux machine you must add two gems to the Gemfile before executing the bundle install open the Gemfile (in the root of the application) and add

   ```
   gem 'execjs'
   gem 'therubyracer'
   ```

3. Setup the database (make sure sqlite3 is installed):

   ```
   jruby -S rake db:setup
   ```

   Note: The Sqlite3 database is used exclusively for the authentication of the admin user.

4. Deploy the application into TorqueBox by executing this command in the project folder:

   ```
   jruby -S rake torquebox:deploy['/agent-farm']
   ```

5. Run TorqueBox:

   ```
   jruby -S rake torquebox:run
   ```

6. Open the browser at `localhost:8080/agent-farm`, signup and you will see the GeoGraph Agent Farm console.

References

[1] WIRED Magazine, "The web is dead. long live the internet." http://www.wired.com/magazine/2010/08/ff_webrip/all/1, 2010.

[2] Abi Research, "Location-based mobile social networking." http://www.abiresearch.com/press/1204-Locationbased+Mobile+Social+Networking+Will+Generate+Global+Revenues+of+$3.3+Billion+by+2013, 2008.

[3] Claudio Schapsis, "Location based social networks links." bdnooz.com/lbsn-location-based-socialnetworking-links.

[4] Google Inc., "Latitude." http://www.google.com/intl/en_us/latitude/intro.html.

[5] Microsoft Corporation, "Vine." http://www.vine.net/default.aspx.

[6] Plazes AG, "Plazes." http://plazes.com.

[7] Pelago Inc, "Whrrl." http://whrrl.com.

[8] BrightKite Inc, "Brightkite." http://brightkite.com.

[9] Loopt Inc, "Loopt." www.loopt.com.

[10] GeoSolutions, B.V., "Gypsii." http://www.gypsii.com/.

[11] Ipoki Technologies S.L, "Ipoki." www.ipoki.com.

[12] Bliin B.V., "Bilin." bliin.com.

[13] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites," in *11th International WWW Conference*, (Honolulu, HI), May 2002.

[14] "Avego." www.avego.com.

[15] "Carticipate." www.carticipate.com.

[16] Chi Cao Minh, Jae Woong Chung, Christos Kozyrakis, Kunle Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Proc. of The IEEE International Symposium on Workload Characterization*, 2008.

[17] Red Hat Inc, "Jboss application server 7." www.jboss.org/as7.

[18] "Active record - object-relation mapping put on rails." http://ar.rubyonrails.org/.

[19] Emmanuel Bernard, Joao Cachopo, Bruno Ciciani, Diego Didona, Francesca Giannone, Mark Little, Sebastiano Peluso, Francesco Quaglia, Luis Rodrigues, Paolo Romano, Vittorio A. Ziparo, "Cloud-tm deliverable, d2.1: Architecture draft." http://www.gsd.inesc-id.pt/~romanop/files/deliverables/D2_1.pdf, 2011.