DataSpace - querying and monitoring deeply networked collections in physical space
Part II - Protocol Details

Samir Goel, Tomasz Imieliński,
Department of Computer Science,
Rutgers University
{gsamir@paul, imielins@cs}.rutgers.edu

October 14, 1999

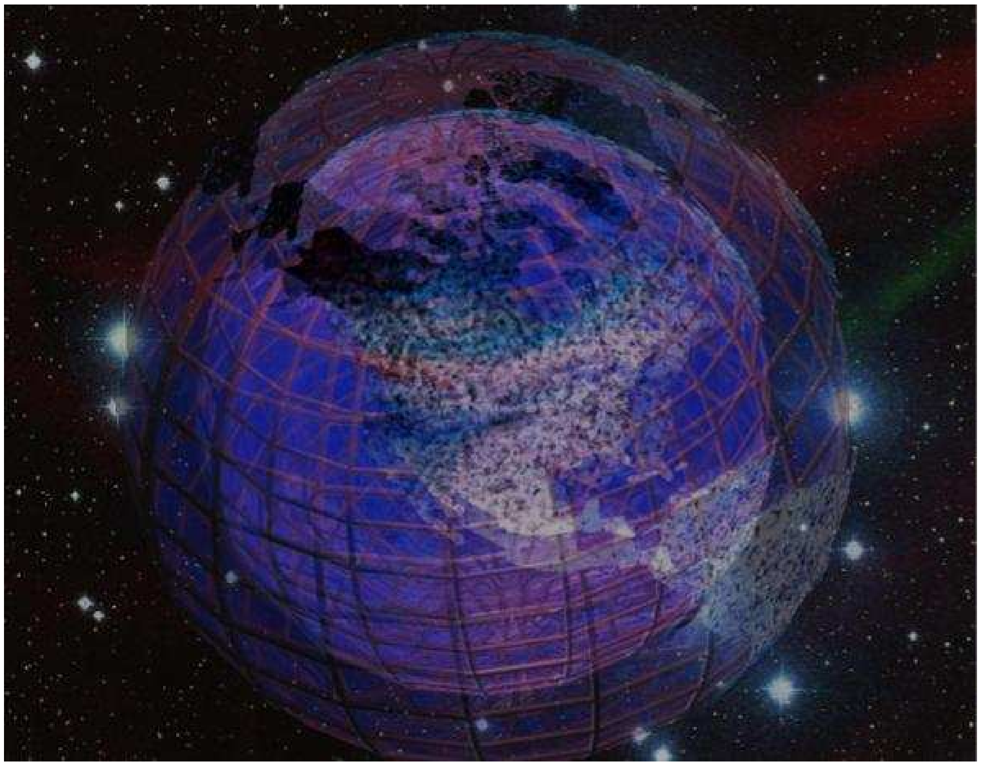

Figure 1: DataSpace is three dimensional physical space stretching from 10 kilometers below the surface of earth to 100 kilometers above the surface of earth, that is accessible to the network. DataSpace enables one to issue queries within every cubic millimeter block of physical space

Abstract

The DataSpace is a three dimensional physical space 100 kilometers above and 10 kilometers below the surface of earth that is accessible to the network. It is addressed geographically as opposed to the current "logical" addressing scheme of the Internet. In [3], we have described the concept of DataSpace and proposed an architecture for it. In this report, we delve into the details of the DataSpace protocols.

1 Introduction

DataSpace [3] is three-dimensional[1] physical space connected to the network (we have described the concept of DataSpace in detail in [3]). In contrast to the logical structure of the Internet, DataSpace is embedded in physical reality. It is built from three-dimensional *administrative* or *geometric* datacubes. Administrative datacubes come in many sizes and encapsulate an administrative "domain" for e.g., a city, a street, a building, a shelf, a drawer, or even left hemisphere of someone's brain. Geometric datacubes can range from a cubic mile around World Trade Center to a cubic millimeter in the retina of someone's eye.

DataSpace is populated by massive number of objects[2], each one either storing or seeking data. In order to be able to participate in DataSpace, an object only needs to be able to communicate with other objects using DataSpace protocols.

We support two operations in DataSpace: *querying* and *monitoring*. Querying operation allows an object to query other objects in DataSpace, while monitoring allows it to monitor the answer to a query. We are interested in querying and monitoring operations that are spatially constrained (for e.g., location of all taxi-cabs *within 2 blocks from me*). In order to organize objects in DataSpace in a way that they can be usefully accessed and manipulated, we group them into classes called *dataflocks*.

In [3], we have proposed an architecture of the DataSpace. At the heart of our architecture is the idea that network can serve as an effective DataSpace engine. We propose to support the querying and monitoring operations at the network level using multicast mechanism. With this approach, the data resides at its source and the network takes care of "routing" queries appropriately such that it reaches the objects that satisfy it. Likewise, it takes care of routing the updates to the objects monitoring the answer to a query. Collectively the data stored with the objects in DataSpace forms a huge data store (albeit massively distributed), and we make use of multicast as a mechanism for indexing it. Figure 2 compares the DataSpace model with the traditional database model. In database model, a database collects data about real physical objects. Here the physical objects become merely the artifacts of their corresponding entry in the database. Users of this database issue operations and receive the results. In DataSpace model, data and the physical object is a single entity — data resides with its source. All entities (data source as well as clients) are connected to the network. Here the network serves as a database engine and processes the querying and monitoring operation issued by the users.

In this report, we describe in detail how querying and monitoring operations are supported in DataSpace. We also describe the "fat" shared tree scheme — our proposed solution to deal with two of the problems that arise because of the presence of huge number of multicast groups

[1]In principle, DataSpace can even be 4-dimensional with time as the fourth dimension, to support querying the object histories as well

[2]In DataSpace, objects may be either stationary or they may move through the physical world while still maintaining (some level of) connectivity to the network

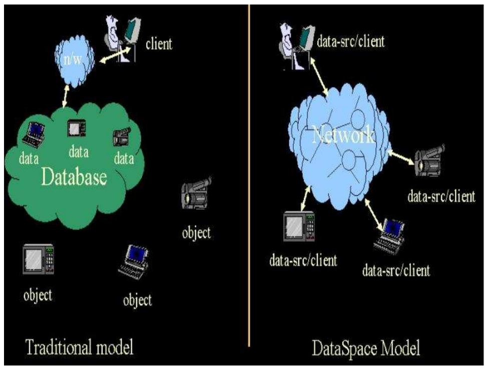

Figure 2: Comparison of DataSpace model with database model

in DataSpace: (1) possible overflow of the multicast forwarding table state, and (2) overhead of maintaining huge number of multicast trees.

This report is organized as follows. In the next section, we briefly recapitulate the architecture of DataSpace. In section 3 we give a high-level overview of how querying and monitoring operations are performed in DataSpace, and in sections 5, 6, 7, we describe them in detail. In section 8 we describe the fat shared tree [4, 5, 6] protocol. In section 9, we enumerate the unresolved issues, and finally, we conclude in section 10 with some concluding remarks.

2 Architecture

DataSpace may be viewed as a collection of datacubes which are either geometric or administrative. *Geometric cubes* are defined as three dimensional cubes ranging from tens of cubic kilometers to the smallest ones possibly even of the size of cubic centimeter (although the smallest datacubes which we propose will be of the order of cubic meters). *Administrative datacubes* include countries, states, towns, streets, buildings, rooms, regions of a body or machine internals. Each datacube has a corresponding *space handle* that encodes its location in 3-dimensional physical space and and its size.

Datacubes are populated by *dataflocks* — which are static or mobile classes of objects. Each object (sensor, machine, piece of software, physical object) is characterized by a set of methods. Dataflocks will be subject of querying or monitoring. Some of the dataflock methods will be supported by *network indexes*. By the network index, we mean an attribute/method for which every pair (method, value) has a corresponding multicast address and all objects which satisfy the predicate (method = value) are members of the corresponding multicast group. Pairs q = (method, value) such that the method is indexed are called *subject handles*. Network indexes are used in processing of the DataSpace queries in a similar way as database indexes are used in processing database queries.

Each datacube has its own local DS which contains entries for the dataflocks that are registered in that datacube. Each dataflock can be registered in many datacubes. The DS will contain the network indexes for such dataflock. Mobile dataflocks will have to re-register in the new datacubes that they enter.

Each subject handle q, in addition to defining the group of objects which satisfy it, may also have another group of objects associated with it. That group, denoted by Interested(q), monitors all changes to the membership of q. In other words every time a new object joins q or an existing object drops out of q it sends out a message to the Interested(q) group. Members of the group Interested(q) can themselves be viewed as a dataflock registered either locally, in the same datacube as the members of q, or in another, possibly remote, datacube. Thus, there are in principle two multicast addresses associated with the subject handle, q: the *passive* address which allows reaching objects satisfying q and the *active address* which allows objects satisfying q to actively update their status for the group Interested(q).

The members of Interested(q) monitor updates to the membership of query q, and hence know the answer to query q. A few of these members may offer this knowledge base to anybody who is interested in finding the answer to query q. Such members are called *Brokers* of query q. They are similar in concept to a network cache. Thus, an object interested in finding the answer to query q, may request the Broker of q for the answer instead of directly querying the objects. Having brokers tends to be more efficient especially for queries whose answer do not change rapidly. For these queries, it is more resource efficient to cache the answer to the query and use it to service the request of the objects, instead of querying the objects directly.

2.1 Datacube Directory Service (DS)

DS for a given datacube will store four types of entries:

- Mapping between the descriptors q and the corresponding subject codes
- Mappings between the descriptors Interested(q) and the corresponding subject codes: these multicast addresses will be used to disseminate updates to q (additions and deletions) among the interested domestic clients *within* the same datacube.
- address of the Core router for the datacube. This acts as core router for all the multicast groups that are local to this datacube.
- address of the broker for the datacube. This acts as broker for all the queries that are local to this datacube.

Notice that subject code corresponding to Interested(q) will be combined with area codes corresponding to that of q, i.e., of the datacube where the monitored objects are present.

3 DataSpace Protocol: Overview

The DataSpace protocol involves supports two operations: querying and monitoring.

3.1 Querying

A DataSpace query is an expression of the following form:

Q = Select
From <Dataflocks>
Where <Condition>
In <Datacubes>

The DataSpace query returns the network identifiers (can be unicast addresses for example) of the dataflock objects which satisfy the query Q.

Condition in a query is a conjunction of elementary descriptors of the form (method OP value) where OP can be an arithmetic operator and Datacube is either a geometric or administrative datacube.

Two parameters that are critical in processing a DataSpace query are : *space handle* and *subject handle.* The subject handle is selected from the group of subject handles that not only appear in the condition of the query, but are also one of the network indexes for the queried <Dataflock>. Space handle is the area code corresponding to the datacube that appears in the query Q.

A query Q is processed first by sending it as unicast message to the broker of Q in the <Datacube> specified in Q. If the broker is out of date then the query is sent as multicast message within the <Datacube>. This part of the querying protocol involves two stages *illuminating* a datacube, and *selective reflection.* Both these terms usually describe the behavior of light waves and are used here on purpose to further emphasize the analogy of DataSpace to the real space. DataSpace illumination involves multicast message directed at entire datacube. Such message is routed entirely on the basis of the area code ignoring the subject code of the multicast address. Selective reflection is implemented through the network layer filtering based on the subject code of the multicast address of the query with the application level filtering corresponding to the rest of the query.

Within a datacube, a query is processed in DataSpace in a way very similar to the way querying is performed in a database system. As an example, consider the processing of query for finding all the tube-lights in CoRE building that are currently not working:

Q_1 = *Select*
From <Electric-Fixtures>
Where <Type=TUBE-LIGHTS AND status=NOT-WORKING>
In <CoRE-Building>

Assuming that there is a database that maintains information about all the electrical fixtures of CoRE building and their status, and assuming that the database has an index on attribute "Type", the query will be processed by first using the index to select all the records that satisfy the condition: *Type=TUBE-LIGHTS*, and then sequentially checking the selected records for the remaining part of the condition: *status=NOT-WORKING.*

In DataSpace, all tube-light's fixtures within CoRE building will subscribe to the *passive* multicast address given by a combination of space handle corresponding to CoRE building and subject handle corresponding to Type=TUBE-LIGHTS (assuming that "Type" is one of the network indexes of dataflock "Electric-Fixtures"). The above query will be processed by sending a mulitcast message to the datacube enclosing CoRE building at the above multicast address. The remaining part (*status=NOT-WORKING*) of the *Conditions* in the query is placed in the body of the message. This multicast message will reach all the tube-lights in CoRE building. Each of them will respond with their identifier if they also satisfy the other conditions present in the body of the message (i.e., if they are not working). Thus the first level of filtering happens at the network layer while the second level of filtering happens at the application layer. In effect, the multicast mechanism serve as a (network) index for the massively distributed information repository, i.e., the DataSpace.

3.2 Monitoring

In DataSpace an object may monitor a query of the form:

M = Select
From <Dataflocks>
Where <Condition>
In <Datacube>

Monitors send updates to the extension of the query M to the clients who are interested in receiving such reports.

The client that is interested in monitoring the query M will have to join the multicast group corresponding to Interested(q) and space handle, S. Here, q is one of the subject handles mentioned in <Condition>, and the space handle, S, corresponds to that of <Datacube> specified in M.

In the next section we describe the DataSpace protocol in more detail.

4 Terminology

In this report, we frequently use the following terms:

- *Dataflocks*: classes of objects in DataSpace.
- *Space handle*: It is an identifier for a datacube. It is obtained from the GPS co-ordinate of the datacube. These preserve the *prefix property.*
- *Prefix property*: Prefix property ensures that space handle of a datacube is the prefix of the space handles of all the datacubes enclosed by it.
- *Datacube identifier*: It is an identifier for a datacube. In this sense, it is similar to a space handle. The difference is that datacube identifiers do not preserve *prefix property.* This allows them to be shorter than the space handles.
- *Network index*: An attribute/method for which every pair (method, value) has a corresponding multicast address and all objects which satisfy the predicate (method = value) are members of the corresponding multicast group.
- *Subject handle*: Pairs q = (method, value) such that the method is indexed are called *subject handles.*
- *Datacube Directory Service (DS)*: It stores the state information of a datacube.
- *Brokers*: Broker of a query caches the answer to the query. Objects interested in finding answer to a query q can request the answer directly from the broker of q.
- *Shared tree*: We assume a shared tree multicast routing protocol. Shared tree refers to the tree built by the routing protocol for delivering multicast packets for a particular multicast group.
- *Core router*: We assume a shared tree multicast routing protocol. Core router for a multicast group is the router on which its corresponding shared tree is rooted.
- *Multicast addresses in DataSpace*: There are three types of multicast addresses in DataSpace, each with its own unique prefix. They are:
  - *Passive* multicast address: a multicast address used for querying. These multicast addresses have the symbolic prefix PASSIVE-MCAST-ADDR.

  - *Active* multicast address: a multicast address used for monitoring. These multicast addresses have the symbolic prefix ACTIVE-MCAST-ADDR.
  - *DS-discovery* multicast address: a multicast address used in discovering DS. These multicast addresses have the symbolic prefix DS-DISCOVERY-MCAST-ADDR.

- *Representative datacube*: DataSpace is sparsely populated, not every datacube has an associated DS. Datacubes without DS are represented by the smallest datacube that encloses them and has a DS. So, for any datacube, its representative datacube is the smallest datacube that encloses it and has a DS.

- *DS-discovery*: Given any datacube, the mechanism of finding DS of the representative datacube is called DS-discovery mechanism.

- *Schema call*: The message sent by an object to initiate DS discovery is called schema call.

In the next section we define the basic concepts of DataSpaces more formally.

5 DataSpace Protocol: Details

In this section, we describe the DataSpace protocols in detail.

5.1 Querying

As mentioned in section 3.1, a query is processed by first sending it as a unicast message to the appropriate broker. If broker is out of date, the query is sent as a multicast message within the datacube.

5.1.1 Querying operation: details

Since DS of a datacube contains the address of the broker and the core router for the datacube, both the operations: querying the broker as well as directly querying the objects, require locating the DS of the datacube. Since not every datacube has a corresponding DS, given a query of the form:

Q = Select
From <Dataflocks>
Where <Condition>
In <Datacube>

processing this query involves first discovering the smallest datacube that encloses the <Datacube> and has a DS. We call this the *representative datacube* of <Datacube>. The process of discovering the DS corresponding to the representative datacube is called *DS-discovery.* The DS-discovery mechanism uses the space handle of <Datacube>. It is described in more detail in section 6.

The second step in processing the above query is to obtain the address of the broker and core router from the discovered DS. Once this is done, the query is first sent to the broker. If the broker has a valid answer to the query in its cache, it responds back with the answer. If the broker does not have the answer to the query, the query is sent as a multicast message within the datacube. The multicast address for this message is a combination of *datacube identifier* and the *subject handle* of the query. The datacube identifier is obtained from the discovered DS, or by applying a hash function to the space handle of the *representative datacube*. The subject handle is selected from the set of subject handles that appear in the <Condition> part of the query and are also one of the network indexes for the queried <Dataflock>. The body of the multicast message contains the remaining set of subject handles from the <Condition>. Based on the multicast address, the message reaches those objects that reside in the *representative datacube* and also satisfy the condition embedded in subject handle. Thus, the first level of filtering happens at the network level. The second level of filtering is done at the application layer using the set of subject handles contained in the body of the message. If a message also passes the application level filtering at an object, it means that the object satisfies the query. Such an object responds back with its network identifier. If a message fails the application level filtering, it is silently discarded. The appendix B gives the pseudo-code for querying operation.

A datacube identifier is different from space handle of the same datacube in that it does not have prefix property. This allows it to be shorter (60 bits) than the space handle. The advantage of having a shorter identifier is that it leaves more bits for the subject handle.

5.2 Monitoring in DataSpace

The monitoring operation proceeds in a manner very similar to querying. In order to monitor a query M of the form:

M = Select
    From <Dataflocks>
    Where <Condition>
    In <Datacube>

an object first initiates DS-discovery mechanism (section 6). As the second step, it queries the discovered DS for the address of the core router for the <Datacube>. As the last step, it maps the query M into a multicast address, and subscribe to that address. The mapping is done picking a subject handle from <Condition> and mapping it to a corresponding subject code, and by using a combination of datacube identifier, subject handle, and a starting prefix to identify the address to be an "active" multicast address. One can map subject handle to subject code either by querying the discovered DS for the subject code corresponding to subject handle in <Condition> or by applying a hash function to the subject handle. Appendix C gives the pseudo-code for monitoring operation.

5.3 Registering in a datacube

If an object wants to be a part of DataSpace, it needs to register in a datacube. The object will register in one of the datacubes enclosing its current location and having a DS. In order to determine which datacube it should register in, it takes into account its mobility rate and its region of movement, and determines a datacube, D, that is sufficiently large for its purpose. Using the DS-discovery mechanism, it determines the smallest datacube enclosing D and having a DS. It then registers its schema with it and obtains the address of core router and broker for the datacube. Note that an object needs to register its schema with the DS only when no other object belonging to the same dataflock is registered with it. The pseudo-code for registering dataflock is given in appendix E.1.

A registration is based on a lease — it expires after some time interval, unless it is renewed. Having lease based registration is effective in handling failure of objects — their registration will expire automatically after a certain time interval.

Establishing DS (DS_{new}) of a datacube is performed similarly. The only operation that is required is to register DS_{new} with the DS of the smallest enclosing datacube. The DS of the smallest enclosing datacube is found using DS-discovery mechanism. The DS_{new} then registers the space handle of its datacube with the discovered DS. This registration is also based on a lease, and needs to be renewed before the lease expires. The pseudo-code for registering a new DS is given in appendix D.

6 Discovering DS

In order to query, an object relies on DS-discovery mechanism for discovering the address of the DS of the *representative datacube* — smallest datacube enclosing the datacube specified in the query and having a DS. There are various possible ways of providing a DS discovery mechanism. One possibility is to assign a geo-node to each established datacube, and using Geographic routing [7] to reach the DS within the representative datacube. We describe another possible mechanism here.

6.1 DS Tree

The idea is to build a DS tree such that if a datacube D_1 encloses datacube D_2, then DS of D_1 is an ancestor of DS of D_2. As an example consider a DataSpace shown in figure 3. The figure shows only the datacubes that have a corresponding DS. Figure 4 shows the DS tree corresponding to this DataSpace. Note that in a DS tree if DS of datacube b is the parent of DS of datacube i, then it means that DS_b is the smallest datacube that encloses DS_i and has a DS.

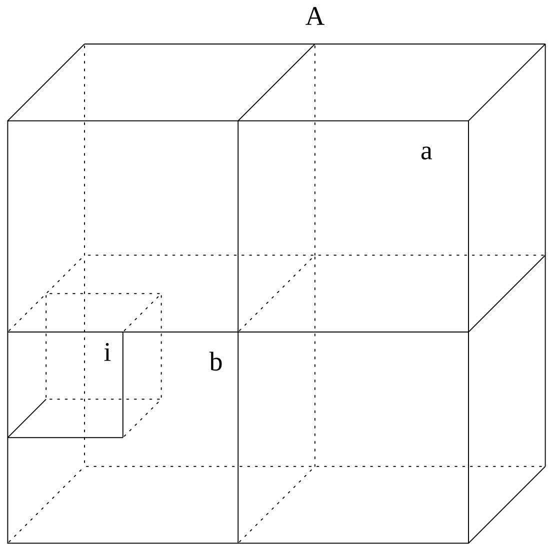

Figure 3: Figure showing a large datacube A enclosing three datacubes (labelled a, b, and i). Only those datacubes that have a DS are shown.

The links in the DS tree are a result of registration operations: A child DS registers itself with the parent DS in a DS tree. As part of the registration operation, a DS registers the *space handle* of its datacube and its (unicast) address. A *space handle* is a unique identifier of a datacube. *Space handle* of a datacube is obtained by encoding the GPS co-ordinate of one corner of the datacube

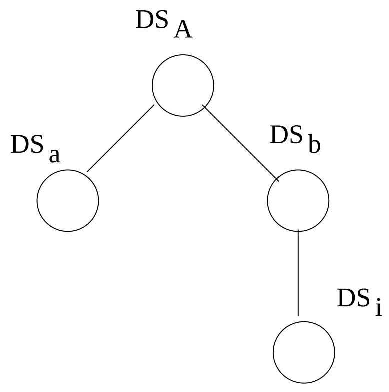

Figure 4: DS tree corresponding to fig. 3

and the length of one of its side, into a bit string in such a way that *prefix property* is preserved. Prefix property ensures that space handle of a datacube is the prefix of the space handles of all the datacubes enclosed by it. The procedure for encoding GPS co-ordinate of a datacube[3] is described in section 7. For now, we will assume that there is a mechanism for encoding GPS co-ordinate of a datacube such that the prefix property is preserved.

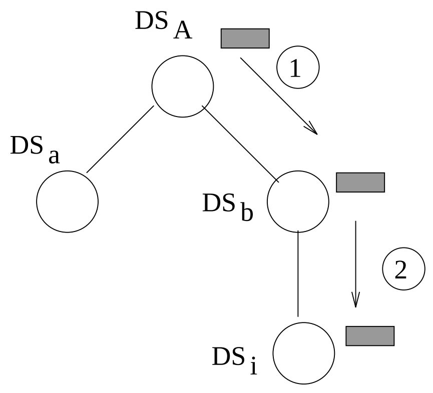

Figure 5: Path traversed by message sent to DS_A when the datacube specified in the query lies within datacube i (figure 3).

Let the datacube specified in a query be D_{query}. The querying object initiates discovery of DS of the representative datacube of D_{query} by making a *schema call*. A *schema call* is a unicast message containing the space handle, S, of D_{query}. It is sent to any one of the DS in the DS tree. As an example, assume that D_{query} is enclosed within datacube i (figure 3). Assume that the object sends the unicast message to DS, $DS_{initial}$. There are two possible cases to consider:

- *Case a*: The space handle corresponding to datacube of $DS_{initial}$ is a prefix of S, the space handle of D_{query}.

[3]By GPS co-ordinate of a datacube, we mean the GPS co-ordinate of an agreed upon corner of a datacube. Note that it is not important which corner of the datacube is chosen as long as the same corner is selected for all the datacubes in DataSpace.

In this case, $DS_{initial}$ will forward the message to one of its children in the DS tree whose corresponding space handle is also a prefix of S. The child DS, in turn, will perform the same operation. This will continue until the message reaches the DS that is either at the leaf of the DS tree, or it does not have any children whose space handle is a prefix of S.

As an example, assume that the object sends the unicast message to DS_A. As shown in figure 5, DS_A will forward the message to DS_b, which, in turn, will forward the message to DS_i. Since DS_i is the leaf of the DS tree, the message cannot be forwarded any further. Hence, DS_i is the DS of the representative datacube of D_{query}.

- *Case b*: The space handle corresponding to datacube of $DS_{initial}$ is *not* a prefix of S, the space handle of D_{query}.

  In this case, $DS_{initial}$ will forward the message to it parent in the DS tree. If the space handle corresponding to datacube of parent DS is also not a prefix of S, it will forward the message to its parent. This will continue until the message reaches the DS whose corresponding space handle is a prefix of S. Then, following the procedure in *Case a*, the message will then get forwarded down the DS tree until it reaches the DS of the *representative datacube.*

  As an example, assume that the querying object sends the unicast message to DS_a. Since space handle of datacube a is not a prefix of S, as shown in figure 6, DS_a will forward the message to DS_A. Since datacube A encloses datacube D_{query}, because of *prefix property* its corresponding space handle will be a prefix of S. The DS_A will then forward the message to DS_b, which, in turn, will forward it to DS_i. Since DS_i is the leaf of the DS tree, the message cannot be forwarded any further. Hence, DS_i is the DS of the representative datacube of D_{query}.

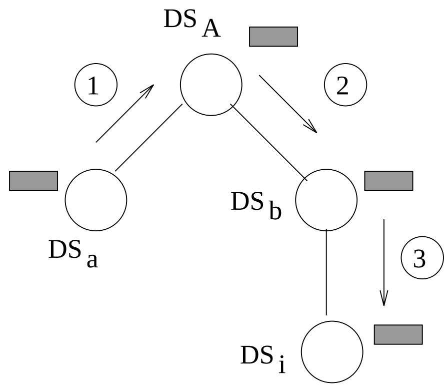

Figure 6: Path traversed by message sent to DS_i when the datacube specified in the query lies within datacube a (figure 3).

Thus, in order to perform DS-discovery, an object needs to know the address of any one of the DS in the DS tree. Since most of the queries in DataSpace would be local, an object will typically cache the address of DSs corresponding to a few small datacubes around it.

When a new DS is introduced for a datacube, x, it attempts to locate the DS in the DS tree that corresponds to the smallest datacube enclosing x. It does this by performing DS-discovery using the space handle corresponding to x. Once it locates the DS of representative datacube of

x, it registers the space handle of x and its own (unicast) address with it. By registering itself, it becomes a part of the DS tree.

The registration information is maintained as soft-state — it needs to be periodically refreshed otherwise it will expire. This is essential in order to keep the DS tree current in face of changes in the state of DataSpace — new datacubes may be established by introducing a corresponding DS, or existing DS may fail or may be removed. We believe that state of DataSpace will change very slowly.

In order to renew its registration, a DS performs the same procedure as when registering itself for the first time. If the DS tree has not changed, it will receive a message from its old parent DS in the tree. If the DS tree has changed, it will receive a message from its new parent in the tree. In this case, it will register itself with the new parent.

7 Mapping GPS co-ordinates to space handle

Given the GPS co-ordinate of a point, we convert it into a bit pattern such that the prefix property (section 6) is satisfied. Prefix property ensures that space handle of a datacube is the prefix of the space handles of all the datacubes enclosed by it. We achieve this by first converting GPS co-ordinates (latitude, longitude, and altitude) into rectangular co-ordinates (x,y,z) (appendix A). We next encode each of the x, y, and z co-ordinate using a division tree. Division tree (figure 7) is a complete binary tree with the following property: All left branches have a weight of zero. Assume that root is at level 0. The weight associated with the right branch between nodes at level i and level (i+1) is given by : W_i = w/$2^{(i+1)}$, where w is the maximum possible value that the tree can encode. Thus the right branch between root and its right child will have a weight of w/2, the right branches, one level below, will have a weight of w/2^2, and so on. The branches are also given a label: all left branches have a label of 0 while all right branches have a label of 1.

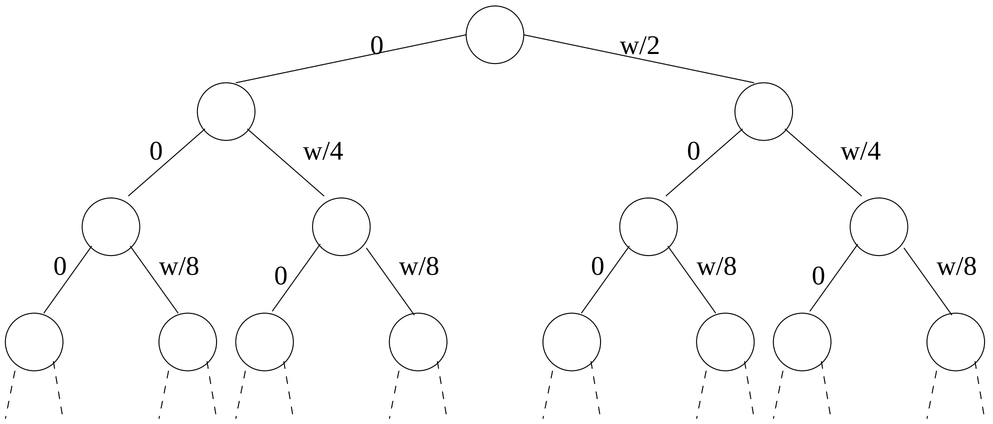

Figure 7: A division tree

In order to encode a co-ordinate, we follow a path in this tree such that the sum of weights of the branches along the path equals the actual value of the co-ordinate to within a desired level of precision. The encoding is obtained by concatenating the labels of branches in this path.

For our purpose, we will choose w to be equal to the diameter of earth. Note that a division tree is a infinite tree. We limit the size of the encoding of each co-ordinate to atmost 30 bits. This is sufficient to get a precision of less than 1.19cm.

When we are encoding GPS co-ordinate of a *point*, we use 30 bits for encoding each co-ordinate. When we are encoding GPS co-ordinate of a *datacube*, we use the size of one side of the datacube to determine the precision; we go down the tree until the weight of the right branches remains greater than the size of one side of the datacube. This also ensures that encoding of all three co-ordinates is equal in length. Note that for larger datacubes, the encoding of their GPS co-ordinate will give a smaller bit-pattern.

As an example, consider the division tree shown in figure 8. The figure illustrates encoding the number 9 by using a division tree (w is chosen to be 8 in the example). The bit-string encoding of 9 is found to be 1001.

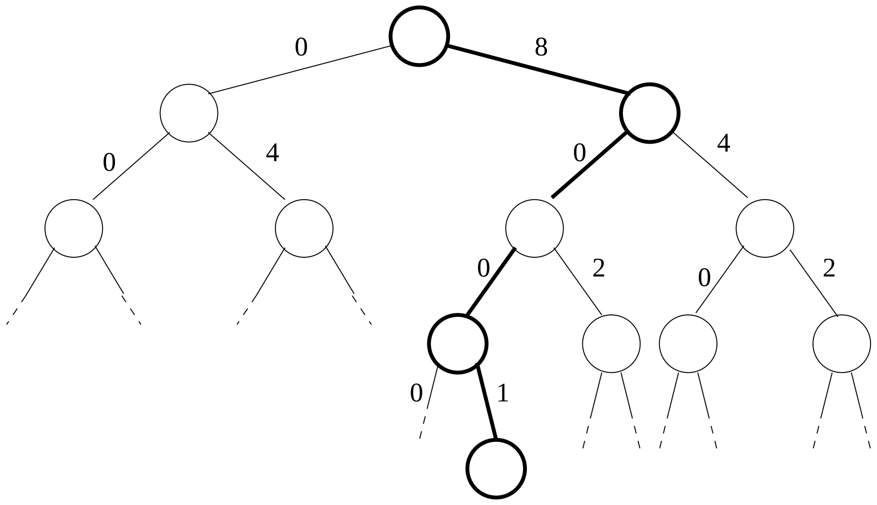

Figure 8: An example division tree. The highlighted path gives the encoding of the number 9

The final bit-string is formed by interspersing the bit-string encoding for x, y, and z such that the following property holds:

- If we start from the leftmost bit, pick every 3rd bit from the final string, we should get encoding for x.
- If we start from the 2nd leftmost bit, pick every 3rd bit from the final string, we should get encoding for y.
- If we start from the third leftmost bit, pick every 3rd bit from the final string, we should get encoding for z.

The maximum size of the space handle is 90 bits. For larger datacubes, it will be smaller than 90 bits.

8 Fat Shared Trees

In DataSpace, the potential number of possible queries runs in billions. Following our approach of network as a DataSpace engine, we map each of these queries to a spacecast (multicast) address.

Thus, we have potentially billions of multicast groups in the system. Presence of such humongous number of multicast groups raises the following serious scaling issues:

- *Multicast forwarding table size*

  The first and foremost issue concerns the size of multicast forwarding table. With so many multicast groups, even with a shared tree scheme wherein a router stores just one entry per multicast group, it is very likely that some routers have unmanageably large multicast forwarding tables.

- *Overhead of maintaing the trees*

  The second issue relates to the cumulative overhead of maitaining shared trees for so many multicast groups. In particular, the cumulative overhead of the *Join* and *Prune* messages may become a significant fraction of the data traffic if it is not handled properly. This is very likely because of the presence of potentially billions of in the DataSpace.

8.1 Overview

We propose the concept of fat shared trees [4, 5, 6] in order to address the two problems mentioned above. A "fat" shared tree represents multiplexing of shared trees with common prefix (area code), just as shared multicast trees represent multiplexing of source-specific multicast trees. They reduce the total number of shared trees and hence, the number of multicast forwarding table entries at the routers. We propose to use "fat" shared trees both in order to reach the datacubes, and within a datacube.

In Fat shared trees, the reduction in forwarding table size comes at a cost. Forcing multiple multicast groups on the same "fat" shared tree results in some unnecessary traffic — a packet for any component multicast group is received by all the hosts connected to the "fat" shared tree. The objective is to strike a balance between unnecessary traffic and forwarding table size. At points where unnecessary traffic exceeds a certain threshold, we need to cut the fat shared tree into multiple thinner fat trees such that unnecessary traffic volume is within acceptable limits. Also, at points where routers are on the verge of experiencing overflow in their forwarding table, they can coalesce a few fat shared trees to get a fatter shared tree.

We intend to achieve this global objective by performing splitting and coalescing operations locally at the routers. In order to perform these operation dynamically at a router, we need a mechanism for detecting when and how to spilt a fat tree, and when and how to coalesce a few fat shared trees. These four components define the fat shared tree protocol.

Note that the characteristic of a shared tree (fat or thin) is determined by the type of its corresponding entry in the forwarding table of a router. A shared tree might be part of a fat shared tree at a router, but might not be multiplexed with any other shared trees at another router. Thus, the characteristic of a shared tree is a view specific to a router. When we talk about coalescing or splitting operations, we are referring to local operations at a router to change its view of the shared tree — it can aggregate a few forwarding table entries into a single entry thereby creating a local view of a fat shared tree, or it can decide to split a shared tree from a fat shared tree creating a local view that the shared tree is not multiplexed with other shared trees.

The proposed protocol may be seen as a modification to CBT [2] (with the difference that core router discovery mechanism is implemeted at application layer [8]).

8.2 Fat Shared tree protocol: Details

The basic idea here is to multiplex packets from more than one multicast group on a single ("fat") shared tree. The multicast groups with identical prefix are multiplexed on the same shared tree. This reduces the total number of shared trees and hence, the number of multicast forwarding table entries at the routers. However, forcing multiple multicast groups on the same shared tree, results in some unnecessary traffic as well — a packet for a multicast group is received by all the hosts connected to the shared tree. The challenge is to strike a balance between the amount of resulting unnecessary traffic and the size of multicast forwarding table at the routers.

The proposed protocol can be seen as a modification to CBT. We describe the modifications below.

8.3 Characteristics specific to DataSpace environment

We design the fat shared tree protocol around the following characteristics which are specific to a DataSpace environment.

- 90-10 rule : we believe that a small fraction of the multicast groups will contribute to a major portion of the overall traffic. In this sense, these multicast groups are analogous to the web sites in WWW — a small fraction of the web sites are "hot", and a major portion of the "hits" are localized to these sites.
- membership to a multicast group is more stable in a DataSpace environment.

8.4 Splitting a fat shared tree

The last hop routers do not coalesce shared trees. Thus, they have complete knowledge of the memberships of the directly connected hosts. Any unnecessary traffic caused by coalescing fat shared trees by routers located higher up in the tree, will be discarded by the last hop routers. This has the advantage of isolating the end hosts from any unnecessary traffic. Also, it makes last hop routers ideal points for generating feedback in order to control unnecessary traffic.

A fat shared tree is analogous to our e-mail account. Daily, we receive e-mails on various topics. Along with the relevant e-mails, we also receive quite a few junk e-mails. While we tolerate some level of junk e-mails, when it gets beyond a particular "threshold", we send a "remove" message to a few originators. Conceptually, this is what the protocol does. This technique is effective because of the 90-10 rule — since a few multicast groups contribute to a major portion of the traffic, a few negative feedbacks can cut down the unnecessary traffic to acceptable levels.

The goal is to identify the top few (a small number) offending groups, and report them to the previous hop router. The previous hop router instantiates a new entry in its forwarding table such that no more traffic belonging to the offending grps is forwarded to this router. For example, assume that a fat shared tree entry at a router is:

Address	Parent i/f	Child i/fs
a*	1	2, 3

if it receives a prune for multicast group abc from child interface 2, it will modify its forwarding table as follows:

Address	Parent i/f	Child i/fs
a*	1	2, 3
abc	1	3

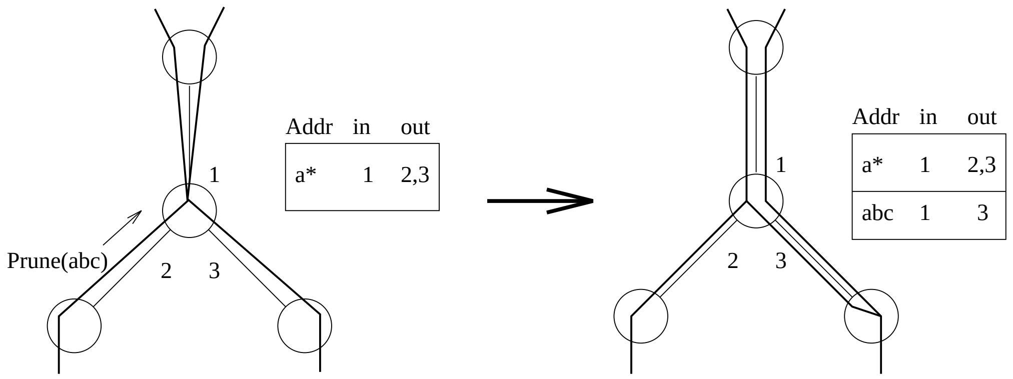

Figure 9: Splitting a fat shared tree

This is illustrated in figure 9. With the above change, packets for multicast group abc will be forwarded only on interface 3 because of the longest prefix match rule.

The above example also illustrates that the fewer the number of prunes, the lesser the number of forwarding table entries created. Thus, it is advantageous to determine the most offending multicast groups and prune them.

8.4.1 Identifying top N offending groups

The last hop routers maintain a cache of size N (as a result of 90-10 rule, we believe that N will be a small number, around 10). Since, the last hop routers do not aggregate their entries, they will be able to differentiate between unncessary traffic and relevant traffic, and drop appropriate packets. When dropping a packet, they cache the multicast address of the packet. If it is already present in the cache, the corresponding counter is incremented. If the cache is full, the entry with the least count is replaced by this group's entry.

The last hop routers also maintain a seperate counter for the total number of packets dropped during an epoch. This is useful to compute the percentage of unnecessary traffic. The psuedo-code for caching algorithm appears in appendix F.1.

8.4.2 When to send prune messages

- At the last hop routers :
  whenever the % of total unnecessary traffic exceeds a certain threshold, the router sends out a prune message containing the top M offending multicast groups (where M may be a function of the amount by which the percentage unnecessary traffic exceeds the threshold).

- At intermediate routers :
  when all the outgoing interfaces corresponding to a forwarding table entry has been pruned, the router sends a prune to its previous hop router.

As a result of relatively static membership in DataSpace environment, we believe that the traffic generated by prune messages will be tolerable.

8.4.3 Reacting to Prunes

As described in section 8.4, in response to a prune message, an intermediate router creates new entries such that the traffic for pruned multicast groups is forwarded only on appropriate interfaces. The psuedo-code describing this operation appears in appendix F.2.

8.4.4 Coalescing fat shared trees

The coalescing action is triggered when the forwarding table size exceeds a certain watermark at an intermediate router. The coalescing operation is illustrated in figure 10.

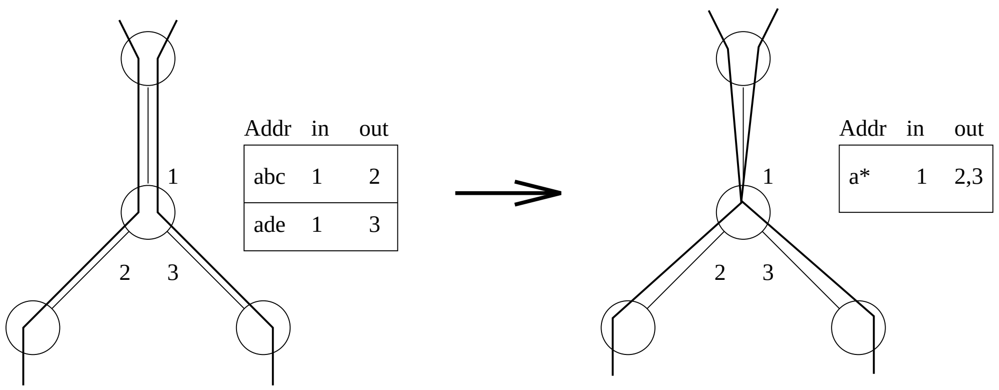

Figure 10: Coalescing shared trees

8.4.5 Coalescing operation

The pseudo-code for coalescing operation at a router appears in appendix F.3. It assumes that the two entries to be coalesced has been selected based on the criterias described next.

8.4.6 Selecting shared trees for coalescing

The idea is to select those shared trees which do not carry much traffic. This will prevent the resulting fat shared tree from generating too much of unnecessary traffic at certain points in the network. In order to achieve this, a router keeps track of the number of packets forwarded by using a particular forwarding table entry, during an epoch. When deciding which two trees to coalesce, it looks for two forwarding table entries with lower percentage traffic, same parent interface, overlapping child interfaces set, and longer common prefix.

Based on the above criterias it selects two shared trees for coalescing. It replaces these two entries by a single entry where the address is the common prefix, and child interfaces is the union of the child interfaces of the individual entries.

8.4.7 Response to a JOIN message

When coalescing two fat shared trees, it is possible that the resulting prefix encompasses more multicast groups than the sum of multicast groups in the two component fat shared trees. This means that in coalescing, the router looses information about the exact shared trees which pass through it. We call this *lossy fusion*. This requires that a JOIN message be handled in a slightly different way.

Traditionally, a JOIN message for a multicast group M, propagates hop-by-hop towards its corresponding Core router, C, until it hits an on-tree router, R. In CBT, R generates an ACK which retraces its path back to the joining host. With lossy fusion, it is possible that R has an entry indicating that it is a part of shared tree for multicast group M, but it may not actually be so — this entry might be an artifact of a prior lossy fusion. In order to resolve the ambiguity, the router allows the JOIN message to propage up towards C (or some router which can unambiguously determine that it is part of shared tree for M).

When the JOIN message finally reaches C (or some router which can unambiguously determine that it is part of shared tree for M), it sends back an ACK. If it happened to receive the JOIN from an interface which is already a part of the shared tree for M, it also sets a bit in the ACK. As this ACK retraces the path back to the joining host, if the bit is set, special processing is required only at those routers where there was an ambguity. At these routers, if the bit in ACK is set, it indicates that the matching forwarding table entry is in fact valid for M. No new forwarding table entry is created, and an ACK is re-generated. If the interface from this router received JOIN is already a part of the shared tree for M, it also sets a bit in the ACK. Every time an ACK is generated, it contains the multicast group prefix present in the matching forwarding table entry. Thus, it is possible that branches of a *fat* shared tree are extended to the joining host.

An implication of the above procedure for handling JOIN is that if core router C functions as Core for a set of multicast groups, then assuming a cold start and that every host is joining multicast group M, instead of a shared tree, a fat shared tree will be created, rooted at core router, C. The prefix of the fat shared tree will be the same as multicast group prefix (i.e., set of multicast addresses) for which C is a core router.

8.5 Related work

Recently, an idea similar to "fat" shared tree [4, 5, 6] has appeared in [9]. These two ideas differ in the way they control the amount of unnecessary traffic. In [9], it is proposed that every router keep an estimate of the unnecessary traffic that it is causing by performing lossy aggregation. A router tries to keep its estimate of unnecessary traffic below a certain “leak budget”. In our proposal, while a router tries to minimize the unnecessary traffic due to lossy aggregation, it leaves it to the last-hop routers to correct its decision, in case they start receiving unnecessary traffic above their tolerance-level. Thus, in our proposal, maintaining the amount of unnecessary traffic in check is a feedback driven process. We intend to compare the relative merits of the two schemes by through simulations.

Another approach to aggregating multicast forwarding table has been suggested in [10]. They approach the problem using non-lossy aggregation. We believe that a lossy aggregation scheme would be more effective than non-lossy one, given the scale of DataSpace system.

9 Unresolved issues: Future Work

The following issues are not yet addressed in our design, and will be a part of our future work.

9.1 Monitoring operation: Issues

Supporting monitoring operation also requires addressing following issues:

- The primary concern is how to keep the overhead associated with sending updates to a minimum. This goal may be broken into two sub-goals:
  - How to make sure that objects satisfying a query q send updates only when the Interested(q) set is non-empty.
  - When should an update be generated — should it be triggered by a change in state of an object, or should it be periodic. An update may be triggered when an object becomes a member so that it can be added in the answer set of the query, and it can be triggered when it is no longer a member so that it can be deleted from the answer set of the query.
- How to make sure that the objects monitoring a query have most up-to-date answer to the query in face of failures of objects and loss of updates in the network. A related issue is, should the entries in the answer be aged out if they are not refreshed.

It is useful to have an object generate an update periodically for three reasons: firstly, it takes care of loss of updates in the network, and secondly, it help members of Interested(q) to catch up with missed updates. Lastly, it allows the entries of the answer at the monitors to be aged out if they are not refreshed. This is an effective way for dealing with cases when objects fail and go out of service without generating an update indicating that they are no longer a member of the answer set of a query. Against these advantages the tradeoff is more number of updates and hence more associated overhead. A related issue that needs to be sorted out is how frequent should the updates be sent.

9.2 Query Zooming: Issues

Queries over a larger datacube should involve aggregation. For e.g., a query about locations of all taxi cabs in Manhattan may be answered by first providing distribution of their total count over two dimensional spatial grid (say, blocks). The user may subsequently issue the same query in a smaller and smaller datacube to get the more and more exact answer. This process is called query zooming.

Intuitively, query zooming mechanism may be implemented by aggregating the answer to the query over smaller sub-datacubes and storing them with their brokers. These aggregates will be returned when an object issues a query over a large datacube. Following issues need to be addressed in order to realize query zooming:

- Different queries require different aggregation function. How does a broker know which aggregation function to use for a particular query.
- In a smaller datacube, how can a broker determine which queries should be aggregated and stored.

9.3 Fat Shared Trees: Issues

It is not clear how a router should select which forwarding table entries to coalesce so that the amount of resulting unnecessary traffic is below the tolerance level of last-hop routers. A bad decision on the part of the router may prompt the last-hop routers to generate feedback, resulting in increase in overhead traffic and further wastage of bandwidth. We intend to perform simulations to test out possible heuristics. A good heuristic will be one that not only is effective in keeping the unnecessary traffic below the tolerance-level of last-hop routers, but is also simple to implement and is scalable to the size of DataSpace system.

9.4 Other Issues

- *Response implosion*

  A DataSpace comprises of huge number of objects. It is very likely that a querying object is flooded by responses, as it will not be uncommon for some queries to be satisfied by a large number of objects. We call this the response implosion problem. We have already described two mechanisms for dealing with this: brokers and *query zooming.* Other possible solutions are: *samplecast* and *gathercast.*

  In samplecast, an object that satisfies query q responds with a probability p. The querying client looks at the number of responses received and can estimate the total number of responses, thus avoiding the response implosion problem. It may then proceed by increasing p to some higher value, $p\prime$ and obtaining a bigger sample of the answer.

  Gathercast [1] is an efficient programmable solution to the problem of applying network wide aggregation and filtering. The aggregation of packets in the network can be carried out by establishing transformation functions at various points in the network. Packets routed through these points can be subjected to transformations that have been programmed previously. Examples of transformations include combining small packets, replacing a set of packets with an aggregate packets, removing redundancy, and replacing a series of packets with a summary packet etc.

10 Conclusions

In section [3], we have defined a new concept of the three dimensional DataSpace which is the physical space enhanced with the connection to the network. The DataSpace is a collection of datacubes which are populated by often mobile classes of objects called dataflocks. Object in dataflocks produce and store their own data, such objects can be queried and monitored on the basis of their properties. In this report, we described in detail how querying and monitoring operations are supported in DataSpace. We also described the “fat” shared tree scheme — our proposed solution to deal with two of the problems that arise because of the presence of huge number of multicast groups in DataSpace: (1) possible overflow of the multicast forwarding table state, and (2) overhead of maintaining huge number of multicast trees. We intend to address the unresolved issues by simulating possible solutions to them.

11 Acknowledgment

We are grateful for discussions and constant encouragement from B.R Badrinath. Earlier discussions with Arup Acharya are gratefully acknowledged although the full responsibility for any of the report's shortcomings is solely with the authors.

References

[1] B. R. Badrinath and Pradeep Sudame. Gathercast: An efficient multi-point to point aggregation mechanism in IP networks. Technical Report DCS-TR-362, Department of Computer Science, Rutgers University, July 1998.

[2] T. Ballardie, P. Francis, and J. Crowcroft. Core based trees (CBT): An architecture for scalable inter-domain multicast routing. In *Proceedings of the SIGCOMM*, 1993.

[3] Tomasz Imieliński and Samir Goel. Dataspace - querying and monitoring deeply networked collections in physical space, Part I - concepts and architecture. Technical Report DCS-TR-381, Rutgers University, July 1999.

[4] Tomasz Imieliński and Samir Goel. Dataspace - querying and monitoring deeply networked collections of physical objects. To appear in Proceedings of of the DIMACS Workshop on Mobile Networks and Computing, March 1999.

[5] Tomasz Imieliński and Samir Goel. Dataspace - querying and monitoring deeply networked collections of physical objects. In *Proceedings of the International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE'99)*, Seattle, WA, August 1999.

[6] Tomasz Imieliński and Samir Goel. Dataspace - querying and monitoring deeply networked collections of physical objects. Position paper for the NSF/DARPA/NIST Workshop on Smart Environments, July 1999.

[7] Julio C. Navas and Tomasz Imielinski. Geographic addressing and routing. In *Proc. of the Third ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'97)*, Budapest, Hungary, September 1997.

[8] R. Perlman et al. Simple Multicast: A design for simple, low-overhead multicast, Internet-Draft, February 1999. http://search.ietf.org/internet-drafts/draft-perlman-simple-multicast-02.txt. Work in Progress.

[9] Pavlin Radoslavov, Deborah Estrin, and Ramesh Govindan. Exploiting the bandwidth-memory tradeoff in multicast state aggregation. Technical Report 99-697, Computer Science Department, USC, July 1999.

[10] Dave Thaler and Mark Handley. On the aggregatability of multicast forwarding state. Technical Report MSR-TR-99-34, Microsoft Inc., 1999.

A Converting GPS into rectangular co-ordinates

The following formulae can be used to convert <latitude, longitude, Altitude> into normal rectangular coordinates :

$R = r + Al$
$x = R\ Cos(La)\ Cos(Lo)$
$y = R\ Cos(La)\ Sin(Lo)$
$z = R\ Sin(La)$

where *La* represents latitude, *Lo* represents longitude, *Al* represents altiude, *r* represents the radius of the earth. Pluggin in *r* as 6377.8Kms and *Al* in kilometers, we get the values of each of the co-ordinates in Kms.

In order to keep the values of x, y, and z positive, we add a constant C = (r + *100*) to each of the co-ordinate's value after the above calculation.

B Querying Operation

```
Query_result query(Dataflock, Conditions, Datacube)
{
   /*
    * Step#A: discover DS of the representative datacube
    */
   ds_addr = discover_ds(Datacube, DS_initial);

   /*
    * Step#B: get the broker and core router address from the
    *         discovered DS.
    */
   <broker, core_router> = query_ds(ds_addr);

   /*
    * Step#C : query broker for answer to the query
    */
   <query_answer, status> = query_broker(broker, Dataflock,
                                         Conditions, Datacube);

   if (status == QUERY_CACHED)
     return query_answer;

   /*
    * Step#D: Broker does not have the answer to this query.
    *         The Query needs to be issued in the representative datacube
    *
    * Step#D.1: get the identifier of the representative datacube
    */
   <space_handle, handle_length> = gps2space_handle(GPS,
                                                    datacube_size);
   datacube_identifier = hash(space_handle, handle_length);

   /*
    * Step#D.2: get the subject handle
    */
   network_indexes = query_ds(ds_addr, Dataflock);
   subject_handle = intersect_and_select_one(network_indexes,
                                              Conditions);
   /*
    * Step#D.3: issue the query
    */
   mcast_addr = concatenate(mcast_addr_prefix,
                            QUERY_ADDR_PREFIX,
                            datacube_identifier,
                            subject_handle);

   data = <Dataflock, Conditions, Datacube>;
   ret_code = send(mcast_addr, data, core_router);

   /*
    * Step#D.4: collect the answer to the query
    */
   query_answer = collect_responses();

   return query_answer;
}
```

C Monitoring Operation

```
void monitor(Dataflock, Conditions, Datacube)
{
   /*
    * Step#A: discover DS of the representative datacube
    */
   ds_addr = discover_ds(Datacube, DS_initial);

   /*
    * Step#B: get the broker and core router address from the
    *         discovered DS.
    */
   <broker, core_router> = query_ds(ds_addr);

   /*
    * Step#C: join the ``active'' multicast address
    *
    * Step#C.1: get the identifier of the representative datacube
    */
   <space_handle, handle_length> = gps2space_handle(GPS,
                                                    datacube_size);
   datacube_identifier = hash(space_handle, handle_length);

   /*
    * Step#C.2: get the subject handle
    */
   network_indexes = query_ds(ds_addr, Dataflock);
   subject_handle = intersect_and_select_one(network_indexes,
                                             Conditions);

   /*
    * Step#C.3: get the ``active'' multicast address
    */
   mcast_addr = concatenate(mcast_addr_prefix,
                            MONITOR_ADDR_PREFIX,
                            datacube_identifier,
                            subject_handle);

   join(mcast_addr, data, core_router);
   return;
}
```

D DS-discovery mechanism

```
/*
 * Given a Datacube D and the address of any DS that is already a
 * part of the DS tree, this procedure discovers the address of DS
 * of the representative datacube of D
 */
unicast_addr discover_ds(Datacube, DS_initial)
{
   <space_handle, handle_length> = gps2space_handle(Datacube.GPS,
                                                    Datacube.size);

   /*
    * set the destination multicast address
    */
   mcast_addr = form_mcast_addr(mcast_addr_prefix,
                                AREA_CODE_ADDR_PREFIX,
                                space_handle, handle_length);

   data = DISCOVER_DS;

   /*
```

```
     * make schema call
     */
    ret_code = send(mcast_addr, data, DS_initial);

    /*
     * listen to the response
     */
    packet = recv_response();

    return packet.src_address;
}

/*
 * Given a Datacube D and the address of any DS that is already a
 * part of the DS tree, this procedure registers DS_new, the new
 * DS of D, with the DS of the representative datacube of D
 */
void register_ds(Datacube, DS_new, DS_initial)
{
  /*
    * Step#A: Discover the address of DS of the representative
    *         Datacube of D
    */
    unicast_addr DS_rep = discover_ds(Datacube, DS_initial);

    /*
     * Step#B: Register the new DS with the discovered DS
     */
    <space_handle, handle_length> = gps2space_handle(Datacube.GPS,
                                                     Datacube.size);

    register(DS_rep, space_handle, handle_length, DS_new);
    return;
}

/*
 * Given a Datacube D, the address of its DS, DS_addr, and the
 * address of previous parent of DS_addr in the DS tree, this
 * procedure refreshes the registration of DS_addr with the DS
 * of the representative datacube of D
 */
void refresh_ds_registration(Datacube, DS_addr, DS_old_parent)
{
  /*
    * Step#A: Discover the address of DS of the representative
    *         Datacube of D
```

```
   */
   unicast_addr DS_rep = discover_ds(Datacube, DS_initial);

   /*
    * Step#B: Register the new DS with the discovered DS
    */
   <space_handle, handle_length> = gps2space_handle(Datacube.GPS,
                                                     Datacube.size);

   if (DS_rep == DS_old_parent)
     refresh_registration(DS_rep, space_handle, handle_length);
   else
     register(DS_rep, space_handle, handle_length);

   return;
}
```

E Registering in a Datacube

E.1 Registering Dataflock

```
void register_dataflock(Datacube, Dataflock)
{
   /*
    * Step#A: Perform DS-discovery
    */
   ds_addr = discover_ds(Datacube, DS_initial);

   /*
    * Step#B: register dataflock
    */
    register(ds_addr, Dataflock.schema);
}
```

F Fat Shared Tree Protocol

F.1 Caching algorithm

```
void cache(Mcast_addr maddr)
{
   Cache_entry *entry = lookup_in_cache(maddr);
   if (entry != NULL)
      entry.count++;
   else
   {
     if (cache_full())
     {
        /*
         * get a pointer to the entry with lowest count
         */
        entry = lowest_count_cache_entry();

        /*
         * overwrite the entry with new address and reset the count
         */
        entry->address = maddr;
        entry->count = 1;
      }
   }
}
```

F.2 Handling Prune messages

```
void Prune(prune_msg p, interface recv_if)
{
   Mcast_addr maddr;
   entry matching_entry, new_entry;

   maddr = p.next_addr();
   while (maddr != NULL)
   {
      matching_entry = lookup_fwding_table(maddr);
      if (generic(matching_entry, maddr))
      {
         new_entry.addr = maddr;
         new_entry.parent = matching_entry.parent;
         new_entry.children =
                  difference(matching_entry.children, recv_if);
         insert_in_fwding_table(new_entry);
      }
      else
         matching_entry.children =
                  difference(matching_entry.children, recv_if);

      maddr = p.next_addr();
   }
}
```

F.3 Coalescing two forwarding table entries

```
entry coalesce(entry a, entry b)
{
   entry new_entry;
   new_entry.addr = longest_common_prefix(a, b);
   new_entry.parent = a.parent; // entries a and b have same parent
   new_entry.children = union(a.children, b.children);
   insert_in_fwding_table(new_entry);
}
```