Web Architectures for Scalable Moving Object Servers

Cédric du Mouza
CNAM, Paris, France
dumouza@cnam.fr

Philippe Rigaux
LRI, Univ. Paris-Sud Orsay, France
rigaux@lri.fr

ABSTRACT

The paper describes how the Web can be used as a support for intensive querying and display of large moving objects databases. We present first an architecture for a system which integrates incoming *events* provided by GPS servers in a spatio-temporal database, and permits to register *queries* over this database. We focus on *continuous* queries that allow users to receive notification of events affecting the initial result, and discuss the process of *matching* queries and events in order to perform this notification. Finally a prototype implementing a simplified version of this architecture is described. It uses standard XML-based languages in order to visualize the result of a query as dynamic maps where the motions can be tracked in real time.

General Terms

Performance

Categories and Subject Descriptors

D2 [**Software**]: Domain-specific architectures

1. INTRODUCTION

The focus of research in Geographic Information Systems (GIS) has strongly evolved, in the past few years, from traditional aspects of data management (modeling, indexing, querying) to novel and exciting challenges raised by the emergence of new technologies. Two of the major recent achievements of these technologies, namely the World Wide Web and the development of accurate positionning systems, have a strong impact on GIS.

The Web encourages exchange and integration of data. These features create new opportunities for the distributed and cooperative processing of geographic data, promoted by several institutions, grouped in the Open GIS Consortium (OGC) [7]. Positioning systems constitute another challenging area. The Global Positioning System (GPS) and the European Galileo project (its launching has been decided very recently, at the end of March 2002), are able to determine the position of an object with a very high precision (a few centimeters). Cars, mobiles phones, boats, planes can be coupled with localisation devices. This will generate, in a near future, the development of new applications for traffic monitoring, assistance, and the management of mobile services.

In this paper we examine architectural issues for *Moving Object Servers* (MOS) providing the following Web-based services: (i) *integration* of information provided by GPS or Gallileo servers related to the trajectories of (possibly large) sets of moving objects and (ii) *notification* of this information to several actors connected to the server. The system handles therefore two types of information, transmitted via the network:

- **Events** are the new locations of moving objects continuously provided by GPS or Gallileo server to the MOS.
- **Queries** are notification requests registered by an actor in the MOS for a given period, in order to monitor the activities of moving objects.

Typical examples of queries are "notify me whenever a truck enters this area in the next 2 days" or "show me all the planes around the airport during the next 2 hours". Such queries, often called "continuous" [18, 5], are peculiar. They do not correspond to fixed results, but rather involve a periodic maintenance of the answer to take into account the movements of objects. We denote as *matching* the process of determining, given an event, the set of queries whose result might be affected. The performance of the system, and in particular its ability to serve a lot of queries under an intensive incoming of events, depends on the efficiency of the matching process.

These functionalities are close to web data-intensive applications relying on a subscribe/publish mechanism, as decribed for instance in [6, 9]. However, moving objects tracking raises some specific problems not found in traditional e-commerce applications. First an object might belong to the result of a query at time t and not at time $t + 1$, (and conversely) because of its time-varying location. In other words, updates of the database are not the only events that trigger an actualisation of a result, and thus require a notification to the client. We propose in this paper some simple data structures and algorithms supporting matching operations in the specific context of moving objects servers, and providing the scalability expected in a world-open application.

Second we need to integrate in the system a visualization mechanism tighly coupled with the server and to take into account notifications in real time. In the last part of this paper we describe a prototype, implementing a simplified version of our architecture, which provides both communication, visualization and animation of data with SVG, an XML-based language dedicated to graphical display (including the animation on the map).

The rest of the paper is organized as follows. In Section 2 we outline the architecture and define more precisely the concepts of

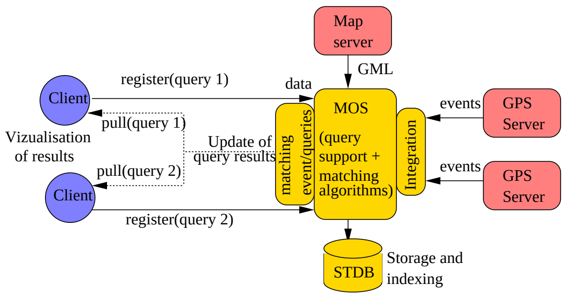

Figure 1: Architecture for Moving Object Server

events, queries and query results. The structures and operations supporting the matching of queries and events are described in Section 3. Section 4 is devoted to the prototype and Section 5 to related work.

2. CONTINUOUS QUERIES

The global architecture is depicted in Fig. 1. It consists of a central part, the Moving Object Server (MOS) which sends queries to a *spatio-temporal database* (STDB), receives events from *data sources* and processes *queries* registered by Web clients. We first define the role of these modules, before focusing on the process of matching queries with events.

Architecture

The spatio-temporal database (STDB) stores a set of moving objects and a network which can be used as a layer supporting the object trajectories. We need only some simple and quite common assumptions on the representation of data and on the querying features of the STDB, summarized below.

A *trajectory* is a piecewise linear function which can be represented as a set of pairwise connected segments in a 3-dimensional space, with constant speed on each segment. It can be viewed as an infinite set of points in $\mathbb{R}^3$ whose finite representation is obtained from a set P of sample positions $P_i(t,x,y)$. The sample points are provided by the GPS servers as described below.

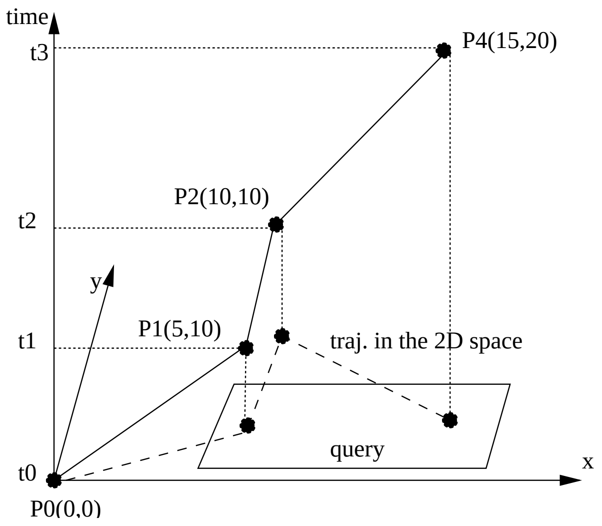

Figure 2: A trajectory

The representation of a trajectory used in the following is a list $< [I_1, f_1^x, f_1^y], \ldots [I_n, f_n^x, f_n^y] >$ where (i) each I_i is a time interval, (ii) the intervals $\{I_i, i \in [1,n]\}$ are pairwise distinct and (iii) each f_i^x (resp. f_i^y) is a linear function defined only on I_i, from time t to abcissa x (resp. ordinate y). The position of an object at any time instant t in I_i is a point $(f_i^x(t), f_i^y(t))$, denoted $loc_i(t)$.

Note that we do not require the time interval to form a partition of the lifespan of an object. This offers more flexibility to represent the result of a query which might consist of non-adjacent intervals. In fig. 2 for instance, the object belongs to the result of the window query at two distinct time intervals.

We denote by $\mathcal{O}$ the set of moving objects and by $\mathcal{T}$ the set of all trajectories. A *moving object relation* (MOR) R is a pair $(\mathcal{O}_R, traj_R)$ where $\mathcal{O}_R$ is a finite subset of $\mathcal{O}$ and $traj_R$ a mapping from $\mathcal{O}_R$ to $\mathcal{T}$.

Data sources consist either in *Map servers* or in *GPS servers*. Map servers provide mainly *networks*. GPS servers constitute the second type of data source. Each of these servers collects information by inspecting continuously a set of moving objects, and transmits *events* to the MOS. We assume the following definition for events:

Definition 1. An *event* is a tuple $< id, t, loc, speed, direction >$ where id denotes the identifier of a moving object o, t the instant when o was inspected, and loc, $speed$ and $direction$ denote respectively the location, speed and direction of o at time t.

An *integration module* receives the events from one or several servers. Its role is to associate, given the location and direction components, the object whose identification is id with a path of the underlying network. This defines a new piece of the object's trajectory which is then inserted in the spatio-temporal database.

Let us turn now to *queries*. There exists several propositions for a spatio-temporal query language handling moving objects, includind the MOST model [18], spatio-temporal data types [12, 10] and the constraint-based approach of [11]. We restrict our attention in this paper to *window queries* of the form $q(rect, I)$ where $rect$ is a rectangle on the map, and I a time interval starting at I_{min}, the time instant when the query is registered, and ending at I_{max}.

Definition 2. Let $M(\mathcal{O}_M, traj_M)$ be a moving object relation in the database. The *result set* of a query $q(rect, I)$ over M is a moving object relation $RES(\mathcal{O}_q, traj_q)$ such that (i) $traj_q$ is the mapping defined as $traj_q(o) = traj_M(o) \cap [rect \times I]$ and (ii) $\mathcal{O}_q$ is the set of objects o in $\mathcal{O}_M$ such that $traj_q(o) \neq \emptyset$

The trajectory of an object in the result set of a query may be split in several, non-connected segments, with non-adjacent time intervals (for instance an object quits the query rectangle and comes back after a while). The fact that an object enters or leaves a query rectangle relates to its movement and is therefore partially independent of events received from the GPS servers. Thus the MOS should watch for such occurrences in order to notify clients.

Notification strategy

There exists two traditional models of notification: *pull()*, where the client explicitely asks the server for new events, and *push()* where the server sends all new events to clients as soon as they are received. We consider here the $pull()$ model, a natural choice in the case of web client/server communication where the client tends to intepret data pushed by the server as new documents replacing the current one, a behavior which is not convenient for an incremental construction. Fig. 1 shows clients sending *pull()* requests to the MOS. These requests are handled by the matching module whose task is to inform queries of the updates affecting their result

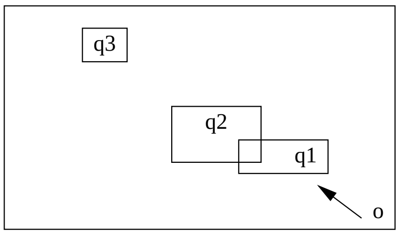

Figure 3: Matching a trajectory with a set of queries

set that occured since the previous *pull()* call (denoted the Δ-result in the following) . In summary, here is the functional architecture of the server, and the main functions it handles.

1. *AddQuery (q (rect, I))* is the function that registers a new query. We assume that q contains all the information needed to communicate and identify a query.
2. *AddEvent(evt)* is the function that inserts a new event in the server.
3. *pull(q)* is the function that returns the Δ-result for a query q.

Computation of result sets

A first solution to update the result set when a *pull()* is issued is to apply a *replace* policy. In that case the query resulset is computed periodically, and sent to the client in order to replace the previous one. This might be considered in situations where the objects move so fast that the result set is subject to drastic changes in very short periods.

In this paper we assume that the objects speed is small with respect to the query's size, and thus that the result set changes slowly. We follow therefore a quite different approach, based on an incremental computation of the result sets, described below. When the query $q(rect, I)$ is first registered (i.e., at time $t = I_{min}$), an initial result set res_0^q is created with all the objects whose position lie in $rect$ at $t = I_{min}$. With each object is associated a path on the network (determined after the integration process described above) where the object is assumed to travel at constant speed. This initial result is transmitted to the client. It must be updated when one of the following conditions is met: (i) an event received by the MOS tells that an object has moved on another path or (ii) an object quits or enters the query rectangle.

It turns out that it is not sufficient to wait for an event, check, in the list of active queries, those which are affected by the event, and put the event in a queue. Consider the situation of Fig. 3. The object's trajectory leads first to (the rectangle of) query q_1, then to query q_2 and finally to query q_3. So we expect that the object will soon enter in turn each of these rectangles and prepare the system to notify the clients of these events, assuming that no update of the object's position is received by the MOS.

On the other hand, since we have to cope with possible updates of the trajectory, such an anticipation of query movements is difficult to manage. Indeed, imagine that we associate object o with the event list of queries q_1, q_2 and q_3, together with the period of time during which the object is expected to come across their respective rectangles. Then, whenever an update of o is received, we should search for all the queries for which the position has been anticipated.

More generally, an object (and consequently the events associated with this object) is possibly shared by many queries, and a query is likely to be the target of a lot of events. In an intensive

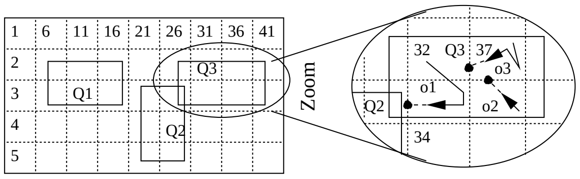

Figure 4: The grid, with queries and events

application dealing with a large number of queries over a massive set of moving objects, a poorly designed management of the association between queries and events will lead to bad performances and possibly to improper information communicated to the clients.

3. EVENTS/QUERIES MATCHING

We describe in this section the data structures and operations used to associate queries and moving objects. They consist of the following components: (1) a spatial data structure which serves as an index on the set of queries, and permits to determine easily whether a given event affects a given query, (2) a uniform handling of the two types of events identified previously, namely those provided by GPS servers, and the quit/enter event of an object in a query rectangle, (3) operations to add queries and events, and to implement the *pull()* operation.

The spatial data structure

In a traditional approach, the data is indexed, and the evaluation of a query relies on the index to retrieve the result. However, in that case, we are dealing with a large number of persistent queries over a spatio-temporal database. It turns out that it is much more difficult to deal with moving objects, because of their intensive updates and of their continuous movement, than with queries which are fixed rectangles and require no update.

Thus, instead of defining a structure that indexes events and using this structure to evaluate a query, we propose a structure that indexes queries, and use this structure to find all the queries affected by an event. The data structure is based on the *fixed grid* [3], a simple spatial index which decomposes the search space into rectangular *cells*. Each cell covers a rectangle $rect$ and is identified by a label l which can be obtained by scanning the structure in a convenient order. We rely on the grid to match queries with events as follows:

- **Indexing of queries**. The set of queries can be viewed as a set of rectangles, and can therefore be indexed with the fixed grid. The technique is traditional: for each query q we compute the set of labels $l_1^q, l_2^q, \dots l_n^q$ of the cells intersected by $q.rect$. We obtain a set $\mathcal{Q}$ of pairs (q, l) where q is a query and l one of its labels.
- **Partitioning of events**. We assign to each cell c the moving object relation, $c.O$, containing all the objects that come across the cell's rectangle, with the fragment of their trajectory that intersects $c.rect$. This fragments starts from the instant when the objects entered the cell, and stops at the instant when the object *is expected* to quit the cell, denoted t_{quit} in the following.

The structure is illustrated in Fig. 4. The grid is a set of $5 \times 9 = 45$ cells, labelled from 1 to 45 following a column-first order. Three window queries are shown : query $Q1$ is associated to labels $\{7, 8, 12, 13, 17, 18\}$, query $Q2$ with labels $\{23, 24, 25, 28, 29, 30\}$

and query $Q3$ with labels $\{27, 28, 32, 33, 37, 38, 42, 43\}$. All the pairs (Q, l) can be inserted in a list and indexed on the id of queries.

The right part of the figure shows the moving objects in the region containing queries $Q3$ and – partially – $Q2$. Three objects, $o1$, $o2$ and $o3$ are currently in cells 33, 38 and 37. Given their current direction and speed, they are expected to leave the cell at a location represented by a black dot, and at a time instant which can be easily computed.

Whenever an event is received from a GPS server, it suffices to search the grid to get the cell where the object is currently located, and consequently the queries whose result is affected. The event is recorded in the relation associated to the cell by updating the local trajectory of the object, and is then ready to be fetched by a *pull()* operation issued by any of these queries.

Unifying events

We must now handle the fact that an object quits or enters a cell of the grid. Indeed, consider object $o1$ in Fig. 4. It will soon quit the cell 33 and enter the cell 28. Then the object will become part of the result of query $Q2$.

This should be recorded in the structure. To this end we can use a list $\mathcal{E}$ storing tuples $< t_{quit}, o, l_{in}, l_{next} >$. Each tuple states that object o is currently in cell l_{in} that it will leave at time t_{quit} to enter cell l_{next}. For instance, the list associated with the instance of Fig. 4 is $< [t_1, o_1, 33, 28], [t_2, o_2, 38, 37], [t_3, o_3, 37, 32] >$, with $t_1 < t_2 < t_3$.

The list $\mathcal{E}$ must support functions $insert(t, o, l_{in}, l_{next})$ and $remove(t, o)$, given a time instant value and an object o, and function $next()$ to retrieve the tuple with minimal time value.

A dedicated module of the server is responsible for retrieving periodically the next object that quits its current cell. This module issues then a call to the *nextEvent()* function of the MOS in order to inform the server. In other words the server considers uniformly as *events* the updates coming from GPS servers and the elements returned by $next()$. The module can thus be seen as a “pseudo” GPS server which detects when an object quits its current cells, and generates an appropriate event.

Operations

Here is a short description of all the operations supported by the above structures:

- *AddQuery* ($q(rect, I)$).
  This a standard operation of adding a rectangle in a fixed grid. The rectangle $rect$ is decomposed in all the cells $c_1, \ldots c_n$ that it covers, and the pairs $(q, c_1.l), \ldots (q, c_n.l)$ are inserted into $\mathcal{Q}$.

- *AddEvent(evt).*
  Whenever an event $< id, t, loc, speed, direction >$ is received, we find the (unique) cell c such that the rectangle $c.rect$ contains loc. Let o be the object identified by $evt.id$ and t_{old} the instant when o was expected to quit the cell. We compute (1) the time instant t_{quit} when the object o identified by $evt.id$ will leave the rectangle $c.rect$ given its new location, direction and speed, and (2) the next cell l_{quit} encountered by o. Then we call $remove(t_{old}, o)$ and $insert(t_{quit}, o, c.l, l_{quit})$ to replace in $\mathcal{E}$ the values associated with o.

  If o already belongs to $c.O$, the new segment is appended to its trajectory, else o is simply inserted in $c.O$.

- *pull(q).*
  Finally the implementation of $pull(q)$ is trivial: search in $\mathcal{Q}$ all the labels $l_1^q, l_2^q, \ldots l_n^q$ associated with the query identified by id, access each of the cells labelled by a l_i^q, and get the events.

All these structures and operations can be implemented either with standard data structures in main memory (e.g., balanced search trees), or, for very large datasets, using a database with B-tree indexing.

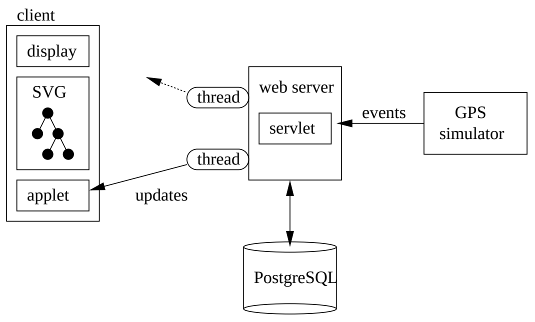

Figure 5: Architecture of the prototype

4. THE PROTOTYPE

We implemented a first version of our architecture with two objectives in mind: first to test the effective applicability of the framework, and second to experiment visualization and animation techniques coupled with the server. In the current state of implementation, clients can only visualize the whole map and the objects moving on this network. In other words one cannot express window queries to focus on a part of the map. Thus the main aim of the prototype is to check that the architecture design can effectively be implemented with current, state-of-the-art technology. The main features of the prototype are summarized below:

- GPS servers are represented by a simulator which generates incrementally the trajectories of a set of moving objects on a constrained network.

- The server is a java Servlet which gets the events from the simulator, and handles the association between queries and events according to the simple framework presented above. The server is also responsible for transmitting data to the client.

- Clients are web browsers equipped with the *SVG browser* from Adobe. When a query is registered, each client receives a small applet which connects to the server and sends *pull()* requests.

- The server sends initially to the client an SVG document (see below) representing the geographic data (map and network) as well as animated objects. This document is represented at the client's side as a DOM tree which is updated by the applet whenever an update is received. The SVG browser modifies automatically the display presented to the user.

Fig. 5 sums up the whole architecture of our prototype. In the following we first introduce SVG, and then propose a brief overview of these components. Because of space limitations, we refer the interested reader to the full version of paper, available at *http://cedric.cnam.fr* for a detailed description.

4.1 The SVG language

XML is a meta-language that allows the definition of specialized languages for a wide range of applications. One of these specialized language is SVG (Scalable Vector Graphics) which permits to describe graphic objects with colors, textures and animation [21]. SVG can represent and display the main primitives used in cartographic representation, in a vector format suitable for various scale display. In addition, SVG provides animation, and is therefore a seductive language for spatio-temporal objects visualization.

The following sample shows the SVG description of a path with five points. The `d` attribute gives the geometry and the `style` attribute gather the specification of the graphic representation.

```
  <path d="M0,0 L 100,0 100,100 100,200 150,250"
style="fill:none;stroke:#00FF00; stroke-width:5" />
```

Most of the information associated to an element is represented by attributes. By adding a nested `animate` element, we can describe a continuous variation of the value of these attributes. The variation consists of the time instant when the animation begins, the duration of the animation, and the interpolation function (linear by default) used to switch from the initial value to the final one. The fragment below creates a red circle moving on a path during 30 s, with constant speed (the "L" in the `path` attribute stands for "linear").

```
  <circle x="0" y="0" r="20" style="fill:red;">
<animateMotion dur="30s" repeatCount="indefinite"
path="M0,0 L 100,0 100,100 100,200 150,250" />
</circle>
```

4.2 The simulator

The simulator provides an on-line generation of trajectories for a set of objects on a constrained network. *On-line* here means that a trajectory is built step by step, rather than fully dertermined in advance. At a given time instant, the simulator knows that an object o moves on a path of the network, in a given direction and at a given speed. Eventually o will reach the next node of the network: the simulator computes then the new path assigned to o, together with a new speed. The choice is driven by a set of properties associated to the network and describing the distribution of the traffic. More specifically, we take into account the following information:

- A *weight* is assigned to each node. It represents the relative "importance" of this node as a starting point for vehicles. In the context of car simulation for instance, a node located in a residential area gets a a weight of 4 or 5, and a crossroad in a desert zone a weight of 0.

- In addition to the weight of a node, an *absorption rate* property defines the attractive power of a node for vehicles. Intuitively, whenever a vehicle reaches a node with a strong absorption rate, it is likely to stop and stay there. A working area presents a high absorption rate, and a crossoad a low one.

- Finally, given a node n, we give to each edge starting from n the probability to be chosen as a path by an object arriving at n. For instance, a highway is more likely chosen that a secondary road.

These parameters, plus some others that, for simplicity, we omit here, are simple, the goal being limited to simulate information provided by a GPS with a reasonable amount of realism. Based on the above parameters, a simulation can be run using a few set of methods.

4.3 The server

The server plays the central role in the architecture (Fig. 5). It is implemented as a java Servlet connected to a PostgreSQL database [15]. The database stores the network, and we also record in PostgreSQL the trajectories of objects built from the events provided by the simulator. We plan to implement in the future a spatio-temporal query language to query this database.

When a client (web browser) connects to the server, a SVG representation of the network is first sent, along with a background map. In a second step, a java thread dedicated to the client is created on the server's side, and a small applet is sent to the browser. As soon as the applet is installed in the browser, it connects to the thread and begins to send *pull()* requests in order to get new events.

When the server receives an event from the simulator, it registers this event in a priority queue, and waits for clients requests. Each time a *pull()* is received by a thread, it retrieves the events from the queue and send them to the client.

An important role of the server is to produce the SVG displayed by the client. Currently we directly create the SVG document that includes the map of the simulation from data stored in the PostgreSQL database. In a more sophisticated implementation where the server must be able to communicate not only with web client, but also with PDA, mobiles phones, or is integrated in a peer-to-peer architecture, a more neutral XML representation (such as GML, the Geographic Markup Language) could be produced and processed by the transformation language XSLT in order to produce the specific format required by the client.

4.4 The client

The client is a web browser providing a visual interface which can be basically divided in three main parts: the HTML page (including some Javascript code), the SVG document, and the applet. We should notice that a plug-in is compulsory for displaying SVG in a HTML document (we use the SVG Viewer from Adobe [2]). When the client loads the HTML page initially sent by the server, it also loads and displays the SVG document. The next task is to update this document (and the graphic representation) when events are retrieved from the server.

A SVG document, like any XML-based document, corresponds to a DOM tree (Document Object Model) where each node is an element [8]. The DOM specification describes an object-oriented interface to represent and manipulate the tree. Thanks to this interface, we can insert, delete, access and modify a particular node. For example the function *GetNodeById(myID)* is used to get the element whose identifier is *myID*. We use a Javascript implementation of the DOM interface to modify the SVG document on the client's side. When the document is modified, the SVG browser immediatly transmits the modification to the graphical display.

In summary the update cycle of the client is as follows: (1) the applet sends periodically *pull()* queries to the server (2) the server sends back events (if any) to proceed and (3) the applet and the javascript apply these updates by modifying the DOM tree of the SVG document.

5. RELATED WORK

The notion of continuous queries, described as queries that are issued once and run continuously, is first proposed in [19]. The approach considers append-only databases and relies on an incremental evaluation on delta relations. Availability of massive amounts of data on the Internet has considerably increased the interest in systems providing event notification across the network. Some representative works are the *Active Views* system [1], the NiagaraCQ system [6], and the prototypes described in [14, 9]. The fact that notification systems receive a large amount of queries has suggested that queries should be indexed rather than data: for instance [13] applies the idea to XML documents servers.

In the area of moving objects, the only paper closely related to our topic is [5] which proposes a web-based architecture for representing and querying moving objects. It consists of a client, an Internet server, a map server and a spatiotemporal database system. XML languages (in particular GML and SVG) are used to send data from the server to the client. Finally it is worth mentioning that several simulators are proposed in the recent literature ([20, 17, 4]), the closest of ours being [4] which explicitely addresses simulation on constraint networks.

Although more elaborated with respect to the specification of the simulation, a difference with our generator is that it provides all the scenario (i.e., trajectories) in one pass, whereas we need to simulate real-time moving objects whose future is uncertain. We could probably adapt the design of [4] to our need.

6. CONCLUSION

In this paper we proposed an architecture for a Moving Object Server acting as a mediator between *queries* submitted by web user and *events* received by the server. Assuming that such a server must be able to support a large amount of queries, we discussed the necessity of designing appropriate data structures to map queries with events. We describe a simple solution relying on an index built on queries, thus avoiding the difficulty of indexing moving objects(see for instance [16]). Finally, our protoype shows that our aproach can be implemented with little effort and standard, though advanced, languages and products.

In the future we plan to implement our simple structure to match queries with events, and test the impact of some important parameters such as the size of the cells with respect to density of queries, the average speed of objects, etc. We shall also consider some optimizations. For instance, with the current design, the search space is uniformly decomposed, sometimes without necessity, in small areas. This is probably harmless in a situation where the query windows cover the major part of the search space, but can become a problem if query windows are sparse: then a lot of manipulation are required as objects move from a cell to another quite often. We plan to investigate cell clustering strategies to improve this situation.

Acknowledgements

We are very grateful to Michel Scholl and David Gross for useful comments on early drafts of this paper.

7. REFERENCES

[1] S. Abiteboul, B. Amann, S. Cluet, A. Eyal, L. Mignet, and T. Milo. Active Views for Electronic Commerce. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 1999.

[2] Adobe. The SVG Viewer, 2001. URL: http://www.adobe.com/svg.

[3] J. L. Bentley and J. H. Friedman. Data Structures for Range Searching. *ACM Computing Surveys*, 11(4), 1979.

[4] Thomas Brinkhoff. Generating network-based moving objects. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, pages 253–255. ACM, 2000.

[5] Thomas Brinkhoff and Jürgen Weitkämper. Continuous Queries within an Architecture for Querying XML-Represented Moving Objects. *Lecture Notes in Computer Science*, 2121, 2001.

[6] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2000.

[7] Open GIS Consortium. Url: http://www.opengis.net, 2001.

[8] WWW Consortium. The Document Object Model, 2000. URL:http://www.w3.org/DOM.

[9] F. Fabret, H. Jacobsen, F. Llirba, K. Ross, and D. Shasha. Filtering Algorithms and Implementations for Very Fast Publish/Subscrib Systems. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2001.

[10] L. Forlizzi, R.H. Güting, E. Nardelli, and M. Schneider. A Data Model and Data Structures for Moving Objects Databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2000.

[11] S. Grumbach, P. Rigaux, and L. Segoufin. Manipulating Interpolated Data is Easier than you Thought. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2000.

[12] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and Michalis Vazirgiannis. A Foundation for Representing and Quering Moving Objects. *ACM Trans. on Database Systems*, 25(1):1–42, 2000.

[13] L.V.S. Lakshmanan and P. Sailaja. On Efficient Matching of Streaming XML Documents and Queries. In *Proc. Intl. Conf. on Extending Data Base Technology*, 2002.

[14] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):610–628, 1999.

[15] Bruce Momjian. *PostgreSQL, Introduction and Concepts*. Addison Wesley, 2000. To appear. See `http://postgresql.org`.

[16] D. Pfoser, Y. Theodoridis, and C. S. Jensen. Indexing Trajectories of Moving Point Objects. Technical report, Chorochronos Network, 1999.

[17] Jean-Marc Saglio and Jose Moreira. Oporto: A realistic scenario generator for moving objects. *GeoInformatica*, 5(1):71–93, 2001.

[18] A. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 422–433, 1997.

[19] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous Queries over Append-Only Databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1992.

[20] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. In *Intl. Conf. on Large Spatial Databases (SSD'99)*, 1999.

[21] A.H. Watt. *Designing SVG web graphics*. New Riders, 2002.