Dealing with Continuous Location-Dependent Queries: Just-in-Time Data Refreshment*

Sergio Ilarri[1†] Eduardo Mena[1] Arantza Illarramendi[2]
[1]IIS Department, Univ. of Zaragoza
Maria de Luna 3, 50018 Zaragoza, Spain
{silarri, emena}@posta.unizar.es
[2]LSI Department, Univ. of the Basque Country
Apdo. 649, 20080 Donostia, San Sebastián, Spain
jipileca@si.ehu.es

Abstract

Among mobile computing applications, location-based services are expected to be a big business for wireless operators. These services provide value added by considering the location of the user in order to give him/her more customized information. However, the processing of continuous location-dependent queries is still a subject of research. One of the key issues is how to update the answer presented to the user efficiently as it becomes obsolete very quickly.

The goal of this paper is to analyze how to deal with situations where the answer cannot be updated with the desired frequency. We present several approaches to this problem and evaluate their performance.

1 Introduction

Nowadays, it is growing the interest in designing data services/applications that can be performed efficiently in mobile computing environments. In particular, location-based services are considered one of most important future applications of mobile and wireless systems. A sample location-dependent query (i.e., queries whose answer depends on the location of objects) is "show me the available taxi cabs within three miles of my current location", which could be very useful, for example, for a user looking for an available taxi cab while walking home. Similarly, a taxi cab could ask about nearby people looking for a taxi cab, offering a better service than that of waiting at taxi stops or wandering the streets.

In contrast to the majority of proposed solutions [1, 7], our approach, defined within the ANTARCTICA project [3], deals with contexts where not only the user issuing the query can change his location, but the objects involved in the query can move as well. On the other hand, users can issue queries concerning not only their own location (e.g., a question about near taxi cabs) but the location of any interesting object (e.g., a question about ambulances near a certain *moving* person who needs help). Besides, we consider continuous queries (i.e., queries whose answer is automatically updated by the system) as opposite to instantaneous queries for which a single answer is obtained, since instantaneous queries are not very useful in dynamics environments (the answer to a query can become out-of-date in a matter of seconds).

In [5] we propose a solution to the location-dependent query processing problem in which location information about moving objects is not centralized but distributed across several base stations; each base station manages location information about moving objects under its coverage area. In this approach, mobile agents (programs that execute in contexts denominated *places* and can travel from one place to another [8]; they are very useful in mobile computing [4]) are used to support a distributed query processing, track interesting moving objects, and optimize wireless communication efforts, due to their autonomy and capability to move themselves across computers.

In the architecture of our proposal, agents are organized at four different levels (see Figure 1):

1. An agent in the user device (called *monitor*) is in charge of a query issued by the user; the monitor delegates the query execution and refreshment to a *MonitorTracker* agent which will always execute on the base station (BS) which provides coverage to the user.

2. The MonitorTracker creates a network of *Tracker* agents for tracking the location of objects that are reference of location constraints in the query (*reference objects*). Each Tracker will execute on the BS

*This work was supported by the CICYT project TIC2001-0660 and the DGA project P084/2001.

†This work was supported by the grant B132/2002 of the Aragón Government and the European Social Fund.

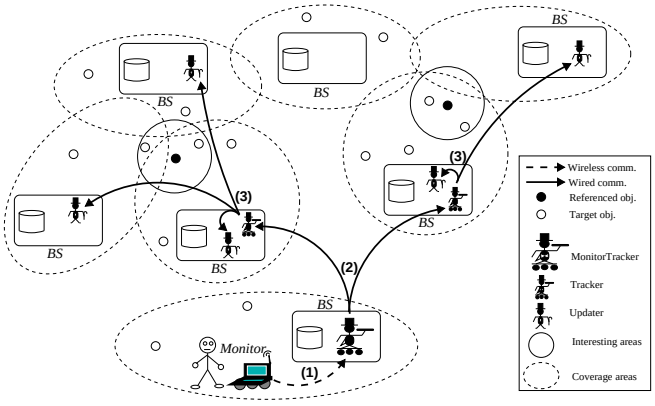

Figure 1: Agent deployment

that provides coverage to its reference object.

3. Each Tracker agent creates a network of (static) *Updater* agents in charge of tracking the location of moving objects that are potential solution of the issued query (*target objects*). To perform this task, Updaters query location tables at base stations.

Each agent in the network correlates the results received from the agents that it created, and returns the results to its creator agent, until the final answer is obtained by the monitor agent. See [5] for more details about our query processing approach.

The rest of the paper is as follows. In Section 2 we briefly describe the main mechanisms that we proposed in [6] to keep location query answers up to date, as we consider that this is important in order to understand the rest of the paper. In Section 3 we explain the problems that arise when correlation and communication delays increase. In Section 4 we focus on the main goal of this paper: how to deal with situations where those delays are so high that the refreshment frequency specified by the user cannot be achieved. In Section 5 we analyze experimentally the performance of the different strategies we consider to deal with that problem. In Section 6 we briefly describe some related work and, finally, conclusions and future work are presented in Section 7.

2 Refreshing the Answer

We briefly explain in this section the technique we described in [6] to refresh the answer in an efficient way. The answer to a location-dependent query could become obsolete whenever any involved object moves to another location. Thus, the answer presented to the user should be refreshed with a certain frequency (that we call *refreshment frequency*). The answer must be refreshed in an efficient manner. Thus, we cannot afford to consider a continuous query as a sequence of instantaneous queries that are re-sent continuously to the data server, as data must be obtained from different places and through wireless networks. It is necessary an approach that assures updated data but optimizing (wireless) communications.

To assure timely refreshments, *every agent in the network will be associated to a certain deadline that indicates the time at which new data must be made available to its creator agent.* The time interval between the deadline of an agent and the instant corresponding to a new refreshment of the user answer (that we call *uncertainty gap*) should be minimized as it will lead to imprecision in the user answer. Thus, the problem of keeping the answer up-to-date can be solved by the network of agents by updating the data of the query processor *right before* a new answer refreshment.

What we consider here are *soft deadlines* instead of *hard deadlines* [10], i.e., last results obtained from an agent are always stored. Whenever a new refreshment must be performed the last results obtained are considered.

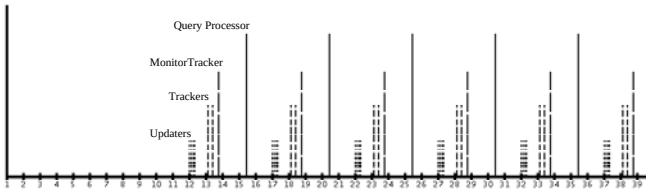

Figure 2: Data refreshment

In Figure 2 we show an example of communications taking place in our location-dependent query processor in order to meet the corresponding deadlines, for a refreshment frequency of five seconds (x-axis represents the seconds elapsed since a query is launched). Each vertical line represents in which moment each agent communicates new data to its creator agent. Notice that it is useless that an agent refreshes its results every second if its creator agent refreshes every five seconds only, since four out of five result communications would be ignored. For example, additional Updater refreshments between seconds 15.5 and 17 will not improve the quality of the answer: the query processor will not refresh the user answer after time=20.5 seconds. On the other hand, you can see how minimizing the uncertainty gap (in the figure, it is about 3.5 seconds; e.g., from 12 to 15.5 seconds) is limited by wireless communications (e.g., from 13.9 to 15.5).

Our proposal relies on agents that behave responsible (they obey their deadlines) and do their tasks as late as possible in order to return the most recent snapshot of moving objects that they can obtain. As result of all the agents in the network behaving in this way, an emergent property arises in the system: the answer presented to the user is based on the latest data that can be retrieved. In order to understand the problems (and their solution) described in Sections 3 and 4, we briefly

explain in the following the technique proposed in [6] to calculate deadlines.

Obtaining Agent Deadlines

Each agent x at layer i (created by an agent z at layer $i-1$) estimates the deadline of its underlying agents at layer *i+1* by subtracting from its own deadline the time spent in its correlation task and its communication with its upper layer ($\forall$ *agent y created by agent x*):

$$absDeadline_{y@i+1} = absDeadline_{x@i} - correlDelay_{x@i} - commDelay_{z@i-1}^{x@i}$$

where $absDeadline_{a@j}$ denotes the absolute deadline of an agent a at layer j (time instant in which the results of agent a must be available to its creator agent at layer $j-1$), $correlDelay_{a@j}$ denotes the estimated time spent by the agent a at layer j in correlating the results received from its underlying agents, and $commDelay_{b@k}^{a@j}$ denotes the estimated communication delay from an agent a at layer j to agent b at layer k. Estimations are based on past executions. Notice that agents at the same level can have different delays (they execute on different computers, communicate with different agents, handle different data sizes, etc.).

In Figure 3 we show how agents interact across layers. We focus on an agent at layer i, $Agent_i$, created by a certain $Agent_{i-1}$ at layer $i-1$ (in our prototype, the query processor creates the MonitorTracker agent, which creates as many Tracker agents as necessary, which create as many Updater agents as needed). We explain the main steps to consider in order to manage deadlines: (1) $Agent_{i-1}$ communicates to $Agent_i$ the deadline for layer *i*; (2) $Agent_i$ receives its deadline and calculates the deadline of agents at layer *i+1*, by considering its own deadline and its estimated correlation and communication delays; (3) $Agent_i$ communicates to agents at layer *i+1* their deadline; (4) $Agent_i$ waits until the deadline of layer *i+1* comes (to meet its own deadline, $Agent_i$ has to begin correlation right after data from layer $i+1$ is obtained); (5) each $Agent_{i+1}$ sends its results to $Agent_i$; (6) $Agent_i$ correlates the received results from agents at layer *i+1*, and (7) $Agent_i$ sends its results to $Agent_{i-1}$. For agents at the lowest layer (Updaters in our architecture), there is not really a correlation task but an access to a location database.

However, as agents could execute at different BSs, dealing with absolute deadlines would require to keep the computer clocks of base stations synchronized with the (wireless) user device internal clock, as just a slight clock shift will make some agents to miss their deadlines. Consequently, deadlines communicated among layers must be relative (e.g., "I want your data in ten seconds") rather than absolute (e.g., "I want your data ready at 13:05:20").

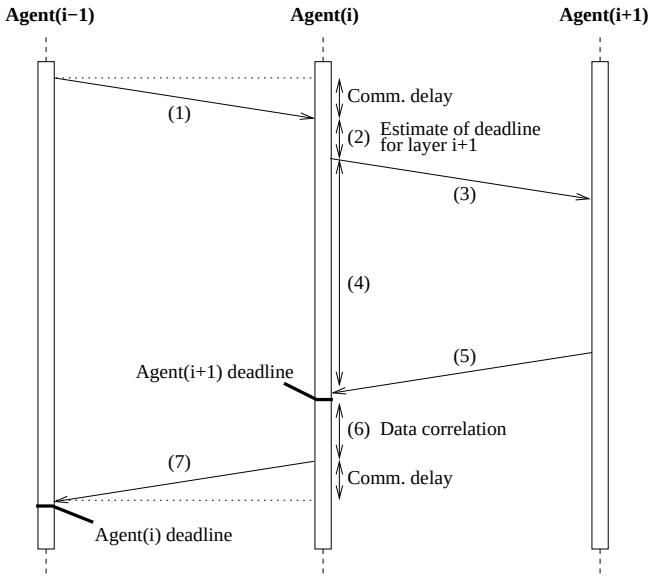

Figure 3: Interaction among agents at different layers

Therefore, an agent x at layer i calculates the relative deadline of its agents at layer $i+1$ by subtracting from the absolute deadline the current time, the time spent in communicating the relative deadline, and the time spent during this calculation ($\forall$ *agent y created by agent x*):

$$relDeadline_{y@i+1} = absDeadline_{y@i+1}^{x@i} - t_{now}^{x@y} - commDelay_{y@i+1}^{x@i} - estimateDelay_{x@i}$$

where $relDeadline_{y@i+1}$ denotes the time remaining before the next absolute deadline of agent y at layer $i+1$, $absDeadline_{b@j+1}^{a@j}$ denotes the absolute deadline of the agent b at layer $j+1$ according to the computer clock of the machine where agent a at layer j resides, $t_{now}^{a@j}$ denotes the current time instant according to the computer clock of the machine where agent a at layer j resides, and $estimateDelay_{a@j}$ denotes the time the agent a at layer j needs to estimate the deadline of its agents at layer $j+1$.

In this way, agents at layer $i+1$ can obtain their own absolute deadline and be synchronized with their creator agent ($\forall$ *agent y created by agent x*):

$$absDeadline_{y@i+1}^{y@i+1} = t_{now}^{y@i+1} + relDeadline_{y@i+1}$$

Notice that $absDeadline_{y@i+1}^{y@i+1}$ and $absDeadline_{y@i+1}^{x@i}$, despite storing different values (as they refer to different computer clocks), represent the same time instant.

An agent y at layer $i+1$ obtains the time that it should wait before correlating and communicating its results (that we call $spareTime_{y@i+1}$) by subtracting its result communication delay and correlation delay from its absolute deadline[1], in order to make its results available

[1] In our implementation, we also subtract 0.25 seconds to avoid that minimum unexpected delays make agents miss their deadlines.

to its creator agent x in time (before $absDeadline_{y@i+1}^{y@i+1}$) but as late as possible (to provide data as recent as possible to the upper level):

$$spareTime_{y@i+1} = absDeadline_{y@i+1}^{y@i+1} - correlDelay_{y@i+1} - commDelay_{x@i}^{y@i+1}$$

In our proposal agents are deployed at four different layers (as mentioned in Section 1). However, the above technique could be used to synchronize communications among agents in other architectures with more or less layers.

3 Consequences of Dynamic Delays

As explained before, the estimation of deadlines is based on parameters that change over time: the communication and correlation delays. So, correlation tasks may become more or less time expensive, as data obtained from the underlying level will change (data obtained by each agent depend on the current location of moving objects in the scenario) and the computer load can vary. Similarly, communication delays among agents will change due to many reasons (number of concurrent user requests, network failures, the amount of data sent, etc.). Besides, both parameters will change any time an agent moves to a new BS (agents move from BS to BS in order to track interesting moving objects).

Thus, after communicating results to the upper layer, each agent re-calculates the deadline of its lower layer taking into consideration the new delay estimations. Whenever a significant change happens, the new deadline will be communicated to agents at the lower layer in order to adapt their behavior to the current situation. The agent whose deadline is adjusted could also consider to communicate new deadlines to its own lower layer.

We show a real example obtained using our prototype when deadlines are not adjusted (see Figure 4) and when deadlines are adjusted according to the technique we propose (see Figure 5). We consider a refreshment period of five seconds and, for simplicity, only one agent of each type is involved in the location query. In such figures, we can see the deadlines of each agent (short lines labeled with time); solid arrows indicate communications of results and dashed arrows represent communications of new deadlines. For readability, we show time instants in format mm:ss.

At time instant T1 the communication delay between the Tracker and MonitorTracker gets higher and therefore the Tracker agent spends more time than expected and misses its deadline. If deadlines of Updaters are not adjusted (see Figure 4), the Updater agent is always late since time instant T2. Notice that, although the Tracker agent meets its deadlines, it is using data provided by the Updater agent a long time ago; for instance,

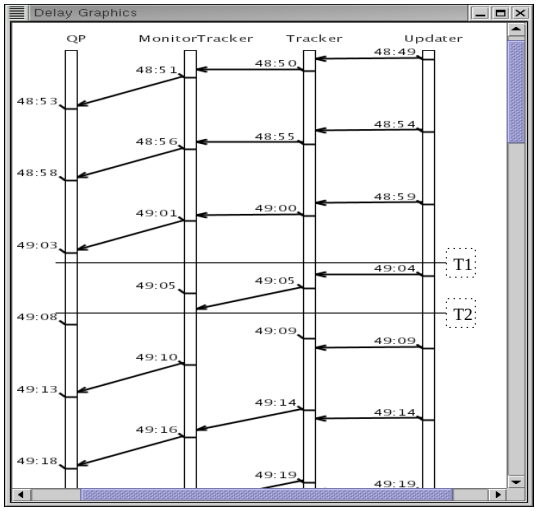

Figure 4: Not adjusting deadlines

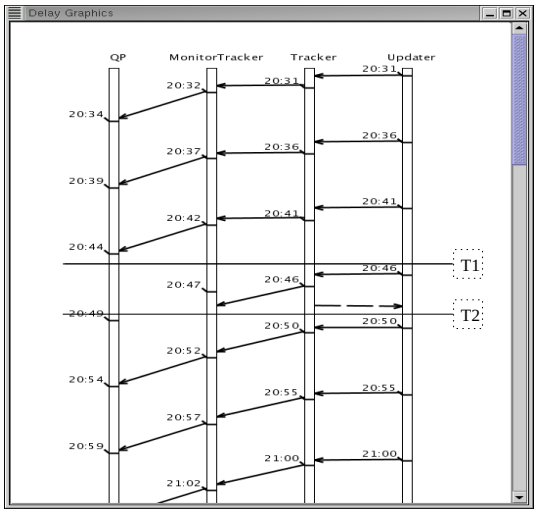

Figure 5: Adjusting deadlines

at 49:14 the Tracker is using data obtained five seconds ago. However, when deadlines are adjusted after detecting a change in delays (see Figure 5), after time instant T2 the Tracker has communicated the new deadline to the Updater and therefore they work as synchronized as before time instant T1.

In Figure 6, we show the difference in the answer quality (the location error) when monitoring every five seconds the location of an object moving at five distance units per second, both in the case of adjusting and not adjusting deadlines. Minimum errors are obtained when a refreshment takes place and maximum error right before a refreshment. Notice how the error is increased when not adjusting deadlines after communication delays get higher (in Figure 6, around time instant 15, where both strategies miss one deadline).

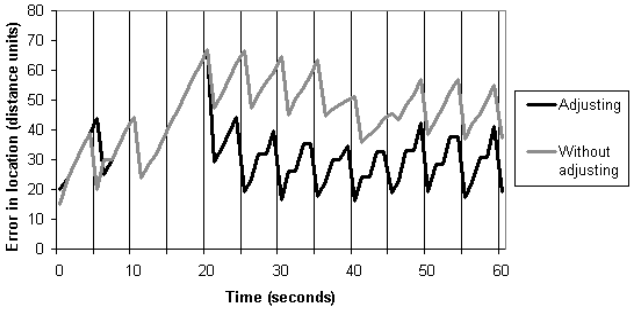

Figure 6: Answer quality when adjusting/not adjusting deadlines.

4 Dealing with Too High Refreshment Frequency

In Section 3 we have explained how to deal with delay increases that can be solved by anticipating agent deadlines. In this section we explain how to deal with situations in which some agents involved in the query processing cannot return new data to their creator agent at the requested frequency. This situation will arise when an agent does not have enough time to perform its correlation and communication tasks between two refreshments, considering the requested refreshment frequency; notice that any other situation is fixed by the technique explained in Section 3. The maximum frequency that an agent x at layer i can support to return *new* data is the following (where agent x is created by agent z, and agents y are created by agent x):

$$TopFreq_{x@i} = \min\left\{ \frac{1}{correlDelay_{x@i} + commDelay_{z@i-1}^{x@i}}, \max\{TopFreq_{y@i+1}\} \right\}$$

where $\{TopFreq_{y@i+1}\}$ denotes the set of maximum frequencies supported by each agent y at layer $i+1$.

Notice that it is necessary that at least one underlying agent supports the required frequency; otherwise, agent x will not be able to give *new data* at such a frequency. If underlying agents refresh data at different frequencies (they are desynchronized) then data correlated by agent x will be a mixture of data obtained at different moments, and this can lead to a loss of quality in the data presented to the user: it could be a snapshot that did not happen in the past. The most desirable situation would be to provide data at a frequency as high as possible with the minimum desynchronization. We present in the following several approaches to deal with this problem.

4.1 First Approach: Synchronization with the Slowest Agent

As some agent cannot support the required frequency, we will have agents communicating results at different rates, leading to an expected harmful desynchronization in the network. The obvious attempt to solve this problem is that all the agents adopt the frequency of the slowest agent. As several agents could become slower at the same time, agents communicate to their creator agents the highest frequency they support (keeping themselves synchronized with their children) at each refreshment (along with the data). This process is repeated recursively until the MonitorTracker obtains which is the lowest frequency of the whole agent network. Then, the MonitorTracker adjusts all the agents (including itself) to that frequency to keep everybody synchronized.

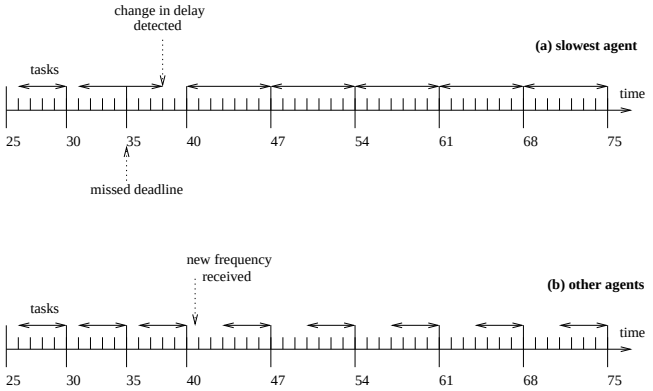

Figure 7: Synchronization with the slowest agent

We show in Figure 7 the behavior of agents when we apply this strategy to a scenario with refreshment period of five seconds; "tasks" means correlation and communication tasks, deadlines are labeled with time annotations and an increase in delays happens at time instant 31. The slowest agent (see Figure 7.a) misses one deadline only, and then all of the agents change to a lower frequency (the period changes from five to seven). So everybody is able to meet their deadline, although the data presented to the user are updated at a lower frequency.

This solution has a clear disadvantage, since a single agent with a poor refreshment frequency will lead to a low answer refreshment frequency.

4.2 Second Approach: To Ignore the Slowest Agent

The simplest solution we can think of is just to ignore that one of the agents cannot support the required frequency. We allow that agent to run desynchronizely with respect to the rest of agents, although it will miss some of its deadlines.

In Figure 8, we show an example where the required refreshment frequency is five seconds. Due to an increase of four seconds in its delays, the agent loses its deadline at instant 35, it is able to meet the deadline at 40, and it realizes that it must begin immediately the new correlation task to meet the deadline at 45. After meeting such a deadline it realizes that cannot meet the next deadline so begin the correlation task at 48 to meet the deadline at 55. Therefore, the agent loses one out of two deadlines after the delays have increased.

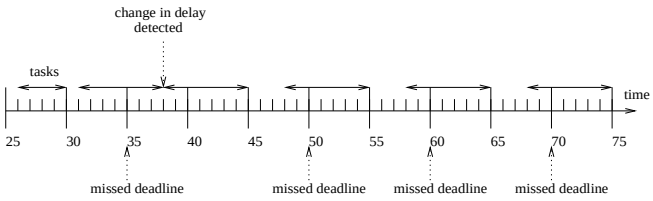

Figure 8: Ignoring the slowest agent

Notice that although the agent tries to keep a frequency of five seconds, it is not able to do that, and finally it refreshes data every ten seconds. The agent misses its next deadline due to the fact that it begins its tasks at the latest moment (in order to synchronize itself with its creator agent).

4.3 Third Approach: No Spare Time

As conclusion of the previous subsection, it seems that it makes no sense to allow spare time when the agent cannot support the required frequency. So, after detecting that the new delays are greater than the refreshment period, the agent could start the tasks for its next deadline immediately, as shown in Figure 9.

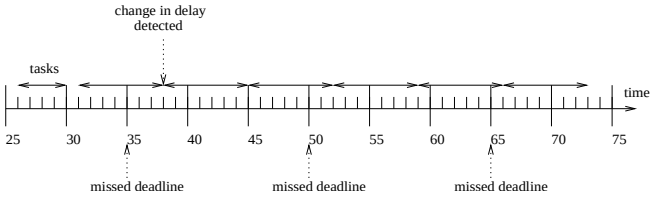

Figure 9: No spare time

Notice that less deadlines are missed comparing to Figure 8 (a deadline is missed when tasks are not finished since the previous deadline). With this solution, the agent will be desynchronized with respect to its creator agent, so the uncertainty gap will not be minimized and it will change over time.

4.4 Fourth Approach: Multithreading

In the three previous approaches, we assumed that an agent could not begin a new correlation task until finishing the communication of the results obtained in the last correlation. This led slow agents to miss many deadlines or to lower their refreshment frequencies.

However, in a multithreading approach we can remove this relevant constraint: an agent will be able to perform several correlation (or communication) tasks *at the same time* whenever the delays become greater than the requested refreshment period. For example, if we have to retrieve new data from a (continually changing) database every five seconds, and each access spends seven seconds, the idea is to query the database every five seconds to meet the requested frequency (data will be returned every five seconds although each access takes seven seconds, as we said before). In this example we will have two concurrent queries.

We show an example in Figure 10. In such a figure, the agent misses its deadline at instant 35 due to the increase of delays at time instant 31. At time instant 43 a new thread is spawned to attend the deadline at 50. Since then, all the deadlines are met thanks to the use of a second thread.

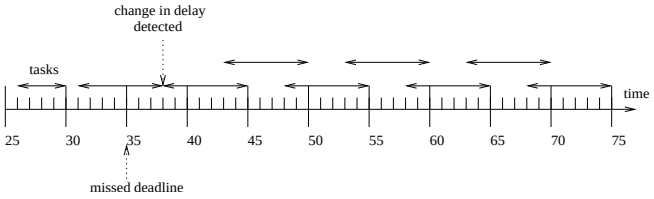

Figure 10: Multithread Solution

So, when delays increase, as new threads as necessary can be spawned to anticipate the correlation tasks properly. We leave as future work how to detect a situation in which the delays become so high (regarding the refreshment period) that the necessary threads overload the machine, and therefore they also increase the delays. In such a case we could use one of the other approaches described in this section.

5 Experiments

In this section, we analyze how the four approaches described in Section 4 impact on the answer presented to the user. Regarding the answer quality, we distinguish two types of error:

- *Absolute location error*: for each target object (objects that the user is interested in) in the scenario, it is the difference in distance between its real location and the location presented to the user.
- *Relative location error*: for each target object in the scenario, it is obtained by subtracting the real distance between the target object and the reference object (the object that is referenced in the query) and the distance between the target object and the

reference object as they are shown to the user (the sign of this difference it is not significant). The greater this value, the greater the probability of error in terms of missing objects (i.e., objects that should be in the answer query but they are not) and/or extra objects (i.e., objects that should not be in the answer query but they are) when the user is querying about target objects in a certain area around the reference object.

We cannot know *a priori* which type of error is more important for the user. Thus, minimizing the absolute location error should be the main goal when the user is interested only in the location of some moving objects, while minimizing the relative location error would be the most desirable situation if the user is interested in *which* objects satisfy certain conditions (e.g., the airplanes inside a radius of 10 miles around another airplane) but the exact location of those objects is not very important.

Desynchronizations among agents will lead, as we explained before, to mixing data of different ages, what could increase the relative location error since inconsistent snapshots of the moving objects' locations could be obtained. On the other hand, lower frequencies imply higher absolute location errors. Consequently, there is a trade-off between maximizing the refreshment frequency and minimizing the desynchronization: 1) increasing the refreshment frequency could increase the desynchronization as some agents could not support such a high frequency, and 2) the only way to decrease the desynchronization could be to decrease the frequency in order to allow the slowest agents (the cause of desynchronization) to meet their deadlines.

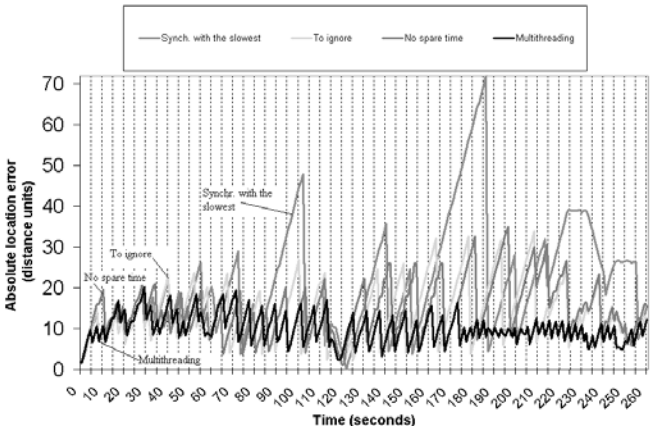

Figure 11: Answer quality: absolute location error

To analyze the impact of each of the alternatives, we use a sample query which retrieves airplanes (the target objects) located around a given airplane (the reference object). We consider airplanes moving freely at two distance units per second and we want the answer updated every three seconds. In this scenario there exist two base stations: *BS1* and *BS2*. In order to simulate overload, correlation delays at BS1 will increase in one second every 10 seconds, starting with one second.

In Figure 11 we show the averaged absolute location error committed by the query processor at each time instant. Absolute location errors are continuously increasing until a new refreshment takes place (error is reduced at each refreshment). At the beginning, the delays can be supported at the required refreshment frequency and therefore all the approaches get similar results. As the correlation delay increases, we can see that the multithreading approach gets the best results (it adapts itself to higher delays by launching new threads), followed by the no spare time approach (agents become more and more slow, although they perform tasks continuously), then the approach of ignoring the slowest agent (agents miss more deadlines to avoid desynchronization), and finally the worst approach is the strategy of adjusting to the slowest agent (to keep agents synchronized, the frequency is lowered, which makes absolute error increases a lot). We can observe how the errors committed by the multithreading approach do not increase due to increases in the delays, as more threads are spawned when needed to support the required frequency without desynchronization. The approach of adjusting to the slowest agent is especially bad in cases like this where delays continuously increase over time; if delays increase while agents are adjusting their frequencies (to fix the previous delay increase), then more deadlines can be missed: this is what happens inside intervals 85-105 and 157-192 in Figure 11 (shown as high peaks).

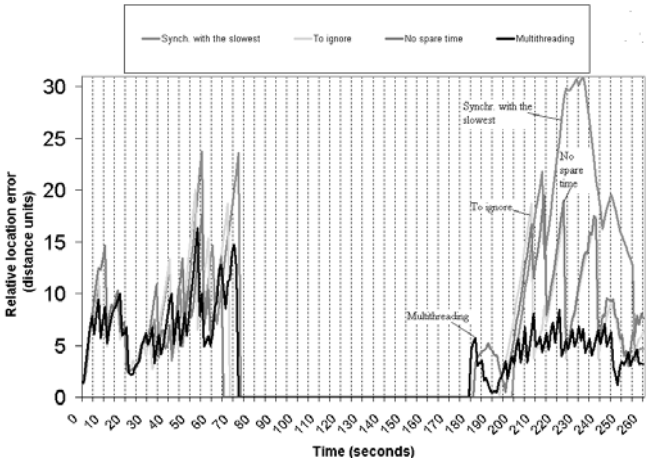

Figure 12: Answer quality: relative location error

In Figure 12 we show the relative location error at each time instant. Between instants 80 and 185, there are no airplanes in the requested area around the airplane that is the reference object for the query, which explains why there is not a relative location error in that interval. As we can observe in the figure, multithreading is clearly the best approach followed by no spare time and ignore the

slowest agent. Synchronization with the slowest agent is the worst approach, especially for high delays. Notice that the y-axis scale in this figure is different from that of Figure 11.

6 Related Work

As far as we know, no other work uses an architecture based on mobile agents to process continuous location-dependent queries in a wireless distributed environment of moving objects. Therefore, we have not found any alternative solution to solve the problems described in this paper.

The closest work to our project of location-dependent queries is the DOMINO project [11, 9]. They focus on the problem of how to store location information about moving objects in an database with the goal of processing spatial and temporal queries in an efficient way. In contrast, we are interested in using that information to process location-dependent queries involving location information stored in different databases (at several base stations), which scales up better when the number of moving objects increases. Thus, both works are complementary.

Some works dealing with processing continuous location-dependent queries in mobile environments are [12, 2, 9]. These works' main concern is when to transmit results to a mobile host in order to minimize communications while providing an up-to-date answer to the user. However, they are not well-adapted to deal with continuous queries which ask for locations of moving objects (such as monitoring a truck fleet), and they assume that expected trajectories are known by the query processor.

7 Conclusions and Future Work

We have proposed four possible solutions (with different advantages and disadvantages) to deal with changes in communication and correlation delays that lead to a decrease in the maximum supported refreshment frequency. We have shown a comparison where the multithreading approach provides the best results, although the others can be considered in some situations.

As future work, we plan to identify which delay changes are significant enough to trigger a synchronization mechanism. Also, we are studying how to detect and deal with situations where the multithreading approach decreases the performance.

References

[1] Vicinity Corporation. Vicinity Wireless Locator. http://home.vicinity.com/us/wireless.htm.

[2] Hüseyin Gökmen Gök and Özgür Ulusoy. Transmission of continuous query results in mobile computing systems. *Information Sciences*, 125(1-4):37–63, 2000.

[3] A. Goñi, A. Illarramendi, E. Mena, Y. Villate, and J. Rodriguez. ANTARCTICA: A multiagent system for internet data services in a wireless computing framework. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems, Scottsdale, Arizona (USA)*. Lecture Notes in Computer Science (LNCS 2538), October 2001.

[4] Robert S. Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents for mobile computing. Technical Report TR96-285, 1996.

[5] S. Ilarri, E. Mena, and A. Illarramendi. A system based on mobile agents for tracking objects in a location-dependent query processing environment. In *Twelfth International Workshop on Database and Expert Systems Applications (DEXA'2001), Fourth International Workshop Mobility in Databases and Distributed Systems (MDSS'2001), Munich (Germany)*, pages 577–581. IEEE Computer Society, ISBN 0-7695-1230-5, September 2001.

[6] S. Ilarri, E. Mena, and A. Illarramendi. Monitoring continuous location queries using mobile agents. In *Sixth East-European Conference on Advances in Databases and Information Systems (ADBIS'2002), Bratislava (Slovakia)*. Springer Verlag LNCS, September 2002.

[7] Vindigo Inc. Vindigo: location-specific content delivered to mobile devices. http://www.vindigo.com.

[8] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF, the OMG mobile agent system interoperability facility. In *Proceedings of Mobile Agents '98*, September 1998.

[9] A. Prasad Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proceedings of the Thirteenth International Conference on Data Engineering (ICDE'97), Birmingham U.K., IEEE Computer Society , ISBN 0-8186-7807-0*, pages 422–432, April 1997.

[10] Régis Vincent, Bryan Horling, Victor Lesser, and Thomas Wagner. Implementing soft real-time agent control. In Jörg P. Müller, Elisabeth Andre, Sandip Sen, and Claude Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 355–362, Montreal, Canada, 2001. ACM Press.

[11] O. Wolfson, A. Prasad Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *invited paper, special issue of the Distributed and Parallel Databases Journal on Mobile Data Management and Applications*, 7(3):257–287, 1999.

[12] Kam yiu Lam, Özgür Ulusoy, Tony S. H. Lee, Edward Chan, and Guohui Li. An efficient method for generating location updates for processing of location-dependent continuous queries. In *Database Systems for Advanced Applications*, pages 218–225, 2001.