Scalable Interactive Middleware Components for Ubiquitous Fashionable Computers

Gyudong Shim and Kyu Ho Park

KAIST, Daejeon, Korea
gdshim@core.kaist.ac.kr, kpark@ee.kaist.ac.kr
http://core.kaist.ac.kr

Abstract. The middleware for location based interactive applications requires scalability in large scale spaces. As the number of users and target services are increased, the server has to process massive spatial queries and event handling requests efficiently. Our middleware components are developed to extend the U-interactive system for large scale environments. The system manages the location information for large number of users and target objects. In addition the system handles events caused by user commands. We developed efficient tuple indexing and query mechanism by composite keys. As a new spatial query, Fan search is invented to provide efficient target selection by distance and angle. We optimized the query processing by efficient node traversing and data-aware interval skipping. The tuple matching process is performed in bounded time up to 100,000 objects. Fan search with C-Cuve has superior performance than Z-Curve in high density nodes in the experiment.

Keywords: Ubiquitous middleware, human machine interactions, tuple spaces, spatial query indexing.

1 Introduction

Many researches have been come out to realize ubiquitous computing environments in the real world. The common philosophy of the researches is to make a convenient life with interaction with surrounding computers. Users in the ubiquitous environment can obtain any information by multiple interfaces and displays. Location based interactive applications in the ubiquitous environments have been developed in many research projects [10], [11], [12], [13].

Our research project teams have developed testbed on KAIST campus called U-TOPIA and a wearable computers named Ubiquitous Fashionable Computer (UFC)[3]. U-TOPIA has been equipped with indoor and outdoor testbed for location services which consist of ZigBee and UWB[9] sensor networks.[7].

UFC has been developed as a wearable gadget and cloth as shown in Fig.1 for convenience and portability. In order to interact with the environments with intuitive gestures, UFC has a gesture recognition device called *iThrow*[4]. *iThrow* is a ring typed intuitive interface device. *iThrow* can recognize specific movements and two dimensional pointing direction of the finger.

C. Giannelli (Ed.): Mobilware, LNICST 0007, pp. 144–156, 2009.

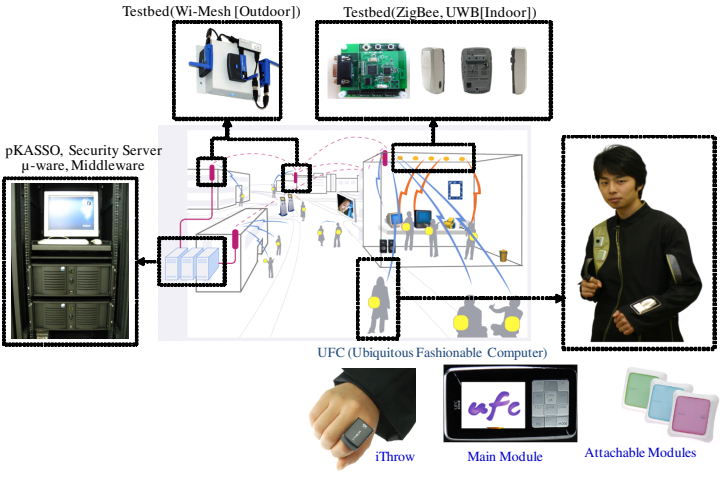


Fig. 1. U-TOPIA architecture and UFC components

A UFC user can interacts with objects and services by gestures. A UFC user can select a target inside the testbed by pointing it with *iThrow*. The selected target is shown at the display of UFC as a feedback. After target selection, the user can control the target through intuitive gestures such as throwing, pulling up, and rolling left or right. For example, user can print a document file by throwing to the printer after pointing the document object.

We designed an interactive middleware system, called U-interactive[5], that provides spontaneous interaction between UFC users and U-TOPIA. U-interactive provides Virtual Map which is a container and interface to the interactive objects. Each physical object can be assigned into the Virtual Map with location and interactive attributes.

In this paper, we introduce scalable middleware components designed upon the U-interactive system. It is important that the middleware should be scalable to be effective even in the massive environments such as subway stations, stadiums, and auditoriums. We assume that the target service areas would have up to hundreds of thousands objects with UFC users. We will focus on data indexing and query processing for the interactive services with scalability.

In U-interactive system, the server contains many objects of tuples for the target services and user locations. We developed a new tuple space based coordination middleware, which manages tuples and provides scalable tuple matching schemes in reverse time orders. In addition, it contains useful functionalities for file transfer and event handling mechanisms.

We propose efficient spatial query processing, which is called Fan search, will reduce computing overhead from massive query processing from users. The objects are searched within a specific fan query region. Fan search differs from previous rectangular based query processing such as NN, k-NN by query region and indexing scheme. Fan search indexes spatial objects by one-dimensional key

which is converted by a space-filling curve. As implementation optimizations, the path stacks of the index tree and skipping query regions are invented.

The paper is organized as follows. In Section 2, we describe the ubispace as basic coordination middleware for our system. The target selection algorithm will be presented in Section 3. In Section 4, the scalability of our components is evaluated. Section 5, 6 related middlewares and algorithms are discussed and concluded.

2 UbiSpace

Since many mobile clients and infrastructures exist in the ubiquitous environments, it is necessary computing paradigm for this distributed environment to communicate each other in the application level. As a black board system, tuple space has been researched and evolved by many researchers such as T-Spaces[6], JavaSpaces[8]. These tuple space provides applications with useful and simplified coordination. Due to the spatial and temporal decoupling effect of tuple space, many ubiquitous projects [1], [2] adopted tuple space as a core coordination middleware.

2.1 Tuple Space

The peers can write or read a tuple from the tuple space. A tuple space contains tuple which consist of a sequence of typed fields. Each field can be formal or actual, which are some type of an attribute or exact value of an attribute. These tuples are inserted into the space by peers who retrieve the tuples by tuple matching.

Tuple matching is explicit addressing method because it finds any tuple which matches a template without considering of the insertion order. Tuple matching between tuple t_1, t_2 can be defined as follow conditions.

1. both t_1 and t_2 have the same tuple name.
2. both t_1 and t_2 have the same number of fields N.
3. each of fields of t_1 matches the fields of t_2 in order, i.e. $t_1.\text{field}_i$ matches $t_2.\text{field}_i$, $\forall i \in \{0..N\}$

The matching of two fields are described as shown in Table 1. Notice that the equals on two actual comparison means java.lang.Object.equals() method. After that this tuple matching operation costs complex comparison of java objects.

Table 1. Tuple matching condition

Field(t_1)	Field(t_2)	matching condition
formal	formal	t1.class = t2.class
formal	actual	t2.object is assignable to t1.class
actual	formal	t1.object is assignable to t2.class
actual	actual	t1.object equals t2.object

UbiSpace is inspired by T-Spaces which is an extension of Tuple space and stable commercial product of IBM. T-Spaces provides event handler registration for incoming tuple by tuple matching. This mechanism is useful for implementing white board applications such as sharing distributed clipboard which reacts the status change of sharing data [6].

2.2 Limitations of Tuple Space

However these tuple space has some problems directly adapted in ubiquitous environments. First, tuple matching is implicit with the insertion order of tuples. If status variables are described with tuples, most of application requires the newest tuples in the tuple space. As a result, tuple space should provides tuple order by LIFO but T-Spaces provides only FIFO tuple matching mechanism. Second, tuple has expressive and general data description but it is a little bit complex for application developers. Application developers should pay attention to the tuple matching syntax. It requires redundant code fragments. So the API should has simple and concise method. Third, In the worst case tuple matching requires scanning of entire tuples to find a matched tuple. This is the most important disadvantage of tuple space. So T-Spaces provides indexed tuple matching by naming a tuple to specific string. However it is a duty of application developer that design and naming of tuple creation. If tuples are not properly manipulated by users, it requires full scanning overhead and not scalable to large number of tuples. Fourth, most of applications in our system operate a file as a tuple. Application developer should convert file contents to a general tuple field but it is burden of space to the server repositories because most of tuple space hold data in the memory of the server. Therefore tuple of File should be handled differently to save the server's memory space.

2.3 Design and Motivation of UbiSpace

From these limitation of tuple space, we designed and implemented a new tuple space or UbiSpace. The important characteristic of UbiSpace as follow.

1. **Design of String key based concise operations.** Like a hash table, tuple are indexed by string name tag. A tuple can be inserted, updated, and read by exact string match. For event handling we added publish and subscribe methods. subscribe method can register event handler for tuple event such as insertion of tuple, delete of tuple, update of tuple. these operations are anomaly of T-Spaces' event register, deregister operation but we added the count of event handler invocation. The event handler can be executed only the count times.
2. **Indexing tuples by name and tuple id** for scalable and deterministic tuple matching. UbiSpace prevents tuple matching from scanning entire tuples. Each tuple has a given name and unique id for indexing. By default there are two index of tuples, the fist is index of tuple ID for direct tuple addressing, the second is index of <tuple name, tuple ID> composite key.

The tuple ID is sorted reverse order for LIFO of tuples. In order to guarantee the bounded time of indexing insert, read operations, we adapted B^+tree[14] as a basic indexing tree.

3. **Automatic file transfer** by tuple operations. Any tuple which has a field of java.io.File objects are inspected during normal tuple space operations. If a tuple has File field, the content of the File object are transferred between client and server. Since both server and client have the same copy of the file in the given directory, the redundant file transfer is not performed by caching of the file. This automatic file operation is very simple and useful for file transferring application, especially in *ithrow* file throwing operations

2.4 Usage Example of UbiSpace

A typical usage of UbiSpace for interactions are outlined in Fig.2.4. In this example, a UFC user throws a file to the target service 2. In order to receive the command from the UFC, the services should register the subscription by subscribe operation. The subscription is identified by the prefix of "ithrow_data_" and "X" of actual target object id. By this manner, each services register their subscription by different string key of own ID in the system. The type of data can be recognized in the event handler of the subscription. After these registrations of subscription, any inserted tuples are examined whether they are matched to a specific subscription tuple template. The template tuples of ID 2, 3 and 4 are examined whether match the template key of "ithrow_data_2". Since the templates are named by "ithrow_data_2", "ithrow_data_3", and "ithrow_data_4", only the template tuple of ID 2 matches the inserted tuple <"ithrow_data_2", cam.jpg>. It is published to the ID 2, and the second field is type of File, the file contents are delivered to the ID 2. Finally the event handler is performed. The event handler has routine to display the image file.

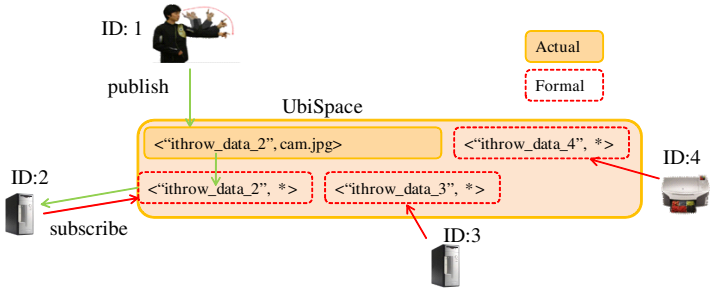


Fig. 2. UbiSpace usage for interaction between UFC and services

2.5 Implementation of UbiSpace

UbiSpace is implemented by traditional thread level server/client model. Fig.3 describes the thread level architecture of UbiSpace implementation. Considering

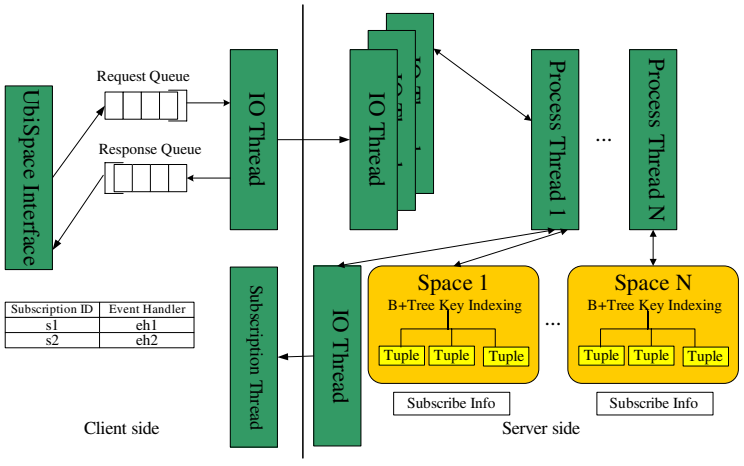


Fig. 3. UbiSpace architecture in thread level

harsh or unstable network conditions, the flow of request and response should be reliable to the packet loss or error of internal server status. Therefore we implemented timer based reliable request delivery mechanism. If a client cannot receive the proper response in a given timeout from the server, the request is retransmitted into the server. If a client fails to receive the response and retransmits the request over than N times in a row, the connection is closed and the client regards the server as unavailable. In order to implement timer based operations, we separated the application thread and IO thread which takes network IO handling. In addition java.lang.Object.wait(timeout)'s monitor is used for timer's timeout. Due to synchronization problem of requests of clients, the requests are serialized by the incoming order, i.e. FIFO in the synchronized queue. There are two queues per the space, request queue and response queue. The request is put on the request queue in a order, IO thread of the client take it from the right side. The response queue is used for waiting condition variables. The application thread waits on this queue to take the response. Because there are multiple spaces in the server, we balanced the workload of the single space to single thread. This approach is proper and beneficial because any further synchronization mechanisms are necessary such as condition variables or mutual exclusive locks.

B^+tree has been implemented basically algorithm of [14]. We modified query routine of nodes into bound check and binary search of a key. Since each node contains ordered keys, finding a key can be performed binary search in $O(logN)$ complexity. However binary search requires constant time of key traversal to escape the loop condition even when the lookup key is left or right side of the node. In order to eliminate unnecessary binary search attempts, UbiSpace checks the boundary of the traversing node whether this node has the key by with the lowest key and the highest key comparison. It may be constant overhead when a search key is inside of the traversing node range. Even though most of our system workloads find the newest key which resides on left side leaf nodes.

Table 2. String key based basic operations of UbiSpace

Return	Method signatures
long	insert(String key, java.io.Serializable obj)
Object	take(String key)
Object	read(String key)
long	publish(String key, java.io.Serializable obj)
long	subscribe (String key, java.lang.Class clazz, EventHandler callback, boolean isWrite)
void	unsubscribe(long seqNumber)
void	update(long itemID, String key, java.io.Serializable obj)
void	delete (long itemID)

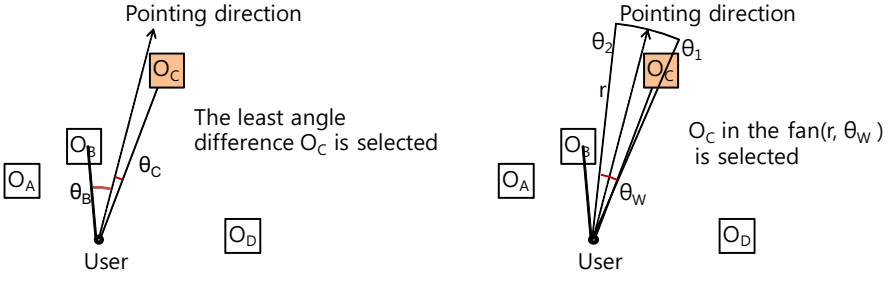
The basic string key based operations are listed in Table.2. The API doesn't require additional tuple manipulation logic for tuple matching. Only single string key can be assigned for tuple matching. The string key and object are sent in method parameters. The general tuple indexing operations are omitted due to the limit of this space. The general tuple space operation is similar with T-Spaces.

3 Fan Search: Target Selection Algorithm

Users can select an target service or an interactive object by pointing gesture of *iThrow*. Yoo et al, proposed target selection algorithm which select a object which has minimum angle difference.[4] They also proposed adaptive angle placement scheme for easy target selection of clusted objects. We devised Fan search in order to support a scalable target selection mechanism even in the large spaces such as square or stadium. Therefore the target selection algorithm has to filter objects by distance and angle efficiently.

3.1 Definition of Fan Search

The fan is defined by 4 arguments - radius, θ_1, θ_2, and origin(the location of the user) as shown in Fig.4(b). The radius is the maximum distance from the user



(a) Ray-based minimum angle (b) Fan search

Fig. 4. Target selection algorithms

location to the target. θ_1 is the start angle in counter clock wise direction. θ_2 is the end angle in counter clock wise direction. The angle of the fan is 0 to 2π.

The target objects residing in the fan are selected as the result of the search. The target objects are sorted by angle difference of the pointing direction and distance from the user location.

Algorithm 1. Fan search pseudo code

```
if θ1, θ2 span multiple quadrants then
   divide the angle from θ1 to θ2 by the piece of quadrants to A_i, i ∈ {1,2,3,4} {A_i
   denotes the angle in the i'th quadrant}
end if
for all A_i do
   calculate MBRs(Minimum Bounding Rectagles)-{R_x̂i}, x̂_i ∈ representative value
   of x-axis of A_i
   add R_x̂i into List_R
end for
merge MBRs -R_x̂i ∈ List_R which have same x-axis interval : x̂_i
for all R_x̂i ∈ List_R do
   lookup target objects by range query R_x̂i in the B+-tree indexed by representative
   location.
   insert the target object O_i into Queue Q_r
end for
for all O_i ∈ Q_r do
   calculate the distance and angle.
   if O_i is outside the fan then
      remove O_i from Q_r
   end if
end for
sort the Q_r by the angle difference and distance from O.
return Q_r
```

3.2 Multiple Interval Query Optimization in B$^+$-tree

The localized objects can be indexed by one-dimensional value which can be resolved by space filling curves such as Z-curve, H-curve, and C-curve. Space filling curve traverses entire points just once with given sequences. Since H-curve and Z-curve has good space locality, they are used in spatial queries such as the nearest neighbor query or k-nearest neighbor query. Most of spatial queries performs rectangular region query by region decomposition or approximation with sub-regions. It is inevitable that this spatial query requires multiple interval queries in transformed address space. If this spatial query is performed in separated queries, it requires redundant index, leaf node traversals on the given index tree. When data are distributed sparsely, we can eliminate unnecessary interval query by skipping the query interval by checking next item in the leaf node.

In order to solve the overhead of separated multiple interval queries over B$^+$-tree, we adopted traverse *path stack* and *data-aware interval jumps* schemes to

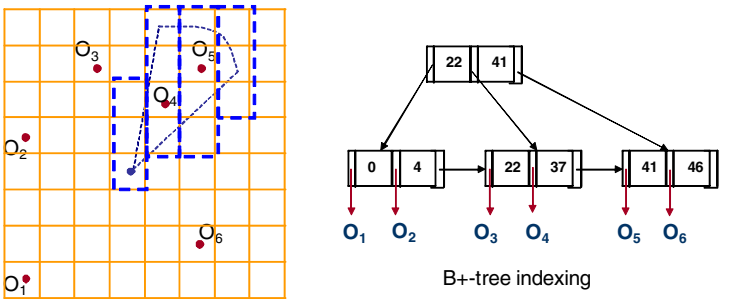


Fig. 5. MBR calculation and B^+-tree indexing for target objects

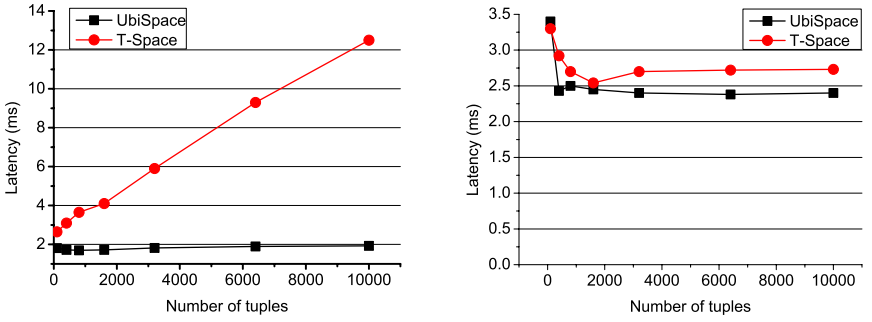
reduce the node traverse overhead. The traversed index nodes in a query are cached in the memory.

4 Performance Evaluation

In order to verify scalable middleware components, the efficient data indexing scheme and the query process are evaluated in scales up to hundreds thousands of data. The latency of implemented our middleware components are evaluated by Linux PCs. As the number of objects in the space increases, the target selection or tuple matching operation whould be the most overhead of the processing. The tuple indexing in time and name will be shown as a scalable and efficient indexing scheme. The complexity of Fan search is measured by the different node numbers in the same space.

4.1 Tuple Indexing Effect of UbiSpace

The performance characteristics of T-Spaces and UbiSpace are evaluated by average latency of read and insert operations. The average latency of 100 operations is collected since the latency of one operation is too small to collect. Each experiment is repeated by 5 times. There are two Linux PCs in the same rack which take the role of a server and a client machine. The average latency of read, insert operations is measured between the server and the client. The latency consists of network packet latency between request and response packet and query processing time. The read operation finds the newest tuple which matches the tuple name. In T-Spaces all of tuples are named by "tuple" and indexed in string key automatically. In UbiSpace, all tuples are indexed by the tuple name and sequence ID as a composite key. Fig.6(a) shows the average latency of read operations on T-Spaces and UbiSpace. T-Spaces cannot take the benefit of tuple indexing for the given tuple matching by name and id. As result, the read operation scans all of tuples in T-Spaces. This result indicates that the indexing scheme for tuples should be designed carefully to be scalable for large number of tuples. On the other hand, UbiSpace index the tuples by name and



(a) Average latency of read operation by tuple numbers

(b) Average latency of insert operation by tuple numbers

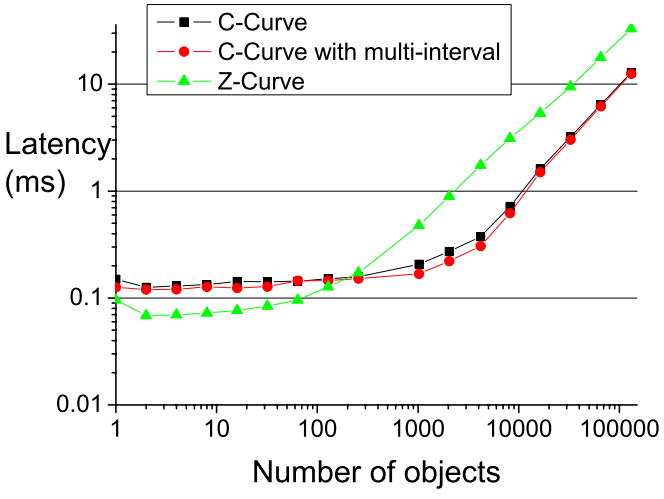
Fig. 6. Scalability of Ubispace in tuple numbers

id by default, it is scalable for the string key based tuple operations with large number of tuples in finding the newest matching tuple.

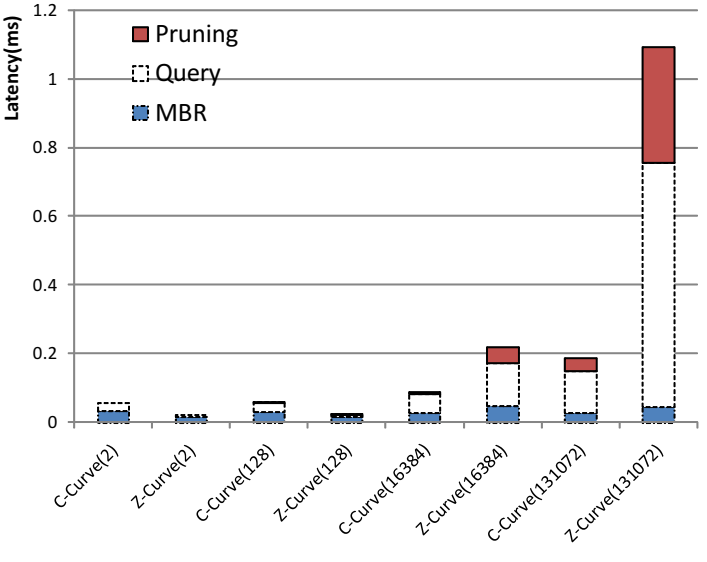
Fig.6(b) describes the average latency of insert operations on T-Spaces and UbiSpace. The overall processing consist of pruning, distance and theta calculation, and Both T-Spaces and UbiSpace update the indexing trees in bounded complexity. The latency is calculated by average of 100 insertions. UbiSpace has about 15% less latency than T-Spaces. Since the initial latency is caused by class loading of JVM, the latency of insertion in 0 tuples are the largest. Most of reduced time comes from that UbiSpace takes benefits of java object caching in the Virtual Machine. Because the inserted objects are cached in the client side, the retrieval of the object can skip the downloading of the cached object from the server side. Since T-Spaces performs deep copying of tuple object for persistency, all of the tuple objects should be downloaded from the server.

4.2 Fan Search with Space Filling Curves

Fan search algorithm is primarily designed by C-Curve space filling curve. In order to find out the relative computing overhead of fan search, we compared the overall latency of the target selection algorithm with Z-Curve and C-Curve. Fan search is performed with Z-Curve by single MBR range query over the given fan. Z-curve is implemented by linear intersection algorithm [15]. Z-curve should traverse more objects out of the fan due to discontinuity of the Z-curve. The space has 1km by 1km space. The position of the fan is selected randomly. The radius is 50 and the angle is 40 to 50 degrees. Each query is repeated 10 times. Fig.7-(a) shows latency of queries over variable node density. In high node density, Fan search with C-Curve outperforms Z-Curve due to query region approximation. Z-Curve suffers from coarse pruning objects out of the fan due to single MBR query. However, Fan search with C-curve takes more latency than Z-curve in low node density, because it has constant MBR calculation and



(a) Latency of different node density. In C-Curve with multi-interval,the path stack and skipping interval optimizations are applied.



(b) Computing time of search operations. The number inside () means the total number of objects.

Fig. 7. Fansearch target selection latency

query overhead. As an implementation optimization, C-Curve with multi interval shows the effect of the path stack and skipping intervals. However they have less than 5% improvements and no benefit on high node density.

Fig.7-(b) shows the computing overhead of target selection. The query processing consists of query decomposition into MBRs, B^+-tree interval queries, and pruning out of the fan objects. Most of latency comes from the B^+-tree interval queries. As the node density increases, the pruning process takes more time because the candidate objects are increased. As the node density goes high, Z-curve suffers from false hit on out of objects. Since Z-curve with DRU algorithm [15] has few advantages of interval skips in high node distribution in the interval query, it requires much computing time in fan query process.

5 Related Works

There are many location based interactive services in the literature such as [11], [17], and [12]. In Virtual Information Tower, mobile users can interact with visible items on the advertising columns. VIT provides a metaphor to access information which is assigned to physical location. In the view of middleware framework, VIT[11] is very similar to our system by server-client model. However VIT focuses on frameworks for information management by web-browser interface of the wearable device.

Juha at el. presented an interactive middleware by gestures.[13]. The target space is small indoors and no interaction with other users. They focus on the flexible gesture sets to extend general intuitions for VCR and lighting controls.

Round Eye[17] provides tracking continuous nearest surrounders by NS query. The NS query is similar to Fan search in query by angle and distance aspects. NS provides the one possible object for a given angle but Fan search may provide multiple objects given directions. The NS query is optimized by MBR management by considering moving objects in the query region. Round Eye achieves low computing overhead in the server by query indexing scheme. In other while, our Fan search tries to minimize given one spatial query by query region decomposition with C-Curves.

The fan search MBR approximation is similar to that of SCUBA[16]. In SCUBA, any arbitrary polygonal objects can be approximated by sum of squares in Z-address. However we designed and implemented the query by different C-Curve not Z-Curve in order to minimize false hits on the interval queries. In addition, the Z-curve should transform Z-address into x, y Cartesian point with relatively high computing overhead.

6 Conclusions

Tuple indexing and spatial query are proposed to be scalable middleware components in massive environments. The tuples are indexed by composite key to provide bounded latency on tuple matching with the newest object. Because UbiSpace manages the tuples by composite keys by default, it can provide scalable computing overhead in our services. Fan search is proposed to provide distance and angle queries for target selection. Fan search with C-Curve can provide low latency than Z-Curve in high node densities. Fan search is optimized in tree traversing by path stack and data-aware interval skipping. The paper focuses only on the indexing

schemes and query processing. We would research further on the overall system architecture to support massive environments with network limitations.

References

1. Johanson, B., Fox, A.: The event heap: A coordination infrastructure for interactive workspaces. In: WMCSA, pp. 83–93. IEEE Computer Society Press, Los Alamitos (2002)
2. Julien, C., Roman, G.C.: Egospaces: Facilitating rapid development of context-aware mobile applications. IEEE Transactions on Software Engineering 32(5), 281–298 (2006)
3. Lee, J., Lim, S.H., Yoo, J.W., Park, K.W., Choi, H.J., Park, K.H.: A Ubiquitous Fashionable Computer with an i-Throw Device on a Location-based Service Environment. In: PCAC (2007)
4. Yoo, J.W., Jeong, Y.W., Song, Y., Lee, J.P., Lim, S.H., Park, K.W., Park, K.H.: iThrow: A New gesture-based wearable input device with target selection algorithm. In: ICMLC (2007)
5. Shim, G.D., Moon, S.K., Song, Y., Kim, J.S., Park, K.H.: U-Interactive: A Middleware for Ubiquitous Fashionable Computer to Interact with the Ubiquitous Environment by Gestures. In: IFIP International Conference on Embedded and Ubiquitous Computing, pp. 694–705 (2007)
6. Lehman, T.J., Cozzi, A., Xiong, Y., Gottschalk, J., Vasudevan, V., Landis, S., Davis, P., Khavar, B., Bowman, P.: Hitting the distributed computing sweet spot with TSpaces. In: Computing Networks, pp. 457–472 (2001)
7. KAIST UFC Project, `http://core.kaist.ac.kr/UFC`
8. Freeman, E., Arnold, K., Hupfer, S.: JavaSpaces Principles. In: Patterns, and Practice (1999)
9. UbiSense, `http://www.ubisense.net`
10. Want, R., Pering, T., Danneels, G., Kumar, M., Sundar, M., Light, J.: The Personal Server: Changing the Way We Think about Ubiquitous Computing. In: Borriello, G., Holmquist, L.E. (eds.) UbiComp 2002. LNCS, vol. 2498, p. 194. Springer, Heidelberg (2002)
11. Leonhardi, A., Kubach, U., Rothermel, K., Fritz, A.: Virtual Information Towers - A Metaphor for Intuitive, Location-Aware Information Access in a Mobile Environment. In: ISWC (1999)
12. Nakajima, T., Satoh, A.: Software infrastructure for supporting spontaneous and personalized interaction in home computing environments. Personal Ubiquitous Comput. 10(6), 379–391 (2006)
13. Kela, J., Korpipaa, P., Mantyjarvi, J., Kallio, S., Savino, G., Jozzo, L., Marca, D.: Accelerometer-based gesture control for a design environment. Personal Ubiquitous Comput. 10(5), 285–299 (2006)
14. Silberschatz, A., Korth, H.F., Sudarshan, S.: Database System Concepts, 5th edn., pp. 481–500. McGraw-Hill, New York (2006)
15. Skopal, T., Kratky, M., Pokorny, J., Snasel, V.: A new range query algorithm for Universal B-trees. Elsevier Information Systems 31(6), 489–511 (2006)
16. Hogers, C.: Approximation of arbitrary polygonal objects using space filling curves versus a bouding box approach. In: Munich University of Technology Faculty for Computer Science, Section III Database Systems, Knowledge Bases (2003)
17. Lee, K.C.K., Schiffman, J., Zheng, B., Lee, W.C., Leong, H.V.: Round-Eye: A system for tracking nearest surrounders in moving object environments. Elsevier Information Systems 80(12), 2063–2076 (2007)