Building and Querying a P2P Virtual World*

Egemen Tanin[†] Aaron Harwood[†] Hanan Samet[‡] Deepa Nayar[†] Sarana Nutanong[†]

[†]NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
University of Melbourne
(www.cs.mu.oz.au/p2p)

[‡]Department of Computer Science
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland at College Park
(hjs@cs.umd.edu)

Abstract

Peer-to-Peer (P2P) systems are known to provide excellent scalability in a networked environment. One peer is introduced to the system by each participant. However current P2P applications can only provide file sharing and other forms of relatively simple data communications, and, in this paper, we demonstrate how this limitation can be bridged by indexing and querying a 3D virtual-world on a dynamic distributed network. We present an algorithm for 3D range queries as well as an algorithm for nearest neighbor queries. We also show how to build such a complex application from the ground level of a P2P routing algorithm.

Keywords: peer-to-peer, octree, virtual-worlds, spatial data

1 Introduction

The use of online virtual-worlds has great potential for government, industry, and in general public domain applications. Recent developments of 3D virtual-world gaming has attracted an enormous amount of participants. Various terrain-modelling based applications on a highly distributed scale can be considered. Yet, online multi-participant virtual-worlds do not easily scale because the communications and computations are centered around some dedicated servers and these servers require high start-up and maintenance costs.

File-sharing systems based on the Peer-to-Peer (P2P) paradigm for distributed computing demonstrate how a large number of participants can self-organize into a coherent global network. Recently, a significant

*This work was supported in part by the US National Science Foundation under Grants EIA-99-01636, EIA-99-00268, IIS-00-86162, and EIA-00-91474, and Microsoft Research.

amount of research has focussed on various indices that can be used for accessing files on P2P networks for various purposes. Examples of this work are [1, 8, 18, 20, 21, 23, 28, 31]. Some prototype P2P systems are already built on top of new low-level P2P utilities that use these indices. For example, the PAST [10] persistent storage system is built on the Pastry [23] routing and indexing service. Such systems aim to form the future of data storage and retrieval.

There are many advantages of P2P systems from an online virtual-world application point of view. First, for each participant a new host joins the system. Second, the bandwidth and processing power is not gathered on a few bottleneck servers. For example, we envision that a P2P massively-multiplayer online game can be easily deployed and can immediately scale to a large number of users. Such 3D applications form the base motivation for our work. Yet, it is not an obvious task to accommodate a virtual-world on a P2P system. Many participants will enter and leave the application along with their machines. Maintaining a complex virtual-world on such a dynamic environment in a transparent manner is very difficult. In addition, current P2P indices cannot provide the necessary functionality to perform many types of queries on 3D data that users would otherwise expect from any regular client-server based system, such as range queries and nearest neighbor (NN) queries that are crucial for spatial applications. Also, it is not obvious how to engineer such a 3D application on a P2P network from the base level of a P2P routing algorithm and using basic constructs such as sockets.

In this paper, we present a P2P solution for enabling online virtual-worlds that admit a large number of participants without placing an excessive burden on any particular host in the system. In the context of file sharing, P2P systems use an index that maps files to peers on the network. Similarly, the basis of a virtual-world is an efficient index of the objects in the world and an efficient querying system to provide for complex queries on this world. We propose an index for distributing the objects over the peers in a P2P network. If each peer is a participant in the virtual-world, then each peer is providing some fraction of the total work required to maintain and process the objects.

The rest of this paper is organized as follows. First, we present and analyze an index that is based on octrees [24, 25] and distributed hashing for enabling more powerful accesses on 3D spatial data over P2P networks (Section 3). Our work (a preliminary version is available in [30]) like some of the related work (Section 2), relies on creating a globally known mapping between the node addresses of a P2P network and the data that will be available in this network. To avoid all-to-all communications, the mapping function has to be known by all the peers of the network. Second, we present algorithms to enable range and NN queries for the P2P virtual-world. Next, we discuss the issues involved in building P2P virtual worlds (Section 4), as well as outline a layered architecture (Section 5), which we call Open P2P Network (OPeN) architecture, for the design and implementation of complex P2P applications such as our P2P virtual-world. We also make some concluding remarks and discuss directions for future work (Section 6).

2 Related Work

In recent years, spatial data management over P2P networks attracted attention from database researchers. Mondal et al. in [19] reported on their preliminary work on the incorporation of spatial data and queries into P2P networks using the R-tree spatial index [13]. Our approach is different than theirs in that we are using an octree spatial index (3D version of the quadtree index) which is based on a regular decomposition of the underlying space. This has various benefits. First, an octree decomposition is implicitly known by all the peers in the system without a need for any communications. Second, future work can benefit from

facilitating operations such as distance joins between separate data sets as the partitions are in registration whereas this is not the case for an R-tree decomposition. A partition takes an octree block and creates eight new blocks. It uses the centroid of the block to determine the new sub-blocks. Hence, the partitions of two separate data sets will always be in registration assuming that they are represented on the same data space.

Recently, [12] describes two approaches for accommodating range queries for point data that can also be applied to other spatial data sets. First, they use space-filling curves with range partitioning to map multi-dimensional data into one dimension. Hence, they can create a one-dimensional data structure for a multi-dimensional spatial data set. Next, they utilize skip-graphs [3] for efficient range queries. Ganesan et al. [11] point out that load-balance can be an issue for such direct mappings and it needs to be fixed with external techniques such as the one stated in [11]. In this case, load-balancing algorithms which can efficiently move data between peers online are utilized. In our work, we do not require explicit load-balancing algorithms which can introduce overheads. Second, they introduce a P2P version of the k-d tree [7]. The routing utilizes the neighboring cells of the grid-like data structure. For this second approach, they argue that load-balance is also an issue that needs to be addressed, although they leave its resolution for future work.

In addition, [4] described an approach based on using a Voronoi diagram to address multidimensional data on P2P networks. In comparison to a grid-like space division they subdivide the space using a Voronoi diagram. They use random graphs for routing which can connect remote peers/regions, in comparison to just using neighbors. However, the random graph approach does not have a deterministic behavior like an octree does. Also, Voronoi diagrams are harder-to-manage structures in high dimensional data due to, in part, their large space requirements as the dimension of the underlying space gets high.

3 Distributed Spatial Indexing

Spatial data consists of objects that also have a locational component. In the context of a virtual-world, a spatial object can be a wooden box or some furniture. Such objects are composed of a locational component as well as other components such as texture. Given an application domain, for example a P2P game, they may also contain values to specify their weight, monetary value, etc. A locational component can be more than just a few coordinate values. The objects can have spatial extents. Often more than one location is associated with each object. There are many ways to describe such objects ranging from their boundary, the locations that make up their interior, their minimum axis-aligned bounding box, etc. Various types of queries can be utilized to browse a spatial data set. Two of the most popular query types are range queries and NN queries. A range query can be defined as a spatial object Q, and the result of the query consists of all the objects in the database that intersect with Q. On the other hand, given a spatial data set and a query point q, we also frequently need to find the spatial objects closest to this query point.

We use peers in a P2P network to store objects of the virtual-world. Each peer is assigned the responsibility for some regions of the virtual-world and for any objects that are within that region. The regions are defined using a recursive octree subdivision of the space and objects are placed into the smallest regions which contain them entirely. Basically, we distribute an octree over a P2P network in a manner that ensures load balancing between peers. We then run spatial queries on this structure.

3.1 Distributed Hashing

Hashing is becoming increasingly popular for mapping and accessing distributed data over large networks (e.g., [16]). Although there are many methods for a distributed system to implement a distributed hash algorithm [18], we use a method as our base that has recently become widely known as the *Chord* method [28]. We use this method to facilitate for mapping of regions of space to peers in a load-balanced manner. Although, our work is built upon the Chord method, we believe that other key-based methods can also be used. With the Chord method, we can accommodate a large number of dynamic peers. In the simplest sense, the Chord method can be viewed as a routing protocol to find an item (e.g., a file given its name) in a P2P network.

Distributed hashing uses a hash function to map arbitrary data strings, i.e., keys, onto a logical space. For example, file names can be hashed to a logical space. Some spaces are multi-dimensional and some spaces are one-dimensional, e.g., as used in Chord. In both cases, each peer is responsible for maintaining some subset of the logical space. Each peer's Internet address is also hashed onto this space, called the peer's location, and then the peer is responsible for some subset of the logical space that is nearest to that location. Plus, each peer records the IP addresses of some of the peers in the logical space. This connectivity holds the P2P system together. In one dimension, the P2P network can simply be viewed as a circle. Fig. 1 depicts the Chord design and also shows other details from the Chord method.

Figure 1: The Chord method.

Each peer in the Chord maintains a table of up to t other peers, where t is logarithmically proportional to the number of locations. Each entry in the table represents an interval that is larger than the interval of the previous entry (usually twice as large). In the example in Fig. 1, the entries in the table for two peers are shown as arrows from the peers. Each interval is associated with a successor peer (termed the owner of the corresponding entry). These are depicted using directed broken lines although only a subset relevant to this example are actually displayed. These table entries are then used to jump from one peer to another, towards the peer that maintains the item for which the user is looking. After completing all the tables for all the peers in the network, we can see that the circled area in the figure shows the locations for which peer 10.28.1.5 is responsible. Similarly, each peer takes over the responsibility of a region in this space. Any file with a name that falls into a region will be owned by a designated peer.

Without giving an explicit algorithm, consider the case when peer 10.28.1.5 is trying to locate the object with key *prog1*. Hence, we are looking for the peer address who has *prog1* so that we can contact that peer and get *prog1* itself. We will now use our table of peers to trace the location of *prog1*. The querying peer 10.28.1.5 checks to see if it owns *prog1* and if not, then it scans through its table and finds the name space

interval to which the key *prog1* maps. In this case, *prog1* is in the interval defined by the third and fourth arrows, counting clockwise from 10.28.1.5, mapping to our third and fourth entries in the peer table. We contact the peer that is the successor peer for the third arrow. Note that using the fourth arrow rather than the third arrow may miss a peer that may exist between *prog1* and the fourth arrow. The request for *prog1* is now forwarded to peer 128.56.32.1, getting us closer to the destination. Now, 128.56.32.1 checks whether it has *prog1* and if not (which is the case here as *prog1* is between the second and third interval defined by the arrows from peer 128.56.32.1), then it repeats the process and hence forwards the request to peer 128.56.32.5 as the broken arrow from the end of the second interval is directed to it. This is the peer that knows who actually has the *prog1* itself. (For this example, we assumed that items are not relocated with keys when they were first hashed.) In general, it can be proven that a request to locate an item will be forwarded $O(\log n)$ times with a high probability, where n is the total number of peers in such an application of the Chord method. The Chord method has been shown to be resilient to peers leaving and joining the system. In general, the Chord [28] method has many other useful properties that makes it a suitable routing algorithm for P2P systems that are not mentioned in this paper.

3.2 Distribution of Spatial Objects

A virtual-world is formed of spatial objects in 3D Cartesian space, possibly including avatars representing the users of this virtual-world. Objects need to be inserted/deleted/modified within this virtual-world. Unfortunately, 3D objects do not have names like files. Also a simple adaptation of the Chord or any other hash-based methods can easily fail. For example, using a simple one or multi-dimensional logical space and mapping regions of the 3D virtual-world to this space (e.g., directly or using space-filling curves) and also hashing peer addresses to the same simple logical space may seem favourable for our application. The ownership of each point in the virtual-world is thereby assigned to some peer, or in reverse, each peer has a location in the virtual-world and owns the region of space that is closest to it with respect to a mapping. This method may work well when the objects in the space are uniformly distributed over the space. However, we have to realistically consider non-uniform distributions of objects for a 3D virtual-world. So, when peers are distributed over the space uniformly at random using a hash function and at the same time when objects are non-uniformly distributed, using a one-to-one mapping from the virtual-world to the logical space, this can lead to some peers receiving a greater number of objects than others.

In $\mathbb{R}^3$ space, queries can be efficiently executed by using a variant of the octree concept. The objects are indexed into the octree using some predefined subdivision rule. For example, this has the effect of exponentially reducing the average number of intersection calculations per range query. Using a suitably defined representation, each descent to a new level of the octree reduces the number of possible resulting objects by a factor of eight.

To avoid load balancing problems arising from non-uniform distributions of objects, and due to the fact that objects in space do not have names that we can immediately use for hashing, we distribute nodes of an octree that subdivides the virtual-world onto the one-dimensional Chord logical space. Any good hash function with a uniformly random distribution of keys to locations can be used to take the center point of the region of the related tree node, form a string out of its coordinates, and map it onto the Chord (i.e., SHA-1 can be used as the base hash function, www.itl.nist.gov/fipspubs/fip180-1.htm). This means that for any given node of the tree that represents some region of the virtual-world, it will be assigned to a peer selected uniformly at random. The assignment provides a good load balancing that counters non-uniform object distributions assuming that the octree subdivisions continue to a deep enough level for each dense

region of the world. Fig. 2 shows level 0 and level 1 of an octree and an example mapping of the nodes onto a logical space of 2^t identifiers. Typically, parameter $t = 160$ for the Chord space for many P2P applications. Each peer (the squares) is responsible for the elements of the logical space that come before it (inclusive) and the previous peer.

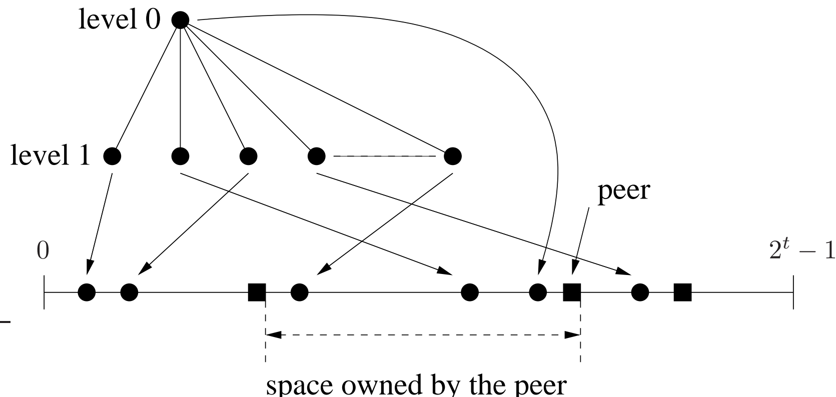

Figure 2: The first two levels of an octree mapped onto the logical space; and 3 peers that are responsible for different subsets of this space.

After our mapping of tree nodes to the logical Chord space, we can see that if a peer is responsible for a region of space, then it is responsible for query computations that may be relevant to that region of space.

In our octrees, each object is associated with the smallest octree block that contains the object in its entirety. Subdivision ceases whenever an octree block contains no objects. This is a three-dimensional variant of the MX-CIF quadtree [2, 17, 27]. For each subdivision, there is a center point, termed a *control point*, that the subdivision planes of the underlying space intersect. Specifically, we hash these control points so that the responsibility for an octree-block is associated with a peer in the P2P system. For example, $H(\text{“}(5,2,7)\text{”})$ is the location of the control point $(5, 2, 7)$. We allow the control points to be dynamically determined using a globally known function to recursively subdivide space. Hence, each peer is made responsible for some regions of space using the control points that hash to that peer. Note that, a control point is used in our algorithm like a bucket for storage of objects. Given a control point, there is a unique mapping to an octree block.

We further modify the octree concept with an alteration that forces objects to be stored and query processing to start at a level $l \geq f_{min}$ where f_{min} is said to be the *fundamental minimum* level, i.e., no objects can be stored at levels $0, 1, \ldots, f_{min} - 1$. We also use f_{max} as the *fundamental maximum* level and this allows us to prevent objects from falling lower than a level that is greater than f_{max}. Values for f_{max} and f_{min} are constant and globally known. With $f_{min} = 0$ our structure reverts to a simple octree. With $f_{min} = f_{max}$ our structure degenerates to a 3D mesh, completely collapsing the tree structure into a single level. The use of f_{min} removes the single point of failure that would have occurred had all tree operations begun at the peer that stores the root control point. It also helps further balance the load and avoid any potential overloading of peers that would have otherwise stored control points at a level less than f_{min}. f_{max} is used in various spatial data structures (e.g., MX-CIF quadtree [2, 17, 27]) to limit the depth of the tree in cases where many tiny objects exist.

For example, Fig. 3 depicts some control points and some spatial objects in a simplified 2D setting. Objects are inserted into the distributed structure by mapping them into quadtree blocks (and then hashing

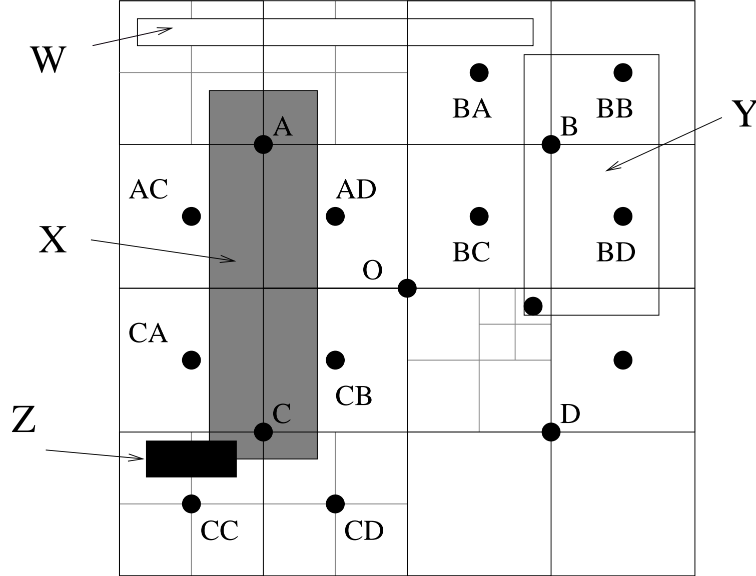

Figure 3: Spatial objects $\{W, X, Y, Z\}$ and control points in 2D

the control points of those blocks on to the Chord). At first glance, query execution starts at the root of the quadtree and propagates down through some branches of the tree, testing for the intersection of data objects with the query object as it proceeds. For a P2P system, the tree traversal becomes a sequence of peer visits. The query is transmitted from a parent block b in the quadtree (i.e., from the peer to which the control point maps) to the child blocks, i.e., to the peers that stores the child blocks, by utilizing the Chord P2P lookup method. Unfortunately, there is a single point of failure that occurs when all tree operations begin at the peer that stores the control point associated with the root. If that peer is very busy or the ownership of that control point has to change hands with peer departures then the whole tree will become unavailable for some time. On the other hand, as we have a distributed structure, there is no reason for us to start all of the operations at the root of the tree. Hence, the concept of the *fundamental minimum* level, f_{min}. In the example, in Fig. 3, f_{min} is set to 2. Hence, the object X is subdivided at level 0 into an upper and a lower part, and then further subdivided at level 1 into 8 different parts. 4 out of the 8 parts of X are stored at level 2 (i.e., associated with control points AC, AD, CA, and CB, respectively). Note that these 4 parts cannot be stored at a deeper level of the tree without having to be subdivided again, which would contradict the original definition of an MX-CIF quadtree which is the particular type of quadtree we use for this example. The remaining 4 parts are inserted at the next (deeper) level as they are smaller. In particular, they are stored at control points AAD, ABC, CCB, and CDA, respectively, although, in the interest of keeping the figure simple, they are not shown in detail. Again, these parts cannot be inserted at a deeper level of the tree as this would require that they be subdivided. As another example, the smallest part of object Y is inserted at level 4.

3.3 Spatial Algorithms

Nodes of the tree are distributed uniformly at random over the peers using the base hash function. Hence, we distribute the tree operations, *insert*, *query*, and *delete*, by branching and *delegating* control to the peers that are responsible for the control points from the octree. Parts of an operation can work in parallel on

different branches of the distributed octree as they map to different peers through different control points.

Each control point u has the following data structure associated with it:

$$D(u) = (\{d_1, d_2, \ldots, d_8\}, list).$$

Downward counts $d_i \in \mathbb{N}$ are used to indicate the number of objects that do exist at or below child i. The *list* is a list of objects that intersect the region $R(u)$ and that could not fall any further down the tree. The default values for $D(u)$ are:

$$\{\overbrace{0, 0, \ldots, 0}^{8}\}, empty$$

which means that there are no objects below u and no objects stored at u.

A peer invokes InitiateInsertion (object X) to insert an object into the tree. We use Delegate(u)→Func() to mean that a peer sends a control message that invokes Func() on another peer that stores control point u. A Delegate() operation is similar to creating a new thread of control. Procedure InitiateInsertion (), shown below, concurrently delegates procedure DoInsert() over all control points at level f_{min} that intersect with the object. Procedure DoInsert() delegates recursively and also concurrently through the distributed tree until the object is inserted. Note that $D(u).field$ is used to access/set $field$ of/for $D(u)$. In our presentation, we make use of the following auxiliary procedures and notation. $Ints(X, Y)$ computes the intersection of X with Y. $R(u) = (x_1, y_1, z_1, x_2, y_2, z_2)$ denotes the region defined (and also controlled) by control point $u = \left(\frac{x_2+x_1}{2}, \frac{y_2+y_1}{2}, \frac{z_2+z_1}{2}\right)$. $L(u)$ denotes the level of control point u. $C(u, i)$ represents the i-th child of control point u, where $i = 1, 2, \ldots, 8$.

```
InitiateInsertion(object X)
{
  control point list G:={}
  Subdivide(X,root,G)
  for each u in G do
      Delegate(u)→DoInsert(X,u)
}

DoInsert(object X,
          control point u)
{
  if (X is not within exactly one R(C(u,i))) or (L(u) = f_max) then
     set D(u).list to include X
  else
    for i:=1 to 8 do
        if (Ints(X,R(C(u,i))) is not empty) then
           increment D(u).d_i by 1
           Delegate(C(u,i))→DoInsert(X,C(u,i))
}
```

Procedure Subdivide(X,$root$,G) is called with initially $G = \{\}$ when inserting X (and also in other tree operations). This initial subdivision of an object down to level f_{min} is performed by the recursive progress of Subdivide(): The procedure modifies G by adding control points to it. First, *root* is used to call this method where $L(root) = 0$ and $R(root)$ is a bounding box that bounds all data and query objects. For this algorithm

it is assumed that all X's will be contained within $R(root)$. The list of control points at level f_{min}, G, that intersect with X is computed locally and processing is then delegated to the peers that store these control points using the Chord method.

If DoInsert() is invoked on a control point that does not exist, then the control point is implicitly allocated with default parameters. Delegation is sent using the Chord method and so ordinarily it takes $O(\log n)$ messages to reach its destination where n is the number of peers in the system. Since each node of the tree has a fixed number of children, we allow each node to maintain a cache of addresses for its children and thereby reduce the delegation message complexity to $O(1)$ (as we will no longer need to use the traversal algorithm of Chord for each child). This is true only when stepping through the tree. The number of peers that are initially contacted is a function of f_{min} and can consist of a large number of addresses. Hence, we do not allow caching of the f_{min} level peers' addresses.

Procedures for InitiateDeletion () and DoDelete() are almost identical to InitiateInsertion () and DoInsert() and so an explicit listing is not given here. The essential difference is that objects are removed from $D(u).list$ instead of being added to it and that $D(u).d_i$ is decremented instead of being incremented by 1.

3.3.1 Range Queries

Peers can initiate a range query with the InitiateRangeQuery() procedure. Similar to insertion (deletion), this procedure takes in an object (named Q for query in this case) and then it finds the control points at level f_{min} and delegates the query to the relevant peers. The results of a query can then be sent back to the initiating peer.

We use the term *hit* to indicate that an object intersects a query. A query which covers multiple control points may return the same object a multiple number of times because some objects may have been copied into multiple control points when forced down to a level at or below f_{min}. Thus we have to eliminate such superfluous hits, i.e., the same object intersecting with the same query multiple times at different peers concurrently.

```
InitiateRangeQuery(query Q)
{
  control point list G:={}
  Subdivide(Q,root,G)
  for each u in G do
      Delegate(u)→DoRangeQuery(Q,u)
}

DoRangeQuery(query Q,
             control point u)
{
  intersect objects in D(u).list with Q
  send results
  for i:=1 to 8 do
      if (Ints(R(C(u,i)),Q) is not empty) and (D(u).d_i > 0) then
         Delegate(C(u,i))→DoRangeQuery(Q,C(u,i))
}
```

3.3.2 Nearest Neighbor Queries

For NN queries, [15] contains a comprehensive analysis of various similarity search algorithms in metric spaces. Their main contribution to the field is through a priority queue based ranking algorithm that can find the results of a ranking query in an incremental fashion. Formally, ranking is a more general case of the NN query where a user can ultimately retrieve all the objects of a database in the order of their distance from a query point. The algorithm works on many classical spatial indexing methods including octrees. The entries of the priority queue consist of spatial objects as well as the blocks of the space that the underlying spatial data structure uses. The first entry in the queue is the root of the data structure. This entry is retrieved in the first iteration of the algorithm and the children of the root are then inserted back to the priority queue using their distance from the query point. The next iteration of the algorithm retrieves the new highest priority child/block or object. Hence, in this fashion, at each iteration of the algorithm, the element with the smallest distance is removed and visited and its children are inserted into the queue. It is important to note that eventually all the spatial objects will be retrieved as they are also inserted into the queue by the previous iterations. We use the same incremental approach in our P2P NN algorithm.

Very recently, [6] focuses on the problem of executing P2P NN queries on metric spaces using a distributed data structure. They use the data structure called GHT (Generalized Hyperplane Tree) introduced in their previous work [5]. This data structure is mainly for non-spatial data and hence cannot be directly applied to the spatial domain. Nevertheless, they mention that exploiting the parallel nature of P2P networks is an issue and their NN algorithm proposes an efficient parallel approach. They state that the priority queue approach that we prefer cannot be easily adapted for parallelism in their case. Hence, they use a radius estimation-based approach to create an ever-increasing search circle. They parallelize this front. In our work, we parallelize the priority queue based NN work [15] for obtaining an efficient P2P NN algorithm.

A NN algorithm using R-trees in sensor networks is also presented by [9]. NN queries are performed in a distributed fashion using a self-stabilizing P2P indexing structure called the peer-tree. They mention the importance of parallelizing the NN search with the main limiting constraint being the power consumption on sensors for their case. Unfortunately, they do not present a detailed and general solution for this issue in their work.

Using our P2P octree-based index, we now present our priority queue based NN algorithm. We use the base concepts from [15]. Yet, a direct implementation of it has disadvantages on a P2P platform. Primarily, we do not need to run the algorithm in a sequential manner as we do not have to have a single thread of control in a P2P setting. For a networked system, the delays per contact to a peer and in a sequential manner can add up to a significant amount.

For P2P settings, in theory, we can send a message to all of the peers in the network to find the NNs simultaneously. But realistically, if other peers take the same approach for their queries, then we will be creating an all-to-all communication mechanism in our P2P network. Hence, for a system with millions of peers, sending messages to all the peers in the network is not a practical approach. So, the question is what is a reasonable amount of parallelism that we can harvest from the independent peers of a P2P network?

Our heuristic that aims at this objective is to maintain a query processing front of all those control points, hence blocks of the spatial data structure, that are in the priority queue and still have the possibility of returning a closer object than the top block of the priority queue. So rather than a priority queue with a single point of entry, we maintain a front of multiple entries that are being processed in parallel. This heuristic attempts to maximize the parallelism that we can harvest on a P2P network from a single peer's point of view, while avoiding having a single peer send messages to many peers which would be redundant

for a NN computation.

A NN query, q, is first initiated on a single peer in the P2P network. This peer maintains the priority queue of octree blocks (mapping to a control point each) that are being processed for the query. To process a block, we have to make a contact from this query initiating peer to the peer that owns that block, i.e., the control point. Hence, in our parallel algorithm, instead of contacting just the peer corresponding to the top entry of the priority queue, we contact a multiple number of these peers. Assuming that $f_{min} = 0$, the query starts at the root and initially there is only the root block in the priority queue. Hence, we contact the peer that has the root block (control point) and wait for a response. In the case of our octree, this block may contain objects (the ones that lie on the subdivision lines for the blocks of the next level), and hence the closest one to the query point can be the first nearest object. So, objects can be returned back to the query initiating peer. Also, the peer that is responsible for the root block knows how many objects the child blocks have. This information is used to return the children that have objects and hence can contribute to the NN query. Next, the query-initiating peer inserts these blocks into the priority queue along with any objects that are returned and proceeds with the next iteration of the algorithm. In the next iteration, rather than contacting only one peer, all those blocks and hence the peers that maintain them that still have a possibility of returning a closer neighbor have to be contacted. The decision for selecting which ones should be contacted is important and guides the behavior of the algorithm. Hence, in comparison to the original priority queue based algorithm [15], there is an additional criterion that controls the flow, namely the *Worst Case*. Abbreviated as WC, this criterion is used to ensure that the relevant peers that can still help in finding a closer neighbor for the next NN are contacted. This algorithm works, without any alterations, even for $f_{min} \geq 1$, where there are many blocks in the queue as soon as the algorithm starts.

Our algorithm, assuming that the current NN is at distance m (i.e., the first spatial object in the priority queue), instead of removing elements from the priority queue one at a time for processing, computes the maximum distance $\mathrm{MaxDist}(q,t)$ at which an object can be found in the top element t of the priority queue and then processes all elements e in the priority queue whose distance $\mathrm{Dist}(q,e)$ from query point q is less than $\mathrm{Min}(\mathrm{MaxDist}(q,t),m)$. This is the WC criterion. Hence, with this criterion we look at two pieces of information: i) the first spatial object in the priority queue that, in the worst case, can be the next NN (this object can be held in a separate buffer or a pointer to this object can be utilized for efficiency) ii) the maximum possible distance from the query point to an object in the top element of the priority queue. Alternative formulations for (ii) are given in the literature in the context of sequential algorithms for various spatial data structures and they can be easily used with our algorithm to parallelize the NN search. For example, an alternative (ii) can be, which might yield a tighter criterion when the elements in the priority queue are minimum bounding boxes, the maximum distance $\mathrm{MaxNearestDist}(q,t)$ at which a NN of q must be found in the bounding box t of the top element in the priority queue. This metric is called the MinMaxDist by [22] and MaxNearestDist by [26]. In general, the difference between the methods of [22] and [26] is that the former is only useful for depth-first nearest neighbor finding and for just one neighbor, whereas the latter is useful for both depth-first and best-first NN finding for arbitrary values of k (i.e., the k-th NN).

When a NN query is initiated at a peer, the peer calls InitiateNNQuery. This method, shown below, inserts all the blocks (control points) at the f_{min} level into the priority queue. In our prototype, the queue is implemented as a sorted linked list that also enables parallel access to the rest of the queue, rather than just to the top element of the queue. Other, more efficient, implementations of this in-memory data structure on the query-initiating peer are also possible but probably unnecessary as the time spent sending messages among the peers is several orders of magnitude greater than the time for the in-memory operations. The

order in the list is based on the distance of the octree block associated with each control point from the query point. As we do not want to contact all the peers with these control points at once, the peer that the query point q lies in is contacted along with all the others within the initial WC distance to start the algorithm. In the beginning, this is equal to the maximum distance between the query point and the borders of the block that contains the control point in which q lies.

When a peer receives a NN query that was initiated on another peer, it calls DoNNQuery which determines whether it has any objects and also checks whether any of the child control points have any objects. Finally a message is sent back to the query-initiating peer containing the objects and the valid children, if any.

At the query-initiating peer, ReceiveNNMessage is used to handle messages that are returned by other peers in a mutually exclusive manner. If a peer returns objects or its child control points, then these are inserted into the sorted list corresponding to the priority queue and they are accessed in the next iteration. It is important to note that the messages from different peers can return in a sequence that is different than the original sequence and the algorithm would still work in the initially intended manner.

SendMessagesWithin() is the method used to contact all control points from whom we are not already waiting for a message and that fall within the new WC distance. Obviously, elements of the priority queue that are not control points/blocks but just spatial objects are returned to the user when they become the top element. We can wait for a user command to continue with the next iteration when a nearest object is found. This makes the algorithm a truly incremental ranking algorithm from the user's point of view. For our NN algorithm, delegation is not utilized as ultimately the query-initiating peer still has control of the progress of the query unlike the range queries presented earlier (3.3.1).

UpdateWCDist is the method used to update the distance of WC when a message is received. This is done by examining the current top element in the priority queue (and the first spatial object available in the queue if one exists as there is no need to look further than this object for the next neighbor).

```
InitiateNNQuery(query q)
{
  priority queue pqueue:=GetSortedControlPoints(q, f_min)
  control point c:=FindControlPoint(q, f_min)
  WCDist:=MaxDist(q, c)
  SendMessagesWithin(WCDist)
}

DoNNQuery(control point u)
{
  msg:=CreateReplyMessage()
  msg.Put(D(u).list)
  for i:=1 to 8 do
        if (D(u).d_i > 0) then
            msg.Put(C(u,i))
  SendMessageBack(msg)
}

Synchronized ReceiveNNMessage(message msg)
{
  for each object X in msg.list do
       pqueue.add(X)
```

```
    for each control point u in msg do
        pqueue.add(u)
    pqueue.remove(SenderOf(msg))
    WCDist:=UpdateWCDist()
    SendMessagesWithin(WCDist)
}
```

Fig. 4 shows how a NN query proceeds using our algorithm on a simple example using our P2P index. To simplify the presentation, we use a 2D setting rather than a 3D setting. The dark dots in the figure represent the 16 f_{min} level control points for $f_{min} = 2$. Three objects X, Y, and Z are represented by shaded rectangles. According to the rules defining our trees and with $f_{min} = 2$, object X is stored at level 2 with control points - BA, BB, BC, and BD. Object Y is stored at level 3 with control point CBA and object Z is stored at level 4 with control point $CBCC$. Note that not all the control points are shown on the figure for simplicity. The f_{min} level control point containing the query point q is CC. We show two WC distances that are computed by the algorithm.

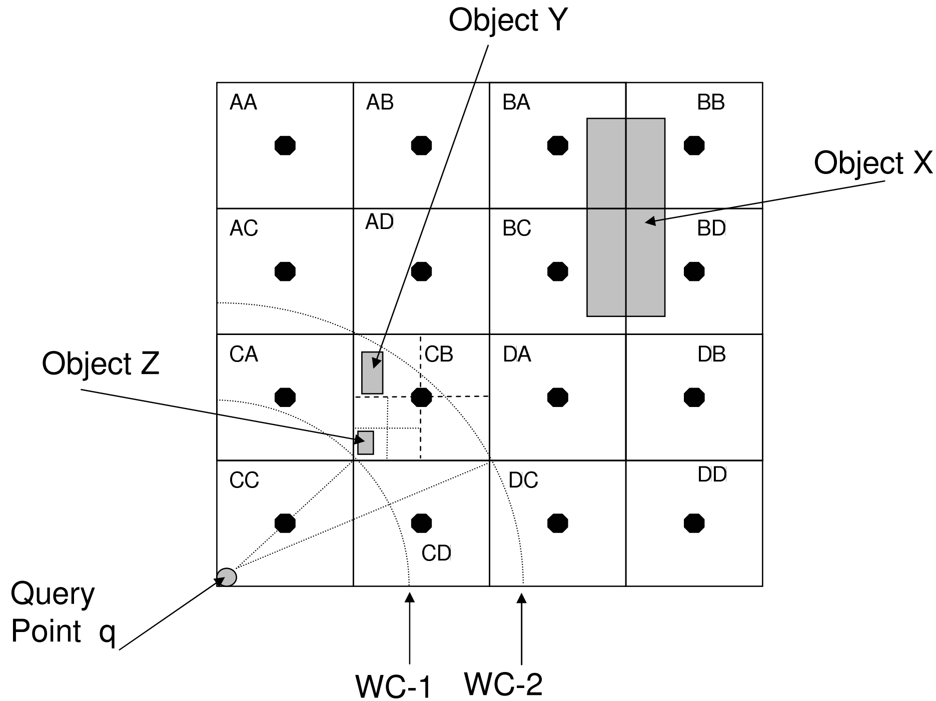

Figure 4: Spatial objects, control points, query point, and two WC fronts in a quadtree with $f_{min} = 2$.

First, the query-initiating peer runs the InitiateNNQuery method that will compute the f_{min} level control points, insert them in a priority queue (with respect to their distance from the query point) and compute the control point in which the query point lies (again at level f_{min}). Next, it computes the first WC distance (as shown in the figure) and sends messages to all peers that own control points that are within the first WC distance. These are CC, CD, and CA.

All the peers receiving a message from the query-initiating peer will run the DoNNQuery method and return the objects they have and the control point information for all the child control points that their

records show as having some objects. In our case, unfortunately, none of the initial contacts contain any spatial objects. They also cannot return any of their children as they also do not have any data (i.e., they are not yet initialized by the P2P index).

The query-initiating peer, after receiving the information from the peer that owns CC that no useful data can come from that region has to act on this knowledge. The method ReceiveNNMessage is used for this purpose. Once we remove CC from the top of the priority queue on the query-initiating peer, CD (or alternatively CA) becomes the next control point to be processed. It is important to note that the messaging scheme works in an asynchronous manner and hence peers that own CA or CD may have returned an answer to the query-initiating peer first. In this case, this example can be traced in a similar manner, while we will continue our analysis with CC being the first peer to respond. In ReceiveNNMessage, the query-initiating peer uses the new top element, CD, and expands the WC distance. It then sends messages to all the peers within the second WC, shown in the figure. These peers are the peers that own control points CB, DC, and AC.

At this stage, we can see that CB is the only control point that contains any data, i.e., among the peers that were contacted. We see that there are two objects that are located within CB both maintained by the children of CB. The progress of the distributed algorithm will at this stage start converging on a single spatial object, i.e., an object that can become the top entry for the priority queue. This will be our first NN, Z. It is also important to note that while discovering Z as the first NN, the peers that own control points AC and DC would already have been contacted in parallel, thereby improving the performance for the next NN of the ranking process. In fact, the peer owning the control point CBA would have already returned Y, the next NN which would be placed into the priority queue.

In summary, the WC criterion should present a shrinking behavior when new, smaller and closer blocks are investigated until an object is retrieved from the top of the queue. But it can also expand without requiring a change from our algorithm. This facilitates the continuation of the algorithm for the purpose of ranking as well as dealing with f_{min} level control points when we start the process. In fact, the expansion capabilities of the WC has an additional benefit for P2P systems that is not shown in the figure. As we cannot lock the whole P2P octree for a single NN query, it is possible that while we traverse the quadtree, a delete request can remove the spatial objects for a block that has been previously inserted into the priority queue, thereby creating a need for expansions in the WC distance to be able to continue with the algorithm.

3.4 Analysis

In this section, we present the complexity analysis for our work. Recently, we have presented the simulation results for our index for a 2D application. This short paper [29] utilized a realistic 2D real-estate application where users can form a P2P network for buying/selling their properties over the Internet. Most importantly, we see the importance of f_{min} value selection for our index. A low value for f_{min} can increase load on some specific nodes of the tree (also increases the risk of single point of failure) and a high value for f_{min} creates redundant messages by converting the tree structure into a grid in 2D.

For the complexity analysis, we let the constant t_m denote the time to transmit a single control message for a query (assumed to be the same for all the peers for simplicity) and n be the number of peers in the system.

The cost of a range query is stated as: $(f_{max} - f_{min} + 1 + (\alpha . \log_2 n)).t_m$. This formula represents the traversal of the distributed tree [30]. The constant $\alpha = 0.5$, for the Chord traversal, is observed experimentally [28]. This value shows the portion of the number of links that will be traversed in a real-life

situation for a lookup in the circular Chord name space. Hence, the propagation of a range query, through the tree, requires $O(f_{max} - f_{min} + \log n)$ steps for n peers. We note that only the part of the distributed octree lying between levels f_{min} and f_{max} is traversed. Due to caching, this traversal will not require Chord lookups. For the f_{min} level control points, $O(\log n)$ steps is unavoidable as we have to use the Chord traversal to find the peers that maintain these control points. Caching a large number of control points in every peer is not practical. The cost of a lookup on Chord is shown to be $O(\log n)$ with high probability in [28]. We also note that multiple branches of the distributed octree will be involved in a large range query and that they will be processed in parallel on different peers corresponding to different blocks of the octree. This analysis assumes that control messages between peers are too small to cause any congestion on any single peer and the CPU processing time per peer is negligible in comparison to network latency.

We also compare our NN algorithm with the sequential algorithm presented by [15] on our P2P setting. To further simplify the analysis, we assume that we have a perfect octree, we again do not need to use the Chord method for tree traversals (i.e., there are no cache misses), the user runs the ranking until completion, $f_{min} = 0$ (i.e., all tree operations start from a root control point/peer that can also be cached on other peers in the system), and the bounding box of the data is a cube.

The sequential version of the algorithm will have to visit every node of the tree in a sequential manner to rank all the objects in the P2P system as we have a perfect octree. Therefore the propagation of the query has a complexity of $O(8^{f_{max}})$. This is basically equivalent to the total number of nodes of the distributed octree.

For our parallel algorithm, the closest NN will be found in $O(f_{max})$ (i.e., the height of the tree) steps while each level is visited in parallel. The algorithm starts at the root level and gradually focuses on one small part of the tree while restricting the WC. Eventually we will reach the leaves. The difference from the sequential algorithm is that at every step of the descent in the traversal of the tree, from one level to another, nodes will be visited in parallel, and many intermediate nodes will have already been inserted into the priority queue. These nodes will also be used later on for ranking, i.e., finding the remaining objects in the order of their distance from the query point. In fact, the blocks that contain the next few neighbors, given our assumptions, may have already been visited while processing the current neighbor and the spatial objects that they contain may have already been inserted into the priority queue. When we process the objects and blocks that have not been visited, all other blocks within a WC will also be automatically visited, again in parallel. This, given our initial assumptions for a perfect octree, will lead to a wave of parallel expansions of the octree blocks from the query point with each ranking request. This wave moves discretely and there will be $O(8^{f_{max}/3}) = O(2^{f_{max}})$ increments (similar to the WC's given in Fig. 4). This is because there are as many increments as there are number of leaves in one dimension of the octree (i.e., the leaf level can be considered as a 3D mesh and there are as many steps as the number of cells in one dimension of this mesh). At each increment, we will need to process only a single level of the octree to reach to the neighboring leaves. Hence, the overall complexity of our algorithm will be $O(2^{f_{max}})$. This is $O(4^{f_{max}})$ times faster than the sequential algorithm assuming the ranking algorithm runs to completion.

4 Building P2P Virtual-Worlds

Fig. 5 shows a sample generic P2P virtual-world application that uses our algorithms and was built in our labs (www.cs.mu.oz.au/p2p). There are two peers and hence two application windows in this figure. Each peer designed and inserted an object (on the top right corner of each application window) to the virtual-

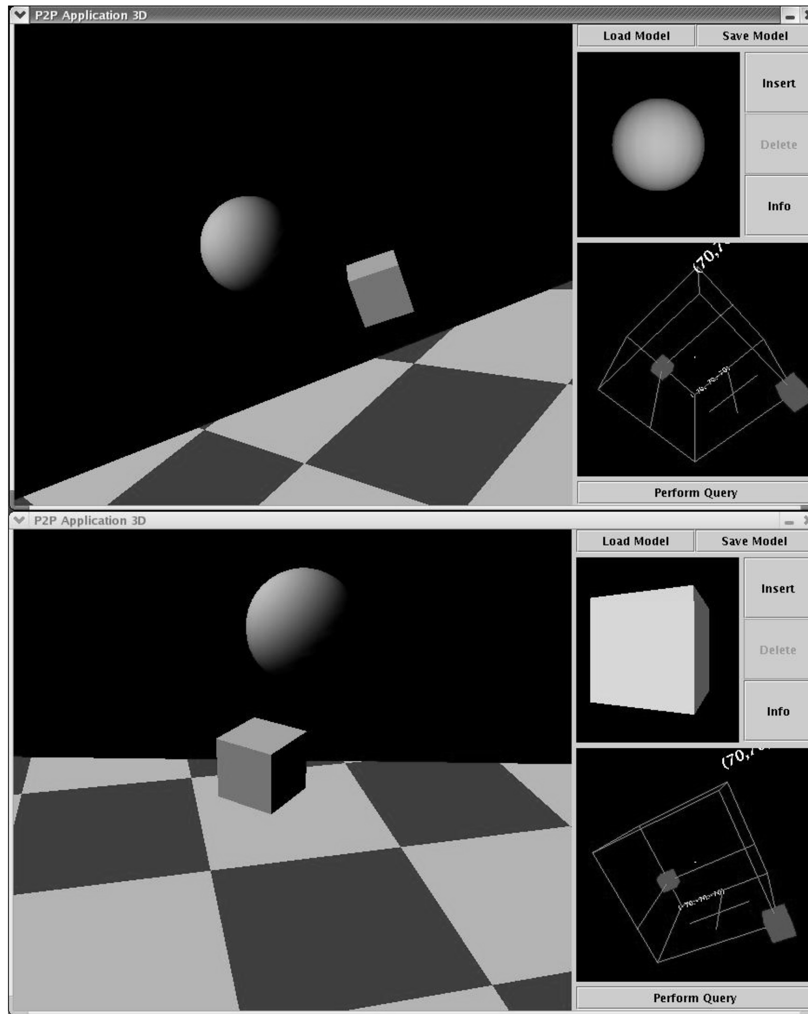

Figure 5: Screen shots of two peers, each viewing the same scene but from different view points. The sphere and cube are different objects in the virtual-world inserted by different peers.

world. The boundaries of the virtual-world are shown on the bottom right corners of the applications as an overview. Two small cubes can be placed freely on this overview to define a range query region on the whole space as shown in the figure. The objects in the query regions are then returned to the large center canvases after the distributed search/query operations complete. Each peer sees the same scene from a different view. We have also implemented a 2D virtual-world, that represents a metropolitan city, as another demonstration of our work. In this prototype, people can post events on a P2P network using a city map and they can also run queries such as range and NN queries to find events posted by other users. This prototype shares the same principles as the 3D prototype but the indices and algorithms work in 2D rather than in 3D with some modifications.

It is not obvious how even such generic virtual-worlds can be implemented on a P2P network. To help this process, we defined and used a layered software architecture, called the Open P2P Network (OPeN) architecture [14], for designing and implementing complex applications such as virtual-worlds on P2P networks. Applications are developed without a need to consider P2P protocols and vice versa; P2P protocols are interchangeable without changing the application semantics. The OPeN architecture provides an explicit

object-oriented solution to data processing on a P2P network. Applications can be developed by using various common primitives. This makes application development simpler. The OPeN architecture ensures that applications adhere to the P2P paradigm so that they are intrinsically decentralized and autonomous.

The OPeN architecture consists of three layers; depicted in Fig. 6 as the *Application* layer, the *Core Services* layer, and the *Connectivity* layer. The Core Services layer ensures consistency and easy development for a large range of applications. The Connectivity layer enables P2P protocols to be developed transparently.

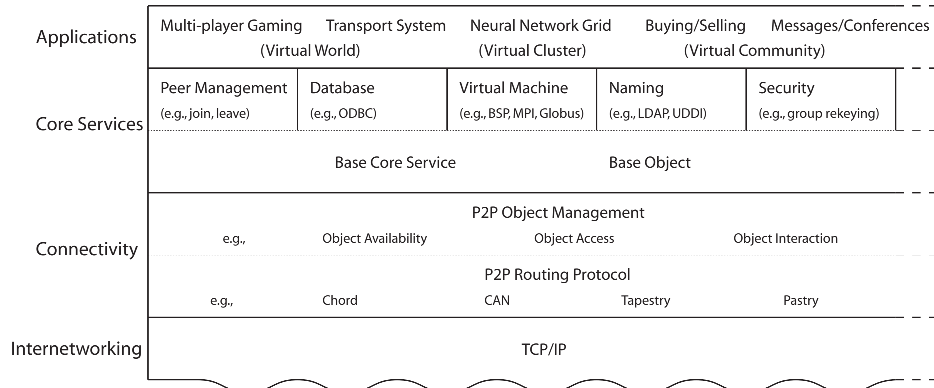

Figure 6: OPeN layered architecture.

The benefits of the architecture can be itemized as:

- Abstraction: The architecture allows application developers to focus on application specific tasks with at most a trivial understanding of the details of the underlying P2P protocols. We also have a set of re-usable services (e.g., data management, directory services, and virtual machines), with interfaces which application developers can use to expedite the application development process.

- Interoperability across protocols: A P2P application should be able to be a part of more than one P2P network, each using a different protocol, or the design should allow for easy replacement of protocols when better technologies emerge. Since different protocols provide different qualities of services it is unlikely that a single protocol will efficiently provide all the qualities required by complex P2P applications. Hence, the architecture provides the flexibility of changing the P2P protocol used by the application with no changes to the application. Another consideration is the ability for applications to choose the protocol at runtime based on the quality of service required by the end user. In the current P2P application development paradigms there is tight coupling between the application and the underlying P2P protocol and applications are limited to using a single P2P protocol. This is currently a serious limitation for developing complex applications.

- Delegation: One of the main advantages of P2P applications is their decentralized nature. To achieve true decentralization, delegation of processing is passed along with the objects from a peer to another peer.

- Reliability: It is widely accepted that P2P protocols allow peers to be autonomous, in the sense that they may join and leave a P2P network independently. This autonomous behaviour of P2P applications poses a significant challenge to application developers in terms of reliability: the application should not be impacted by peers leaving or joining the network. Furthermore, the reliability should scale,

working both with small and large numbers of peers. P2P applications should handle peers joining and leaving the network transparently to the application user. P2P applications should not be concerned with the techniques used to provide autonomic behaviour. All of this behavior is primarily embedded in the protocol layer.

- Security: While security may be provided in various ways (e.g., secure sockets, anonymizing networks, and data encryption), it is not clear how these approaches integrate to provide a secure P2P application environment. This is particularly so when considering a public P2P network upon which multiple, potentially adversarial, parties execute their applications. A P2P application development infrastructure should provide a means for applications to control the level of access they provide. Protocols that facilitate trust are important and could be developed and plugged independently of the application. Encryption algorithms used at lower layers can be easily updated without requiring a change at higher layers.

- Maintainability: Garbage collection is essential for systems that share data from many sources. Data in the P2P network needs to be deleted if it is no longer in use. Traditional distributed systems use a registry of objects, which is inappropriate in the P2P paradigm (unless such a registry can be completely decentralized and operate according to other P2P principals). A known approach is to delete data if it has not been accessed within a given period of time. The affect of this approach on complex P2P applications is not clear. Applications should be able to define the data management strategy and garbage collection should be managed transparently to the application. Garbage collection details are allowed to be hidden in lower layers while simple control mechanisms can be given to services and application layers.

- Validation: Another challenge that P2P application developers face is finding the means for validation. Debugging and scalability testing are important considerations for P2P applications. However, testing of such applications in a real world setting (in contrast to a client-server application) with a large number of simultaneous peers is not practical. Therefore, the architecture should intrinsically support application testing by various simulation mechanisms. Lower layers can be easily tested independently from the applications.

- Client/Server compatibility: Most P2P applications today allow the application process to participate either passively or actively in a P2P network. Active participation in a P2P network requires acting as a server and allowing connections to be made to/from other peers in the network. It generally involves providing some storage space and computation cycles for general use by the peers on the P2P network. Passive participation allows a P2P application to access a P2P network, as a peer-client, (essentially as a client connecting to a local peer) without being exposed to the peers in the network. This may also be the preferred mode of use by the P2P system to avoid certain connection overheads. When P2P applications are allowed to participate in more than one P2P network simultaneously, it may be necessary to actively participate in some networks and passively in others: hence we support both of these modes of operation simultaneously and transparently at lower layers.

- Extendibility: As new complex applications are developed using the architecture, there will be a need to support new services that will help develop these applications. It will also be necessary to add these services to existing peers so that the peers can provide the new services. Therefore, extendibility, by adding new services and protocols, is an essential property.

We designed our architecture layers to facilitate these benefits. Our architecture can be viewed as a projection of the OSI model onto the TCP/IP model for the Internet for complex P2P applications. The OPeN architecture can be described as the layers that sit on top of the Transport Layer, i.e., a P2P session is loosely described by: (i) connecting to the overlay P2P network, (ii) obtaining/supplying some service, and (iii) disconnecting from the P2P network. These layers are described in greater detail in Section 5.

5 Architecture Layers

5.1 Application Layer

Applications define and allocate objects. More specifically, objects contain data and methods. The use of objects allows applications to *delegate* processing from one peer to another. Applications can invoke methods of objects and objects can execute autonomously to invoke methods of other objects, i.e., objects can interact with each other. Fig. 6 proposes a number of example complex applications in the Application layer. General classes of applications are shown in parentheses. For example, multi-player-gaming is in the domain of virtual-worlds.

5.2 Core Services Layer

The Core Services (CS) layer provides a variety of flexible services that support applications. The example CSes in Fig. 6 are *Peer Management*, *Database*, *Virtual Machine*, *Naming*, and *Security*. CSes are built on top of a Base CS, also shown in Fig. 6, and make use of a Base Object. The Base CS and Object are used to ensure that all CSes are implemented with a consistent view of the underlying P2P networking operations. The CSes shown in Fig. 6 are not meant to be an exhaustive list. New CSes can be added if it is found that existing CSes are inappropriate for the required application support. We list these CSes as examples of key CSes that either cover a large variety of complex applications or are essential to include for most applications. Applications can use the interfaces of the CSes for easy development. Existing standards can be preserved, e.g., ODBC. CSes can also utilize delegation to handle their tasks. For example, a range query can spread across many peers without central control. In our case, we implemented a simple CS to insert/delete spatial objects to a virtual space. The generic virtual-world application is built using this simple interface. The service itself implements the distributed octree index but it uses the Chord method implementation at the Connectivity layer to store objects on control points. A range query is a spatial object that can traverse the tree recursively and visit control points (hence peers) to find intersecting spatial objects. The Chord protocol can be replaced with another protocol without changing the higher layers.

5.3 Connectivity layer

The Connectivity layer serves the purpose of separating the P2P protocol from the rest of the architecture. Different protocols (e.g., Chord) provide different qualities and it is unlikely that a single protocol can efficiently provide all of the qualities. The P2P Object Management sub-layer provides the basis for a complete object-oriented approach to P2P applications. Objects are used to hide all of the underlying activity. A Base Object is used to represent the simplest kind of object that all of the underlying P2P protocols can work with. Remote Method Invocation is used for applications to interact with objects and for objects to interact with other objects. This abstraction allows programmers to make use of familiar object-oriented

techniques. The separation provided by the P2P Routing Protocol and P2P Object Management sub-layers allows protocols to be changed without requiring changes to occur at the higher layers. It furthermore allows protocols to be bridged.

6 Concluding Remarks and Future Work

The use of online virtual-worlds has great potential for government, industry, and in general public domain applications. Yet, they suffer serious scalability problems when implemented in a client-server manner. P2P networks are becoming a common form of scalable online data exchange. But in P2P networks, users cannot perform many types of queries on complex data, such as 3D data, and it is not obvious how to build a complex application such as a 3D virtual-world on a P2P network easily. In this paper, we introduced and analyzed a distributed octree-based index and algorithms for enabling more powerful accesses on 3D spatial data over P2P networks. We also presented an architecture for building complex 3D applications over such dynamic networks. Basically, we indicated how a generic scalable 3D virtual-world can be implemented without a server. We believe that our work can be applied to various multi-dimensional spaces and using other methods than the Chord method. As future work, we are currently investigating updating our indices to facilitate handling of moving objects. One direction for future work is replication (for especially f_{min} level control points/peers) for robustness. This can be achieved at the protocol level. The obvious issue with replication is efficiency and coherency when maintaining multiple replicas and hence finding an optimum level of replication given a control point. Another direction is security. Although P2P systems are known to be resilient to malicious activity due to decentralized control, a P2P virtual-world, unlike file-sharing systems, has the disadvantage of having to maintain one unified view of the application. Hence, any peer, where the authenticity of code is not frequently checked, can use malicious code to insert objects to the virtual-world that are potentially disruptive to other participants' activities. Hence, enforcing the hard-coded rules of the virtual-world, e.g., in a P2P game, would be an interesting research topic.

References

[1] K. Aberer and M. Punceva. Efficient search in structured peer-to-peer systems: Binary v.s. k-ary unbalanced tree structures. In *Proceedings of the International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (held in conjunction with VLDB)*, Berlin, Germany, September 2003.

[2] A. Aboulnaga and J. F. Naughton. Accurate estimation of the cost of spatial selections. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 123–134, San Diego, CA, February 2000.

[3] J. Aspnes and G. Shah. Skip graphs. In *Proceedings of SODA*, pages 384–293, Baltimore, MD, January 2003.

[4] F. Banaei-Kashani and C. Shahabi. SWAM: A family of access methods for similarity-search in peer-to-peer data networks. In *Proceedings of the Conference on Information and Knowledge Management-CIKM*, pages 304–313, Washington, DC, November 2004.

[5] M. Batko, C. Gennaro, P. Savino, and P. Zezula. Scalable similarity search in metric spaces. In *Proceedings of the DELOS Workshop on Digital Library Architectures: Peer-to-Peer, Grid, and Service-Orientation*, pages 213–224, Padova, Italy, June 2004.

[6] M. Batko, C. Gennaro, and P. Zezula. A scalable nearest neighbor search in P2P systems. In *Proceedings of the International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (held in conjunction with VLDB)*, pages 64–77, Toronto, Canada, August 2004.

[7] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[8] S. Bhattacharjee, P. Keleher, and B. Silaghi. The design of TerraDir. Technical Report CS-TR-4299, Department of Computer Science, University of Maryland at College Park, October 2001.

[9] M. Demirbas and H. Ferhatosmanoglu. Peer-to-peer spatial queries in sensor networks. In *Proceedings of the IEEE International Conference on Peer-to-Peer Computing*, pages 32–39, Linkoping, Sweden, September 2003.

[10] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the IEEE HotOS VIII Workshop*, pages 65–70, Schloss Elmau, Germany, May 2001.

[11] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *Proceedings of the International Conference on Very Large Databases-VLDB*, pages 444–455, Toronto, Canada, August 2004.

[12] P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multidimensional queries in P2P systems. In *Proceedings of the ACM SIGMOD'04, WebDB Workshop*, pages 19–24, Paris, France, June 2004.

[13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD'84*, pages 47–57, Boston, MA, June 1984.

[14] A. Harwood, S. Karunasekera, S. Nutanong, E. Tanin, and M. Truong. Complex applications over peer-to-peer networks. In *Poster Proceedings of the ACM Middleware'04*, page 327, Toronto, Canada, October 2004.

[15] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, December 2003.

[16] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigraphy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 654–663, El Paso, TX, May 1997.

[17] G. Kedem. The quad-cif tree: a data structure for hierarchical on-line algorithms. Technical Report TR-91, Department of Computer Science, University of Rochester, September 1981.

[18] W. Litwin and T. Risch. LH*g: A high-availability scalable distributed data structure by record grouping. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):923–927, July 2002.

[19] A. Mondal, Yilifu, and M. Kitsuregawa. P2PR-tree: An R-tree-based spatial index for peer-to-peer environments. In *Proceedings of the International Workshop on Peer-to-Peer Computing and Databases (held in conjunction with EDBT)*, pages 516–525, Heraklion-Crete, Greece, March 2004.

[20] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32(3):241–280, May 1999.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM'01*, pages 161–172, San Diego, CA, August 2001.

[22] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD'95*, pages 71–79, San Jose, CA, May 1995.

[23] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the ACM Middleware'01*, pages 329–350, Heidelberg, Germany, November 2001.

[24] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, 1990.

[25] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[26] H. Samet. Depth-first k-nearest neighbor finding using the maxnearestdist estimator. In *Proceedings of the International Conference on Image Analysis and Processing*, pages 486–491, Mantova, Italy, September 2003.

[27] K. Sevcik and N. Koudas. Filter trees for managing spatial data over a range of size granularities. In *Proceedings of the International Conference on Very Large Databases-VLDB*, pages 16–27, Mumbai, India, September 1996.

[28] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM'01*, pages 149–160, San Diego, CA, August 2001.

[29] E. Tanin, A. Harwood, and H. Samet. A distributed quadtree index for peer-to-peer settings. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 254–255, Tokyo, Japan, April 2005.

[30] E. Tanin, A. Harwood, H. Samet, S. Nutanong, and M. Truong. A serverless 3D world. In *Proceedings of the Symposium on Advances in Geographic Information Systems*, pages 157–165, Arlington, VA, November 2004.

[31] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB-CSD-01-1141, Department of Computer Science, University of California, Berkeley, April 2001.