Geo2Tag Performance Evaluation

Mark Zaslavskiy, Kirill Krinkin
FRUCT LETI Lab
Saint-Petersburg, Russia
{mark.zaslavskiy, kirill.krinkin}@gmail.com

Abstract

Today volume of the Internet traffic growths very fast. This trend affects Web-application, with the number of users and amount of traffic also increasing their workload. That's why developers need to achieve maximum performance on existing hardware. Software optimization allows solving this problem. Geo2Tag is an open source platform for location-based services (LBS), which provide web interfaces for them. Initially, it was developed as an educational project which goal was to give students experience in open source projects development. But now number of supported functions and number of users (users of LBS and developers) for platform is increasing, and in this situation platform performance is not enough. This paper describes Geo2Tag platform performance evaluation and optimization.

Index Terms: Location-based services, Performance evaluation.

I. Introduction

Nowadays global Internet traffic is about a 44 EB and growing very fast [1]. This huge number means increase of workload at web-services. And hardware update is not a solution because main processor vendors are increasing number of physical cores instead of increasing processor frequency. This fact limits achievable performance gain – it is limited by Amdahl's law [2] and depends from program structure and part of a sequential computations. That's why software optimization is required.

Geo2Tag is an open source platform for Location Based Services (LBS). It was started as educational project, which first goal was to give students experience of open source software development. Now with much new functionality developed by students it goes to non-educational usage as a complete product. In this case such systems characteristics as reliability, performance and security take the first place. At the current moment no studies about this aspects state for Geo2Tag where done. This paper is a first step of Geo2Tag quality research and improvement, and it's focused on platform performance (by the term performance we will understand number of requests which platform can process per second).

II. Performance evaluation

A. Problem statement

Geo2Tag architecture was described in details at recent works [3-4]. From this papers follow that platform performs two global functions - user REST requests processing and synchronization of data between in-memory cache and DB. Goal is to achieve maximum available performance for both functions, because they are executing at one server. So, tasks of this work are:

– Investigate which REST requests are used most frequently.
– Measure the most frequent requests processing performance.

- Profile REST requests processing, determine bottlenecks.
- Profile DB synchronization mechanism, determine bottlenecks.
- Maximize performance for REST requests processing and DB synchronization.
- Compare results before and after optimization.

B. Mathematical modeling of platform clients

For determining the most frequently used requests was performed mathematical modeling of average client application. As client application was chosen Location client [5], because its use cases are the most common among all existing clients [5-7]. As formalism for modeling Markov chains theory were chosen, because it allow making conclusions using small amount of information about modeling system.

Location client performs next requests: *login, subscribeChannel, unsubscribeChannel, subscribedChannels, loadTags, writeTag, applyChannel*. They were chosen as a Markov chain states (1-8). Situation when mobile client are shutting down represents absorbing state [8]. Due to Location client structure – connections between different screens which perform different requests – we can construct system transition matrix:

$$P = \begin{bmatrix} 0 & 0 & 0 & P_{14} & P_{15} & P_{16} & P_{17} & P_{18} \\ 0 & P_{22} & P_{23} & P_{24} & P_{25} & P_{26} & P_{27} & P_{28} \\ 0 & P_{32} & P_{33} & P_{34} & P_{35} & P_{36} & P_{37} & P_{38} \\ 0 & P_{42} & P_{43} & 0 & P_{45} & P_{46} & P_{47} & P_{48} \\ 0 & P_{52} & 0 & P_{54} & P_{55} & P_{56} & 0 & P_{58} \\ 0 & 0 & 0 & P_{64} & P_{65} & P_{66} & 0 & P_{68} \\ 0 & P_{72} & 0 & P_{74} & P_{75} & P_{76} & 0 & P_{78} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\forall i, j = \overline{1,8} : 0 < P_{i,j} \le 1, \forall i = \overline{1,8} : \sum_j P_{i,j} = 1,$$

where $P_{i,j}$ – is a probability to go at j state from i. Zeros in this matrix means that client structure makes impossible to do j request after i request.

Numerical experiments where performed using MATLAB package. Their goal was to determine average count of each request execution before the system goes to absorbing state for different frequency of track sending and program shutdown. For calculations we used next formula:

$$T = (E - Q)^{-1} = \{T_{ij}\}, E \in \mathbb{R}^M, Q \in \mathbb{R}^M,$$

where M – number of non-absorbing states, E – identity matrix of order M, Q – submatrix of P containing all non-absorbing states, T_{ij} – average number of times when system where in state j starting from state i before absorbing. As a result value we take minimal average number of times when system was in state j depending on initial state:

$$T_j = \min_i T_{ij}.$$

For the modeling, non-zero probabilities of matrix P (probability of program shutdown and sending WriteTag request) where defined in the next way:

$$P_{i6} = P_{j6} = P_{writeTag}, P_{i8} = P_{j8} = P_{exit},$$
$$\forall i, j = 1..7.$$

They are defined as independent to the start state because such assumption makes model simpler to calculate and is more realistic that opposite hypothesis – application can stop work properly (sending requests to the server) during many factors, and most of them don't depend on the last send request. Setting P_{i6} is equal for all states, because this request is performed periodically and this period does not depend from previous send request too.

Other non-zero elements of P are selected using:

$$P_{ij} = 2 \cdot \frac{1 - P_{writeTag} - P_{exit}}{k+2}, j = \{4,5\},$$
$$P_{ij} = \frac{1 - P_{writeTag} - P_{exit}}{k+2}, j = \{2,3,7\},$$
$$i = 1..7,$$

where k – number of non-zero probabilities in j string of matrix P. These probabilities are selected equal also for the simplicity – we have not enough statistics about user behavior.

Experiment contains one simulation for fixed value of $P_{exit} = 0.05$ and varying $P_{writeTag}$ from 0 to 0.95.

Results of modeling are represented below:

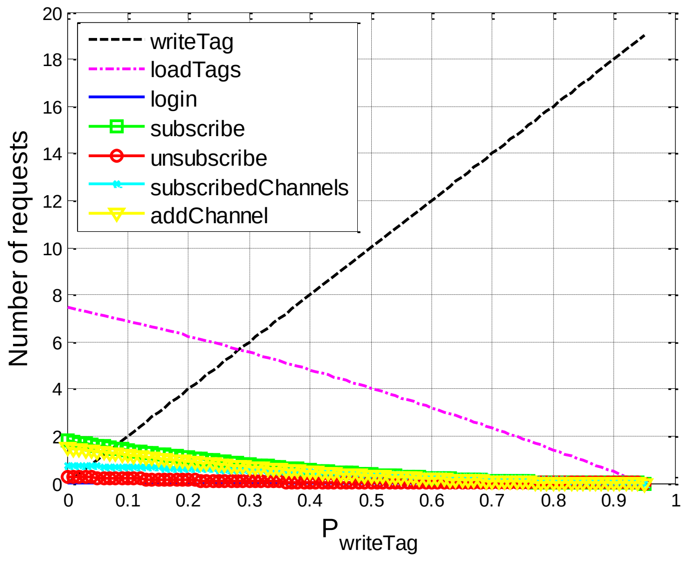

Fig. 1. Number of requests performed to the platform depending on track sending frequency ($P_{exit} = 0.05$)

Fig.1 shows how minimal average number of requests is changing with changing frequency of **WriteTag** request. On this graph can be seen that **WriteTag** is dominating request for $P_{writeTag}$ greater than 0.3 and **LoadTags** is dominating when $P_{writeTag}$ is less then 0.3.

C. Control program profiling

In this work profiling was used for determining is DB interaction bottleneck or not.

Because platform use write-through policy for cache writing request processing time consists request handler code execution time and DB interaction time.

For collecting data about request processing time **PerformanceCounter** class was implemented. This class measures execution time of a code part using **gettimeofday(..)** system call: in object of **PerformanceCounter** constructor current time is measured, in its destructor current time is measured again and difference between two times is the execution time of profiled code part.

Gettimeofday(…) allow to achieve precision about 0.1 ms, and because all our measurements where bigger than this value, we doesn't use more accurate instruments. As an experimental server Lenovo Thinkpad T420 with Ubuntu 11.10 was used. All experiments where performed on clean system.

Next write requests to platform were profiled **AddUser, AddChannel, DeleteUser, RegisterUser, SubscribeChannel, WriteTag, UnsubscribeChannel**; also time of DB interaction was measured for each request. Each request was performed 40 times and using profiling data average total and DB interaction time was calculated.

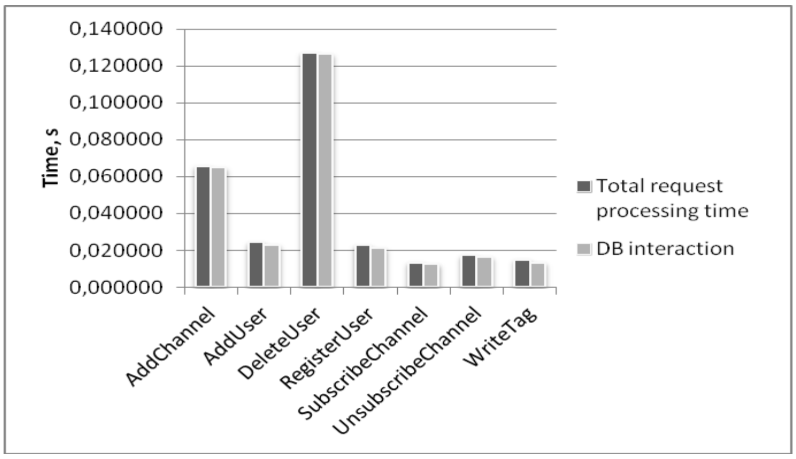

Fig. 2. Write requests profiling results

Fig.2 shows that for each write request profiling shows that DB interaction time is about 99% of total request processing time and that's why DB is a bottleneck. According to this conclusion there are three paths for achieving better performance of requests processing – usage of lazy cache write policy, DB structure optimization and usage of faster DBMS.

D. Performed optimizations

In previous section the most frequent requests were found and discovered that DB interaction is a bottleneck. In this section will be concrete optimizations that were added to Geo2Tag LBS-platform.

1) DB structure optimization: Because **WriteTag** request is the most frequent write-request, its speed affects whole system speed. **WriteTag** performance can be increased by reducing its DB interaction. Before optimization request processing for correct data contains following steps:

- Check of user credentials.
- Check of user permission to write into selected channel.
- Tag creation (SQL INSERT request into **tag** table).
- Tag and channel connection (SQL INSERT request into **tags** table).

- Tag is written into cache.
- REST response generation.

From the list above can be seen that WriteTag processing contains two SQL requests into different tables. For WriteTag speedup next DB structure is proposed:

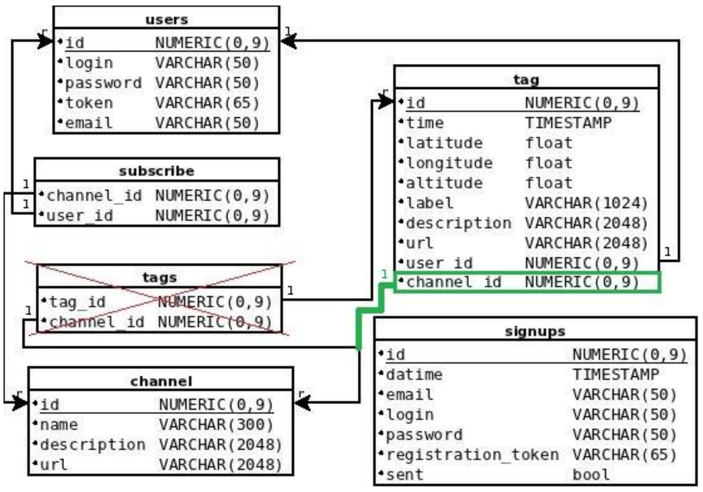

Fig. 3. DB structure optimization

On the Fig.3 DB structure transformation shown. Transformation includes remove of **tags** table and creation additional attribute called **channel_id** in **tag** table; **channel_id** is foreign key from **channel** table.

After such DB transformation **WriteTag** processing will need execution of only one SQL request.

2) Thread-synchronization optimization: In the beginning of the article two main tasks of platform where listed and the second one is a DB synchronization. This task is performed periodically (period of synchronization - **m_updateInterval**) by platform in separate thread and synchronization speed depends linear from DB objects count. That’s why synchronization speed optimization can increase total speed of platform.

During **WriteTag** processing profiling anomaly was found. For more information additional experiment was done. Big number of tags (12000) was added to platform sequentially using **WriteTag** request. Processing time of each request was recorded and below graphical representation of this experiment is presented:

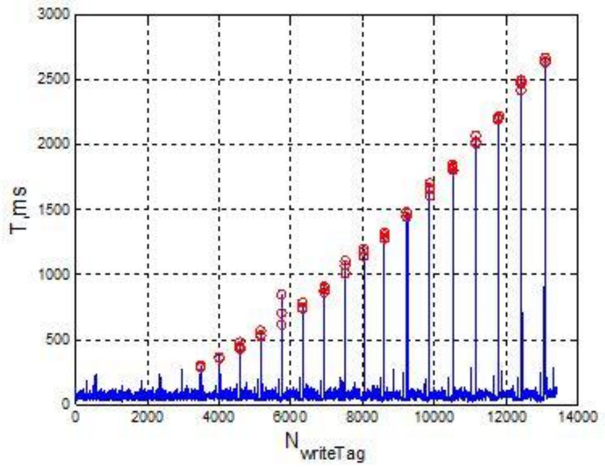

Fig. 4. Dependency between WriteTag processing time and amount of tags in DB

Fig.4 shows anomaly in details – anomaly points are marked as red circles. **WriteTag** processing time can be represented as sum of processes – nonperiodical and periodical (circles). Because platform contains only one periodical process – DB synchronization – with big probability it is a source of problem. Additional profiling shown that this hypothesis was true and problem source is an ineffective usage of low level synchronization primitives. Flowchart below shows problem code part and refactoring:

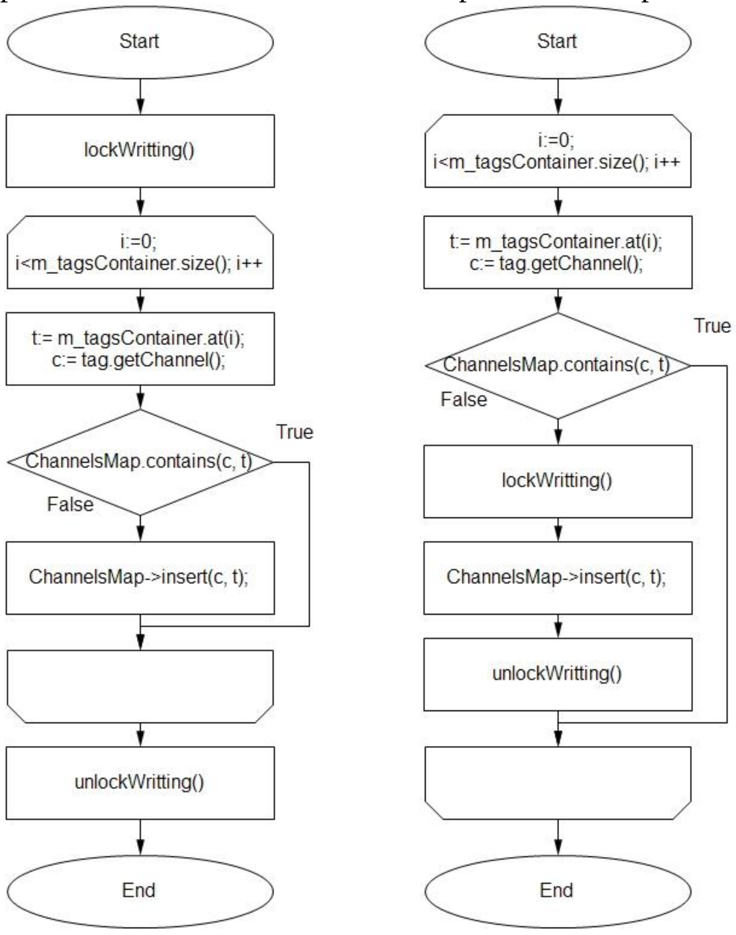
Start
lockWritting()
i:=0;
i<m_tagsContainer.size(); i++
t:= m_tagsContainer.at(i);
c:= tag.getChannel();
ChannelsMap.contains(c, t)
True
False
ChannelsMap->insert(c, t);
unlockWritting()
End
Start
i:=0;
i<m_tagsContainer.size(); i++
t:= m_tagsContainer.at(i);
c:= tag.getChannel();
ChannelsMap.contains(c, t)
True
False
lockWritting()
ChannelsMap->insert(c, t);
unlockWritting()
End

Fig. 5. Tags synchronization code flowchart: before (left) and after refactoring (right)

Refactoring showed on Fig.5. allows to remove anomaly by moving lock to moment when it definitely needed.

Applied refactoring significantly reduced the time of DB synchronization and time of potential cache blocking during this synchronization.

3) DB-synchronization optimization: After tags synchronization refactoring possibilities of platform performance optimizations still exist. Because cache and DB are not compared during synchronization situation when data, which are already in cache, are still copied in cache and CPU is used ineffectively. Solution for this problem is a creation an algorithm for synchronization decision-making.

For decision-making about synchronization cache and DB data comparison needed. Byte comparison is effective but very expensive operation – it has linear dependency from DB size. For the fast decision-making SQL transaction number recording is better approach. Idea is to count each transaction, which platform executed, and compare this number to DBMS statistics.

Solution architecture is next – platform creates separate thread, where decision-making algorithm is executed periodically (time is set by user in milliseconds - **m_updateInterval**). If decision-making algorithm returns true – synchronization is performed, else thread sleeps for **m_updateInterval** until next check.

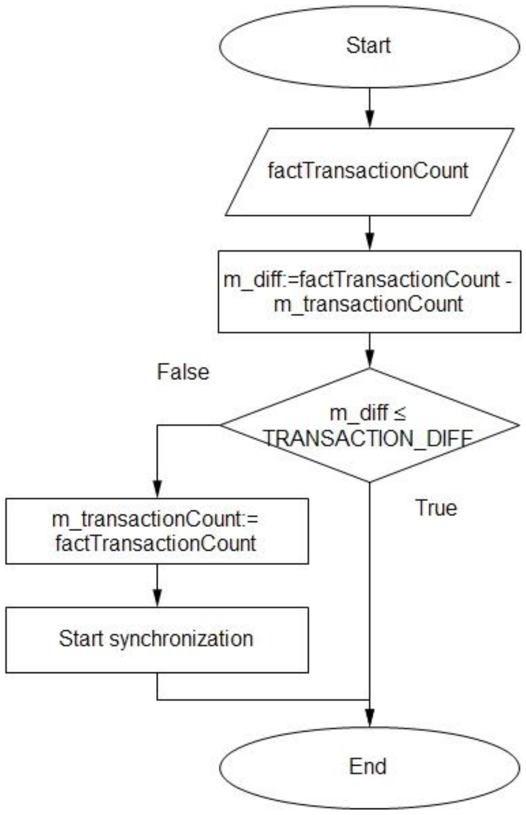
Start
factTransactionCount
m_diff:=factTransactionCount - m_transactionCount
False
m_diff ≤ TRANSACTION_DIFF
True
m_transactionCount:= factTransactionCount
Start synchronization
End

Fig. 6. Algorithm for decision making about DB synchronization.

Algorithm for decision-making works by comparing internal platform transaction counter (**m_transactionCount**) with transactions count received from DBMS statistics table (**factTransactionCount**). If difference between **factTransactionCount** and **m_transactionCount** is more than user-defined value (**TRANSACTION_DIFF**) then synchronization is required and value of actual transaction count should be assigned to internal platform counter (this assignment allows to avoid useless synchronizations in future); else synchronization is not needed (situation when internal counter is more than actual value is possible because statistic collection in PostgreSQL works slow sometimes).

By setting different values of **TRANSACTION_DIFF** and **m_updateInterval** platform can achieve different ratios "performance/data consistency". For example, performance can be increased and sacrifice with data consistency by rare synchronization (big value of **m_updateInterval**) or/and synchronization only when cache and DB has big difference (big value of **TRANSACTION_DIFF**).

E. Effectiveness of optimizations

Performance testing of original and optimized system was performed for getting numerical value of optimization effectiveness. For testing **LoadTags** and **WriteTag** were choosen because they are the most frequently used requests.

For performance measurements simple Qt-based console application was written. Qt was chosen because author is more familiar with QtThreads API than with other multithread API. This application main goal is to perform many requests to locally Geo2Tag instance and log requests results – request processing time and error code (special code for successful request). We measure local instance performance because for distributed instance request processing time will also contain random network delay value. For time measurement we use the same instrument as in profiling section – gettimeofday(…) system call.

1) LoadTags: For **LoadTags** performance testing following experiment was performed:

- Number of tags (N_{db}) in platform were increased sequentially from 0 to 54000 with step of 1000 tags using **WriteTag** requests.
- At every point of N_{db} variable 10000 **LoadTags** requests were send sequentially, time of each request processing was recorded.

By measurements results for each point of N_{db} was recorded sampling distribution of **LoadTags** processing time ($T_{LoadTags}$). For each distribution average, variance, max and min values were calculated. And this values dependency from N_{db} was selected as a representation of request processing performance before and after optimization.

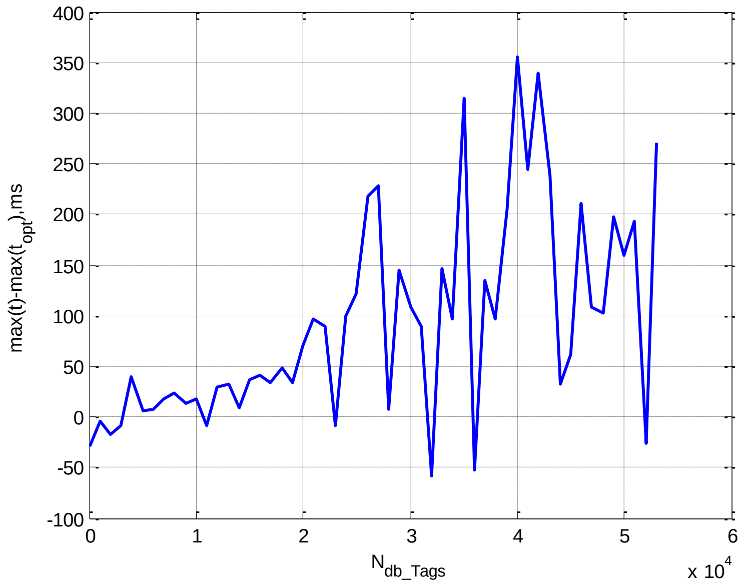

Fig. 7. Difference between optimized and original system max LoadTags processing time

By comparing calculated dependencies at original and optimized systems was found that for **LoadTags** requests processing max time decreased at optimized system; comparison of other dependencies doesn't give any univocal conclusion.

1) WriteTag: For checking optimization effect for **WriteTag** request following experiment was performed on original and optimized systems. Number of tags in platform was sequentially increased by 12000 using send of **WriteTag** request, until required number of tags will not reached. During execution time and error flag (1 if error exists, 0 if request was processed successful) of each request processing was recorded.

Possible reason of errors during **WriteTag** requests processing – ineffective multithread synchronization which was reviewed at sections D.2-3. When platform performs DB synchronization its internal data structures became locked and all requests received in this moment have to wait until unlock. This waiting is probable source of errors.

After the measurements were done both datasets was processed – average time, variance time, max time, number of errors were calculated. Comparison of both systems results are represented in table below:

TABLE I
COMPARISON BETWEN ORIGINAL AND OPTIMIZED SYSTEMS WRITETAG PROCESSING TIME DISTRIBUTION

Parameter	Original system	Optimized system
Average time, ms	72.8558	39.7050
Variance of time, ms^2	11845.0000	16.3266
Max time, ms	2664.0000	255.0000
Number of errors	1426.0000	0.0000
Number of errors per added tag	0.1188	0.0000

This comparison in Table I shows growth of performance and reliability of WriteTag processing after optimization – distribution parameters (average, variance, max) decreased, number of errors became equal zero.

III. CONCLUSION

In this work performance evaluation and optimization for Geo2Tag platform was performed. Math model of client application was created and the most frequent request where founded. Also, request processing bottlenecks and problems with multithread synchronization where found. Future plans contain next steps for further performance improvement:

- Replacement of PostgreSQL by noSQL or GIS-oriented DBMS.
- Usage of lock-free algorithms and data structures.
- Usage of data structures for effective geodata storage.

ACKNOWLEDGMENT

The author would like to thank Finnish Russian University Cooperation in Telecommunication Program for provided equipment and support.

REFERENCES

[1] J. Robertson (2012, May 30). Cisco Web-Traffic Forecast Points to Slowing Growth. Available: http://go.bloomberg.com/tech-blog/2012-05-30-cisco-web-traffic-forecast-points-to-slowing-growth/ (URL)

[2] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", *AFIPS Conference Proceedings* (30), pp.483–48.

[3] I. Bezyazychnyy, K. Krinkin, M. Zaslavskiy, S. Balandin, Y. Koucheravy, "Geo2Tag Implementation for MAEMO" in *the 7th Conference of Open Innovations Framework Program FRUCT*, Saint-Petersburg, Russia 26-30 April 2010.

[4] V. Romanikhin, M. Zaslavsky "Spatial Filters For Geo2tag LBS Platform", in *the 11th Conference of Open Innovations Association FRUCT*, Saint-Petersburg, Russia 23-27 April 2012.

[5] R. Dorohova, S. Kassaye "ThereAndHere Project," in *the 11th Conference of Open Innovations Association FRUCT. Saint-Petersburg*, Russia 23-27 April 2012.

[6] I. Bezyazuchnyy, K.Krinkin "Geo2tag client: Doctor Search," in *the 11th Conference of Open Innovations Association FRUCT*, Saint-Petersburg, Russia 23-27 April 2012.

[8] J. G. Kemeny, J. L. Snell. *Finite Markov chains*. The University Series in Undergraduate Mathematics, Princeton: Van Nostrand, 1960.