ABSTRACT

Title of Dissertation: REPLICATION TECHNIQUES FOR PEER-TO-PEER NETWORKS

Bujor D. Silaghi, Doctor of Philosophy, 2003

Dissertation directed by: Professor Peter J. Keleher
Department of Computer Science

A close examination of presently deployed peer-to-peer networks (P2P) and existing proposals reveals several trends regarding data management. First, only a small percentage of the queried data is actually retrieved by users. Second, the vast majority of data exported by participants in P2P networks is unaltered after creation. Third, membership changes in P2P networks (users/sites leaving or joining the network) occur on a scale not seen before in distributed systems.

We address the challenges posed by the above observations using two data replication techniques. First, we define a routing protocol and replication model for hierarchical namespaces that adaptively replicates routing information in response to fluctuating load incurred by the lookup and search procedures. Second, we define a replica control protocol (d-spaces) based on quorum consensus in multi-dimensional spaces that is well suited for scenarios where reads are the dominant type of access to data. We

extend the basic d-space protocol with a quorum reconfiguration mechanism which uses local connectivity and cached quorum groups instead of global views to achieve seemingly contradicting goals: improved fault tolerance through local reconfiguration policies, and good global latencies.

Our main contributions are as follows: We show that hierarchical bottlenecks can be eliminated, with the resulting system behaving like a true symmetrical P2P network. Temporal locality in the stream of requests (hot-spots) is managed by replicating on demand popular portions of the namespace. We argue that non-virtualized object namespaces maintain desirable properties that are lost through virtualization. We present a token-based method of resolving complex search queries through hierarchical query decomposition and result recomposition. The method allows the initiator to ascertain query completion, and offers a broad range of trade-offs between network bandwidth consumption and the number of connections required to complete the search. We show that d-space quorums have optimal communication complexity. We define the Read-Few Write-Many replica control protocol and argue that the quality of trade-off between read efficiency and update availability is not matched by existing replica control protocols. Finally, we present a quorum reconfiguration mechanism that allows membership changes without requiring global reconfiguration phases.

REPLICATION TECHNIQUES
FOR PEER-TO-PEER NETWORKS

by

Bujor D. Silaghi

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2003

Advisory Committee:

Professor Peter J. Keleher, Chair and Advisor
Professor Virgil Gligor, University Representative
Professor Samrat Bhattacharjee
Professor Samir Khuller
Professor Sudarshan Chawathe

ACKNOWLEDGEMENTS

I thank my academic advisor Peter Keleher for his continuing support and professional guidance, and for giving me the opportunity to learn what research is all about. I also thank my de facto co-advisor Samrat Bhattacharjee for the many and engaging brainstorming sessions that he made me be part of. I am professionally indebted to both of them.

I thank Vijay Gopalakrishnan for being a dependable and fun colleague to work with. To Vijay along with Gabe Rivera, Iulia Ungureanu, Omer Horvitz, Margaret and John Shilling, Catalin Popescu, Houman Alborzi, Diana and Cristian Tapus, Rob Sherwood, Ioana Rus, Vasile Gaburici, Ray Chen, Anamaria and Martin Segesten, and many, many others, the warmest of thanks. You have made for a wonderful company throughout my years at Maryland.

Last but not least, I thank all the professors whose classes I have heartily enjoyed attending (Yiannis Aloimonos, Sudarshan Chawathe, David Cooper, Virgil Gligor, Peter Keleher, Samir Khuller, Clyde Kruskal,

David Mount, William Pugh, and Nick Roussopoulos), the thesis committee members for having contributed to the validation of my work, Gwen Kaye, Udaya Shankar, and finally the staff in the department and at the Graduate School for making it all possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

With the recent advent of peer-to-peer networks (P2P), managing replicated information has become once again a priority topic for the research community. The current interest has at its roots both classical concerns (e.g. data availability and access latency), as well as new directions specific to peer-to-peer environments (user anonymity, non-erasable data repositories, etc.).

Replication is used in distributed systems to increase data availability, to improve the access efficiency to data items, and to distribute the workload across participating sites. Storing copies of data at multiple sites increases the likelihood of data remaining accessible to users despite host and communication failures. Having multiple up-to-date copies of data offers a choice at retrieval time, and allows the system to efficiently fetch a copy that is locally present or at a nearby site. Data replication often implies replicated functionality and thus may offer a choice of distributing processing load across functionally equivalent sites. Replication can also be used in peer-to-peer networks to meet other quality of service requirements such as user anonymity and non-erasable data repositories.

A close examination of presently deployed P2P networks and existing proposals, reveals two trends as far as data management is concerned. First, we note that only a small percentage of the data looked-up or searched for is actually retrieved by users. While the reasons for this can be multiple —users may not know exactly what they are looking for and perform keyword-based searches, some data may be replicated and the user is satisfied with any one copy, or the lack of preference for semantically related but still different data— the implication is that it makes sense to distinguish between processing load incurred by lookups and searches (*routing load*), and load due to the actual retrieval of data. Second, we note that the vast majority of data exported by participants in P2P networks is unaltered after creation. Hence, replica control protocols in this setting should favor the execution of read operations as they are expected to occur orders of magnitude more often than updates. We also note that *membership changes* in P2P networks (users/sites leaving or joining the network) occur on a scale not seen before in distributed systems. A session may last for as little as a few minutes and a principal (user or site) may initiate as many as a few tens of sessions per day [SGG02]. Existing quorum reconfiguration protocols rely on some form or other of global knowledge and are not designed to withstand radical membership changes. Methods of approaching quorum reconfiguration by using local information and lazily inferring global knowledge only to the extent needed, seem to present a more feasible alternative.

In this dissertation we address these issues by:

- Describing a routing protocol and replication model for hierarchical namespaces that adaptively replicates routing information in response to fluctuating load incurred by the lookup and search procedures.

- Defining a data replica control protocol based on quorum consensus in multi-dimensional spaces that is well suited for scenarios where reads are the dominant type of access to data. We extend the basic replica control protocol with a quorum reconfiguration mechanism whereby local connectivity and caching quorum groups are used instead of global views to achieve seemingly contradicting goals: local reconfiguration policies and good global latencies.

The protocols presented in this dissertation were studied and developed in the context of TerraDir, which is an overlay network based on hierarchical namespaces [BKS01]. The routing protocol presented in the first part is used in TerraDir as the location service for lookup and search queries, and can be integrated with any data replica control protocol. The replica control protocol presented in the second part defines a general solution with wider applicability and can be used as the data replica control protocol in TerraDir. Throughout our dissertation we argue for the choice of hierarchical overlay network (TerraDir) chosen to illustrate the protocols.

In the following sections we present an overview of the dissertation and discuss our main contributions.

1.1 Hierarchical Routing with Soft-State Replicas

TerraDir is an overlay network that defines hierarchical namespaces among its participating nodes. Connectivity in the network is maintained on a per node basis, with each node knowing its immediate ancestors and descendants. TerraDir enjoys a powerful query model that allows simple node lookups, as well as more complex search queries based on node attributes and partially-qualified node specifiers. In the first part of this dissertation we define and evaluate TerraDir namespaces, the query model, and its associated routing procedure.

Our study is performed in the context of hierarchical namespaces. However, the replication model can be applied to any peer-to-peer network that maintains direct connectivity among its peers. On the other side, virtualizing the object namespace is not compatible with our approach. Mapping objects and/or servers into an abstract space using pseudo-random functions (*namespace virtualization*) destroys the routing context that is maintained through direct connectivity. As examples of overlay networks that virtualize the object namespace we identify approaches based on the Distributed Hash Table (DHT) paradigm [SMK$^+$01, RFH$^+$01, RD01a].

The replicated information consists of a node's identifier and its routing context. Routing a query on a hierarchical namespace is performed with *incremental progress*: at each step the query gets closer to the destination by at least 1 in the namespace distance metric. Replicating a node's routing context guarantees that routing through a replica of a node is functionally equivalent to routing through the node itself. The implication is that, assuming accurate replica information, the routing procedure guarantees incremental progress toward the destination when integrated with the replication protocol.

We evaluate our approach given the following goals:

- *Local Information and Scalability:* Exclusive use of local information in making routing and replication decisions is imperative if we are to target decentralized systems. Similarly, large-scale deployment in wide-area environments makes scalability a first priority in building the routing and location layer of P2P networks.

- *Adaptation vs. Stabilization:* The resulting system must be highly adaptive to varying conditions such as availability of system resources, and most notably to changing or skewed patterns in user input. Additionally, fixed-point states have to be reached for stabilized scenarios. Finally, we are not willing to sacrifice efficiency for the sake of adaptivity.

- *Fairness:* Many P2P systems are and will be used to present services to end users. End users are often skeptical of services that consume local resources to support anonymous outside users. User acceptance is often predicated on the extent to which end users feel they have fine-grained control over the intrusiveness of the service.

- *Fault Tolerance:* Replication is redundant by its nature. We expect a routing procedure that uses dynamically replicated state to be highly resilient to server or network failures. Ideally, the model should be able to detect weak points caused by failures, and replicate routing state hosted in critical areas of the network.

- *A Posteriori Information:* Any model that is to cope with unpredictable or changing input patterns need not rely on information other than that collected on-the-fly. Information profiling is a center component of our scheme and anchors on the assumption that recent past behavior is a reasonable approximation for short-term future behavior.

1.1.1 Contributions

Our major contributions in this part of the dissertation are as follows:

- We show that all the above requirements can be met for hierarchical structures by employing a lightweight adaptive replication model, and that the model's implementation in real systems is feasible. Our approach combines a solution for two related problems. First, processing bottlenecks that are specific to hierarchical namespaces are eliminated with the resulting system behaving like a true symmetrical peer-to-peer network. Second, temporal locality in the stream of requests (*hot-spots*) is managed by replicating on demand popular portions of the namespace.

- We define a method of resolving complex search queries through hierarchical query decomposition and result recomposition. The novelty of our approach stems from a token distribution mechanism that allows the query initiator to ascertain query completion. The scheme is flexible and offers the whole range of trade-offs between network bandwidth consumption and the number of open connections required to complete the search.

- We argue that having the object namespace define the peer-to-peer connectivity is central to the applicability of our replication model. Overlay networks that virtualize the namespace cannot benefit from a similar mechanism of replicating routing information. More generally, we argue that a non-virtualized object namespace (e.g. hierarchies) maintains properties that are lost through virtualization. In particular, virtualization destroys locality, and therefore opportunities for efficient browsing, pre-fetching and searching are lost. Further, virtualization discards useful application-specific information, such as object relationships that would be otherwise exposed by a hierarchical namespace.

Many of the above arguments call for hierarchical overlay networks for applications where data is naturally described using hierarchies (e.g. file-systems, resource discovery, auctions). This is in fact a recurring point of our dissertation. Having addressed the inherent problem of hierarchies (i.e. hierarchical bottlenecks), we finally argue that for such applications, hierarchical namespaces are the preferred alternative.

Replicating routing information in the sense presented here requires no consistency models be specified. New replicas can be created and existing replicas discarded in a decentralized manner without enforcing consistency or notification requirements. In this sense the model manages only soft-state information. Further, the model does not specify how data exported by users is managed, whether or not it is replicated, and how it is replicated. In the second part of this dissertation we define a replica control protocol that addresses data replication in TerraDir.

1.2 Multi-Dimensional Quorums Sets for Read-Few Write-Many Replica Control Protocols

The benefits of replication come at the price of complex and expensive synchronization mechanisms needed to maintain the consistency and integrity of data. The overhead incurred by replica control protocols is in direct relation with the consistency guarantees offered. The stronger these guarantees are, the less efficient and with lower availability are operations performed on replicated data items. Ideally, the set of all copies of an object should appear as a single item to clients. Further, efficient schemes are desirable in the sense that only a small fraction of the set of all copies should be accessed per operation. This is especially important for read operations, which tend to represent the dominant access mode to data for many application classes. Finally, we would like to achieve both strict consistency and operational efficiency without sacrificing the availability of data, as this is the main reason for replicating data.

Many of these requirements (read efficiency, update availability, and strict correctness) are emerging for applications that belong to the classical distributed systems domain, as they are being migrated or re-engineered to work in a P2P environment. Distributed databases is one such domain, and the PIER project [HHL$^+$03] probably the most notable effort in this direction. The consistency condition all-copies-like-one referred to above is termed *one-copy equivalence* [BG87]. Coupled with serializable executions that many databases provide, the correctness criteria for replica control protocols in transactional environments is *one-copy serializability* [BG84]. Quorum consensus protocols [Gif79] are among the replica control protocols that can achieve one-copy serializability — when used in conjunction with appropriate mechanisms for concurrency control (e.g. two-phase locking), and atomic commitment (e.g. two-phase commit).

Synchronization in quorum consensus replica control protocols takes place by defining groups of sites that need to agree before launching an activity, and requiring the intersection of groups defined for conflicting activities. A read operation on a copy conflicts with all write operations on any copy of the object. A write operation on a copy conflicts with all read and write operations. We will refer to the group of sites needed to perform a read (write) operation as the *quorum group* for that operation. The collection of read (write) quorum groups is called a read (write) *quorum set*. Thus, any element of a read quorum set must intersect all elements of a write quorum set, which in turn must intersect among themselves in a pairwise fashion.

In this part of the dissertation we present *d-space*, a quorum consensus protocol for replicating data in distributed systems. Sites participating is d-space quorums are arranged in a multi-dimensional space, and quorums groups are formed through subspace projection. The protocol is a generalization of the Grid replica control protocol [CAA92] to multiple dimensions. It is also reciprocal to Grid in the sense that it defines inverted read and write quorum groups with respect to subspace covers. The motivation for d-space quorums is the commonly accepted observation that read accesses to data occur orders of magnitude more often than updates. We believe that read accesses will be a defining characteristic of usage in future distributed systems, including peer-to-peer architectures. Presently deployed file-sharing utilities fit this description, with the vast majority of data exported by participants unaltered after creation.

1.2.1 Contributions

Our major contributions in the context of replica control protocols are as follows:

- We define the *read/write access ratio* as the proportion in which read and write requests are submitted to the system. We show that d-space quorums can naturally accommodate all access ratios of interest and show how to define quorum groups to meet any access ratio. We prove that, for any access ratio, d-space quorums are optimal with respect to communication complexity. In other words, the method uses the minimum number of messages per read or write operation such that the given access ratio is met. Our proof involves prerequisite fairness conditions regarding the distribution of sites in quorum groups.

- We define the Read-Few Write-Many (RFWM) replica control protocol as a special case of d-space quorums. RFWM approximates ROWA with respect to read efficiency while maintaining much better levels of availability for write operations. We show that even for highly skewed access ratios (where reads occur orders of magnitude more frequently than writes), the availability of updates can approximate or even match the availability of read operations. We argue that the quality of trade-off between read efficiency and update availability in RFWM is not matched by existing quorum-based replica control protocols. In particular, we show that implementing a read-few write-many protocol using d-spaces yields superior operational availability as well as message complexity to the hierarchical quorum consensus method.

- We advocate using local connectivity instead of global views to arrange participating sites into a logical structure. This addresses the inherent rigidity of logical structures based on global views, and allows membership changes without requiring global reconfiguration phases. Local connectivity is combined with a relaxed best-effort form of global views (cached quorums) to achieve seemingly contradicting goals: improved fault tolerance through local reconfiguration policies, and good global latencies.

1.3 Organization of the Dissertation

The remainder of the dissertation is organized as follows. Chapter 2 presents a survey of the existing relevant literature both in the context of hierarchical topologies and peer-to-peer networks, as well as in the context of replica control protocols and quorum-based approaches. Chapter 3 describes and evaluates TerraDir namespaces and the replication model that is part of the routing procedure on hierarchical namespaces. Chapter 4 defines d-space quorum, elaborates on read-few write-many replica control protocols, and explores a local quorum reconfiguration protocol. We conclude our dissertation and outline future research directions in Chapter 5.

Chapter 2

Literature Survey

We present a survey of relevant approaches and systems including P2P file-sharing systems, wide-area directories and distributed systems that feature hierarchical namespaces or hierarchical routing procedures, overlay networks using distributed hash tables to locate content, query locality in P2P networks, quorum consensus protocols for managing replicated data, and finally quorum reconfiguration mechanisms to improve fault-tolerance.

2.1 Peer-to-Peer File Sharing Applications

Among the deployed P2P file-sharing systems that have received considerable attention from the research community we identify Napster, Gnutella, Morpheus and Freenet.

2.1.1 Napster

Napster [nap] is a music exchange service and provided much of the original motivation for peer-to-peer research even though it is not in itself a pure P2P system. More exactly, it maintains a centralized database of indices that clients can query, while the actual transfer of files among clients is done in a decentralized manner. Napster is thus exposed to the same problems that most other centralized services are: single point of failure, centralized and single control authority, and most importantly it is not guaranteed to scale well for arbitrary number of clients.

2.1.2 Gnutella

Gnutella [gnu, gnu00] was designed as a replacement of Napster and unlike its predecessor it features a pure peer-to-peer file sharing service. Each participant in Gnutella maintains a number of arbitrary neighboring sites, and is able to reach any other member by successive hopping in the network thus created. However this routing process is not informed and the retrieval is done simply by having each site broadcast queries to all its neighboring sites. While the extent of the search is controlled by a TTL parameter, the search is still very resource intensive and clearly not scalable beyond a certain point. Furthermore, due to fact that only sites within a specified hop-radius from the query initiator are visited, the service does not provide reliable content location. Even though modest improvements can be achieved by limiting the extent of network flooding [YGM02], the lack of object location guarantees is conceptual in Gnutella's case.

Recent work [LCC$^+$02, CS02] considers static replication in combination with a variant of Gnutella searching using k random walkers. The authors show that replicating objects proportionally to their popularity achieves optimal load balance, while replicating them proportionally to the square-root of their popularity minimizes the average search latency. We addressed a static replication approach for TerraDir in previous work [SBK02].

2.1.3 Morpheus

Morpheus [mor] features an architecture that is somewhere between Napster's centralized database and Gnutella's broadcasting search. A number of distinct nodes (*supernodes*) are connected through a peer-to-peer network, while each of these nodes is accessed in a centralized manner by a few other nodes that act as satellites. Morpheus partially addresses the major problems with Napster and Gnutella, but it also borrows to a lesser extent some undesirable features from both.

2.1.4 Freenet

Freenet [CSWH00] is a distributed information storage and retrieval system designed to address concerns related to user privacy and data availability. Files are referred to in a location-independent manner, and are dynamically replicated in locations near requesters and deleted from locations where there is less interest. Requests for file retrieval operate as hill-climbing searches with backtracking. Freenet's routing performance is based on the assumption that nodes tend to specialize over time in locating sets of similar file keys, and in storing clusters of files having similar keys as well. Freenet does not provide reliable content location and by automatically deleting content of little or no interest it cannot even make guarantees on how long a given file resides in the network.

All of the approaches above feature a flat object namespace while TerraDir defines a hierarchical namespace. Searching in flat namespaces is ideal if the client has little semantic information about the requested objects, aside from names and possibly a few attributes. A hierarchical structure exhibits richer naming and searching semantics, and enables additional operations that are specific to networks of linked objects (e.g. browsing). Note that flat searching primitives can be easily implemented on hierarchical namespaces. On the other side, it is rather difficult to efficiently emulate a hierarchical structure on top of a flat namespace service. Other undesirable characteristics of existing file-sharing services like unreliable content location, network flooding and centralized indexes make them less attractive for our purposes.

A recent analysis [SGG02] of two popular P2P file sharing systems (Napster and Gnutella) concludes that the most distinguishing feature of these systems is their heterogeneity. We believe that the adaptive nature of TerraDir's replication model makes it a first-class candidate in exploiting system heterogeneity.

2.2 Wide-Area Directories

Two landmark wide-area directory protocols are the Domain Name System (DNS) and X.500 together with its derivatives.

2.2.1 The Domain Name System

DNS [Moc87a, Moc87b] is a cornerstone for the Internet and one of the most widely deployed directory service to date. The design of DNS was a careful compromise between functionality, scalability, and implementability [MD88] — years of experience with DNS has shown that almost all of the original design choices were the correct choices.

DNS is not flexible enough to be used by arbitrary applications since it does not provide for configurable consistency, fault tolerance, or sophisticated searches. Thus even though current DNS servers are required for their name resolution functionality, the deployed infrastructure is not provisioned to handle queries for arbitrary RRs from client applications. In fact, it has proven somewhat difficult to upgrade the existing

DNS infrastructure to support even Secure DNS [EK97].

The TerraDir routing procedure is similar to the one in DNS except that, upon failure to resolve a name locally, we do not necessarily visit the upper levels of the namespace. Furthermore, the TerraDir query model allows for complex searches featuring regular expressions and attribute qualifiers. The key to the success of DNS lies in a carefully tuned caching scheme that enables queries to be resolved in the local domain. TerraDir caches consist of pointers in the namespace and provide only routing functionality.

2.2.2 X.500, DAP, and LDAP

X.500 [X5088c, X5088b] is an international distributed directory standard jointly developed by the ISO and CCITT. The X.500 *Standard* includes hierarchical namespaces, a class-based hierarchical object model, a hierarchical attribute model, and an administrative information model. X.500 also defines a family of internal protocols including DAP (Directory Access Protocol) [X5088a] used to query a directory, DSP (Directory System Protocol) used to cooperatively answer a query, DOP (Directory Operational Bindings Management Protocol) used to set up long-term peering relationships between X.500 peers, and DISP (Directory Information Shadowing Protocol) used to replicate directory nodes. X.500 also defines an internal user authentication protocol and a master-slave consistency model for replicated data.

X.500 is a protocol well-suited for federated international directories for large organizations, but imposes far too much overhead to be useful in our context. TerraDir is designed to work across a much wider range of connectivity: inter-server links are implicit and failures are detected only when a server that owns a requested node cannot be accessed during a query.

This model of extremely relaxed connectivity requirements is directly counter to the requirements of X.500 which is designed for long-term (on the timescale of years) inter-server relationships that are negotiated using the DOP. Even DAP, the basic access protocol was so resource intensive that the *de facto* standard for accessing X.500 is LDAP (the Lightweight Directory Access Protocol [YHK93]). LDAP does not incur the presentation and session layer overheads of DAP.

To summarize, DNS is not flexible enough to support other applications beyond the service for which it was originally designed, while X.500 and its derivatives impose a far too complex protocol to be useful for our purposes.

2.3 Hierarchical Routing

Globe and Grig/GLS are distributed systems that feature hierarchical routing procedures.

2.3.1 Globe

Globe [vSHT99] is an object-based worldwide distributed system aimed to support a billion users each owning thousands of objects. The primary assumption of the designers was that a large fraction of the objects of interest (software components, hardware, or a combination thereof) are mobile and may change location quite frequently. To provide location independence, Globe employs a naming service to map object names into globally unique object handles, and a location service to bind the object handle to specific addresses [vSHHT98, vSHBT98].

The location service is implemented through a hierarchical decomposition of the (worldwide) network into regions, and associating a directory node with each such region. Addresses are normally stored at the leaf node representing the region in which the address lies. For each object a path of forwarding pointers is maintained from the root node to the leaf where the corresponding address is stored. Part of this path needs to be updated every time an object leaves or joins the network, or when it changes regions by moving. Moreover, depending on the position of the client and the position of the requested object, requests are forwarded upwards potentially all the way to the root, and then downwards following the chain of pointers to the object's region.

Having to maintain a chain of pointers for each object is the price Globe pays for decoupling the naming of objects from their addresses and regions. Not only will the higher-level directory nodes have to handle a disproportionately large number of request, but they also have enormous storage demands. For instance, the root node stores a contact record for each object in the network. To alleviate this problem the authors propose partitioning directory nodes into independent sub-nodes, and make each such sub-node responsible for a subset of the records originally stored at the node.

Our target environment is different than Globe's. We do not expect TerraDir directory objects to change their location frequently, and even if they do such situations are gracefully handled by the name-based location procedure.

2.3.2 Grid and GLS

Grid [LCKM00] combines geographical routing in physical space with the Grid Location Service (GLS) in identifier namespace, to locate mobile hosts in an ad-hoc network. GLS defines a hierarchically partitioned space, and forwards requests along paths that bring them closer to the destination in the identifier space using only information about nodes with nearby identifiers at each step along the path. To achieve this, each node maintains its location in a number of location servers distributed throughout the network, such that the location servers are relatively dense near the node but sparse farther from the node.

GLS uses *consistent hashing* [KLL$^+$97] to define its identifier namespace and its procedure for choosing the location servers of a node resembles a multi-dimensional variant of Chord. Grid defines a service for real-world tracking of physical objects and therefore its applicability is less relevant to our immediate domain of interest.

Among other approaches relevant to our work we identify the Landmark hierarchical routing [Tsu88], and resource discovery and naming systems such as the Intentional Naming System (INS) [WSBL99], the Global Naming System (GNS) [Lam86], and Discover [SDWG95].

2.4 Distributed Hash Table Overlay Networks

The following approaches share many characteristics and are based on the same idea of using a *distributed hash table* (DHT) for object location. This seemingly simple function can be used as the basic building block of more complex systems that perform a variety of sophisticated functions (file systems, event notification systems, etc.). Assignment of objects to hosts is performed by randomly mapping the object space into a virtual namespace, which is convenient because of the uniform spread of objects across sites. Load balancing is thus automatically achieved for uniformly distributed requests. Additionally, depending on the actual type of object mapping, the path length to locate an object, as well as the per-server state necessary to accommodate this, can be analytically bounded for stabilized scenarios. We expand on each of these approaches individually next.

2.4.1 CAN

CAN [RFH$^+$01] defines a virtual d-dimensional Cartesian coordinate space on a d-torus. Assuming an evenly partitioned space among N servers, the path length between any two servers is $O(dN^{1/d})$ with each server maintaining information about $O(d)$ neighbors. CAN allows for different redundancy schemes: multiple coordinate spaces can be in effect simultaneously, zones can be overloaded by assigning each of them a set of peer servers, while replication can be achieved by using multiple hash functions on the same data item. Additionally, the overlay network is topologically sensitive to the underlying IP network in the sense that a CAN node will likely have its neighboring zones assigned to nodes in its proximity as given by the IP-metric.

2.4.2 Chord

Chord [SMK$^+$01] uses consistent hashing [KLL$^+$97] and modulo operations to arrange both the keys and the servers on an identifier circle. An object's key is assigned to the first server whose identifier is equal to or follows the key in the identifier space. A Chord server requires information about $O(\log N)$ other servers to achieve efficient routing in $O(\log N)$ steps. However, good key placement balance can only be achieved by mapping multiple virtual servers to each real server, thus increasing the per-server space requirements to $O(\log^2 N)$. Chord offers weaker guarantees about worst-case performance: with *high probability* each server joining or leaving the network involves $O(\log^2 N)$ messages to maintain up-to-date routing information. In the light of extreme network restructuring routing performance could decay to linear paths while the system restabilizes.

CFS [DKK$^+$01] is a peer-to-peer read-only file system that uses the Chord lookup service to locate files dispersed throughout the network. Data placement granularity is very fine and consists of file blocks. Replication and caching is achieved on a file block basis which is convenient since (i) select portions of a large file may be more popular than others, and (ii) parts of popular files are distributed across different servers. However, it also increases the number of servers contacted and messages required to fetch a whole file, or large portions of it.

2.4.3 Pastry

Pastry [RD01a] is the routing and object location layer used by PAST [DR01, RD01b], a persistent storage utility, and SCRIBE [RKCD01], a scalable publish/subscribe system. Given b, a configurable parameter denoting the numerical base of the virtual space, Pastry maintains routing tables of size approximately $O(2^b \log_{2^b} N)$ and performs routing in $O(\log_{2^b} N)$ steps assuming accurate routing tables and no recent server failures. Replication takes place by storing an object on the k Pastry servers whose identifiers are closest to the object key in the namespace. Pastry's locality properties make it likely that among the k replicas of an object, the one that is closest to the requesting client (as given by IP metrics) is reached first. However in order to maintain such locality in the routing tables, a cost of $O(\log_{2^b} N)$ is involved whenever a new server joins the network.

2.4.4 Tapestry

Tapestry [ZKJ01] is very similar in spirit to Pastry, both being inspired by the routing scheme introduced by Plaxton et al [PRR97]. Tapestry was designed as a routing layer for Oceanstore [KBC$^+$00], a utility infrastructure for persistent and nomadic data. Perhaps more interesting is the fact that the authors initially intended to use a variant of Bloom filters [Blo70], the *attenuated* Bloom filter, that would capture content from other servers beside the local server, with corresponding decreasing accuracy. By comparison, TerraDir digests are lightweight and since they incorporate only local data, they are not exposed to such a high degree to inconsistencies between digest content and actual host content. Among other proposals using the distributed hash table approach we note Kademlia [MM02] and Viceroy [MNR02].

The routing tables used by Chord, Pastry and Tapestry implicitly capture a tree with the nodes of interest located at the leaves. Each server has a different tree-like view of the overlay network, and hence load balancing is automatically achieved for uniformly distributed queries. By contrast, TerraDir explicitly defines a single namespace tree and this view is globally imposed on all directory participants. Replication addresses the inherent problems of hierarchical structures by defining additional globally-shared tree views.

In virtualized namespace schemes, owners of data have no control over data placement or authoritative pointer to data placement. Moreover, when new participants join or leave the network, part of the data (or pointers to data) changes locations automatically. This arrangement may be unacceptable to some client applications or users, for security and management reasons. In TerraDir, object location is achieved through hierarchical naming. This means that (i) owners of data have full control over object placement, (ii) object location is independent of membership mutations in the network, and (iii) object location can change without requiring name changes and vice-versa. Opportunities to exploit temporal and spatial locality are also lost through namespace virtualization [KBS02], an argument which we expand upon in the next chapter.

2.5 Query Locality and Zipf Distributions

Studies [ABCdO96, BCF$^+$99] show that both spatial and temporal reference locality are present in requests submitted to web servers or proxies, and that such requests follow a Zipf-like distribution [Zip49]. Distributed caching protocols [KLL$^+$97] have been motivated by the need to balance the load and relieve hot-spots on the World-Wide-Web. These approaches target client-server communication models which are different than ours.

Similar Zipf-like patterns were found in traces collected from Gnutella, one of the most widely deployed peer-to-peer systems. Caching the results of popular Gnutella queries for a short period of time proves to be effective in this case [Sri01]. Our path propagation is a generalization of this caching scheme.

A recent study [GDS$^+$03] shows that the popularity of objects in Kazaa, the most widely used file-sharing utility presently, is often short-lived. There is significant turnover popularity within Kazaa, and the most popular objects tend to be recently born objects. This stresses the need for replication models that are adaptable to user demand patterns, and is partly the reason for using shifting hot-spots benchmarks in this dissertation.

Freenet replicates objects both on insertion and retrieval on the path from the initiator to the target mainly for anonymity and availability purposes. It is not clear how a system like Freenet would react to query locality and hot-spots.

Adaptive replication algorithms for database applications change the replication scheme of an object to reflect the read-write patterns and are shown to eventually converge toward the optimal scheme [WJH97]. Concurrency control mechanisms need to be specified for data replication to guarantee replica consistency.

None of the DHT-based overlay networks define adaptive replication mechanisms similar to TerraDir's. Instead, caches are used to spread popular objects in the network, and lookups are considered resolved whenever cache hits occur along the path. CFS for instance uses k-replication similar to Pastry for data availability, and populates all the caches on the query path with the destination data after the lookup completes. Backslash [SMB02] is a collaborative web mirroring system using caches to address unanticipated, short-term and rapid increase in the popularity of a resource (flash crowds). Dissemination trees [CKK02] build a dynamic content distribution network satisfying certain QoS and efficiency/capacity requirements. Both of them use DHT-based overlay networks for location services and feature adaptive elements.

2.6 Data Replication and Consistency

A great deal of work addresses data replication in the context of distributed systems [HHB96], and in particular in the context of distributed database systems [GHOS96]. Application correctness in these environments mandates that all updates be propagated and applied to all replicas. Many of the applications of interest also rely on some notion of *causality* as far as data semantics is concerned. To help formalize this concept the notion of *transaction* has been introduced. A transaction is a grouping of a sequence of related operations to form a unit of execution such that the following properties are maintained: atomicity (either all or none of the operations of the transaction are executed), consistency (a transaction that finds the database in a consistent state, leaves it in a consistent state after its execution), isolation (transaction are executed separately from one another), and durability (when a transaction commits, its effects are permanent).

In a serial execution, a group of transactions are executed in serial order such that their operations do not interleave at any one time. An execution of a group of transactions is *serializable* if it is equivalent to some serial execution. Thus, transactions committed through a serializable execution appear to have been executed in sequence. Serializability is a strong consistency condition and supported by the concurrency protocol of all popular commercial databases. Two-phase locking (2PL) is one of the concurrency protocols that can ensure serializability. The notion of serializability can be extended to replicated databases in conjunction with replica control protocols. A *replica control protocol* specifies which copies of a data item are accessed whenever an operation is to be performed involving the item. The condition when all replicas of an item appear as one copy is referred to as *one-copy equivalence* [BG87]. Coupled with serializable executions, the correctness criteria for replicated data in transactional environments is *one-copy serializability* (1SR) [BG84]. Among the replica control protocols that can achieve one-copy serializability, we note the read-one write-all (ROWA) protocols, the voting protocols, and the quorum consensus protocols.

Quorum consensus protocols have received a lot of interest both in the database and the distributed systems communities. This is mainly due to their reasonably good fault tolerance properties while still being able to offer support for one-copy serializability. In quorum consensus protocols an operation is allowed to proceed to completion if it satisfies the consensus of a group of sites (a quorum). Quorum groups are defined such that conflicting operations (e.g. read-write, write-write) have their groups intersect in a pairwise fashion, a property known as the *quorum intersection property*. In the next section we discuss the read-one write-all, the weighted voting, the majority quorums, and the multi-dimensional voting protocols.

2.7 ROWA and Voting Protocols Using Quorum Consensus

The Read-One Write-All [AAC94] is the simplest family of protocols for managing replicated data. Read operations are allowed to access any one copy, while write operations are required to update all copies. The *primary-copy* [Sto79] protocol is a popular variant of ROWA where one of the replicas is designated as the master copy. In the primary copy protocol reading is performed at any copy. Writing is performed at the primary copy, with writes being subsequently propagated to all other sites. ROWA protocols suffer from imbalanced access availability. Reads are executed at a very low cost and are highly available. Write operations on the other hand are inefficient and have very low availability even compared to the case when data is not replicated: all copies need to be operational for a write to proceed. The *available copies* protocol (Read-One Write-All-Available) [GSC$^+$83] overcomes some of the limitations of ROWA by requiring that only operational copies be updated. If network partitioning occurs —sites in a partition can communicate with each other, but not with sites in other partitions— the available copies protocol may cause inconsistencies in the database.

2.7.1 Weighted Voting

Gifford defines quorum sets in terms of weighted voting [Gif79] for synchronizing concurrent accesses to shared files. If the total number of votes is v, v_r votes are needed to read a file, and v_w votes are needed to write a file, such that (i) $v_r + v_w > v$ and (ii) $2v_w > v$. A site can be assigned more than one vote, and the distribution of votes to sites need not be uniform. Version vectors are used to identify the latest update, and locking is used to guarantee serial consistency.

Weighted voting does not assume a logical structure of copies when realizing quorums. Further, for symmetrical quorum configurations ($v_r \approx v_w$) the approach can tolerate failures of up to half of the participating sites. In this case however the communication costs are high as each transaction involves accessing half of the copies. Weighted voting can model scenarios where read operations dominate accesses to data ($v_r \ll v_w$). In such cases, weighted voting approximates the ROWA approach both with respect to its poor write availability, as well as its high costs for write operations. In fact, the read-one write-all protocol is a special case of voting assignment for which $v_r = 1$, and $v_w = v$.

Various proposals exist for distributing the votes across participating sites such that efficiency, availability or a combined metric is optimized. In general, replicas that are more reliable should receive more weight [BGM84]. For instance, optimal weight distributions can be derived from individual failure probabilities at sites [AW98]. Dynamic weighted-voting schemes [BGMS89] adaptively reassign votes to sites such that more reliable sites receive more voting weight. If sites have the same availability, the optimal vote assignment for the weighted voting protocols is one where all sites are assigned one vote each [TK88].

2.7.2 Coteries and Majority Quorums

The notion of quorum sets is closely related to *coteries* as defined by Garcia-Molina and Barbara [GMB85]. In coteries, an added minimality condition is imposed upon quorums groups, and no distinction is made between different types of quorum group conflicts. For efficiency reasons, quorum sets exploit this distinction between read and write operations. The authors also show that quorum based schemes are more expressive than voting. More exactly, there are quorum sets that cannot be modeled through voting assignments such that mutual exclusion is still guaranteed.

Thomas defines majority quorums as quorum sets for which each quorum group contains a majority of copies [Tho79]. Again, this is a special case of weighted voting for which $v_r = v_w = \lfloor v/2 \rfloor + 1$. As note above, this assignment provides the best symmetric availability for read and write operations. The replica control protocol proposed by Thomas uses timestamps [Lam78] instead of version vectors to synchronize accesses to a replicated database. When timestamps are used, intersection of write quorum groups is not necessary [Tho78], while read quorums still need to intersect write quorums. Different non-intersecting writes are registered with unique timestamps generated from a totally ordered domain, and a read operation is able to identify the latest update as long as it intersects with all write quorums. A similar technique (logs with timestamped entries and non-intersecting writes) has been used by Herlihy [Her86] in defining a general consensus method for abstract data types that extends the read/write semantics of operations on data.

2.7.3 Multi-Dimensional Voting

Logically structured quorum sets cannot be modeled with voting assignments in the general case [GMB85, AAC91], making the quorum consensus approach more appealing than weighted voting [Gif79] in a number of instances. Multidimensional (MD-) voting [AAC91] addresses this problem by generalizing weighted voting to voting in multiple dimensions. The d-space quorum consensus method is conceptually different than MD-voting. The latter defines an abstract voting space, and requires a quorum of votes to be gathered in every dimension of interest. However, MD-voting can emulate d-space quorums in a rather inefficient manner. For an m by n 2-space quorum set, the MD-voting mapping is achieved by defining voting assignments in an n-dimensional space, and requiring that a quorum of votes be gathered in all n dimensions.

2.8 Logically Structured Quorums

Quorum sets defined on logical structures use structural information to define intersecting quorum groups. We present the Grid protocol, the tree protocol, and the hierarchical quorum consensus protocol.

2.8.1 Mutual Exclusion on a Grid

The problems of enforcing one-copy equivalence through quorum consensus or voting schemes, and distributed mutual exclusion are closely related. Given a solution to the distributed mutual exclusion problem, replica quorums can easily be established. In fact replica control protocols generally use locking in conjunction with quorums to guarantee one-copy equivalence. Replica control protocols defining quorums are based on the exclusion principle: while an activity changes state in the system no other conflicting activity is allowed to perform.

Maekawa proposed using finite projective planes to obtain a distributed mutual exclusion protocol [Mae85] where the number of sites contacted per transaction (a quorum group) is on the order of $O(\sqrt{N})$. As an alternative protocol having the same performance characteristics, Maekawa further suggested organizing the sites in a two dimensional grid. Assuming a $\sqrt{N} \times \sqrt{N}$ logical grid, a site requesting the lock would contact all the sites found in some arbitrary line and column of the grid. Every two pairs of lines and columns intersect each other, and correct arbitration is ensured for lock grant/release primitives. Starting with Maekawa's finite projective planes approach and given his conditions, it can easily be shown that the solution is optimal. More exactly, quorum sets such that (i) each quorum group has (roughly) the same number of elements, and (ii) each element appears in (roughly) the same number of groups, will consist of quorum groups having $O(\sqrt{N})$ elements.

2.8.2 Grid Quorums

Maekawa's mutual exclusion protocol has inspired Cheung et al in developing the Grid replica control protocol [CAA92]. Quorum groups for read operations consist of one line, and quorum groups for write operations consist of one line and one column. The authors observe that instead of a line, for both read and write quorums, a more relaxed configuration that requires one node in each column (a *column cover*) can be employed. Two-phase locking is used for concurrency control and timestamp ordering is mentioned as an alternative scheme. The Grid protocol has low communication costs ($O(\sqrt{N})$), and is best suited for scenarios where the frequency of read and write operations are on the same order.

2.8.3 Quorum Consensus on the Tree

The tree protocol proposed by Agrawal and El Abbadi organizes the set of copies in a binary tree with $\log N$ levels [AA89]. A quorum group is formed by including all the sites in some arbitrary path from the root to a leaf. If a site along a path is unavailable, the quorum group is formed by substituting the failed site with the two child nodes of that site, and continuing recursively along both paths to the leaf level of the tree. The tree protocol features the lowest message complexity ($O(\log N)$) among all structured quorum schemes, assuming no site failures. In the presence of failures, the algorithm degrades gracefully as progressively more sites are involved in a quorum group, for a maximum of $N/2$ (when all sites on all levels but the leaf level have failed).

Despite its low communication costs and good tolerance to site failures, the tree protocol is less appealing when considering the distribution of accesses over the set of copies. The root site is part of all quorum groups (assuming no failures), while a leaf site is part of $N/2$ times fewer quorum groups. The tree protocol is not truly distributed and employs a weak form of decentralization to ensure exclusion of accesses.

2.8.4 Hierarchical Quorum Consensus

Kumar extends weighted voting to voting on multiple levels of a hierarchy comprising the set of all replicas [Kum91]. In contrast to the tree protocol, physical copies of objects are stored only at the leaves of the tree, while all other levels serve a logical grouping purpose. In effect, the protocol performs a hierarchical partitioning of the replica set. Given a perfectly balanced tree with $m + 1$ levels (with the root on level 0 and replicas on level m) such that a node on level i has l_{i+1} children, the overall number of replicas is $\prod_{i=1}^{m} l_i$. A node assembled on level i must in turn recursively assemble r_{i+1} (w_{i+1}) of its l_{i+1} children nodes on level $i+1$ for a read (write) quorum group. The root node is part of all read (write) quorum groups. The quorum intersection condition is satisfied if (i) $r_i + w_i > l_i$, and (ii) $2w_i > l_i$ for all levels $i = \overline{1, m}$.

A read quorum group defined by the hierarchical quorum consensus scheme consists of $\prod_{i=1}^{m} r_i$ copies, and a write quorum group of $\prod_{i=1}^{m} w_i$ copies. Weighted voting can be regarded as the special case of a two level hierarchy ($m = 1$). Optimal quorum group sizes are obtained for the hierarchical consensus method when each group contains three subgroups, i.e. $l_i = 3$. In this case symmetrical quorum groups consist of $N^{0.63}$ sites, which is closer to the optimal complexity $O(\sqrt{N})$ than to the default weighted voting performance ($\lfloor N/2 \rfloor + 1$). Further, the hierarchical quorum consensus method allows for imbalanced quorum groups for read and write operations to be specified. For these reasons we have chosen it to contrast the performance and availability of the approach that we advocate in this dissertation.

2.9 Quorum Reconfiguration Protocols

The quorum consensus protocols emphasize fault tolerance. They can gracefully handle independent and isolated failures. If a quorum group for a read or write operation is not available due to site failures, another quorum group can be chosen to perform the operation. The protocols seem nonetheless less appealing when considering positively correlated failures or network partitions [Woo98]. If a read quorum fails, no write quorum group can be assembled since any write group has at least one site in common with the failed group. Similarly, if a write quorum fails, no read or write quorums can be assembled. If a partitioning of the network occurs, then updates *could* take place in at most one of the resulting partitions. It could very well happen that there is no such partition.

Assuming that failed copies never recover, one may be tempted to allow operations to make progress as long as all available copies (as opposed to all copies) in a quorum group are properly locked. However, one-copy serializability follows from ordinary serializability only if two conflicting transactions that have conflicting accesses to a data item, have conflicting accesses to some copy of the data item [BHG87]. If copies that could expose the conflict have failed, and are not included in the quorum in which they would otherwise be included, illegal executions (with respect to one-copy serializability) can occur.

Logically structured quorum protocols rely on pre-defined identity for participating sites (given by the assignment of sites to quorum groups), and thus seem less tolerant to failures than voting protocols. Voting protocols are more adaptable through reconfiguration because they define quorums based on the number of votes, regardless of the identity of the votes. However, in general structured quorum protocols access considerable fewer sites than voting-based quorums and have higher overall operational availability. Quorum reconfiguration protocols have been proposed to prevent conditions when no quorum groups can be assembled. Quorum groups are dynamically and transparently adjusted to reflect the evolution of failures and repairs in the system [AA96].

In *dynamic voting* [JM90] quorum readjustment is performed within the protocol for write operations. Other approaches such as the *epoch protocol* [RL92, RL93] check for changes in system topology or even reconfigure it as a separate transaction, asynchronously with regard to read and write operations. The *virtual partition* algorithm [AT89] operates by maintaining at each site a view of available sites. Within a view a quorum protocol is in effect, and each view must include at least a read and a write quorum of the whole set of sites. Sites change their views using a two-phase commit protocol, and all active transactions at the sites that accept the new view must be aborted. Dynamically adjusting quorums [Her87] is similar in spirit to the virtual partition algorithm. On failures the system may not be able to find available write quorums, and the protocol can switch to other sets of read and write quorum sets that feature smaller write quorums groups and larger read quorums groups. The technique is referred to as *quorum inflation* and a complementary one (*quorum deflation*) exists for reducing the size of write quorums when sites recover.

In the following section we briefly describe the epoch protocol.

2.9.1 Reconfiguring in Epochs

The epoch protocol [RL92, RL93] infers a structure for the network, based on a rule and a total ordering of participating sites. This means that given any set of sites, a logical structure can be constructed without recurring to a static mapping based on site identities. An *epoch* is a set of operational sites such that each site in the epoch knows about the availability of others. The current epoch describes the current state of the system. Initially, all replicas of the data item form the current epoch. Epoch checking is run periodically to poll all replicas in the system. If members of the current epoch are not accessible, or any replicas outside the current epoch have been successfully contacted, an attempt is made to form a new epoch.

The new epoch must contain a write quorum of the previous (current) epoch, and the set of new epoch members along with the new epoch number is recorded on every member of the new epoch. If network partitions occur, the intersection property of quorums guarantees that the attempt to form a new epoch will be successful in at most one partition. Further, a read or write operation must contact at least one member of the current epoch and therefore obtain the current epoch set. The transition from one epoch to another (quorum reconfiguration) need not be performed with every change in the topology, i.e. with every site failing. This gives failing sites the opportunity to recover and reduces the number of adjustments performed over time.

Common to all approaches above is that they use (partial) global knowledge in reconfiguring quorum groups. Further, the implicit assumption for many quorum reconfiguration schemes is that failures (whether isolated or correlated) are transient, or otherwise that the system will experience few membership changes, if any. Global knowledge, even in limited forms is not a feasible option for P2P networks, and membership changes (users/sites leaving or joining the network) occur on a scale non seen before in distributed systems. In this dissertation we define a quorum reconfiguration mechanism that uses local knowledge and can withstand high rates of membership changes. We illustrate and analyze local reconfiguration using a replica control protocol based on d-spaces.

Chapter 3

Hierarchical Routing with Soft-State Replicas

A new generation of sophisticated, multi-party peer-to-peer applications are being proposed and implemented over the Internet. Examples of such applications include Internet-based encyclopedias [nup, wik], distributed auctions conducted over the network without involving a central authority; ubiquitous file systems that allow distributed access to data regardless of the user's point of attachment to the network; distributed resource-sharing applications which allow processing and other resources to be shared or coordinated over a wide-area network; and distributed publish-subscribe systems where resources are advertised and accessed over the network.

A key requirement for all these examples is the need to compose coordinated views of advertised resources, and to be able to perform resource lookups and searches in a scalable and fault-tolerant manner. To address these emergent needs we have developed TerraDir [BKS01], a set of distributed protocols that can be used to build a broad range of wide-area resource discovery applications, including all of the applications described above. TerraDir defines a decentralized hierarchical directory in which resources are advertised and accessed using cooperative nodes distributed throughout the network.

Much of the initial work in wide-area P2P systems has been with file sharing applications like Napster, Gnutella, and Freenet [nap, gnu, CSWH00]. They helped crystallize many of the issues central to peer-to-peer systems, most notably decentralized control, self-organization, adaptation and fault tolerance. Limitations such as centralized content indexing, search protocols based on network flooding, and unreliable content location render these systems less attractive for many other types of applications.

More recently, a number of interesting ideas addressed the problem of query routing and object location in distributed systems. The most distinguishing feature of all these proposals [RFH$^+$01, SMK$^+$01, RD01a, ZKJ01] is a self-organizing *virtual overlay network*, whereby messages can be routed in a provable bounded number of steps. Additional properties such as network scalability and recovery protocols triggered in the light of server failures give an extra flavor to an already appealing infrastructure. However, these features come at the cost of decoupling object location from naming through a process of uniform virtualization of the object namespace.

One of the contentions of this dissertation is that for applications that naturally describe their data in a hierarchical manner, a hierarchical directory metaphor with naming semantics and built-in search primitives with attribute qualifiers is more appropriate. We support this contention by showing that such a design can exploit this integration to the benefit of client applications, while keeping in pace with the same performance and service quality indices as virtualized overlay network schemes. More exactly we argue that:

- Problems traditionally associated with hierarchical structures can be overcome by employing a combination of lightweight mechanisms and protocols

- Temporal and spatial locality are best handled by integrating the naming and location services, as opposed to decoupling them or virtualizing the namespace

- For applications that need support beyond simple lookup operations (e.g. browsing and searching) it is more desirable from an efficiency standpoint to implement such support at the directory service level than at the application level.

A key component of TerraDir's protocols is a replication model that adaptively replicates routing information in response to fluctuating (routing) load in the network. Processing bottlenecks that are specific to hierarchical namespaces are eliminated with the resulting system behaving like a true symmetrical network. Further, temporal locality in the stream of requests is managed by replicating on demand popular portions of the namespace. Finally, we argue that having the object namespace define the peer-to-peer connectivity is central to the applicability of our model. In particular, overlay networks that virtualize the object namespace cannot benefit from a similar mechanism of replicating routing information.

The rest of this chapter is organized as follows. Section 3.1 describes the TerraDir framework for building lightweight hierarchical directories. In Section 3.2 we expand on the routing and replication protocols. Section 3.3 performs a comprehensive evaluation of the TerraDir routing and replication protocols in a simulated environment. We summarize our main results and conclude the chapter in Section 3.4.

3.1 TerraDir: A Framework for Lightweight Hierarchical Directories

TerraDir is specifically designed for data that can be arranged into a rooted hierarchy, the *directory*. Two other concepts central to its architecture are the *nodes* and the *servers*. A node is a directory entry owned by the server that exports it. Nodes are in ascendant-descendant relationships with other nodes, based on their position in the hierarchy. A server is an autonomous process that participates in a *single* directory by exporting nodes and/or accessing other nodes. Multiple nodes can be exported by the same server. More than one server (peers in the same or different directories) can reside at the same network site (physical host). While bearing in mind the distinction between servers and sites (hosts), we may still occasionally use them in an interchangeable fashion. Note that due to the nature of data and potential client applications, we adopt a slightly different terminology than most of the related work in P2P systems. Thus, a node in our context is associated with a data item present in the network and not with a server.

3.1.1 Directory Namespaces

The primary abstraction exported by participants in TerraDir directories is the *hierarchical namespace*. Client applications use the namespace to store information about a potentially very large number of objects.

Node names are constructed much like DNS or Unix file-system names. Each node, except the root, has a non-null label. Duplicated labels for different nodes are not prohibited. Each non-root node in the namespace must have at least one directed edge incident upon itself. Exactly one of these edges is known as the *canonical edge*. Thus, the set of nodes and canonical edges form a directed tree rooted at the root node. The *canonical name* of node is formed by the sequence of labels encountered in the unique path from the root to the node, using only canonical edges. The *canonical parent* of a node is the node such that there is a directed canonical edge from the parent to the *child*.

We allow the creation of additional edges (*symlinks*) between nodes in the directory, such that the overall structure of the namespace is that of a rooted graph without cycles. Symlinks are a convenient way of supporting the common situation where an object might be filed under multiple names. A node can be accessed using any of its names, including its canonical name or names derived through a symlink. In Figure 3.1 we show a TerraDir namespace. Each node has its label annotated to it, and the node owners are identified by numbers. Canonical edges are shown using solid arrows while symlinks are shown using dashed arrows. */animals/dogs/puppies/Bert* is the canonical name for the node whose label is *Bert*, while */Duckie/Friends/Bert* is an auxiliary name referring to the same node. For simplicity and unless otherwise mentioned, we assume throughout this dissertation that all namespaces of interest are trees.

Namespace Credentials

To accommodate and control various operations on namespaces we introduce *namespace credentials*. Such a credential consists of a namespace identifier, a set of attributes, and a set of cryptographic secrets. At the very least, the attributes describe what level of access (read-only, read-write, etc.) the holder of the credential has over the nodes in the associated namespace. Other attributes may describe the caching characteristics, consistency semantics, and version information (temporal constraints) for the directory.

Credentials provide a convenient means of securely exporting namespaces and named views. The following operations are available for use with namespace credentials: copying (credentials are merely byte-streams, and can easily be copied and distributed), grafting (namespaces can be combined to form derived namespace), splitting (the owner of a directory might not always want to export all nodes with the same permissions; such operations are possible by splitting the rights contained in a credential), and constraining (a namespace can be constrained in several ways; for instance,

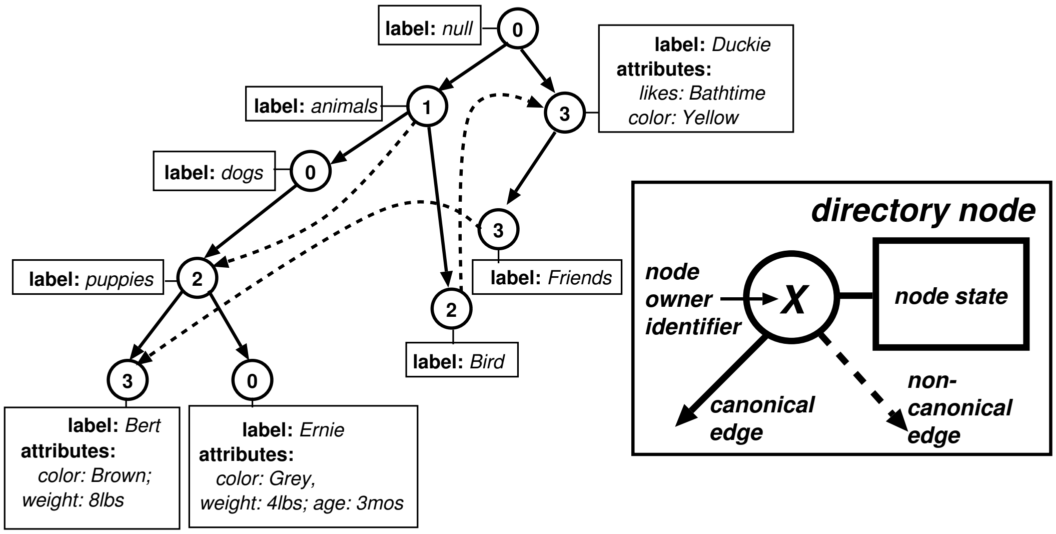

Figure 3.1: A TerraDir namespace.

the extent of the namespace can be constrained by changing the root to point a child node).

Derived Namespaces: Composing Aggregate Views

A self-contained namespace in which all links are to nodes within that namespace is called a *primitive namespace*. Often it is useful to be able to support composite directories that contain nodes from multiple primitive namespaces. TerraDir supports such abstractions using *derived namespaces*. A derived namespace is a namespace that has links to nodes of other primitive or derived namespaces. Note that some of the nodes in a derived namespace are reachable using at least two names, one in its primitive namespace and one using the derived namespace. Derived namespaces allow the creation of "virtual distributed directories" by combining nodes from arbitrary other namespaces.

Let $\mathcal{D}$ be a derived namespace which contains nodes from some other namespace $\mathcal{N}$. In order nodes in $\mathcal{N}$ to be included in $\mathcal{D}$, there must be an edge from a node in

$\mathcal{D}$ to some node in $\mathcal{N}$, n. Node n has no state for incoming derived links. Grafting links may be added or deleted without interacting with the owner of n. Once created, derived namespaces can be accessed and searched much like primitive namespaces, and credentials for derived namespaces can be distributed analogously. The creation and modification of derived namespaces is a particularly lightweight operation, since nodes upon which derived edges are incident are not involved in any derived namespace-specific operation.

3.1.2 Data Model and Query Semantics

We describe the assume data model and the semantics of query processing.

Nodes and Servers

The directory entry (node) is the atomic entity for exporting data in a namespace. Every node has a label and its position in the namespace fully qualifies its canonical and auxiliary names. In addition, the node state includes a set of incoming edges from its immediate ancestors, and a set of outgoing edges to its immediate descendants and bookkeeping information. Nodes export two types of application supplied information: *node data* and *node meta-data*. Both are optional. Node data is the actual contents of a node, while meta-data are node annotations and are most commonly found in the form of attributes (name-value pairs). For a file-system implemented on top of TerraDir or a file sharing utility, there is a 1-to-1 correspondence between files and nodes. The node's data in this case is the file, and the meta-data are attributes and searchable keywords annotated to the file. In Figure 3.1 we show node meta-data as attributes for some of the nodes in the namespace.

Every node is owned by exactly one server known as the *owner*. The owner of a node keeps the node's data and meta-data as well as additional state needed by the TerraDir protocol in making routing decisions. Note that the owner of a node is also the server that exports its data. Hash-based peer-to-peer systems virtualize the object namespace and the server that exports the object data is most likely different than the server that keeps the object's hash key. For such systems the location of data and the location of pointers to data are different.

Lookup and Search Queries

The service offered by TerraDir is retrieval of data, based on node names and optional attribute qualifiers. Names of interest can be discovered either by browsing the namespace —knowing a node's name we can infer its parent's name or query for its children— or by performing search operations with regular expressions containing wildcards.

We are not indexing individual node data and do not specify mechanisms for distributing and composing indexes across a namespace. Incorporating such functionality would restrict the applicability of our service. More exactly, in order to support indexes the data exported by directory nodes would have to be text data or otherwise structured data. Since we do not want to make assumptions about application data, we are currently not concerned with *indexed searches* [Fal85] that perform searches on data content.

TerraDir differentiates between lookup and search queries. A *lookup query* is a request for meta-data of a fully-qualified object which can be specified using any of its names (e.g. */university/public/people* for the namespace in Figure 3.2). A *search query* is a request for a set of nodes that match specific criteria. Search queries that span arbitrary portions of a namespace and resolve to multiple nodes can be specified. Search criteria consist of two parts: the *scope*, defining the scope of the namespace to be searched, and the *filter* determining which candidate nodes in the scope are part of the result. The scope is a partially-qualified path where each label in the path can be specified as a regular expression (e.g. */university/.+/people/student/(MS|PhD/candidate)/.+* for the namespace in Figure 3.2). Filter matching is performed on node meta-data using regular expressions and boolean operators (e.g. *(age>25)&(name=smith|jo.*)&car*).

Our choice in defining the query framework is driven by the following contradictory goals:

- *Expressiveness vs. side-effect control:* The structure of the search criteria is both expressive and limiting. It is expressive in that searches can be made across multiple levels of the namespace, and allows a great deal of freedom in using arbitrary attributes to narrow a search. It is limiting in that regular expressions do not extend over multiple levels. Thus, there is no way to directly write a general search that returns all nodes in the subtree rooted at some node. This limitation is crucial to the scalability of the system, as the number of objects whose meta-data can be stored and accessed via a single namespace may reach arbitrarily large values.

- *Redundancy vs. integration:* A lookup query is essentially a search query that specifies a fully-qualified path and an empty set of attribute qualifiers. We explicitly define lookup query support mainly because object location and retrieval functionality is considered to be the basic primitive for many P2P systems. In corollary to this argument, search functionality is not a requirement at this level, and probably not a recommendation either if we are to heed end-to-end design principles [SRC84]. However, we argue in Section 3.1.7 that implementing search queries as sequences of lookups would adversely affect search performance.

A query returns the node's name, its meta-data, and mapping information for the requested node. The *node mapping* is a set of servers that host the node's data. Given the result of a query, the client applications can further request the node's data from one of the servers in the map. Note that getting node data is a two-step process: a node lookup, followed by the actual data retrieval. Complex search queries are decomposed hierarchically into individual lookup queries, the requested nodes are resolved, and then the results are aggregated and sent back to the requester. Subsequently, the query initiator may ask for the data of some of the nodes in the search result.

All P2P systems that have been deployed so far make a similar distinction between searching for data, and retrieving some of the data thus identified. The reasons are twofold: (i) enormous savings in bandwidth and resources, and (ii) a choice (connection bandwidth, freshness of data, etc.) for retrieving the data when presented with multiple peers that host it. This dissertation focuses on the searching process and how lookup queries are routed in the network using a lightweight replication mechanism.

3.1.3 Query Processing

We describe the routing procedure performed upon lookup and search queries. The effects of caching and replication are ignored for purposes of this discussion.

Lookup Queries

A lookup query is a request for the attributes and mapping information of a node. Lookup queries consist of the name of the desired node in the directory, and can be submitted by a participating server or an arbitrary site (query initiator). Assume server

S generates query n_q, and that Q is the owner of node n_q. Further, assume that the longest common prefix between n_q and *any* node owned by S is n_p. If S and Q are the same server then the lookup is a local operation and is resolved immediately. Otherwise, a number of remote operations are involved: the query is passed "upwards" the directory hierarchy to the owner of n_p, and then "downwards" to Q. Finally, a reply, the requested node information or $\perp$ if n_q does not exist, is sent back from Q to S.

Such a hierarchical traversal implies network communication each time the query is passed between two nodes that are not owned by the same server and will incur costs on the order of the directory tree's height. Incremental progress toward the destination is made at each step and the routing finishes whether the queried node exists or not (Assuming node n_d does not exist, there must be a largest integer i, such that d_i belongs to the directory and d_{i+1} does not. Thus, the owner of d_i can reject the query since it is the authoritative source for all its children.).

A variant of this routing procedure is implemented by DNS. In DNS a cache miss incurs visiting one of the root servers and then a walk down the namespace. Note that we are assuming recursive query forwarding while some other systems (DNS, Chord, etc.) generally use iterative query forwarding models. There are no outstanding arguments for such a choice, and everything presented in this dissertation can be applied to either approach.

Search Queries

Searches are queries that contain regular expressions in the node name specification and optionally filtering constructions to further discriminate based on attributes. A search expression with one regular expression R has the form $/d_0/\ldots/d_i/R/d_{i+1}/\ldots/d_k$. The procedure for routing through the first portion of the path, $/d_0/\ldots/d_i$, is similar to the one outlined for lookups.

Assume the owner of node $/d_0/\ldots/d_i$ is S and let its children be denoted by $/d_0/\ldots/d_i/c_j$, for some labels c_j. For each label c_j that matches R a new query having the form $/d_0/\ldots/d_i/c_j/d_i+1/\ldots/d_k$ is initiated at server S and forwarded to the owner of $/d_0/\ldots/d_i/c_j$. When these queries reach their destinations, regular expressions in filters are matched against attribute names and values to determine what nodes are part of the query result. On receiving replies for all the queries it initiated on behalf of the original query, S composes these results and sends the final result to the initiator, thus completing the query processing. The exact same procedure (*query decomposition* and *result recomposition*) is used recursively and separate sets of subqueries are initiated at other servers along the path if the suffix of the query contains additional regular expressions.

To facilitate routing, a server maintains for each owned node its *routing context* in the namespace. The routing context consists of neighboring nodes and guarantees routing with incremental progress. Thus, a server maintains mappings for the neighboring nodes of every node that it owns. The mapping is the association of a node's name and a set of servers hosting the node. For tree namespaces with N nodes on S participating servers we can only bound the number of neighboring maps (links) maintained per node by $O(N)$. However, the cumulative number of links for all nodes is exactly $2(N-1)$, which yields a comfortable mean of 2 links per node. Thus, on the average each server will keep at most twice as many neighboring maps as the number of nodes it owns.

The number of steps taken to route a query is bounded by the diameter of the namespace which for arbitrary topologies is $O(S)$, and $O(\log N)$ for all topologies of interest, i.e. trees. In practice, with limited amounts of caching ($O(\log S)$ cache

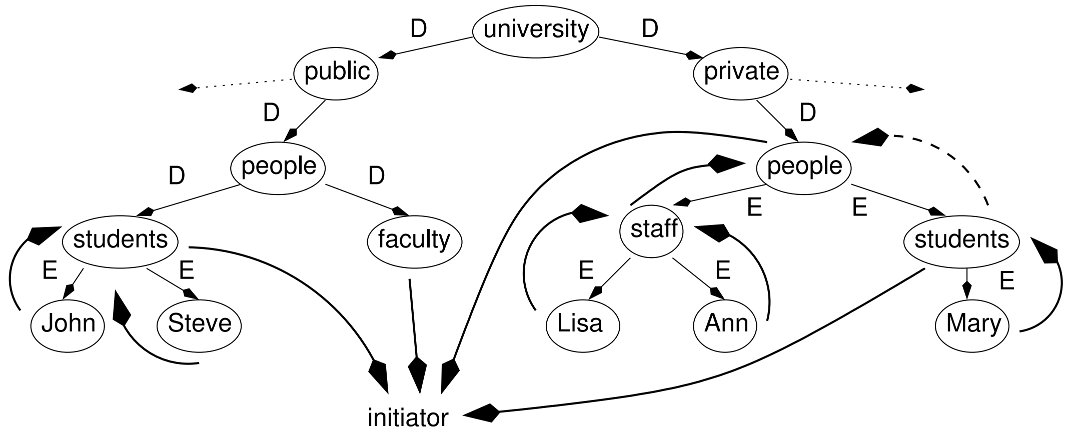

Figure 3.2: A recursive search for query */university/.+/people/.+/.+* as performed by the token protocol. We show only the result aggregation phase.

entries per server), the number of routing steps per query is $O(\log N)$ for all namespace topologies.

3.1.4 A Query Decomposition/Result Recomposition Algorithm Using Tokens

Hierarchical query decomposition is performed when more than one children of the current node matches the current label in the scope. For instance, upon reaching node */university*, query */university/.+/people/students?/(MS|PhD/candidate)/.+* is decomposed and forwarded to all children of */university*. Filter matching is performed locally at each node that is part of the search scope. Upon successful filter matching, partial results can be returned directly to the query initiator, or they can be recomposed hierarchically and returned as one final aggregate result.

Trading off Bandwidth and Connections

Assume there are k decomposition levels in the scope, and that at each point the query branches into b sub-queries. Thus, if we consider only the decomposition points of the namespace for the present query, we get a perfectly balanced tree of depth k where every interior node has b children. Further, assume that all b^k nodes match the scope and satisfy the filter, and that the byte length of an individual result is l.

The search results can be returned directly and individually from matching nodes to the query initiator. b^k messages are required to achieve this, for a total bandwidth of lb^k. There are a few problems with this approach. First, the query initiator does not know how many messages are in the result, and thus won't know how long to wait for query completion (unless of course we are making strong assumptions about the synchrony of the system). Second, opening b^k connections to the initiator within a small window of time may cause denial of service conditions at the initiator.

Alternatively, the results can be recomposed hierarchically and returned to the query initiator as an aggregate result. For the example above, node */university* would return the result after receiving replies from all relevant children. This approach requires $\frac{b^{k+1}-1}{b-1}$ messages, and the bandwidth is increased by a factor of $k+1$ to a total of $(k+1)lb^k$. Furthermore, $\frac{b^k-1}{b-1}$ of the nodes involved in the search procedure will process a query twice: once when decomposing it, and a second time when aggregating partial results. Finally, nodes high up in the hierarchy (which constitute the bottleneck for a hierarchical namespace), will end up transferring large amounts of data, as partial results are almost totally aggregated by the time they reach top levels.

Query Result Recomposition With Tokens

Neither of the two approaches presented above is satisfying. We define a token based algorithm that enables the initiator to determine when a query completes. The protocol allows for a configurable scenario between the two variants described to be implemented. We associate one of the two states *expecting* or *deferred* with each edge that is traversed by query q. An edge in the *expecting* state means that the ancestor is expecting to hear from its descendants regarding the query results; only *some* of the results have to be aggregated back to the ancestor, with the restriction that the ancestor still has to hear from all its descendants. An edge in the *deferred* state means that the ancestor allows the descendant to choose how to process the results. In this case, the ancestor has no further interference with the processing of the query, i.e. all possible results gathered ”below” the ancestor will be returned to the initiator instead of being aggregated back to the ancestor. The algorithm proceeds as follows:

1. The initiator of query q establishes a large number (e.g. 2^{128}) of tokens, t, and sends the number along with the query.

2. At query decomposition points, the current node splits the tokens equally among its participating children. If the query is *deferred*, the node can forward it to each of its participating children either as *expecting* or as *deferred*. If the query is *expecting*, the node must forward it as *expecting* to all its participating children. Figure 3.2 annotates each edge visited by the query with D or E, depending on whether the ancestor defers the query to the descendant or not.

3. Each successfully matched node sends to the last decomposition point (or directly to the initiator) the tokens it has gotten, along with the query result. With failed matches the tokens still need to be returned. In Figure 3.2, node */univer-*

sity/public/people/faculty returns the tokens to the initiator even though there are no possible matches on this branch (for query */university/.+/people/.+/.+*).

4. On result recomposition, the current node adds up the tokens and joins the results from its participating children. If its ancestor has deferred the query, the node must send the results and tokens to the initiator (e.g. */university/private* in Figure 3.2). Otherwise, the node has a choice and can send the results either to the initiator, or to its ancestor. In the former case, the node still has to let its ancestor know about shortcutting results to the initiator, since the ancestor is expecting to hear from it. In Figure 3.2 we exemplify this case by the dotted arrow between node */university/private/people/students* and */university/private/people*.

5. The query initiator learns of query completion after receiving the original number of tokens.

Discussion

Failed or delayed sub-queries can be detected at any level by timing out, and pretending that the missing tokens have been received. Alternatively, more sophisticated token regeneration protocols can be applied to recover lost tokens.

If the number of tokens t_n at the current node n becomes too small to be further split, a new number of tokens, t', can be generated at n. On query decomposition, n will delegate to each of its children a tuple of tokens consisting of t_n and a fraction of t'. If node m ”below” n decides to shortcut partial results to the initiator, it will send both t and its share of t' to the initiator; m will also let n know about the shortcut if n is expecting to hear from m. Upon receiving t' tokens, the initiator will replace them with t_n, and continue to wait for the original number of tokens, t. Alternatively, if n manages to recover all tokens t' (i.e. there are no shortcuts to the initiator ”below”

n) it further forwards t. In this case the initiator won't have to learn about t', and that an intermediate token regeneration process took place. Token regeneration can be generalized and performed on multiple levels by using n-tuples of tokens.

Note that in the query decomposition phase only decomposition points need to be visited. In Figure 3.2, node */university/public* can be skipped at decomposition if node */university* knows about node */university/public/people* (e.g. by caching a pointer to it). A similar argument can be made about node */university/private*. Thus, query forwarding may not necessarily follow all the edges in the namespace as long as all decomposition points are visited.

In the query result recomposition phase only decomposition points should be visited. All other nodes have a mere forwarding role and visiting them would only increase the cost of query processing without any benefit. For instance, aggregating results from */university/private/people* to */university/private* (assuming */university/private* would be expecting) is pointless since they can be sent directly to */university*. Thus we augment the expecting/deferred rules by allowing a node to be expecting results only if it is a decomposition point. In Figure 3.2 */university/private* cannot be expecting since it is not a decomposition point.

Finally we note that a node at a decomposition point can be expecting from some of its children, and be in the deferred state for the rest of them. This allows more flexibility and is the reason for associating processing states with edges and not nodes. The token protocol can naturally be extended and applied to any scheme that employs query decomposition into an arbitrary number of sub-queries, and that requires all partial results to be eventually collected by the initiator.

3.1.5 Namespace Management

Node Creation

Upon node creation, data structures used to hold node information are allocated and populated in a straightforward manner. The server that instantiates a node is known as the *owner* of that node and has complete control over the data named by the node, including meta-data and bookkeeping information, and handless all operations regarding the node, except queries which can be resolved by any of a node's replicas.

A directory's root node can be created by any server on behalf of the application that wants to start a new TerraDir directory. The root node's existence and address are both assumed to be either well-known, or disseminated in an application-specific manner.

Grafting Nodes and Local Namespaces

Non-root nodes are grafted onto the directory tree by adding edges from existing nodes to the new node. A request to graft a new node in a directory can be accepted or denied (subject to application-specific policies) by the owner of the new node's to be parent. After a node is added to the directory, new nodes can be added under that node without further notifying the owner of the parent. Nodes (and by closure arbitrary canonical trees consisting of nodes) can be added to a directory by just communicating with a single server that owns some part of the directory.

Multiple nodes in immediate proximity in the directory can be added much like a single node and with the same incurred overhead. Servers can define *local namespaces*, build the local structure invisibly from other peers in network, and then graft the resulting namespace to the desired place in the directory. A similar argument can be made regarding node removal in group.

Edge Removal

If all the nodes reachable from an edge are owned by the same server, removing the edge does not pose any particular problems. However, it may happen that some of the reachable nodes are owned by servers different than the one who initiates the removal. For those servers it may be unacceptable that their corresponding nodes be deleted from the current namespace. A similar situation occurs when changing a node's label (*node renaming*). Node renaming is equivalent to a series of node removals and then node graftings using the new names.

TerraDir supports these operations by augmenting the node creation protocol to include the notion of *prefix immutability*. When a node is grafted to its parent a guarantee is needed that the ancestors of that node will not change until after the node has itself been removed from the namespace. The prefix immutability property is enforced across a namespace by the following invariant (and its derived closure):

> *If a node has prefix immutability, its parent must have prefix immutability as well.*

Next we summarize some of the key properties of the prefix immutability granting and revocation protocols:

- *Operation locality:* Granting and revocation are local operations: a node can be granted immutability by its parent and cannot accept a revocation from its parent if it granted immutability to any of its children.

- *Asynchronous protocol:* Nodes can engage in immutability granting and revocation operations at any time, not just upon adding or removing nodes from the namespace.

- *Optional guarantees:* Upon grafting or anytime later, a node need not request prefix immutability.

- *Orthogonality to namespace membership:* Immutability denial does not preclude the denied node from being grafted in the namespace.

- *Application policies:* If a node is requested immutability by its descendants, such requests are satisfied by default if applicable. Default behavior can be overridden by applications by specifying customary policies for granting or revoking immutability.

3.1.6 Namespace Mutations and Eventual Consistency

While each node is identified by a label, its name is given by the succession of labels from the root to the node. The fully qualified name is kept as part of the node's state at the owner. The implication is that a change of any of the labels in a node's name must be reflected at the owner of the node. Otherwise, inconsistent behavior may occur where the node's owner resolves queries for names that have become invalid, or it rejects queries for names that are in fact valid. Correctness is violated in both cases and explicit mechanisms are required to handle them.

TerraDir does not impose consistency models at the directory level. We describe *path checks*, a technique that allows applications to define their own tradeoff between flexibility, overhead, and consistency guarantees. Path checks are a mechanism to ensure consistency by verifying that servers own the nodes that they claim as such, and can be configured to provide absolute guarantees (all queries are answered correctly) or *eventual consistency* — there is a configurable "window of vulnerability" after an update to the namespace during which queries may be answered incorrectly. Caching and replication require additional mechanisms to enforce namespace consistency. We discuss these aspects in Section 3.2.11.

Path Checks

Assume the directory is configured to have a window of vulnerability of τ_{PC} seconds. More exactly, every node in the directory must learn about changes to its name due to ancestor label changes within τ_{PC} seconds. For some applications (e.g. a phone directory) this window could be hours, while it may be on the order of minutes for others (e.g. a distributed file system). For each node, a counter is maintained to keep the number of seconds until the next path check.

Initially all nodes have their counters set to τ_{PC}. The root node's counter is maintained at τ_{PC}, as there are no consistency issues involving its name (/). The counters of all other nodes are decremented every second. When the counter of a node expires, its parent is contacted, the node's name is validated, and its counter is set to τ_{PC}. This is a path check. Improvements to the name validation procedure involves piggybacking path check information on query messages. Whenever a node forwards a query to one of its children, its counter τ^{p}_{left} is included in the message. Upon receiving the message, the child node can safely set its counter τ^{c}_{left} to:

$$\tau^{c}_{\mathit{left}} = \min(\max(\tau^{c}_{\mathit{left}}, \tau^{p}_{\mathit{left}}), \tau_{PC}) \tag{3.1}$$

The improvement is based on the observation that validating a node's name can be delayed to τ_{PC} with respect to changes in the parent's label, and to the maximum of its own counter and that of its parent with respect to changes in other ancestor labels. Counter propagation is part of the path propagation scheme, and the counters of all nodes encountered by the query are carried in query messages.

Path checks involve reconfiguration of node names that are descendants of the node whose name has changed. This excludes nodes that have been granted prefix immutability. We expect this to be the case for nodes in the upper levels of the hierarchy (e.g. */home*, */home/bujor*), or otherwise their names not to change very often with respect to the window of vulnerability. Note that server clocks do not require any synchronization since path checks are triggered based on intervals. Also, τ_{PC} need not be the same for all nodes in directory, yet it should remain constant for each node. If the window of vulnerability is not a directory-wise constant, then rule (3.1) for updating counters upon path propagation is augmented such that τ_{PC} refers to the node's window of vulnerability.

Path check is an information-pull mechanism and requires at most N messages every τ_{PC} seconds (assuming N nodes in the directory, and that every node is owned by a distinct server). As an alternative approach to path checks we note an information-push mechanism where changes in a node's label are recursively and without delays propagated to all its descendants. Assuming that the likelihood of change is uniformly distributed across all nodes (i.e. irrespective of their position in the hierarchy), the amortized cost of restructuring for the synchronous approach is reasonable, and no worse than that incurred with DHT-based overlay networks [SMK$^+$01, RFH$^+$01, RD01a]. More exactly, assuming a perfectly balanced tree namespace with an average branching factor of b, N nodes, and exactly one label change per node, we have that: changing the label for one of the top most nodes involves at most $\frac{N}{b}$ messages, for one of the second top most nodes at most $\frac{N}{b \cdot b}$ messages, and so on, for a total of $O(N \log_b N)$. This yields an average cost of $O(\log_b N)$ messages per label change.

The performance of the synchronous approach depends on how many label changes occur, whereas the performance of path checks depends on the rate of changes relative to the window of vulnerability. Path checks amortize the cost of having multiple namespace mutations within a τ_{PC} period. Patch checks are more advantageous than the synchronous approach if the rate of mutations is high relative to τ_{PC}. More exactly, if more than $\frac{N}{\log_b N}$ changes are expected to occur over τ_{PC} seconds, namespace consistency should be maintained using patch checks, otherwise using the synchronous push-based approach.

3.1.7 Hierarchies Using Overlay Networks that Virtualize the Object Namespace

Virtualized approaches help load balance because data items from one very popular site are served by different nodes; they are distributed randomly among participating peers. Similarly, routing load is distributed because paths to items exported by the same site are usually quite different. Just as importantly, virtualization of the namespace provides a clean, elegant abstraction of routing, with provable bounds on routing latency. In this section we discuss some of the implications of implementing a hierarchical structure on top of overlay networks that virtualize the object namespace. Among approaches that virtualize the object namespace we note overlay networks based on the DHT paradigm [RFH$^+$01, SMK$^+$01, RD01a, ZKJ01]. Our contention is that implementing hierarchical structures on top of virtualized networks is less appealing when compared to systems that integrate the hierarchical namespace in the overlay P2P network (e.g. TerraDir).

Application Specific Data Semantics

Virtualization discards useful application-specific information. The data used by many applications (file systems, auctions, resource discovery, etc.) is naturally described

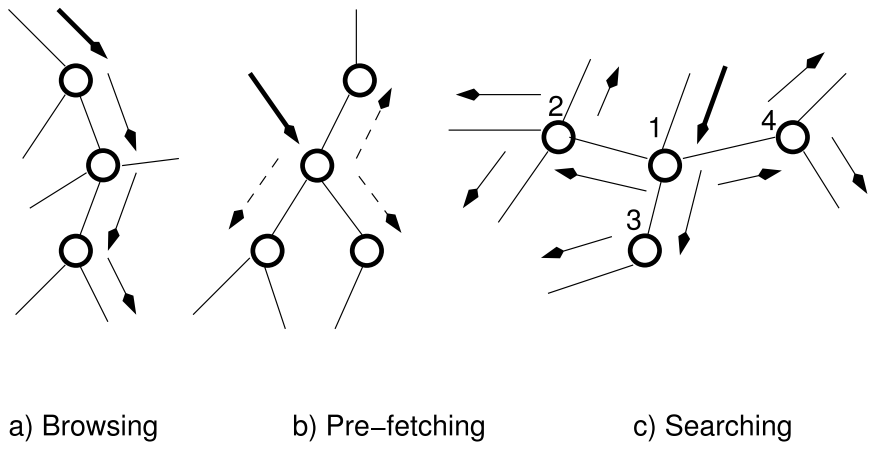

Figure 3.3: Browsing, prefetching and searching are examples of operations where object relationships can be exploited by integrating them with the routing procedure in the overlay network layer.

using hierarchies. A hierarchy exposes relationships between items near to each other in the hierarchy. Virtualization of the namespace discards this information. This may affect the semantics of data, and more importantly may have a negative operational performance impact as well.

Consider for instance *pre-fetching*. On retrieving the requested item, the system may additionally request in advance data items that are ”close” to the requested one, in case they will be needed in the short run. In a distributed file-system, when retrieving a file from a directory the system could transparently pre-fetch some or all the files in the same directory. It is usually the case that many of the files in the same directory are exported by the same site. The implication is that for a system like TerraDir pre-fetching can be performed locally without incurring additional overhead. This is not the case with virtualized namespace schemes. Items in the same directory that would otherwise be co-located are now dispersed throughout the network. Pre-fetching becomes a remote operation and its benefit questionable.

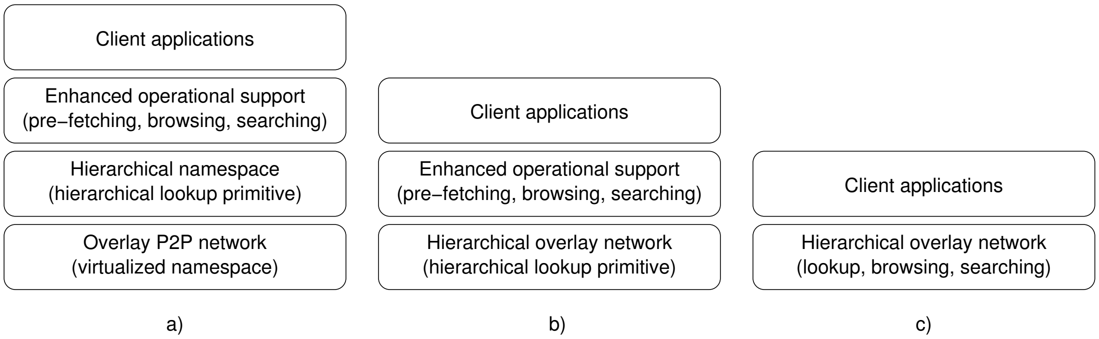
Client applications
Enhanced operational support
(pre−fetching, browsing, searching)
Hierarchical namespace
(hierarchical lookup primitive)
Overlay P2P network
(virtualized namespace)
a)
Client applications
Enhanced operational support
(pre−fetching, browsing, searching)
Hierarchical overlay network
(hierarchical lookup primitive)
b)
Client applications
Hierarchical overlay network
(lookup, browsing, searching)
c)

Figure 3.4: Alternative configurations for stacking the protocols: a) Using generic overlay networks that virtualize the object namespace. b) Separating the lookup primitive from complex hierarchical operations. c) Complete integration of the overlay network with the hierarchical namespace and complex operations.

Spatial Locality

Virtualization misses opportunities for exploiting spatial locality. Spatial locality is a property of a stream of requests whereby the next item to be requested is with reasonably high probability in the vicinity of the currently accessed item. Pre-fetching, browsing and searching are examples of operations that exhibit spatial locality for hierarchical structures. For these operations object relationships can be exploited by integrating them with the routing procedure in the overlay network layer.

We discussed pre-fetching in the previous section. We extend that argument by noting that even if entries in the same file-system directory are not co-located at the same site, virtualization of the namespace incurs a high performance penalty. In Figure 3.3.b we present an instance of pre-fetching on a hierarchical structure. For a system like TerraDir pre-fetching remote items can be performed in $O(1)$ network hops as opposed to $O(\log N)$, $O(dN^{1/d})$, etc. for virtualized namespace schemes.

Browsing is mode of operation where a current cursor is maintained, and at each step the cursor is advanced to one of the current node's neighboring nodes. *Searching* resembles a general browsing operation executed in parallel on all available paths from the current node. In Figure 3.3.a and 3.3.c we present instances of browsing and searching on a hierarchical structure. Similar to pre-fetching, each browsing or searching step requires $O(1)$ network hops for a system that preserves the hierarchical topology in the routing layer, and $O(\log N)$, etc. for systems that virtualize the namespace.

We argued that integrating the hierarchical namespace in the overlay network (as opposed to defining it on top of symmetric networks that virtualize the object namespace), enables the system to exploit object relationships defined by the hierarchy, and efficiently perform operations specific to hierarchical topologies (pre-fetching, browsing, searching, etc). We illustrate various configurations of the resulting architecture in Figure 3.4. It is the case that the potential of a hierarchical overlay network is fully exploited if enhanced operational support is integrated at this level. This is the main reason why TerraDir defines complex search query operations in the overlay network layer. Implementing search queries by exclusively using the lookup primitive would adversely affect searching performance. For instance, in Figure 3.3.c after a lookup for node 1, additional lookups are required for its children (nodes 2, 3, and 4), even though they might be hosted by the same server. In conclusion, the desired configuration of the resulting protocol stack is one featuring complete integration of the overlay network layer with the hierarchical namespace and enhanced operational support (see Figure 3.4.c).

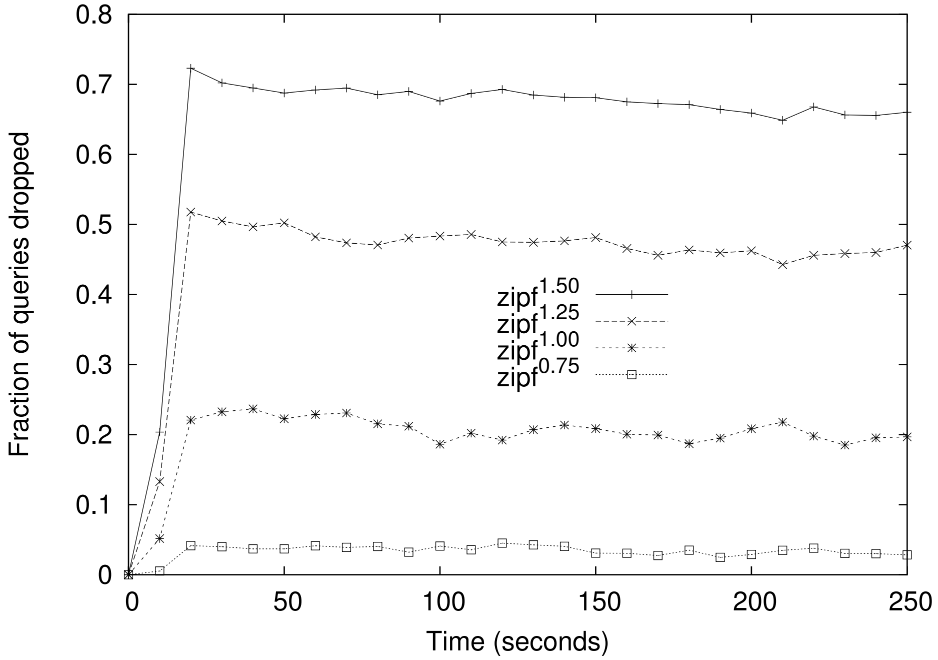

Figure 3.5: Queries dropped by Chord for various Zipf input streams with 4K servers and 32K nodes (keys). Targeted utilization (see Section 3.3.1): 15%.

Temporal Locality

Virtualization misses opportunities for exploiting temporal locality. Temporal locality is a property of a stream of requests whereby the next item to be requested is with reasonably high probability the same as the currently accessed item. Virtualization distributes load well assuming that the only form of locality present is spatial, not temporal. Stated another way, virtualization cannot improve load balance in the presence of single-item hot-spots; it can only distribute multiple (possibly related) data items across the network. In Figure 3.5 we show how Chord reacts to temporal locality present in the query stream.

Virtualized schemes do not define caching mechanisms in the overlay network layer to address hot-spots. Instead, applications build on top of these systems (e.g.

PAST [DR01], CFS [DKK$^+$01]) specify their own caching mechanisms. CFS for instance populates all the caches on the query path with the destination object's data after the lookup completes. While such ad-hoc forms of caching may help with the retrieval of data, they do not address the routing load incurred by hot-spots. Further, even if they did address routing load, the solution (replicating data) would be too heavyweight for a much simpler problem (routing congestion).

Various methods exist for replicating the distributed hash table. CAN [RFH$^+$01] replicates routing state by using multiple hash functions or by overloading coordinate zones. Multiple hash functions (mapping a key to multiple points or a server to multiple zones) define a uniform replication scheme that does not address hot-spots. Overloading coordinate zones is a local approach but less scalable as each server needs to know about every other server in its zone. In Chord [SMK$^+$01] the hash table can be replicated in an ad-hoc manner (a server caches another server's finger table), or uniformly by using multiple hash functions. An alternative proposal is to store a key on the k servers following the key's identifier on the circle.

None of these replication approaches can cope well with hot-spot scenarios and this is inherent in their design. Virtualization of the namespace decouples how items are located in the network from what items are responsible for generating routing load. By integrating the hierarchical namespace in the overlay network layer (e.g. TerraDir) this association is still apparent and can beneficially be used to perform targeted replication on selected items.

3.2 Hierarchical Routing with Soft-State Replicas

Routing performance on a hierarchical namespace is severely constrained by namespace topology. Even assuming uniformly distributed queries (both source-wise and destination-wise), servers hosting nodes at the top of the namespace will incur exponentially disproportionate more load than servers hosting leaf nodes. Furthermore, variations in user input may cause other parts of the namespace to be similarly afflicted (e.g. hot-spots). Finally, we do require some form of routing state redundancy to increase the resiliency of the routing procedure and routing state availability. While hierarchical bottlenecks can be addressed by static replication mechanisms [SBK02], the last two arguments call for an adaptive replication scheme.

3.2.1 Replication Protocol Overview

The replication protocol addresses replica and mapping management operations: (i) when, what, and where to replicate, (ii) when, and what to de-replicate, (iii) what servers to keep in a map, and how many, (iii) what servers in a map to advertise, and how many, and finally (iii) how to combine two maps for the same node.

We dynamically replicate nodes that comparatively incur more traffic on their owners. The amount of state replicated per node is minimal such that:

1. Lookup queries can be resolved by reaching a replica of the node. Thus replicated state includes node meta-data, and some mapping for the node. The mapping can be used by the query initiator if it further wishes to retrieve node data.

2. Routing through a replica needs to be functionally equivalent to routing through the original node. Therefore, a replica will also keep the context of the original node: mapping information for each of its neighboring nodes.

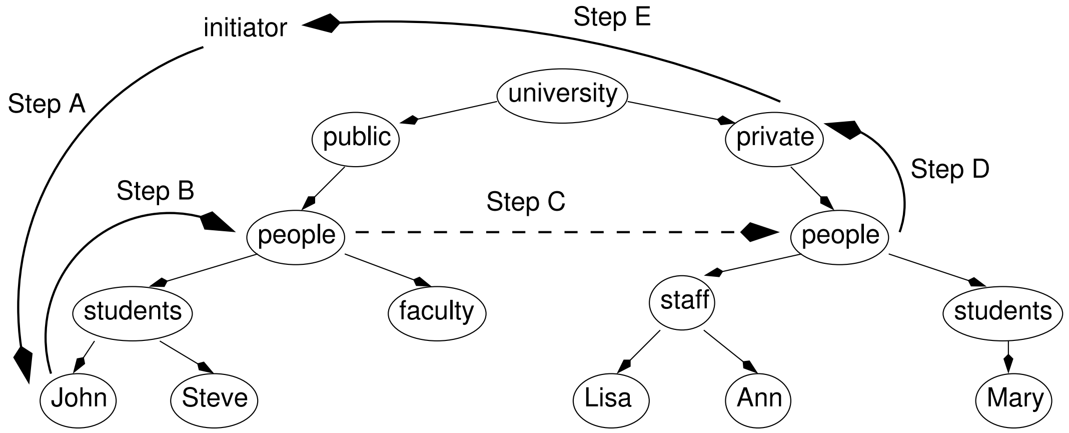

Figure 3.6: Route for a query (steps A–E) submitted to the owner of */university/public/people/students/John* and resolved to node */university/private*.

The term *host* will denote hereafter the owner, or one of the replicating servers (in the sense presented here) of a node. In Figure 3.6, the owner of node */university/public/people* hosts a replica of node */university/private/people* (the fact that both nodes have label *people* is a coincidence). Step C is thus abstract and does not incur query forwardings or network hops. Through replication not only is the owner of */university/public/people* present in other parts of the namespace (*/university/private/people*), but it can also further forward and resolve requests that come in on behalf of the replicated node. This serves a dual purpose: improved query latencies (additional shortcuts), and a mechanism to balance routing load by shedding some of it from the owner of */university/private/people*.

Note that we only replicate routing state and meta-data. Inconsistent routing state (nodes leaving or joining the system) can manifest in less precise forwarding steps. A query could reach a server on behalf of some node even though the server does not host the node any longer. In such cases incremental progress cannot be guaranteed for the current forwarding step. We assume that node meta-data is invariant or else that there are no consistency/freshness requirements for its update/use. Only the owner server of a node is allowed to modify meta-data, and replicas will keep the newest version that they have encountered.

3.2.2 Caching in TerraDir

Each participating server maintains a small cache with recently accessed nodes. Caches increase the routing state maintained per server to $O(\log S)$ (S is the size of the network) and substantially improve query latency. Caches are able to exploit both temporal and spatial locality in the query stream. Even in the absence of locality, the routing procedure benefits from caching by taking shortcuts over potentially large portions of the namespace. Routing resiliency is also augmented by the ability to jump over namespace partitions created by network failures.

A cache entry for a node consists solely of some mapping for that node. A hit in the cache cannot by itself bring query resolution. The query still needs to be forwarded to one of the resolved node's hosting servers (found in the node map). Caches provide only indirect routing functionality: they lack routing context, and act as mere pointers in the namespace. Step B of Figure 3.6 corresponds to the shortcut taken with a cache entry and thus node */university/public/people/students* is skipped.

Caches are ad-hoc state in the sense that there is no correlation between cache contents at different servers. Replacement, eviction and aging is performed locally. Cache entries are replaced using an LRU policy with an entry being touched whenever used in routing. We experimented with various cache replacement policies. In our experience the LRU replacement policy performs better than others such as "prefer higher nodes" or "prefer higher and recently accessed nodes". Applications that exhibit particular access behavior (e.g. most of the accesses are to higher nodes in the hierarchy) can

Table 3.1: Server-node relationships and state maintained.

Node\State	Name	Mapping	Data	Meta-data	Context
Owned	√	√	√	√	√
Replicated	√	√		√	√
Neighboring	√	√			
Cached	√	√			

specify customary cache replacement policies.

TerraDir caches differ from straightforward caches in that the path “so far”a is cached at every step along the query path; culminating in the entire path being cached at the source when the query completes (*path propagation*). Path propagation not only brings “far” nodes into the cache (the source caches the destination, and vice-versa), but also nodes from different levels of the tree, and nearby nodes. This mixture of close and far nodes performs significantly better than simply caching the query endpoints.

We summarize the various relationships between servers and nodes in Table 3.1, along with the type of state maintained. Note that cached nodes and neighboring nodes are similar except that cached nodes can be arbitrarily replaced and are not imposed by topological constraints of the namespace. The routing context (last column) refers to maintaining links to neighboring nodes in order to guarantee incremental progress.

3.2.3 Server Load Metrics

Replication is used to improve server load balance and routing resiliency when network failures are present. The first objective is pursued explicitly while the latter follows implicitly from the first: hosting servers for nodes with failed replicas will incur more load after failure than before, and will replicate again to meet new load conditions. Load balance is our first and most important fairness criterion.

We assume that a *normalized load metric* can be defined for all participating servers. A server's load in this metric is valued in the interval $[0, 1]$ with the semantics of the extremes being "no load" and "full capacity load" respectively. The normalization process ensures that system heterogeneity is accounted for. Additionally, the load metric must be:

1. *Linearly comparable:* given two load values, l_1 and l_2, the value l_1/l_2 should mean that server 1 has l_1/l_2 times more load than server 2.

2. *Locally defined:* the load must be defined exclusively based on local server information (busy cycles, memory requirements, incoming queue occupancy, etc), and independent of other servers' load condition. Load definition need not be the same for all servers. This accounts for machine heterogeneity.

The replication model is independent of load metric as long as such metrics are defined given the above requirements. In this study we evaluate the replication protocol using a simple but effective load measure: fraction of server busy time over a window period τ. The load window period is chosen such that it is short enough to allow for fast adaptation, and long enough to smooth over short periods of bursty activity. Stated differently, τ depends on what is meant by ”load balance”. If load balance is defined in terms of tens of seconds, then τ should be expressed in terms of seconds; if load balance is defined in terms of seconds, then τ should be expressed in terms of hundreds of milliseconds. The evaluation section considers τ to be half a second.

A server initiates load balancing sessions by replicating nodes on other servers when its load exceeds a *high-water threshold*, l_{high}. This threshold is a measure of the load-imbalance we are willing to tolerate, and can automatically be set in proportion to the overall system utilization. We will show that even simple policies (e.g. keeping a constant $l_{high} = 0.75$) perform well for a wide range of utilization factors. A server accepts to host new replicas if between the load of the requester and its own load there's a difference of at least l_{diff}. Any value in $[0.3, 0.7]$ is reasonable for l_{diff}, and load balancing performance is largely unaffected by parameter choice.

The chosen server load metric (fraction of busy time relative to τ) implicitly captures server overloading conditions (getting more requests than a server can process). The property follows from setting a threshold to determine highly-loaded servers: any server that is overloaded and has outstanding requests will soon have its load greater than l_{high} and start replicating its hosted nodes. More involved load metrics that take into account queue occupancy and are more responsive to flashing requests can be defined.

3.2.4 Inferring a Ranking for Replication Candidates

The unit of replication consists of a directory node. This allows a high degree of refinement and flexibility in that only the elements that trigger replication are selectively replicated. Replicating servers (i.e. all the nodes owned by a server in group) would require less state to be maintained per replicated node, but would unnecessarily replicate some of the nodes. When a server's load exceeds the high-water threshold, hosted nodes (owned or replicas) that comparatively incur more load on the server are further replicated. Load based node ranking is achieved by identifying the nodes for which processing is performed whenever routing a query.

Virtualized approaches decouple how items are located in the network from what items are responsible for generating routing load [KBS02]. Routing is not performed

”on behalf of” exported items and the TerraDir replication model is not applicable to these schemes. TerraDir does not virtualize the object namespace and message forwarding is performed on behalf of nodes. For instance, a query for node q will be forwarded from server S_1 to server S_2 on behalf of node n, where n is *most likely* hosted by S_2. Node n may not be hosted by S_2 if the replica was deleted and S_1 did not get a chance to find out about the deletion. In such cases S_2 cannot identify a hosted node on behalf of which the query was forwarded to it. In most cases servers are able to do proper identification.

The criteria for node ranking is given by assigning *node weights* to each node hosted by a server. The weight for each node is proportional to the load incurred by the server on the node’s behalf. Simple counter variables are maintained to implement node weights. With each incoming query the appropriate counter is incremented, and all counters are rescaled periodically to approximate recent demand patterns.

3.2.5 Replica Creation

The protocol for creating new replicas proceeds as follows. Assume source server S_S has load l_S.

1. Replication is triggered when a server’s load exceeds the high-water threshold, $l_S > l_{high}$. A server checks its load after each processed query.

2. Among all the servers that it knows about, S_S picks the one with minimum load, S_D. S_S makes this decision based on load information that it has for servers, not the actual load of servers. S_S contacts S_D and learns about its actual load, l_D.

3. If $l_S - l_D \geq l_{diff}$, S_S will replicate nodes on S_D. Given a node ranking for server S_S, w_i s.t. $w_1 \geq w_2 \geq \cdots \geq w_n$, the top ranked m nodes will be replicated on

S_D, where m is the smallest number s.t. $\sum_{i=1}^{m} w_i / \sum_{i=1}^{n} w_i \geq \frac{1}{2}\frac{l_S - l_D}{l_S}$.

4. S_S and S_D will adjust their loads to $l_S = l_S - \frac{1}{2}(l_S - l_D)$ and $l_D = l_D + \frac{1}{2}(l_S - l_D)$ to reflect the ideal load redistribution targeted after replication. This acts as a hysteresis and will prevent replica thrashing.

5. If $l_S - l_D \geq l_{diff}$ above is not met, then S_S makes another attempt of selecting a destination server, and the protocol continues with step 2. After a few failed attempts, S_S aborts the current replication session and initiates another one after a short delay.

3.2.6 Controlling the Extent of Replication

Our second fairness criterion is the amount of replicated state maintained per server. We need to bound the number of replicas, either globally or on a local basis. Otherwise, in response to fluctuating load, nodes will keep getting replicated, and the system could easily converge to extreme configurations where each node is hosted by every server. We constrain the number of replicas hosted by a server to be proportional to the number of nodes owned by the server.

The *replication factor*, k_{repl}, controls the maximum number of replicas hosted per server relative to the owned nodes. The replication factor need not be the same for all servers. Allowing servers to replicate nodes in proportion to the number of hosted nodes is a locally enforced and, we believe, fair policy. A locally enforced replication factor translates easily to global constraints. The overall number of replicas is bounded by the highest replication factor relative to the number of all nodes in the system. Note that we impose constraints on a per server basis. Nodes can still have as many replicas as globally allowed by the replication factor.

3.2.7 Replica Deletion

To heed the replication factor and at the same time allow the model to continuously adapt to changing demand patterns, node replicas have to be deleted. A hosting server may decide at any time to evict replicas that haven't been in use for a long time, i.e. low ranking nodes. Additionally, some replicas may have to be evicted (as dictated by k_{repl}) by a server S_D when some other server S_S requests that some of its nodes be replicated on S_D. In such cases, S_D will delete as many replicas as needed starting with the lowest ranking node and proceeding in increasing rank order.

Replica deletion is a local process as it involves only the server that hosts the replica. Other servers will learn about deletions in a lazy manner, or may not learn at all about some of the evictions. Inconsistent views are caused by stale mapping configurations and adversely affect routing performance. We do not specify any consistency model for managing mapping configurations. A limited form of control can be exercised by removing stale entries from maps when they are routed through servers. The degree of inconsistency can be further reduced by using inverse-mapping information.

3.2.8 Inverse-Mapping Digests

Maps provide a name resolution service by identifying some of the servers that host a node. Resolving node names to hosts is needed every time routing is done on behalf of the node. The inverse function, resolving a server to node names hosted by that server, improves performance and routing quality.

A straightforward enumeration of all hosted nodes by a server is expensive. Instead, TerraDir uses inverse-mapping approximations (*digests*) of the list of hosted nodes for each server. The replication protocol benefits from digests as follows. Each server generates a digest regarding its hosted nodes. Currently this is a Bloom fil-

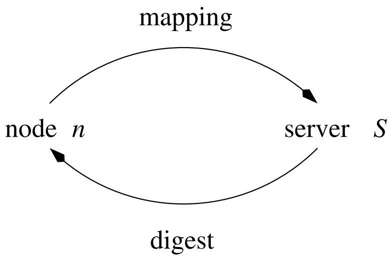

Figure 3.7: Node maps and server digests as associations between nodes and servers.

ter [Blo70] and its value is determined by hashing the names of all the nodes hosted by the corresponding server. The only allowed operation on a digest is testing node names against it, and producing a ”yes”/”no” answer with possible false positives.

Inverse mapping digests provide a second level of association between nodes and hosting servers (see Figure 3.7). A node’s map explicitly describes some of the servers that host the node. It may have false positives if a server in the map deletes the node’s replica. It may have false negatives if a server not in the map creates a new replica for the node, or if the number of replicas exceeds k_{map}. The digest associated with a server succinctly describes the nodes hosted by the server. It may have false positives if the server deletes one of its hosted replicas or due to the Bloom collision. It may have false negatives if the server creates a new replica for a node. Due to the redundant nature of association between nodes and servers, one form can be used to improve the quality of the other.

Digests are used to discover additional shortcuts in the namespace, and to maintain up-to-date node maps.

Discovering Additional Shortcuts

The standard routing algorithm is a minimizing procedure. A server S routing query q always chooses the closest node to q that it knows about, n, and forwards the query to one of the servers in n's map. Using inverse-mapping digests, S might indirectly know about some other node, h, that is even closer to q than n. Node h is discovered as follows.

1. Assume S generates all node names that it can infer, and let this set be Pref. Pref includes hosted, neighboring, and cached node names, as well as destination name q.

2. By performing prefix extractions, S will also include in Pref ancestor names —all the way to the root— for all these nodes. Each of the names in Pref can be tested against digests of servers that S knows about.

3. Assume a hit for some name $h_i \in \mathit{Pref}$ occurs in a digest associated with server S_i, and that h_i is closer to q than n. In this case S can optimize the routing by forwarding the query to S_i instead of some server from n's map.

4. By choosing h to be the closest h_i to q, server S is guaranteed to make the best decision it can in terms of namespace distance.

We illustrate the procedure in Figure 3.8. Server S hosts nodes */university/public/people/faculty* and */university/public/people/student/John*; names in Pref are shown using dashed ellipses. S also has an entry for */university/public/people/students/Steve* in its cache. The mapping for the cached entry includes one of the node's hosts, S_d, and S_d's digest. S_d also hosts */university/public*, and S gets a hit for node */university/public* in S_d's digest present at S. Thus S can forward the current query to S_d and skip node */university/public/people*.

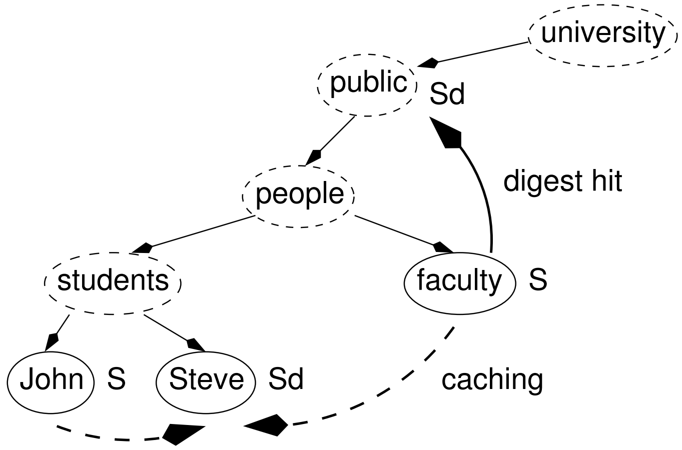

Figure 3.8: Shortcut taken from server S due to a hit of */university/public* in the digest of S_d present at S.

Pruning Node Maps

Consider server S that knows about node n by keeping some map of it. For each of the servers in the map, S keeps the corresponding inverse-mapping digest. Thus S can produce a potentially more accurate map for n by testing node n against each of these digests. Servers in n's map whose test fails can safely be eliminated from the map.

False positives associated with digests may preclude S from eliminating one or more servers from a map. Further, not only the node's map but some of the digests used to prune the map may be outdated as well. Despite the fact that node map pruning is a conservative operation, it enables the routing procedure to perform with close-to-perfect accuracy.

Cached and neighboring nodes are not included in inverse-mapping digests. Caches are expected to change frequently and their inclusion would derive less benefit, if any. Neighboring maps, just like cache entries, lack namespace context. Taking a shortcut due to a hit in digests that were generated to include neighboring nodes would not guarantee incremental progress.

3.2.9 Node Mapping Management

The node map associates a node name with a (possibly incomplete and inaccurate) list of servers that own or replicate the node. Servers maintain mapping information for owned, replicated, neighboring and cached nodes. The following policies govern how replica information is spread in the network, and how servers use replicas for routing queries in the network.

Map Size A node map contains at most k_{map} entries for scalability reasons. The constraint is in effect for maps kept at servers, as well as maps propagated through network messages. Maximum map size is orthogonal to how many replicas a node can get, or how many replicas a server is willing to manage.

New Replica Advertisement Servers advertise replicas created for their hosted nodes. Each server that has replicated one of its hosted nodes keeps entries for the most recently created replicas (up to k_{map}) in the node's map. These entries are advertised with every outgoing message that includes the node's map. Traffic in excess will quickly be diverted to newly created replicas.

Map Merging Maps are merged whenever a server keeps a map for a node, and an incoming query contains another map for the same node. Map merging is performed such that (i) the above conditions are met, and (ii) the rest of the entries in the resulting map are chosen at random from the choice left. The same two maps may have to be merged twice, once for the resulting map kept at the server, and a second time for the resulting map to be further propagated with the query currently processed.

Disseminating Replica Information Map replica configurations for a node are disseminated along query paths whenever information about the node is present in forwarded messages. Additionally, information about newly created replicas is back-propagated at each forwarding step if applicable. For instance, if server S_1 forwards a query to S_2 on behalf of node n, and S_2 has recently created any replicas for n, then S_2 will let S_1 know about such replicas.

Replica Selection Given a node's map present at a server, replica selection is performed by the server whenever a message needs to be forwarded to one of the node's hosts. For lookup messages any data or map replica of the node can be selected, whereas for data retrieval messages only data replicas can be selected. In both cases the destination host is chosen at random from the available choice, i.e. servers in the node's map present at the server.

Map Filtering Inverse-mapping digests are used to filter out stale entries from maps. Map filtering is a best-effort procedure carried out locally whenever a server updates the inverse-mapping information of other servers that it maintains. Our evaluation is based on a more conservative and thus less efficient approach: filtering is performed at replica selection and map merging, i.e. whenever node maps are used or modified at a server.

3.2.10 Query Paths and Information Flow

We disseminate information in a lazy manner by appending it to query messages. For nodes we disseminate mapping information, and for servers we disseminate load and inverse-mapping information. The number of nodes present in routing messages is bounded by the length of the query path. The number of servers is at most k_{map}

times the number of nodes. Through path propagation TerraDir servers learn about new servers, update load and inverse-mapping digest information for already known servers, spread replica configurations by combining node maps, and replace cache entries.

Load information is disseminated using version vectors. Servers always keep the most recent load of other servers that they learned about. Similarly, inverse-mapping digests are disseminated and updated using version vectors. Every time the set of nodes hosted by a server changes, the server's digest is updated and its version incremented. Note that server information disseminates faster than node information. Server information spreads with every map that the server is included in. A given server usually hosts more than one node, and thus may be present in more than one node's map. The efficacy of inverse-mapping digests benefits from this property.

We augment path propagation with *back-propagation*. Back-propagation is similar in spirit to other approaches that advocate populating the caches (or inserting replicas) on the path from source to destination with the node looked up [CSWH00, RFH^{+}01, DKK^{+}01]. Its utility rests on the assumption that lookups for the same node tend to access the same nodes in the network, once we get close enough to the target. In TerraDir the assumption holds since the last hop to the destination is usually one of the destination's neighbors. Thus if a target becomes a hot-spot, and repeated forwardings occur from the same set of nodes to the target, it makes sense to let such nodes know about the target's newly created replicas. In our experience back-propagation proved critical in developing highly-responsive decongestion protocols.

We illustrate a high level description of the routing procedure in Figure 3.10. Auxiliary function definitions are provided in Figure 3.9. Additionally we assume the existence of the following functions: *Parent* (the parent node of a node), *Children*

```
function Context (node $n$ → node list) : {Neighboring nodes of a node}
Parent($n$) ∪ Children($n$)

function Context (node list $N$ → node list) : {Neighboring nodes of a set of nodes}
$\bigcup_{n \in N}$ Context($n$)

function Nodes (server $s$ → node list) : {All the nodes a server knows about}
"/"∪Owned($s$)∪Replicated($s$)∪Cached($s$)∪Context(Owned($s$))∪Context(Replicated($s$))

function Servers (server $s$ → server list) : {All the servers a server knows about}
$\bigcup_{n \in \text{Nodes}(s)}$ Hosts($n$)

function Ancestors (node $n$ → node list) : {A node's ancestors}
if $n =$ "/" then
  nil
else
  Parent($n$) ∪ Ancestors(Parent($n$))
end if

function Ancestors (node list $N$ → node list) : {The ancestors of a set of nodes}
$\bigcup_{n \in N}$ Ancestors($n$)
```

Figure 3.9: Auxiliary functions used by the routing procedure (see Figure 3.10).

(the children nodes of a node), *Owned* (the nodes owned by a server), *Replicated* (the nodes replicated at a server), *Cached* (the nodes cached at a server), *Hosts* (the servers in a node's map), *Digest* (the digest associated with a server), and *Dist* (the distance between two nodes in the namespace distance metric).

3.2.11 Namespace Consistency with Caching and Replication

There are no consistency requirements for caching with respect to namespace mutations (e.g. changing node names, adding or removing nodes). If a cached shortcut is followed and the name of the cached node has changed in the meantime, correctness is not affected because cache entries cannot resolve a queries. Only the performance of the routing procedure may be affected. This scenario is no different than node replicas

```
procedure Forward (node $query$, node $curN$, server $curS$, server $init$, query state $qState$) $\{$Incoming $init$-initiated query $query$ on behalf of current node $curN$ to current server $curS\}$
if $curN \in (\text{Owned}(curS) \cup \text{Replicated}(curS))$ then
    adjust node weight for $curN$
end if
for all $s \in \text{Servers}(curS) \wedge s \in qState$ do
    update load and digest info for $s$ at $curS$ and $qState$
end for
for all $n \in qState$ do
    if $n \in \text{Nodes}(curS)$ then update $n$'s map (merging, pruning) at $curS$ and $qState$
    if $n \in \text{Replicated}(curS)$ then update $n$'s meta-data at $curS$ and $qState$
end for
update Cached$(curS)$ using $qState$
if $query \in (\text{Owned}(curS) \cup \text{Replicated}(curS))$ then
    query is resolved; return result to $init$
end if
$n \Leftarrow$ the closest node to $query, n \in \text{Nodes}(curS)$
$m \Leftarrow$ the closest node to $query, m \in \text{Ancestors}(\text{Nodes}(curS)) \wedge m \notin \text{Nodes}(curS) \wedge \exists s_m \in \text{Servers}(curS)$ s.t. $m \in \text{Digest}(s_m)$
if $\text{Dist}(m, query) < \text{Dist}(n, query)$ then
    add $m$ info to $qState$
    forward$(query, m, s_m, init, qState)$
else
    if $n \in \text{Cached}(curS)$ then
        LRU touch $n$
    end if
    choose $s_n \in \text{Hosts}(n)$
    add $n$ info to $qState$
    forward$(query, n, s_n, init, qState)$
end if
```

Figure 3.10: High level description of the TerraDir routing procedure.

being created and deleted on demand. If a followed shortcut is stale (due to namespace mutations or replica changes), the server that used the shortcut will remove it from its cache pool.

Replication on the other hand can cause correctness violations because a replicated node's state can by itself bring query resolution. The replication model deals exclusively with soft state and upon node addition, node removal and node name changes, none of a node's replicas is informed of such conditions. We extend the path checks mechanism described in Section 3.1.6 to account for replicated state. Each server has to initiate path checks for each of the nodes it replicates. Path checks are always performed by contacting the owner of an ancestor of the node being validated (as opposed to contacting any of the servers in the ancestor's map). This ensures that path checking is carried out on the original namespace without involving any replicated state. Upon path checking, node replicas are dropped if the node has itself been removed from the namespace.

Replication increases the cost of performing path checks. Assuming that nodes are uniformly replicated (i.e. nodes have the same number of replicas), the amortized cost of path checks increases by a factor of k_{repl}, from $O(\log_b N)$ (see Section 3.1.6) to $O(k_{repl} \log_b N)$ per label change.

Note that path checks for replicated nodes can be performed by contacting the owner of the node replicated (as opposed to the ancestor's owner). This approach has the advantage of synchronizing meta-data among nodes (the primary copies) and their replicas. As a third alternative we note replacing the path check mechanism for replicated nodes with one that drops replicas when the counter τ_{PC} expires.

3.3 Performance Evaluation

We evaluate the presented replication protocol in a TerraDir simulated environment. We focus on adaptivity to user input, system load balance and global utilization, stabilization, long-term behavior, and finally scalability.

3.3.1 Methodology

We consider $4,096$ servers. Service times are exponentially distributed with a mean of $T_s = 20$ milliseconds for each server. The mean query arrival rate is modeled with a Poisson distribution and varies from $\lambda = 2,000$ requests per second to $\lambda = 20,000$ requests per second, globally. Each server has a request queue of size 32 with queries arriving in excess being dropped. The application layer network time is constant at 25 milliseconds. We do not model network contention.

Let S be the network size and S_q be the average number of servers that a query is processed by (initiated, forwarded or resolved). Assuming no query drops, the average system utilization is $U = \lambda T_s \frac{S_q}{S} * 100\%$. We will target certain utilization factors and infer the required mean arrival rate to achieve them.

Lookups are initiated uniformly at source servers. Destination nodes are chosen either uniformly at random ($unif$ traces), or with locality according to the Zipf law of popularity vs. ranking ($zipf$ traces). A Zipf-like distribution [Zip49] sets the relative probability of a request for the i'th most popular node in proportion to Ω/i^{α}, where the order α is around 1, and Ω is a normalizing constant.

Our study involved both synthetic and real-world TerraDir namespaces.

- As example of a synthetic namespace (N_S) we will consider $32,767$ nodes arranged in a perfectly balanced binary tree. Uniform query streams for this namespace are denoted by $unif_S$, while streams with locality are denoted by $zipf_S^{\alpha}$. The order α covers the whole domain of interest: $.75$, 1, 1.25, and 1.5 for heavily skewed requests.

- File-systems are the most common hierarchical structures in use. We will consider one of the Coda [SKK$^+$90] servers (*barber*) logged throughout one month

Table 3.2: Cumulative popularity for the top θ nodes and Zipf order α.

$zipf^{\alpha}_{S\|C}$	$\theta = 10^0$	$\theta = 10^1$	$\theta = 10^2$	$\theta = 10^3$
$zipf^{0.75}_S$	2.0%	7.5%	18.3%	37.8%
$zipf^{1.00}_S$	9.1%	26.7%	47.3%	68.2%
$zipf^{1.25}_S$	23.3%	55.2%	77.5%	90.4%
$zipf^{1.50}_S$	38.4%	76.7%	92.8%	98.0%
$zipf^{0.75}_C$	1.5%	5.7%	14.0%	29.0%
$zipf^{1.00}_C$	8.3%	24.5%	43.3%	62.5%
$zipf^{1.25}_C$	22.9%	54.4%	76.4%	89.0%
$zipf^{1.50}_C$	38.4%	76.6%	92.6%	97.8%

of activity (January 1993). Files accessed during this month together with their ancestors were included in this namespace (N_C), for a total of $89,382$ nodes. Uniform query streams for N_C are denoted by $unif_C$, while streams with locality are denoted by $zipf^{\alpha}_C$. Our study involved traces of other Coda servers and the results presented are representative of all the servers considered; barber is the server with the highest activity during that period.

Both namespaces considered are mapped uniformly at random on the $4,096$ servers. In preparing the $zipf$ streams, node ranking was established by randomly ordering all the nodes in the namespace. We show in Table 3.2 the degree of locality for all the Zipf query streams considered. Node popularity is independent of rank ordering and slightly smaller for larger namespaces, as shown in the table ($zipf_C$ vs. $zipf_S$).

Query streams used in experiments are combinations of $unif$ and $zipf$ streams. For instance, we may prepend a sequence of $zipf$ streams with a $unif$ stream to allow a "cold" system to compensate for hierarchical bottlenecks and replicate the top nodes in the namespace. We will thus limit the interference between system warmup effects and those that are a result of the demand distribution.

Some experiments are run with locality streams that instantly and at random change node rankings, so that we can quantify how the model adapts to sudden variations in popularity (i.e. shifting hot-spots). We will only mix Zipf streams having the same order. Results are consistent and virtually the same for both namespaces; for space considerations we alternatively show only one of the cases.

3.3.2 Adaptation

We assess the model's capability to adapt to changing conditions, in this case locality variations in the query stream. For each of the two namespaces we run two types of streams for 250 seconds: $unif$ and $uzipf^{\alpha}$. The latter is given by the sequence $\{unif, zipf^{\alpha}, zipf^{\alpha}, zipf^{\alpha}, zipf^{\alpha}\}$.

Figure 3.11 shows the fraction of dropped queries relative to the query insertion rate for namespace N_S. For ease of presentation we allowed the $unif$ component of the $uzipf^{\alpha}$ streams to run longer in increments of 10 seconds, for various Zipf order values. The drops in the first seconds are due to hierarchical stabilization when the system replicates nodes at the top of the namespace hosted by overloaded servers. The spikes of the graph correspond to instantaneous and random changes in node popularity (for $uzipf^{1.25}$ such changes occur at seconds $70, 120, 170$, and 220). Figure 3.12 shows the system's reaction to overloading conditions in terms of the number of replicas created.

The same data is shown for namespace N_C in Figures 3.13 and 3.14. We doubled

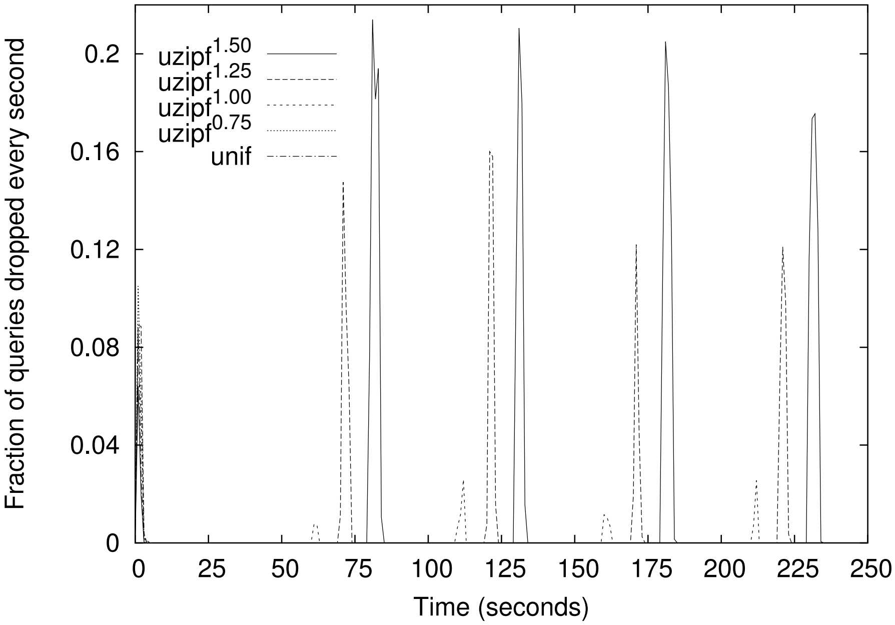

Figure 3.11: Dropped queries versus time for namespace N_S. Numbers are relative to λ $(2,000$ queries/second).

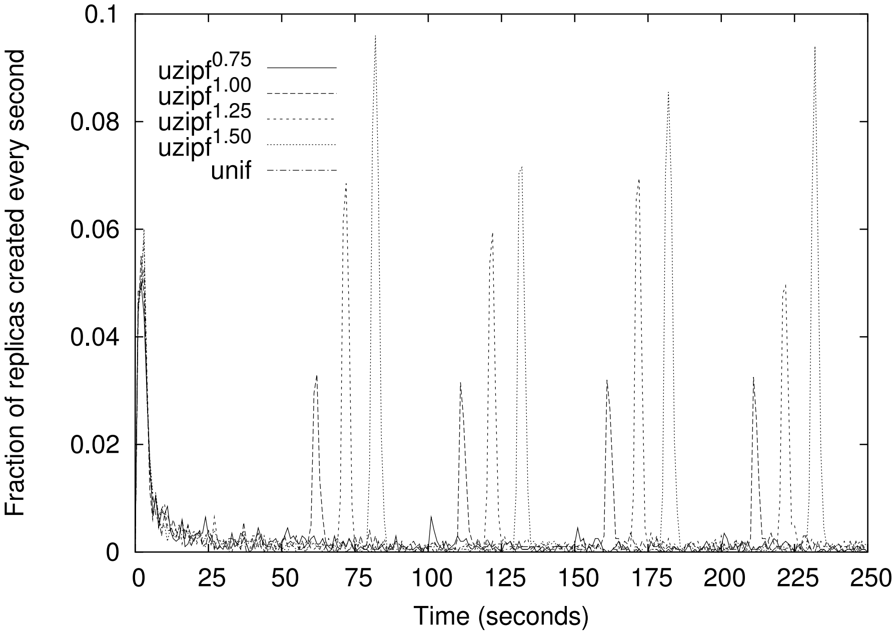

Figure 3.12: Created replicas over time for namespace N_S. Numbers are relative to λ $(2,000$ queries/second).

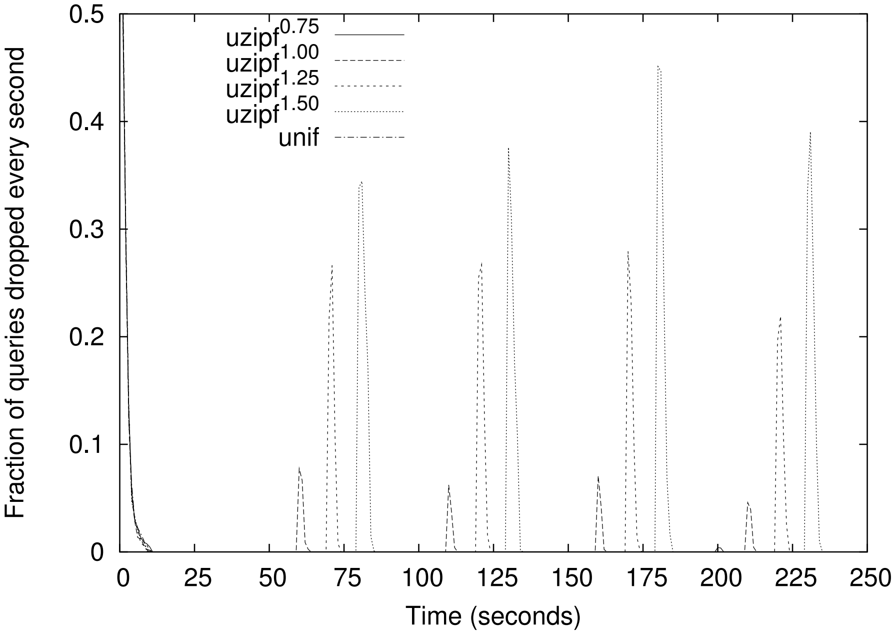

Figure 3.13: Dropped queries versus time for namespace N_C. Numbers are relative to λ $(4,000$ queries/second).

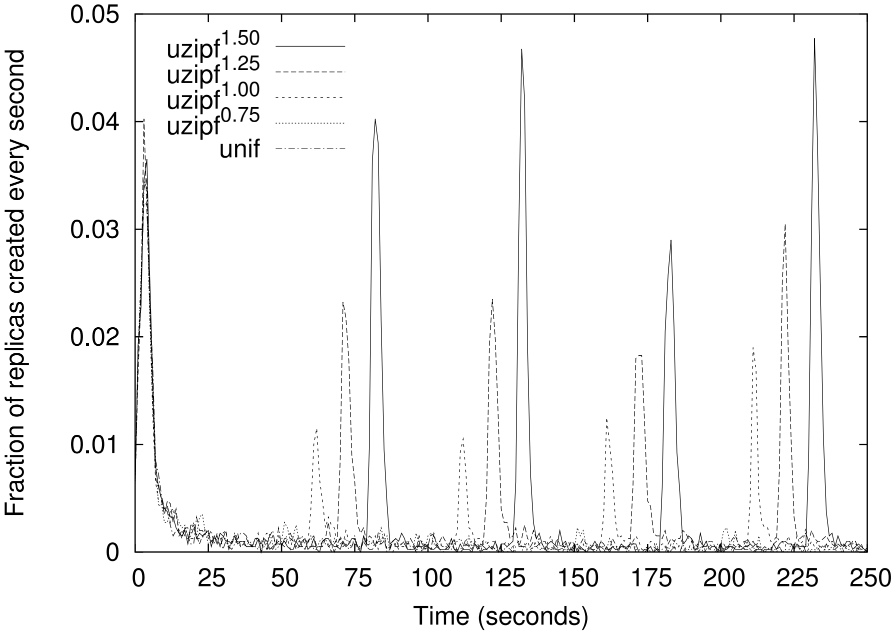

Figure 3.14: Created replicas over time for namespace N_C. Numbers are relative to λ $(4,000$ queries/second).

Table 3.3: Performance indices at end-of-run. Large numbers are rounded.

Stream	Q$[10^3]$	Lat	Drops	Bal	Repl
$unif_S$	500	11	287	925	893
$uzipf_S^{0.75}$	500	10.8	265	$1,093$	$1,058$
$uzipf_S^{1.00}$	500	9.9	582	$1,748$	$1,681$
$uzipf_S^{1.25}$	500	7.9	$2,433$	$2,589$	$2,421$
$uzipf_S^{1.50}$	500	6.3	$5,134$	$3,292$	$2,989$
$unif_C$	$1,000$	4.7	$6,282$	$1,515$	$1,253$
$uzipf_C^{0.75}$	$1,000$	4.7	$6,420$	$1,690$	$1,413$
$uzipf_C^{1.00}$	$1,000$	4.5	$8,330$	$2,563$	$2,238$
$uzipf_C^{1.25}$	$1,000$	3.9	$16,124$	$3,699$	$3,199$
$uzipf_C^{1.50}$	$1,000$	3.4	$24,351$	$4,513$	$3,747$

the query arrival rate for namespace N_C to keep the system at approximately the same utilization. The replication model adapts well to both hierarchical bottlenecks and sudden hot-spot fluctuations. Even with drastic changes in the request stream ($\alpha = 1.5$), less than half the queries (relative to λ) are lost and such for just a few seconds.

We show in Table 3.3 some of the performance parameters at the end of the run: the number of queries ran (Q), the average query latency as number of hops (Lat), the number of dropped queries ($Drops$), the number of load balancing messages (Bal), and finally the number of replica creations ($Repl$). $Repl$ is essentially the fraction of successful Bal messages that resulted in remote replica creations. No replicas were evicted.

Table 3.3 reiterates the same evidence as Figures 3.11– 3.14. The number of drops

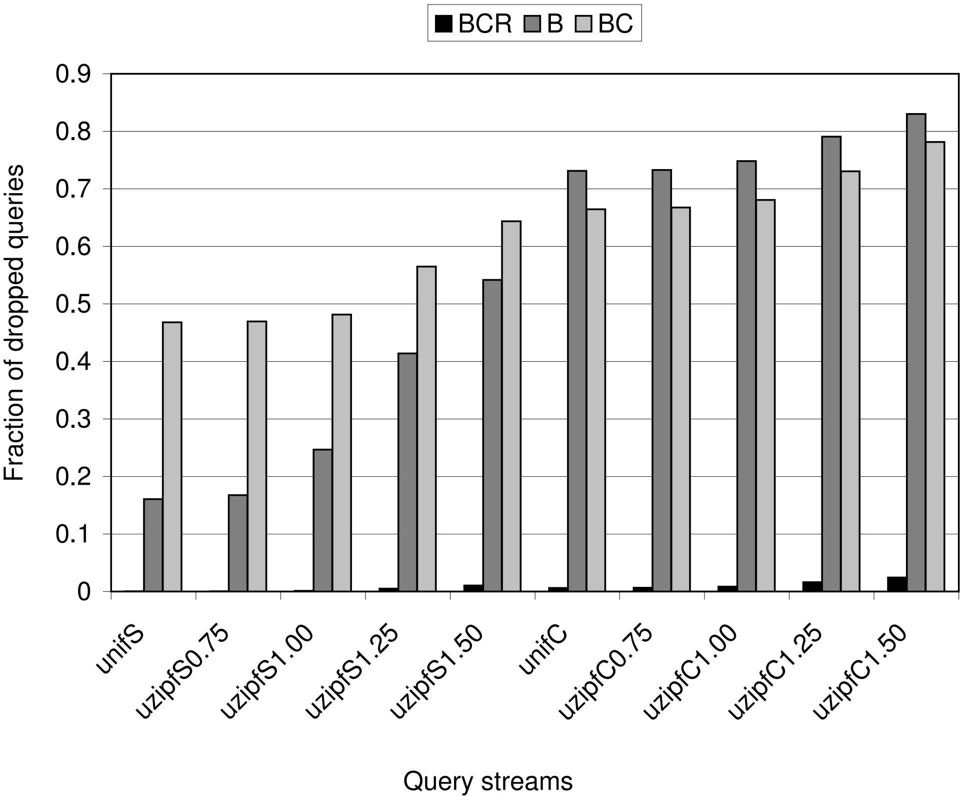

Figure 3.15: Fraction of dropped queries with combinations of the base system (B), caching (C), and replication (R).

is at most 2.5% when randomly changing highly skewed input ($\alpha = 1.5$) four times in a row over a short period of time (250 seconds). A quarter of the 2.5% are simulation side-effects due to hierarchical stabilization. Note that load balancing messages are at least two orders of magnitude less than the number of queries submitted, with the number of replicas created being correspondingly low. The replication protocol adapts efficiently.

Running the same experiments with replication disabled causes a large fraction of queries to be dropped to a point where the system is barely usable. If only caching is used while replication is still disabled, we see further aggravation in performance for namespace N_S, and slight improvements for namespace N_C. These points are made clear by Figure 3.15 where we compare the replication protocol with a base system, and one which employs only caching. The dissimilarity noted above is due to structural considerations: N_C is bushier and has its top nodes more of a bottleneck than N_S (compare along the streams axis for the base system). Caches alleviate this condition.

3.3.3 Utilization and Load Balance

Utilization distribution is our main fairness criterion. We define the computational utilization of a server over a second as the fraction of that second that the server is busy processing queries. We target three utilization factors: $U = 10\%; 20\%; 40\%$, and approximate them with query rates $\lambda = 2,000; 4,000; 8,000$ for namespace N_S, and $\lambda = 4,000; 10,000; 20,000$ for namespace N_C. We use query streams $unif$ and $uzipf^{1.00}$ as introduced previously.

Figure 3.16 shows on the left side the mean measured load, and the load on one of the most heavily loaded servers every second, for namespace N_S. Periodical peaks are due to locality changes in the $uzipf$ query stream. Note that the maximum load tends to go below l_{high} $(0.75$ in this case) if given enough time. With higher query rates the global mean load is itself approaching the threshold. In such cases it is proportionately harder to bring the maximum load below a constant high-water threshold. Note that servers at full utilization stay there only for a few seconds with each Zipf change, for all query rates shown.

We establish the transiency of highly-loaded server conditions when looking at larger than 1-second intervals. At each second we identify the most heavily loaded servers and average their load over 11 seconds. We show the load thus smoothed in the right side of the figure. The load distribution has improved substantially with the max-

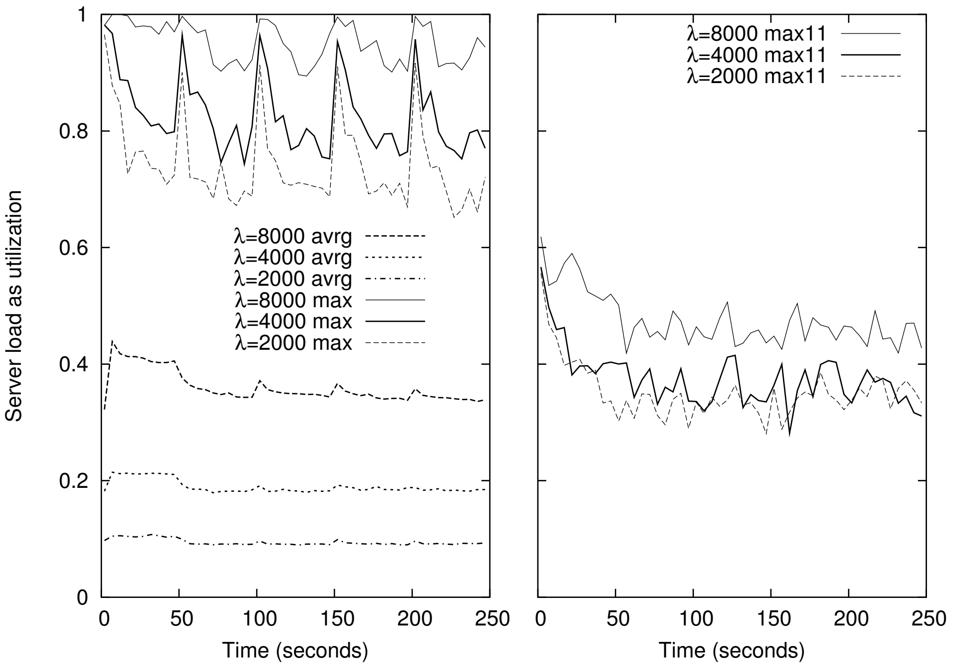

Figure 3.16: Average and maximum server load for query streams $uzipf_S^{1.00}$ (left). Maximum server load averaged over 11 sec. (right).

imum load approaching the mean, notably for higher λ values. Highly-loaded servers experience transient conditions, and by defining load balance over larger intervals we get increasingly better results.

For N_C we show uniform query results in Figure 3.17; the comments above apply here as well. Note that the query arrival rate has been increased to meet the utilization factors targeted.

In Figure 3.18 we show how the system reacts to hierarchical bottlenecks. For each level of namespace N_S we show the average number of replicas created for nodes on that level, with $unif$ and $uzipf^{1.00}$ query streams, and various query arrival rates (the data corresponds to the run in Figure 3.16). Note that nodes on level 2 tend to have more replicas than their ancestors. Pointers to nodes on level 2 have a high chance of

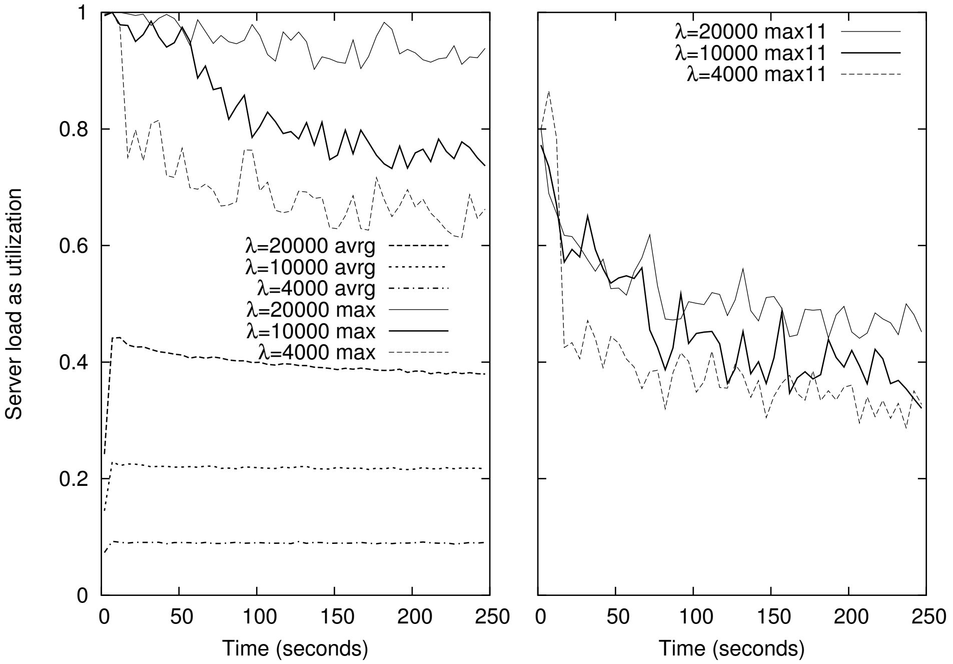

Figure 3.17: Average and maximum server load for query streams $unif_C$ (left). Maximum server load averaged over 11 seconds (right).

staying in a server's cache. Many of the routes that would normally go all the way up to nodes on levels 0 or 1 are thus using level 2 shortcuts. With nodes on level 3 for instance it is less likely that they are found in a server's cache since there are more nodes on this level than on level 2, and so forth.

3.3.4 Stabilization and Long-term Behavior

We are interested in establishing long-term behavior characteristics:

- whether with no changes in input patterns the replication model reaches a quiescent state where very few or no replicas are being created, and
- whether routing accuracy and efficiency is preserved with extreme changes in input patterns that will entail many replicas creations and deletions.

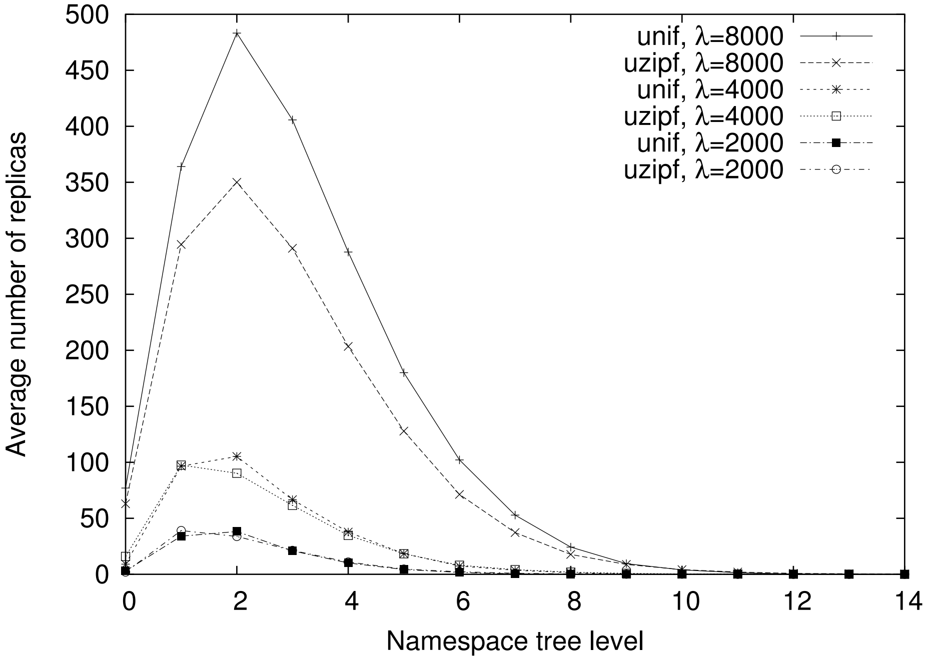

Figure 3.18: Average number of replicas created for each level of namespace N_S (the root is on level 0) with uniform and Zipf queries.

To asses stabilization we present results from runs with $unif$ and $uzipf^{1.00} = \{unif, zipf^{1.00}\}$ queries for $10,000$ seconds; the uniform component of $uzipf^{1.00}$ lasted for 100 seconds. 20 million queries were run for namespace N_S ($\lambda = 2,000$), and 40 million for N_C ($\lambda = 4,000$).

Figure 3.19 shows the number of replicas created every minute. The replication protocol reaches a rate of 2.5 replicas created per minute after $10,000$ seconds. The rate keeps decreasing beyond the timeframe shown, with the curve for the whole run resembling an exponentially decaying variable. The replica creation rate is equivalent to one replica created every $48,000$ queries run for namespace N_S, and $96,000$ queries for N_C. The replication protocol stabilizes with time for constant request distributions.

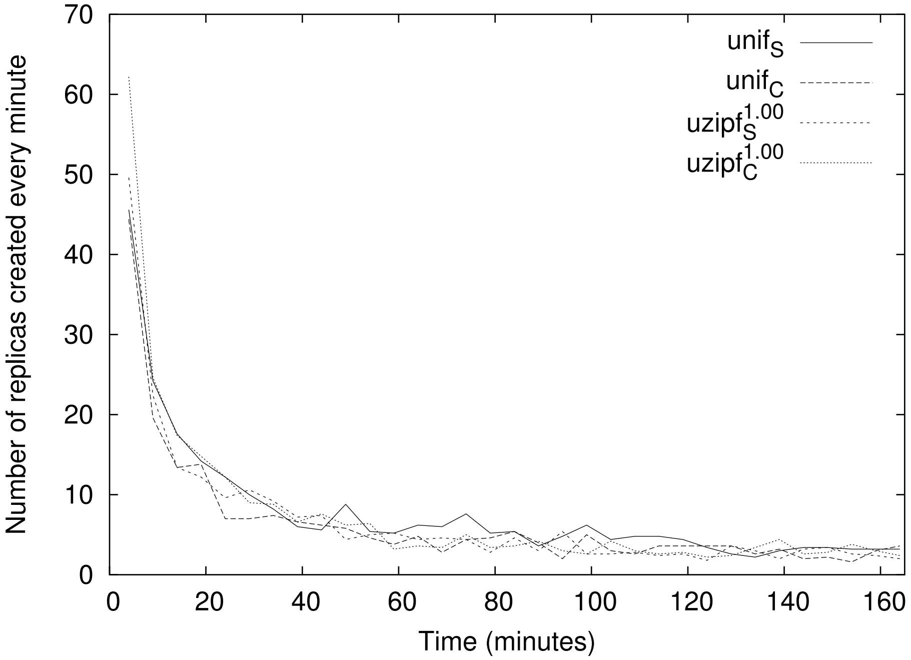

Figure 3.19: Replicas created over $10,000$ second runs for both namespaces, with uniform and Zipf queries.

The evaluation conducted so far was based on a replication factor $k_{repl} = 2$, and lead to very few replicas being deleted. We ran experiments with $k_{repl} = 0.125; 0.25; 0.5$ on query streams $uzipf^{1.50} = \{unif, [zipf^{1.50}]^{99}\}$. Each $zipf^{1.50}$ component lasted for 100 seconds, for an overall of $10,000$ seconds. Again, 20 million queries were run for namespace N_S and 40 million for N_C. Low replication factors together with repeated shifts of high-order hot-spots ($\alpha = 1.50$) trigger major changes in replica configurations.

The results for namespace N_S are shown in Table 3.4 with three map-pruning configurations: (i) using oracle information (lines O), i.e. knowing exactly whether a node is replicated or not at a server (ii) using inverse-mapping digests implemented as Bloom filters (lines I), and (iii) using the default and limited filtering approach as

Table 3.4: Comparison of map filtering configurations using the oracle approach (O), inverse-mapping digests (I), and the default approach (D). Large numbers are rounded.

k_{repl}	Lat	Q[10^3]	$\mathrm{R_c}$[10^3]	$\mathrm{R_d}$[10^3]	$\mathrm{M_u}$%
.5O	5/22.5	114	60.5	44.0	0
.5I	5/22.8	114	60.6	44.1	0.2
.5D	5.5/60.9	151	70.5	53.5	9.0
.25O	5/22.9	115	62.0	53.5	0
.25I	5/23.5	115	62.9	54.3	0.7
.25D	6.1/66.5	206	84.4	75.2	18.6
.125O	5/22.8	119	61.3	57.0	0
.125I	5/27	118	62.2	57.6	2.2
.125D	6.5/68.5	243	89.8	84.7	24.5

exemplified in Section 3.2.7 (lines D). Table columns correspond to the average and maximum query latency (Lat), the number of dropped queries (Q), the number of replica creation (R_c) and deletion (R_d) events. Finally, as a direct measure of routing accuracy, we show the percentage of forwarded messages for which no node identification was possible (M_u).

We note that even the default approach performs reasonably well. It drops 1% of queries under extremely stressful conditions ($\alpha = 1.50$, 99 changes in demand distribution, and $k_{repl} = 0.25$). However routing accuracy decays inversely proportional to k_{repl}, as indicated by the percentage of unidentified messages and the maximum query latency. Inverse-mapping digests are good approximations of optimal behavior (I lines vs. O lines). There is enough opportunity to transitively disseminate inverse-mapping

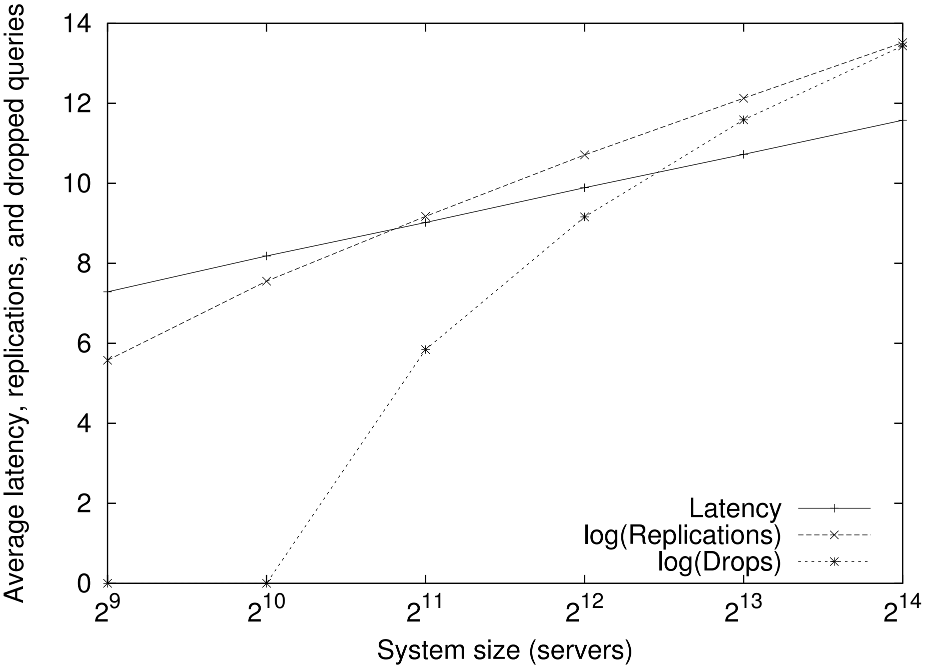

Figure 3.20: Scalability of average query latency, degree of replication, and dropped queries with system size.

information in the network, such that routing accuracy is maintained within the optimal range. Note that our implementation of inverse-mapping digests (8 bytes per digest) is subject to false positives, and thus not optimal itself with respect to explicit inverse-mapping schemes.

3.3.5 Scalability

We scale system size exponentially and look at query latency, the number of replica creations events, and the number of dropped queries.

Servers range from 2^9 to 2^{14} in incremental powers of 2. The number of nodes per server is kept constant at 8, with the overall number of nodes ranging from 2^{12} to 2^{17}, arranged in a perfectly balanced binary tree. Cache sizes are logarithmic in system size, ranging in increments of 2, from 18 to 28 cache slots per server. k_{repl} is kept constant 2, and k_{map} varies logarithmically with system size, from 2 to 7. Finally, λ is proportional to system size and takes values $250, 500 \ldots 8,000$. We use streams $uzipf^{1.00}$ similar to those described in Section 3.3.2. In fact, the settings in Section 3.3.2 for namespace N_S correspond to 2^{12} servers, 2^{15} nodes, caches of size 24, $k_{repl} = 2$, $k_{map} = 5$, and $\lambda = 2,000$.

Figure 3.20 shows the average query latency, the number of replication events, and the number of dropped queries as a function of system size. The last two are shown on a logarithmic scale. Latency scales logarithmically with system size, the degree of replication scales linearly, while the number of dropped queries scales proportionally and approaches linearity for large systems. The replication protocol is scalable.

3.4 Summary

We have presented a novel approach to replication in peer-to-peer systems. Our work stems from the observation that routing load incurred by P2P lookup and search services is orthogonal to load incurred by data retrieval, and can be efficiently managed if approached separately. Routing state is replicated in an ad-hoc manner, and the replication protocol can be combined with any data replication mechanism. Our study has been performed in the context of TerraDir which is an overlay network featuring hierarchical namespaces and search queries based on node attributes and partially qualified node specifiers.

The replication protocol addresses both hierarchical bottlenecks as well as those caused by temporal locality in the query stream. Adaptivity is based upon server load information profiled online. This enables it to deliver low latencies and good load balance even when demand is heavily skewed. Further, it withstands arbitrary and instantaneous changes in demand distribution. Our approach is lightweight, scalable, and deals with soft-state in the sense that there is no need for replica consistency models to be specified. New replicas are created and existing replicas are deleted in a decentralized manner without enforcing notification or synchronization requirements.

We have defined a method of resolving complex search queries through hierarchical query decomposition and result recomposition. The novelty of our approach stems from a token distribution mechanism that enables the query initiator to ascertain query completion. The scheme is flexible and offers the whole range of trade-offs between network bandwidth consumption and number of open connections required to complete the search.

We argued that having the object namespace define the peer-to-peer connectivity is central to the applicability of our replication model. In particular, overlay networks that virtualize the namespace cannot benefit from a similar mechanism of replicating routing information. We further argued that a non-virtualized object namespace maintains properties that are lost through virtualization. In this respect, we observed that virtualization destroys locality and opportunities for efficient browsing, pre-fetching and searching are lost. Finally, virtualization discards useful application-specific information, such as object relationships that would be otherwise exposed by a hierarchical namespace.

A few lessons have been learned throughout our study. First, simple heuristics for estimating server load, node weights, and replication policies perform well. We did not see a strong enough case for more sophisticated methods. Second, back-propagation as presented in this dissertation, or similar mechanisms are needed if we are to develop highly-responsive decongestion protocols. Finally, there is enough opportunity to transitively disseminate information in the network by exclusively using in-band means. Inverse-mapping functionality benefits from this property in maintaining close to optimal levels of routing accuracy even with drastic changes in replica configurations.

Chapter 4

Multi-Dimensional Quorum Sets for Read-Few Write-Many Replica Control Protocols

We define a new quorum protocol based on multi-dimensional logical structures called *d-spaces*. Our work is motivated by two dominant characteristics of presently deployed peer-to-peer infrastructures and proposed overlay networks. First, data reads account for the vast majority of accesses to items exported by P2P services. Second, active participation in such networks is highly transient, with users/sites frequently joining and leaving the service.

Use of d-space quorums provides the following advantages:

- They provide a better combination of efficiency and availability than existing methods. In particular, a d-space can be configured to give optimal communication complexity (quorum set sizes) for any given read/write access ratio, without compromising availability.
- Their use of local information allows much more flexibility than approaches based on global views. One implication of this flexibility is that membership changes do not require global reconfiguration phases.

We support one-copy equivalence [BG87] and serializable executions. Together, these give us one-copy serializability, which is also the correctness criteria supported by the read-one write-all approach (ROWA) [AAC94], the primary-copy approach [Sto79], the quorum consensus scheme [Gif79], and the available copies method [GSC$^+$83].

Supporting one-copy equivalence implies that all copies of an object should appear as a single item to clients. Efficiency requires that only a small fraction of the set of all copies be accessed during any read or write operation. This is especially important for read operations, which tend to represent the dominant type of access for most application classes. Finally, we would like to achieve both strict consistency and operational efficiency without sacrificing the availability of data, as this is the main reason for data replication.

There is a clear trade-off between the efficiency of a quorum protocol and its operational availability. The break-even point depends primarily on the logical structure (or the lack thereof) that arranges participating sites. We argue that protocols based on d-space quorums are superior to existing protocols both in terms of efficiency and in availability.

Sites participating in d-space quorums are arranged in a d-dimensional space; quorum groups are formed through subspace projection. For highly asymmetrical access patterns, our approach approximates the ROWA protocol with respect to read efficiency, while maintaining much better availability for write operations. The method subsumes some of the existing replica control protocols such as ROWA, and Grid [CAA92]. Further, d-space replica quorums are highly flexible in the sense that arbitrary read/write access ratios can easily be modeled. Finally, read accesses can be executed orders of magnitude more efficiently than updates without compromising the availability of either.

The remainder of this chapter is structured as follows. Section 4.1 defines quorum sets on multi-dimensional spaces, and the read-few write-many replica control protocol using d-spaces. In Section 4.2 we show that d-spaces are optimal with respect to communication complexity. In Section 4.3 we analyze the availability and compare the performance of our approach with the hierarchical quorum consensus method. A lightweight reconfiguration mechanism that combines local information and global views is discussed in Section 4.4. We summarize our findings and conclude the chapter in Section 4.5.

4.1 D-Space Quorums

We define quorum groups on multi-dimensional spaces and show how read-few write-many replica control protocols can be implemented using our method.

4.1.1 Definition

Assume we have N replicas of a data item arranged in a d-dimensional discrete space D, and that each dimension i in D is indexed from 1 to n_i, i.e. $N = \prod_{i=1}^{d} n_i$. D is an abstract space, and the N replicas do not necessarily correspond to physical copies. For this reason we will refer to replicas as *nodes* and not *sites*. In Section 4.4.2 we discuss how nodes are mapped to sites. Until then we assume that nodes (replicas) are sites (servers).

We choose k of the d dimensions in D, and let $u_{i_1}, u_{i_2}, \ldots, u_{i_k}$ be k arbitrary coordinates on selected dimensions $i_1, i_2, \ldots i_k$. Similarly, let $v_{j_1}, v_{j_2}, \ldots, v_{j_{d-k}}$ be arbitrary coordinates on the rest of the $d - k$ dimensions ($j_1, j_2, \ldots, j_{d-k}$). We define subspace U to be:

$$U = \{(x_1, x_2, \dots, x_d) | (x_1, x_2, \dots, x_d) \in D \wedge (x_{i_t} = u_{i_t})_{t=1 \dots k}\} \qquad (4.1)$$

and subspace V to be:

$$V = \{(x_1, x_2, \dots, x_d) | (x_1, x_2, \dots, x_d) \in D \wedge (x_{j_t} = v_{j_t})_{t=1 \dots d-k}\} \qquad (4.2)$$

Subspaces U and V overlap and their intersection is minimal. There exists a single point (node) common to U and V. The intersection point is given by coordinates $u_{i_1}, \dots, u_{i_k}, v_{j_1}, \dots, v_{j_{d-k}}$ written in canonical order. In a 2-dimensional space, U and V represent intersecting lines parallel to the two axes. Note that U and V factorize space D in the sense that $|D|$=$|U||V|$. Thus each of the two subspaces contains considerably fewer nodes that the original space D. In particular, the sum $|U| + |V|$ is minimized when $|U| = |V|$.

Let a read quorum group be V, and a write quorum group be $V \cup U$. Any two write quorum groups intersect, and any read quorum group intersects any other write quorum group. The quorum intersection property is satisfied and reads and writes are serializable. In particular, one-copy serializability [BG84] is guaranteed. On the left side of Figure 4.1 we illustrate read and write quorum groups for a 3-space quorum set. A read quorum requires 3 points, a write quorum requires $9 + 2 = 11$ points.

For clarity of exposition we will assume hereafter that the extension of the replica space is the same along all dimensions, i.e. $n_i = N^{\frac{1}{d}}$ for all $i = 1 \dots d$. All the results presented can easily be extended to account for the general case formally defined above. Given the new constraints, the replica space becomes a regular d-dimensional space, with space V containing $N^{\frac{k}{d}}$ points and space U containing $N^{\frac{d-k}{d}}$ points. A read quorum group will thus consist of $N^{\frac{k}{d}}$ nodes while a write quorum group will consist of $N^{\frac{k}{d}} + N^{\frac{d-k}{d}} - 1$ nodes.

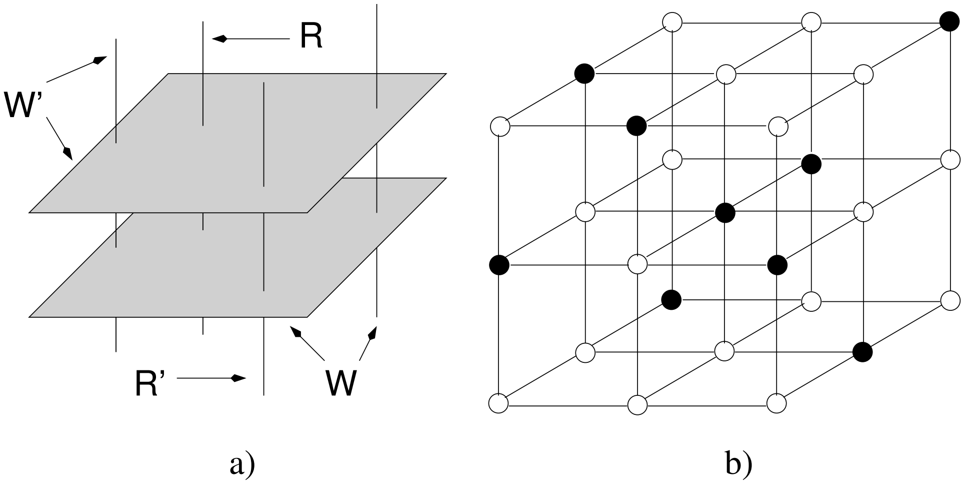

Figure 4.1: Example of 3-space read and write quorum groups. a) A read quorum (R and R') is a line and intersects all write quorums. A write quorum (W and W') is a plane and a line, and intersects all read quorums and all other write quorums. b) A cover of a plane (filled points) can be used instead of the plane to form a write quorum group.

4.1.2 The Read-Few Write-Many Approach

More interesting for distributed systems are quorums sets that are highly asymmetrical, i.e. for which d is relatively high compared to k. We call this instantiation of d-space quorum sets the *read-few write-many* approach (RFWM). One such instance is given by read quorum groups consisting each of $N^{\frac{1}{d}}$ nodes (a line), and write quorum groups consisting each of $N^{\frac{1}{d}} + N^{\frac{d-1}{d}} - 1$ nodes (a line and a hyper-plane).

Read-few write-many replica control protocols are amenable to scenarios where the frequency of read operations is orders of magnitude higher than the frequency of write operations. We call the ratio of frequencies for read and write operations the *read/write access ratio*. The communication complexity of a replica control protocol is the expected number of replicas to be contacted per operation. Our goal is to minimize the protocol's communication complexity, irrespective of operation type (i.e. read or write). Therefore, the ratio of quorum group sizes for read and write operations should be inverse to the read/write access ratio. By ρ_{wr} we denote the ratio of the write quorum size to the read quorum size. When ρ_{wr} equals the read/write access ratio, the communication complexity of a read-few write many replica control protocol is minimized. Any proportion of read and write operations can be modeled by choosing appropriate values for k and d (or more generally, for d and dimension extensions n_i).

4.1.3 Hyper-Plane Covers

Although we have defined read and write quorum groups in terms of projective subspaces, the set of $N^{\frac{d-1}{d}}$ nodes in a write quorum group need not strictly conform to the definition. We allow any cover of a hyper-plane to act as the second component of a write quorum group. A cover of a hyper-plane is any set of points such that their projection covers the whole hyper-plane. For our discrete space we are interested in the minimal cover of a hyper-plane, i.e. a cover having exactly $N^{\frac{d-1}{d}}$ points. Relaxing a write quorum group to a line combined with the cover of a hyper-plane greatly improves the availability of write operations. On the right side of Figure 4.1 we show a (minimal) cover for a plane that can be part of a 3-space write quorum group. Note that any of the three planes parallel to the base of the cube is covered by the cover shown.

The choice of strict subspaces and covers that we advocate is not unique. We could have relaxed the definition of a write quorum to be a hyper-plane and the cover of a line. In this case a read quorum group would be itself the cover of a line and would have excellent availability. However, the availability of write operations would be adversely affected to the point of making it not substantially better than in the read-one write-all protocol. In Figure 4.1 write availability would translate to having one of the three planes parallel to the base of the cube fully operational.

4.1.4 Replica Control Protocol

We briefly describe a read-few write-many replica control protocol that uses quorum consensus on d-spaces. We present the classical approach of using version numbers to identify the latest update, and locking to enforce mutual exclusion.

Access Semantics and Locking

Each copy has associated a version number. A read operation reads all the values of nodes in a read quorum group, together with their version numbers. The highest ranking version is considered to be the latest update of the item, and its associated value is the result of the read. A write operation reads the values of a subset of the nodes in the write quorum group, identifies the highest ranking version, and writes its associated value to another subset of the nodes in the group. For RFWM, a write operation will read from a line and write to the cover of a hyper-plane.

For both read and write operations, all the nodes in the corresponding quorum group are locked. If locking fails for a read group, a different group is chosen randomly or deterministically. Similarly, if locking of a write group fails on the line, the whole line needs to be replaced. If locking of a write group fails on the cover, the cover is changed by replacing only the unsuccessfully locked nodes. Locking failures are detected through timeouts. A timeout can be caused by failed nodes or by deadlocks. Locking lines cannot cause deadlocks since they form a partition of the replica space. Locking covers, on the other side, can easily deadlock two or more concurrently executing write operations. If no quorum group can be locked it means the system is unavailable due to massive node failures. We discuss fault tolerance in Section 4.4.

Performing operations on quorum groups can benefit from multicast services provided at either the network layer or the application layer. All the nodes in a quorum group can be members of the same multicast group and accessed using available multicast primitives.

The locking procedure described above provides transactional semantics for accesses, and ensures one-copy serializability. If single-access semantics is desired, read quorums need not be locked. A read operation can be performed by reading a read quorum group and identifying the value with the highest ranking version. Similarly, the read component of a write quorum (a line for RFWM) need not be locked for write operations. However, the write component of a write quorum group needs to be locked to guarantee single-update consistency.

Locking and Replication Granularity

The performance of a replica control protocol also depends on the update semantics, and the granularity of locks and of the replicated objects.

Update atomicity can be defined irrespective of the size of a data item. The cost of performing an update grows in proportion to both the size of the data item (*locking granularity*), as well as the number of items updated (write quorum size). *Partial writes* allow updates to occur on granularity smaller than the locking granularity. Distributed file-systems benefit from partial writes semantics (e.g. Coda [SKK$^+$90], Echo [HBJ$^+$90]) since they usually offer locking granularity on a file, or even larger data units. Performing partial updates in version-based replica control protocols comes at the price of more complex algorithms. This is because only the most recent version of a data item can safely be replaced through a partial write in such protocols. We assumed total update semantics in the description above.

Replication granularity is orthogonal to locking granularity, and affects the state maintained per data item by the replica control protocol. At one extreme, replication granularity is the same as locking granularity. In this case, a d-space needs to be maintained individually for each object. The flexibility of selectively replicating objects at nodes (and thus allowing fine-grained control over the amount of replicated state), is achieved through high protocol state storage costs. At the other extreme, the granularity of replication is given by the set of all data items that the system manages. A node will replicate either all objects or none. Only one d-space needs to be maintained in this case for the whole system, keeping the protocol's state overhead to a minimum. In between the two extremes, various trade-offs between protocol state and replication flexibility can be attained by replicating data items in group.

Note that for the case when the granularity of replication covers all data items, objects need not be effectively replicated at all sites. Instead, some of the nodes can maintain pointers to other nodes that actually store the replica. When a request arrives at a node holding a pointer, the request is forwarded to the node storing the replica. The cost of storing a replica is replaced by the cost of storing a pointer to it. However, node failures are correlated. When a node fails, all nodes that hold pointers to the failed node will logically fail as well. This also implies that node failure semantics changes. A node can be considered as failed for some data items (for which the node holds pointers to failed nodes), but operational for other data items (for which the node stores actual replicas).

4.2 Optimality of Communication Complexity

As noted above we expect read and write quorum group sizes to be inversely proportional to the access frequency for the corresponding operations. We argue that replica control protocols using d-spaces provide optimal communication complexity, i.e. read and write quorum sizes are minimal for a given access ratio. We require the following constraints on any quorum set defined on d-spaces:

QS1 Each read (write) quorum group in the set has r (w) nodes. The condition ensures that the message complexity of an operation is independent of the quorum group chosen.

QS2 Each node appears in at least one quorum group. The condition ensures that all copies are used effectively.

QS3 Each node is contained in the same number of read (write) quorum groups. The condition ensures uniform load sharing over the set of all copies (assuming quorum groups are selected uniformly at random when performing operations).

Given conditions QS1-3, Theorem 1 states the optimality of the approach. The following lemma will help us prove the theorem.

Lemma 1. *A set $\mathcal{W}$ that intersects all elements of a read quorum set satisfying conditions QS1-3 contains at least $w = N/r$ elements.*

Proof. Assume each node is contained in g distinct read quorum groups (by QS3). We also have that $g > 0$ (by QS2). The total number of nodes, considering all duplicates as distinct elements, is gN. Since there are r nodes in each read quorum group (by QS1), there are gN/r read groups in the read quorum set.

We construct $\mathcal{W}$ starting with the empty set, such that $\mathcal{W}$ intersects all gN/r read quorum groups. Every node added to $\mathcal{W}$ is contained in exactly g of the read quorum groups, and ensures the intersection of $\mathcal{W}$ with the corresponding groups. Thus, with the addition of one node we can cover the intersection of $\mathcal{W}$ with at most g groups in the quorum set. Since there are gN/r quorum groups, at least N/r nodes need be added to $\mathcal{W}$ to cover its intersection with all groups in the read quorum set. □

Theorem 1. *Read quorum sets defined using d-spaces are optimal with respect to quorum group size for any read/write access ratio ρ_{wr}. Write quorum sets are optimal within a factor of* 2.

Proof. Let k and d be such that such that $N^{\frac{d-k}{d}} \approx N^{\frac{k}{d}}\rho_{wr}$. We define d-space read quorum groups of size $N^{\frac{k}{d}}$ and write quorum groups of size $N^{\frac{k}{d}} + N^{\frac{d-k}{d}} - 1$ in the usual manner. We have that $(N^{\frac{k}{d}}+N^{\frac{d-k}{d}}-1)/N^{\frac{k}{d}} = O(w/r)$, and the quorums satisfy the read/write access ratio.

A read quorum group contains $N^{\frac{k}{d}}$ nodes. Every node appears in exactly one of the read quorum groups ($g = 1$). In fact, the read quorum set defines a partition on the set of all copies, where each member of the partition has the same number of nodes. Conditions QS1-3 are thus satisfied and by Lemma 1 we have that write quorum groups must contain at least $N^{\frac{d-k}{d}}$ elements. Since $N^{\frac{k}{d}} + N^{\frac{d-k}{d}} - 1 \leq 2N^{\frac{d-k}{d}}$ (assuming $k \leq d - k$), we have that write quorum groups are within a factor of 2 from the optimal size.

Read quorum groups cannot contain less than $N^{\frac{k}{d}}$ nodes since that would proportionately increase the size of write quorums (as given by Lemma 1). This would break the read/write access ratio. Thus, read quorum groups are optimal with respect to size. □

Note that write quorum groups are within a factor of 2 from optimality due to their V subspace component ($N^{\frac{k}{d}}$). For read-few write-many protocols we have that $N^{\frac{k}{d}} \ll N^{\frac{d-k}{d}}$, and write quorum sizes are themselves close to optimality.

We describe how k and d can be chosen such that the access ratio condition is satisfied. Given N nodes and access ratio ρ_{wr}, we are looking for quorum sizes for write and read operations, w and r, such that (i) $w = N/r$, and (ii) $\rho_{wr} = w/r$. Thus we have that $w = \sqrt{N\rho_{wr}}$ and $r = \sqrt{N/\rho_{wr}}$. N can be factorized in the list of its prime factors. The list can then be partitioned in two sublists such that the multiplication of prime factors in one list approximates w, and in the second list approximates r.

Given w and r, values for k and d can easily be identified such that $N^{\frac{k}{d}}$ approximates r, and $N^{\frac{k}{d}}+N^{\frac{d-k}{d}}-1$ approximates w. For the general case, where the extension of the replica space does not have to be the same along all dimensions, we have more flexibility in choosing the parameters. If N has few prime factors (e.g. N is a prime number itself), a neighboring number of N can be factorized instead of N. In this case, a few holes will be present in the structure, and quorum groups containing the holes will not be operational. In Section 4.4.2 we discuss how nodes can be mapped onto sites such that N is chosen at will, and any number of physical copies is naturally accommodated.

4.3 Availability Analysis

Fault tolerance is reflected in the availability of the last update (data availability), and the availability of read and write operations. We establish these availabilities next. We assume that the network is reliable, node failures are both independent and fail-stop [SS83], and all nodes are identical. Let p be the probability of a node being operational, i.e. the node's availability. Given p, the probability of finding m operational nodes among the N nodes is given by the binomial distribution:

$$b(N,m,p) = \left(\begin{array}{c} N \\ \\ m \end{array} \right) p^m (1-p)^{N-m} \tag{4.3}$$

4.3.1 Data Availability

Data availability is expressed in terms of the probability that the last update of an item is accessible on at least one of the nodes that stores a copy. Since the last update is given by the last successfully executed write operation, at the time of the update $N^{\frac{d-k}{d}}$ sites have committed the new value. The probability that at least one of these sites will survive until the next update occurs is given by:

$$\alpha^D = 1 - (1-p)^{N^{\frac{d-k}{d}}} \tag{4.4}$$

Note that data availability depends on the size of a write quorum, and thus implicitly captures the logical structure of a d-space. However, it does not depend directly on the particular organization of sites that is imposed by d-spaces. The same formula applies to any scheme that updates quorums of the given size (in this case $N^{\frac{d-k}{d}}$).

4.3.2 Read Availability

The availability of read operations is the probability that the operation concludes successfully, assuming no state changes while it is in progress. Read operations are robust: any line of $N^{\frac{1}{d}}$ nodes needs to be operational for a read to succeed. There are $N^{\frac{d-1}{d}}$ candidate lines to choose from. A line is available with probability $p^{N^{\frac{1}{d}}}$, while m

selected lines are available with probability:

$$\alpha_{line}(m) = p^{mN^{\frac{1}{d}}} \tag{4.5}$$

Knowing that there are $N^{\frac{d-1}{d}}$ potential lines to choose from, the availability of read operations is given by:

$$\begin{aligned} \alpha^R_{RFWM} &= 1 - (1 - \alpha_{line}(1))^{N^{\frac{d-1}{d}}} \\ &= 1 - (1 - p^{N^{\frac{1}{d}}})^{N^{\frac{d-1}{d}}} \end{aligned} \tag{4.6}$$

For the general case we have that:

$$\alpha^R_{RFWM} = 1 - (1 - p^{N^{\frac{k}{d}}})^{N^{\frac{d-k}{d}}} \tag{4.7}$$

The read availability of the hierarchical quorum consensus method is expressed recursively at each level [Kum91]. A logical node on level i is considered available if at least r_{i+1} of its l_{i+1} children are available. The availability of read operations, α^R_{HQC}, corresponds to the availability of the root node, while each leaf node is operational with independent probability p.

$$\begin{aligned} \alpha^R_{HQC} &= \alpha^R_0 \\ \alpha^R_i &= \sum_{j=r_{i+1}}^{l_{i+1}} \begin{pmatrix} l_{i+1} \\ j \end{pmatrix} (\alpha^R_{i+1})^j (1 - \alpha^R_{i+1})^{l_{i+1}-j} \\ \alpha^R_m &= p \end{aligned} \tag{4.8}$$

For reference we also present the read availability of the read-one write-all approach, and the weighted voting approach. [1]

[1] Read availability for ROWA: $\alpha^R_{ROWA} = 1 - p^N$. Read availability for weighted voting assuming one vote per node: $\alpha^R_{VOT} = \sum_{m=v_r}^{N} b(N, m, p)$.

4.3.3 Write Availability

ROWA is deemed unsatisfactory due to its stringent requirement that all copies be available whenever an update occurs. The read-few write-many approach approximates ROWA from the read efficiency standpoint, and dramatically improves over its write availability. The availability of write operations is the probability that the operation concludes successfully, assuming no state changes while it is in progress. A write is successful if one line of $N^{\frac{1}{d}}$ nodes together with a cover of a hyper-plane of $N^{\frac{d-1}{d}}$ nodes can be accessed. Note that since the line already covers one node in the hyper-plane, only $N^{\frac{d-1}{d}} - 1$ nodes need to be further covered. More generally, a write is successful if m lines and a cover of $N^{\frac{d-1}{d}} - m$ nodes are operational.

The cover of a hyper-plane of m nodes is available if there is at least one available node in each of the m corresponding lines. We further require that each such line not be fully available. At least one site, but not all of them, is available in a line with probability:

$$\begin{aligned} \alpha_{pointC} &= \sum_{m=1}^{N^{\frac{1}{d}}-1} b(N^{\frac{1}{d}}, m, p) \\ &= 1 - p^{N^{\frac{1}{d}}} - (1-p)^{N^{\frac{1}{d}}} \end{aligned} \tag{4.9}$$

The availability of a hyper-plane cover of m *selected* nodes is:

$$\begin{aligned} \alpha_{planeC}(m) &= (\alpha_{pointC})^m \\ &= (1 - p^{N^{\frac{1}{d}}} - (1-p)^{N^{\frac{1}{d}}})^m \end{aligned} \tag{4.10}$$

To compute the availability of write operations we add the probabilities for all combinations of m available lines (4.5) and $N^{\frac{d-1}{d}} - m$ additional nodes available in a cover (4.10):

$$\begin{aligned} \alpha^W_{RFWM} &= \sum_{m=1}^{N^{\frac{d-1}{d}}} \begin{pmatrix} N^{\frac{d-1}{d}} \\ m \end{pmatrix} \alpha_{line}(m) \cdot \alpha_{planeC}(N^{\frac{d-1}{d}} - m) \\ &= (\alpha_{line}(1) + \alpha_{planeC}(1))^{N^{\frac{d-1}{d}}} - \alpha_{planeC}(N^{\frac{d-1}{d}}) \\ &= (1 - (1-p)^{N^{\frac{1}{d}}})^{N^{\frac{d-1}{d}}} - (1 - p^{N^{\frac{1}{d}}} - (1-p)^{N^{\frac{1}{d}}})^{N^{\frac{d-1}{d}}} \end{aligned} \tag{4.11}$$

For the general case we have that:

$$\alpha^W_{RFWM} = (1 - (1-p)^{N^{\frac{k}{d}}})^{N^{\frac{d-k}{d}}} - (1 - p^{N^{\frac{k}{d}}} - (1-p)^{N^{\frac{k}{d}}})^{N^{\frac{d-k}{d}}} \tag{4.12}$$

The write availability of the hierarchical quorum consensus approach, α^W_{HQC}, is expressed recursively and is similar to the read availability of the same method (4.8), except that the series $r_1, r_2 \ldots r_m$ is used instead of $w_1, w_2, \ldots w_m$ when defining α^W_i. For reference we also present the write availability of the read-one write-all approach, and the weighted voting approach. [2]

4.3.4 Scalability of Operation Availability

We determine how read and write availability changes when increasing the system size while keeping the access ratio constant. In Figure 4.2 we show the operational availability of the d-space approach for access ratio $\rho_{wr} = 9$, and number of nodes $N = 81$, $N = 729$, and $N = 6,561$. The read quorum sizes that correspond to the given access ratio and system sizes are 3, 9 and 27, while the write quorum sizes are $27 + 2$, $81 + 8$ and $243 + 26$. The availability of read operations does not scale well. Surprisingly, the update availability can improve with increasing system sizes (see $N = 729$ vs. $N = 81$). Further, while for $N = 81$ the read and write availabilities

[2] Write availability for ROWA: $\alpha^W_{ROWA} = p^N$. Write availability for weighted voting assuming one vote per node: $\alpha^W_{VOT} = \sum_{m=v_w}^{N} b(N, m, p)$.

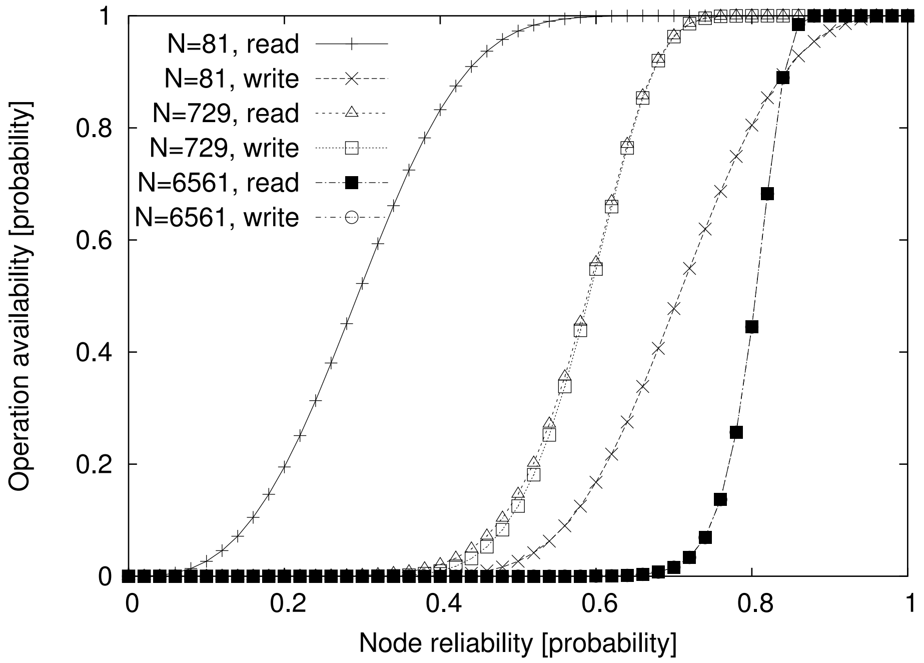

Figure 4.2: Scalability of read and write availability for constant access ratios and varying system size.

are substantially different, for $N = 729$ the difference is barely noticeable, and for $N = 6,561$ they are virtually the same (note that write availability is in no instance better than read availability which is what we expected).

What changes dramatically from one system size to another is the read availability, and for larger number of nodes it becomes the bottleneck (and hence matches the write availability due to the read component of the write quorum groups). Thus, a higher degree of replication (N) can have the paradoxical effect of actually making data less available. The key point to make is that we kept the read/write access ratio constant. The availability of data can be maintained at the same level if we increase the access ratio when scaling up the system. We exemplify this in the next section where we keep the read quorum size constant, and increase the access ratio in proportion to the number of nodes.

It is arguable whether or not existing systems that have higher degrees of replication also exhibit higher access ratios. Note however that keeping the access ratio constant and increasing N, increases the read quorum group size proportionally. This may pose efficiency problems if a substantial number of replicas have to be contacted for every read operation. Thus, at least from an efficiency standpoint it may make sense to assume that a higher degree of replication implies a higher access ratio, in which case the scalability of read availability is not an issue. It is not clear at this point how the access ratio should vary with N such that the a given availability level is preserved. Stated differently, it is not clear how the degree of replication should vary with the access ratio given the operation availability.

4.3.5 D-Spaces vs. Hierarchical Quorum Consensus

We compare the communication complexity and operation availability of d-spaces (referred to as DSP) and the hierarchical quorum consensus (referred to as HQC) methods. We examine both symmetrical scenarios, where the access ratio is close to 1, and skewed scenarios, where read operations dominate accesses to data. Hierarchical quorum consensus is both relatively efficient (in terms of communication complexity), and can naturally model arbitrary access ratios. For these reasons we have chosen it to contrast the performance and availability of the d-space approach. The results shown correspond to formulas derived in the previous sections.

Hierarchical quorum consensus is optimal when each logical group is decomposed in three subgroups, and the resulting hierarchy is perfectly balanced. We choose the number of nodes with respect to such criteria, i.e. $N = 3^m$ ($l_i = 3$). The quorum intersection condition is satisfied if (i) $r_i + w_i > 3$, and (ii) $2w_i > 3$. Possible com-

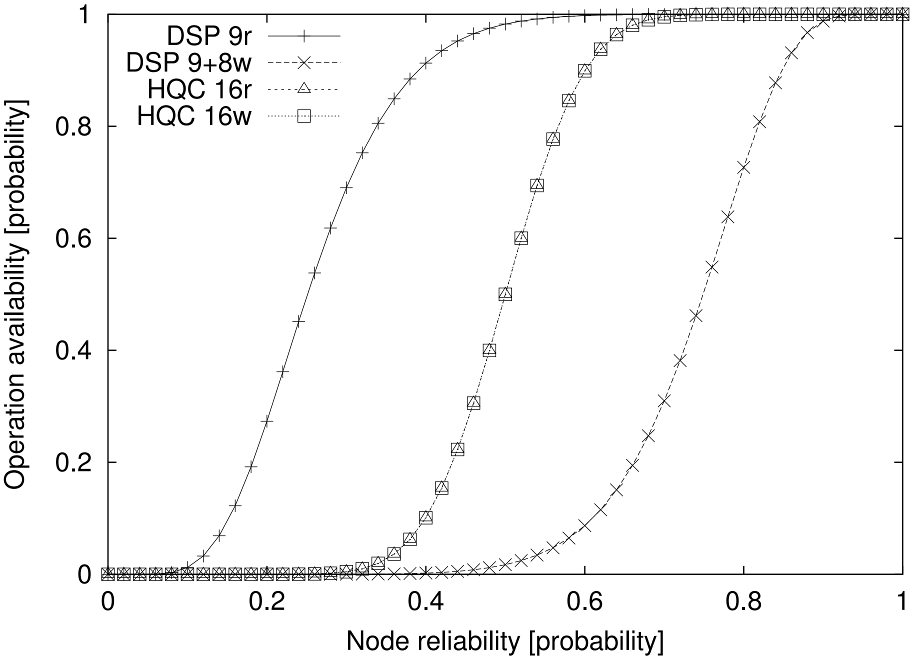

Figure 4.3: Operation availability for DSP and HQC with 81 nodes and targeted $\rho_{wr} = 1$.

binations for establishing appropriate quorums at each node are $(r_i = 1, w_i = 3)$ and $(r_i = 2, w_i = 2)$. Thus, a read quorum will contain $1^t 2^{m-t}$ sites, and a write quorum will contain $3^t 2^{m-t}$ sites, for some t such that $0 \leq t \leq m$ and $3^t \approx \rho_{wr}$.

We distribute the quorums at each level such that operation availability is maximized. More exactly, we strive for optimal write availability since they are the critical component with respect to failure (i.e. consistently having lower availability than read operations). It is beyond the scope of this dissertation but it can be shown that the best write availability, given the constraints above, is achieved when the top levels have $w_i = 3$ ($1 \leq i \leq t$), and lower levels have $w_j = 2$ ($t < j \leq m$), and vice-versa.

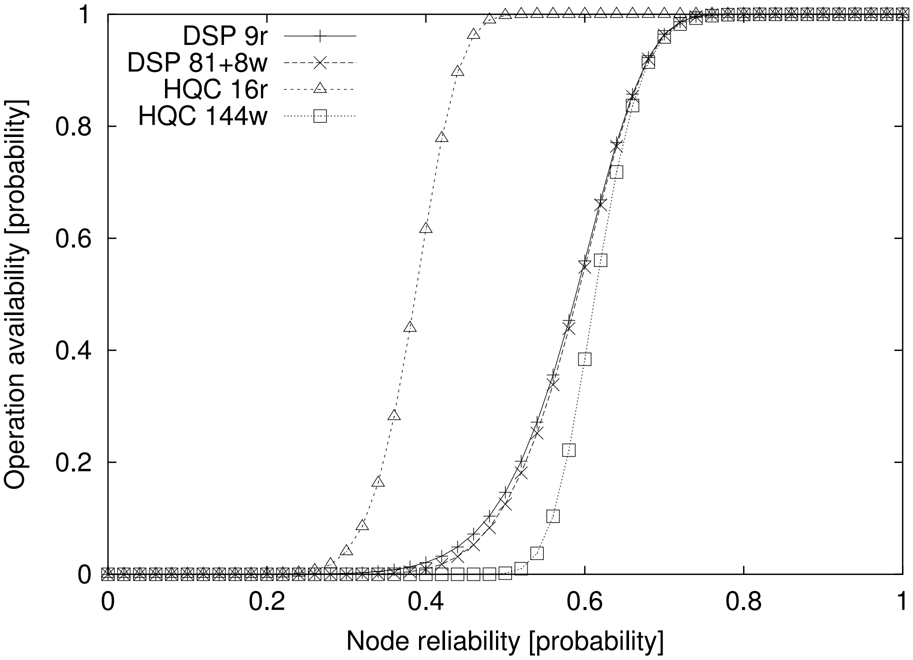

Figure 4.4: Operation availability for DSP and HQC with 729 nodes and targeted $\rho_{wr} = 9$.

Operational Availability

Figure 4.3 shows read and write availability as a function of site reliability for d-spaces and hierarchical quorum consensus. 81 nodes are considered, arranged in a 3^4 hierarchy for HQC, and a 9×9 2-dimensional space for DSP. This arrangement captures balanced read and write access frequencies ($\rho_{wr} = 1$). In this case, the read availability for d-spaces is derived using the Grid quorum definitions and given by (4.15). Read and write availability curves for HQC overlap due to the symmetry of read and write quorum definitions for $\rho_{wr} = 1$. Operation availability in d-spaces for a 9×9 space is similar to the Grid protocol and is highly asymmetrical.

In Figures 4.4– 4.6 we show the operational availability in HQC and DSP for increasing access ratios $\rho_{wr} = 9$, $\rho_{wr} = 81$, and $\rho_{wr} = 729$. In Figure 4.4, 729 nodes

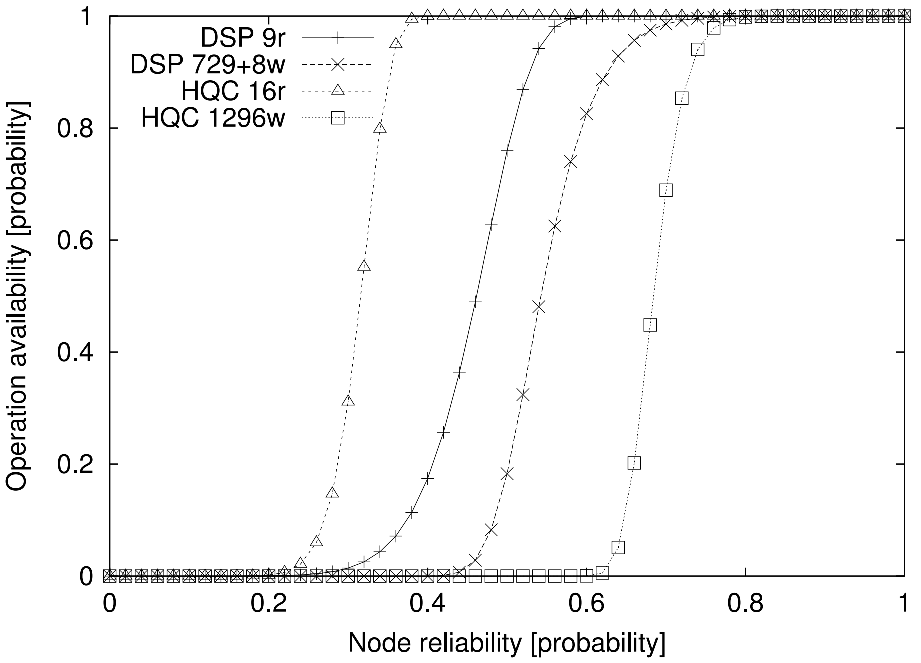

Figure 4.5: Operation availability for DSP and HQC with $6,561$ nodes and targeted $\rho_{wr} = 81$.

are considered arranged in a 3^6 hierarchy for HQC, and a $9 \times 9 \times 9$ 3-dimensional space for DSP. In Figure 4.5, $6,561$ nodes are considered arranged in a 3^8 hierarchy for HQC, and a 4-dimensional space for DSP. Finally, in Figure 4.6, $59,049$ nodes are considered arranged in a 3^{10} hierarchy for HQC, and a 5-dimensional space for DSP.

Starting with $\rho_{wr} = 9$, HQC and DSP change places as far as availability symmetry is concerned. For $\rho_{wr} = 9$ DSP is almost perfectly balanced, for $\rho_{wr} = 81$ it is well balanced, and less balanced for $\rho_{wr} = 729$. HQC is consistently less balanced than DSP. Note however that, with the exception of Figure 4.4, the two methods are equivalent if we abstract over the degree of asymmetry. If one approach gains for read availability, it loses the same amount for write availability.

It seems that since reads are executed more frequently than updates, the actual

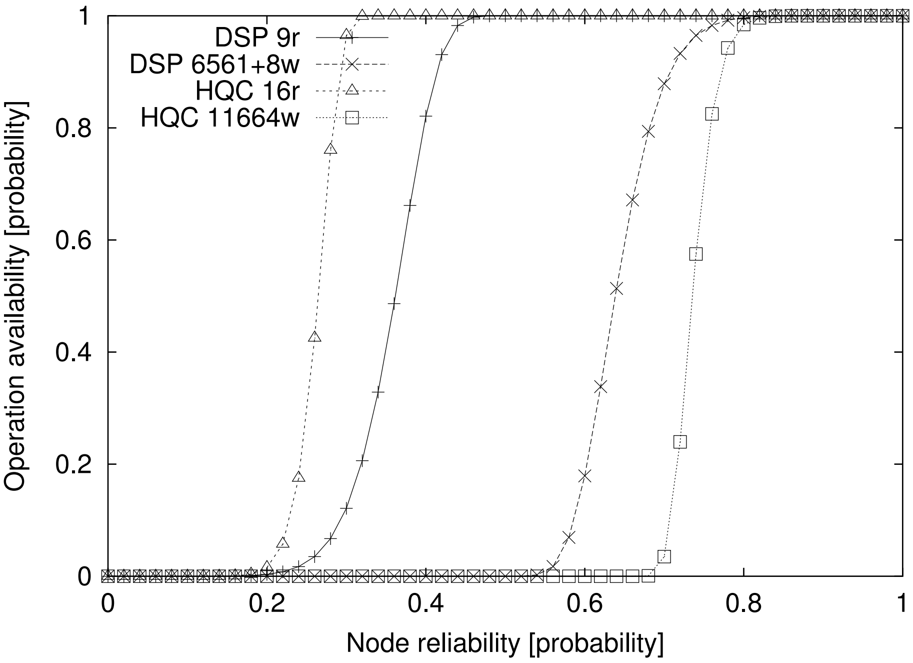

Figure 4.6: Operation availability for DSP and HQC with $59,049$ nodes and targeted $\rho_{wr} = 729$.

availability of updates could be better than for reads even if the write availability curve is to the right of the read curve (which is always the case for most existing protocols, including all those mentioned in this dissertation). This is not the case if we measure the degree of fault tolerance of a protocol in terms of the expected time until the first failure (i.e. inability to carry out a read or write operation). The overall availability of a protocol is determined by the rightmost availability curve, i.e. by either one of the read and write curves that defines poorer availability.

The claim is based on the observation that the sets of unavailable sites between two consecutive operations read (write) operations are not independent. First, we assumed that processors are fail-stop [SS83], and thus the set of unavailable servers when executing an operation draws from the set of unavailable servers for the previous operation. Further, even if we did not assume fail-stop processors, there is strong correlation between unavailable sites for consecutively executed operations. If this correlation is high between any two consecutive write operations (or read operations, for that matter), then the frequency of operations does not play a role in determining the expected time to failure. We believe that there is indeed high correlation between two reads or two updates, and that this correlation is diluted only when considering a very large sequence of operations.

Thus, read and write availability curves are directly comparable and we can abstract over ρ_{wr} for the purpose of establishing the degree of fault tolerance. The overall availability of a protocol is determined by the rightmost availability curve. This is always the write curve for both DSP and HQC. In Figures 4.4– 4.6 write availability is consistently poorer for HQC than for DSP. In conclusion, implementing a read-few write-many protocol ($\rho_{wr} \gg 1$) using d-spaces results in better operational availability than using the hierarchical quorum consensus method.

Communication Complexity

We argued that d-space quorum sets have optimal message complexity for any access ratio. We now show how communication costs for the hierarchical quorum consensus relate to d-spaces. A read in HQC has communication cost $1^t 2^{m-t}$, where $3^t = \rho_{wr}$ and $3^m = N$. The same ρ_{wr} is obtained in DSP by defining read quorum groups of size $\sqrt{N/\rho_{wr}} = \sqrt{3^{m-t}}$. The cost ratio for read operations in HQC versus DSP is given by:

$$\begin{aligned}\gamma^R &= \frac{2^{m-t}}{\sqrt{3^{m-t}}} = \left(\frac{N}{\rho_{wr}}\right)^{\log_9\left(\frac{4}{3}\right)} \\ &\approx \left(\frac{N}{\rho_{wr}}\right)^{0.13}\end{aligned} \tag{4.13}$$

When $\rho_{wr} = 1$, the ratio becomes $N^{0.63}/\sqrt{N}$ which is exactly the ratio of quorum

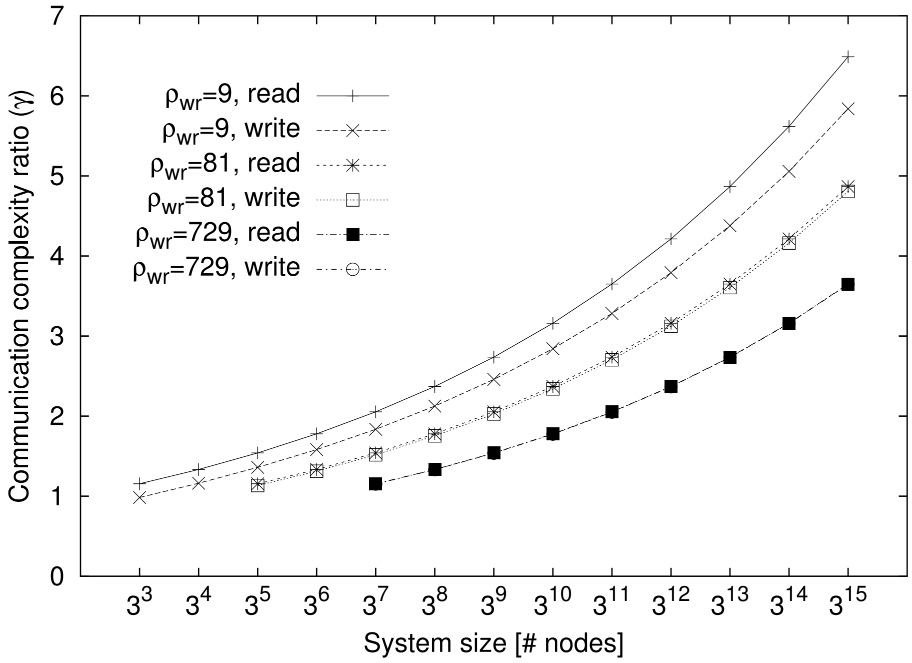

Figure 4.7: Communication complexity ratio (HQC/DSP) for read and write operations as function of system size and access ratio.

sizes for HQC and DSP in the symmetric case. Starting with write quorum sizes in HQC ($3^t 2^{m-t}$), and DSP ($\sqrt{N/\rho_{wr}} + \sqrt{N\rho_{wr}} - 1 \simeq \sqrt{3^{m-t}} + \sqrt{3^{m+t}}$), the cost ratio for write operations is:

$$\begin{aligned} \gamma^W &= \frac{3^t 2^{m-t}}{\sqrt{3^{m-t}} + \sqrt{3^{m+t}}} = \frac{3^t}{3^t+1}\left(\frac{N}{\rho_{wr}}\right)^{\log 9\left(\frac{4}{3}\right)} \\ &= \frac{\rho_{wr}}{\rho_{wr}+1}\gamma^R \end{aligned} \tag{4.14}$$

For Figures 4.3– 4.6, $N/\rho_{wr} = 81$ and thus $\gamma^R = 1.78$, while γ^W takes values 0.94, 1.62, 1.76 and 1.78, depending on the value of ρ_{wr}. Note that for a given ρ_{wr}, the ratio of communication costs for both read and write operation grows with N. In Figure 4.7 we show how γ^R and γ^W vary with system size for some of the used access ratios (9, 81, and 729).

Implementing RFWM using d-spaces results in a message complexity 2–3 times lower than implementing it using the hierarchical quorum consensus. Beside saving network resources this also materializes in better load sharing at sites holding copies, and increased system throughput. The expected increase in system throughput is proportional to γ^R for read operations, and γ^W for write operations.

4.3.6 D-Spaces and Grids

The d-space protocol is a generalization of Grid to multiple dimensions. It is also reciprocal to Grid in the sense that it defines inverted quorum groups with respect to subspace covers.

The availability of read operations in Grid is different than in d-space even when considering a 2-dimensional replica space for which $N^{\frac{k}{d}} = N^{\frac{d-k}{d}}$. The Grid protocol targets symmetrical scenarios for which the expected frequency of read and write operations is approximately the same ($\rho_{wr} \approx 1$). A write in Grid is performed similarly to 2-spaces, by locking a column and a cover of all columns. However, a read is performed by locking the cover of all columns, as opposed to locking a column. The Grid read availability is thus greatly improved and is given by [CAA92]:

$$\alpha^R_{Grid} = (1 - (1 - p)^{N^{\frac{d-k}{d}}})^{N^{\frac{k}{d}}} \tag{4.15}$$

Subspace covers in d-spaces are defined inversely to Grid to accommodate highly skewed access ratios. In the spirit of Grid, we could have defined a write quorum to be a hyper-plane and the cover of a line. In this case a read quorum group would be itself the cover of a line and would have excellent availability (4.15). However, the availability of write operations would be adversely affected to the point of making it

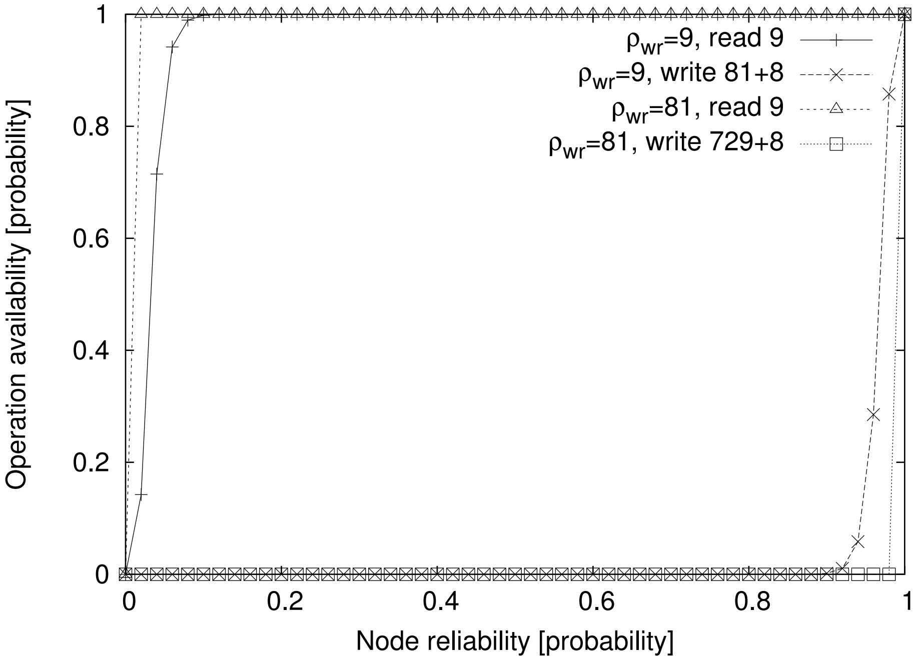

Figure 4.8: Operation availability in Grid for asymmetric scenarios ($\rho_{wr} = 9$ and $\rho_{wr} = 81$).

not substantially better than in the ROWA protocol:

$$\alpha^{W}_{Grid} = (1-(1-p)^{N^{\frac{d-k}{d}}})^{N^{\frac{k}{d}}} - (1-p^{N^{\frac{d-k}{d}}} - (1-p)^{N^{\frac{d-k}{d}}})^{N^{\frac{k}{d}}} \tag{4.16}$$

For the 3-space example in Figure 4.1, write availability as given by Grid would translate to having one of the three planes parallel to the base of the cube fully operational. We further make the case for this argument in Figure 4.8, where we show the read and write availability in Grid for access ratios $\rho_{wr} = 9$ (729 nodes) and $\rho_{wr} = 81$ ($6,561$ nodes). While reads have good availability, updates have very poor availability even when nodes are reliable. For $\rho_{wr} = 81$ the protocol is close to tolerating no failures whatsoever when executing updates. For $\rho_{wr} = 729$ (not shown in the figure) the protocol literally tolerates no failures. The corresponding configurations are illustrated for the d-space approach in Figures 4.4 and 4.5.

In conclusion, if the difference between read and write quorum sizes is substantial, quorum groups should be defined using the d-space approach, otherwise they should be defined using the Grid approach. Note that for the sole purpose of targeting high access ratios, higher dimensional spaces are not a requirement. The same effect can be obtained with a Grid-like 2-dimensional space, as long as quorums are defined using the d-space method. In Section 4.4.3 we discuss this more fully.

4.4 Local Quorum Reconfiguration Using Virtual Replicas

The availability analysis carried out in Section 4.3 establishes the likelihood that executing read and write operations will commit by assembling all the nodes in a quorum group. A client executing a read or write operation must assemble one quorum group from the operation's quorum set. If no quorum group can be assembled, the corresponding operation is blocked and the client that issued the operation will not be able to make further progress. Mechanisms are needed to reconfigure the d-space structure when quorum groups can no longer be assembled.

In dynamic voting [JM90], quorum readjustment is performed within the protocol for write operations. Other approaches such as the virtual partition algorithm [AT89], and the epoch protocol [RL92, RL93] check for changes in system topology or even reconfigure it as a separate transaction, asynchronously with regard to read and write operations. The epoch protocol can readily be applied to globally reconfigure d-space quorums.

Node space virtualization provides a local alternative to global reconfiguration protocols. The method defines the node space independent of the set of sites holding the physical copies. Local information is combined with a best-effort form of global views to achieve seemingly contradicting goals: lightweight local reconfigurations and good global latencies. In this section we discuss how local reconfiguration can be performed on logical structures, and address related issues such as replica recovery, and membership changes.

4.4.1 Global View vs. Local Information

The fact that logically-structured quorum approaches are not accommodating to mutations (reconfiguring the space, adding or removing nodes individually or in group) is inherent to its global view of a logical structure. Flexibility is sacrificed for the sake of efficiency. At the other extreme, similar structures are maintained by CAN [RFH$^+$01] to route among participants in peer-to-peer networks. CAN lacks a global view and uses local information (each sites knows its $2d$ immediate neighbors) to arrange the sites in a d-dimensional torus.

Reconfiguration can be localized if only local information is used to route packets. However latency is hurt as an isolated remote contact requires $O(dN^{\frac{1}{d}})$ incremental hops toward the destination. The latency to access a quorum group with local information is given by the spatial diameter of the space defining the group, i.e. $N^{\frac{1}{d}}$ for reads and $N^{\frac{d-1}{d}}$ for writes. Even though quorum sizes are the same irrespective of implementation choice (global view or local information), the high latency of the latter makes it prohibitive as a support mechanism for implementing structured quorums. Further, the message complexity of the local information approach may increase theoretically by a factor of up to the diameter of the space. We propose a solution that combines the advantages of global views and local information.

4.4.2 Mapping the Replica Space

Though we have heretofore assumed that they were the same, there are advantages to making a distinction between the replica (node) space, and the set of sites (servers) hosting the physical copies. Which replicas are hosted by servers is given by the mapping procedure. A server can host more than one replica; a replica is hosted by exactly one server. Replicas form a static and virtual space in the sense that there is no one-to-one correspondence between nodes and physical copies of a data item. Instead, all nodes mapped to a site correspond to one physical copy. Servers form a dynamic and unstructured space. Thus, the server space is the same as the space of physical copies, and there is a many-to-one hosting relationship among nodes and servers.

Data values and version numbers of replicas mapped to a server are automatically kept consistent at all times. Despite the fact that site failures may be independent, node failures are not due to the range-mapping of nodes to sites. Either all nodes mapped to a site are operational, or none of them. However, if the number of nodes per site is approximately the same, and the zone of nodes mapped to any site forms a regular subspace, then the availability and communication complexity analysis of Section 4.3 remains valid (with the caveat that they refer to the availability and complexity in the server space, not the virtual replica space).

Assume that the replica space is very large, such that the number of sites will never match the number of nodes, and will feature high dimensionality. In this dissertation we chose the number of nodes to be $N = 2^{16}$, fixed ($d = 16$). The results remain qualitatively unchanged for larger replica spaces (e.g. $N = 2^{32}$), and there are no scalability issues involved with higher N values. With $N = 2^d$, the d-space becomes a hypercube. Each node has exactly d neighbors in the replica space, and the shortest path between any two nodes is bounded by d. Assuming that each site hosts a contiguous

Table 4.1: Distribution of number of nodes (zone size) to servers. $N = 2^{16}$.

Servers\Nodes	$N/4S$	$N/2S$	N/S	$2N/S$
$S = 2^8$	0.78%	12.89%	$\mathbf{79.30}\%$	7.03%
$S = 2^{10}$	0.39%	10.94%	$\mathbf{82.91}\%$	5.76%
$S = 2^{12}$	0.44%	9.40%	$\mathbf{85.13}\%$	5.03%
$S = 2^{14}$	0.35%	8.62%	$\mathbf{86.44}\%$	4.58%
$S = 2^8$	0%	7.81%	$\mathbf{88.28}\%$	3.91%
$S = 2^{10}$	0%	8.40%	$\mathbf{87.40}\%$	4.20%
$S = 2^{12}$	0%	7.37%	$\mathbf{88.94}\%$	3.69%
$S = 2^{14}$	0%	6.41%	$\mathbf{90.39}\%$	3.20%

area of the node space, the nodes mapped to a server can be described by a sequence of 0, 1, and X of length d. X is a free variable that can take values 0 and 1. Thus, each server hosts a number of nodes equal to some power of 2. For instance, the sequence $01101X011X01X011$ represents the subspace in a 16-dimensional space mapped to a server; it contains exactly 8 nodes. If a site hosts more than one contiguous zone, then the nodes mapped to it can be described by a set of such sequences, and each site will host a number of nodes equal to some sum of powers of 2.

The node space is equitably divided and mapped onto the sites such that each site holds a set contiguous subspaces (range mapping). This is similar to CAN, and we advocate a similar protocol for performing zone-reassignment in the background to prevent uneven fragmentation of the node space. Given the number of sites in the system S, the assignment protocol can perform an almost perfectly balanced mapping of nodes to sites: 90% of sites will hold N/S nodes, while the rest will hold half or twice of that [RFH$^+$01]. More recently analytical bounds have been proven to this effect [AHKV03]. We reinforce the argument in Table 4.1 where we show two examples of the distribution of the number of nodes to servers for various system sizes (S). Note that even though the mapping procedure virtualizes the node space, our approach is different than virtualized namespaces (in particular CAN) in that we are not using a distributed hash table. The association between replicas and sites is direct, and arguments presented in Chapter 3 regarding the advantages and disadvantages of namespace virtualization do not hold in this case.

4.4.3 Membership Changes in the Server Space

A site has as many neighboring sites as defined by the mapping of nodes to sites. Two sites are neighboring each other in the server space if and only if they host at least two nodes neighboring in the replica space. Two nodes are neighbors in the replica space if their representation using sequences of 0 and 1 differ in exactly one position. When sites join or leave the structure, we use a protocol similar to CAN's for routing through the grid, and for splitting and merging the zones (subspaces) assigned to affected sites. A virtual node space gives us the flexibility to make localized adjustments upon joining and leaving.

Joining the Network

A new server joining the network must contact one of the participating servers. Subsequently, routing on the hypercube is performed to a random point in the node space mapped to some site S. The joining server identifies among S and its neighbors the server that maps the largest node subspace — each site knows what zones are mapped to its neighboring sites. This subspace is equally divided between the selected server and the joining server. This procedure ensures good distributions of replica volumes to participating servers (see Table 4.1).

Splitting a node area in two equal areas is performed by instantiating one of the free variables (X) and assigning it to 0 on the original server, and to 1 on the new server. In Figure 4.9 we show two examples of the distribution of neighbors per server as a function of system size. Every time a free variable needs to be instantiated it is chosen at random among the available choices. The number of neighbors increases with system size, and then decreases as the number of nodes mapped to a server approaches 1.

We substantially decrease the number of neighbors per server by establishing a priori an order in which the dimensions are instantiated. The order itself can be random, but it must be consistently observed whenever an instantiation (zone splitting) is to be performed. The average number of neighbors drops 2-3 times in this case over random instantiation (see Figure 4.10 vs. Figure 4.9). Splitting subspaces in a consistent manner increases the chance of zones hosted by neighboring servers to be properly aligned (e.g. two parallel lines in a cube). The number of zones that a zone touches is reduced, and so is the number of neighboring servers of a server.

Upon joining no data transfer is necessary. The newly joined site will simply initialize the version numbers for its assigned data items to null values.

Leaving the Network

A server leaving the network selects some of its neighboring servers and hands over its subspace to the chosen servers. The leaving server may have to split its zone to accommodate the needs of the receiving servers. The receiving servers are selected such that: (i) each receiving server must be able to join its currently hosted zone with the received zone, and (ii) all components of the zone split must be transferred to

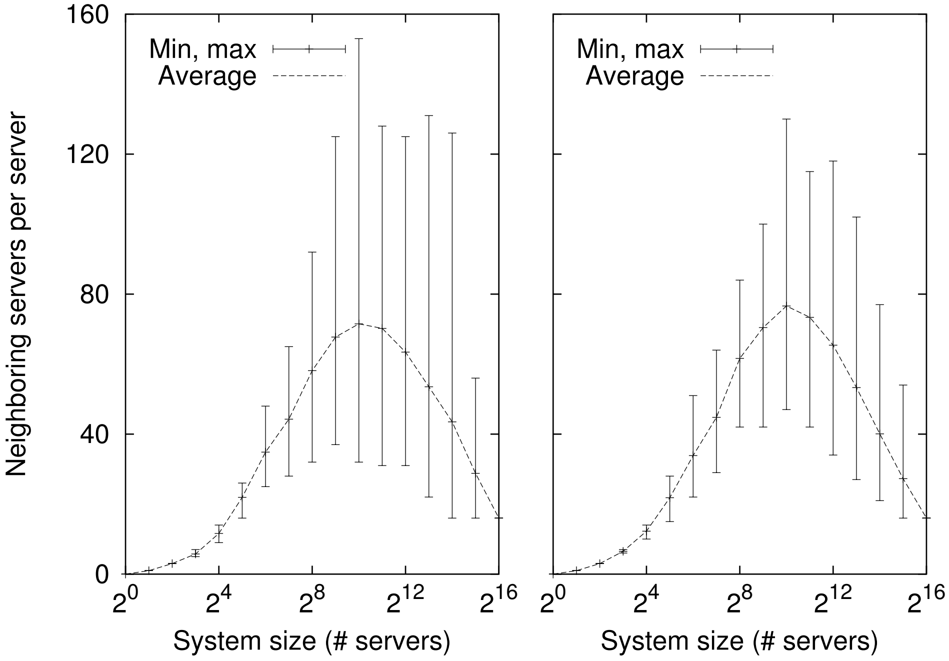

Figure 4.9: Distribution of neighbors (server space) using random zone splitting.

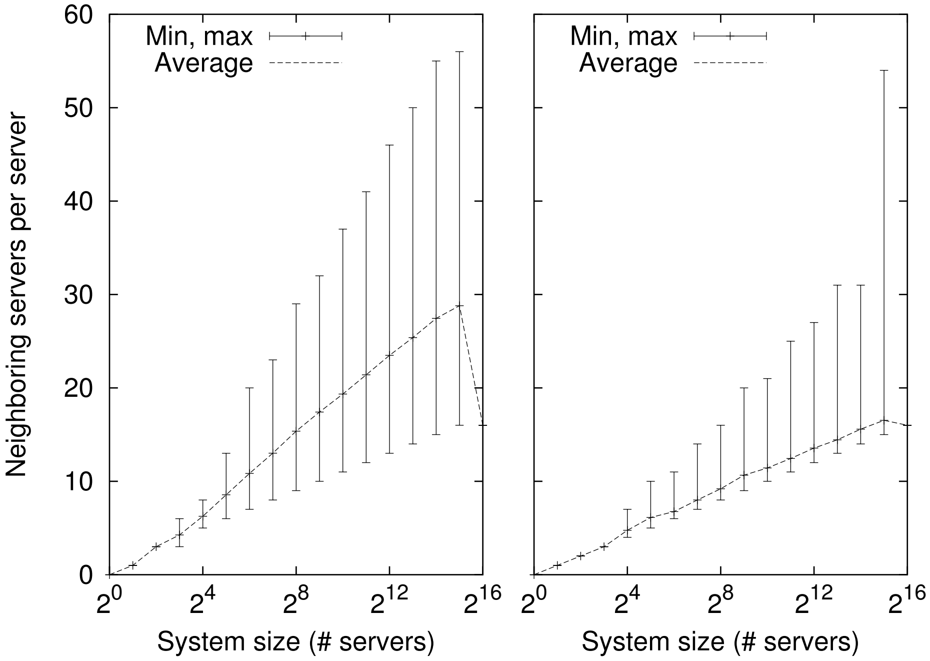

Figure 4.10: Distribution of neighbors (server space) using ordered zone splitting.

receiving servers.

It may not be possible at all times to perform a leave given the above conditions, if the leaving server could not find a suitable combination of neighbors to take over its subspace. In such cases, a receiving server will have to temporarily host more than one zone. A zone reassignment protocol is run in the background to exchange and merge zones among servers, in effect making such conditions transient.

Note that our zone transfer protocol is different than in CAN, where the zone of the leaving server is always taken over by one receiving server. Our extension to the protocol decreases substantially the chance of having servers temporarily host more than one zone, and thus relies to a lesser extent on the zone reassignment protocol. Upon leaving, data transfer is necessary only if the recipient's versions are smaller than the leaving site's versions.

In conclusion, even though the replica space is static, the mapping of nodes to sites enables arbitrary mutations of the network through purely local reconfigurations.

The Case for Multi-Dimensional D-Spaces

Given the mapping of replicas to sites, the choice for multi-dimensional spaces (as opposed to 2-dimensional Grids) becomes clear:

- Routing latency in a CAN-like network improves with higher dimensional spaces. In the node space, latency is bounded by 16 for a hypercube as opposed to 256 for a 2-space (for $N = 2^{16}$), or 32 as opposed to $65,536$ (for $N = 2^{32}$). If the 2-space has a rectangular shape (to accommodate high access ratios), then the bounding constant becomes even higher.

- Fault tolerance (along with the amount of state maintained) increases with the number of neighbors. More neighboring sites enables a server to remain in the network despite a considerable number of sites around it having failed. For a 2-space each node has 4 neighbors which translates to at most 4 neighbors in the server space. We show in Figure 4.10 examples of neighbor distributions in the server space.

- Replica spaces of the form $N = 2^d$ easily accommodate through namespace factorization any access ratio within a factor of 2. More importantly, the access ratio can be adjusted on the fly in response to varying read/write access patterns. The adjustment will not affect the neighboring relationship in the node or server space. Thus, with changing access ratios only the quorum groups need to be redefined. The replica space, the mapping of nodes to servers, and a node's or site's neighbors remain the same.

4.4.4 Mapping Goodness

Replica operations (read, write) are expressed in terms of the node space. The mapping procedure specifies how replica operations translate in the server space. The *mapping goodness* indicates how quorum sizes for read and write operations in the node space and server space relate to each other. Given the system size (S) and the read/write access ratio (ρ_{wr}), the read and write quorum sizes in the node space are $R_N = \sqrt{N/\rho_{wr}}$ and $W_N = \sqrt{N\rho_{wr}} + \sqrt{N/\rho_{wr}} - 1$, while the *expected* (ideal) read and write quorum sizes in the server space are $R_S = \sqrt{S/\rho_{wr}}$ and $W_S = \sqrt{S\rho_{wr}} + \sqrt{S/\rho_{wr}} - 1$. The actual quorum sizes in server space are given by the mapping of R_N and W_N and depend on the goodness of mapping. The goodness of mapping for read quorums (γ_R)

and write quorums (γ_W) is thus defined as follows:

$$\gamma_R = \frac{\mathcal{M}(R_N)}{R_S} \tag{4.17}$$

$$\gamma_W = \frac{\mathcal{M}(W_N)}{W_S} \tag{4.18}$$

where $\mathcal{M}$ symbolizes the mapping in effect.

In Figure 4.11 we present two examples of mapping goodness for reads and writes, and access ratios $\rho_{wr} = 4$ and $\rho_{wr} = 16$. In Figure 4.12 we show the same data for access ratios $\rho_{wr} = 64$ and $\rho_{wr} = 256$ (we are considering only access ratios that are powers of 2 and whose square root is an integer). R'_S and W'_S are analytically derived while $\mathcal{M}(R_N)$ and $\mathcal{M}(W_N)$ are averages and represent the actual quorum sizes reflected through mapping. For instance, $\mathcal{M}(R_N)$ is derived by considering for each read quorum the number of servers involved, and computing an average over all read quorums.

A mapping goodness of 1 is ideal, while the range 0.5–2 is acceptable. 2 (0.5) means that the number of servers involved in the corresponding operation is twice (half) as much as it could be under ideal circumstances. The mapping goodness is the first indicator that the performance of the protocol (in terms of communication complexity, i.e. quorum group sizes) is preserved through mapping. In all cases presented the mapping goodness is within the range 0.5–2. Note that when it is larger than 1 for read operations, it is correspondingly (inversely) smaller than 1 for write operations, and vice-versa.

We showed in Section 4.4.3 that enforcing a consistent order for instantiating free dimensions throughout all join events decreases the average number of neighbors per server. We now describe what that order should be. Given a string with d positions (0, 1, and X), without loss of generality we can assume that read quorums correspond to free variables on the right side, and the write component of write quorums to free

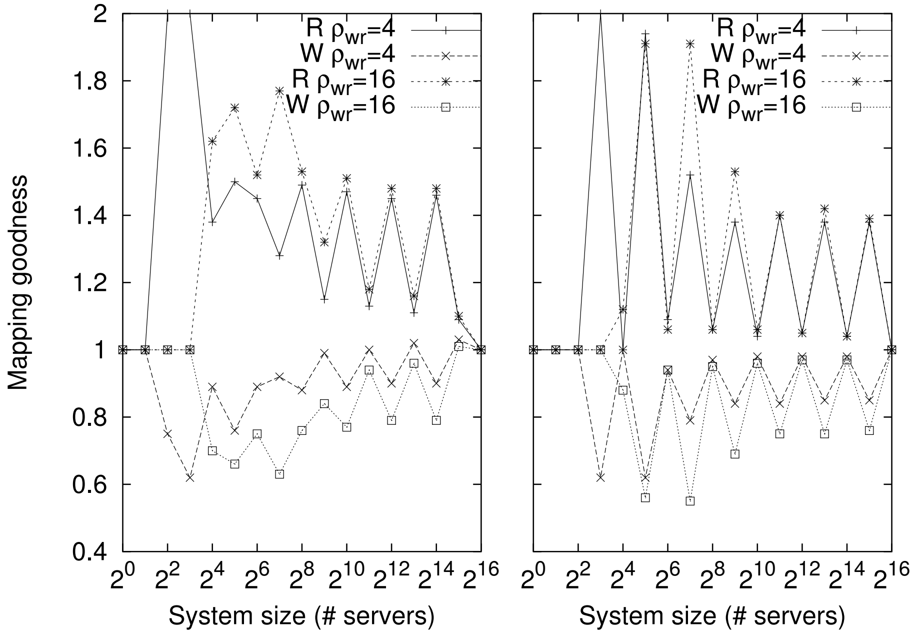

Figure 4.11: Mapping goodness as a function of system size, type of operation and access ratio ($\rho_{wr} = 4$ and $\rho_{wr} = 16$).

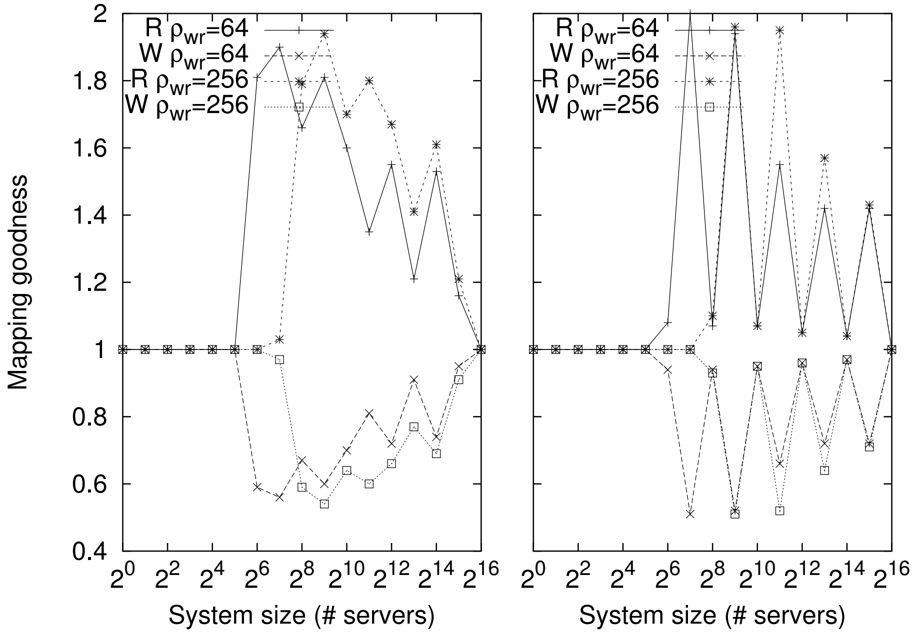

Figure 4.12: Mapping goodness as a function of system size, type of operation and access ratio ($\rho_{wr} = 64$ and $\rho_{wr} = 256$).

variables on the left side. For instance, if $d = 16$ and 6 positions are used to describe read quorums (2^6 nodes per group), and 10 positions are used to describe the write component of write quorums (2^{10} nodes per group), then $0100101110X^6$ could represent one of the read quorums, and $X^{10}011001$ could represent the write component for one of the write quorums. The cover for a node in the write component is any node that will preserve the quorum intersection property when replacing the original node with its cover. The candidate cover set for a node has the same representation as a read quorum. For instance, taking node 001011101101101 in write component $X^{10}101101$, the candidate cover set is given by $0010111011X^6$. Any node in $0010111011X^6$ can act as a cover node for 001011101101101.

The order of instantiating free variables is as follows. Assume that we have r free variables describing a read quorum, and w free variables describing the write component of a write quorum. We have that $r + w = 16$, and we satisfy the read/write access ratio by having $2^w/2^r = \rho_{wr}$. r and w are determined given these relations. In the node space we are able to accommodate the targeted access ratio within a factor of 2. We want to make sure that we will satisfy the access ratio in the physical space as well.

We identify the position that splits X^{16} into $X^w X^r$. Initially, the first server hosts the entire virtual space X^{16}. When the second server joins, we instantiate one bit such that the original server hosts $X^{w-1}0X^r$. Note that we are assuming that $w > r$. Also, note that if we have less than ρ_{wr} servers in the network, we cannot possibly satisfy the access ratio. When the third server joins, we further split (let's say) $X^{w-1}0X^r$ into $X^{w-2}00X^r$ and $X^{w-2}10X^r$. We keep instantiating variables to the left of the split position until we have subspace of the form $X^r0...1X^r$, where $0...1$ denotes an arbitrary instantiation of free variables (of length $w - r$). At this point we are finally starting to satisfy the access ratio. From now on, we instantiate in both directions. For instance, after more servers join the network we would (ideally) have subspaces of the form $X^{r-1}0...1X^{r-1}$ mapped to servers, and so on.

The procedure described above together with the observation that we have relatively equal volumes of virtual replicas (subspaces) mapped to servers, enable us to satisfy the access ratio in the server space as well. The procedure also explains why the goodness of mapping favors read and write quorums in turn, and has a jagged graphical representation (see Figures 4.11 and 4.12). The availability formulas were derived in Section 4.3 with respect to independent node failures. Given the mapping of nodes to sites, the availability analysis holds in the server (site) space, and not in the node space (where we have totally correlated failures for all nodes mapped to the same server). [3] Therefore, the availability analysis still holds when making a distinction between virtual replicas and physical replicas, except that it is in terms of the server space and not the virtual replica space (and thus N is the number of servers for the purpose of establishing the availability).

4.4.5 Fault Tolerance

Virtualizing the node space enables sites to join and leave the structure without invalidating existing quorums (by creating holes in the structure), or requiring a global reconfiguration mechanism. Faulty sites that do not recover (or recover too late) can also be eliminated through local reconfiguration. We informally describe a failure

[3] One way to interpret this is that the mapping of the d-space to the server space makes the server space look like a d-space, with the caveat that it is not a regular d-space. It were regular if the number of servers were 2^k at all times, and if we could somehow have perfectly equal volumes of nodes assigned to servers.

detection protocol that ensures the proper transfer of zones and locks from the site declared failed to one of its operational neighbors.

They key of the protocol lies in requiring that a site declares itself failed (upon recovery) before any of its neighbors do so. This ensures that when the site is declared failed by its neighbors and evicted from the structure, no client holds the site's locks (assuming additional constraints on how long a lock can be held). We assume the following timeouts are in effect. τ_{access} is the timeout of an individual read or write access. A lock grab/release request will timeout after τ_{access}, and the requester will try to assemble other quorum groups. τ_{oper} is the timeout of a read or write operation. If a site has granted its lock to a requester and the lock hasn't been released in τ_{oper}, it is by default considered to be released. The site is available to perform the next read or write operation.

Each site exchanges periodic heartbeat messages with all its neighbors. We assume that incommunicado states are symmetric: sites can either contact each other both ways or none. A site declares itself failed after not hearing from any of its neighbors for at least τ_{self}. The period for heartbeat messages should be much smaller than τ_{self} to allow for a few exchanges to occur within τ_{self}. A site tentatively declares its neighbor failed after not hearing from it for at least τ_{fail}. When a site declares one of its neighbors failed, it will contact other neighbors of the failed site to concur on the decision, and evict the failed site from the structure if necessary.

We have the following timing constraints: $\tau_{access} \ll \tau_{oper} \ll \tau_{self} < \tau_{fail}$. We have that $\tau_{access} \ll \tau_{oper}$ to give enough time the initiator to complete an operation while allowing it to timeout on individual accesses. Naturally, $\tau_{oper} \ll \tau_{self}$ since we want to avoid blocking on operations for the whole length of time that a transient failure is allowed. Finally, we have that $\tau_{self} < \tau_{fail}$ because we want to ensure that a site will declare itself failed before its neighbors will. τ_{fail} can be chosen to be $2\tau_{self}$.

If a site fails and recovers before τ_{self} elapses, the failure is considered transient. With quorum consensus schemes transient failures require no special treatment. A recovering copy that missed writes, when included in a quorum group will have its version number smaller than some other copy in the same group that took part in the intervening writes. Thus, its value will not be read until written. The only condition that breaks the invariant above is if a whole quorum group fails, misses some writes, and then recovers. This is an impossible condition since (i) either no writes could have taken place due to the failed quorum group, or (ii) if writes took place, the recovering quorum must have been evicted from the structure through reconfiguration.

If a site fails and recovers after τ_{self} elapses, or does not recover at all, the failure is considered permanent. Sites that fail permanently and recover subsequently must join the structure anew before further processing requests. Since $\tau_{oper} \ll \tau_{self}$, we are guaranteed that the lock of a permanently failing site has been released by the time the site declares itself failed. Since $\tau_{self} < \tau_{fail}$, we are guaranteed that the lock has been released by the time the site is evicted from the structure by its neighbors. Thus, the lock can safely be transferred to the neighbor that will take over the failing site's subspace. The data and associated versions hosted by the failing site are lost and cannot be recovered. To ensure consistency, the neighbor that takes over the subspace and the lock will have to perform read operations for the reassigned data items.

4.4.6 Caching Quorum Groups

The resulting system uses caching to match the low latencies achieved with global views. Every site caches a write quorum. Since a write quorum includes a read quorum (the line for RFWM), sites implicitly cache a read quorum as well. Caching a quorum

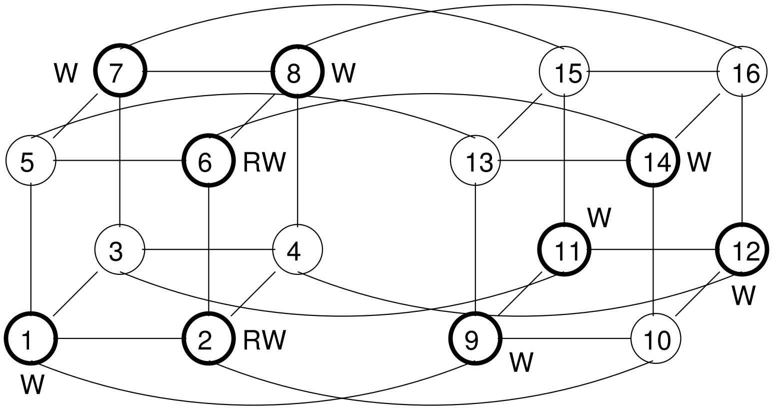

Figure 4.13: Example of cached read and write quorum groups for a 4-dimensional hypercube. Nodes in the write quorum are annotated with W and nodes in both quorums with RW.

translates to caching a list of mappings from subspace ranges to site identities (a site's identity is the tuple of its network address and port number). When a read or write request arrives at a site, the corresponding quorum cached at the site is used.

In Figure 4.13 we show a 4-dimensional space with 16 nodes, and examples of read and write quorums cached at a server. The read quorum set consists of groups of the form $\{x, x+4\}$ where $x \in \{1,2,3,4,9,10,11,12\}$; for the example shown in the figure, nodes cached for the read quorum ($\{2,6\}$) are annotated with RW. The write quorum groups are constructed by combining one of the read quorum groups and one of the following two write components: $\{1,2,3,4,9,10,11,12\}$ and $\{5,6,7,8,13,14,15,16\}$. In fact any cover of a write component will do. For the example shown, nodes cached for the write quorum are annotated with W and RW, depending on whether or not they are shared with the read quorum.

No cache replacement policies (e.g. LRU) are in effect since all cache entries are equal with respect to expected benefits. A cached mapping entry is either refreshed or replaced when it becomes stale. This can happen under the following circumstances:

- The server that hosts the node has changed because of some combination of servers leaving the network, joining the network, or permanent server failures. In this case, the node's new owner is identified through routing. If the owner can be identified the entry is refreshed, otherwise the node cannot be locked for the current operation and the entry is replaced.

- The server that hosts the node is temporarily unavailable and cannot be contacted (read and write locking attempts time out after τ_{access}). The unavailable server will either recover within τ_{self}, or otherwise will be evicted from the network. In this case the node cannot be locked for the current operation and the entry is replaced (even though the mapping may be up-to-date).

If a cached entry needs replacement upon performing a read operation the whole read quorum group must be replaced. This is because read quorums form a partition of the replica space. Similarly, if a cached entry that is part of the read component needs replacement upon performing a write operation, then the whole read component must be replaced. If a cached entry that is part of the write component needs replacement, then replacing it with a mapping that covers it suffices. This property is highly convenient since replacing the whole write component of a cached write quorum is expensive. For the write quorum shown in Figure 4.13 if node 1 is unavailable, then node 2 can be selected as an alternative without having to change any other nodes in the cached quorum group.

Staled entries are verified and refreshed by performing routing on the hypercube. More exactly routing is performed when the contacted server for a read or write operation

- has timed out on a lock request. Routing is performed because the owner of the node to be locked might have changed and the mapping cannot be confirmed directly.

- asserts that the mapping in question is invalid. Routing is performed because the server is hosted by a different server.

The maximum hop count for a routing request is equal to the dimensionality (d) of the node space. In practice, this is dramatically reduced since servers are mapped to contiguous node subspaces, and routing hops translate into jumps from one area to another. Further, with high probability every routing request can be initiated at a point very close to the destination node. For every node in a read or write quorum there are other nodes in the quorum group in its proximity. The routing can be initiated at the closest node whose mapping has been validated. For the example in Figure 4.13 if we assume that the destination node is 14 (its mapping needs to be validated/discovered), then routing can be initiated at node 6 which is one hop away, or one of nodes 2, 8, 9, 12 which are two hops away.

In Figures 4.14 and 4.15 we show how much latency and message overhead is to be expected when stressing the system with server failures. One million requests (read, write) were simulated, with a global mean request arrival rate of 222 requests per second. Each server fails an average of 15 times over the simulated time ($4,500$ seconds). All failures are temporary for the "Temp" lines. 1 in 6 failures are permanent for the "Perm" lines. Whenever a server fails permanently and is evicted, a new server joins the network such that system size is kept constant. Thus, the overall number of servers participating over a run is $3.5S$.

The message overhead is shown as a percentage of total messages. The overhead is incurred by performing routing and forwarding to refresh stale mapping entries and discover new quorums. The message overhead is in the lower percentages and does not

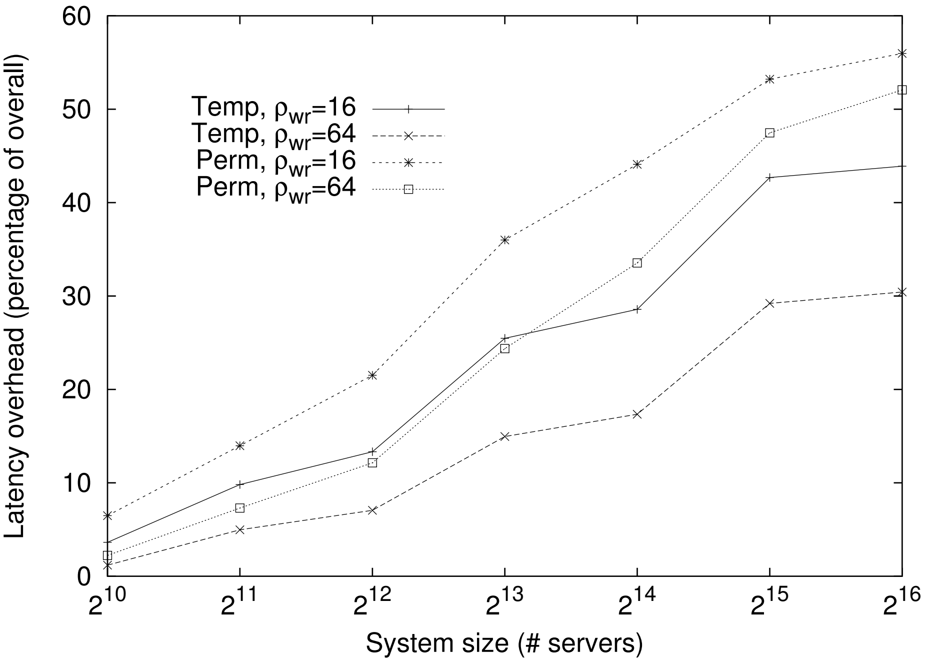

Figure 4.14: Latency overhead as a function of access ratio, failure mode and system size ($N = 2^{16}$).

increase with system size. Also, if changes in membership occur (joins and evictions), the message overhead is a few times higher than having only temporary failures.

The latency overhead is shown as a percentage of the total latency. It increases with system size and can reach up to more than half the total latency for large systems, which is to be expected given the high rate of server failures and membership changes. Most of the latency overhead is due to read operations or the read component of the write quorum. The latency overhead depicted is the individual operational latency averaged over all operations. It does not say what the client perceived latency is or how the request throughput is affected. Thus, it shows the worst case that would correspond to not having to wait for locks on reads or writes. Since clients have to wait for locks, depending on contention, much of the latency overhead could be hidden and amortized.

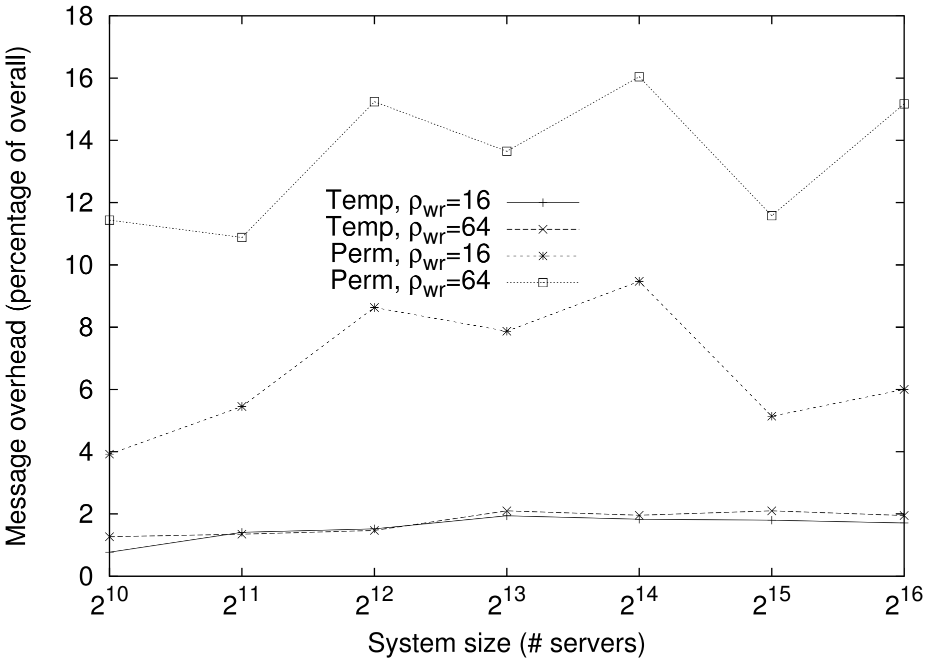

Figure 4.15: Message overhead as a function of access ratio, failure mode and system size ($N = 2^{16}$).

This is because mapping entries can be validated through routing, and requests can be forwarded to other servers, without having to hold any locks and while the network is satisfying previously queued requests.

4.5 Summary

We have described a method for defining replica quorum groups by arranging participating sites in logically structured multi-dimensional spaces. Our work is motivated by the commonly accepted observation that read accesses to data occur orders of magnitude more often than updates. We believe that read accesses will be a defining characteristic of usage in future distributed systems, including peer-to-peer architectures. Presently deployed file-sharing utilities fit this description, with the vast majority of data exported by participants unaltered after creation.

D-space is a generalization of the Grid protocol to multiple dimensions. It is also reciprocal to Grid in the sense that it defines inverted read and write quorum groups with respect to subspace covers. The central argument of our study is that d-space quorums offer a flexible way to build protocols with ideal balances of low communication complexity and high availability. First, d-spaces enable the more frequent read operations to execute efficiently, at a limited and controllable expense of more rarely executed write operations. This leads to our first result: for any given read/write access ratio, a d-space can be configured to give optimal communication complexity. Second, read operations can be performed efficiently without adversely affecting the availability of updates. To our knowledge, the quality of trade-off between read efficiency and update availability of our approach is not matched by existing quorum protocols. Surprisingly, for high access ratios the availability of updates can approximate or even match the availability of read operations. The read efficiency of the read-few write-many approach combined with its good write availability make it a good candidate for implementing replica control protocols for databases, file-systems and peer-to-peer networks.

Existing structured quorum schemes are based on global views. This allows good access latencies but hurts a protocol's adaptivity, as it must rely on heavyweight global reconfiguration mechanisms. We showed how to implement lightweight d-space reconfiguration through a combination of local information and a relaxed form of global views. The key to our approach is in making distinction between the virtual replica space and the server space, and maintaining the mapping between the two spaces as cached quorums at servers. Reconfiguration manifests in stale cached entries and the scheme's cost translates to refreshing such entries. We have shown that using virtual replicas preserves all the desirable properties of the original scheme, and that the cost of reconfiguration is within acceptable bounds even when the system is highly stressed with failures and membership changes.

Chapter 5

Conclusions

This dissertation describes two techniques for replicating information in peer-to-peer environments. First, we showed how routing state can be replicated on hierarchical namespaces such that both hierarchical bottlenecks and those triggered by hot-spots are addressed by the routing procedure. Second, we defined a replica control protocol on multi-dimensional spaces that ensures strict correctness for replicating application data. Both parts of our dissertation were studied and developed in the context of TerraDir, which is an overlay network defining hierarchical namespaces and search queries based on node attributes and partially-qualified node specifiers.

Our work has been motivated by the following observations regarding the target environment. First, only a small percentage of data looked-up or searched for is actually retrieved by users. The implication is that it makes sense to distinguish between processing load incurred by lookups and searches, and load due to the actual retrieval of data. Second, the vast majority of data exported by participants in peer-to-peer networks is unaltered after creation. Data reads account for almost all accesses to items exported by P2P services. Hence, replica control protocols should favor the execution of read operations without adversely affecting the availability of updates. Finally, membership changes occur on a scale not seen before in distributed systems. Active participation in P2P networks is highly transient with users/sites frequently joining and leaving the service. Thus, quorum reconfiguration for replica control protocols should be approached using local information, while global knowledge can be lazily inferred only to the extent needed.

We summarize the results of the dissertation in Section 5.0.1, and outline future research directions in Section 5.0.2.

5.0.1 Summary of Results

We have defined a replication model for hierarchical namespaces that replicates routing state to address both hierarchical bottlenecks as well as hot-spots due to temporal locality in the query stream. Central to the model is the node mapping, an association between a node's name and servers that host the node. Replicated state for a node consists of the node's mapping along with the node's routing context, i.e. mappings for the node's neighboring nodes. This guarantees incremental progress, and that a node's replica is functionally equivalent for routing purposes to the node itself.

Routing state is replicated in an ad-hoc manner, and the replication protocol can be combined with any data replication mechanism. Adaptivity is based upon server load information profiled online. This enables it to deliver low latencies and good load balance even when demand is heavily skewed. Further, it can withstand arbitrary and instantaneous changes in demand distribution. The approach is lightweight, scalable, and deals with soft-state in the sense that there is no need for replica consistency models to be specified. New replicas are created and existing replicas are deleted in a decentralized manner without enforcing notification or synchronization requirements.

We reconciled namespace consistency and mutations using prefix immutability and path checks. Prefix immutability is a node attribute that can be recursively granted or revoked, and precludes namespace mutations from occurring in select portions of the namespace. Path checks is a low cost mechanism that defines eventual consistency of namespace mutations through a window of vulnerability during which queries may be resolved incorrectly. We extended path checks to account for caching and replication.

We have defined a method of resolving complex search queries through hierarchical query decomposition and result recomposition. The novelty of our approach stems from a token distribution mechanism that enables the query initiator to ascertain query completion. The scheme is flexible and offers a broad range of trade-offs between network bandwidth consumption and the number of open connections required to complete the search.

We argued that having the object namespace define the peer-to-peer connectivity is central to the applicability of our replication model. We further argued that a non-virtualized object namespace maintains properties that are lost through virtualization. First, virtualization destroys locality and opportunities for efficient browsing, pre-fetching and searching are lost. Second, virtualization discards useful application-specific information, such as object relationships that would be otherwise exposed by a hierarchical namespace. Finally, we argued that integrating enhanced operational support (browsing, searching, etc) with the hierarchical overlay network layer improves the performance of the operations.

We have described d-space, a method for defining replica quorum groups by arranging participating sites in logically structured multi-dimensional spaces. Replica control protocols using d-spaces can provide one-copy serializability correctness. D-space is a generalization of the Grid protocol to multiple dimensions. It is also reciprocal to Grid in the sense that it defines inverted read and write quorum groups with respect to subspace covers. Thus, as opposed to Grid, d-space quorums are ideal for scenarios where read operations occur orders of magnitude more often than updates (RFWM).

We argued that d-space quorums offer a flexible way to build protocols with ideal balances of low communication complexity and high availability. First, d-spaces enable the more frequent read operations to execute efficiently, at a limited and controllable expense of more rarely executed write operations. This leads to our first result: for any given read/write access ratio, a d-space can be configured to give optimal communication complexity. Second, read operations can be performed efficiently without adversely affecting the availability of updates. For high access ratios the availability of updates can approximate or even match the availability of read operations. To our knowledge, the quality of trade-off between read efficiency and update availability of our approach is not matched by existing quorum protocols. In particular, we showed that implementing the RFWM protocol using d-space yields superior operational availability as well as communication complexity to the hierarchical quorum consensus method.

Existing structured quorum schemes are based on global views. This allows good access latencies but hurts a protocol's adaptivity, as it must rely on heavyweight global reconfiguration mechanisms. We showed how to implement lightweight d-space reconfiguration through a combination of local information and a relaxed form of global views. The key to our approach is in making distinction between the virtual replica space and the server space, and maintaining the mapping between the two spaces as cached quorums at servers. Reconfiguration manifests in stale cached entries and the scheme's cost translates to refreshing such entries. We have shown that using virtual replicas preserves all the desirable properties of the original scheme, and that the cost of reconfiguration is within acceptable bounds even when the system is highly stressed with failures and membership changes.

5.0.2 Future Research Directions

There are many challenging avenues for future research in replication protocols for peer-to-peer networks. We outline several promising research directions next.

The main reason for replicating data is to improve its availability. If nodes are highly available and failures in the system are not the common case, then a high level of availability can be achieved with a small degree of replication. Further, replicating data excessively may result in high communication costs as generally the communication complexity is a function of and increases with the degree of replication. This is true of the d-space method where the size of a quorum group depends directly on the degree of replication. Finally, a higher degree of replication can have the paradoxical effect of actually making data less available. This becomes apparent for d-spaces when the read/write access ratio is kept constant and the degree of replication increases. Ideally, the degree of replication should be set such that data availability is not hurt, and at the same time communication costs are kept at a minimum. It is not clear at this point how to derive the optimal degree of replication (N) given a targeted access ratio (ρ_{wr}) and operational availability.

The communication and processing overhead of a replica control protocol depends on its consistency model. The stronger the guarantees offered by the protocol, the higher these costs are. The d-space protocol ensures one-copy serializability and thus has a relatively high processing overhead. While strict correctness is necessary for many application classes, there exist many application domains where weaker consistency criteria are acceptable. An example of such a domain may be file systems where users can tolerate occasional inconsistencies in exchange for high data availability at low cost. In fact, we exploited the semantics of data in developing a replication model for routing on hierarchical namespaces in TerraDir. The motivating observation was that inconsistencies of replicated routing state can only manifest in less precise forwarding steps. This incurs a relatively small penalty with respect to routing performance without affecting the correctness and semantics of query processing. We made no assumptions regarding the nature of potential client applications for TerraDir, and noted that d-spaces can be used as the replica control protocol for data exported by applications. If precise characterizations of weaker consistency models for client applications can be developed, they can be used to design more efficient replica control protocols.

BIBLIOGRAPHY

[AA89] D. Agrawal and A. El Abbadi. An efficient solution to the distributed mutual exclusion problem. In *Proc. of the* 8^{th} *ACM Symposium on Principles of Distributed Computing*, pages 193–200, Edmonton, Alberta, Canada, June 1989. ACM Press.

[AA96] D. Agrawal and A. El Abbadi. Using reconfiguration for efficient management of replicated data. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):786–801, October 1996.

[AAC91] M. Ahamad, M. H. Ammar, and S. Y. Cheung. Multidimensional voting. *ACM Transactions on Computer Systems*, 9(4):399–431, November 1991.

[AAC94] M. Ahamad, M. Ammar, and S. Cheung. Replicated data management in distributed systems. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 572–591. IEEE Computer Society Press, Los Alamitos, CA, January 1994.

[ABCdO96] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of PDIS'96: The IEEE Conference on Parallel and Distributed Information Systems*, Miami Beach, FL, December 1996.

[AHKV03] M. Adler, E. Halperin, R. Karp, and V. Vazirani. A stochastic process on the hypercube with applications to peer-to-peer networks. In *Proc. of the* 35th *ACM Symposium on Theory of Computing*, San Diego, CA, June 2003.

[AT89] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264–290, June 1989.

[AW98] Y. Amir and A. Wool. Optimal availability quorum systems: Theory and practice. *Information Processing Letters*, 65(5):223–228, 1998.

[BCF$^+$99] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the INFOCOM Conference*, pages 126–134, March 1999.

[BG84] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9(4):596–615, December 1984.

[BG87] P. A. Bernstein and N. Goodman. A proof technique for concurrency control and recovery algorithms for replicated databases. *Distributed Computing*, 2(1):32–44, January 1987.

[BGM84] D. Barbara and H. Garcia-Molina. Optimizing the reliability provided by voting mechanisms. In *Proc. of the* 4th *International Conference on Distributed Computing Systems*, pages 340–346, San Francisco, CA, May 1984.

[BGMS89] D. Barbara, H. Garcia-Molina, and A. Spauster. Increasing availability under mutual exclusion constraints with dynamic voting reassignment. *ACM Transactions on Computer Systems*, 7(4):394–426, November 1989.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, Boston, MA, 1987.

[BKS01] B. Bhattacharjee, P. Keleher, and B. Silaghi. The design of TerraDir. Technical Report CS-TR-4299, University of Maryland, College Park, MD, October 2001.

[Blo70] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the Association for Computing Machinery*, 13(7):422–426, 1970.

[CAA92] S. Y. Cheung, M. H. Ammar, and M. Ahamad. The Grid protocol: A high performance scheme for maintaining replicated data. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):582–592, December 1992.

[CKK02] Y. Chen, R. Katz, and J. Kubiatowicz. Dynamic replica placement for scalable content delivery. In *The 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[CS02] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *The ACM SIGCOMM Conference*, August 2002.

[CSWH00] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the ICSI*

Workshop on Design Issues in Anonymity and Unobservability, pages 311–320, Berkeley, CA, July 2000.

[DKK$^+$01] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, October 2001.

[DR01] P. Druschel and A. Rowstron. PAST: a large-scale persistent peer-to-peer storage utility. In *Proc. of the 8th IEEE Workshop on Hot Topics In Operating Systems VIII*, Schloss Elmau, Germany, May 2001.

[EK97] D. Eastlake and C. Kaufman. Domain Name System Security Extensions. RFC 2065, January 1997.

[Fal85] C. Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1):49–74, March 1985.

[GDS$^+$03] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. of the* 19th *ACM Symposium on Operating Systems Principles*, pages 314–329, Bolton Landing, NY, October 2003.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of ACM SIGMOD*, pages 173–182, Montreal, Canada, June 1996.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proc. of the* 7th *Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, CA, December 1979.

[GMB85] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.

[gnu] Gnutella home page. http://www.gnutella.com.

[gnu00] *The Gnutella protocol specification* . http://dss.clip2.com/GnutellaProtocol04.pdf, 2000.

[GSC+83] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox, and D. Ries. A recovery algorithm for a distributed database system. In *Proc. of the* 2nd *ACM Symposium on Principles of Database Systems*, pages 8–15, Atlanta, GA, March 1983.

[HBJ+90] A. Hisgen, A. Birrell, C. Jerian, T. Mann, M. Schroeder, and G. Swart. Granularity and semantic level of replication in the Echo distributed file system. In *Proc. of the IEEE Workshop on Management of Replicated Data*, pages 2–4, Houston, TX, November 1990.

[Her86] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[Her87] M. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2):170–194, June 1987.

[HHB96] A. A. Helal, A. A. Heddaya, and B. B. Bhargava. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, Boston, MA, 1996.

[HHL+03] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *Proc. of the* 29th *Inter-*

national Conference on Very Large Data Bases, pages 321–332, Berlin, Germany, September 2003.

[JM90] S. Jajodia and David Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15(2):230–280, June 1990.

[KBC+00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of the* 9^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, Cambridge, MA, November 2000.

[KBS02] P. Keleher, S. Bhattacharjee, and B. Silaghi. Are virtualized overlay networks too much of a good thing? In *Proc. of the* 1^{st} *International Workshop on Peer-to-Peer Systems*, volume 2429, pages 225–231, Cambridge, MA, March 2002. Springer-Verlag.

[KLL+97] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, El Paso, TX, May 1997.

[Kum91] A. Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Transactions on Computers*, 40(9):996–1004, September 1991.

[Lam78] L. Lamport. Time clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam86] B. Lampson. Designing a global name service. In *Proc. of the 5th ACM Symposium on the Principles of Distributed Computing*, pages 1–10, Minaki, Canada, August 1986.

[LCC$^+$02] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th ACM International Conference on Supercomputing*, New York, USA, June 2002.

[LCKM00] J. Li, D. S. J. De Couto, D. R. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. In *Proc. of the 6th ACM International Conference on Mobile Computing and Networking*, Boston, MA, August 2000.

[Mae85] M. Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[MD88] P. V. Mockapetris and K. J. Dunlap. Development of the Domain Name System. In *Proc. of ACM SIGCOMM*, pages 123–133, Stanford, CA, August 1988. also in *Computer Communication Review* 18 (4), Aug. 1988.

[MM02] P. Maymounkov and D. Mazire. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of the* 1^{st} *International Workshop on Peer-to-Peer Systems*, volume 2429, Cambridge, MA, March 2002. Springer-Verlag.

[MNR02] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proc. of the* 21^{st} *ACM Symposium on Principles of Distributed Computing*, pages 183–192, Monterey, CA, July 2002.

[Moc87a] P. V. Mockapetris. Domain names - concepts and facilities. Request for Comments 1034, Internet Engineering Task Force, November 1987.

[Moc87b] P. V. Mockapetris. Domain names - implementation and specification. Request for Comments 1035, Internet Engineering Task Force, November 1987.

[mor] Morpheus home page. http://www.musiccity.com.

[nap] Napster home page. http://www.napster.com.

[nup] Nupedia home page. http://www.nupedia.org.

[PRR97] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of the ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, Newport, RI, June 1997.

[RD01a] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of the* 18^{th} *IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, November 2001.

[RD01b] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the*

18th ACM Symposium on Operating Systems Principles, Chateau Lake Louise, Banff, Canada, October 2001.

[RFH$^+$01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proc. of the ACM SIGCOMM*, San Diego, CA, August 2001.

[RKCD01] A. Rowstron, A. M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proc. of the 3rd International Workshop on Networked Group Communication*, London, UK, November 2001.

[RL92] M. Rabinovich and E. D. Lazowska. Improving fault-tolerance and supporting partial writes in structured coterie protocols for replicated objects. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 226–235, San Diego, CA, June 1992.

[RL93] M. Rabinovich and E. D. Lazowska. Asynchronous epoch management in replicated databases. In *Proc. of the* 7$^{\text{th}}$ *International Workshop on Distributed Algorithms*, pages 115–128. Springer-Verlag, 1993.

[SBK02] B. Silaghi, B. Bhattacharjee, and P. Keleher. Query routing in the TerraDir distributed directory. In *Proc. of SPIE ITCOM*, Boston, MA, August 2002.

[SDWG95] M. A. Sheldon, A. Duda, R. Weiss, and D. K. Gifford. Discover: A resource discovery system based on content routing. In *Proc. of the 3rd International World Wide Web Conference*, Darmstadt, Germany, 1995.

[SGG02] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN)*, San Jose, CA, USA, January 2002.

[SKK$^+$90] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[SMB02] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *The 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[SMK$^+$01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the ACM SIGCOMM Conference*, San Diego, CA, August 2001.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[Sri01] K. Sripanidkulchai. The popularity of Gnutella queries and its implications on scalability. February 2001.

[SS83] R. D. Schlichting and F. B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[Sto79] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, 5(3):188–194, May 1979.

[Tho78] R. H. Thomas. A solution to the concurrency control problem for multiple copy databases. In *Proc. of the* 16th *IEEE Computer Society International Conference*, New York, NY, Spring 1978.

[Tho79] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[TK88] Z. Tong and R. Y. Kain. Vote assignments in weighted voting mechanisms. In *Proc. of the* 7th *Symposium on Reliable Distributed Systems*, pages 138–143, Columbus, OH, October 1988.

[Tsu88] P. F. Tsuchiya. The Landmark hierarchy: A new hierarchy for routing in very large networks. In *Proc. of the ACM SIGCOMM Conference*, pages 35–42, Stanford, CA, August 1988.

[vSHBT98] M. van Steen, F. J. Hauck, G. Ballintijn, and A. S. Tanenbaum. Algorithmic design of the globe wide-area location service. *The Computer Journal*, 41(5):297–310, 1998.

[vSHHT98] M. van Steen, F. J. Hauck, P. Homburg, and A. S. Tanenbaum. Locating objects in wide-area systems. *IEEE Communications Magazine*, pages 104–109, January 1998.

[vSHT99] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, January–March 1999.

[wik] Wikipedia home page. http://www.wikipedia.com.

[WJH97] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.

[Woo98] A. Wool. Quorum systems in replicated databases: Science or fiction? *Buletin of the Technical Committee on Data Engineering*, 21(4):3–11, 1998.

[WSBL99] W. A. Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island, SC, December 1999.

[X5088a] The Directory: Abstract Service Definition. CCITT Recommendation X.511, ISO/IEC JTC 1/SC21; International Standard 9594-3, 1988.

[X5088b] The Directory: Models. CCITT Recommendation X.501 ISO/IEC JTC 1/SC21; International Standard 9594-2, 1988.

[X5088c] The Directory: Overview of Concepts, Models and Service. CCITT Recommendation X.500., 1988.

[YGM02] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer systems. In *Proc. of the* 22nd *International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

[YHK93] W. Yeong, T. Howes, and S. Kille. X.500 Lightweight Directory Access Protocol. Network Working Group RFC 1487, ISODE Consortium, 1993.

[Zip49] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, Cambridge, MA, 1949.

[ZKJ01] B. Zhao, K. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, University of California at Berkeley, April 2001.