Abstract

In this article we introduce a new conception of three-dimensional DataSpace, which is physical space enhanced by connectivity to the network. DataSpace is addressed geographically as opposed to the current logical addressing scheme of the Internet. Here, a local area network is replaced by a room, a street, a mountaintop, and so on. Billions of objects populate DataSpace, each aware of its own geographic location. These objects move through DataSpace, and produce and locally store their own data. They can be selectively queried, monitored, and controlled based on their properties. We propose two architectures for DataSpace. We describe mechanisms to use the network as a DataSpace engine in order to perform querying and monitoring operation in a highly scalable way.

DataSpace: Querying and Monitoring Deeply Networked Collections in Physical Space

TOMASZ IMIELINSKI AND SAMIR GOEL, RUTGERS UNIVERSITY

When the physical space around us is illuminated with different types of radiation, specific sets of objects respond to specific types of radiation, even though the radiation may fall on all objects. For example, most of the objects in our environment reflect visible light; bones reflect X-ray radiation, tissues resonate in magnetic fields. On similar lines, we can imagine "illuminating" the physical space around us by "beaming" various types of requests on deeply networked collections of objects contained in space. A request "beamed" on the network may ask for retrieval of data or completion of a task. Furthermore, we can imagine specific sets of objects responding to the request placed by a user, depending on the request type and the properties of the object. Through such a network-mediated illumination of the physical world, one can experience and control billions of objects in one's immediate or remote environment. We call such a world *DataSpace* and define it simply as physical space containing data. It can be the 3-D physical space encompassing Earth (Fig. 1) or that encompassing our galaxy (Fig. 2). Of more immediate relevance is the 3-D real physical space shell starting from 10 km below the surface of Earth and extending up to 100 km above the surface of Earth (Fig. 3). In this article, we use the term DataSpace mostly to refer to this space shell.

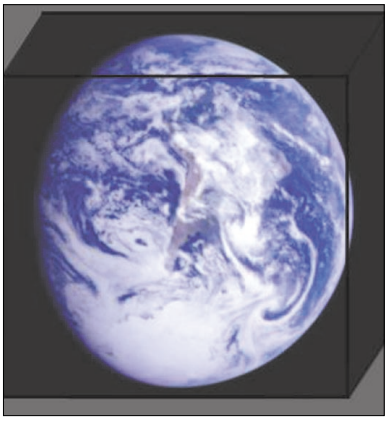

Figure 1. *A 3D physical space cube encompassing planet Earth.*

DataSpace is populated by a very large number of objects that produce and locally store data pertaining to themselves. In DataSpace, physical objects are not just characterized by shape, size, and color, but also by processor type, amount of memory, and network connectivity.

Just as in *real space* through which we move, spatial coordinates are the basic points of reference to navigate and query DataSpace. However, unlike in real space, the user can navigate through and query objects far beyond the range of his senses. Thus, by "moving" through DataSpace, a user, connected to the network, acquires enhanced awareness of his/her surroundings as well as even very remote areas, including information that was previously invisible or beyond his/her access. DataSpace in effect gives the user a "sixth sense" through which to perceive the world.

DataSpace is spatial by definition, built from three-dimensional *administrative* and *geometric* cubes of data. An administrative datacube can encapsulate a city, a street, a building, a basement, a room, or even a shelf or drawer; it can also include the interior of an engine (e.g., one of the cylinders) or the left hemisphere of someone's brain. A geometric datacube can range from a cubic mile around the World Trade Center to a cubic millimeter in the retina of an eye.

In DataSpace, collections of objects inhabit various administrative and geometric datacubes. So for these objects to be usefully accessed and manipulated, they need to be organized into logical clusters. This is achieved by grouping objects into classes called *dataflocks*. Dataflocks are classes of often mobile objects that move through the physical world while still maintaining (some level of) connectivity to the network. Dataflocks are accessed through the datacubes in which they are located and their own class properties.

New objects become part of a datacube by a simple plug-and-play mechanism. Rather than requiring a complex prior protocol, this event involves little processing other than registering a physical object in the scope of its "vision." Plug and play is emphasized in JINI [1], the most serious attempt today to build "smart spaces" of devices such as home appliances, printers, and photocopier machines. Using JINI, these devices can be connected to the local area network and electronically controlled. However, JINI is still oriented toward the logical structure of the Internet, where locality and distance are defined not by geo-

1070-9916/00/$10.00 © 2000 IEEE

graphic space but rather by the Internet's logical structure of subnets. In contrast, DataSpace is spatial and embedded in the physical reality that surrounds us. Here, the local area network is replaced by a room, a street, or the top of a mountain, depending on where the user is located. Also, DataSpace is a global concept — apart from accessing this immediately surrounding "local" space, the user is potentially allowed to query, monitor and control objects in remote spaces as well; in fact, in the whole DataSpace.

The concept of a DataSpace is in contrast to the traditional concept of a database. A database stores information locally about remote physical objects. Here physical objects become merely the artifacts of their corresponding entry in a database. In DataSpace, on the other hand, data is inherently dispersed and connected — it "lives" on the physical object; it is an inherent part of that physical object; it is stored *with* the object, and may be queried by reaching that object through the network. Data becomes another natural characteristic of the object similar to its weight, color, and general appearance.

Inherent in this conception of data that is dispersed in and intrinsic to objects in a DataSpace are the processes of querying and monitoring, and the underlying mechanisms of communication and messaging. This work builds on the work by Navas and Imielinski where they have introduced GeoCast [2] for geographic messaging and implemented it within DARPA's GLOMO program.

Querying and Monitoring

In DataSpace, querying and monitoring of object collections is spatially driven. Users first identify the datacube they want to query. Each datacube has a datacube directory service (DS) which lists all dataflocks *registered* in that datacube. Only dataflocks registered in a given datacube can be the subject of queries in that datacube. In general, very local dataflocks (such as door locks in a building) will only be registered in small datacubes immediately encompassing them. Dataflocks desiring greater visibility will be registered in larger encompassing datacubes. Answering queries in larger datacubes will invariably involve aggregation. For example, a query inquiring information about all taxi cabs in Manhattan may be answered by first providing their numerical distribution over a two-dimensional spatial grid (e.g., blocks); the exact information can be obtained eventually through the process of *query zooming* to the smaller datacubes.

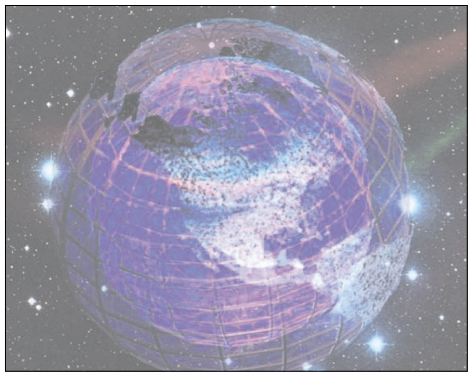

Figure 3. *A 3D physical space shell stretching from 10 km below the surface of earth to 100 km above the surface of earth.*

We view querying as analogous to illuminating the physical space. A datacube is "illuminated" with a message (radiation) carrying a query. Only objects that satisfy the query respond to it ("reflect" the radiation). In larger datacubes, instead of individual objects, one may have sub-datacubes responding with aggregates of responses from inhabitant objects. This has a clear visual analogy: details are usually not visible from far distances; one can zoom in when details are required.

In the next section we describe possible architectures of DataSpace. Later, we define parameters for evaluating any DataSpace design. Subsequently we look at key technical challenges in realizing the proposed DataSpace architecture. Finally, we conclude the article.

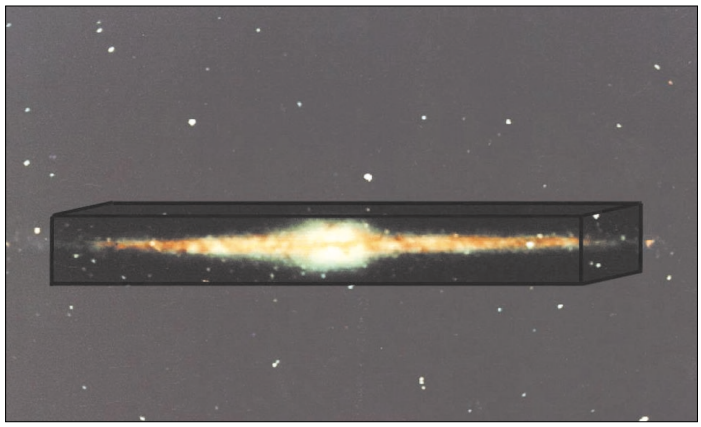

Figure 2. *3D physical space cube encompasssing our galaxy.*

Architecting DataSpace

In this section we describe possible architectures for the DataSpace system.

The Network as a DataSpace Engine

The Basic Approach — In this architecture we propose to provide the basic functionalaties (querying and monitoring) of DataSpace at the network level using multicast mechanisms. "Beaming" a request to a specific datacube (datacube "illumination") such that specific sets of objects respond to it is naturally amenable to implementation via multicasting. For this purpose one may use shared multicast trees [3,4], where multicast group membership for a given datacube is calculated on the basis of its physical location. This way the network itself can take care of ensuring that a given message "illuminates" only a specific datacube. Moreover, the network can also support indexing on selected attributes of dataflocks through multicasting. More precisely, for an indexed attribute, each pair $h = (Attribute, value)$ is mapped to a corresponding multicast address, and all objects which satisfy the predicate ($Attribute = value$) belong to that multicast group. This is analogous to maintaining a list of objects associated with an index entry (through so-called inverted indices) in databases.

Conceptually, it is simpler to think of a one-to-one mapping between the set of pairs $h = (Attribute, value)$ for an *index attribute* and a set of multicast addresses. However, in a practical implementation one may choose a many-to-one mapping, mapping a set of possible values of the index attribute to a distinct multicast address. A many-to-one mapping is especially useful when an index attribute takes continuous values. For example, assume a DataSpace consisting of temperature sensors, and assume that the index attribute for the dataflock of temperature sensors is *temperature*. We believe that queries using this index attribute are more likely to be of the form "*Find all temperature sensors in CoRE building reading temperature between 65 and 75° F*," where a range is specified. In such a case, mapping ranges of temperature values ([54, 64], [65, 75], etc.) to distinct multicast addresses is more efficient.

In the extreme case, one may map all the values of the index attribute to a single multicast address. Choosing a proper mapping is an implementation issue. In the remainder of this article we assume that appropriate mapping is

chosen to handle the common case efficiently, and deliberately leave it unspecified. We use the phrase (*attribute, value*) *pair* to also refer to the case when the attribute is continuous and "*value*" refers to a range of values of the attribute. In such cases, this phrase should be interpreted to be equivalent to the predicate (*attribute IN value*). In cases where "value" refers to a single value of the attribute, the phrase (*attribute, value*) *pair* should be interpreted as equivalent to the predicate (*attribute = value*).

There will be a simple, standard, predefined mechanism in DataSpace for mapping an (*attribute, value*) *pair* to a multicast address. The mechanism may be as simple as a hash function. Choosing an appropriate mapping mechanism is an implementation issue, and we deliberately leave it unspecified.

The process of querying in DataSpace parallels that in a traditional database. In a traditional database, a query is processed by first filtering tuples based on the condition on the index attribute. The tuples that satisfy the condition on the index attribute are then sequentially checked for the satisfaction of the remaining conditions of the query. In a similar way, a query is processed in DataSpace by first filtering the objects based on the condition on the index attribute. This is done by mapping the condition on the index attribute to a multicast address. It is important to note that *only* the condition on the index attribute is mapped to a multicast address; the whole query is *not* mapped to a multicast address. A message containing the query is sent at this multicast address. This message reaches all those objects that satisfy the condition on the index attribute. This way, in DataSpace, multicast serves as an indexing mechanism for this distributed collection of data. The objects that receive the multicast message extract the query from the message and check whether they satisfy the remaining conditions of the query. If they do, they respond with their *id*; otherwise, they silently discard the received message. Thus, the filtering on the index attribute happens at the network layer, while the filtering on the remaining conditions of the query is done at the application layer. We explain the querying process in more detail via an example later.

An important feature of the querying process in DataSpace is that the mechanism for mapping the condition on the index attribute to a multicast address does not need to ensure that it reaches exactly those objects which satisfy the condition. Even if some objects that do not satisfy the condition on the index attribute receive the multicast message, by checking if the objects satisfy all the conditions of the query, application-layer filtering makes sure they do not respond to it.

One may provide the basic functionalities (querying and monitoring) of DataSpace at either the network or application layer. We believe that implementing these functionalities at the network layer has several advantages over implementing them at the application layer. The network layer provides us with an efficient mechanism for sending a message to groups of objects via multicast. An application-layer implementation will have to either use multiple unicast connections or broadcast to achieve the same result. Clearly, performing a single multicast is much more resource-efficient than performing multiple unicasts or broadcast. This makes the network-layer solution much more scalable than an application-layer solution. A network-layer solution delegates the task of providing the basic functionalities to the lower level of the abstraction hierarchy. This allows the network to deal with network-specific issues such as disconnection and router failure, rather than having to reimplement what the network does on the application layer. This is closer to the plug-and-play philosophy of seamless network presence for devices and objects; implementing join/drop/response operations on the lower levels of the network hierarchy makes them easier to standardize and frees an application of an unnecessary burden.

DataSpace Architecture — As stated earlier, DataSpace is a collection of datacubes which are either geometric or administrative. Each datacube has a corresponding *space handle* that encodes its size and its location in 3-D physical space.

Datacubes are populated by dataflocks, which are either static or mobile. Each object (sensor, machine, piece of software, physical object) is characterized by a set of attributes. Dataflocks are subjects of querying or monitoring. Some of the dataflock attributes are supported by *network indices*. By *network index* we mean an attribute for which every pair (*attribute, value*) has a corresponding multicast address, and all objects which satisfy the predicate (*attribute = value*) are members of the corresponding multicast group. Pairs h = (*attribute, value*) such that the attribute is *indexed* are called *subject handles*. Network indices are used in processing DataSpace queries the same way database indices are used in processing database queries [5,6]. Thus, the network in effect serves as an index of this distributed collection of physical objects.

A query is encoded as a DataSpace address with two components: a *space handle*, which identifies a datacube, and a *subject handle*, which denotes an individual predicate (on the index attribute) extracted from the query. Querying is effected by sending a multicast message on its corresponding DataSpace address. Objects addressed by the query respond to it by sending their identifiers. These DataSpace addresses occupy a part of the IPv6 [7] multicast addressing space.

Each datacube has its own local directory service (DS), which contains entries for the dataflocks registered in that datacube. Each object can be registered in many enclosing datacubes.

Each query q is associated with two groups of objects: one that satisfies q, and another, denoted by *Interested*(q), that monitors all changes to the membership of query q. 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 *Interested*(q) group are themselves a dataflock and may be queried.

Since the members of *Interested*(q) monitor updates to the membership of query q, they know the answer to query q. A few of these members may offer this membership information to anybody who is interested in finding the answer to query q. Such members are called *brokers* of query q. Any 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 answers do not change very much over time. In general, for such queries it is more resource efficient to cache the answer at the broker and use it to service the request of the querying objects.

An Example — To illustrate the concepts presented, let us consider an example. Consider a DataSpace consisting of multimodal sensors for measuring temperature and humidity. Assume that one would like to process the query:

"*Find all temperature sensors in CoRE building reading temperature between 65 and 75° F, and relative humidity between 50 and 55 percent*."

Assume that *temperature* is the index attribute for the dataflock of multimodal sensors. Figure 4 illustrates how the query would be translated into a DataSpace address (refer to [5, 6] for details). The spatial constraint on the query is mapped to a space handle using the Global Positioning System (GPS) [8] coordinates of the region enclosing the CoRE building. The condition involving the index attribute is mapped to a *subject*

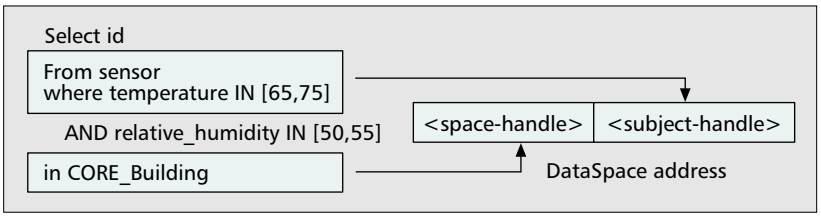

Figure 4. *An example illustrating the mapping operation in DataSpace.*

handle. The resulting DataSpace address is formed by clubbing the space and subject handles together.

The query is processed by sending a multicast message on the DataSpace address. The message contains the query in its body. This query reaches all sensors located in the CoRE building that satisfy the condition (*temperature IN* [*65*, *75*]). The application running on these sensors receives the message. It checks whether the sensor satisfies the remaining conditions of the query. If it does, it sends back the id of the sensor; otherwise, it silently discards the received message.

In summary, in this architecture a query is filtered at two levels: at the network layer based on the physical scope of the query, and on the indexed attribute of the query. The second level of filtering happens at the application layer.

DataSpace on Top of a Geographic Routing Infrastructure

Another possible architecture for DataSpace uses a geographic routing [2] infrastructure as the base layer, and builds DataSpace on top of it. This architecture is different from the previous one in only one aspect: it does not make use of interdomain multicast routing. Instead, it relies on a combination of geographic routing and link-local multicasts to perform querying and monitoring operations. In this section we only describe the changes from the previous architecture.

Geographic routing [2] aims at routing a message based on the geographical location of the destination rather than the IP address. The destination may even be a region bounded by a set of GPS coordinates, in which case the message is broadcast in the specified region. The geographic routing infrastructure consists of three types of elements in order to accomplish this goal: *geographic router*, *geographic node*, and *geographic host*. A geographic router (geo-router) routes messages based on the geographic location of the destination region. A geographic node (geo-node) is an entry/exit point for the routing system. The main function of a geo-node is to store incoming geographic messages for the duration of their lifetime and to periodically multicast them on all of the subnets or wireless cells to which it is attached. A geographic host (geo-host) is located on all objects capable of receiving and sending geographic messages. Its role is to notify all applications running on the object about the availability of geographic messages, to determine the object's current geographic location, and to determine the address of the local geo-node.

The Basic Approach — This architecture is based on the observation that a querying operation in DataSpace can be decomposed into two stages:

- Routing the query to the specified destination datacube
- Issuing the query in the destination datacube

In the first stage, the message containing the query is routed based on the geographical location of the destination datacube. For this purpose, we make use of the geographic routing base layer. The geographic routing infrastructure routes the message to the set of geo-nodes responsible for the datacube specified in the query. In the second stage, the query is issued within the destination datacube. In order to accomplish this, geo-nodes extract the query from the received message, map it to the corresponding DataSpace address, and send a multicast message at this address with a time to live (TTL) of 1. The message contains the query in its body. The DataSpace address is computed from the query in the same way as described in the previous architecture.

Similarly, the operation of monitoring a query can be decomposed into two stages. We elaborate on this below.

DataSpace Architecture — In this architecture for DataSpace, every device runs a geo-host. If a device has limited capabilities, it can be configured to contact a proxy running geo-host. We change the behavior of geo-nodes slightly to support DataSpace operations. If the incoming geographic message is a query, a geo-node transmits it only once. This is done to preserve the semantics of querying as an operation on the snapshot of the system state. On the other hand, if the incoming geographic message is a request for monitoring a query, a geo-node stores the incoming message and periodically multicast it.

As described earlier, a querying operation is performed in two stages: in the first stage, geo-routers route the query to the geo-nodes responsible for the specified datacube. In the second stage, geo-nodes map the query to its corresponding DataSpace address. The method for mapping a query to a DataSpace address is the same as described in the previous architecture. Geo-nodes multicast a message (containing the query) at this address using a TTL of 1. This message reaches all the objects located in the specified datacube that satisfy the condition on the index attribute. The geo-hosts running on these objects check if the object satisfy all the conditions of the query, and respond with the object's id if they do.

The operation of monitoring a query can also be decomposed into two stages:

- Routing the request for monitoring a query to the datacube specified in the query
- Performing a monitoring operation in the destination datacube

The geographic routing infrastructure is a natural fit for performing the first stage. It delivers the message to the set of geo-nodes responsible for the destination datacube. To perform the second stage, a geo-node maintains a state for every query being monitored. The state contains the query being monitored and a list of IP addresses of senders interested in monitoring that query. For every monitored query it multicasts a *keep-alive* message. The multicast address is computed from the query in the same way described in the previous architecture. Such keep-alives reach all devices in the datacube that satisfy the query. They indicate to the devices in the datacube that some objects are interested in monitoring the answer to the query. On receiving such keep-alives, devices start sending updates to the geo-node, which in turn sends them to the interested objects. The timescale of these updates may be very different from the frequency at which keep-alives are issued (keep-alives are sent at very low frequency). Devices stop sending their updates if they don't receive a keep-alive for a "long" time. The state maintained by geo-nodes is "soft" and needs to be refreshed periodically by the objects monitoring the query.

An Example — To illustrate the concepts presented, let us consider an example. Consider again a DataSpace consisting of multimodal sensors for measuring temperature and humidity. Assume that one would like to process the query:

"*Find all temperature sensors in CoRE building reading temperature between 65 and 75° F, and relative humidity between 50 and 55 percent*."

In this architecture, the query is first routed to the set of

geo-nodes responsible for the CoRE building. These geo-nodes map the query to a DataSpace address as shown in Fig. 4. They use this address to send a multicast message with a TTL of 1. This message contains the query in its body. It reaches all sensors located in the CoRE building that satisfy the condition (*temperature IN* [*65*, *75*]). The geo-hosts running on these sensors receive the message and check if the remaining conditions of the query are also satisfied. If they are, geo-hosts respond with the id of the sensor; otherwise, they silently discard the received message.

In summary, in this architecture a query is filtered at three levels. The geographic routing layer filters the query based on the physical scope of the query. The network layer filters the query based on the indexed attribute of the query. The final level of filtering happens at the application layer.

Observability and Awareness

The two architectures for DataSpace presented in this article are not the only ones possible. In fact, they are among many possible architectures for DataSpace. In this section we identify two quality-of-service parameters for characterizing any architecture for DataSpace: *observability* and *awareness*.

A query q is R-observable at an object o if o can be provided with the answer to q such that the answer satisfies restriction R. Restriction R specifies certain assurances on the quality of the answer. For example, consider a DataSpace of sensors sensing temperature in all parts of a building. Consider the query: "*Find all sensors reading between 200 and 250˚F.*" If a querier receives a response from all sensors satisfying this query within 60 s, then this query is said to be *60-second-observable* at this querier.

An object o is R-aware of a query q if at any time it knows the current answer to q, where current is defined by restriction R. Restriction R specifies certain assurances on the quality of the answer known at o. For example, in a DataSpace of sensors sensing temperature in all parts of a building, the building administrator is said to be *5-s-aware* of the query "*Is there fire in any part of the building?*" if he receives a triggered/periodic update within a maximum of 5 s of time anytime some portion of the building catches fire.

R-observability and R-awareness reflect the need to define approximate (e.g., *50-percent-complete-observable*) query answering and query monitoring in volatile networked environments with varying reachability, disconnections, and so on.

Challenges

In this section we discuss the key technical challenges that must be addressed before the proposed DataSpace architectures can be realized:

- **Monitoring operation: issues** — The primary concern is how to keep the overhead associated with sending updates to a minimum while still ensuring that the objects monitoring a query have the most up-to-date answer. This goal is especially challenging in the face of failures of objects and loss of updates in the network.
- **Brokers: issues** — The presence of brokers in our system presents the following challenges:
  - –It is not always resource-efficient to cache the answer of a query. This is because a cached answer requires receiving constant updates in order to keep the answer current. These updates represent an overhead that may far exceed the savings gained by not querying the objects directly. How does a broker decide which queries are worth caching and which are not? The problem is compounded by the fact that the set of queries worth caching changes over time.
  - –How to architect brokers so that they scale well with respect to their load.
  - –How to support *Query Zooming*. A related issue is that different queries require different aggregation functions. How does a broker know which aggregation function to use for a particular query?
- **Supporting observability and awareness**: In the TCP/IP protocol suite, the network layer offers best-effort connectionless service. Supporting *observability* and *awareness* with some non-null guarantees is an open issue. Even in networks with QoS support, it is not clear how the QoS at the network layer translates to guarantees on the answer to a particular query, or on the knowledge of the answer at any given time to a particular query when performing monitoring.
- **Other issues:**
  - –*Response implosion*: A DataSpace comprises a huge number of objects. It is very likely that a querying object is flooded by responses, since it may be common for some queries to be satisfied by a large number of objects. We call this the *response implosion* problem. We have already described two possible mechanisms for dealing with this: having *brokers* and *query zooming*. Other possible solutions include *samplecast* and *gathercast* [9].

In samplecast, an object that satisfies query q responds with probability p. The querying object 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', and obtaining a bigger sample of the answer.

In Gathercast [9], a number of small response packets with the same destination are combined into one larger packet. This combining operation is done at the routers as the responses flow toward their destination. This reduces the number of response packets received by the querier.

- –*Nearcast*: If an object is querying in DataSpace in order to discover services (e.g., discover a broker for a query), it would certainly like to contact the nearest available one. This suggests a need to be able to nearcast. The challenge is how to implement a nearcast mechanism in a manner scalable to the potential size of DataSpace.

Summary

We have defined a new conception of three-dimensional *DataSpace* which is physical space enhanced with connectivity to the network. DataSpace is a collection of datacubes often populated by classes of mobile objects called dataflocks. Objects in DataSpace produce and store their own data. Such objects can be queried and monitored on the basis of their properties. DataSpace is the next generation of the World Wide Web, with two major differences: it is embedded in physical reality, organized in a geographic/spatial manner rather than logically as the Web is today; and it supports a huge number of objects that produce and store their own data and move through DataSpace. While browsing is the main navigation mechanism for the Web, querying and monitoring are the main navigation mechanisms for DataSpace. In this article we mention the concepts required to implement these mechanism in the context of DataSpace. We describe mechanisms to use the network as a DataSpace engine to perform querying and monitoring operations at the network layer. We also describe possible architectures of DataSpace and introduce two quality-of-service parameters, observability and awareness, in order to evaluate any DataSpace design. Finally, we present key technical challenges in order to realize our proposed architecture.

Acknowledgments

We are grateful for discussions and constant encouragement from B. R. Badrinath. We would like to thank Shirish Phatak for providing comments on an earlier draft of this article. Thanks are also due to the anonymous reviewer for providing helpful comments.

References

[1] Sun Microsystems Inc., "Jini Architecture Specification," Jan. 1999; http://www.sun.com/jini/specs/jini-spec.ps.
[2] J. C. Navas and T. Imielinski, "Geographic Addressing and Routing," *Proc. 3rd ACM/IEEE Int'l. Conf. Mobile Computing and Networking*, Budapest, Hungary, Sept. 1997.
[3] T. Ballardie, P. Francis, and J. Crowcroft, "Core Based Trees (CBT): An Architecture for Scalable Inter-Domain Multicast Routing," *Proc. SIGCOMM*, 1993.
[4] S. Deering *et al.*, "The PIM Architecture for Wide-Area Multicast Routing," *IEEE/ACM Trans. Net.*, vol. 4, no. 2, Apr. 1996, pp. 153–62.
[5] T. Imielinski and S. Goel, "Dataspace - Querying and Monitoring Deeply Networked Collections of Physical Objects, Part I — Concepts and Architecture," Tech. rep. DCS-TR-381, Rutgers Univ., July 1999.
[6] T. Imielinski and S. Goel, "Dataspace - Querying and Monitoring Deeply Networked Collections of Physical Objects, Part II — Protocol Details," Tech. rep. DCS-TR-400, Rutgers Univ., July 1999.
[7] R. Hinden and S. Deering, "RFC2373: IP Version 6 Addressing Architecture," July 1998; ftp://ftp.isi.edu/in-notes/rfc2373.txt.
[8] "GPS SPS Signal Specification," 2nd ed., June 1995, http://www.navcen.uscg.mil/gps/geninfo/gpsdocuments/sigspec/default.htm
[9] B. R. Badrinath and P. Sudame, "Gathercast: An Efficient Multi-Point to Point Aggregation Mechanism in IP Networks," Tech. rep. DCS-TR-362, Dept. of Comp. Sci., Rutgers Univ., July 1998.

Biographies

TOMASZ IMIELINSKI (imielins@cs.rutgers.edu) is professor and chair of the Dept. of Computer Science, Rutgers Univ., New Brunswick, NJ. He received his Ph.D. from the Polish Academy of Science (Warsaw) in 1982. His current interests include database mining and mobile wireless computing. He has published nearly 100 articles and two books on these subjects. He is co-editor of the book *Mobile Computing* (Kluwer, 1996). In 1999 he was Program co-Chair for ACM/IEEE Mobicom '99, the primary conference in the area.

SAMIR GOEL (gsamir@cs.rutgers.edu) is currently a research assistant in the Computer Science Department at Rutgers University, New Brunswick, New Jersey. He received his Master's degree from the Computer Science Department of the Indian Institute of Technology, Kanpur, in 1997. His research interests include mobile computing and IP multicast.