A Novel Architecture for Mobile Distributed Trie Hashing System

Amel Bennaceur, Djamel Eddine Zegour, Walid Khaled Hidouci

A Novel Architecture for Mobile Distributed Trie Hashing System

A. Bennaceur
ARLES Project-Team
INRIA Paris-Rocquencourt
France

DE. Zegour
Institut National
d'Informatique, INI
Algérie

WK. Hidouci
Institut National
d'Informatique, INI
Algérie

Abstract

Scalable and Distributed Data Structures (SDDS) are a class of data structures completely dedicated to distributed environments. They allow the management of large amounts of data while maintaining steady and optimum performances. Several families of SDDS have been proposed: LH*, RP*, DRT*, CTH*. None of these SDDS deals with the mobile environment. In this paper we present a novel architecture that uses a scalable and distributed data structure to manage insert/find/range query operations for mobile clients. We describe the design and the implementation of a mobile CTH* prototype. Our experimental results prove the validity of the design choices and show interesting access performances. The capabilities of the mobile CTH* platform offer new perspectives for high performance and ubiquitous data intensive applications.

Keywords: Mobility, scalability, distributed data structure, trie hashing.

1 Introduction

Scalable and Distributed Data Structures (SDDS) are new data structures designed for multicomputers (i.e. collections of autonomous workstations or PCs connected through a high-speed network)[1]. An SDDS file is organized into data buckets which are stored in specific nodes called servers. The number of servers dynamically scales with the file growth. Servers are accessed from independent and autonomous nodes called clients. Each client has its own image of how data is distributed. Since this image is not updated synchronously, the client may have an outdated image and make an addressing error i.e. contact a server which does not contain the required data. This server is able to forward the client's request to the correct server which sends an Image Adjustment Message (IAM) to the client. The information in the IAM does not necessarily make the image totally accurate, but at least it avoids repeating the same error twice [1]. There are many SDDS schemes which extend traditional data structures. Thus, There is those based on hashing techniques like LH* [16] and those based on trees like RP* [17]. More information can be found in [2]. CTH* (Distributed Compact Trie Hashing) is a scalable and distributed data structure which adapts a dynamic hashing method called trie hashing for distributed environments. Trie hashing (TH) is one of the fastest access methods to primary key ordered dynamic files [3]. Nowadays, it is not sufficient to offer simple and fast access to a large volume of data. It is necessary to create an environment of mobility where the user can access the data and resources anywhere, anytime. Mobile end-user devices and wireless connection technologies allow for anytime, anywhere vision [18] and recent developments [19] show remarkable approaches toward mobile computing. However, these developments often face the problem of striving towards mobility in a rather isolated way. Mobile technologies are somewhat limited and these limitations are not expected to disappear in the near future. Thus, we argue that mobile information systems can reach their expected advantages and user acceptance only through integration with existing wired infrastructures, i.e the full value of mobile computing can only be achieved by taking advantages of a wired infrastructure such as reliability, availability, bandwidth, richness in capabilities in conjunction with the benefits of wireless technologies such as mobility, ad hoc connectivity or context awareness [20, 21]. In this paper we present a novel architecture that extends a scalable and distributed data structure scheme toward mobile environments. Section 2 presents an overview of the main concepts and characteristics of distributed compact trie hashing. Section 3 describes the proposed architecture. Section 4 presents the performance study. Section 5 suggests some potential applications. Section 6 positions our work with related work, and finally, Section 7 concludes by underlining the main results and highlighting future research directions.

2 Distributed Compact Trie Hashing (CTH*) concepts

We recall here the principles of CTH* schemes as defined in [10]. A CTH* file is stored on server computers (nodes), and is accessed by applications on client nodes. The file consists of records identified by primary keys. Keys are character strings of fixed maximum length with characters, called digits, from a finite alphabet. The digits are assumed to be lexicographically ordered. Hence the set of all possible keys is totally ordered. The smallest digit of the alphabet is '_' and the largest is '|'. Non-key parts of records are not of interest in what follows so we will identify records by their keys. Records are stored in buckets with a capacity of b records; $b >> 1$. Buckets are numbered $0, 1, 2...N$. Basically, there is one bucket of a file per server, although different servers may share files. Buckets are assumed to be in RAM. The file starts with bucket 0, and scales up with inserts, through bucket splits. Bucket addresses are mapped to the network addresses of the servers using physical allocation. The splitting and addressing rules of CTH* are based on those of compact trie hashing (CTH) [3]. CTH [3] is a variant of the well-known method TH (trie hashing). CTH represents the digital tree in a compact preorder sequential form to minimize the memory space necessary to represent the trie. The idea is to represent the links in an implicit manner to the detriment of maintenance algorithms slightly longer than those of the standard method. CTH consumes only three bytes per file bucket, which makes it possible to address some millions of records with a small memory capacity. Every split moves about half of the records in a bucket into a new bucket at a new server, appended to the file. The splits are triggered by bucket overflows. In CTH*, a bucket that overflows reports to a dedicated node called the coordinator. The coordinator uses a PAT (Physical Allocation Table) to find a new server. If such a server exists, it informs the overflow server which initiates the record transfer. The address a is defined by the CTH addressing algorithm [3]. To avoid a hot spot, CTH* clients do not access the coordinator for the address computation. As for any SDDS, a CTH* client has therefore its own image of the file. For CTH*, this consists of a trie; |0 for a new client. These values may differ from one client to another. The client uses its image to calculate the address a' for c while issuing a (point-to-point) request (a search of key c, an insert or a delete of a record). It then sends the request to server a'. It might occur that $a' \neq a$. Hence, every server s receiving a request first tests whether $s = a$. For this purpose, every server keeps its own trie. If the test fails, the server forwards the request to another server. The CTH* test and the forwarding algorithm is as follows :

- Use the server trie to calculate the address a' of c
- if $a' = a$ then accept c
- forward c to server a'

The forwarding process could a priori create many hops. The major property of CTH* is, however, that every request to a CTH* file is delivered to the correct address after at most four hops, [10]. The server also sends a message back to the client, called an Image Adjustment Message (IAM). For CTH*, an IAM contains the trie part that will be changed. The client executes the IA algorithm. The result is a better image, with a trie closer to the actual values. Also, as long as there is no new split, the same addressing error cannot occur.

The system is initialized with only server 0 which has an empty bucket and (|0) tree i.e. all keys are mapped to server 0. We can have several logical servers for the same physical server.

3 Architecture

3.1 Messaging

When designing enterprise applications, for example, in a corporate intranet, the choice of transport method is fairly trivial. Application developers can choose whatever solution suits their needs the best, since typically there are no firewalls disturbing the data connections inside the intranet. When the enterprise application is meant to be accessed from outside the intranet, or as in this case, from a mobile device, the number of suitable transport methods is considerably reduced. Hypertext Transfer Protocol (HTTP) is an application-level, request-response type of protocol that was originally designed for transferring raw data, such as Web pages, across the Internet. Today the protocol is used for various other purposes as well. The current version of the protocol is HTTP/1.1 and, compared to the earlier version (HTTP/1.0), it contains more enhanced features. For example, the underlying Transmission Control Protocol (TCP) connections can now be utilized more efficiently. Two of these features are the support for persistent connections and pipelining. By specifying the HTTP header value Connection: keep-alive, the underlying TCP connection can be kept alive for the duration of more than one request-response pair. This feature is useful in many situations related to enterprise applications, since bandwidth is not wasted by unnecessarily opening multiple TCP connections (with multiple TCP handshake

messages for each connection). With pipelining, multiple requests can be sent at once in a single TCP segment. In addition to these, HTTP/1.1 has incorporated transport compression of data types to further reduce the use of bandwidth. [22] There are a couple of properties that make HTTP an attractive choice. Through the success of the Web, HTTP has become a very widely used protocol and today practically every device that has network access supports it - including mobile devices. There are many ready-to-use commercial and free HTTP servers available on the market. Moreover, since HTTP traffic typically uses TCP port 80 or 8080, it can easily navigate through firewalls without any additional reconfiguration. These advantages led us to choose HTTP for communication between the mobile client and the fixed servers. The downside of HTTP is that despite the improvements in HTTP/1.1, it is still not the fastest protocol available. This is mainly due to the properties of TCP, such as the so-called slow start mechanism, and the fact that HTTP traffic has low priority in the routers along a network. This means that especially when the network is congested, HTTP traffic slows down considerably. Sockets are the oldest and, as a concept, probably the simplest form of communication between applications on different devices. We simply have to create an application on the server to listen to an appropriate port for incoming connections and then set up the client application to connect to that host and port. Naturally, if there is a firewall between the client and the server, we need to reconfigure it to pass the traffic of the application. There are two types of socket connections available; the socket connection can use either User Datagram Protocol (UDP) or Transmission Control Protocol (TCP). Both of these are available essentially in all networked devices. The difference here is that with TCP a connection is always made between the client and the server and the same connection is used for consecutive messages, whereas each UDP datagram contains the target and source addresses and is forwarded individually through the network. This means that TCP guarantees that the messages are delivered in correct order to the application - UDP does not even guarantee that the datagrams are delivered at all. On the other hand, establishing and tearing down the connections takes more time and generates a lot more traffic than merely sending a single datagram when needed. Thus, we use UDP sockets for communications in the local network; in addition, we have a dedicated flow control protocol for requests using UDP messages, if losses are unacceptable. Thus, CTH* servers manage two communication protocols: UDP for the messages coming from fixed clients, the other servers or the coordinator and HTTP for mobile clients' requests. In a typical client/server model it is the server's duty to be running and available at all times and the client's duty to initiate connections to the server whenever needed. This type of interaction is called the "pull" technology (the client effectively "pulls" information from the server whenever needed).

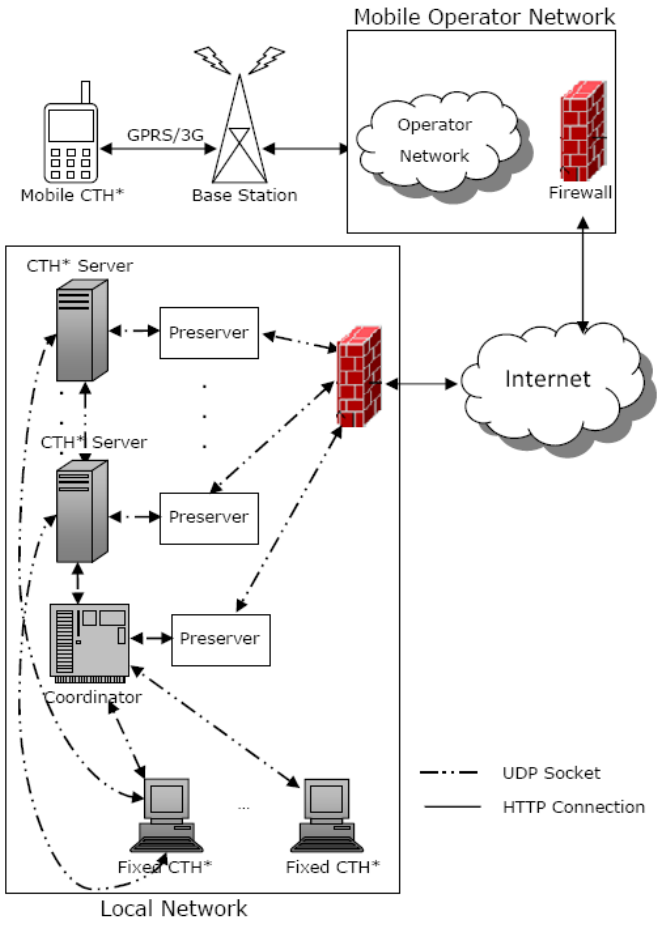

Figure 1: Global mobile CTH* platform Architecture

If the resources on the mobile device allow it, the client application can be up and running as well and the connection can be kept alive all the time. However, the mobile client in our application must be accessed from a server to allow image adjustment messages; thus, we use an intermediate program (proxy) that we call preserver. This preserver intercepts the mobile clients' HTTP requests, codes them to the adequate format and finally transmits them using UDP sockets. When it receives the servers' answers, it transmits them to the suitable client. It keeps the HTTP connection alive until the client receives all the IAM and the response to its request. This strategy offers the flexibility of the HTTP in a mobile environment, the speed of UDP in a local environment and allows the server to process the requests of both fixed and mobile clients in the same way.

3.2 Server

A server is created empty, and listening for a bucket creation request. These requests may create several buckets from different SDDS files. The buckets are allocated using the Bucket Allocation Table (BAT). BAT contains the bucket address, size and the ID of its file. The bucket creation request comes from a client for a new file or from a server with a bucket to split. For the clients, the server is identified by its IP address and UDP port. A site can support multiple servers on different ports.

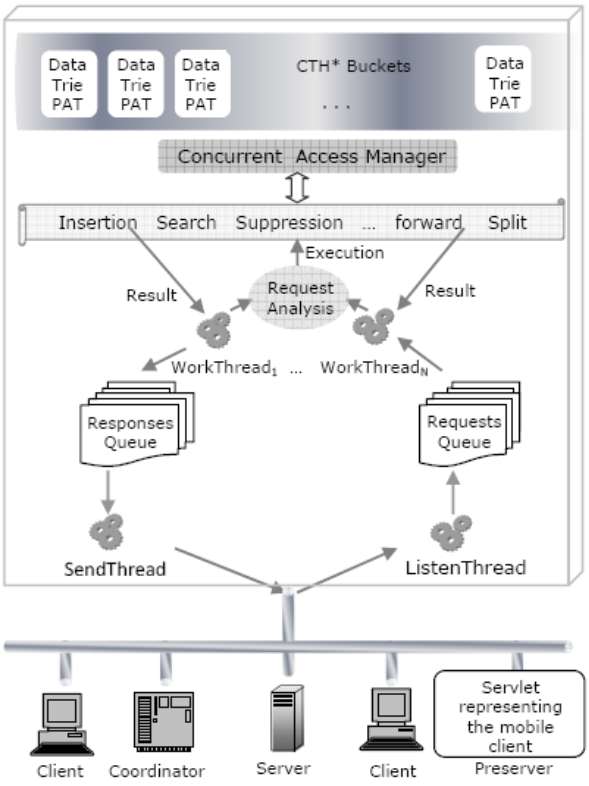

Figure 2: Server's Architecture

The servers use multithread processing, Figure 2. Several threads take care of the processing, asynchronously and in parallel. They communicate through queues and semaphore. The ListenThread receives a client's request, and puts it into a FIFO RequestsQueue and waits for the next request. A WorkThread, from a pool of such threads noted $WorkThread_i(i = 1, .., N)$ in Figure 2, wakes up. The number of active WorkThreads depends on the server load. The thread reads the next request in RequestsQueue. It identifies the operation to perform. WorkThreads continues with the processing of the requested operation. Then, it puts the answer into the ResponsesQueue. Finally, the SendThread sends the server's responses. The creation of a new bucket or the splitting of an old one, are as follows. The latter case subsumes the former so we present only that one. Let S_0 be the splitting server, and let the S_i be any server among M currently started, $1 \leq i \leq M$. The split at S_0 is done as follows. First, S_0 sends the SplitRequest message with the requested bucket size to the coordinator. If there is a server S_i that has enough space, the coordinator replies to S_0 and gives it the S_i address. S_0 locates the middle key C_m in its bucket and splits the bucket into two groups. One group, let it be G_1, contains records with keys $C > C_m$ and their CTH sub-trie. The other group G_2 contains all the other records and their sub-trie. S_0, sends G_2 to S_i. It then updates its trie and data zones. Finally, S_i creates the bucket with the received records and sub-trie. The clients' requests in RequestsQueue received during the split are dealt with as usual once the split is over.

3.3 Fixed client

The CTH* client architecture is structured into three modules: send, reception and processing. They run in parallel, and are further structured as in Figure 3.

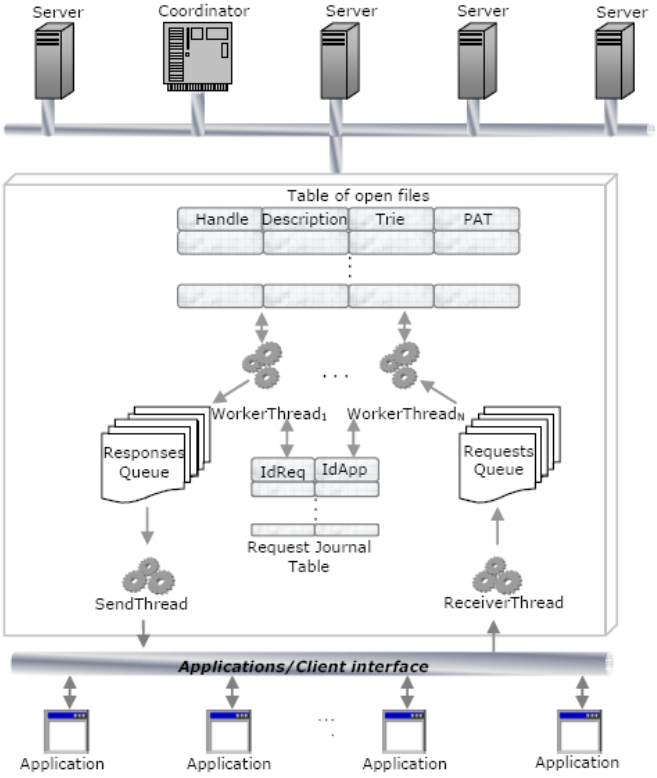

Figure 3: Fixed Client's Architecture

The processing module, composed from a pool of threads, processes the application request and sends it to the servers. It consults the client image, to determine the IP address and the type of the message to use. It also updates the client image from the IAMs. The reception module puts the incoming request ID (IdReq) and that of the application (IdApp) into the RequestJournal table. Both Ids are provided by the application. Next, it processes the request and puts it into a FIFO queue. The SendThread reads the queue. It builds the messages to the servers accordingly and sends them. It also manages application responses. It searches IdApp in the request journal for each IdReq in the reply retrieved from the queue and returns the data to the application.

3.4 Mobile client

The CTH* client architecture is structured into two modules: send and the processing. They run in parallel, and are further structured as in Figure 4. The processing module, composed from WorkThreads, processes the application request and sends it to the servers. It puts the outgoing request into a FIFO queue. The SendRequest thread reads the queue. It builds the messages to the servers accordingly and sends them. The thread WorkThread can then treat other functions until it receives the answer to its request. It then finishes the treatment and returns the result to the interface thread which displays it. As a mobile device can manage many HTTP connections simultaneously, we use a set of SendThreads to treat HTTP connections.

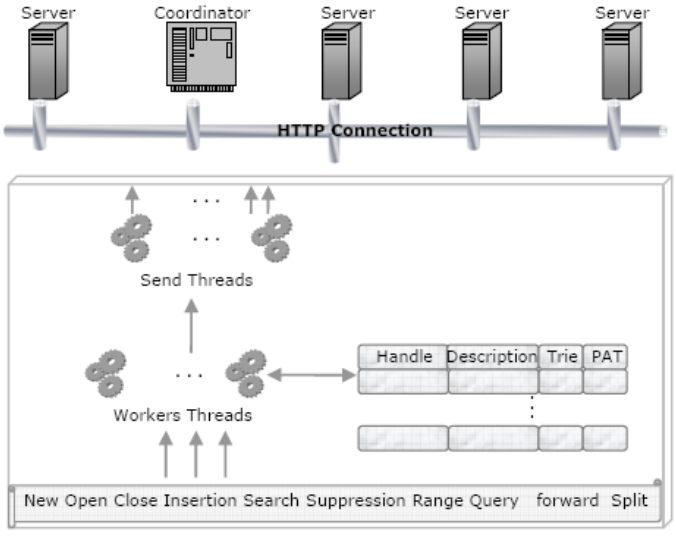

Figure 4: Mobile Client's Architecture

3.5 Preserver

The preserver plays a mediator's role between the mobile client and the CTH* servers. When an HTTP request is received, the preserver creates a new thread that encodes and sends the message to the server using a UDP socket. If this server is the correct one, it transmits the answer to the preserver. In the case of an addressing error, the server transmits an IAM and forwards the request to another server s'. If s' is the correct server, it transmits the answer to the preserver, otherwise it forwards it and so on. The preserver must keep the HTTP connection alive during the request processing and disconnect only when all the IAMs and the response have been received.

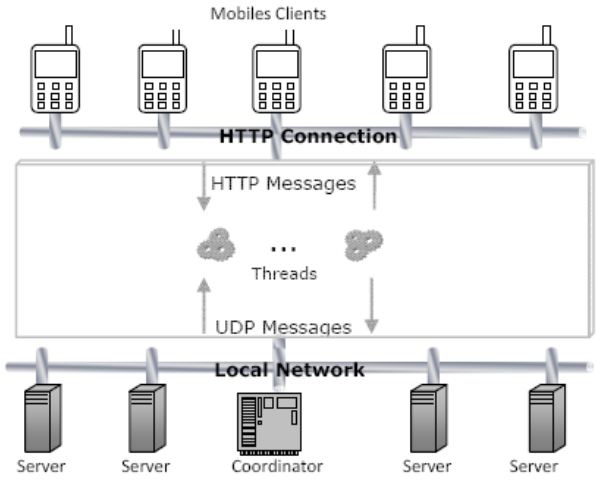

Figure 5: Preserver's Architecture

4 Performance analysis

The platform used for our experiments is a cluster of workstations consisting of four nodes. The first is a 2.66 GHz Pentium VI processor with 248 Mb main memory. The second is a 1GHz Pentium III with 256 Mb RAM. The third is a 688 MHz Pentium III with 256 Mb RAM and the fourth is a 1.73 MHz Pentium Centrino laptop with 512 Mb RAM. These workstations are connected through a 100 Mb/s Ethernet network and each machine has a public IP address. To test the mobile client we use a cell phone, Nokia 3200, with 800 Kb of memory Java MIDP 1.0 enabled and connected to the Internet via a GPRS network.

To conduct the performance analysis of our prototype implementation, we used an application which builds the CTH* file by inserting a large number of

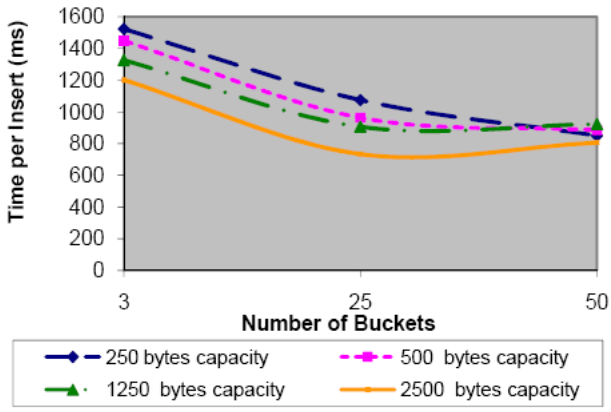

Figure 6: Average insertion time for a mobile device

data elements. The application uses random data keys. Attached to each key is a data element which has a random size. Bucket capacities range from 250 to 2500 bytes. To test servers' scalability in a more efficient way, we use virtual servers on the same machine. Given the range of capacities, we then have files with between 3 and 50 buckets. This section only presents a selection of our experimental results. More results can be found in [11]. Adding a mobile client maintains the performances of the fixed stations: same access time ($< 1ms$) and same load factor (%70). Figure 6 shows the average insertion time. The curve demonstrates that mobile CTH* is truly scalable since the average insertion time is constant (about 1.5 s) and independent from the number of insertions. Hence, the insertion time does not deteriorate for large data structure sizes. In addition, Figure 6 indicates that the insertion time decreases when many records are used. This is due to the increased parallelism as each client operates concurrently. However, it is greater than the average insertion time for a fixed client. This is due to the low bandwidth and the longer latency time which are inherent in mobile networks. Hence it should decrease if we use a more advanced packet-switched mobile network like 3G. We also study its performances with range queries. A range query is an operation which is sent by a client, in parallel, to select a group of servers and its execution at each server is mutually independent [14]. Queries to all the file buckets are the most typical and we restrict our attention to these queries. We obtain the graph in Figure 6. Support for range queries is a major advantage of order preserving methods like RP* and CTH*.

Subsequently, it would be interesting to incorporate the latency periods experienced in practice when using different transmission technologies. Whereas test runs of the client on an emulator located on the same computer or in the same network as the server exhibit scarcely any differences among the protocols, such differences become truly noticeable with a GSM or even a GPRS link.

5 Some potential applications

The general requirements for enterprise applications are that they are scalable, can be run in heterogeneous environments, and their development and deployment cycles are as short as possible. These can be achieved by choosing the right architectural model and the right technology for the task at hand [15]. All these requirements apply to mobile applications as well. Most notably, the wireless environment is yet another piece in the puzzle of heterogeneous environments. As a rule, the mobile environment is much more restrictive than, for example, a normal corporate intranet environment as regard a how complex and heavy the client part of the application may be. Many enterprises already have a good selection of applications to which they might want to add access from company's mobile devices. To totally rewrite the applications would be a waste of time and resources. If possible, mobile client applications should be adapted to these existing applications and use existing code as much as possible. Some technologies are better than others for this purpose of acting as a wrapper over existing systems. If, on the other hand, the developer can make a clean start on both the client and the server side of the application, some other technology might be more suitable. Our main target was to provide a platform with a flexible protocol architecture that allows insert/find operations for mobile clients in a scalable manner. Examples of a typical application that would benefit from this platform might include database applications. Thus, we developed a sample application based on this platform. In our application scenario, different journalists take notes on their PDAs or cellular phones and transmit them instantly to the newspaper database. Every journalist can consult the notes of journalists working on the same topics. When returning to their office, they can find all the notes and start to write their articles. The primary purpose of this example is to provide an example application that uses our platform. Thus, the application has not been optimized in terms of user friendliness; it has been intentionally kept simple to maintain focus on the architecture itself. This framework inserts an additional security layer between the HTTP transport protocol and the application layer. When the mobile application starts and the user enters his or her name and password in the login screen, the application sends a login request to the preserver, which

validates the user name and password and responds with either a login-OK or login-error response. While logged in to the server, the user can add new notes, update, search or delete old ones from different articles (groups of notes). He can also, create new articles. Due to wireless connection of the mobile device and the integration of the existing infrastructure, the user can access his database anywhere, anytime.

6 Related work

There is a lot of related work on SDDS, specially LH*. Many variants exist, such as LH*, LH*$_{lh}$, variants that implement high availability (such as LH*s and LH*$_{RS}$) and also those adapted for P2P environments. They have been integrated into different DBMS, to provide self-managing and self-organizing data storage of potentially unbounded size. However, none of them deal with the mobile environment, which is of growing interest. We investigate this application area and take an important step towards implementing distributed data structures for mobile environments. Our architecture can be adapted to other SDDS like LH*. In addition, HTTP application request routing championed by Sarvega and subsequently by Cisco AON are examples of HTTP application requests from other computers. If one were to imagine HTTP requests from millions of handhelds, it is not clear that current hardware solutions can support them. We are proposing a novel scalable architecture. Whereas hardware solutions such as Sarvega (Intel) and AON (Cisco) can speed up the execution of an HTTP request routing, this method is the first approach to do something similar in a scalable fashion with online inserts. This paper has shown that even a software solution can get response times down to 1.5 seconds. This in conjunction with potential hardware solutions may be another interesting next step.

7 Conclusion

In this paper, we present a new mobile CTH* prototype which integrates fixed and mobile clients and offers simple, fast and efficient access to a great deal of information. It shows that scalable and distributed data structures can be ubiquitous. The object oriented design also gives this prototype more flexibility and extensibility. The experiments show that a client finds or inserts a given record in about 1.5 s. Among potential applications, we can cite modern database systems, continuously requiring larger scalable databases. Many existing databases or warehouses are growing extremely rapidly and require universal access. Further work, in progress, is extending our platform to include more aspects of mobile CTH* platform functionality. We are introducing mobile servers and adopting a peer-to-peer model. We are also optimizing the quantity of data exchanged between the mobile client and the preserver using compression methods and provide better security support using encryption methods. We are also managing high-availability using Reed Solomon codes [24, 25].

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments, which helped us improve the article.

References

[1] W. Litwin, MA. Neimat, D. Schneider, *LH*: Linear Hashing for Distributed Files*, ACM-SIGMOD International Conference on Management of Data, 1993

[2] http://ceria.dauphine.fr/SDDS-bibliographie.html

[3] D.E Zegour, W. Litwin, *Trie hashing with the sequential representations of the trie. Revue internationale des technologies avancées*, CDTA, Algiers, 1994

[4] W. Litwin, *Trie hashing: Further properties and performance*, Int. Conf. F.D.O Kyoto, May 1985

[5] MS. Birech, Y. Laalaoui, *Protocole de communication pour la nouvelle classe de SDDS : CTH**, mémoire d'ingénieur, INI, 2002

[6] A. Khaled, A. Bekkouche, *Implémentation d'une Structure de Donnes Distribue et Scalable sous Linux : Une nouvelle version de CTH*, mémoire d'ingénieur*, INI, 2004

[7] B. Saad, A. Boularias, *Implémentation et étude des performances de la SDDS Compact Trie Hashing (CTH*) sous Windows 2000*, mémoire d'ingénieur, INI, 2004

[8] A. Bouridah, A. Kheldoun, *Implémentation et étude de performance de la SDDS CTH* avec T-trees comme organisation interne du serveur*, mémoire d'ingénieur, INI, 2005

[9] B. Merani, H. Yakouben, *Implémentation de CTH* avec arbre central sous Linux*, mémoire d'ingénieur, INI, 2003

[10] DE. Zegour, *Scalable Distributed Compact Trie Hashing*, Information and Software Technologie, 25 may 2004

[11] A. BENNACEUR, *Intégration d'un client mobile CTH**, mémoire d'ingénieur, INI, 2006

[12] K. Al Agha, G. Pujolle, G. Vivier, *Réseaux de mobiles et réseaux sans fil*, Eyrolles, septembre 2001

[13] F. Thénoz, *Les nouveaux objets communicants*, Orange Press, 2001

[14] W. Litwin, M.A. Neimat, D. Schneider, *RP*: A Family of Order-preserving Scalable Distributed Data Structures*, In Proceedings of the 20th VLDB Conference, Santiago, Chilli, 1994

[15] *Enterprise: Developing End-To-End Systems*, Version 1.0, Nokia, January 9, 2006

[16] IST eGov project http://www.egovproject.org

[17] W. Litwin and, M-A Neimat, D. Schneider, *LH* - A Scalable Distributed Data Structure*, ACM Transactions on Data Base Systems, December 1996

[18] W. Litwin and, M-A Neimat, D. Schneider, *RP* : A Family of Order-Preserving Scalable Distributed Data Structures*, VLDB-94, Chile.

[19] L. Kleinrock, *Nomadicity: anytime, anywhere in a disconnected world*, In Mobile Networks and Applications 1, pages 351-357, 1996

[20] A. Frescha and S. Vogl, *Pervasive webaccess via public communication walls. In Pervasive Computing*, Springer LNCS 2414, pages pp 84-97, Zurich, Switzerland, 2002

[21] G. Banavar and A. Bernsein, *Software infrastructure and design challenges for ubiquitous computing applications*, Communication of the ACM, 45(12):92-96,2002,

[22] K. Raaikainen, H. B. Christensen and T. Nakajima, *Application requirements for middleware for mobile and pervasive systems*, ACM SIGMOBILE Mobile Computing and Communications Review, 6(4):16-24

[23] World Wide Web Consortium, *Network Performance Effects of HTTP/1.1, CSS1 and PNG*, http://www.w3.org/Protocols/HTTP/-Performance/Pipeline.html

[24] W. Litwin and J.E. Schwarz, *LH*RS, A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes*, p.237-248, Proceedings of the ACM SIGMOD 2000

[25] J. S. Plank, *A Tutorial on Reed-Solomon Coding for fault-Tolerance in RAID-like Systems, Software*, Practise & Experience, 27(9), Sept. 1997, pp 995- 1012