Peer-to-Peer Architectures for Data Discovery, Distribution and Consistent Replication

A Dissertation

Presented to the

Graduate Faculty of the

University of Louisiana at Lafayette

In Partial Fulfilment of the

Requirements for the Degree

Doctor of Philosophy

Ian Chang-Yen

Fall 2014

Peer-to-Peer Architectures for Data Discovery, Distribution and Consistent Replication

Ian Chang-Yen

APPROVED:

DEDICATION

Dedicated to the memory of A. Edwin Alexander, who started me on this academic journey.

ACKNOWLEDGEMENTS

I would like to thank my dissertation advisor Dr. Nian-Feng Tzeng for providing me with guidance throughout both my dissertation and academic publications preceding it. I would also like to thank my dissertation committee of Dr. Magdy Bayoumi, Dr. Vijay Raghavan and Dr. Hong-yi Wu for providing valuable advice and direction.

I would like to thank my father, Dr. Ivan Chang-Yen, for his invaluable help in proofreading my dissertation drafts and providing technical assistance with my research; his trained eyes picked up so much that I would have missed. I would also like to thank my mother, Hazel Chang-Yen, for her continual support and encouragement in my academic endeavors, and for always being there to give me the pushes when I needed them the most. Special thanks go out to my close friends and in-laws for encouraging me to push towards the light at the end of the academic tunnel.

Finally, I would like to thank my wife, Elizabeth for her love, support and infinite patience as I labored my way to completion.

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

1.1. Collaborative Computing

As the paradigm of collaborative computing has matured, its requirements began to affect the architecture of server rooms, or “data centers” on a large scale. Companies, once forced to buy hardware for their individual data-processing needs, were now provided with a new alternative—virtual computing. Enabled by the development of software, such as Xen and VMWare, and the Linux-based Kernel Virtual Machine [4] [44] [62], virtual computing allows the components of an entire computer to be emulated or “virtualized,” so that operating systems and applications are no longer beholden to the hardware on which they were installed. Instead, a hosting piece of software or “hypervisor” could provide the hardware emulation required, thus reducing an entire computer to little more than a collection of data within a file-system. In the virtual computing sphere, this collection of data is termed a *Virtual Machine* (VM*).* With this new capability came new possibilities; since the VM “computers” were little more than software, they could now be sent from machine to machine, using the same inter-networking protocols as the applications they ran would use. This allowed these VMs to “migrate” whenever the need arose, e.g. if hosting hardware became overloaded or potentially faulty [18]. With the advent of clever management systems to intelligently shuffle VMs around to respond to various needs, the data center became ever more efficient and flexible. Essentially, a new dimension was added to the original concept of shared computing, since now a business could rent out its processors and storage for use by other companies. Termed “cloud computing,” this became a very successful business model for companies like Amazon, Google and Microsoft [3]

[17] [71] [91]. Additionally, open-source tools such as AppScale [15] and Eucalyptus [64] were also developed to create applications in these new virtual environments.

At the same time as cluster computing was evolving into the virtual space, another concept was being developed by researchers—the idea of an abstracted general-purpose computing space where the entire environment for computational problem-solving was stratified into a "middleware" layer, removed from the idiosyncrasies of the underlying operating systems and hardware. First coined by Ian Foster, the term "Grid computing" [29] [31] was used to describe this collection of software, dubbed Globus [22], that enabled even diverse and heterogeneously-created computers to contribute resources to a computational solving space in a homogeneous and comparable manner. Within the Globus framework, researchers were presented with a unified space of both processing power and storage, vastly simplifying application development. This idea did not go unnoticed in the business world, where companies like Yahoo created unified storage infrastructures such as Sherpa [19] and in collaboration with the Apache Foundation created the Hadoop clustering environment [87] [92]. General file-systems and even database servers could now be abstracted into these environments, which freed computational clusters from the previous requirements of homogenous participating hardware, to allow virtually any computer that could meet certain basic software requirements to be utilized in large-scale simulation. This level of abstraction enabled the modern implementations of what we would consider the "Cloud computing" networks of today.

1.2. Virtual Machine Deployment Strategies

While large-scale computing with virtual machines opened up new capabilities to businesses and individuals, the management of these VMs posed new challenges for data center operators. The traditional scenario of operating systems running natively on computers no longer applied, effectively shrinking the time-frame in which computational resources could be constructed and deployed. However, this compression of the time-scale for deployment of both individual machines and clusters of servers exposes the issue of *data movement*. In other words, although the hardware specifications were virtualized (allowing near-instantaneous hardware "creation" within hosting hypervisors), the file-systems which comprise the operating systems and applications of these virtualized machines must still be transported from the location of their creation to the hypervisor, sometimes between distant data-centers [88]. Adding to this complexity is the concept of VM migration, where the VM is moved from one hypervisor to another, as resource or failure conditions dictate [18]. [18] As the data sets which make up each VM can be very large, techniques for rapid VM deployment and migration became more prevalent [14] [15] [47], thus enabling the automated deployment and management of extremely large numbers of VMs for both research and business-oriented end-users.

1.3. Cloud Computing and Data Management

The idea of utility-based cloud computing [30] began to show up in the large-scale computational arena in the mid-2000s, with the launch of dynamically-shared computation services such as Amazon Web Services [2] and Eucalyptus [64]. While encompassing many definitions [3] [30], the phrase "cloud computing" can be broadly described as an environment in which the actual storage and processing of data are deliberately abstracted from the end-user. In other words, a user does not need to know exactly on which servers or data centers his/her information is stored, just that it is stored somewhere accessible. In network architectural nomenclature, a cloud is used to symbolize an area of a network or data storage facility which, while some of its function may be known, is otherwise the equivalent to a “black box” to the end-user. However, what is important to the end-user is the perceived availability of these consumed services. In the case of businesses which may be dependent on the cloud computing architecture to function properly, certain guarantees of data availability, computational power, data security and a host of other requirements must be met by the cloud provider. These requirements, often codified as Service Level Agreements (SLAs) [12], impose particular challenges to the hosting provider.

To help meet these needs, hosting providers utilize the range of aforementioned technologies to ensure that the data which they host and the computational resources needed to process the data could be made rapidly available to their clients, whenever the need arose. In addition, client data should be preserved for as long duration as possible, even in the face of multiple hardware or software-related failures. Originally, this was achieved using off-site backup utilities, generally copying the actively used data to a more robust (but generally slower medium) such as magnetic tape [26]. However, users now require that their data must not only be safe, but must also be readily available, often at a short notice. While multi-level backups can alleviate some of the inherent performance disadvantages of tape backup, a more readily acceptable method makes multiple live copies (or *replicas*) of the same data and distributes it to different machines, sometimes in different data centers or even multiple states or countries [1] [48]. In this approach, the probability of multiple hardware failures affecting all the disparate machines simultaneously is very low, thereby improving the reliability of data storage and retrieval. An additional advantage of this approach is that if the physical location of a client can be determined, then the data can be copied to a data center closer to them, thus reducing the amount of time taken for the client to access their data (termed “access latency”). Modern Content Distribution Networks (CDNs) run by companies such as Akamai [65] use this technique to improve the performance of hosted high-demand data items, such as streaming media.

1.4. Management of Replicated Data

While data replication among multiple data centers solves some data reliability issues, it simultaneously creates new problems. For instance, if a data set is replicated among several computers and a client wishes to change the data at one access point, the changes on the other machines must be made simultaneously, or the other locations will contain different versions of the original data. This essential requirement of ensuring that all the accessing machines have the same “view” of the data, no matter which machine is used to access and change the data, is termed “mutual consistency” [24]. Unfortunately, maintaining this consistency, especially when combined with a client’s requirements for ready availability and modifications, creates many challenges (but also opportunities for the development of novel solutions), especially in environments where hardware or communications failures are common.

Thus if machines hosting copies of the same data cannot communicate fully with each other, they will find it difficult, if not impossible to consistently register any changes to the shared data. This problem was crystallized by Eric Brewer as the *CAP* issue [10], where CAP is an acronym for *Consistency* and *Availability* in the face of Network *Partitions*. This occurs when a communication failure causes one section of a network to become disconnected from the rest, forming a partition which severs communications between storage networks. Given a solution to this problem which satisfies all three requirements (data access, modification, consistency) was found to be mathematically impossible [34] [35], Brewer reasoned that by temporarily relaxing at least one of the *CAP* requirements, the other requirements could be more readily met. For instance, if an end-user can tolerate not being able to access their data for short periods of time, then the replicated data-hosting system could temporarily make the data unavailable to the user, while the consistency between copies is enforced. This arrangement is warranted in environments where consistency is the over-arching requirement, superseding all other conditions, such as in banking data networks. Since a loss of consistency could potentially cause major monetary losses for either the bank or their customers, every effort must be made to prevent this occurrence, even at the cost of temporary unavailability. Under these conditions, customers may be denied temporary access to their money during periods of communication failure among various bank branches. This is considered a case of enforcement of *strong consistency* [10].

On the other hand, data availability can be improved if the end-user tolerates some measure of inconsistency. For example, Amazon utilizes this relaxed consistency in their online shopping carts, where their shoppers were found to tolerate temporarily inconsistent contents in the carts, rather than the cart being completely unavailable. Using this relaxation in consistency, Amazon was able to increase the availability of carts for shoppers by establishing a *write-always* policy for their shopping carts. This policy ensures that a customer is always able to add an item even if the shopping cart data are temporarily unavailable, as may happen when a partition is caused by a temporary loss in communications between the client’s machine and Amazon’s servers. However, once the network issue has been corrected and the partition is resolved, the shopping cart is automatically merged with the added items, to give a consistent result to the customer [90]. Since this model assumes that there will be some temporary data inconsistencies which can be resolved within a short time interval, it is considered to be a form of *weak* or *eventual consistency* [10]. Using such eventual consistency, the data objects may change dynamically during acknowledged inconsistency periods, but will have consistency enforced at specific and well-defined points in time. Allowing consistency to lapse temporarily allows the components of a replicated system to make their (potentially out-of-date) data available to the client even in the face of network partitions. This is dependent on the client users' ability to tolerate the inconsistency periods and that the system can eventually resolve the consistency issues, once network connectivity has been restored.

However, the assumption that the end users simply accept and cope with an inconsistency (even a temporary one) in their operational data is too narrow an approach to take, especially if a data provider is expected to serve a wide and varied clientele. Traditionally, the approaches to solving these distributed concurrency issues have involved forms of *locking* protocols [6]. Using locking, a client accessing shared data sets a special flag associated with the data, indicating to other clients that the data are in the process of being modified. Once the client has completed its required operations to the shared data, it resets the flag to allow other clients to gain access to the data.

Unfortunately, this approach comes with several attendant issues, such as deadlocks, starvation, priority inversion and convoying conditions. While the exact details of the problems will be covered in more detail in Chapter 2, any concurrent-access system which uses traditional locking protocols is subject to additional complexity to deal with the aforementioned problems. In addition, traditional locking imposes a performance penalty in highly concurrent systems. As the overhead required to serialize accesses to the shared data resources lengthens the execution time, the various client programs attempting to read from, or write to the data may have to wait for long periods of time before being granted access, thereby delaying their completion times.

1.5. Transactional Synchronization

Herlihy et al. [37] proposed an alternative to these issues in the form of *transactional* data access. Using techniques gleaned from the database world [8], a system was proposed where a program's access to shared data was presented in the form of *transactions,* each of which could be aborted and/or restarted as needed to achieve transactional synchronization among concurrent operations on shared data. Therefore, programs were no longer restricted by locking protocols to access the shared data—each program simply accesses the data as if it was the only client, with the understanding that any section of code which was couched as a transaction could be aborted and restarted at any time. This form of *optimistic* shared data access via transactional synchronization frees up the programs from the possibility of deadlocks and other associated lock-based problems, while incurring only moderate costs in programmatic management and overhead costs. Transaction synchronization serves as the basis of our data replication management strategy to be detailed in Chapter 4.

1.6. Motivation and Dissertation Outline

While the trend of collaborative computing moves towards large-scale data management and abstracted virtual computing, current large-scale implementations of this technology are dependent on the infrastructure of large consolidated data centers, such as those operated by Google, Akamai or Microsoft [88]. However, the majority of operational server rooms are less than 5000 sq. ft. [56] and this has encouraged the development of integrated data storage and virtual computing frameworks. These frameworks combine the resources of smaller data centers and server rooms into a single addressable infrastructure, to overcome issues of low bandwidth, high latency and/or unpredictable network reliability between smaller data centers. The function of such an integrated system would be to allow a client to dynamically deploy virtual computational resources within any participating data center in the system, while simultaneously providing a shared space for the storage of very large data sets. Essentially, each participating smaller data center would become part of a widely distributed large-scale "data center." This can leverage the advantages of disaster mitigation via geographic site separation, while minimizing the bandwidth and latency associated costs. Therefore, it would serve as an attractive alternative to customers who are seeking to enhance the reliability, storage or computational capability of their existing data centers. This is especially important for customers who are being forced to undertake expensive data center expansions or to consolidate their data and machines within a foreign cloud-enabled data center. To this end, this research proposes a Peer-to-Peer (P2P) based overlay for storing and retrieving large replicated data items from widely-spaced collections of computers. Using techniques gleaned from distributed CDNs and *Distributed Hash Table* (DHT) based search networks, the distribution of and search for replicated data within a decentralized network overlay are investigated as both a distribution system for virtual clusters and a more general transaction-based data workspace. The focus of the research revolves around the application of performance-enhancing techniques for data retrieval and manipulation under both loose and tight replication conditions, while maintaining acceptable limits over both the reliability of the overlay and controllable consistency of the replicated data it is designed to store.

Our proposed system described in this dissertation consists of two major components. The first component, named the *Peer-to-Peer Virtual Cluster Deployment System* (PVC-DS), combines and extends improvements to existing virtual machine data structures, paving the way for more enhanced search and retrieval of loosely-replicated VMs. This PVC-DS component has been designed to augment existing virtual hosting environments, thus reducing the deployment time of VMs. In a simulated network environment, PVC-DS is compared to existing virtual machine replication and distribution topologies, to determine what performance advantages it offers. Using the same underlying communications infrastructure as PVC-DS, the second component adds transactional support to enable the replicated storage of general data objects. As replication is also employed in the proposed overlay for the purposes of reliability and performance improvement in data retrieval and manipulation, mutual consistency techniques for concurrent data accesses are investigated. We investigate the trade-offs between maintaining serializable consistency in replicated data and reducing the delays incurred by clients attempting to use the shared data items. To this end, a particular form of fault-tolerant replication managed by transactional synchronization, utilizing *soft-state* [73] replication is developed and evaluated. Soft-state registration was another important component of the eventual consistency scheme envisioned by Brewer [10], and is discussed in more detail in Section 2.2. While no prior work in this exact context is aware to us, we will nevertheless attempt to provide measures of comparison between our proposed mechanisms and existing general file dissemination and replication consistency enforcement schemes.

Following an extensive literature search on data replication, search and dissemination architectures within P2P networks, our primary objectives of this research are:

- The integration and enhancement of multiple existing P2P-based overlays into a single cohesive system, with the specific aims of (i) improving the search capabilities of replicated VM data-repositories; (ii) the performance of deploying the VM-data to end-point clients; (iii) providing a replicated data-storage space for use by applications operating within the deployed VMs.
- The development of a multi-layered storage architecture within the P2P overlay, along with the creation of new publishing and querying algorithms which leverage this architecture.
- The development of a fault-tolerant P2P-based architecture for low-latency distributed transactional data storage, designed to operate as part of the cohesive P2P-based system.
- The design of a simplified transactional algorithm for operating both in concert with, and taking advantage of the specific features of the underlying fault-tolerant architecture.

To verify the correctness and performance of these new approaches, we have also implemented a scalable event-driven simulation environment. This environment has been designed to be as flexible as possible, to allow the testing of any number of extreme end-user requirements and network topologies, and yet to be reasonably representative of real-world communications architectures.

In the following chapter, we will elaborate on prior research aimed at improvements in VM deployment time and provide technical explanations of the network infrastructures used for the underlying communications fabric of our proposed work. This is followed by an in-depth exploration into different forms of consistency management of both single-instance and replicated data objects. Explanations will be provided for the rationale for selecting and modifying certain algorithms and architectures and how these augmented mechanisms are specifically applied to our research.

2. Foundational Concepts and Prior Research

2.1 Virtual Clusters

As new uses for virtually created and managed servers in both corporate environments and research communities were created, the requirement for ever larger computing clusters necessitates the use of scalable software tools to assist with the creation and management of these large groups of VMs [78]. Many large VM providers such as Amazon and Microsoft house these VMs in multiple data centers spread geographically across countries or even continents [88]. Despite the increase in bandwidth into these data centers, the large distances between the VM-hosting facilities complicate the deployment of the VMs to their hosting hardware, especially as the size of the operating systems running within these VMs have become larger over time. The demand in automation for the creation, deployment & management of virtual machines have led to the development of several deployment frameworks for virtual servers and grouped computing clusters. Foster et al. [32] first described a framework for creating and distributing VMs, using an implementation of Web Service Resource Framework (WSRF) metadata, called *workspaces.* Their framework largely covered the description of VM clusters, assuming that the workspace nodes were capable of accessing a centralized repository, to download the fully configured VM images. Unfortunately, as the virtual cluster sizes were increased, the download times increased significantly as well, due to the repository having to service multiple download requests, resulting in network links becoming saturated. Moreover, as virtual cluster deployments were being co-opted for use in cloud computing systems such as Eucalyptus [64], the issue of VM data propagation continued to be a pressing issue. Schmidt et al. [84] acknowledged this problem and outlined possible alternatives to the centralized systems. They concluded that a P2P distribution system such as BitTorrent [70] would be the most efficient method for distributing VM images and software, compared to unicast, multicast or tree-based distribution architectures. While this approach does improve the VM download times, it still relies on the transfer of fully configured VM images from the repository. While this can be compensated for by the use of local client-based caching, as done in the Schmidt approach, or in Eucalyptus which employs some form of VM data caching, the client nodes still required large amounts of local disk space to hold the multiple cached VM images.

In an attempt to alleviate some of the issues inherent in earlier implementations [32], Nishimura et al. [63] applied individual application-specific packages from the base *Operating System* (OS) image and downloaded them separately. Subsequent use of a dendrogram tree-based package caching strategy resulted in a reduction in deployment time. However, such an approach had major shortcomings, since their simulation assumed that only a single OS type and architecture is applicable. However, in large-scale heterogeneous *Virtual Cluster* (VC) networks, the hypervisors would host many different types, versions and architectures of operating systems. Given that many applications and packages are OS- and version-dependent (e.g. compilers tend to rely on kernel and system library versions), the client or local VC site manager would have to cache multiple versions of each package. This situation is aggravated by the fact that many OS distributions use far more than the 180 different package combinations tested in Nishimura's evaluation. As a result, the caching site managers would either require *Virtual Disk* (VD) caches or employ very aggressive cache-clearing policies to make the system work.

Nishimura's system also assumed that the cached packages do not change over time with respect to their underlying OS. However, this assumption does not hold true in general, since application packages associated with a specific OS may evolve at a very rapid pace, calling for re-deployment of the same packages multiple times, sometimes within a very short period of time. For example, consider a graphics research group which is developing and testing a parallel ray-tracing engine using virtual clusters, with each updated version of the software being re-deployed into the VM repository. As a result, the client nodes hosting the virtual cluster would have to repeatedly resynchronize new packages with the centralized repository, with the potential for overloading, as described by Foster [32]. What was needed to help solve this issue was an effective way to distribute the VM repository data among multiple nodes, such that data were no longer tied to a single node on a network. Distributing the task of serving the VM repository would alleviate issues of overloading repositories, along with providing resistance to failure of hosting nodes, via the replication of the VM data. In order to facilitate the distribution of the data among nodes, P2P overlay networks provide a solid basis for building distribution and retrieval systems for clusters of virtual machines.

2.2 Peer-to-Peer Networks

To meet the needs of network communications for clients, many forms of Peer-to-Peer (P2P) networks have been developed and can be broadly categorized as either *unstructured* or *structured* networks [52]. Unstructured P2P networks include the Gnutella [79] and Kazaa [51] file-sharing networks, where nodes locate each other by simple expanding-ring broadcasts. To do so, a node searching for a data item will simply broadcast its search terms to its nearest neighbors, which will in turn forward the request to their neighbors, up to a maximum number of hops. While this architecture is extremely simple to implement and is fairly resistant to network failures, it does suffer from the drawback of searches generating extremely large traffic volumes.

A more efficient alternative is the structured P2P network, whereby each node maintains internal data structures which have the end-effect of organizing the network of nodes into more effective communication topologies. These networks are conducive to more rapid searches, more efficient lookups for rarer items, searches which produce less generated traffic overhead, or any combination of these desired features. Unfortunately, these capabilities have the disadvantages of slower and less efficient structured network operations during times of high network churn, when a very large number of nodes either enter or leave the network during a relatively short period of time [76]. Nevertheless, overall structured P2P networks exhibit properties which make them desirable for use in either data storage or data advertising systems. For example, the BitTorrent [70] P2P file-sharing application uses a type of internal structure called a *Distributed Hash Table* (DHT) to advertise items which data-hosting nodes wish to make available for download. DHT-based structured P2P networks include Chord, Pastry and the Content-Addressable Network [74] [82] [89]. For the purposes of this dissertation, the focus will be on the Chord network, since each of the other DHT-based networks operates in a broadly similar fashion.

Chord [89] is a distributed lookup protocol which provides a simple, relatively robust way for individual nodes to store and retrieve information in a de-centralized manner. Chord uses a *consistent hashing* [59] mechanism based on the *Secure Hash Algorithm* (SHA-1) [28] to map both stored data and network nodes to unique hashed binary keys of fixed length m. Each node is assigned a hash based on its network address, whereas each item of data to be stored is similarly processed via the hashing function, to produce a

Chord-suitable key. The data-key is stored on the node whose node-key is either equal to or immediately follows (or succeeds) the data-key. By using a consistent hashing function to map the data-keys to the nodes, the data-keys are usually evenly distributed and rarely need to be redistributed to new nodes, even if a large number of nodes join or leave the network. The Chord network is arranged as a ring, with nodes in the ring being arranged in increasing order of hash-values. The node with the largest hash-key value in the system simply precedes the node with the smallest hash-key in the Chord network, thereby completing the circle. Each node maintains at minimum a pair of *predecessor* and *successor* links, pointing to the nodes with the next-lowest and next-highest hash-keys, respectively. In addition to these two links, each Chord node with hash n also maintains a fixed-size *finger table* of node hash-keys, where a node at line k in the finger-table immediately succeeds the value of $(n + 2^{k-1}) \mathit{mod}\ 2^m$; the *mod* 2^{m} component keeps the calculated finger within the hash-key limits of the network. These links provide shortcuts to reduce the number of nodes which a Chord search would need to traverse in order to find a particular data-key. An example of a Chord network with stored keys is shown in Figure 1. In this example, a data-key 95 is stored at node 99, as the node-key 99 immediately follows the data-key 95. The search for the key from node 15 is also shown, as it traverses the finger-link shortcuts to reach the key-hosting node. The finger table for each node involved in the search is shown as $F = \{f_1, f_2, f_3, f_4, f_5, f_6\}$. As shown by the arrows, search is performed in a series of hops from each node, with each node along the path choosing the largest finger which precedes the search-key. If the last hop in the search does not fall directly on the hosting-node, the search simply continues sequentially from successor to successor until the key-hosting node is found. In this manner, the finger tables serve simply as a performance improvement, allowing a sequential next-successor search to “skip” nodes along the search path, thereby reducing the hop count and resulting search time.

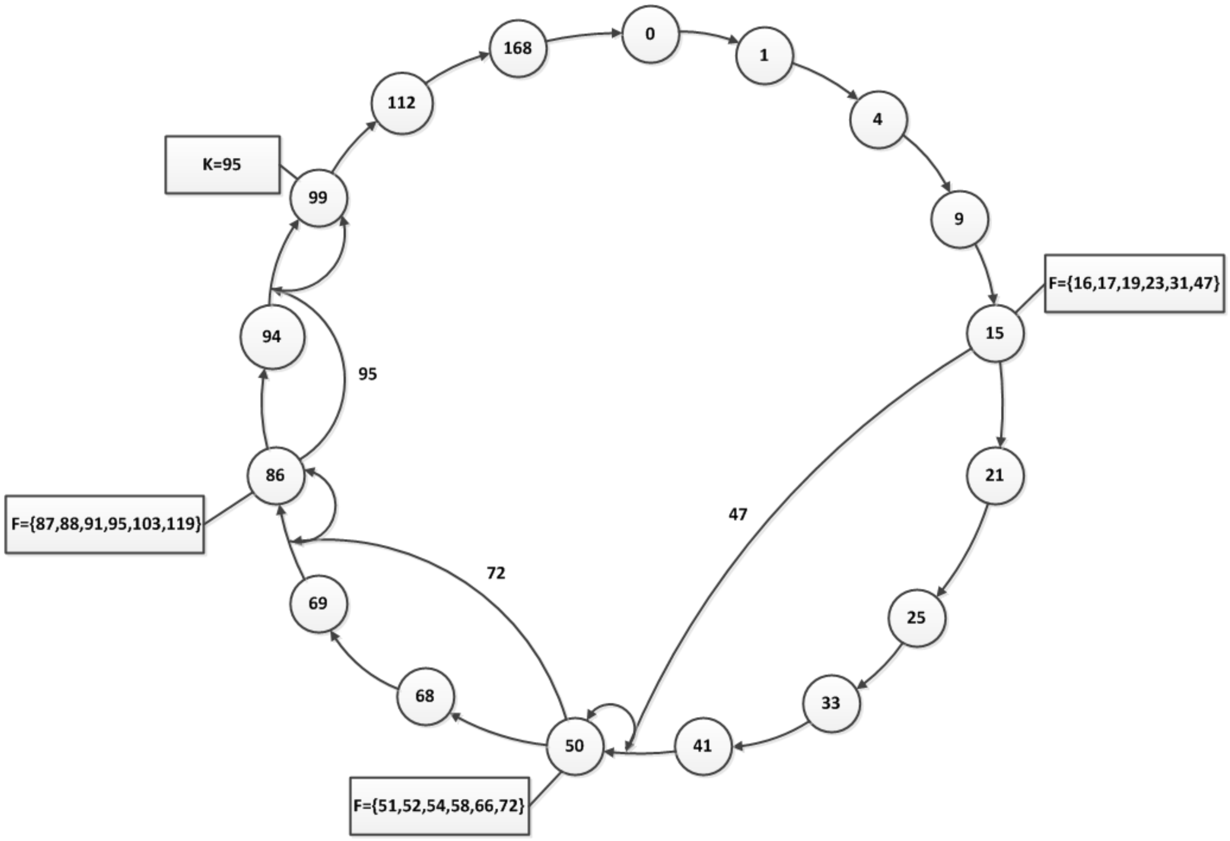


Figure 1. Chord network example.

With these structures, a Chord node is normally able to complete a search within *Olog(n)* node hops. Whenever a node needs to join the Chord network, it simply inserts itself into the ring at the appropriate point determined by its assigned hash-key. Then, after informing the relevant predecessor and successor nodes, it creates its new finger-table links via searches for the appropriate calculated keys. Similarly, upon the event of a node leaving the network (whether intentionally or by network failure), the remaining nodes can use the existing data in their predecessor/successor links and finger tables to reconstruct any broken links which may have arisen as a consequence of the node loss. This gives the Chord network a degree of resilience against network failure and provides a basis upon which to build a fault-tolerant data storage and/or dissemination network. An important characteristic of Chord (and indeed other DHT-based networks) is that searches for a particular data item are done in a series of hops from the requesting node, towards the data-hosting node. For each search, the path traced from the requester to the hosting-node traverses links common to those used by other searching nodes for the same key, with increasing probability of common links being traversed as the search nears the hosting-node. Therefore, the nodes whose hashes closely precede those of a particular data-hosting node will be included in searches for that particular data-key. This has the net effect of producing a "tree" of search paths for each data-key, with the data-hosting node being the "root" of the tree. This particular property can be utilized when dealing with replicated data, to assist in choosing viable replica nodes, as the paths of various nodes which lie along the search path can be mapped to determine which chosen replicas will reduce data transfer time between nodes. Even in the absence of replication, nodes which lie along the search path for a particular data-key can cache the search information locally. Subsequent queries for that key can then be intercepted by the node, improving query response time, but with the potential for querying clients to receive query responses which may contain inconsistent or out-of-date information. As DHT networks traditionally store immutable data, the issue of outdated information is not a concern. However, if a DHT-based storage network is to incorporate mutable data storage capability, then it must be modified to provide this capability. As envisioned by Brewer [10], *soft-state data* (or *soft-state registration*) provides some of the capability for dynamically changing data storage. A node which stores a copy of a data item in the soft state adds a timed condition to the item, which is deleted upon expiration of the timer, unless the original owner of the data periodically sends a refresh signal to the storing node. This signal “re-registers” the data, resetting the timer and informing the storing node of any changes made by the data owner to the item. In this fashion, the data-storing node can still make the data available to other clients without requiring continuous communication with the data-owner, but will eventually remove the stale data item in the event of the loss of the original owner. In addition, the periodic updates ensure that the stored data are kept up-to-date within the bounds of the refresh interval. Even when the data-storing and data-owning nodes may become inconsistent between refreshes (e.g. if the owner changes the original data item, while another client is reading the soft-state copy), this inconsistency is *eventually* resolved with the refresh. This technique is used in Chunkcast [16], a *Content Distribution Network* (CDN) built on top of a distributed hash table. Chunkcast is described in more detail in the following section.

The assignment and searching of data-keys in the Chord network are performed simply by virtue of the hashing function, and the overlay contains no information about the underlying network, with node-keys being assigned without regard for location suitability. Therefore, a client node may be searching for a data-key which is stored on a node very close to the client, but the search may still traverse many other network-distant nodes, thereby increasing search time. As stated above, soft-state caching of the data-key at intermediate-hop nodes may reduce such search times, but at the expense of the client retrieving out-of-date data, particularly if the values referenced by the data-key are updated frequently. Thus in the absence of caching, all data-key searches are routed to the single hosting-node, which may overload it if the requested data-key is extremely popular and/or if the hosting node is connected to the network by a low-bandwidth link. Consequently, to allow the DHT to be used as a hosting medium for higher-demand storage networks, such as for hosting large volumes of data with many widely distributed clients, improvements have been made, by building upon the base DHT and by adding new search, storage and caching functionality. Some of these techniques are described in the following section.

2.3 ChunkCast

One of the storage systems developed out of structured P2P networks was ChunkCast [16], intended as a decentralized storage and search medium for hosting frequently accessed immutable data items, such as large multimedia files. Each item to be published is referenced in the Content Distribution Network or CDN, by a tuple $\{o, n, l, a\}$, where o is the *Object Identifier* (OID); n is the network address of the hosting node; l refers to the network location (such as a co-ordinate system of the hosting node provided by Vivaldi [23], or the distributed binning algorithm of Ratnasamy et al. [74]); a is a bit array representing the object data divided into equally sized chunks. In ChunkCast, the DHT data-key is used to represent the OID. Accordingly, a provider may only possess a subset of all the chunks in a given object, while the bit array reflects which of these chunks it can provide. An illustration of the publish/query operations of ChunkCast is shown in Figure 2, of the tree structure created for an object (OID = 7) in ChunkCast. Using the underlying DHT, each object host of Nodes 2, 3 and 5 sends a publication message towards the OID-root, which resides on the node whose DHT-generated *Node-ID* (NID) is bit-wise closest to the OID. In addition to the meta-data describing the actual stored data on the publishing node, the publish and query messages also contain network locality information for each node n (indicated by $loc(n)$), provided by a location service such as Vivaldi. As the message is routed via the DHT from node to node on its way to the root, the intermediate nodes cache this publish information as well. Since subsequent queries for this OID (such as the one from Node 4) are routed in the same fashion towards the root, they can be intercepted and replied to by an intermediate node (in this case, Node 3), thus improving query response times.

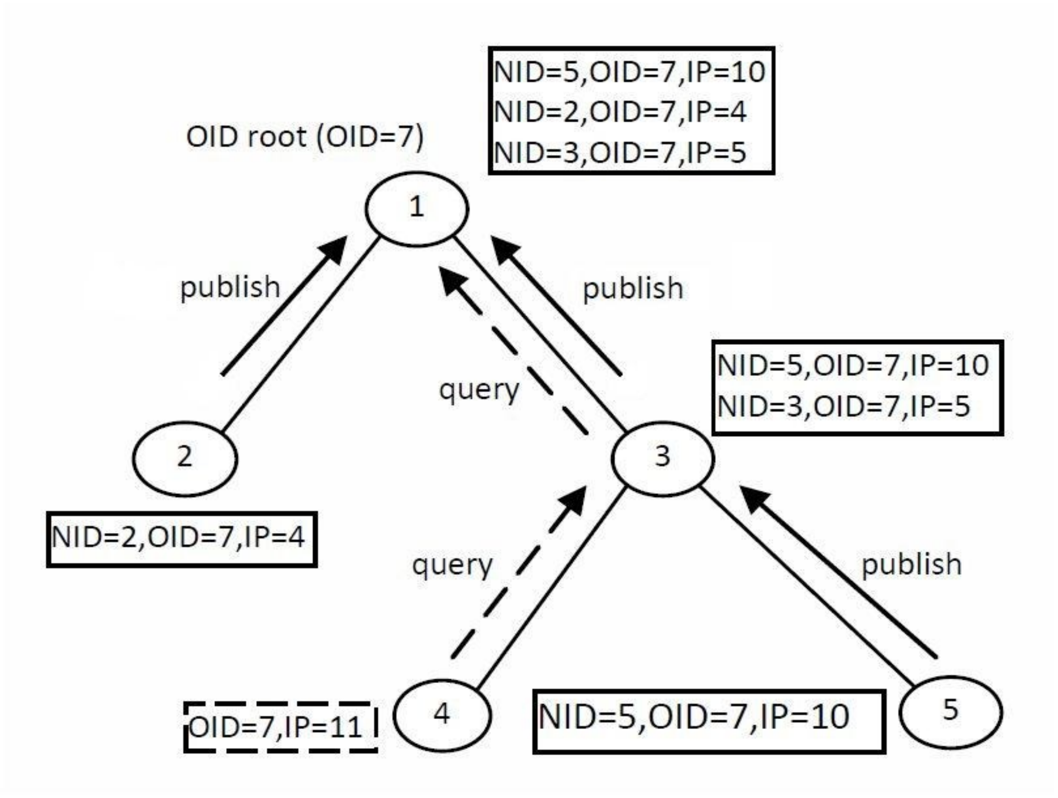


Figure 2. ChunkCast publish and query example.

Although soft-state registration is used in the caching of the object metadata, the design of ChunkCast is oriented towards static data objects such as video files. Soft-state registration, while ensuring eventual consistency between the data-storing nodes and the metadata-caching nodes, is not used to convey any *changes* in the actual content or values of the data itself, merely its *availability*. Therefore, ChunkCast assumes that once defined, each OID-represented object is immutable and will not be subsequently modified. Consequently, publish-messages are sent from all data-storing nodes in a unidirectional fashion, propagating solely towards the root. Therefore, even if multiple data-storing nodes publish their availability of the same data item, the root node does not have to deal with inconsistent publication information about the data-item, as the immutability condition precludes this scenario. As a result, only the OID-root would contain the most recent version of the object structure. Unfortunately, this limitation forces all search queries to be bottlenecked at the OID-root and prohibits the modification of existing published objects.

Given that the first stage of our research focused on developing a CDN which could serve dynamically-changing application installations to virtual clusters, incorporation of the ChunkCast architecture requires modification to allow for mutable data items. This is primarily realized via the addition of bi-directional publication mechanisms, described more fully in Chapter 3. In addition to the restriction of data immutability, each ChunkCast query is limited to queries for single OID at a time. To provide range query capabilities, the ChunkCast network was additionally augmented by the addition of *Prefix Hash Trees* (PHTs).

2.4 Prefix Hash Trees

A Prefix Hash Tree, or PHT, is a data structure built on top of a DHT, arranged in the form of a retrieval tree, or *trie* [72]. In a PHT, a data attribute (e.g. OS version) is indexed by a binary string of fixed length. Each node is assigned a *prefix,* which indicates its position in the PHT-trie. To function within the underlying DHT, each attribute prefix is treated as a key to be searched and thus has an equivalent DHT-key representation as well. Each non-leaf node in a binary PHT has exactly two children. Only the leaf nodes in the PHT-trie store the indexed data, or keys, each leaf holding a certain number of keys. However; if a leaf is tasked to store more keys than its store capacity, it must perform a *split* and become a *branch node*.

Every leaf node thus maintains a set of traversal links, which enable a search to traverse the tree within a chosen prefix range. An example of a basic PHT trie is shown in Figure 3, with a *range query* illustrated. In essence, a range query allows a querying client to search for a full range of data values, as compared to just a single data-key (such as in a DHT, like Chord). For example, while a Chord query would only be able to cover the query "Is the installation package for application version *X* present?", a PHT-based range query could be formulated to answer the query "find me all versions of the application installation package which fall between versions X_1 and X_2." However, while the PHT provides the required range-query abilities, it does not account for any extended attributes possessed by packaged applications (such as compiled options), known as application-specific packages. For example, two ray-tracing applications with the same version number could have been built with different compilation options specifying different features, such as parallelization or debugging support. Therefore, PVC-DS uses an enhanced two-stage query mechanism to better adapt the system to cluster creation scenarios, where package choice and deployment impact the functionality of the deployed VMs. In this manner, the PHT structure is used to augment a modified ChunkCast overlay to provide support for mutable objects, extended object attributes and range-query capability. Further modifications are made to the metadata structure describing both VM-image and associated applications to suit the range-query functions of the PHT for use in deploying virtual clusters. The details of this adapted range-query mechanism are provided under *Searching* in Section 3.5.2.

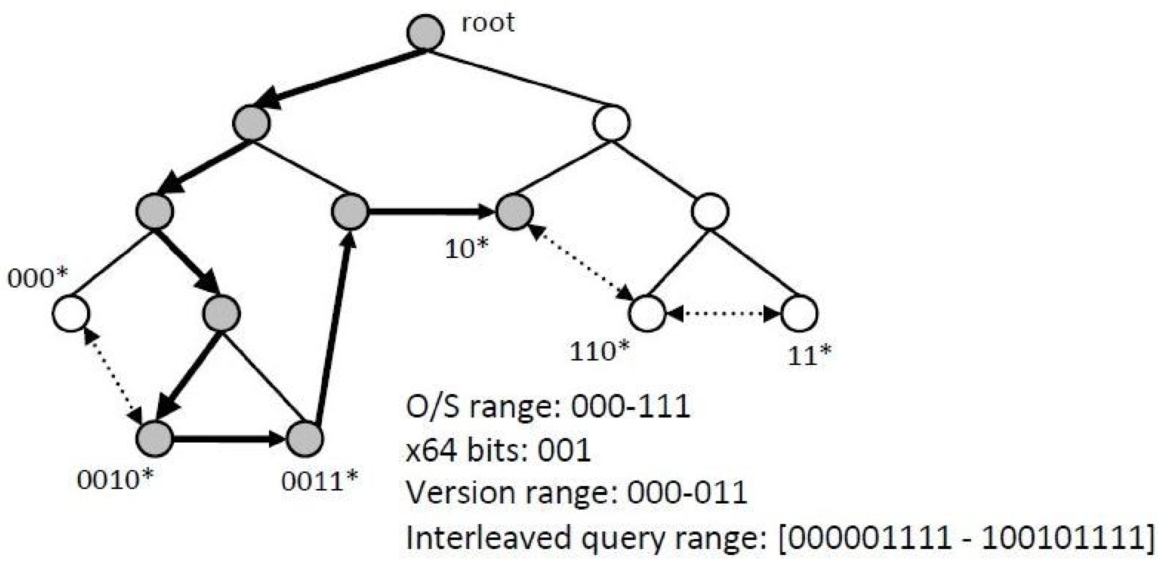


Figure 3. Prefix Hash Tree range query example.

2.5 Mutable Replicated Data

While the adaptations within PVC-DS enhance its capability as a virtual cluster deployment framework, it remains a relatively specialized overlay. Being a purpose-driven design, it provides a specific means to an end—namely, rapid search and dissemination of virtual cluster data—rather than the more general-purpose role of storing information for use in computational clusters. The metadata structure and format within PVC-DS are oriented around VM-image and application data, whereas general purpose computational jobs require a more flexible representation to suit their varied needs (e.g. fluid simulation or statistical analysis data). Moreover, the rather specific requirements of PVC-DS allow it to follow a more relaxed consistency role; if the metadata describing a newly created application package has not fully propagated through the overlay due to publishing delays, it does not invalidate searches conducted during this period. The lack of the updated package description in the search results may simply result in a delay in the deployment of dependent virtual cluster installations, as the deploying client waits for the package to become available. Given that the assembly of a virtual cluster (both VM-images and applications) is a slow process compared to the time taken for the publishing metadata to propagate throughout PVC-DS, the time-bounds and consistency requirements of the publishing and querying are not particularly stringent.

In contrast, within the context of a fluid simulation or similar distributed computation, long inconsistency periods or delays would adversely affect the computation time, or even the correctness of the results. In virtual cluster deployment, the transfer of VM-image and package information may take minutes to complete, while the data items being both read and written to in the course of a fluid simulation may be accessed by multiple processes in rapid succession within periods of only seconds or even fractions of a second. Clearly, a more stringent consistency model is required to facilitate these kinds of computations. Consequently, while PVC-DS provides a useful search capability for established data items, additional functionality is necessary to maintain more strict bounds on mutual consistency. Such functionality is incorporated in the second part of our research. In addition, the consistency framework is designed to accommodate the more generalized data structures and primitives required for general-purpose computation. Before delving into the mechanisms we have developed, we will briefly cover some of the core aspects of data consistency, followed by an examination of the current techniques used to ensure consistency in replicated data.

For the purposes of our research, the operations requested by clients attempting to access shared resources can be described using database terminology as a sequence of *transactions* acting to modify *data*. In database parlance, *data* is considered a set of items which support read or write operations, while *a transaction* is a program or process which applies either read, write or read/write operations to the data items, which transform the data from one state to another. For a data item, a state corresponds to a particular value assigned to it. For example, a transaction T_1 could apply a write operation to integer data items d_1 and d_2, which assigns a value of 5 to item d_1 and a value of 10 to item d_2. In this example, d_1 and d_2 would constitute the *write-set* of T_1 [7]. Similarly, if transaction T_2 accessed d_1 without modifying it, then d_1 would be the *read-set* of T_2. Although this terminology originated in the database sphere, it is applicable to general data storage and manipulation contexts as well.

2.6 Single-copy Serializability

As mentioned in Section 1.4, mutual consistency describes the requirement that multiple transactions that access the same data must agree on the "current value" of the data item [24]. While there are many models for consistency [60], for the purposes of this research we will focus on a strict form of consistency assurance known as *single-copy serializability* (1CS) [8] [24]. A set of a multiple interleaved transactions on a replicated database is said to conform to 1CS requirements, if they result in equivalence of the same set of transactions executing serially (i.e., one at a time) on a single non-replicated copy of the database in the order of their arrival times. In the absence of replicas, there are no synchronization issues to deal with in the single-copy scenario and the arrival times of transactions dictate their execution order. Furthermore, no more than one transaction is attempting to access the database at any point in time, so mutual consistency is guaranteed.

This equivalency can be expressed by the following two requirements or *Conditions*:

1. Operations against replicated data items are ordered as if they were acting on single un-replicated items.

2. Parallel interleaved transactional operations acting on shared data items produce the same final item state as the same operations applied in a serial fashion.

Let us examine these two conditions separately. For a moment, we will assume that Condition 2 is satisfied and focus instead on the requirements for Condition 1. A trivial approach to fulfilling this requirement would be to designate a single node hosting one copy of the data to be a *master* and force any clients attempting to access the data to only use this master node. As all requests are channeled through this single node, they can be ordered according to arrival time, forcing a serial order and fulfilling Condition 1. However, this forces all inbound read/write requests to be funneled through that single node, which in the case of high request rates may become overloaded. Also, a failure of this node would result in the data becoming unavailable, unless some form of failure-recovery algorithm was incorporated into the system. At the other end of the spectrum, we can allow any replica node which is hosting the data item to answer read or write requests. However, allowing multiple nodes to process simultaneous writes would very quickly result in inconsistencies between replicated copies. While these inconsistencies could potentially be resolved, in the face of asynchronous (and potentially unreliable) communications between nodes and no global clock to determine order of arrival of the transactions at each node, determining the order of write operations becomes an extremely difficult problem to solve.

One popular solution to Condition 1 is via the use of a consensus algorithm such as Paxos [50], whereby each write attempt is preceded by all the replica-hosting nodes querying each other to: 1) determine which node should be the designated master for the duration of the request and once chosen, or 2) decide what the new value of the accessed data item should be. Indeed, several DHT-based data storage implementations utilize this technique [5] [40] [61]. While this does provide the necessary fault-tolerance required for the single-master architecture to operate in a fault-tolerant manner it comes with a high overhead cost, as a large number of broadcast operations between replica nodes are required before a write is successfully completed. Given that such lengthy and expensive operations would adversely affect the performance of large distributed computations, this approach would be unsuitable for more computationally time-sensitive or demanding problems. Thus, it would be preferable to avoid the expensive overhead of a consensus algorithm, while still incorporating some form of fault-tolerance in the event of master-replica failure. Our research proposes one such alternative approach to this problem, the particular details of which are discussed in Chapter 4.

Now if we assume that Condition 1 is satisfied instead, we are left with the requirement of serializing data access to a single object. Traditionally, this problem has been handled in shared-data systems by the use of *locking* [6], to ensure mutually exclusive access for each transaction. Using locking, a special flag is associated with each shared data item, which a requesting transaction can set to indicate that it is attempting to modify the item. Any other transactions attempting to access the data item must wait until the initial transaction has completed its task and unsets the flag, indicating that the data item is safely available for access. In this manner, the flag acts as a *synchronization* mechanism, allowing the transactions to access the data in a serial manner and enforcing Condition 2. Note that if multiple transactions are attempting to *read* from a shared item, synchronization is not necessary as the item state remains unchanged throughout the execution of the transactions; synchronization is only necessary if one or more concurrent transactions are attempting to write to the same data item.

2.7 Lock-free Transactions

While traditional locking mechanisms have been well-established as a method of achieving synchronization among multiple transactions, they do exhibit several significant drawbacks. Herlihy et al. [37] list some of the major issues facing conventional algorithms which implement mutual exclusion via locking:

- *Priority inversion:* This occurs when a low-priority process acquires a lock, but is then pre-empted by a high-priority process which requires access to the locked data. For example, a low priority user-level process *A* acquires a lock on a shared file *f*. Subsequently, a high priority kernel-level process *B* is activated, which preempts process *A*. Since process *B* requires access to *f*, which is still locked by the now-preempted process *A, A*'s preemption by *B* effectively prevents it from releasing the lock on *f*. As a result, neither process can make any progress.
- *Convoying:* If a process holding a lock for a data-object crashes or is suddenly halted, it may cause to stall multiple other processes also waiting to acquire the same object. For example, consider processes X_1, X_2 and X_3, all requiring write-access to a file. As process X_1 runs first, it acquires a lock on the file. However, due to a run-time logic error in X_1, it crashes as soon as it has acquired the lock. Once X_1 has crashed, the lock it acquired on the file is never released, which prevents both X_2 and X_3 from making progress.
- *Deadlock:* This occurs when two different processes both require the same set of data-objects in order to proceed, but each process attempts to acquire the objects in a different order. For example, if processes *A* and *B* both require access to objects d_1 and d_2 in order to complete, but *A* acquires object d_1 at the same time that B acquires

object d_2, then neither process can proceed, as A is now waiting for d_2, while B is similarly waiting on d_1.

Examination of these issues reveals a commonality among them: a failure of the concurrent processes to "make progress," brought about by the requirements of maintaining consistency. This progress property, also referred to as *liveness* [49], can be roughly defined as "something good eventually happens," compared to the *safety* requirements of consistency which state in similar terms that "nothing bad happens" [43]. Put another way, given a number of concurrently executing processes accessing one or more shared data items, the concurrency protocol controlling access to the data items exhibits the liveness property if at any point in the execution, at least one of the processes in the system is making progress towards completion. Similarly, the safety property states that at no time during execution will any of the processes encounter any inconsistent data items. Examination of the three aforementioned progress issues demonstrates that possession of the liveness property would in each case preclude them from occurring. For example, if the liveness property existed within the deadlock scenario, then either process A or B would be able to complete and release its lock on the shared data item, allowing the other process to complete successfully.

Therefore the locking protocols, in their enforcement of safety, run the risk of violating the liveness property which leads to non-progress conditions. Addressing these problems is no easy task [20], especially when the liveness requirements are combined with the conditions of distributed and replicated data. The solutions may introduce additional complexity to the parallel code, which makes it more difficult to troubleshoot. This form of concurrency control via locking can be considered *pessimistic*, i.e., that a process assumes that it cannot proceed unless it has obtained access to the required objects. Pessimistic control also becomes a performance hindrance when the number of transactions attempting to access the same data becomes very high [86]. It would be preferable for a distributed database or data storage system to implement some form of concurrency control which incorporates intrinsic liveness properties and so does not suffer from these issues. One such form of concurrency has been developed by Herlihy as an alternative, in the form of *lock-free transactional memory* [37]. While this technique was initially developed for uses in the hardware environment, *non-blocking* software implementations of transactional memory were soon developed as well [86]. Herlihy and Fraser later independently expanded the concept *of Software Transactional Memory* (STM) to include dynamic data structures [33] [39].

These various transactional implementations can be placed into several categories [33], each of which defines what liveness guarantees are made by the implementations that they encompass. In order of increasing liveness guarantees, these categories are as follows:

- *Obstruction-freedom*: If a number of processes or threads are executed in parallel, any single thread is guaranteed to make progress, provided that it does not compete (or *contend*) with any other threads for a shared data item. This liveness condition does not prevent the possibility of *livelock*, where two threads will repeatedly abort each other in their attempts to gain control of a contended resource. Therefore, obstruction-free implementations, such as Herlihy's *Dynamic STM* (DSTM) [39] require an external *contention manager* to deal with livelock occurrences.
- *Lock-freedom*: Lock-freedom includes the same guarantees as obstruction-freedom, but also adds the condition that a collection of parallel processing threads must

make progress as a whole, even if two or more of the threads are in contention with each other for shared resources. Implementation of a system guaranteeing lock-freedom usually entails contending processes "helping" each other to complete execution, in the hope that once the "helped" thread has completed and released its resources, other contending threads will be able to resume progress. As a result, the livelock condition is also precluded. Fraser's *Object-based STM* (OSTM) architecture [33] is an example of a lock-free transactional implementation.

- *Wait-freedom*: The most difficult of all conditions to meet, wait-freedom incorporates all the progress guarantees of lock-freedom, with the added condition that *every* thread in the system must make progress, even in the face of contention for resources. In other words, no process or thread is the system will be starved for resources during any stage of its execution. This is the most difficult of all the progress guarantees to achieve without major impacts in performance. However, recent work by Kogan and Petrank has resulted in efficient implementations of wait-free queues [45].

In contrast to locking protocols, the transactional non-blocking approach is considered to be *optimistic*, where transactions always execute as if they are operating completely isolated from others. Under such an assumption, a transaction will *always* obtain access to the required shared data, regardless of which other processes are also accessing the same data. Note that since every transaction is guaranteed data access they cannot be blocked from progressing, thus fulfilling the non-blocking liveness property. For each transaction, a record is kept of the operations being performed on the shared data object, much like the transactional log of a database [36]. In the event of two transactions simultaneously attempting to write into each other's read or write-sets (thus encountering a contention condition), either transaction can be "rolled back" to reverse any of its effects on the shared data. To facilitate this capability, transactions as described by Herlihy make use of *atomic* instructions [36]. As the name suggests, atomic instructions are designed so that they will either complete without interruption, or will be cancelled without affecting the data they were trying to manipulate. Implementations of these instructions include *Test-and-Set* and *Compare-and-Swap* [38]. When a transaction uses Test-and-Set, a value is written to a data item and its original value is returned to the transaction in a single uninterruptable operation. Similarly a transaction using Compare-and-Swap will supply an item value to compare against the existing data item and if the values of the two items match, swaps the original item value with a new value. Using atomic instructions, transactions are designed so that when they are initiated, only one of two outcomes is possible:

- The operation completes successfully, or *commits.* Any temporary updates to the data during the transaction are made permanent.
- The transaction fails, or *aborts.* Any temporary changes made to the data are discarded, leaving the data unchanged.

Let us examine the method by which these transactions are designed to make use of the atomic instructions in order to facilitate both the safety and liveness properties. For example, the implementation of Shavit & Touitou's software transactional memory [86] in Figure 4 illustrates the design and operation of the data structures.

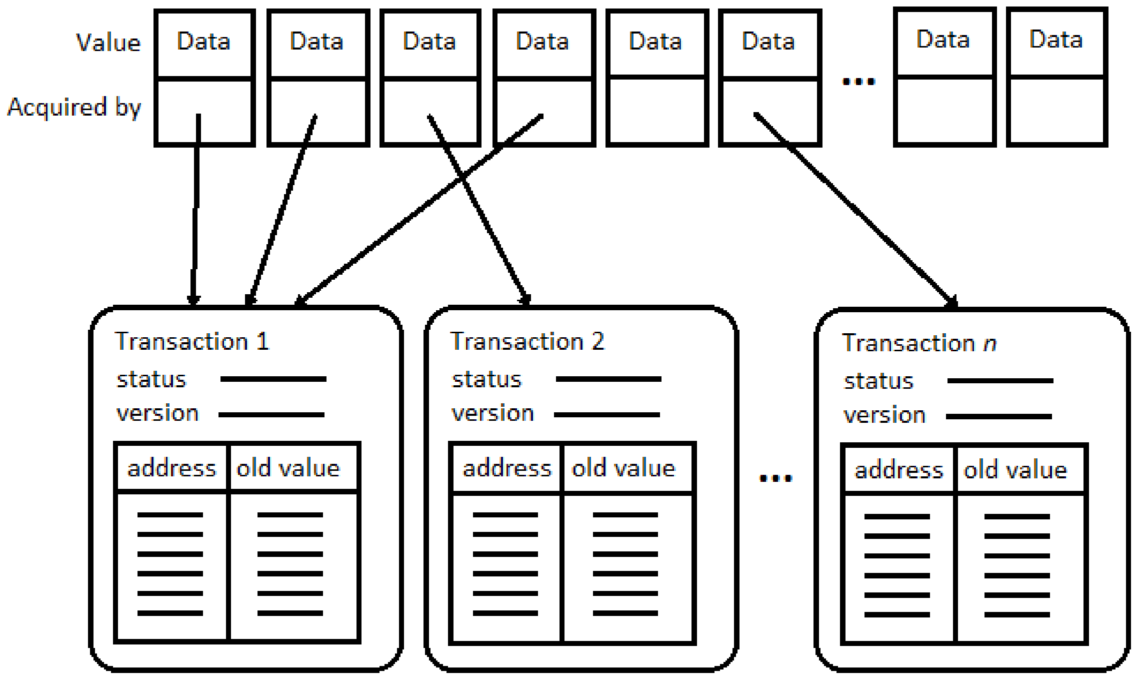


Figure 4. Diagram of STM structures.

In this lock-free implementation, the shared data items are shown in the upper part of Figure 4, while the transactions accessing the data are in the lower part of the diagram. Each transaction attempts to acquire one or more data items for processing, indicated by the arrows. If Transaction *1* acquires a data item for the purpose of updating it (i.e., write instead of read), it atomically copies the original value of the item to a local internal list and creates the acquisition pointer to itself. Once Transaction *1* has completed acquiring all the data items required, it updates the items, then atomically releases ownership of each item. Upon release of all ownerships, Transaction *1* has effectively committed successfully. In the event that one or more data items that Transaction *1* is acquiring have already been acquired by a contending transaction (say, Transaction *2*), then Transaction *1* is forced to abort. It will also atomically release ownership of any items which it has managed to acquire up to that point, marking each of the contending data items with a flag and instructing any other transactions not to attempt to acquire the item. In this manner, Transaction *1* "helps" the contending Transaction *2* by effectively reserving the data item for it. Upon reading the reserved status of the item, any other contending transactions will then abort and release their own acquired data items, even including items which are required by Transaction *2* as well, but which were not already flagged by Transaction *1*. Those contending data items will also be summarily flagged by their respective aborted transactions as reserved for Transaction *2*. Thus, all the contending transactions cooperatively assist Transaction *2* in completing which eventually serves their own interests, since the shared resources become available when Transaction *2* has completed execution.

If contending transactions do not attempt to assist each other, a *contention manager* [83] is needed to prevent livelock situations, as the absence of transaction cooperation could result in contending transactions repeatedly aborting each other in an attempt to obtain each other's resources. For example, consider the deadlock scenario examined earlier, but in the context of transactional execution instead. Two transactions *A* and *B* both require write access to items d_1 and d_2. Transaction *A* attempts to acquire d_1 first while Transaction *B* similarly acquires d_2. *A* then attempts to acquire d_2, but observes that *B* has acquired it already. *A* therefore signals *B* to abort, resulting in item d_2 being freed. Likewise, *B* has detected that *A* has acquired d_1 and aborts it. However, both transactions immediately attempt to acquire items d_1 and d_2 again and the process repeats endlessly.

To prevent this scenario, the contention manager would force either of the two transactions to wait a random period of time before attempting to run again, giving the other transaction time to finish. Alternately the manager could assign a higher priority to one transaction and impose a rule that a lower-priority transaction cannot abort one which has a higher priority. The addition of the contention manager thereby removes the requirement that transactions need to cooperate with each other, thereby simplifying the contention decision-making process, but at the expense of some additional functionality being required by the architecture. Implementing the contention management function brings about an additional advantage, since a manager may be privy to scheduling and resource information which cannot be easily obtained by an individual transaction. This allows for more intelligent and better-informed decisions as to which transactions should be aborted and which should be allowed to complete, i.e., establishing priority of transactions. Furthermore, if the function of the contention manager is made modular, then multiple strategies for contention management can be "plugged in" to the manager to provide optimal contention management, depending on the types of transactions being encountered. This technique has been employed in the *Dynamic STM* (DSTM) obstruction-free architecture developed by Herlihy [39], the *Distributed STM* or DiSTM architecture of Kostelidis et al. [46] and by the closely related *Adaptive STM* architecture created by Marathe et al. [55]. Further analysis and testing of various contention managers interfacing with the DSTM infrastructure was also performed by Scherer et al. [83].

These transactional implementations are considered *eager-acquire* models, where each transaction attempts to obtain exclusive access to the shared data-items very early on, in the transaction's processing flow. Using eager-acquire, any contention issues that a transaction may have with competing transactions are detected and aborted early in the transaction's execution lifetime. In contrast, *lazy-acquire* transactional mechanisms are designed to delay concurrency resolution between conflicting transactions to a much later stage in the transaction's executions, referred to as the *commit phase*. Implementations such the *versioned-boxes* approach by Cachapo et al. [13] and Fernandes et al. [27] follow this alternative approach and will be examined in the following section.

2.8 Replicated Transactional Systems

Although the aforementioned implementations solve the various issues associated with concurrency locking on a single computer, the scenario of running transactions in parallel on multiple computers poses additional challenges and opportunities. Both the eager and lazy acquisition approaches have been used within various distributed systems; however, as our research is in the area of *replicated* data items, we will focus on the prior work within this sphere and thus deal only with implementations which demonstrate specific advantages in replicated-data environments. A replicated transactional system can be divided into two main areas:

1. The previously described transactional system, including the data structure design used internally within a single computer, to ensure consistency among multiple transactions.
2. A *cache-coherence* system [95] to maintain consistency among multiple copies of data items hosted on separate computers.

These two components are direct analogues to the challenges of single-copy consistency, with the transactional system providing the solution to multiple interleaved processes accessing a single copy of a shared data item (Condition 2 of the single-copy serializability or 1CS requirements.) In comparison, the cache-coherence system ensures that multiple replicated copies of a data item act as single copy, even if the copies are distributed among several widely-spaced computers on a network (Condition 1 of *1CS* requirements).

Cache-coherency can be broken down further into two major data-processing components; namely *access* (or read) operations and *update* (or write) operations. As read operations do not by definition change the data item being read, any number of transactions which contain only read operations can be overlapped safely without causing contention. Thus, in the absence of write operations a cache-coherence algorithm merely has to account for any new data items being added, as the data is effectively immutable. The challenge therefore lies in the management of write operations to the replicated data. Consider the following scenario shown in Figure 5.

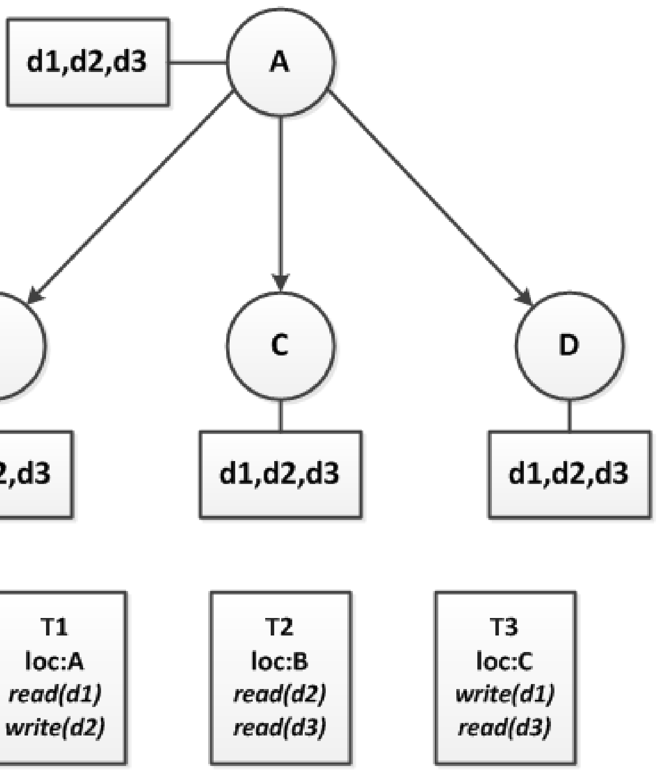


Figure 5. Data replication and transaction processing example.

In this example, data items *d1*, *d2* and *d3* are fully replicated in nodes *A*, *B*, *C* and *D*. Transactions *T1*, *T2* and *T3* are attempting to perform either read or write operations, as indicated in italics within their descriptions. The entry "loc:" refers to the node on which a transaction is executing. We can determine that transactions *T1* and *T2* are in contention with each other, as *T1* is attempting to update value *d2* while *T2* is trying read from it. Similarly, *T1* is in contention with *T3*, as *T3* is updating *d1* while *T1* is attempting to read from it. *T2* and *T3* do not contend, as they have no overlapping write operations.

In order to resolve this contention, we must consider the *order of operations*, i.e., which transaction started first. If we assume that all the nodes in the system have the same synchronized clocks, then we can simply compare the timestamp of each transaction to determine which started first. Even with a non-zero communication delay between nodes, time-stamping each transaction using a synchronized global clock ensures that the transactions are ordered correctly. For example, if *T1* arrived before *T2*, then *T2* would have to wait until *T1* updated *d2*'s value before reading the updated value. Conversely, if *T2* arrived before *T1*, *T1* would have to wait until *T2* finished reading the old value of *d2* before updating it. As transaction T1 originated on node A, the time-stamped transaction would be communicated to the other nodes, where it would be compared to the timestamps of any existing transactions. In the case of node *B* transaction, *T1* would be compared to *T2*, while on node *C* transaction *T1* would be compared to the locally originating transaction *T3*.

However, ensuring that all the nodes in a distributed system have synchronized clocks is not an easy condition to enforce, especially if the nodes are widely spaced on the network. Even assuming that some form of clock synchronization existed, the tolerance required for the relatively short duration of transactional read/write operations [21] [66] [80] would entail that the global clocks be synchronized very frequently, adding a large volume of traffic overhead. Any failure in synchronization communications between nodes could very rapidly create enough time difference between nodes to cause incorrect transaction processing results. Another troublesome issue which may arise in a distributed transactional architecture is the possibility of multiple messages sent from one node to another arriving out of order, or even failing to arrive at all. Out-of-order synchronization messages could cause the clocks on nodes to rapidly fall out of sync with each other. A communications infrastructure which embodies these undesirable characteristics is referred to as an *asynchronous* system [21], codified as having the following properties:

- There is no globally synchronized clock. Each node may have its own local clock, but they are not synchronized with the clocks on other nodes in the system.
- The local clocks on each node may run at different rates, i.e., some clocks may be faster while others are slower.
- Messages sent within the system may be delivered out-of-order, may be arbitrarily delayed, or may not arrive at their intended destination at all (infinite delay.)

While a fully synchronous system would be ideal for a distributed transactional system, a more realistic assumption would be that the average communications infrastructures embody the characteristics listed above, thus categorizing them as asynchronous networks. Given that there is no global clock with which to compare the arrival times of contending transactions, how do we maintain a correct order of operations?

If we assume that any node in a replicated system may process multiple update transactions, then the system would require a communications algorithm which would ensure that any messages broadcast from each of the nodes would be ordered correctly. Algorithms capable of broadcasting in this manner are referred to as *atomic broadcast* or *total order broadcast* [25], with the following properties:

- *Validity*: If a node correctly broadcasts a message, then all the recipients correctly receive the message.
- *Uniform Agreement*: If a node receives a message, then all other intended recipients will eventually receive the message.
- *Uniform Integrity*: Any broadcast message is received by each intended recipient at most once, and only if it was previously broadcast.
- *Uniform Total Order*: If a recipient node receives some message *A* before message *B*, then every other recipient will receive *B* only after message *A* has been received.

Embodying these properties, atomic broadcast systems such as Appia [57] and the Group Communication System [25] can be used to build replicated transactional systems. Therefore, each node atomically broadcasts any updates to its corresponding replicas. The underlying atomic broadcast ensures that any conflicting updates to a data item can be resolved using the order of operations, as the total order broadcast precludes the possibility of any updates being received out-of-order by other nodes. Using the topology displayed in Figure 5, we assume the creation of two new transactions:

- *T4 = {loc(A), write(d1, d2)}*
- *T5 = {loc(B), write(d1,d3)}*

As their descriptions imply, Transaction *T4* is located at node *A* and updates items *d1* and *d2*. Similarly, *T5* is located at node *B* and updates items *d1* and *d3*. Assume that *T4* arrives at node *A* first and the corresponding update is broadcast atomically. Similarly, transaction *T5* arrives at node *B* next and its update is processed. Even if the update messages from *A* and *B* arrive at nodes *C* and *D* out of order, the atomic broadcast algorithm ensures that they are re-ordered correctly, so that *C* and *D* both see the updates in the correct order i.e., from

T4 and then T5. Total order broadcast also ensures that node *B* will also process transaction *T4* before *T5*, even if the communications delay between *A* and *B* causes the *T4* update from *A* to arrive later than the local processing of *T5*. In this case, the local transactional processing system at node *B* would, after determining the correct order of operations, abort *T5* if it had already started and let *T4* run to completion before running *T5*.

While atomic broadcasts ensure update consistency between data replicas, the process of ensuring total order negatively affects performance within the system, as each node must wait for verification of the correct message-delivery order before proceeding. To help alleviate this issue, the concept of *optimistic atomic broadcast* has been developed, using both locking [41] [42] and lock-free implementations [67]. For the purposes of our research, we will not examine the locking implementations and focus instead on the lock-free implementation, which has the potential to solve many of the mutual exclusion problems experienced in transactional processing.

In lock-free implementations, the transactional processing systems on each node optimistically assume that update messages arrive in order; thus processing begins immediately, rather than waiting for confirmation of update order from the atomic broadcasting system. If the messages did indeed arrive in the correct order, then the transactions can commit immediately. Otherwise, the incorrectly ordered transactions are aborted and restarted. Therefore, the stages of transaction processing and update-ordering are effectively overlapped, thus decreasing the time taken for each update to complete and improving overall performance. In addition to their original work within this area, Palmieri et al. [67] subsequently added speculative execution with the goal of increasing this overlap, further improving performance.

However, there is a simpler method of ensuring update consistency, while still enjoying some of the advantages of replication. Referring to the example in Figure 5 again, if we restrict write operations to *only* node *A*, but allow read operations to occur at *any* of nodes in the group, then we can reduce the need for total order broadcast throughout the entire system. Using this method, node *A* would be designated the *master* node and nodes *B*, *C* and *D* would be designated as *slave* or *replica* nodes. Using this alternative arrangement and using the same previously described transactions *T1*, *T2* and *T3* from Figure 5, node *A* would process *T1* locally, while node *C* would send the transaction *T3* information to *A* to be processed directly, as it contains at least one write. However, node *B* would only need to *notify* the master *A* about its transaction *T2*, as it contains no write-operations. The transactions would be ordered simply by their arrival time at *A*, as it is the only node to process data updates. By restricting the updates to a single node (their master node), single-copy serialization is enforced for contending transactions. Time stamping a transaction with the local clock at each replica node is rendered unnecessary, as the asynchronous communication model makes any local timestamps unreliable with respective to other nodes in the replica system. Consequently, only the local clock of *A* is required to order the transactions and maintain single-copy serialization.

Thus, only the master node needs to ensure that the order of its broadcasted updates is the same at all replicas (one-to-many), as opposed to maintaining the order of updates from multiple sources (many-to-many) for conventional coherency enforcement under replication. The next step for maintaining correctness is to ensure that once *A* completes processing the write-transactions, each node correctly receives updated information. More importantly, any updates broadcast out from the master node must conform to the correct order of operations. This step is roughly analogous to single atomic broadcast, whereby the master node must ensure that all the replicas have correctly received the update broadcast before sending any more updates. This basic requirement results in less complex ordering mechanisms, whereby the update ordering from the master node can be accomplished simply by having the master wait for confirmation from each of the replicas, before proceeding with the next update broadcast. Thus any replica nodes which have begun read operations to updated data but have not finished by the time they receive the corresponding update would need to abort and restart the transaction containing the contending read, giving the transaction the opportunity to read the updated data item.

As our proposed research utilizes a replica-aware *versioning* mechanism to enforce transactional consistency (described in Section 4.3.2), we will take a closer look at the *Java Versioned Software Transactional Memory* (JVSTM) concurrency control system by Cachopo et al. [13] [27]. Each shared item in JVSTM keeps a versioned update history, which records any writes made to that shared item, assigning it with a version number. Transactional writes are stored in a linked-list, in the order that they were applied. Each new write is appended to the front (or *head*) of the list, so that a subsequent read to the data-value can be performed with a single reference, i.e., without needing list traversal. To understand how concurrency control functions properly in JVSTM, let us examine each of the transaction types in detail:

- *Read-only transactions:* A read transaction obtains the data value from the latest version in the update-list for the shared item. Subsequent updates to the data-item do not affect the read-transaction, as they append the new values to the list without affecting the list-entry that the read-transaction is using. Therefore a read-only transaction acquires the most up-to-date value and is serialized instantaneously during its beginning phase and subsequently never becomes inconsistent. The read-transaction can then safely commit at any time, as any transactions reading this data-version are isolated from any subsequent changes.

- *Write-only transactions:* As described, a write transaction simply adds the new data-value to the version-list at the end of the transaction's execution, during its commit-phase. Once the new value has been added, the write-transaction has committed. Thus, like the read-transaction, the write-transaction essentially commits instantly and thus never becomes inconsistent.
- *Read/Write transactions:* A read-write transactions is the only transaction type which does not serialize instantaneously, as its read-component acquires shared data-items at the beginning phase of the transaction and commits the writes at the end phase. During the execution of the transaction (i.e., after it has read a data-value but before it has committed), other transactions may apply updates to the same data-value. In this case, when the read-write transaction is ready to commit, it must *validate* any open data-reads, by checking to see if its version number is still at the head of the linked-list of data-versions. If the transaction determines that a data-value it is reading is still at the head of the list, then it can safely assume that no other updating transactions have updated the data value and its consistency state is thereby preserved. The transaction can then apply its writes and commit safely. On the other hand, if the transaction discovers that any data-items it is reading have been updated with new versions, then it must abort, as its read-state has become inconsistent.

In addition to the version-lists for each data-item, JVSTM also maintains a list of *active transactions*, which it uses to track which data-items each transaction has updated, along with which data-item versions are still being read by active transactions. Once it has been determined that a data-version has been superseded by a newer version and has no read-transactions associated with it, the old data-version is deleted, saving storage space.

JVSTM has also been extended for use with replicated architectures, via the addition of atomic broadcast mechanisms [68] [69], to ensure write-order consistency across replica nodes [21] [66]. In comparison, Manassiev et al. [53] used a modified version of the Paxos protocol [50] to enforce consensus among the distributed nodes, to enable a single node to perform transactional updates one at a time and so maintain serialization. Manassiev also proposed a centralized version of this mechanism, whereby all data-item writes are fixed at a single node and read-updates are distributed to replicas by a transaction scheduler. The read-transactions are distributed in a manner to minimize version conflicts, by sending read-only transactions with differing version requirements to different replicas. This concept of centralizing all writes to a single node removes the need for a node which executes update-transactions, to obtain permission from all other nodes before broadcasting the update, a potentially time-consuming process. This is because all the replica nodes in the system assume that only the master-node is authorized to update the shared data-items. Our replica management on transactional updates leads naturally to a similar fashion that updates are carried out at designated nodes (i.e., master copies of update data-sets determined by key-hashing) following DHT-based transaction routing. However, read transactions under our replica management are handled differently, with better performance than achievable via the earlier centralized scheme [53], by avoiding the nodes with master copies of read data sets to schedule and distribute those read transactions.

While our considered replica management provides a much simpler alternative to the relatively complex total order broadcast described a priori, there are still some inherent issues which must be addressed. First, as all write operations are centralized at the master-node, that node will become overloaded if the number of write-transactions becomes very high. Additionally, the master-node introduces a single point of failure within the system, which would render the system unable to process updates in case of master node failure. The first issue can be partially alleviated by increasing the comparison *granularity* of contending transactions. As an illustrative example, consider a shared data-item D being simultaneously accessed by transactions T_1 and T_2. Let us now assume that D can be sub-divided into five smaller sub-items, such that $D = \{d_1, d_2, d_3, d_4, d_5\}$. If we examine T_1 and find that it is attempting to write to sub-items d_1 and d_3, while T_2 is attempting to write to d_4 and d_5, then we can declare that T_1 and T_2 are not in contention with each other. This is possible, since even if T_1 and T_2 may be considered in contention over item D, neither transaction is attempting to modify the sub-items of the other. Depending on the level of granularity chosen, transactions can be grouped into *contention classes*, where a group of transactions in the same class is contending with a common group of specific data items [42]. For this reason, our proposed research adheres to applying versioning to low-level data primitives instead of entire data pages, to maximize data granularity and avoid contention situations wherever possible. As for the second issue of node loss, failure risk of a single master node and/or scheduler can be mitigated by replicating the data on the master to one or more backup nodes and using some form of failure-detection and recovery. This will initiate a switch-over of master operation to a backup node in the event of master failure. Manassiev et al. mentioned this capability in their work [53], but did not elaborate on how the failure mechanism was to be implemented.

To summarize, our research utilizes a software transactional approach, first developed by Herlihy et al. [39] and Shavit et al. [86] to provide consistent access to shared data items. As this research has been designed as an integral component operating within our PVC-DS infrastructure, the transactional components have been specifically tailored to operate within a version of the DHT-based ChunkCast CDN [16]. Following the Chunkcast topology, the transactional components are designed to take advantage of the centralized topology, with the objective of minimizing complexity of write operation contention-management, while maintaining the performance advantages of parallel read operations. As our design includes replication to improve both performance and reliability, we have avoided the additional complications of atomic broadcast mechanisms [41] [68] [69], electing instead to rely on both the read-isolation properties of versioning [13] and the centralized write-control afforded by the PVC-DS topology, to maintain correct read and write ordering. Utilizing centralized contention management also opens the door for more advanced contention-control techniques in the future [39]. We use some of the design elements evident in Manassiev's design [53] and explicitly add a fault-recovery mechanism to address master-node failure. The specific details of this system can be found in Chapter 4, although we will first elaborate on the details behind the PVC-DS infrastructure in the following chapter.

3. Data Dissemination & Search: Peer-to-Peer Virtual Cluster Deployment System

3.1 Motivation

Advancements in virtual machine technology have changed the paradigm of modern high-performance computing or HPC. In place of homogenous fixed-hardware clusters, the advent of hypervisors such as Xen and VMware [4] [62] has allowed the creation of *virtualized clusters* (VCs). These clusters can be created, used and destroyed dynamically, allowing for greater flexibility and customization of HPC clusters and server farms. Groups of machines can be created specifically around a particular role, prebuilt with applications deployed on an instance-by-instance basis. However, one of the hurdles faced by this desirable dynamic cluster installation is the time-consuming process of locating, transferring and deploying the virtual machine (VM) images and application-specific packages needed to kick-start the virtual cluster nodes. In this project, a Peer-to-Peer Virtual Cluster Deployment System (PVC-DS) is proposed as a viable alternative to the traditional centralized VM-repository systems. PVC-DS is an overlay network designed to operate on top of typical DHT-based P2P networks, such as OpenDHT, Pastry, or Chord [77] [82] [89], taking advantage of the inherent fault-tolerant capabilities of DHT networks, while conferring new search and data dissemination features. Building upon the DHT-based ChunkCast [16] content distribution network and implementing prefix hash tables, PVC-DS provides both advanced publish, search and data deployment capabilities for VCs.

3.2 VM Deployment Challenges

Similar to the system outlined by Nishimura [63], PVC-DS takes advantage of the fact that disk space used by a VM can be divided into a bootable base image (termed *VM-image* in PVC-DS) and one or more associated application-specific packages (termed *VM-*

packages). For example, a deployed VM which acts as a network address assignment and naming-service server, could consist of a base OS image (such as 64-bit CentOS 5.5) and the *bind* and *dhcpd* packages. For brevity, we use the term VM-data to collectively refer to both VM-images and VM-packages. As depicted in Figure 7, once a hosting hypervisor acquires all the VM-data (i.e., both the VM-image and the VM-packages), it can mount the VM-image, install the VM-packages, then boot the now fully-equipped VM. Given the multiplicity of available packages in an average OS distribution (e.g., a fully equipped CentOS installation may have more than 1000 separate packages), the task of efficiently publishing and querying the availability of such a large variety of VM-images and VM-packages becomes a daunting task. This task is compounded in difficulty, when VM-data contributing nodes wish to update their existing published packages.

Consider the scenario of a large and widespread developer group, which is performing distributed cross-platform testing of a large application using VMs. The developers would require the hosting CDN to respond rapidly to redeploy their applications, so that the various platform testing VMs can be quickly powered down, rebuilt with the updated applications, and powered back up for the next round of application tests. The PVC-DS developed in this research is intended to achieve efficient and scalable VM-data (sized in the order of gigabytes) distribution, thereby outperforming the conventional technique of sourcing by one single (or small group of) repository or repositories.

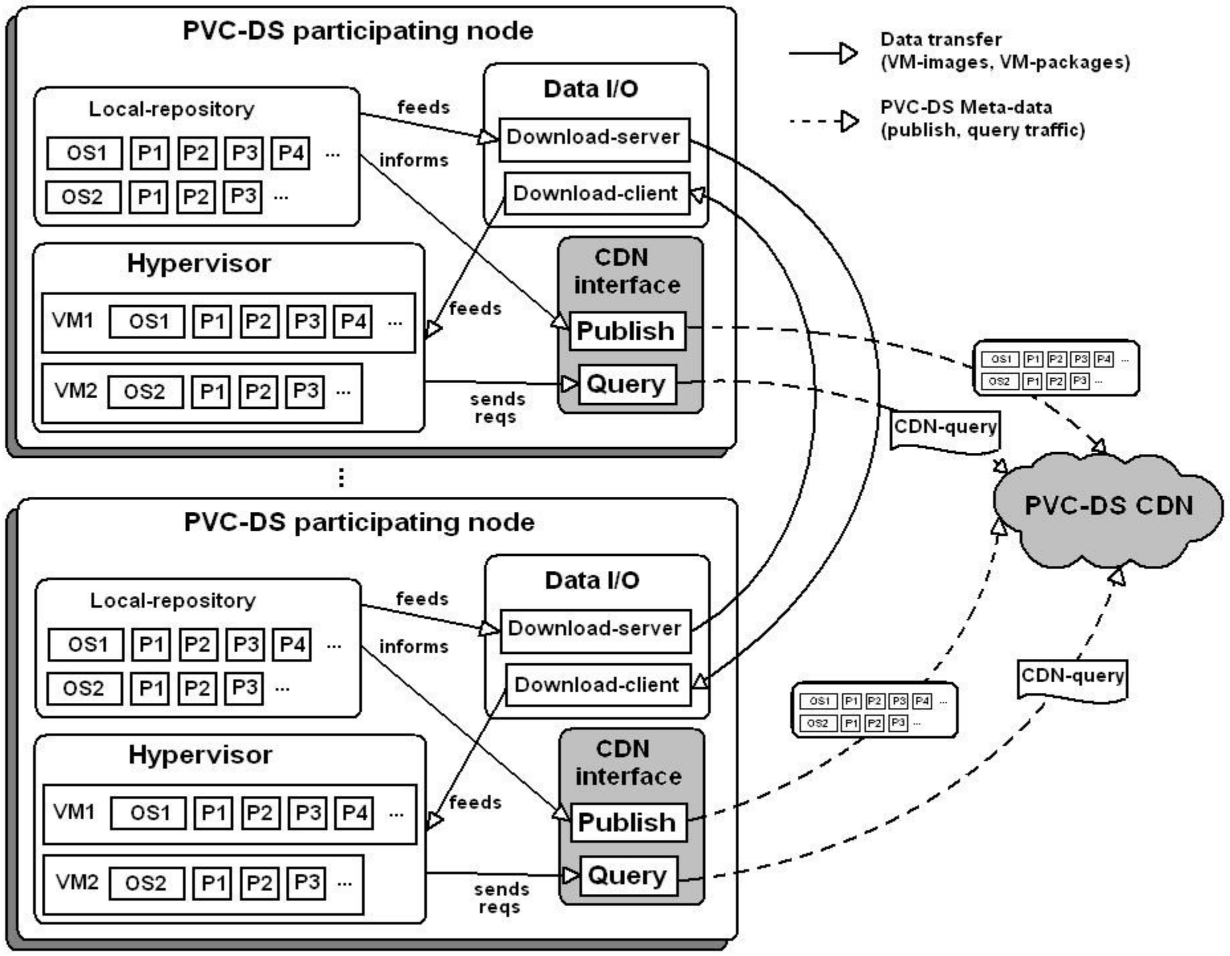


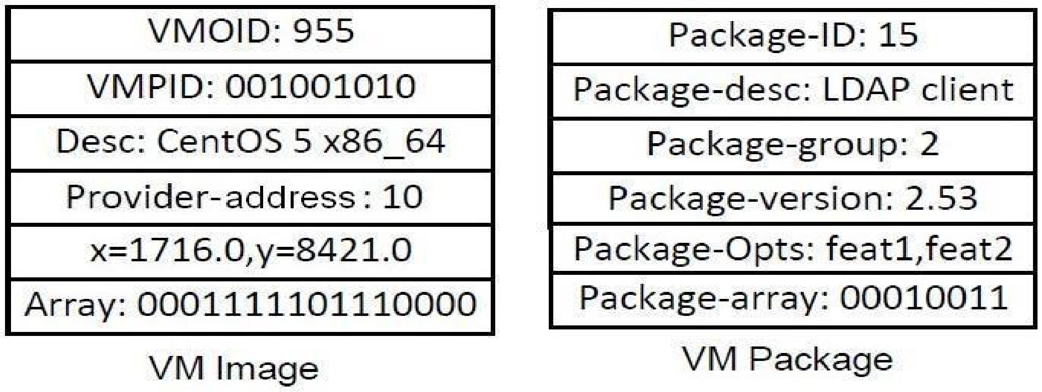
Figure 6. PVC-DS overall layout.

As illustrated in Figure 6 above, in order to fulfill these needs, a PVC-DS consists of two major functions, namely:

- A fully distributed publishing and querying structure for advertising and locating VM-data. This structure, described more fully in Section 3.4, is a distributed architecture built on top of a DHT.
- A sequence of components to process the results of a query, with scheduled download of the actual VM-data from the distributed data-hosting node(s) to the querying client.

The proposed PVC-DS architecture supports distributed publishing of VM information and simultaneous query processing to select favorable downloading sources. This facilitates speedy VM-data transfer from VM-data hosting nodes to VM-data requestors, which initiate CDN queries. Participating nodes also publish their VM-data hosting updates over the PVC-DS CDN accordingly, in support of future CDN queries. VM-data transfer is thus carried out simultaneously over concurrent paths, from multiple hosting nodes to a VM-data requester. This system was chosen, based on path latency and bandwidth amounts which are measured periodically in the Internet, to take into consideration real-world traffic contention and service bandwidth available for hosting nodes. The operational details of publishing and querying are provided respectively in Sections 3.5.1 and 3.5.2., while Section 3.5.3 outlines the decision-making process for constructing the download queues.

3.3 Data Representation



VMOID: 955
VMPID: 001001010
Desc: CentOS 5 x86_64
Provider-address : 10
x=1716.0,y=8421.0
Array: 0001111101110000

VM Image

Package-ID: 15
Package-desc: LDAP client
Package-group: 2
Package-version: 2.53
Package-Opts: feat1,feat2
Package-array: 00010011

VM Package

Figure 7. PVC-DS VM-image/package metadata example.

In PVC-DS, each VM-image is represented by a unique combination of OS type, OS version, and OS architecture. VM-packages are similarly represented, albeit with additional meta-information, to assist in search operations. This representation is termed the *VM-object.* An example of the metadata fields for VM-images and VM-packages is shown in

Figure 7. The *VM Object-Identifier* (VMOID) is an assigned hash for representing the VM-object within the ChunkCast DHT. For simplicity, the proposed evaluation simulator uses a randomly assigned 16-bit string for the VMOID, but a real-world implementation can use a hashing function, such as SHA-1 [28], to provide an adequate non-colliding key-space for the data items. The *VM PHT-Identifier* (VMPID), equivalent to an attribute prefix in a PHT, is a string created by Z-curve "interleaving" of the bit-string representation of the three OS attributes [59].

Attribute	Value Range	Bit Representation
O/S type	8 types	000 - 111
O/S version	6 types	000 - 101
machine architecture	5 types	000 - 100

		O/S		O/S		Interleaved
O/S type	bits	version	bits	arch	bits	key
Redhat	001	2	001	x86	000	000000110
SuSE	010	1	000	PPC	010	000101000
Windows	000	2	001	x86_64	001	000000011

Figure 8. Z-curve linearization example.

In this project, an example of Z-curve interleaving can be seen in Figure 8, illustrating eight different commonly-used OS types (Windows, RedHat, SuSE, Slackware, Fedora, Ubuntu, Debian, and Gentoo), six OS versions (versions 1-6), and five popular server processor architectures (x86, x86_64, PPC, SPARC, and Itanium). Given the three attributes, namely OS type, OS versions and machine architecture, a Z-curve interleaving representation is obtained by interleaving the bit strings which denote the three attribute values. If the O/S type "RedHat" is represented by the sequence "001," the version number

“2” by “001” and the architecture “x86” by “000,” then the interleaved sequence for these bit strings of “000000110” signifies a VM with “RedHat OS, Version 2, on x86 architecture.” Similarly, the VM with “Gentoo Linux 2 for SPARC architecture” is represented by “100101111.”

Each attribute in a VMPID is represented by a string of *log*(*n*) bits, where *n* is the range of an attribute. For attributes to be interleaved, their bit strings must be of the same length, as listed in Figure 8. The bit ranges which have no corresponding attribute values are represented by zeroes. Although these attributes do not necessarily lend themselves well to range queries (OS versions being the exception), they are ordered to provide as much benefit as possible for range queries. For example, the major Linux distributions are grouped together, as are the major Windows releases.

The VM package fields are represented similarly, having both Description and Bit-Array fields. The Package-Identifier denotes a particular package type, while the Package-Group is used to assist in searching, indicating the type of application the package used. Such Package-Version and Package-Options allow a package to change its version and compiled options, while keeping the same Package-ID and Package-Group. Use of both VMOID and VMPID now allows the VM-object to be referenced in both the ChunkCast DHT and the PHT-trie. The Provider-address represents *the Internet Protocol* (IP) address of the node hosting the VM image, while the Chunk-Array is a bit array representation of the available chunks of the actual VM-image data.

3.4 PVC-DS System Architecture

PVC-DS is a 3-layer CDN, based on the ChunkCast [16] distribution scheme. As the VM packages are tied to a particular VMOID, they are similarly tagged with OID metadata.

Like ChunkCast, PVC-DS represents the VM image or package data as an array of bits, with each bit representing a fixed-size chunk of data. From observation of various installed packages, most applications average at least 1MB in size, so the implementation maps one bit to a 1MB chunk. For simplicity, the 1MB-to-1bit mapping is fixed across both VM-images and packages. Alternate implementations may map each bit to smaller chunks, conferring greater granularity of the data representation, but at the cost of higher metadata overhead, since more bits are required to represent a VM image or package. A fully functional VM virtual disk would thus consist of an OID-identified VM-image (e.g. RHEL5, x86) and a number of associated VM-packages.

As mentioned above, the metadata contains other important information, such as the network address, bandwidth and the co-ordinates of the node hosting the VM-data. This information is critical for making good download scheduling decisions. It is important to note that the extended attributes of an associated VM-package may change dynamically over time, as a hosting node may choose to upgrade or recompile a particular package. While these package changes do not alter its VMOID state, the metadata changes may affect subsequent search results. Packages which change in this manner are referred to as *mutated* and must be accounted for in the implementation of PVC-DS. Our evaluation in Section 3.6 thus considers dynamic mutating packages common in practice and compares the results with those under no mutating packages, as found in video streaming, whose video data remain unchanged.

Like ChunkCast, PVC-DS can run on top of most DHTs (e.g. OpenDHT, Pastry or Chord [77] [82] [89]), without any modifications to the underlying DHT. In this project, the test system was built on top of a simulated Chord DHT. As a result, quality is dependent on latency and bandwidth amounts of paths from a client to those VM-data hosting nodes, which are in turn determined by the contention degrees of concurrent queries, which share links of download paths. Thus downloading decisions utilize active probes to a set of 25 VM-hosting nodes over the Internet, to gather bandwidth and latency measurements periodically. This gathered real latency and bandwidth amounts of download paths permitted system performance to be evaluated instrumentally.

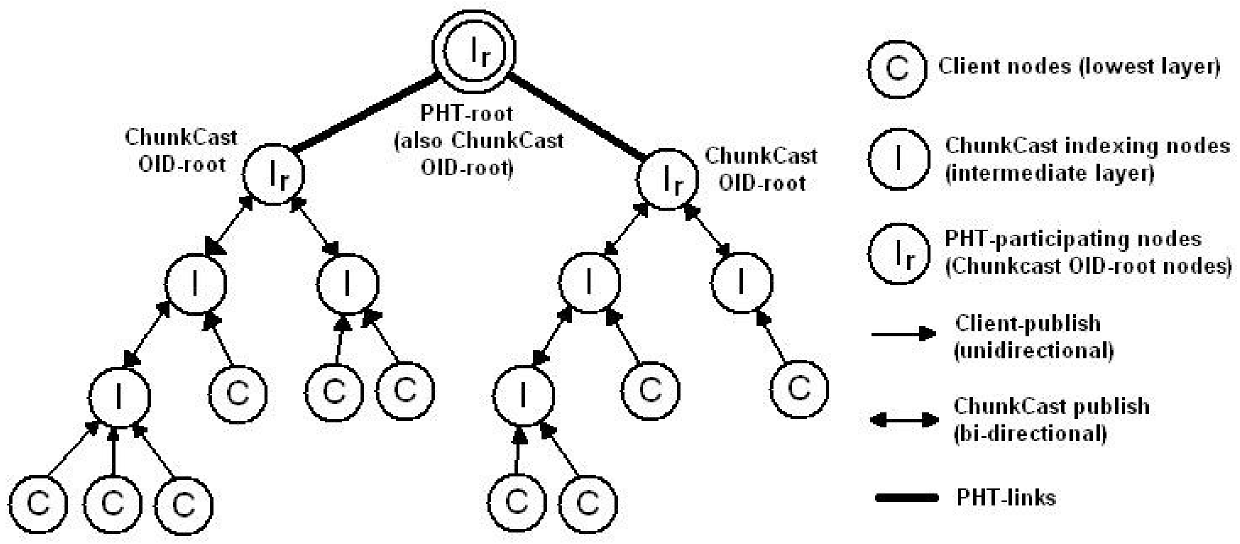

PHT-root
(also ChunkCast OID-root)
ChunkCast OID-root
ChunkCast OID-root
Client nodes (lowest layer)
ChunkCast indexing nodes (intermediate layer)
PHT-participating nodes (Chunkcast OID-root nodes)
Client-publish (unidirectional)
ChunkCast publish (bi-directional)
PHT-links

Figure 9. PVC-DS publishing diagram.

As illustrated in Figure 9, PVC-DS involves three separate levels. Any node may contribute and publish VM-objects to the system, regardless of the level in which it participates. The lowest level contains *client* nodes, indicated by C. These nodes contribute VM-data, but do not participate in the DHT and thus do not index any published information. The nodes are assigned to a single DHT-participating parent (lowest layer), to which they periodically send publish messages containing a list of VM-data which they host. The second (intermediate layer) consists of nodes which participate in the DHT and are termed *index* nodes, indicated by I. These index nodes may have zero or more client nodes as children and participate in one or more ChunkCast trees. The third (top) level comprises solely the OID-root nodes of ChunkCast trees, indicated by $I_{r.}$ To facilitate range queries, a PHT trie overlay is mapped onto these top-level participating nodes. Note that for clarity, the ChunkCast publication trees are shown separated from each other. In practice, an indexing node may participate in multiple ChunkCast trees. However, a ChunkCast indexing node can only act as the OID-root for a single tree. Similarly, a ChunkCast OID-root node may host multiple PHT nodes. The arrowheads in Figure 9 indicate the directions of publish-traffic, as will be explained in the following section.

3.5 PVC-DS System Operation

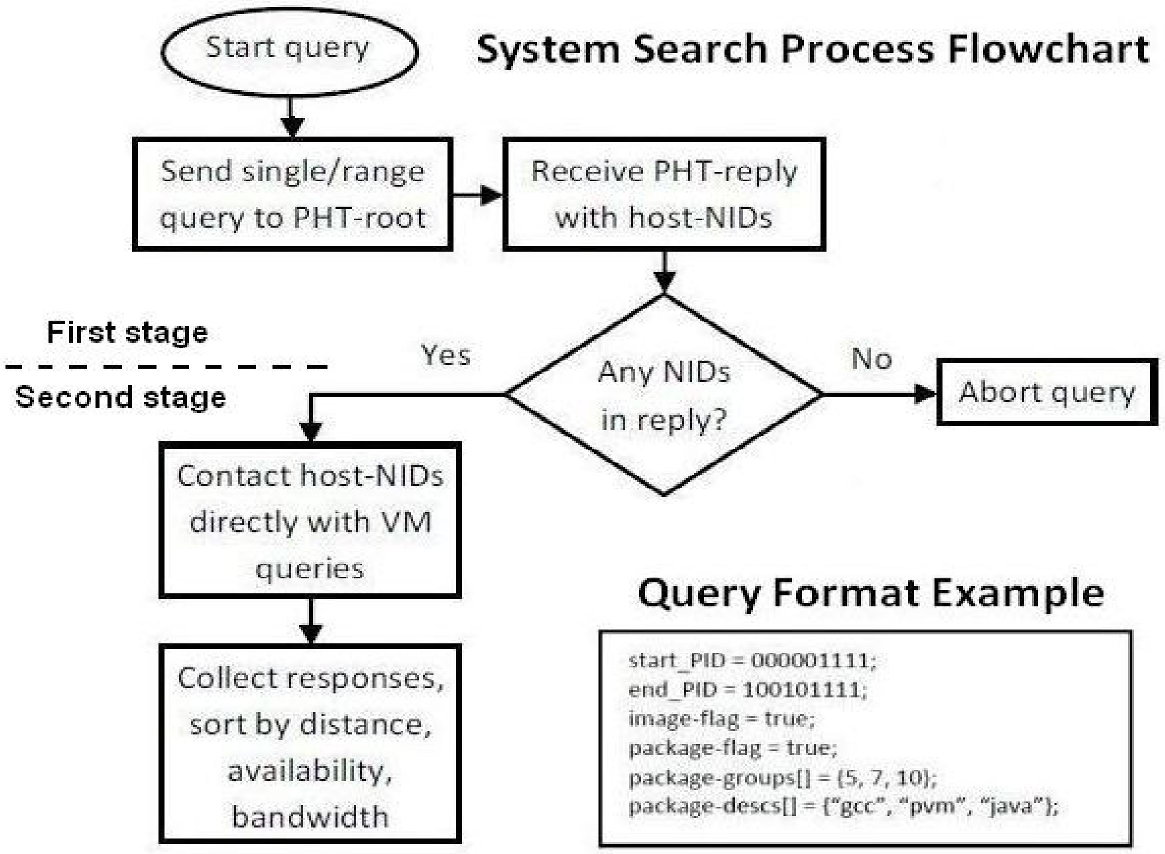
3.5.1 Publishing

Client nodes perform publish operations at fixed 5-minute intervals. They simply send publish-messages directly to their respective parents, but require no publish-replies. Using a similar 5-minute publishing interval, index nodes construct a list of parent nodes to which publish-messages can be sent, based on the VMOIDs of the images in their local VM stores. The index node can then send out a unicast publish-message via the DHT to each receiving parent node, thereby propagating the VM-objects towards the DHT root for each particular VMOID. In this manner, a publish-tree is constructed for each VMOID, using the DHT as the vehicle for VM information to ascend the tree towards the VMOID root. Unlike ChunkCast, for each DHT publish-message, the receiving parent node sends a publish-reply message back to the sending child and the child updates its local VM store with the new VM information. While this causes an increase in publish traffic, the aggregate traffic volume is controllable via the publish update interval and has the effect of causing the contents of a publish tree to be replicated to all participating nodes. This enhancement can allow any changes made to a VM-object (such as the addition of new package-types) to be propagated automatically to all nodes, effectively “cross-pollinating” VM-objects across the individual OID-trees within ChunkCast. This capability is particularly useful when a package undergoes several mutations within a short period of time, as these changes are propagated quickly to all the indexing nodes within the VMOID tree. This brings a particular performance advantage to PVC-DS over the static tree on which ChunkCast is based: since any indexing node in PVC-DS can answer a VMOID query directly for a client node, this reduces the amount of traffic being aggregated at a particular VMOID root, thus lowering the query response time. Moreover, unlike Chunkcast, our proposed PVC-DS is not limited to the storage of immutable-only data and can tolerate changes to the structure (e.g. by adding or removing attributes) of hosted VM-data and propagates this revised information to the rest of the OID tree in a relatively short period of time.

It should be noted that all indexed metadata VM-objects are held in *soft-state*, whereby the failure of a publishing node to send a periodic update to its parent will result in any VM-objects from that child being purged after a timed interval. This ensures that any stale publication information from failed nodes will eventually be expunged from the system, but the occasional dropped publish packet will not result in immediate loss of VM-object publication information from an indexing node. Therefore data in PVC-DS is kept up to date within bounded intervals, but is still relatively tolerant to interruptions in network communications, thus improving the reliability of the system.

3.5.2 Searching

In PVC-DS, a node may search for either a single VM-object *PHT Identifier* (PHTID) or a range of PHTIDs, depending on the desired attributes. An example of an executed range query is given in Figure 10. In the query, a node searches for all VM images which run a 64-bit operating system. In this simplified case, the PHTID bit-pattern corresponding to the machine architecture is fixed, with the bit-pattern for the OS type being allowed to vary from 000 to 111 and the OS version ranging from 000 to 011. Therefore each bit-pattern range is large enough to cover the entire range of attribute values. The three attributes are z-curve interleaved [59] to produce two 12-bit patterns representing the range of PHTIDs. This range information, along with any application-specific package requirements, constitutes a range query. In PVC-DS, the flexibility of the extended package attributes allows for a very large variety of search options. Each VM-package can be identified by its Package-Identifier or description and is classified into exactly one group. Therefore, search criteria can include descriptions and/or package groups. To facilitate these query extensions, the search function of the system operates in two stages, separated by the dotted line, as illustrated in Figure 10.


Start query
System Search Process Flowchart
Send single/range query to PHT-root
Receive PHT-reply with host-NIDs
First stage
Second stage
Yes
Any NIDs in reply?
No
Abort query
Contact host-NIDs directly with VM queries
Collect responses, sort by distance, availability, bandwidth
Query Format Example

```
start_PID = 000001111;
end_PID = 100101111;
image-flag = true;
package-flag = true;
package-groups[] = {5, 7, 10};
package-descs[] = {"gcc", "pvm", "java"};
```

Figure 10. PVC-DS search process and range query format example.

In the first stage, the requesting node interleaves the attributes into either a point or range query and performs a regular PHT-type query against the top layer of the system. If the node is a non-indexing client, the request is proxied through its parent, with the PHT-responses being sent directly back to the client. The requesting node receives the PHT-query responses, containing a list of ChunkCast *Node Identifiers* (NIDs) of VM-data-hosting nodes.

In the second stage of the query, the querying client contacts each node indicated in the results from the PHT-query to request the query-specific VM-objects, as depicted by those below the dotted line. Indexing nodes simply send the requests to their respective OID-roots and collect the responses. As client nodes do not participate in the DHT, they proxy the DHT request to their respective parent, which performs the ChunkCast query on their behalf. Since the queries proxied by the parent contain the network addresses of the client, the indexing nodes can reply directly to the client node, thereby bypassing the proxying parent on reply and thus improve the query response time. These second-stage responses are aggregated to form the full query-response and are sent to the downloading requestor.

3.5.3 Downloading

Once a requester has received the query results from the system, it constructs a series of parallel download queues, based on chunk availability of the hosting nodes and the latency and bandwidth values to those nodes (from the requestor). As VM-data can be very large, the querying requester performs a latency and bandwidth test to each of the prospective VM-data hosting nodes, to determine the priority of hosting-nodes in the download queues. To realize the download path determination properly with real-world Internet traffic contention taken into consideration, a set of live bandwidth metrics was created by actively probing 25 Internet-connected hosts, to obtain 200 bandwidth and latency measurements periodically. These real-time latency and bandwidth measures of download paths were gathered and their performance evaluated instrumentally, driving the inter-node latencies within the simulator. The download procedure described above thus proceeds as follows:

1. For each VMOID, the querying node constructs a list of VM-image and VM-package chunks to download.
2. Using the latency and bandwidth measurements, hosting nodes are chosen as download sources for VM-data chunks.
3. A series of download-queues are created on the querying node for each VM-data host. Downloads are initiated in parallel to maximize performance.

4. Once completing VM-data downloading, the client node updates its local VM-object list with new VM-data information and also publishes its updated VM-data availability.

3.6 Evaluation & Results of PVC-DS Simulations

3.6.1 Simulator Configuration

The PVC-DS was evaluated via an event-driven simulation written in Java, fed with real-time latency and bandwidth values, gathered by periodically probing representative 25 VM-data hosting nodes. The initial network topology was generated with the Inet-3.0 generator [93]. Each topology was generated with 4000 routers and a varying number of client nodes. Using a script from the ModelNet package [94], the weighted topology was converted to an intermediate XML input file, which was used to generate the final network input file for the simulator. The routing tables for each participating node were pre-computed using the Djikstra Shortest-Path-First algorithm. The VM image and package sizes were approximated based on existing minimal VM images for VMWare [62] and Xen [4] images, as well as measurements of installed packages on active file-systems. Each instrumental simulation was run using the following inputs, which were fed to our simulation modules for instrumental evaluation, as depicted in Figure 11:

- A pre-generated network topology and routing graph.
- A list of path latency and bandwidth metrics, generated from real-world periodic probes from client nodes, to various widely spaced data-hosting machines across the Internet. Repeated bandwidth samples were taken from each node, in order to emulate varying network conditions possibly present during download sessions.

- A list of pre-generated VMs and VM-packages randomly assigned to nodes in the simulation. The combination of OS types, OS versions, and machine architectures resulted in 138 different OS types being made available, along with 32 different package types.
- A pre-generated DHT overlay, using Chord [89] for the simulation.
- A query-input file, with a randomly generated mixture of single-OID (or *point*) query and/or *range* queries. The queries contained a mixture of VM-image requests, VM-package requests, or both together in an individual query. All the simulations used the same query input data, regardless of configuration or size.

To simplify message-transfer delay calculations, data fragmentation and reassembly costs were not considered in our evaluation. It was also assumed that the bandwidth of transit-transit and transit-stub connections exceeded the maximum bandwidth of any client-stub connections. The event-driven simulator recorded and aggregated for both publish and query traffic from the system nodes, to determine both per-node and total system traffic for given system sizes. The simulator was also configured to run several comparative configurations, in order to compare the PVC-DS against other known CDN structures, which are described below.

In the first configuration, all the VM images and packages are hosted on a single-repository node, with all searches being conducted at the repository. In this case, the DHT is reduced to a single-node stub, and all stages of the search function occur locally within the repository node. This scenario would most commonly be seen in small single-cluster configurations, such as that used by Foster [32] and Nishimura [63]. In the second configuration, a subset of VM-hosting nodes is grouped together on the network. The DHT-participating index nodes are thus limited to a fixed size, regardless of overall system size. This scenario emulates the configurations used by data warehouses serving large numbers of nodes, where multiple nodes in a single location act in concert to provide VM hosting services in a coordinated fashion. This behavior was achieved in the simulator by running a cluster analysis [81] on the network topology, to locate the closest group of *K* nodes, where *K* is a fixed group size. All the VM images and packages were then assigned to these *K* nodes. It should be noted that for all the tested configurations, all participating nodes cached both VM image and package information. The caching duration of the VM packages was much shorter than that of the VM images, to account for frequently occurring package mutations. The VM images remained unchanged throughout the simulations.

Multiple scenarios were created to test the three configurations by varying several inputs, as follows:

- System size: A total of 10 system sizes were tested, ranging from 100 nodes to 1000 nodes. Each simulation was pre-populated with a fixed set of VMs. For both PVC-DS and the group configuration, the VM-data were distributed randomly among the designated hosting nodes. The number of VMs equaled the number of system nodes, i.e., a 300-node system was assigned 300 VMs.
- Query types: All three configurations were tested with both single-VMOID queries (called *point* queries) and *range* queries. Like the VM-data distribution, the number of queries assigned to each evaluation was matched to the total number of nodes within that particular evaluation run. As such, the total number of queries equaled the number of nodes and the queries were randomly distributed among the nodes within the system.

- Package mutations: Since unpredictability and variance of mutating packages affect query outcomes, each configuration was tested with both static and mutating packages.

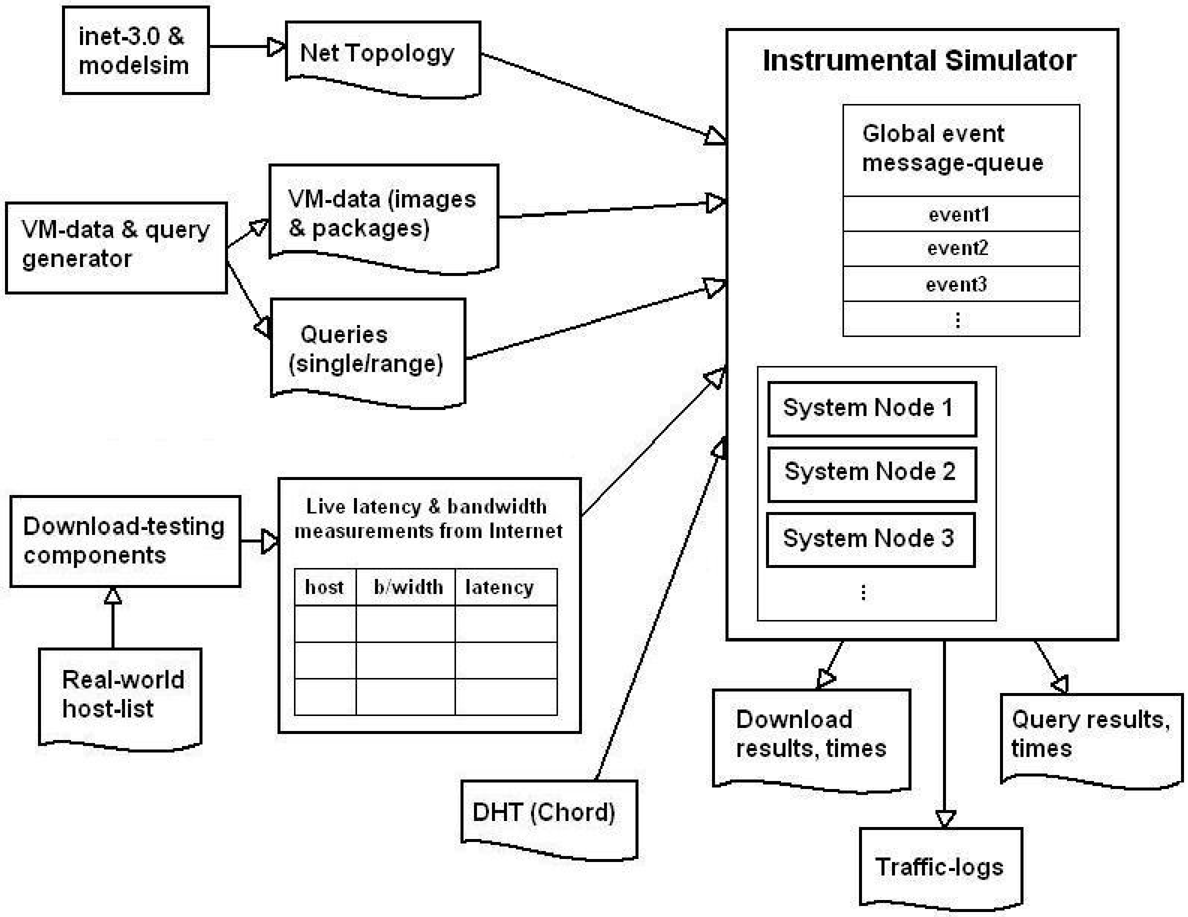


Figure 11. Components of PVC-DS instrumented evaluation.

Upon initialization of each evaluation run, the tested system was allowed to “settle” for 10 minutes before processing any queries, to give the publish-overlay adequate time to be properly established before initiating any queries. As the queries for both PVC-DS and the group repository configurations would potentially involve multiple sources from which to download VM data, upper limits were set on the number of concurrent download channels which each VM-data hosting node could create. Since many configuration parameters exist in each evaluation scenario, some parameter values were fixed to facilitate evaluation and comparison, as listed below:

- The DHT size of PVC-DS was set at 10% of the total system size. This ratio was chosen based on observations of the number of computational clusters and servers relative to the number of data-storage nodes in a medium-sized server room. As PVC-DS is intended to be used to link multiple smaller data-centers and server rooms together, compared to running within the confines of a largely homogenous cluster or within the closely-spaced network environment of a large data-center, this choice of ratio was found to produce overlays which are roughly representative of the intended real-world environments.
- For group-repository evaluation, the group size was set at 4 nodes. Assuming an average VM image with associated packages to be approximately 1GB, each group node in a medium-sized network of 100-150 nodes would host some 250GB of data (assuming that each participating node was hosting 1-2 VMs.) Thus, a collection of 4 nodes would serve 1TB of data, comparable to a small/medium sized CDN repository.
- For all simulations, the delay between queries was set at 2 minutes, with a chunk size of 1MB. This chunk size was chosen based on an average minimum package size for the various OS installations, while the delay was increased or decreased to simulate heavier or lighter workloads, with the publish-intervals varied to match, as described below.
- The publish-interval for each node was set at 2 minutes, with soft-state checking to occur every 5 minutes. This publish-interval was configured based on the delay between queries: if the load on the system was increased by shortening the interval between subsequent queries, the publish-intervals had to be correspondingly shortened as well.

This allowed changes in node repositories to be reflected rapidly enough throughout the system to influence the results of subsequent searches. While this publish interval was varied manually within the simulator, in practice it would be adjusted automatically by each node, in response to varying load conditions. Consequently, provided that the publish-intervals were matched to the system load, the comparative results remained the same, with the side-effect of increased traffic overhead due to the more frequent publish-updates being disseminated by participating nodes.

- Package mutations, if present, occur once in 2 minutes. This is matched to the publish interval, to provide a worst-case scenario for generating update-traffic.

For PVC-DS, once a client node downloaded a VM image or package, it could cache that data, making it available to the rest of the system for a period of time (referred to as the *lifetime* of the data), with larger caching storage at client nodes being required for longer lifetimes. For this study, VM-images had lifetimes of 5-6 minutes, whereas VM packages were retained for 1-2 minutes. While the average time for a single-OID query to complete within any of the tested architectures was under 1400 milliseconds (and under 1800 milliseconds for a range-query), in comparison the time taken to download the actual VM-image and VM-package data was on the order of minutes, as shown in Figure 13. This domination of time by the download rendered the actual query-times as a relatively insignificant component of the total query/download time. Thus, our evaluations focused on the metrics of total download time and traffic overhead imposed by the publishing and querying mechanisms.

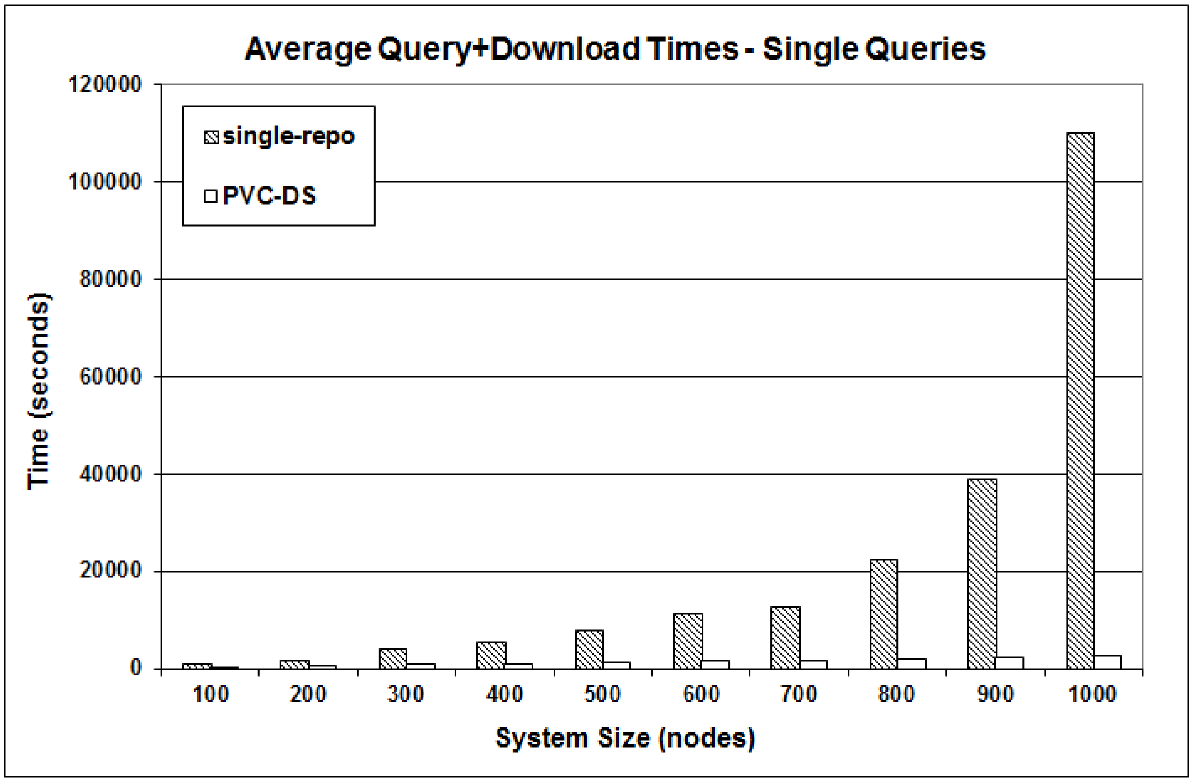


Figure 12. Single-repo and PVC-DS download times under point queries.

3.6.2 Results

Figure 12 illustrates the average time taken for both the single-repository configuration and PVC-DS to perform single-OID (i.e., non-range) queries and download the resulting VM-images and packages. For clarity, the output from the group-repository is not excluded from here and will be examined in Figure 13 instead. As seen in Figure 12, the performance of PVC-DS compared to the single-repository configuration is clearly superior, as the large number of download-requests concentrated at the single-repository results in extremely high congestion and slow download times. In addition, the distributed nature of the VM-data download-sources available to a querying client, compounded with the ability for the client to selectively choose which nodes offer the best performance for downloading, results in far shorter average download-times. The relative time-savings for PVC-DS compared to the single-repository configuration increase as the system size increases, but the average cost savings over the tested system set ranged from a 70% time-savings for the 100-node network to a 95% reduction in download-time for the 1000-node network. Averaged over the 10 tested scenarios, the average time-savings of PVC-DS compared to the single-repository scenario was 85%.

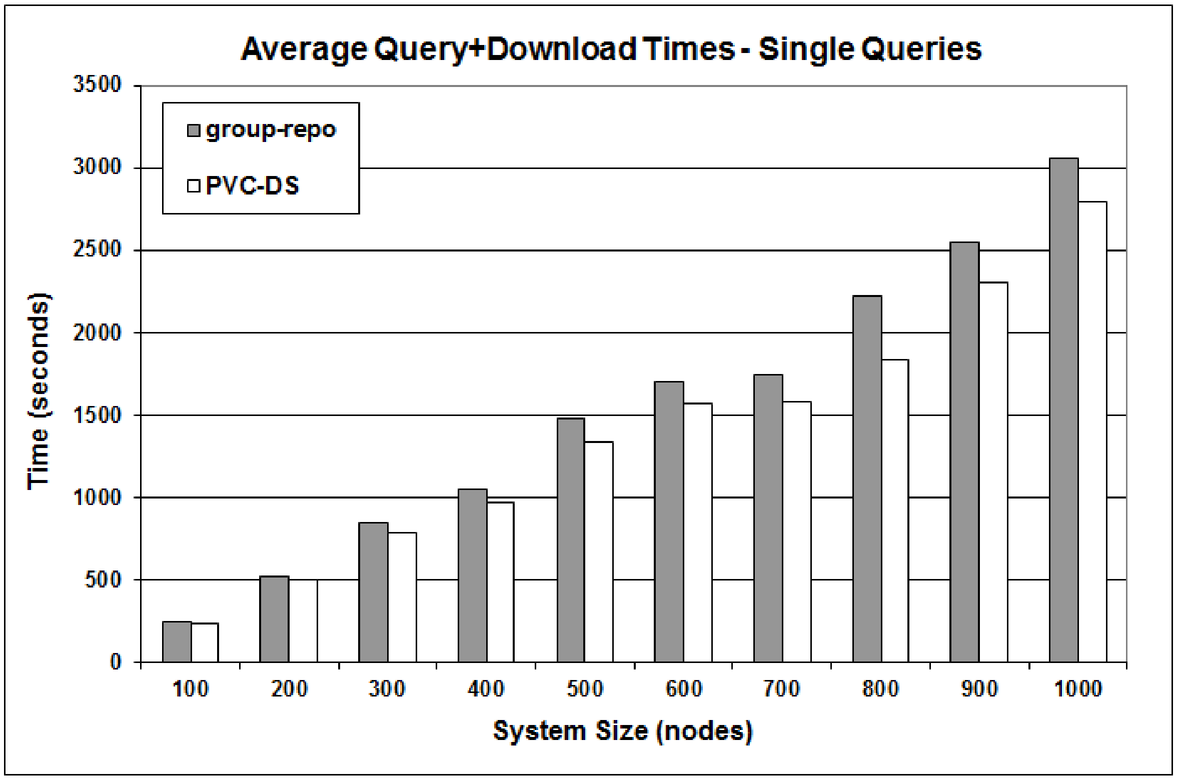


Figure 13. Group-repo and PVC-DS download times under point queries.

Figure 13 shows the mean time taken for both PVC-DS and the group-repo configuration to complete single-OID (i.e., non-range) queries and download the resulting VM-image and package data. In this simulation, the download time involves obtaining both VM-images and VM-packages, with mutations enabled. While both PVC-DS and the group-repository configurations perform in a far more scalable manner when compared to the single-repository scenario, PVC-DS manages to affect better cost-savings than the group-repository configuration. The cost-savings for data-downloads by PVC-DS was calculated to be an 8% improvement over the group-repository scenario.

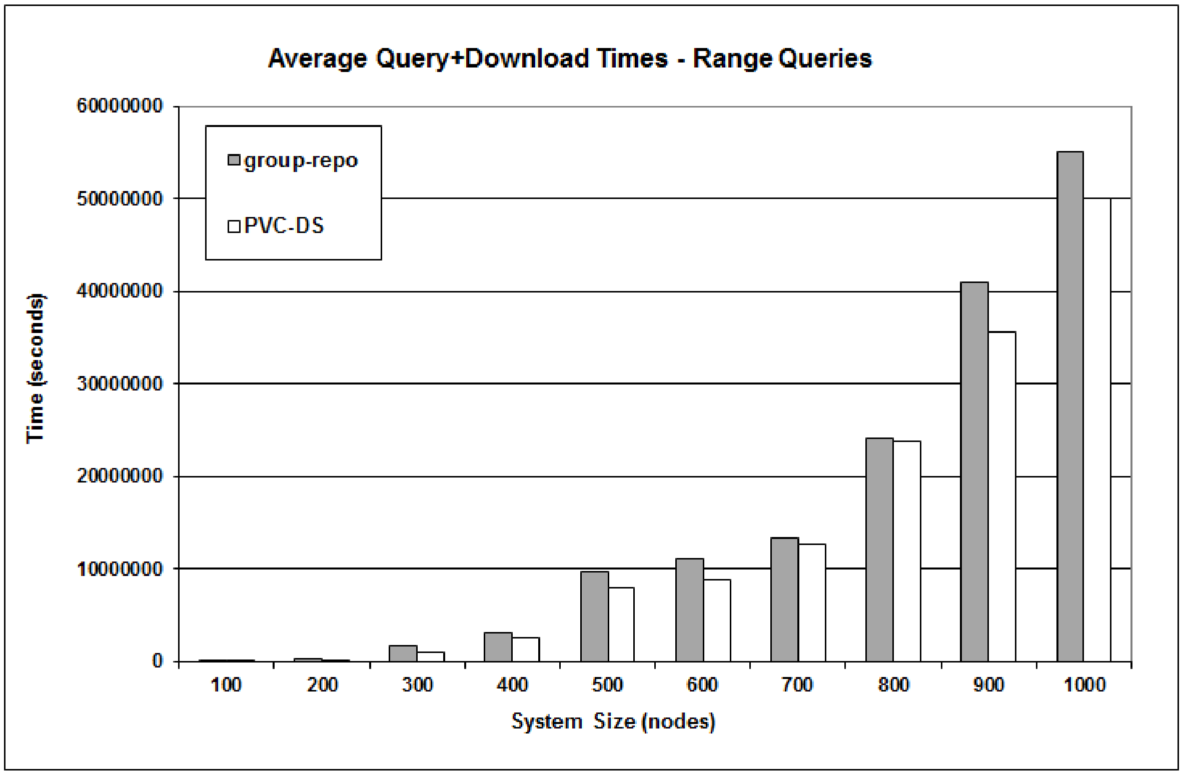


Figure 14. Group-repo and PVC-DS download times under range queries.

VM-data distribution time versus system size under range queries is demonstrated in Figure 14. As the difference in download-times between the single-repository configuration and PVC-DS was so high as to affect the scaling in the graph, the single-repository download-times were not included in the graph. However, when the average download-times for all three configurations over all the tested configurations were calculated, it was discovered that the mean download-times for both the group-repository and PVC-DS were less than 1% of the times required for the single-repository configuration. Also on average, PVC-DS demonstrated an average time reduction of 15% over the group-repository configuration for range-queries. This is due mainly to the wider dispersion of VM-data among the nodes under PVC-DS. This leads to a higher probability of queried VM-data present on nodes with large-bandwidth links, in comparison to the group repository configuration, with a relatively limited number of VM-data hosting nodes.

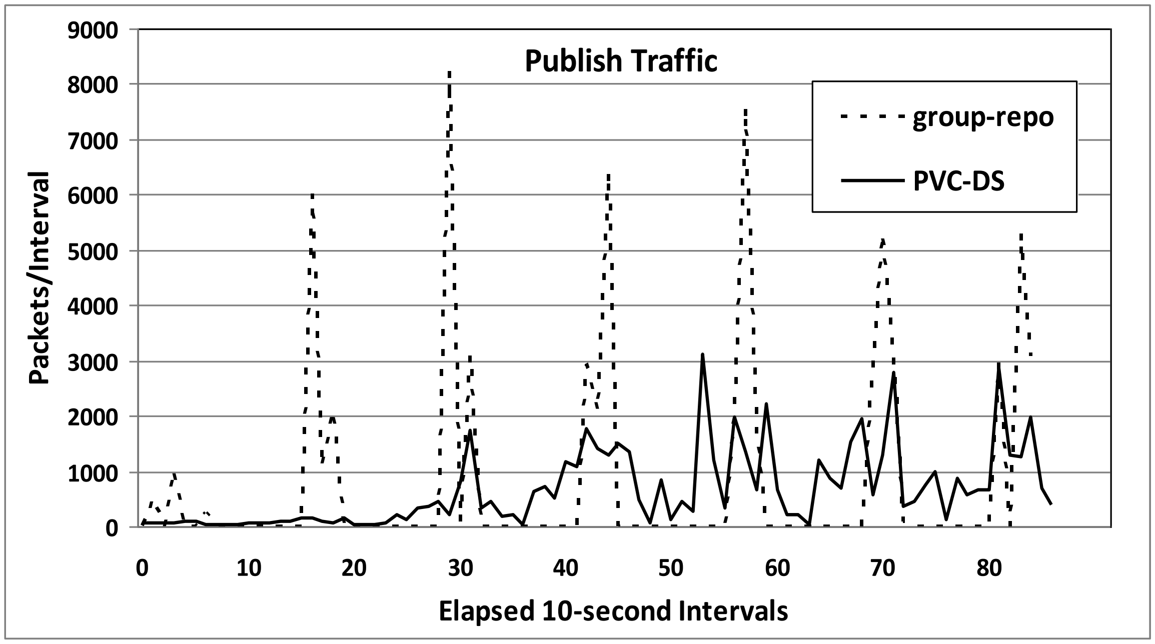


Figure 15. Publish-traffic for 1000 nodes.

Figure 15 depicts traffic overhead (in terms of the number of packets) over the elapsed time interval for publish operations under PVC-DS and the group-repository configuration with 1000 nodes. The results include overhead messages among all the system nodes upon range queries with mutating packages. As all the VM-data in the single-repository configuration are stored on one node, no other nodes in the single-repository configuration hold any VM-data to publish. Since the single repository node does not publish data to itself, the single repository configuration produces no publish-traffic and is hence excluded from the publish-traffic graph.

When averaged over the entire evaluation interval, the publish-traffic overhead of PVC-DS is observed to be about 17% lower than that of its group-repository counterpart, averaged over the entire evaluation interval. Also PVC-DS generated approximately 150-200 publish-messages per second in order to maintain the PHT/DHT structures and to account for any mutated or added packages. However, the publish packet-count for PVC-

DS rarely exceeded 3000 packets per 10-second interval during its publish-periods. In comparison, the group-repository configuration exhibited very large spikes of at least 5000 packets/interval, coinciding with its publish-periods. For a large configuration such as that illustrated, there are many changes in the VM-packages, due to both downloads of new VMs and mutations of existing ones in the repositories. In PVC-DS, these changes are filtered and spread across nodes, thereby lessening their impact at the publish windows.

Figure 16 shows the range-query traffic produced by all three configurations for the 1000-node scenario, where the traffic pattern for the single-repository configuration is very similar to that of the group-repository or PVC-DS. The measured traffic for the single-OID queries follows a very similar profile, but is only about 30% that of the range-queries. That is, the traffic graph for single-OID queries is scaled to 1800 packets/interval, compared to 6000 packets/interval for the range-query traffic. While the query-traffic overhead for PVC-DS is marginally higher than that of the group-repository scenario, we believe that this small increase in query-traffic is compensated for by the reduction in publish-traffic from PVC-DS and is worth the gain in VM-data download performance.

As mentioned earlier, the load on the system was varied via a reduction of the interval between queries. In order for the queries within PVC-DS and the group-repository configurations to receive up-to-date information within the now-compressed timeframes, the publish-intervals for the nodes needed to be reduced correspondingly. Although this measure functioned to keep both configurations competitive compared to the single-repository scenario, the side-effect of this change was a large increase in publish-traffic. However, PVC-DS spreads this traffic among more nodes, instead of concentrating the updates at a small number of nodes, thus lessening the network load felt by individual nodes.

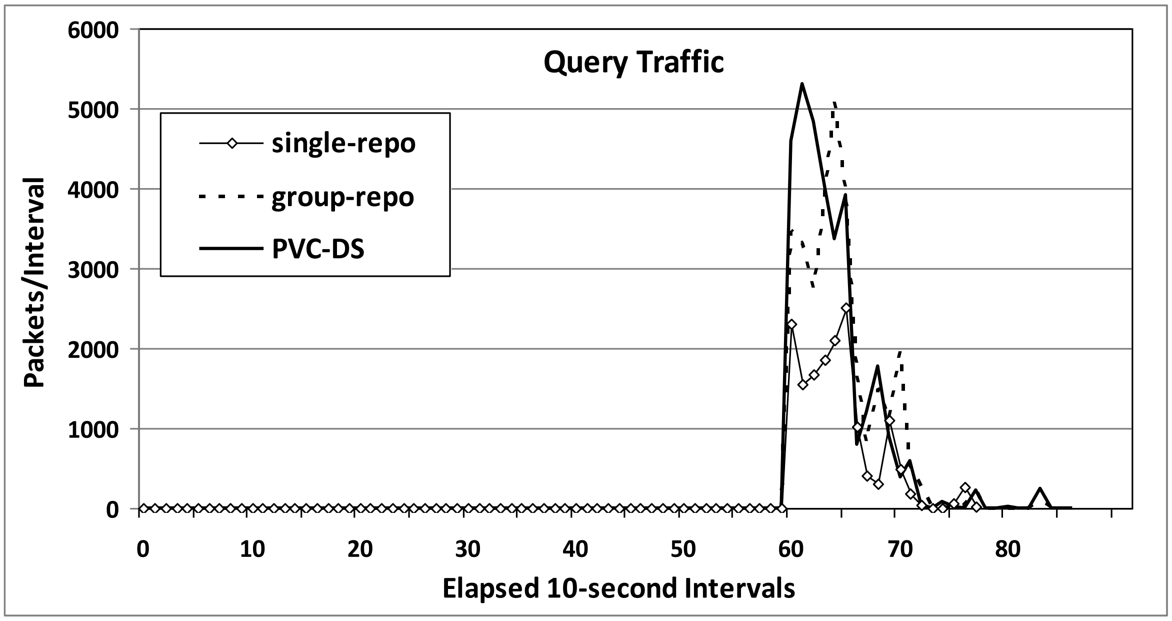


Figure 16. Range-query traffic for 1000 nodes.

Nevertheless, even with the increased download performance afforded by PVC-DS, the average download-time for large system scenarios can be very large compared to the query-time. For example, while the average time for a single-OID query in the 600-node scenario is no more than 1.5 seconds for any tested configuration, the mean download time for that query is approximately 25 minutes. As a querying node can only publish any newly acquired VM-data once its download has completed, at least 25 minutes must elapse before the new VM-data can be published to the rest of the system. Consequently, decreasing the interval between queries also adds additional publish-traffic to both PVC-DS and the group-repository configurations, with little benefit to the download-times. Fortunately, the wider distribution of publish-traffic by PVC-DS means that even in unfavorable load-conditions, the publish-traffic cost is still lower than the group-repository configuration. While the single-repository configuration is relatively unaffected by the load-increase, its overall download-time requirements render it uncompetitive compared either PVC-DS or the group-repository configuration.

3.6.3 Contributions & Conclusions

PVC-DS extends the mechanism of ChunkCast and applies its added capabilities to the search and dissemination of virtual machines. It implements a new multi-layered architecture to provide advanced VM metadata publishing and querying capability not previously possible with ChunkCast. In addition, the modification of the publishing mechanism to include bi-directional communication overcomes the immutability shortcoming of the original ChunkCast design, allowing dynamically-changing published metadata to be effectively disseminated throughout the architecture. The unique two-stage query algorithm complements the new data-format, enabling both range-query capability and just-in-time collection of dynamic application information, thereby assisting in more rapid deployment of VM-images and associated applications to querying clients.

Overall, our PVC-DS has demonstrated a viable VM search and distribution mechanism built on top of a loosely replicated overlay network, with demonstrated advantages in reducing deployment-time for virtual clusters in multiple query scenarios. While there a moderate traffic overhead and query-response cost associated with PVC-DS, the subsequent gains in download-time allow the architecture to compare favorably to existing VM-distribution scenarios. While the other advantages with PVC-DS such as the inherent fault-tolerance of the underlying ChunkCast and Chord networks were not evaluated within the bounds of this dissertation, they serve to reinforce the strengths of replicated data storage architectures, even ones with relatively loose consistency parameters.

Instead, the second half of our research will focus on the application of these replication concepts to storage scenarios with more stringent consistency requirements, and outlines the approach we take to solve the subsequent issues which arise.

4. Transactional Operations on Replicated Objects (TORO)

4.1 Motivation and Assumptions

The second phase of our research next considers adding functionality to the topology underpinning PVC-DS, which was created as a mechanism for the discovery and distribution of virtual machine and related package data. While PVC-DS serves the more focused role of augmenting virtual cluster deployment architectures, this research component augments the virtual clusters created via PVC-DS. This is done by creating a generalized medium upon which applications executing within PVC-DS-deployed computational nodes can store and manipulate VM images and application packages as well as input data. As replication is a key facet of the underlying CDN, this added functionality realized by Transactional Operations on Replicated Objects (dubbed TORO) introduces transactional read/write operations to the existing overlay, allowing data items stored within the system to be both accessed and manipulated in a lock-free manner. Consequently, applications can be rapidly developed and/or deployed within virtual machines using the PVC-DS architecture, whereupon they will have access to a large shared data space for computations. While this has previously only been realized within the realms of large consolidated data centers, the combination of PVC-DS and TORO bring this capability to smaller-tier data centers and server rooms.

As described in Chapter 2, the underlying network infrastructure on which the TORO overlay operates is assumed to be asynchronous, with variable (and possibly infinite) delays between nodes, resulting in possible out-of-order message arrivals. Nodes are expected to have local clocks for timing, but the clocks are not synchronized with each other and may run at variable rates. Therefore, as read or write ordering cannot be guaranteed by message arrival order, TORO must rely on alternative methods for ensuring correct transactional ordering. As the transaction management in TORO is focusing on very fine-grained operations directly against data primitives, the duration of the transactions become much shorter when compared to conventional transactions on replicated databases [66] [80]. Therefore, the performance of individual transactions becomes much more important. Any time advantages to be gained, even small ones, will have a pronounced cumulative effect on the overall performance of the entire system.

Unfortunately, the distributed and replicated nature of the data items complicates these efforts to save time, potentially adding unwanted delays to transaction operations. However, by taking advantage of the existing overlay structure, TORO simplifies the mechanisms required for maintaining coherency between the replicated data items. It also uses the underlying CDN features to assist in the reduction in time required for transaction decision-making. Like comparable replication architectures such as those by Jiang et al. [40] and Etna [61], we assume that node failures in TORO are of the *fail-stop* category, whereby a failed node always restarts in a "clean" state, with no memory of its previous role prior to failure. In other words, the TORO network does not account for Byzantine or malicious node failures.

The overall goal of TORO is to provide:

- replicated and reliable data storage via an underlying DHT-based CDN, and
- high-performance read access via parallel transactional operations, with tunable consistent write access to shared data items.

In order to achieve the goal, TORO involves several separate components, each fulfilling a specific function towards the aforementioned goal. These components are described in the next subsection.

4.2 System Design of TORO

4.2.1 Component Overview

As TORO and PVC-DS were designed to operate as integrated components of a single cohesive architecture, they share major underlying components, specifically the ChunkCast-based CDN [16] and its underlying DHT, which were previously described in both Chapters 2 and 3. Thus, each participating node in the TORO architecture is referenced by a node-key, assigned via the DHT hashing function. As with PVC-DS, any DHT topology can be used for inter-node communication, but for the purposes of simulation, the Chord DHT is used. Like ChunkCast, the topology layout of TORO revolves around the tree-based structure created by the routing of DHT packets towards a closest-prefix destination. The major components of TORO can be seen in Figure 17 below.

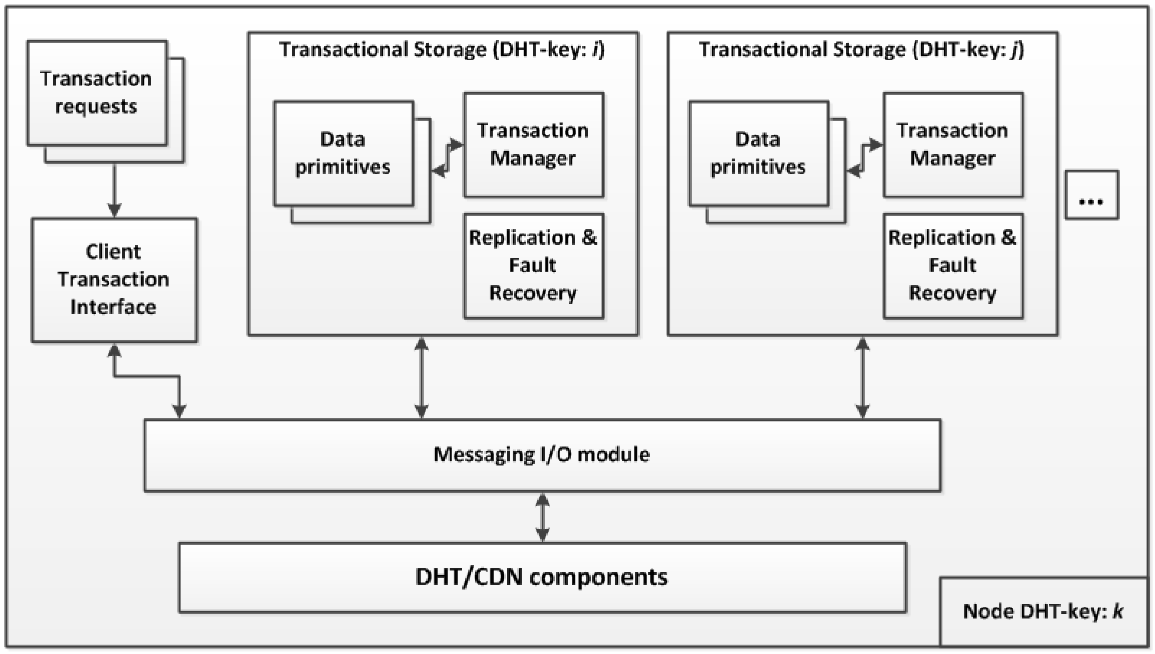

Transaction requests
Client Transaction Interface
Transactional Storage (DHT-key: i)
Data primitives
Transaction Manager
Replication & Fault Recovery
Transactional Storage (DHT-key: j)
Data primitives
Transaction Manager
Replication & Fault Recovery
...
Messaging I/O module
DHT/CDN components
Node DHT-key: k

Figure 17. TORO components within a single node.

The CDN components within each TORO node communicate with the DHT network overlay via the Messaging I/O module. The I/O module forms the link between the DHT network and the inner transactional components of TORO. Inbound read/write requests for shared data items arrive in the form of *Transaction Requests*, which are queued by the *Client Transaction Interface* (CTI). The counterparts to the CTI are the *Transactional Storage* (TS) modules, of which there may be more than one (or none at all) on a single TORO node. Each Transactional Storage or TS module corresponds to a single DHT-key and is referenced accordingly by the CDN components. This is analogous to the VMOID used in PVC-DS, where each data DHT-key in TORO represents a *set* of related data items. In the case of PVC-DS, these data items were the binary chunks which constituted an individual VM image. In contrast, each DHT-referenced set of data consists of individual *data primitives* that are computationally related to one another, i.e., all the items in the set hold some significance to one or more processes. For example, a data-set may consist of all the working inputs and outputs of a single fluid-dynamics simulation. A data-set could also represent all the account holdings of a single customer at a bank, with the corresponding transactions attempting to either read from or modify the customer's financial information. This grouping allows a client transaction to conveniently address an entire set of closely related data with a single DHT-key, rather than a series of individual data-item requests.

It should be noted that although the DHT-key associated with each TS module is generated by the DHT hashing algorithm, one or more TS modules hosted by a TORO module may not necessarily have assigned DHT-keys which are immediately succeeded by the node-key of the hosting node. In other words, while a TORO node may host TS modules whose DHT-keys it immediately succeeds (as per the normal DHT-key storage algorithm described in Chapter 2), the node may also host TS modules whose DHT-keys it does *not* immediately succeed. To explain why a TORO node would do this by design, we must examine the *roles* which a TS module may play. A TS module may operate in one of two roles:

- *Data-set master TS*: These modules are hosting the *master copy* of the data-set. As per the DHT protocol, the TS module for the master copy is stored on a TORO node whose node-key does immediately succeed the DHT-key of the TS module. For example, if the TS module referenced by DHT-key *i* in Figure 17 contains the master copy of a data-set, then the TORO node-key *k* immediately succeeds DHT-key *i*.
- *Data-set replica TS*: These nodes are hosting either entire or partial copies of the data-set, dynamically assigned by the master node. In order to facilitate off-node replication, a TORO node must replicate copies of data items stored in its local TS

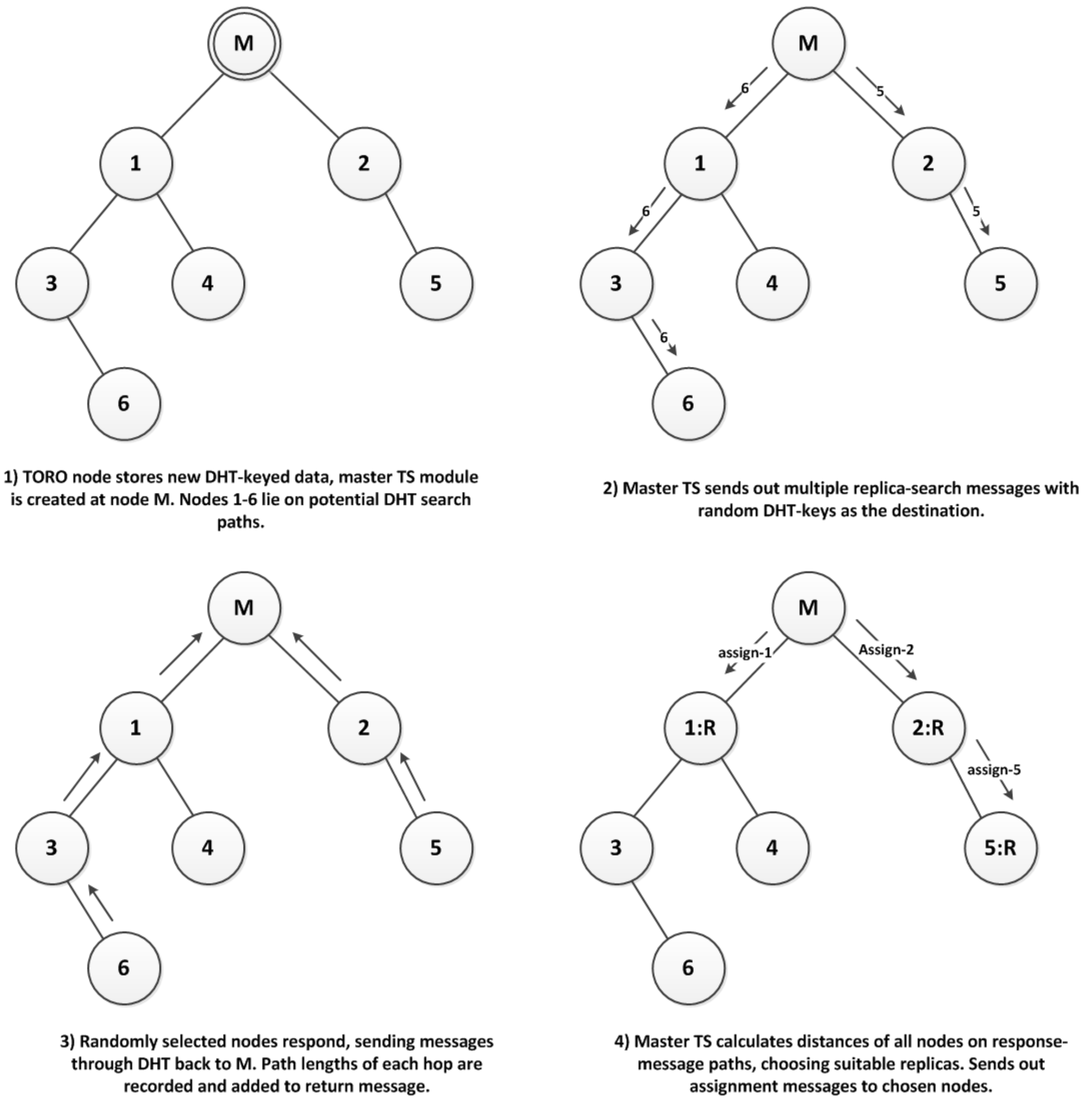
modules to other nodes with correspondingly different node-keys. Therefore, if the TS module *j* in Figure 17 was a designated replica, then *j* would not be immediately succeeded by *k*.

The replication algorithms used in existing DHT-based storage architectures such as FamDHT, Etna and Scalaris [40] [61] [85] are derived from the route taken by DHT messages through the overlay on their way to their addressed destinations and the resulting routing tables created on each node. For example, if a data-storage message with key 959 is being transmitted through a Chord DHT and is routed through nodes 350, 601, then finally node 854 on its way to its destination at node 981, then the first three nodes could be used as replicas for storing DHT-key 959. Therefore, any data-search messages for key 959 would have a high probability of being routed through at least one of the replicas on their way to node 981. As a result, any of the replicas could intercept the data-search messages intended for node 981 and answer them directly, thus improving data-search performance. ChunkCast uses this technique for improving search performance, although ChunkCast replication is more akin to opportunistic soft-state caching, rather than the explicitly assigned replica nodes of Scalaris or Etna. Regardless of how the replication is performed, each of the aforementioned systems is merely choosing the replica nodes, as a result of the routing paths directed by the DHT algorithm.

A drawback to these replication techniques is that since the node-keys for DHT-participating nodes are chosen via a hashing function [28], the route which the replication messages follow is determined by the matching of the DHT-keys with little regard to the actual network path that the messages may take. Therefore, these replication algorithms are said to be *topologically unaware*, which contributes to excessive communications delays between replicas. Using the prior example, if nodes 350, 601 or 854 were network-distant from node 981 then using them as replica-nodes will result in lengthy replication times. This shortcoming becomes a significant problem when combined with the small durations of transactions commands, as the inter-node communications delays are proportionally much larger.

An alternative to these route independent replication techniques is to explicitly assign replicas in a manner which is sensitive to the underlying routing path topology. That is, if replicas could be chosen which only have to traverse short distances to reach each other, then the time taken for data synchronization between nodes would be correspondingly reduced. This is the alternative approach chosen for TORO. A result of this non key-dependent assignment is that any TS modules within a TORO node which are classed as replicas are not located by the DHT-key for their stored data-sets. Instead, they are instead located directly by the master node, using the node-key of the hosting TORO node for that replica. The replica-assignment procedure is covered in Section 4.2.2 and is implemented by the *Replication and Fault Recovery* module in Figure 17.

4.2.2 *Replica Assignment*



1) TORO node stores new DHT-keyed data, master TS module is created at node M. Nodes 1-6 lie on potential DHT search paths.

2) Master TS sends out multiple replica-search messages with random DHT-keys as the destination.

3) Randomly selected nodes respond, sending messages through DHT back to M. Path lengths of each hop are recorded and added to return message.

4) Master TS calculates distances of all nodes on response-message paths, choosing suitable replicas. Sends out assignment messages to chosen nodes.

Figure 18. TORO replica-assignment procedure.

The replica-assignment procedure in TORO follows a 4-step procedure, illustrated by Figure 18. Replica-assignment on a TORO node is triggered whenever a new data-set is created on a node, causing the creation of a new TS module to store the data-items (Step 1). As this is the master-copy of the data-set, the node-key for the hosting TORO node immediately succeeds that of the new TS module. In Step 2, the TS module begins by creating a number of *Replica-Discovery* messages, each of which has a randomly generated DHT-key as the destination. These discovery messages are routed to their destination TORO nodes via the DHT. When a discovery message arrives at a destination node, it creates a *Replica-Reply* message tagged with the DHT-key of the originating TS module and sends back through the DHT network (Step 3). As these reply-messages traverse the DHT, each intermediate node adds a record of itself to the reply-message along with the network latency of the intermediate hop to the intermediate node. Once all the reply messages have been received by the master-copy TS module, it can construct a candidate list of nodes with which it can choose viable replica nodes (Step 4). In the example in Figure 18, the path-information gleaned by *M* would be:

- $M \rightarrow 1 \rightarrow 3 \rightarrow 6$
- $M \rightarrow 2 \rightarrow 5$

Each hop (represented by the "→" symbol) in the DHT-routed path has a recorded latency attached to it. From these discovery responses, *M* can determine the aggregate distances to nodes *1, 2, 3, 5,* and *6*. *M* may repeat this discovery process as many times as needed to procure a sufficiently large pool of candidate nodes, depending on the number of replicas *M* is trying to create for itself. The discovery process can even be repeated, in order to discover as many nodes as possible whose distances from *M* fall under a specified latency threshold. In this manner, the distances between *M* and its replicas are no longer a random choice, dependent on the matched prefix of the destination node, but one which takes the inter-node latency into account. This provides a significant advantage, given the time-sensitive operations of the transactional components of TORO. In Step 4, *M* has chosen nodes *1, 2* and *5* as viable replicas and sends *Replica*-Assignment messages to each node.

However, there are a few issues which must be managed using this technique. As new nodes enter or exit the system, the aggregate latency to each replica may change, as the new nodes are inserted or removed along the DHT-routed path between *M* and its replicas. In TORO, *M* can pro-actively take action, by periodically polling the network with Replica-Discovery messages to discover any new paths and potentially more viable replicas to co-opt for its own use. If a candidate node with better communications latency than any of the existing replicas is discovered, *M* can assign it to its replica-group and simultaneously drop the slowest replica from its group. *M* can also increase or reduce the size of its replica group, as client demands or network conditions dictate. Therefore, the quality of network latency can be monitored and compensated for by the master-copy TS module, thereby providing a superior replica environment for time-sensitive transactions to operate within. The other major issue to deal with is fault-tolerance, covered in the next sub-section.

4.2.3 Fault-Tolerance

Like the eventual consistency mechanism used by PVC-DS, TORO master-nodes use soft-state registration by their respective replicas to determine if there have been any node failures. Each member within the group of replicas periodically sends a keep-alive message to the master-node; this also serves to update the latency information between the replica and master. This latency information is compiled at the master-node along with any new replica-candidates discovered via periodic polling by the master-node itself. The actively ongoing nature of the replica-assignment algorithm ensures that in the event of a replica failure, the master-copy transaction storage module has a ready supply of viable replacement candidates with which to create new replicas. As a master-node can forcibly demote a replica and remove it from the group for performance purposes, any nodes which "fail" due to temporary network failures (as compared to node crashes) can be signaled to remove themselves from the replica-group upon the restoration of their network links. This prevents the rejoining node from attempting to send updates to a replica-group to which it no longer belongs, effectively enforcing the *fail-stop* conditions of the network.

However, a more problematic situation must still be resolved: what happens if the TORO node hosting the master-copy TS module fails? This problem can be broken down into two basic stages: detection and recovery. To facilitate the fault-detection component, each replica-node periodically sends "heartbeat check" messages to the master-node, which fulfils two purposes. First, the master-node sends a response to each received heartbeat-message, acknowledging the continued existence of the replica-node in the group. Thus, this bidirectional messaging ensures that each node in the replica group is aware of the master and vice-versa. Second, as the heartbeat messages are routed through the DHT from the replica nodes to the master-node, they store the latency information of the intervening hops. Thus, the health of both the replica nodes and the connecting DHT network is monitored regularly, which the master-node can use to determine whether or not to add or remove replicas from the group. Each replica-node has an onboard clock which monitors the time taken for each heartbeat-reply to be sent from the master to itself—effectively a round-trip timer. It can compare the time taken for each heartbeat against an averaged timeout of heartbeat-response times, which is updated with each heartbeat. If the replica does not receive a heartbeat-response within the expected timeout, it can assume that the master-node has failed and can begin recovery procedures.

The second component of fault-tolerance is the recovery phase. While the detection mechanism allows the replicas to determine whether or not the master-node has unexpectedly failed, it can only detect the issue, but not resolve it. To assist in the resolution, the structure of the replicas is augmented with the notion of *ranked recovery nodes*. Using ranked recovery, during the replica-assignment phase of operation, the master-node assigns a *recovery rank* to each replica, from the highest to lowest. For example, using the replica group in Figure 18, the master-node can recovery-rank the nodes in order *1, 2, 5* (with *1* being the highest rank and *5* being the lowest). This recovery rank information is disseminated to all the replica nodes piggy-backed on the heartbeat-response messages sent from the master-node to the replicas. The mechanism for recovery ranking operates as follows: if a replica-node suspects that its master has failed, it sends out a *Master-Rebuild* signal to the next highest-ranking replica in the group.

Continuing the example given above, if replica *5* suspected a failure of the master-node *M*, it would send the Master-Rebuild signal to node *2*. If node *2* has also detected the failure of *M*, it acknowledges the message sent from *5* and forwards the Master-Rebuild message up to the next-highest rank (node *1*). Node *1*, being the highest-ranking recovery node, has the final decision on whether or not *M* has really failed. If Node *1* determines that M has indeed failed, then it constructs a Master-Rebuild message which includes a copy of the items in the data-set and is tagged with the DHT-key of the data-set as its destination. Assuming that the integrated fault-recovery features of the underlying DHT are operational, the Master-Rebuild message sent out by node *1* should be routed to the TORO node in the network whose node-key is the next-closest prefix-match to the DHT-key for the data-set.

Upon receiving the Master-Rebuild message, the TORO node assumes responsibility as the host for the new master-copy Transactional Storage (TS) module. It creates the TS module anew and copies the data-items in the Master-Rebuild message into the TS module. If the new master-node does not have a full copy of the entire data-set, it can query the other nodes in the original replica-group to obtain the requisite missing data-items. At this point, the new master-node has no replicas of its own, so it begins a replica-search for new candidates. While it is conducting its search, it can notify the old replica nodes that the master-node recovery process is complete. The old replicas can then demote themselves from the now-defunct replica group, freeing the space taken up by the replica TS modules. Alternately, the master can be configured to wait until it has chosen the new replicas before demoting the old ones. If any members of the original replica group also turn out to be viable candidates for the new replica-group, this strategy saves the master-node the time and bandwidth needed to resynchronize with all the replicas, as they will still have the data-items from the original group stored in their TS module. These nodes would only have to reconfigure their TS modules to point at the new master-node instead, thus reducing both the synchronization traffic and recovery delay.

4.2.4 Data Structures

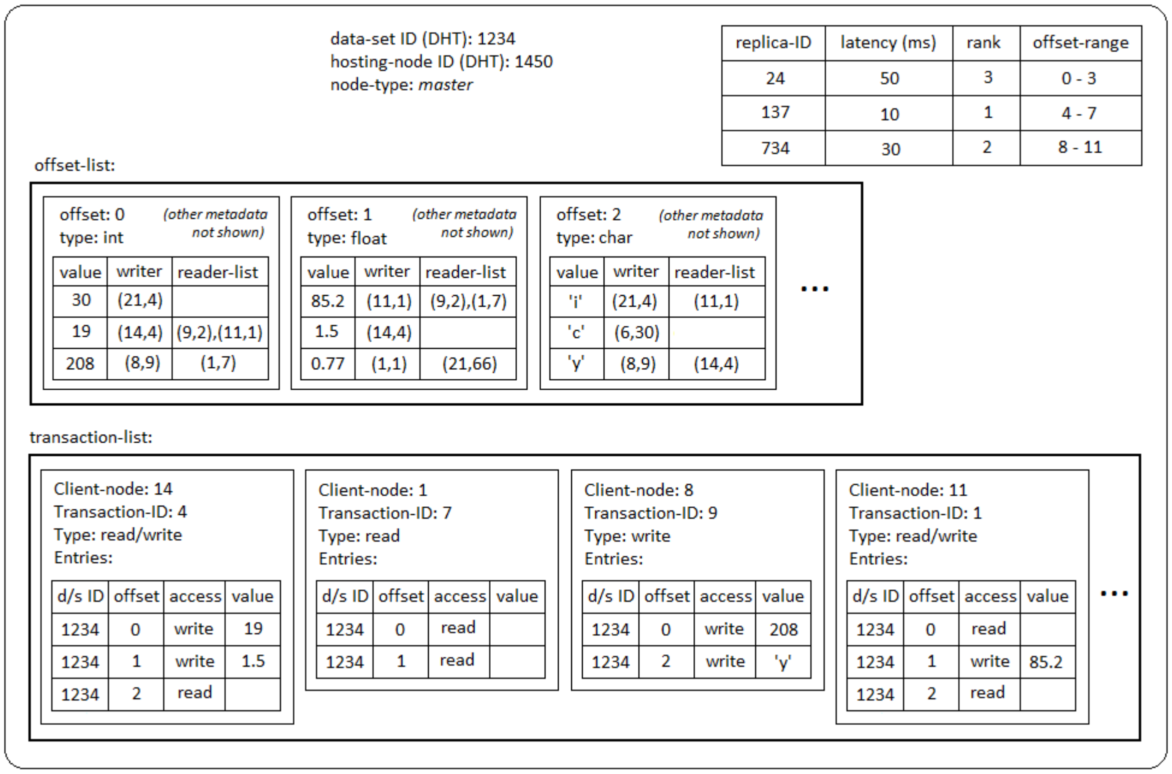
We will now provide a more precise description of the data structures used within each Transactional Storage module and examine how they facilitate transactional operations on the stored data-items. As described, the sets of data-items stored within TORO are located using DHT-keys. Like the VM-info objects in PVC-DS which represent an array of operating system data-chunks, the data-items in TORO are grouped into *data*-sets referenced by a DHT-key, hereby referred to as the *Data-Set-Identifier.*

As the aim of TORO is to provide transactional support for very large data sets, full data replication may become very costly in both time and network resources. To reduce such costs, data to be stored in the TORO network were sub-divided into individual "chunks," referred to as *data-objects,* as outlined in PVC-DS. This sub-division into data-objects enables partial replication of large sets and reduces traffic overhead. Each data-object in the system represents a single typed data primitive (such as a string or integer value), but the data-object can be used to represent items such as binary data-blobs as well, at the cost of transactional granularity (as described in Chapter 3.) The data-object consists of a wrapper around the value containing the necessary meta-information required for both replication and transactional operations, such as the current transactional state. Each data-object contains the following fields:

- *dataSetID:* A DHT data-key which identifies the data-set to which the data-object belongs.
- *dataOffset:* The index into the data-set at which the data-object is stored. It is analogous to a memory offset or array index.
- *dataType:* The type of data primitive being stored within the object (e.g. integer, float, etc.)
- *dataValues:* The update history of the data-object, ordered from newest updates to oldest, e.g. for a data-object with 2 versions, the item at index 1 represents an earlier version, while the later version is stored at index 0. Each item in the list consists of the following items:
    - *dataValue:* The actual value of the data primitive.

- *dataWriter:* The identifier of the node/transaction which applied this update. This field is in the form of the tuple *{writerNode,writerTrans}*, indicating the originating node and transaction number for that writer.
- *currentReaders:* A list of read-transactions which are currently accessing this data-object. Similar to the *dataWriter* field, each *currentReader* entry consists of the tuple *{readerNode,readerTrans}*. If this list is empty and there exists another *dataValue* newer than this one, i.e., this entry is no longer at the top of the list, then this *dataValue* entry is safe to delete to save storage space (see Section 4.3).

These fields provide the necessary structures to implement versioned transactional memory storage, similar to the *Java Versioned Software Transactional Memory* (JVSTM) implementation [27]. We have elected to modify the internal data structures from that of the original JVSTM implementation, by adding reader-lists to each data-item version; in doing so, we no longer need to maintain a separate data structure for the active transactions, as each data-object now carries enough information internally for TORO to determine when it needs to remove outdated data-item versions. Each data-object in in the list is uniquely located via the *{dataSetID, dataOffset}* tuple, in a similar manner to the multi-key location tuple employed by PVC-DS. As the data-objects in our data-set implementation are represented via a vector-list as opposed to a simple array, data-objects which are absent from the set do not consume any extra storage. This minimizes the volume of data used to store a data-set and reduces the time and/or bandwidth required to transfer the data-set between nodes. While the absolute size of the data-set is essentially fixed to simplify replica offset assignments, it would be relatively straightforward to allow dynamically variable-sized data-sets, at the cost of some complexity of replication.


data-set ID (DHT): 1234
hosting-node ID (DHT): 1450
node-type: *master*

replica-ID	latency (ms)	rank	offset-range
24	50	3	0 - 3
137	10	1	4 - 7
734	30	2	8 - 11

offset-list:

offset: 0
type: int
(other metadata not shown)

value	writer	reader-list
30	(21,4)	
19	(14,4)	(9,2),(11,1)
208	(8,9)	(1,7)

offset: 1
type: float
(other metadata not shown)

value	writer	reader-list
85.2	(11,1)	(9,2),(1,7)
1.5	(14,4)	
0.77	(1,1)	(21,66)

offset: 2
type: char
(other metadata not shown)

value	writer	reader-list
'i'	(21,4)	(11,1)
'c'	(6,30)	
'y'	(8,9)	(14,4)

...

transaction-list:

Client-node: 14
Transaction-ID: 4
Type: read/write
Entries:

d/s ID	offset	access	value
1234	0	write	19
1234	1	write	1.5
1234	2	read	

Client-node: 1
Transaction-ID: 7
Type: read
Entries:

d/s ID	offset	access	value
1234	0	read	
1234	1	read	

Client-node: 8
Transaction-ID: 9
Type: write
Entries:

d/s ID	offset	access	value
1234	0	write	208
1234	2	write	'y'

Client-node: 11
Transaction-ID: 1
Type: read/write
Entries:

d/s ID	offset	access	value
1234	0	read	
1234	1	write	85.2
1234	2	read	

...

Figure 19. Structure of a TORO data-set and associated transactions.

Figure 19 illustrates the structure of a data-set of data-objects, along with a number of associated transactions. Given the hardware-based history of transactional memory [39], the data-objects are also referred to interchangeably as *offsets*. The node-type field indicates whether or not the Transactional Storage module containing the data-set is the master-copy or a replica. As this example illustrates a master-copy, a list of replica-nodes is shown, including the various necessary attributes required for each replica. The attributes are:

- *Replica-ID:* This is the DHT node-key of the replica-node, obtained via the replica-discovery/assignment mechanism outlined in Section 4.2.2.

- *Latency:* This is the measured round-trip latency between the master-node and the replica. The information is updated with each replica heart-beat.
- *Rank:* The replica-rank indicates the priority that each replica follows in the event that a replica detects the loss of a master-node. This information is replicated to all replicas and is updated with the addition/removal of replica nodes.
- *Offset-range:* This is the range of data-objects (or *offsets*) for which the referenced replica is responsible. For example, replica node 24 stores copies of offsets 0-3.

As illustrated, the data-set is primarily referenced by the *data-set ID*, which is the DHT-assigned key. Using this key, any node in the TORO network can locate the entire data-set via a DHT-query. The *hosting-node ID* is the node-key of the hosting node, and the *node-type* indicates whether or not this data-set is the master-copy or a partial replica. As this example shows a data-set master-copy, the assigned replica nodes for this set are also shown, along with a list of the offsets which each replica is hosting. In the interest of space, only the first three of ten offsets are shown in Figure 19 (specifically offsets 0-2).

In Figure 19, the assigned-offsets are non-overlapping, although the master-node can overlap the assigned offsets to increase redundancy, but at the potential cost of increased replication data being sent between nodes. For example, the master-node could assign offsets 0-8 to replica 24 while replica 137 hosts offsets 5-11. Thus offsets 5-8 would be hosted by both replicas, thereby increasing redundancy in the event that both the master and a replica fail simultaneously. Alternately, the master-node may direct all 4 replica-nodes to host the entire offset-range, maximizing the redundancy at the cost of replication overhead. Therefore, TORO provides a form of tunable redundancy and query response, whereby a master-node can chose the level of replication desired, to balance redundancy against replication traffic overhead.

Each of the data-objects within the set is referenced by their offset, analogous to indexing into an array. The objects may represent varying data-types, such as integers, characters, or float values (for clarity, the other meta-data used by the underlying DHT and replication mechanisms are not shown). This provides the flexibility for querying clients to reference multiple objects of varying data-types within a single transaction, unlike traditional transactions which are limited to a single typed data-object. For example, a parallelized computational fluid dynamics simulation may require a computational node to query the values of multiple points within a simulation data-set, within a single time-step of the simulation. The computational node can reference the required objects in the data-set via a single transaction, by simply listing each requested offset along with type of data-access required (read or write). The version history of each data-object is also shown, as originally described; each offset has been updated three times in Figure 19. Each entry in the update-list of a data-object refers to a specific update of the data-object, including the new value, writing transaction and a list of readers which are currently accessing that particular version of the data-object. The format of each entry is described in the following paragraph.

Three write/read-write transactions associated with the data-set are also shown in Figure 19. Each transaction can be located via the tuple *(Client-node, Transaction-ID)*; in the figure, transactions *(14, 4)*, *(8, 9)* and *(11, 1)* are shown, along with the associated object-accesses. For example, transaction *(14, 4)* indicates that it originated at node with DHT node-key 14, with transaction number 4. This tuple-format is required, as there is no central authority assigning global transaction numbers in TORO. Instead, the combination of client-node and transaction-ID allows the contention manager to uniquely address any transaction within TORO. In practice, each participating node in TORO has an incrementing local counter with which to create transaction IDs. Transaction *(14, 4)* is indicated to be of the read/write type, which is shown in its access-list: it is attempting to write into offsets 0 and 1, while reading from offset 2. The *value* field shows the new value which is to be written into the data-object. The updates are also reflected in the offset-list of the data-set, illustrated as the second update at offsets 0 and 1 and the third reader-transaction entry at offset 2. In this example, the transaction would read value *'y'* from the data-object, indicating that transaction *(14, 4)* requested read-access to offset 2 after it was updated by transaction *(8, 9)* with this value.

4.3 Data management and Transaction Concurrency Control

4.3.1 Data-set Creation and Deletion

Data management in TORO is performed on-demand, with the assumption that the internal layout of each pre-defined DHT-located data-set is shared by the transactions accessing it. For example, if defined data-set *D* consists of a 1000-size array, where the first 500 data-objects are integers and the second half consists of double-precision numbers, each transaction which will access *D* is implicitly aware of this layout. We believe this assumption to be reasonable, as we envision TORO being used as a writeable data-space for multiple co-operating processes to share computational data. For example, a large-scale parallel chip simulation package would consist of a pre-defined set of values representing a microprocessor topology, while the transactions accessing that topology would already be intimately aware of the structure of that topology, in order for them to conduct the simulation. However, in the event that a process needs to access data stored in TORO without any prior knowledge of its structure, the system could be readily augmented with a structure-query command which simply returns a metadata message describing the internal structure of the data-set.

Although the structure of a data-set is pre-defined in advance, the data-objects within that data-set are created on-demand, in order to save storage space. To that end, the offset-list is implemented as a dynamically-sized vector list which is resized as required. In a further effort to save space, only the offsets which are accessed by transactions are created within the offset-list. For example, if offset 1 in Figure 19 was never accessed by any transactions, then the offset-listing would be shown as {*0, 2,…*}. Removal of data-sets is also handled on-demand via the DHT-key deletion commands specific to the underlying DHT; in our implementation, data-set removal would be facilitated by the delete-data-key function in Chord [89]. However, the Transactional Storage module on a TORO master-node performs a safety check before deleting a data-set, via the following procedure:

- As a sanity-check, only a master-node is authorized to delete a data-set, which the node will verify.
- Once the node verifies that it is indeed the master, it broadcasts a query to all the replicas in this group with an *intent-to-delete* message.
- Each replica that receives this message will check its own copy of the data-set and verify that there are no transactions which are accessing any of the data-offsets. If there are no such transactions, then the replica-node sets an internal *reject-transactions* flag indicating that it will accept no further transactions for the

referenced data-set and replies to the master-node with an affirmative *delete-OK* message. Otherwise, it returns a *cannot-delete* message to the master-node.

- Once the master-node has received affirmative messages from each replica, it performs the same transaction-check on its own data-set. If there are no active transactions operating on the master-copy of the data-set, the master-node deletes the data-set and signals its replicas to do the same. Otherwise, it sends a *delete-cancel* message to the replicas.
- If a replica receives a *delete-cancel* message from the master-node, it unsets the flag which prohibits acceptance of new transactions. Note that a timeout period is appended to a replica's *reject-transactions* flag, which ensures that in the event of the loss of the delete-cancel message from the master-node, the data-set will resume normal operation. This two-step procedure defers on the side of safety, ensuring that a data-set is not mistakenly deleted from a replica due to network interruption. In this case, it is preferable for a replica to defer to saving the data instead of deleting it, as we consider this an important component of fault-tolerant data storage.

Once a data-set has been defined, the client managing the data-set can be afforded a great deal of control over the replication characteristics of the data-set. In addition to the regular add/delete functions inherited from the underlying DHT, TORO is also planned to include commands which direct the master-node to adjust the size of a replica-group and to specify the replication level of each data-object in the data-set. This will allow a master-node to specify whether or not it requires full replication of all offsets to all replicas, partial replication or no replication at all. For example, a client which manages a particular data-set may choose to explicitly disallow replication of a small portion of the data-set for security reasons, or a number of the offsets within a data-set refer to data-items which cannot be replicated at all, such as direct references to external hardware attached to one of the computational nodes. However, some of these planned functions may come at a cost, as a replica-node may be forced to temporarily disallow transactional access to its data-set while it reconfigures itself. While the current implementation of TORO does not include these on-the-fly modification functions, a client can still control the data-set replication characteristics during the data-set definition phase. Nevertheless, further enhancements can provide this capability to the implemented system in the near future. Thus, the flexibility of the replication system within TORO allows clients to essentially "dial in" the level of replication desired for any individual data-object within the data-set, in order to balance reliability against network overhead or security/hardware constraints.

4.3.2 Concurrency Control

TORO's transactional control operates in a manner very similar to the JVSTM versioning scheme developed by Cachapo et al. [13] [27] and the *Distributed Multiversioning* (DMV) mechanism of Massiniev et al. [53]. Any transaction which contains write-operations is processed only at the master-node in the replica group, in order to enforce write-order serialization. Thus, if a write or read/write transaction is created on a replica-node, the transaction is immediately forwarded to its master-node via the DHT for processing. Additionally, as TORO can utilize partial replication to save on replication and/or data-storage overhead, the Transactional Storage module must also check to see if it hosts the offsets requested by the transaction. For example, if a replica-node for a 0-20 size data-set has been assigned offsets 0-10 by the master-node and it receives a read-only transaction which references offsets 0-15, then it will be forced to forward the transaction to the master-node, as it does not host all of the data-objects which the transaction is requesting. The behavior and conflict management of the transactions is described as follows:

- If a *read-only* transaction arrives at a replica node and if the replica can service the transaction, it locates the relevant offsets and appends the (client-node/transaction-ID) tuple to the reader-list of the newest offset version. Once it has completed appending the reader-entries, the replica-node sends a notification to the master-node to inform it of the read. The notification message includes the list offsets being accessed by the transaction, along with the observed versions of each data-object which the transaction is reading. The read-transaction will continue to access this entry for the duration of its execution, independent of any subsequent updates which are appended to the update-list of the offset. Therefore the read-transaction is essentially executing in isolation, as none of the referenced data-values for that version changes during the transaction's life, thereby preserving consistency.
- Upon arrival of a *write-only* transaction at the master-node, TORO examines the list of offsets which the transaction is attempting to update. If the offsets do not exist, they are created on the fly, but the updates specified within the write-transaction are not applied to any of the offsets. When all missing offsets have been created, the master-node then broadcasts a list of offset updates to the relevant nodes within its replica-group. Each replica which receives the update-notification initially responds to the master-node with an affirmative synchronization message. It then appends the write-update to the relevant offset's version list, allowing existing read-transactions to continue executing without being affected by the new data. Once the master-node

receives confirmation of the accepted updates from each replica, it commits the write-transaction by appending to each relevant offset a new entry to the *dataValues* list of the offset, populating it with the necessary value/writer entries.

- If the master-node receives a *read-write* transaction, it locates the relevant offsets and appends the relevant node/transaction tuples to the reader-list of the most up-to-date offset version, in the same manner as the replica. It must then wait until the transaction is ready to commit its write-updates, then it must perform a transactional validation, whereby each offset in the read-list of the transaction is checked to determine that the transaction's node/transaction tuple is still appended to the reader-list of the *latest* version of the offset. If any node/transaction tuples are no longer appended to the reader-list of the latest offset version, this indicates that another writing transaction has updated the offset with a new version after the read-write transaction began. As continuing with this condition would violate mutual consistency, the read-write transaction is aborted and restarted. Otherwise, the read-write transaction is still valid and is thus immediately committed, using the write-commit mechanism as described above.

Figure 20 illustrates the transaction-processing logic used by TORO. In the interest of clarity, the write-transaction response logic between the master-node and replicas is not shown, although the expected responses are included as part of the transactional path logic. The centralized-write mechanism provides the scalability of allowing any replica to service read-only transactions with little delay, while retaining a relatively simple method of enforcing write serialization. TORO's replica optimization and ranked failure-recovery mechanism thus mitigates the performance penalty and risk associated with centralized transactional control.

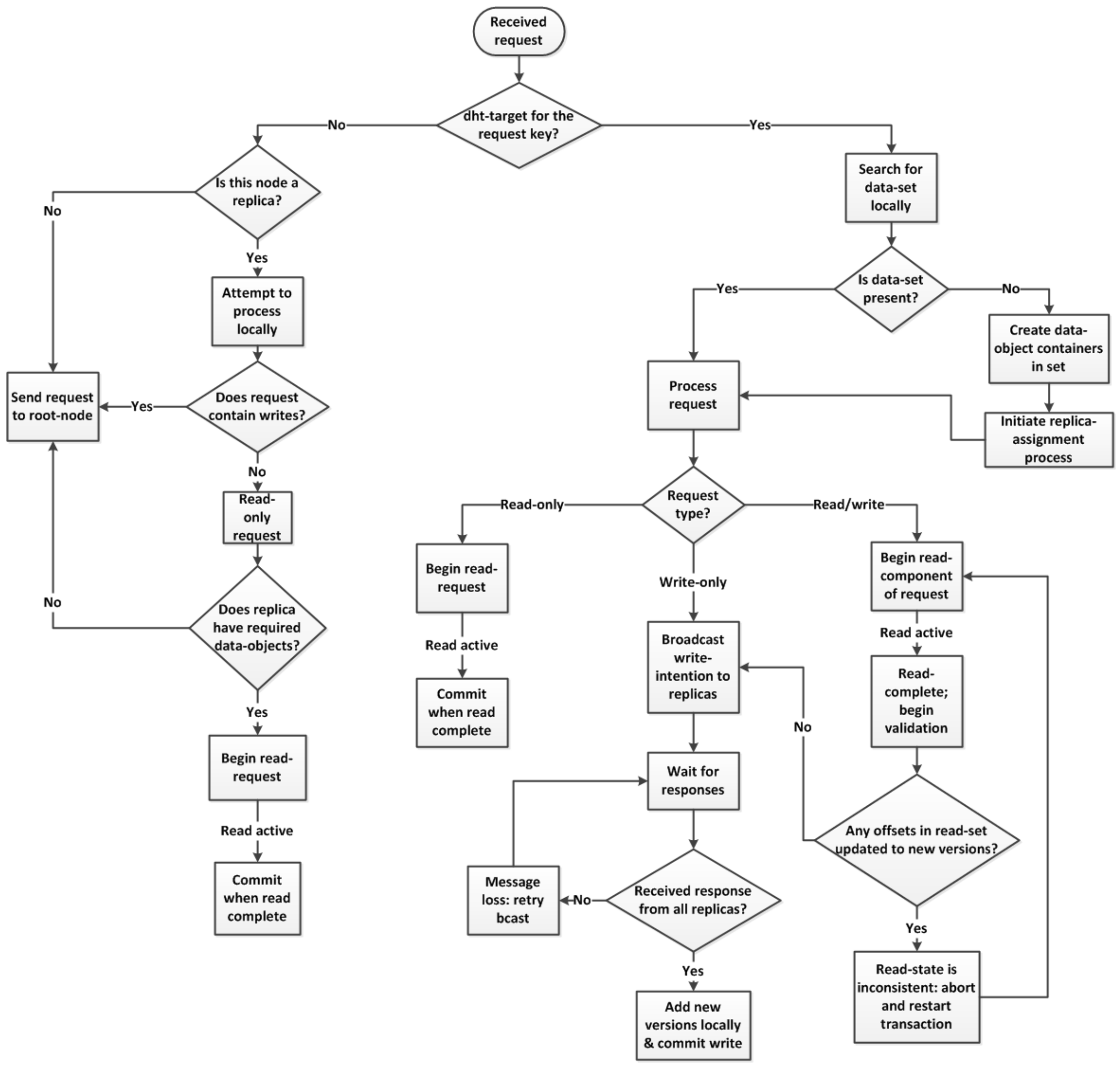

Received request
dht-target for the request key?
No
Yes
Is this node a replica?
No
Yes
Attempt to process locally
Send request to root-node
Yes
Does request contain writes?
No
Read-only request
No
Does replica have required data-objects?
Yes
Begin read-request
Read active
Commit when read complete
Search for data-set locally
Is data-set present?
Yes
No
Create data-object containers in set
Initiate replica-assignment process
Process request
Request type?
Read-only
Read/write
Write-only
Begin read-request
Read active
Commit when read complete
Broadcast write-intention to replicas
Wait for responses
Received response from all replicas?
No
Message loss: retry bcast
Yes
Add new versions locally & commit write
Begin read-component of request
Read active
Read-complete; begin validation
Any offsets in read-set updated to new versions?
No
Yes
Read-state is inconsistent: abort and restart transaction

Figure 20. TORO transaction request flow.

4.4 Evaluation Environment

As TORO was designed as a distributed transactional workspace, our primary interest lies in the following metrics:

- Transaction execution time
- Transaction abort rate

- Traffic overhead

These metrics are the same factors shared by most existing research evaluations of distributed and replicated transactional architectures. Given the primary contribution aims of TORO, we tested the concurrency components by comparing them to traditional locking scenarios, in both replicated and non-replicated environments. The locking architecture implemented for comparison closely resembles both the DHT-based Etna architecture [61] and the Chubby locking service developed by Google [11]. Both of these systems depend on a centralized manager to control serialization and to enforce read and write access to shared objects via this central manager. This gives us several configurations to test, namely:

- Non-replicated data with traditional locking for concurrency
- Replicated data with traditional locking for concurrency
- Non-replicated data with lock-free transactional access
- Replicated data with lock-free transactional access (the TORO configuration)

Each of the listed configurations was subjected to multiple tests, varying both the topology size and the types of transactional inputs. The different types of transactional inputs are discussed below.

4.4.1 Simulator Configuration

Like PVC-DS, TORO has been developed inside an event-driven simulator, utilizing similar network conditions to PVC-DS. The simulated network was generated using Inet-3.0 by Winick [93], processed through a ModelNet-supplied script [94] to produce the initial network topology. To provide realistic and variable network latencies, measurements were taken of real-time latency tests between randomly-chosen network hosts and used to provide realistic variances in inter-node latency within the simulator. Given the aforementioned purpose of TORO, the transactional inputs for initial testing were synthetically generated, as so to produce worst-case contention scenarios. These generated inputs thus provided us with a method of verifying the correctness of the concurrency control mechanisms, as well as giving us initial performance results which assisted in tuning the various parameters (e.g. replication timers) of the replication components. At the same time, the comparison architectures were tested with comparable inputs and parameter settings, in order to provide an unbiased comparison between them and TORO. To provide parity between the comparison systems and TORO, the replication group size was set to 5 replicas with full replication, to match the replication factor of Chubby [11].

Once it was determined that both TORO and the comparison systems were functioning correctly and producing consistent results, the simulations were provided with inputs taken from real-world benchmarks. The *Princeton Application Repository for Shared-Memory Computers* (PARSEC) benchmarking tool [9] was chosen as an input source, as it provides a wide range of test scenarios, which were adapted to our simulations in a relatively straightforward manner. The PARSEC benchmarks also cover a wide range of data-set sizes and include workloads with both low and high read-to-write ratios. The input sets which we have chosen from the PARSEC suite are the *BlackScholes* financial analysis benchmark which is dominated by read-only transactions, and the *Canneal* engineering benchmark which contains large numbers of writes to shared data-spaces. Both benchmarks have an emphasis in data-parallelism, making them ideally suited for testing the distributed architectures of both TORO and the comparison architectures. It can be reasonably assumed that the majority of large-scale jobs submitted to a transactional architecture such as TORO would fall under this classification of highly-parallelized applications. However, in practice it was discovered that neither the BlackScholes nor Canneal benchmarks actually generated much contention in the evaluations: the Canneal benchmark only generated sporadic contention due to the sparse nature of its write-commands within data-sets, while BlackScholes contained no overlapping data-write commands at all.

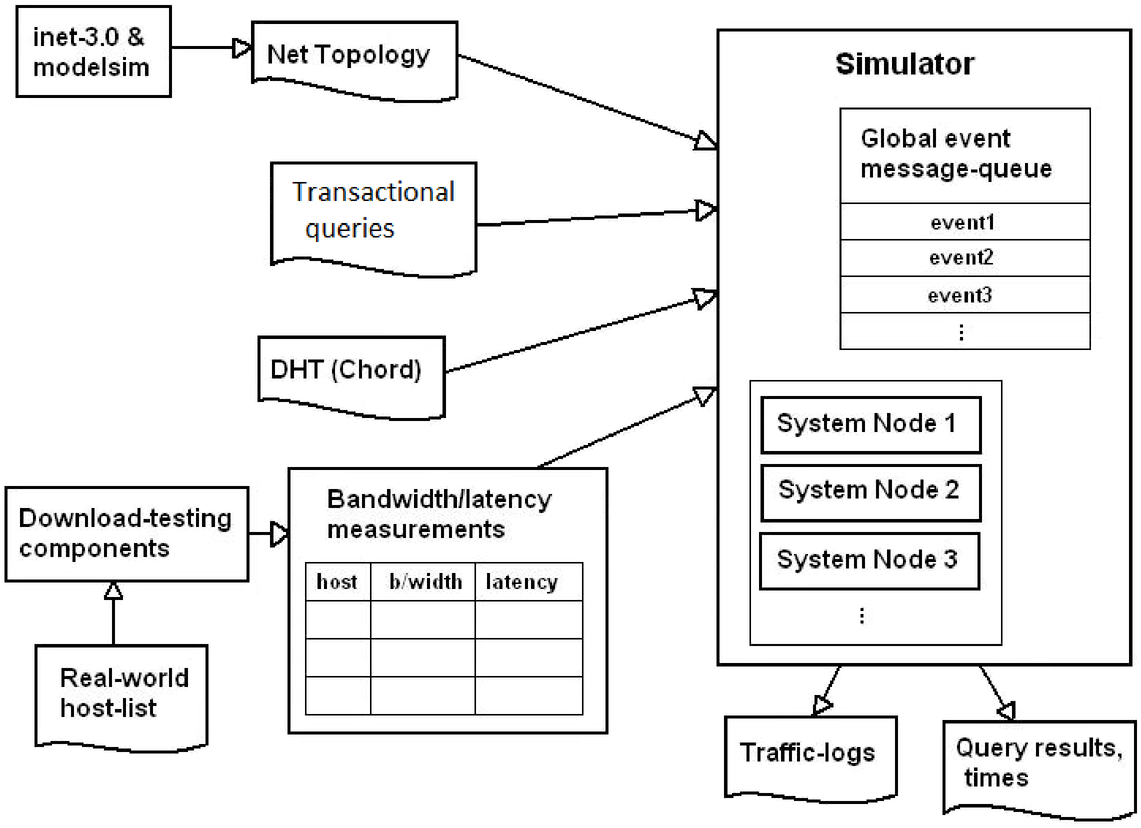


Figure 21. TORO instrumental evaluation.

Therefore, we generated a special high-contention data-benchmark to test both TORO and its comparison locking configuration. This benchmark contains a large number of overlapping data-reads and data-writes from differing clients to a relatively small number of data-sets which increases the probability of aborts occurring within the evaluation. Thus, TORO and the comparative systems were subjected to scenarios which range from conditions of little/no contention, to input sets which may trigger high contention and/or abort rates during operation.

A diagram of the evaluation setup is shown in Figure 21. As illustrated, a very large number of the testing components from the PVS-DS simulator have been re-purposed for use in testing TORO. As the original network topology was designed to be both very flexible and representative of real-world network conditions, we have elected to re-use it completely intact. The event-driven nature of the simulator allows for out-of-order message arrivals, thus creating asynchronous network conditions within which to test TORO. While we are not using the same multi-layered architecture as PVC-DS, TORO was designed to take advantage of a fully decentralized communications overlay. Thus adding a lower layer of non-DHT nodes would provide little benefit to the system, or would give TORO an unfair advantage over comparison systems. However, it would be a relatively straightforward task to add this functionality into the existing TORO architecture in the future.

Like PVC-DS, TORO and its comparison architectures were tested with varying network sizes, ranging from 100 nodes to 1000 nodes, in increments of 100 nodes. As the query-inputs were derived from real-world benchmark calculations, the inputs were processed by each configuration in sequential order with no delay between transactions. This corresponds to the real-world equivalent programs accessing the data items needed for execution in serial order. As both BlackScholes and Canneal were designed to operate in a multi-threaded capacity, each execution thread within the inputs was assigned to a different node, corresponding to the parallel execution of a program on a shared data-set by multiple nodes within a cluster. Scenarios like this are how we envision virtual clusters, similar to those deployed by PVC-DS, would be utilized. Once deployed by PVC-DS, each cluster node would begin execution of its sub-task on a large shared data set, while relying on TORO to maintain the consistency of shared data objects used by the executing worker-processes on each virtual cluster node. Like PVC-DS, the size of the system load was varied to match the system size.

In the case of BlackScholes and Canneal, the total number of transactions input into the evaluation was set as a fraction of a single static input set. For example, as the BlackScholes benchmark corresponded to 12800 total data-access requests, the 100-node evaluation was made to execute the first 1280 transactions, the 200-node evaluation the first 2560 transactions, and so forth, with the 1000-node evaluation receiving the full BlackScholes transaction-input set. This ensured that every evaluation was directly comparable to each other, as each successively larger input-load included the same transactions as its smaller preceding evaluation load.

4.5 Results

The execution times for the BlackScholes benchmark are shown in Figure 22 and Figure 23. The non-replicated configuration in Figure 22 exhibits a modest decrease in execution time of 5.5% over the traditional locking implementation. However, the benefits of transactional memory become clear when combined with replication, as illustrated in Figure 23, where TORO exhibits a decrease of approximately 83% over the locking configuration. This is due to the BlackScholes benchmark being comprised of *Read-Only* (RO) and *Write-Only* (WO) transactions, where the transactional components of TORO allow any RO or WO transaction to be serialized and committed immediately.

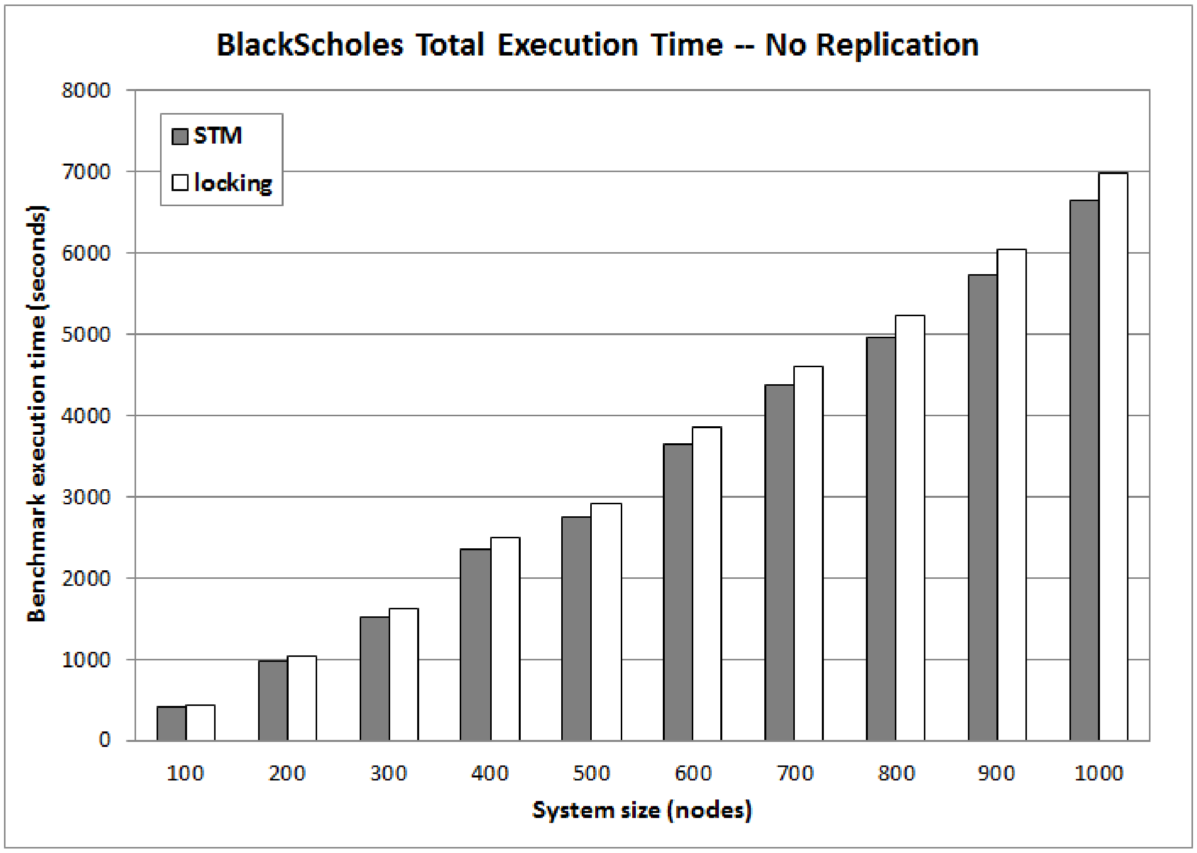


Figure 22. Execution times for non-replicated BlackScholes benchmark.

In comparison, the locking implementation requires that any shared object being opened in RO-mode must acquire at least a non-exclusive lock by the reading process (denying any other process from writing to the object), while WO access demands an exclusive access (denying other processes access to the shared object entirely). Transactional access removes both of these limitations, thus allowing processes to either read from or write to shared objects without waiting for other processes to release the shared object.

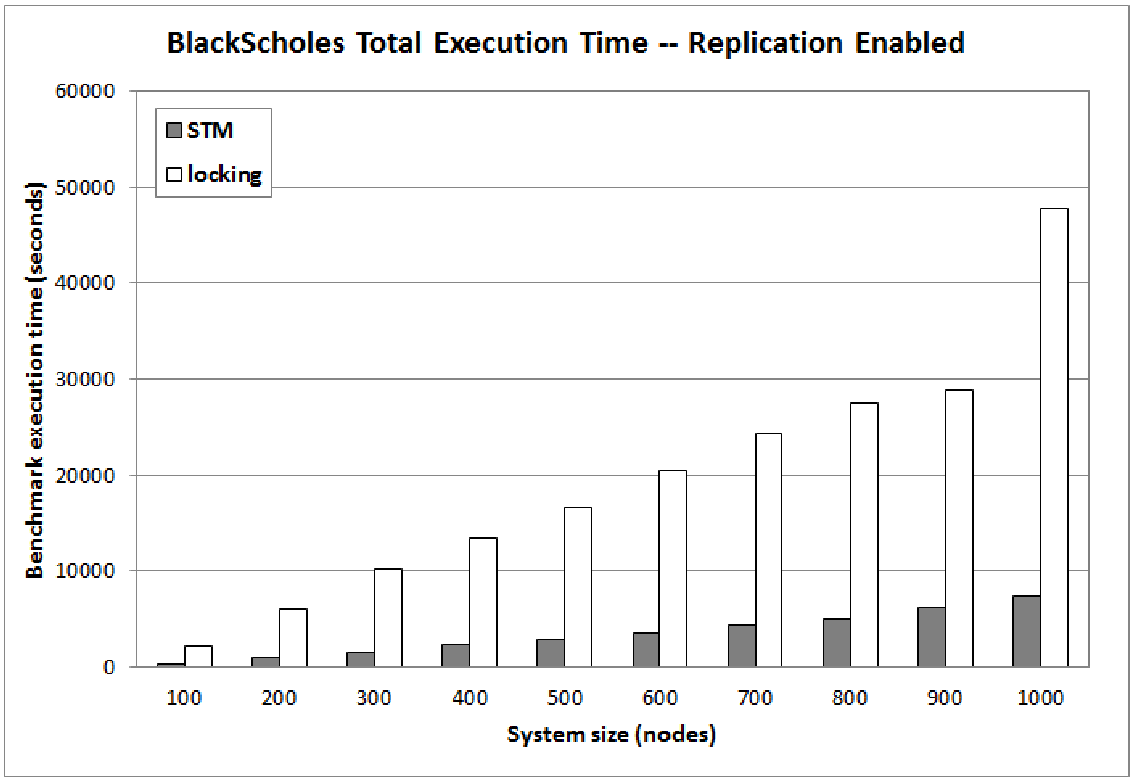


Figure 23. Execution times for BlackScholes benchmark with replication enabled.

While both the non-replicated (Figure 22) and replicated (Figure 23) configurations benefit from transactional data access, the replicated data configuration allows multiple copies of the object to be made available for read-access. Write access to the replicated objects requires more time, as the writes in TORO are serialized through the master-node. However, as the majority of the transactions in the BlackScholes benchmark are RO-access requests, the replica-nodes can intercept and service these read-requests from remote clients without having to wait for synchronization from the master-node. In contrast, as all read-access requests must be serialized through the master-node in the locking configuration, the master-node becomes a bottleneck for the read-requests, thereby inducing performance loss.

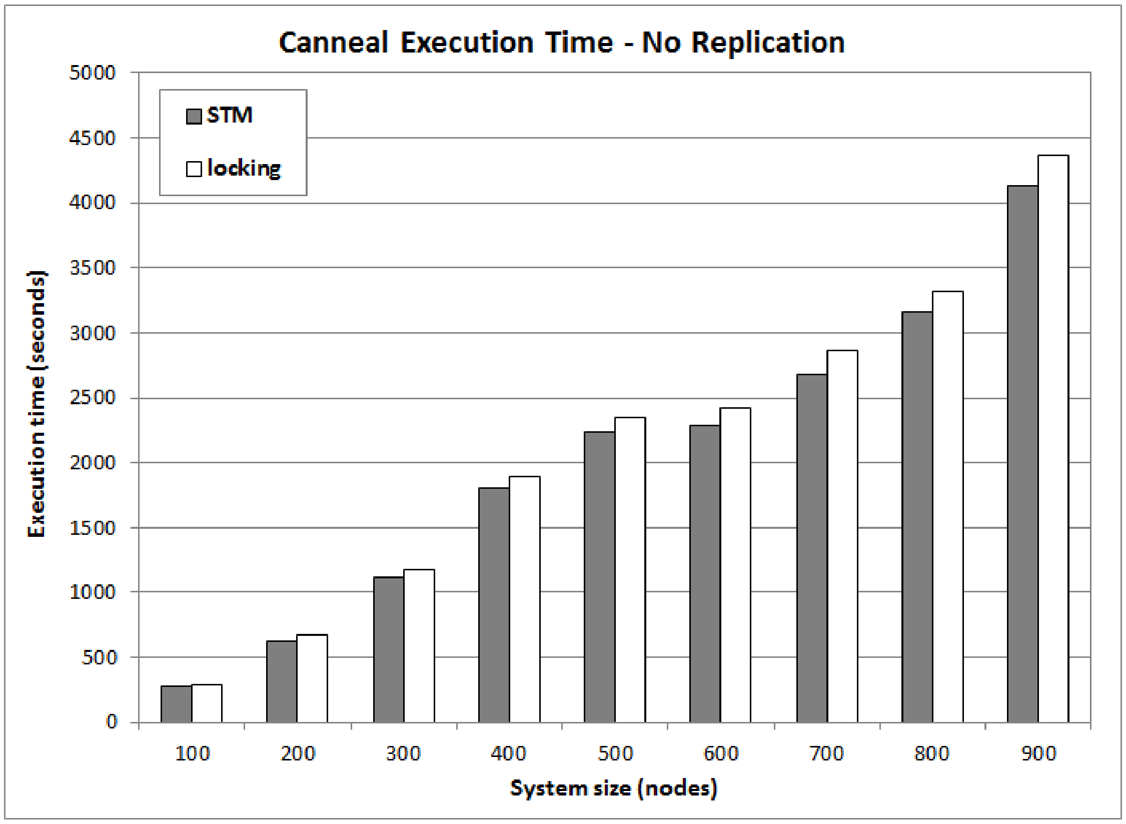


Figure 24. Execution times for non-replicated Canneal benchmark.

Like BlackScholes, the Canneal benchmark also exhibits decreases in execution time under TORO. Figure 24 shows the execution of Canneal under non-replicated configurations. In this case, the performance gain is approximately the same as that shown by the BlackScholes benchmark, with an approximate decrease in execution time of 5.7% compared to the locking configuration. Similarly, with replication enabled, the execution time decreases by 22.5%. The performance gain provided by TORO, while significant, is somewhat mitigated under Canneal by the large number of write-access requests to shared objects within the benchmark. This is due to each write-access having to be serialized through the master-node to maintain consistency. Nevertheless, write-access requests are a relative minority in Canneal, allowing TORO to provide performance gains across the majority of the benchmark.

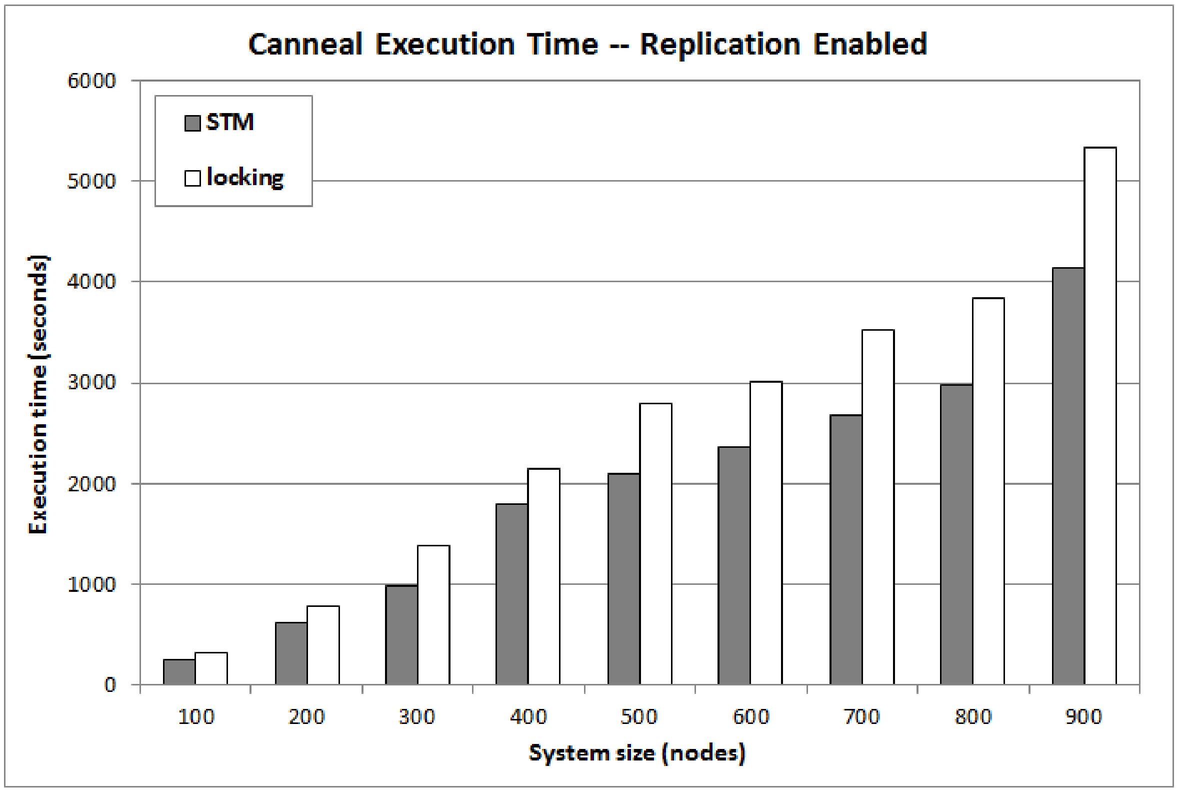


Figure 25. Execution times for Canneal benchmark with replication enabled.

In addition, WO transactions also benefit from instant serialization, allowing any such transactions in Canneal to enjoy this performance gain, with the only loss in performance due to the replica-synchronization requirement imposed by TORO for write-access requests. However, as the replica-nodes in TORO are chosen with network link-delay in mind, the loss of performance due to inter-node communications latency is minimized, thus allowing enhanced synchronization performance between the master-node and replicas.

The high-contention scenario was the final evaluation run against the tested architectures. Although this evaluation contains a much higher incidence of contending read/write transactions than would normally be encountered by most parallel applications, it serves to demonstrate how well TORO can deal with these worst-case scenarios, compared to the traditional locking architecture. Figure 26 and Figure 27 illustrate the performance of non-replicated architectures with and without transactional support respectively.

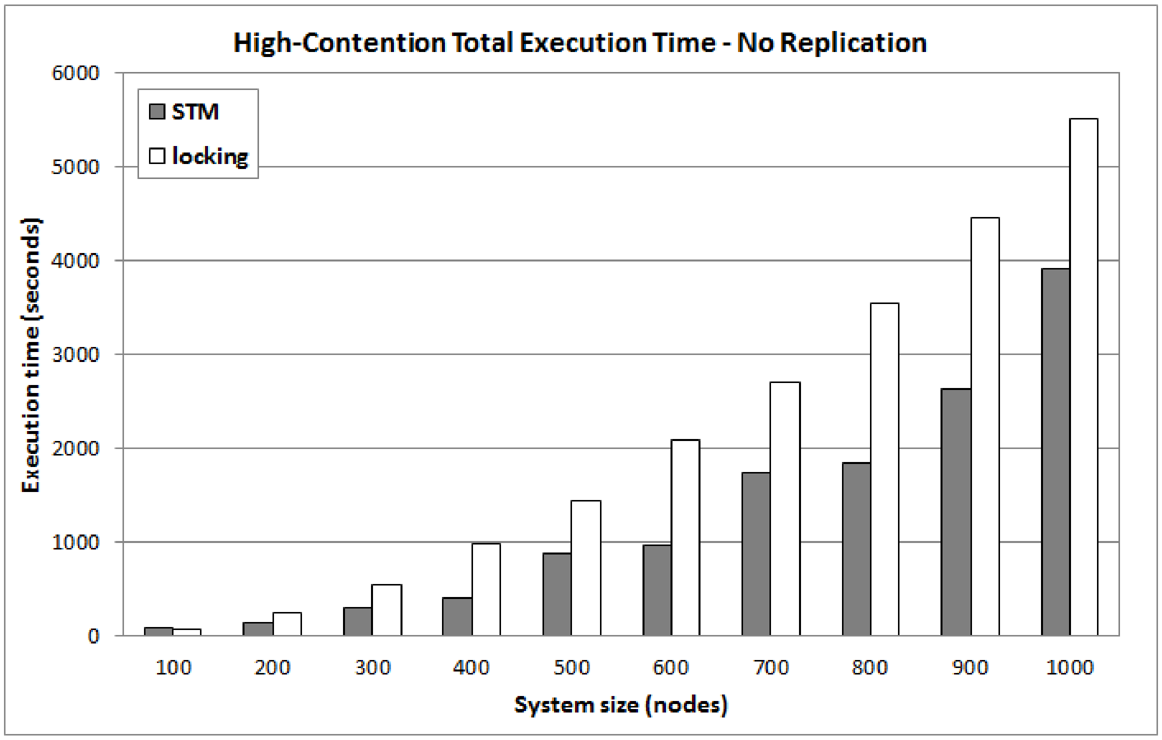


Figure 26. Execution Times for high-contention evaluation without replication.

The transactional components enable about 37% better execution times than the traditional locking scenario, which suffers from the high wait-times associated with such severe contention scenarios. While the contention instances did result in some transactions being forced to abort, the number was relatively low compared to the over transaction count; in the worst-case scenario, approximately 50 transactions per evaluation were forced to restart, with none being permanently aborted. As a result, the overall time for TORO to complete the evaluation was only moderately affected, compared to the locking scenario.

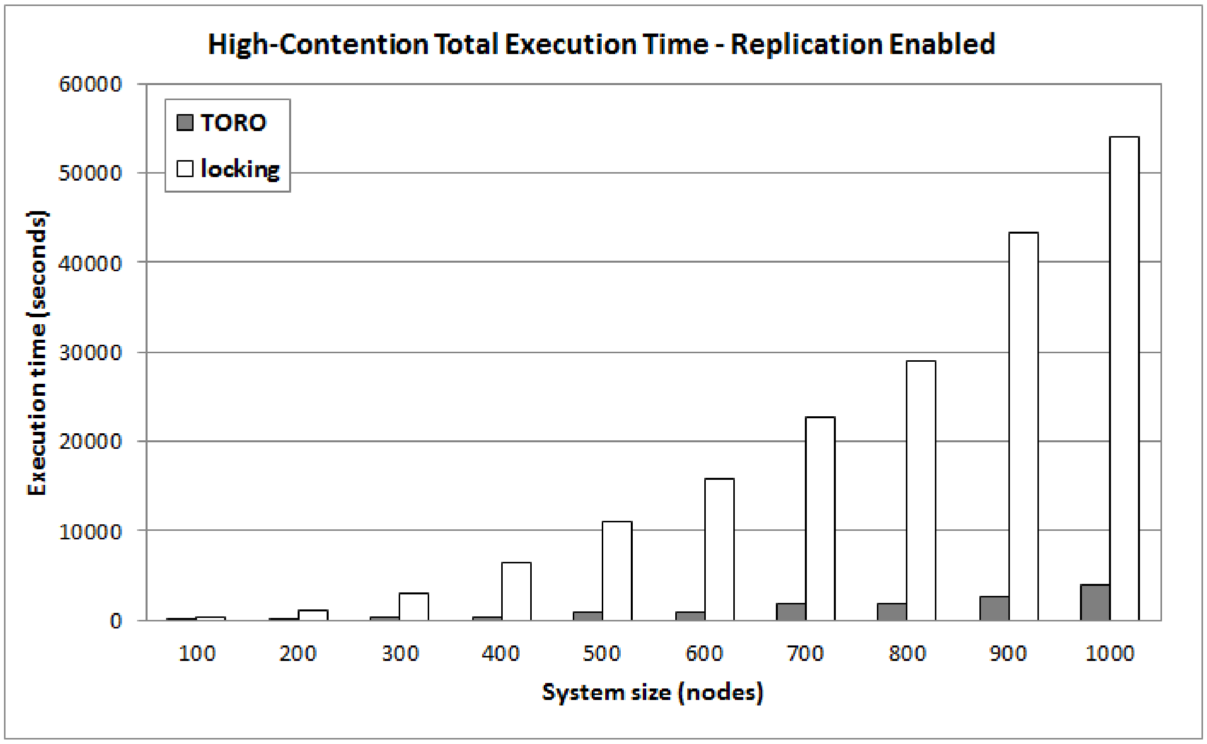


Figure 27. Execution times for high-contention scenario with replication enabled.

This difference in performance is even more marked when replication is introduced, as shown in Figure 27. In a replicated scenario, TORO is much better able to take advantage of the transactional components to provide performance gains to the non-contending components within the evaluation. Like the non-replicated configuration, contention can only occur during the serial validation and execution of *Read-Write* (RW) transactions at the master-node. As a result, the average number of transaction-aborts within TORO is approximately the same as for the non-replicated architecture. However, the instant-serialization of both read-only and write-only transactions allows even heavily contending transactions to execute without waiting for mutual exclusion, unlike the locking scenario. As a result, the execution of the high-contention scenario is almost 90% faster in TORO than with locking.

The effects of larger or smaller sized replica groups were also investigated. Although the evaluations showed little difference in benchmark execution times for TORO, largely due to the additional time required for write-synchronization being offset by the decreased time required for a portion of read-only requests, they are intercepted and answered by replica-nodes in lieu of the master node. In practice, increasing replica nodes may have a beneficial effect when executing RO-dominant jobs, albeit at the cost of additional replica-group initiation time and increased traffic overhead required to maintain the soft-state status of each group replica.

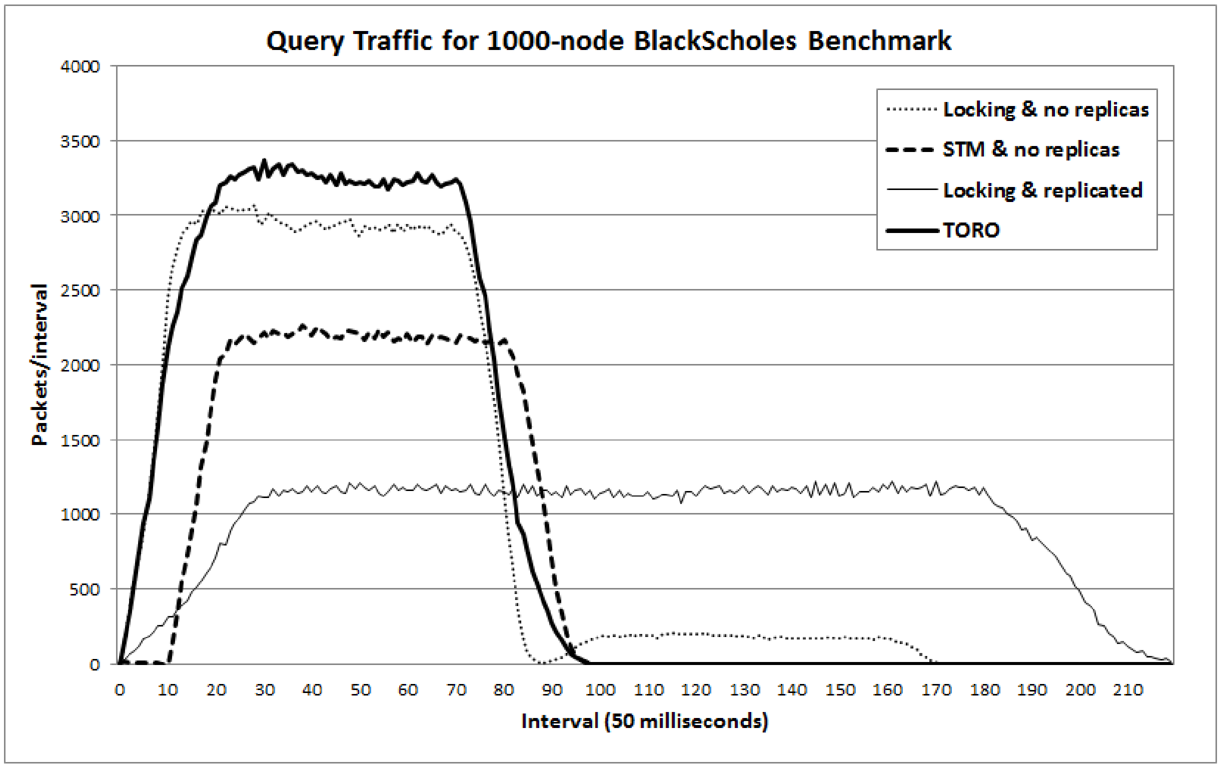


Figure 28. Query traffic generated by the BlackScholes benchmark over 1000 nodes.

Finally we examine the traffic generated by TORO and the comparison architectures for both the BlackScholes and Canneal benchmark evaluations. Figure 28 shows the query traffic generated by each of the architectures as they evaluated the BlackScholes benchmark over 1000 nodes. As the query traffic comprised the vast majority of the overhead generated by the tested systems, it was omitted from the graph. It should be noted that the synchronization communication between the master-node and its replicas was considered as part of the query traffic and is thus included in the overall traffic calculations. The other traffic which we have chosen to omit is related to initial replication-group construction and associated keep-alive messages, which is very infrequent and is not considered significant in this evaluation.

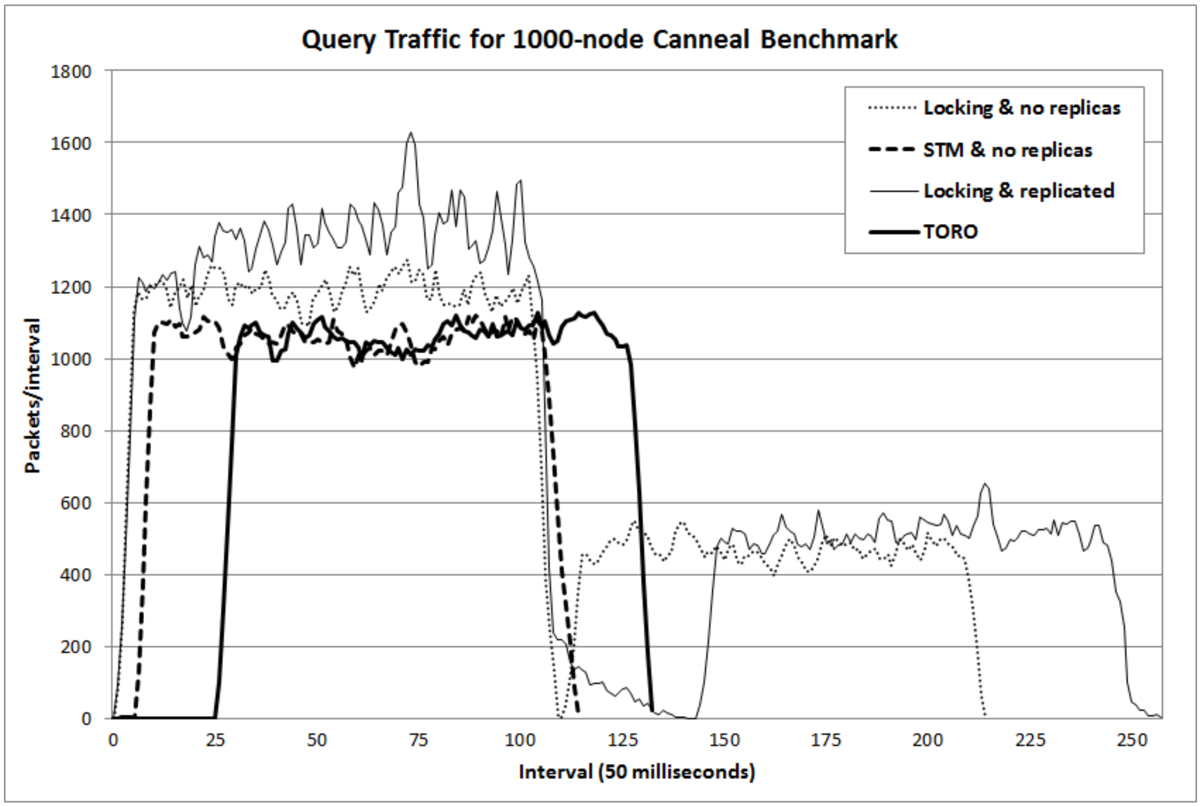


Figure 29. Query traffic generated by the Canneal benchmark over 1000 nodes.

As Figure 28 shows, TORO generates a moderate amount of additional traffic compared to the alternate systems, but the reward associated with this additional overhead is worth the additional performance in a replicated scenario. Even under the relatively contention-free conditions of the BlackScholes benchmark, the locking architecture generates its traffic over a much longer period of time, indicating much lengthier transaction durations. Traffic-wise, TORO is only slightly more expensive than the traditional centralized locking architecture.

In comparison, under the Canneal evaluation (Figure 29), the TORO traffic overhead is lower than the traditional locking architecture in both replicated and non-replicated configurations. Traffic-wise, TORO produces about the same volume of traffic as its non-replicated STM-enabled counterpart, with similar execution times. As the graph illustrates, both locking architectures continue to produce traffic well after both the non-replicated Software Transactional Memory (*STM*) and TORO have completed, indicating longer execution times.

4.6 Contributions & Conclusions on TORO

Our research demonstrates a new mechanism for storing and replicating data in a distributed Peer-to-Peer overlay architecture. In contrast to existing replicated storage architectures which rely on existing overlay constructs or configurations, TORO's replica-discovery mechanism enables dynamic data replication with a strong focus on the reduction of access time, thus improving the performance of applications utilizing the replicated data. Our modifications to the transactional algorithm simplify contention management, yet retain performance gains matched to the increase in replica-group size, thereby increasing the scalability of the architecture.

Overall, TORO has demonstrated significant improvements over traditional locking architectures in the execution of several well-established benchmarks and test scenarios. While the improvements made in some areas have been moderate, they illustrate that as a replicated storage system, TORO can at least remain on par with traditional storage architectures whilst providing the increased redundancy afforded by replicated data. In conjunction with PVC-DS, a user can now rapidly deploy applications over a decentralized communications overlay and perform computational work on those distributed applications with little to no performance loss, compared to a monolithic computational architecture. Both PVC-DS and TORO, by use of various replication and data-synchronization techniques, combine to form an integrated framework for the discovery, dissemination and manipulation of shared data, even if it is stored on widely-spaced computational resources. Thus, the PVC-DS/TORO pair serves to bring many smaller data storage & computation resources together to create a unified large resource for users who wish to perform either parallel or serial computation on large shared data items. This combination of mechanism avoids the additional costs, potential security implications or site disaster risks associated with consolidated data-center solutions.

5. Conclusions and Recommendations for Further Work

5.1 Conclusions

The PVC-DS and TORO systems developed in this project demonstrated two major components which together, formed (i) an integrated system for the rapid deployment of virtual machines (VMs) and applications and (ii) a fault-tolerant distributed storage architecture which enables high-performance concurrent access to large data sets. Built upon Peer-to-Peer overlay architectures, each component incorporated new features and modifications of existing mechanisms to provide such capabilities.

PVC-DS featured a ChunkCast-based multi-layered architecture which, in concert with a new two-stage query system, allows the publishing of dynamic VM-image and package data, coupled with the advanced range-query search of this published metadata. Modifications to the existing publishing mechanisms within ChunkCast enable contributing nodes within PVC-DS to dynamically change the content of their published information while allowing these changes to propagate throughout the system.

Our evaluations of PVC-DS have demonstrated an 85% average reduction in single-VM deployment times compared to a single-node repository configuration and an 8% improvement over a grouped multiple-node repository configuration. PVC-DS performed even more favorably when subjected to multiple-VM range queries, demonstrating a 99% and 15% improvement over single-node and group-node repository configurations, respectively. In all cases, PVC-DS exhibited only minor increases in query overhead traffic and reduced the publish-traffic by about 17% compared to the grouped-node repository configuration.

TORO uses the routing features of the Chord distributed hash table (DHT) architecture to build data-set based replica groups of nodes. Unlike existing DHT-based replication approaches, TORO replica groups accommodate network latency during both construction and maintenance. This allows replication groups to be dynamically modified to improve reliability and/or performance, as required by the end-user. Our modified form of transactional data access provides rapid access to shared data items within the replica group, while the read-write policy used by TORO simplifies the concurrency mechanism and avoids the expensive consensus algorithms of comparative storage architectures.

Evaluations of TORO with various parallel benchmarks compared favorably to traditional non-locking concurrency mechanisms under replicated configurations. Under evaluations with a high percentage of read-requests, TORO demonstrated an average improvement in benchmark execution time of 83% over traditional locking mechanisms. When evaluated using inputs with a moderate number of read-inputs, TORO showed a 22% improvement over locking, while a scenario with an extremely high volume of contending inputs resulted in TORO gaining a 90% improvement in execution time over traditional locking.

Together, PVC-DS and TORO create an architecture which an end-user can utilize to connect multiple small data-centers together to produce a single, unified computational resource. This capability provides a viable alternative to either expensive data-center upgrades or consolidation of data and computational resources into foreign large-scale data-centers.

5.2 Recommendations for Further Work

Since our evaluations of PVC-DS and TORO were conducted instrumented simulation environments, actual implementations of PVC-DS should be created to fully test the integrated components, using actual computational and network conditions. In addition, the use of TORO should be investigated as a mechanism for creating scalable variable-accuracy computation algorithms will be investigated in the future. By inserting selective termination conditions into the transaction abort/restart controls of TORO, end-users will be able to control the precision of specific classes of large-scale and long-term computations, simultaneously controlling execution time and overhead.

References

[1] Allcock, Bill, et al. “Secure, Efficient Data Transport and Replica Management for High-performance Data-intensive Computing.” 18th IEEE Symposium on Mass Storage Systems and Technologies, 2001. 13-13.

[2] Amazon, AWS. “Amazon Web Services.” Available in: http://aws.amazon.com/es/ec2/ (November 2012) (2010).

[3] Armbrust, Michael, et al. “A View of Cloud Computing.” Communications of the ACM, 53.4 (2010): 50-58.

[4] Barham, Paul, et al. “Xen and the Art of Virtualization.” ACM SIGOPS Operating Systems Review, 37.5 (2003): 164-177.

[5] Bartlang, U. and J.P. Muller. “DhtFlex: A Flexible Approach to Enable Efficient Atomic Data Management Tailored for Structured Peer-to-Peer Overlays.” 3rd International Conference on Internet and Web Applications and Services, 2008. 377-384.

[6] Ben, M. Principles of Concurrent and Distributed Programming, Second Edition. Second. Addison-Wesley, 2006.

[7] Bernstein, Philip A. and Nathan Goodman. “Multiversion Concurrency Control - Theory and Algorithms.” ACM Transactions on Database Systems, 8.4 (1983): 465-483.

[8] Bernstein, Philip A., Vassos Hadzilacos and Nathan Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.

[9] Bienia, Christian. “Benchmarking Modern Multiprocessors.” Ph.D. dissertation. 2011.

[10] Brewer, Eric A. “Towards Robust Distributed Systems.” PODC, 2000. 7-7.

[11] Burrows, Mike. “The Chubby Lock Service for Loosely-coupled Distributed Systems.” Proceedings of the 7th Symposium on Operating Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, 2006. 335-350.

[12] Buyya, Rajkumar, Chee Shin Yeo and Srikumar Venugopal. “Market-oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities.” 10th IEEE International Conference on High Performance Computing and Communications, 2008. 5-13.

[13] Cachopo, Joao and Antonio Rito-Silva. “Versioned Boxes as the Basis for Memory Transactions.” Science of Computer Programming, 63.2 (2006): 172-185.

[14] Chase, J. S., Irwin, D. E., Grit, L. E., Moore, J. D., & Sprenkle, S. E. “Dynamic Virtual Clusters in a Grid Site Manager.” Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing, 2003. 90-100.

[15] Chohan, Navraj, et al. “Appscale: Scalable and Open Appengine Application Development and Deployment.” Cloud Computing. Springer, 2010. 57-70.

[16] Chun, Byung-Gon, et al. “ChunkCast: An Anycast Service for Large Content Distribution.” IPTPS, 2006.

[17] Ciurana, Eugene. “Developing with Google App Engine.” (2009).

[18] Clark, Christopher, et al. “Live Migration of Virtual Machines.” Proceedings of the 2nd Symposium on Networked Systems Design & Implementation, 2005. 273-286.

[19] Cooper, Brian F, et al. “PNUTS: Yahoo!'s Hosted Data Serving Platform.” Proceedings of the VLDB Endowment, 1.2 (2008): 1277-1288.

[20] Corbett, J.C. “Evaluating Deadlock Detection Methods for Concurrent Software.” IEEE Transactions on Software Engineering, 22.3 (1996): 161-180.

[21] Couceiro, M., et al. “D2STM: Dependable Distributed Software Transactional Memory.” 15th IEEE Pacific Rim International Symposium on Dependable Computing, 2009. 307-313.

[22] Czajkowski, K., et al. “Grid Information Services for Distributed Resource Sharing.” Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing, 2001. 181-194.

[23] Dabek, Frank, et al. “Vivaldi: A Decentralized Network Coordinate System.” ACM SIGCOMM Computer Communication Review, 34.4 (2004): 15-26.

[24] Davidson, Susan B., Hector Garcia-Molina and Dale Skeen. “Consistency in a Partitioned Network: A Survey.” ACM Computer Survey, 17.3 (1985): 341-370.

[25] Defago, Xavier, Andre Schiper and Peter Urban. “Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey.” ACM Computer Survey, 36.4 (2004): 372-421.

[26] Douglis, Fred, et al. “Content-aware Load Balancing for Distributed Backup.” Proceedings of the 25th International Conference on Large Installation System Administration, 2011. 13-13.

[27] Fernandes, Sergio Miguel and Joao Cachopo. “Lock-free and Scalable Multi-version Software Transactional Memory.” Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, 2011. 179-188.

[28] FIPS, PUB. “180-1. Secure Hash Standard.” National Institute of Standards and Technology 17 (1995).

[29] Foster, Ian, Carl Kesselman and Steven Tuecke. “The Anatomy of the Grid: Enabling Scalable Virtual Organizations.” International Journal of High Performance Computing Applications, 15.3 (2001): 200-222.

[30] Foster, Ian, et al. “Cloud Computing and Grid Computing 360-degree Compared.” Grid Computing Environments Workshop, 2008. 1-10.

[31] Foster, Ian, et al. “The Physiology of the Grid.” Grid Computing: Making the Global Infrastructure a Reality, (2003): 217-249.

[32] Foster, Ian, et al. “Virtual Clusters for Grid Communities.” 6th IEEE International Symposium on Cluster Computing and the Grid, 2006. 513-520.

[33] Fraser, Keir and Tim Harris. “Concurrent Programming without Locks.” ACM Transactions on Computer Systems, 25.2 (2007).

[34] Gilbert, Seth and Nancy Lynch. “Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services.” SIGACT News, 33.2 (2002): 51-59.

[35] Gilbert, Seth and Nancy Lynch. “Perspectives on the CAP Theorem.” Computer, 45.2 (2012): 30-36.

[36] Haerder, Theo and Andreas Reuter. “Principles of Transaction-oriented Database Recovery.” ACM Computer Survey, 15.4 (1983): 287-317.

[37] Herlihy, Maurice and J. Eliot B. Moss. “Transactional Memory: Architectural Support for Lock-free Data Structures.” Proceedings of the 20th Annual International Symposium on Computer Architecture, 1993. 289-300.

[38] Herlihy, Maurice. “Wait-free Synchronization.” ACM Transactions on Programming Languages and Systems, 13.1 (1991): 124-149.

[39] Herlihy, Maurice, et al. “Software transactional Memory for Dynamic-sized Data Structures.” Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing, 2003. 92-101.

[40] Jiang, Yi, Guangtao Xue and Jinyuan You. “Toward Fault-Tolerant Atomic Data Access in Mutable Distributed Hash Tables.” 1st International Multi-Symposiums on Computer and Computational Sciences, 2006. 485-490.

[41] Kemme, B., et al. “Using Optimistic Atomic Broadcast in Transaction Processing Systems.” IEEE Transactions on Knowledge and Data Engineering, 15.4 (2003): 1018-1032.

[42] Kemme, Bettina, et al. “Processing Transactions over Optimistic Atomic Broadcast Protocols.” Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, 1999. 424-431.

[43] Kindler, Ekkart. “Safety and Liveness Properties: A Survey.” Bulletin of the European Association for Theoretical Computer Science, 53 (1994): 268-272.

[44] Kivity, Avi, et al. “KVM: the Linux Virtual Machine Monitor.” Proceedings of the Linux Symposium, 2007. 225-230.

[45] Kogan, Alex and Erez Petrank. “Wait-free Queues with Multiple Enqueuers and Dequeuers.” Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, 2011. 223-234.

[46] Kotselidis, C., et al. “DiSTM: A Software Transactional Memory Framework for Clusters.” 37th International Conference on Parallel Processing, 2008. 51-58.

[47] Lagar-Cavilla, H. A., Whitney, J. A., Scannell, A. M., Patchin, P., Rumble, S. M., De Lara, E., et al. “SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing.” Proceedings of the 4th ACM European Conference on Computer Systems, 2009. 1-12.

[48] Lamehamedi, Houda, et al. "Data Replication Strategies in Grid Environments.” Proceedings of the 5th International Conference on Algorithms and Architectures for Parallel Processing, 2002. 378-383.
[49] Lamport, L. “Proving the Correctness of Multiprocess Programs.” IEEE Transactions on Software Engineering, SE-3.2 (1977): 125-143.
[50] Lamport, Leslie. “Paxos Made Simple.” ACM Sigact News 32.4 (2001): 18-25.
[51] Leibowitz, Nathaniel, Matei Ripeanu and Adam Wierzbicki. “Deconstructing the Kazaa Network.” Proceedings of the 3rd IEEE Workshop on Internet Applications, 2003. 112-120.
[52] Lua, Eng Keong, et al. “A Survey and Comparison of Peer-to-Peer Overlay Network Schemes.” IEEE Communications Surveys and Tutorials, 7.1-4 (2005): 72-93.
[53] Manassiev, Kaloian, Madalin Mihailescu and Cristiana Amza. “Exploiting Distributed Version Concurrency in a Transactional Memory Cluster.” Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2006. 198-208.
[54] Marathe, Virendra J and Michael L Scott. “A Qualitative Survey of Modern Software Transactional Memory Systems.” University of Rochester Computer Science Dept., Technical report (2004).
[55] Marathe, Virendra J, William N Scherer III and Michael L Scott. “Adaptive Software Transactional Memory.” Distributed Computing. Springer, 2005. 354-368.
[56] Marko, Kurt. “Research: 2012 State of the Data Center.” Available in http://reports.informationweek.com/abstract/6/8845/data-center/research-2012-state-of-the-data-center.html (May 2012) (2012).
[57] Miranda, H., A. Pinto and L. Rodrigues. “Appia, a Flexible Protocol Kernel Supporting Multiple Coordinated Channels.” 21st International Conference on Distributed Computing Systems, 2001. 707-710.
[58] Mishra, Asit K., et al. “Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters.” SIGMETRICS Performance Evaluation Review, 37.4 (2010): 34-41.
[59] Morton, G.M. A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. International Business Machines Company, 1966.
[60] Mosberger, David. “Memory Consistency Models.” SIGOPS Operating Systems Review, 27.1 (1993): 18-26.
[61] Muthitacharoen, Athicha, Seth Gilbert and Robert Morris. “Etna: A Fault-tolerant Algorithm for Atomic Mutable DHT Data.” Technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory (2005).

[62] Nieh, Jason and Ozgur Can Leonard. "Examining VMware." Dr. Dobb's Journal, 25.8 (2000): 70.

[63] Nishimura, Hideo, Naoya Maruyama and Satoshi Matsuoka. "Virtual Clusters on the Fly - Fast, Scalable, and Flexible Installation." 7th IEEE International Symposium on Cluster Computing and the Grid, 2007. 549-556.

[64] Nurmi, Daniel, et al. "The Eucalyptus Open-source Cloud-computing System." 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, 2009. 124-131.

[65] Nygren, Erik, Ramesh K Sitaraman and Jennifer Sun. "The Akamai Network: a Platform for High-performance Internet Applications." ACM SIGOPS Operating Systems Review, 44.3 (2010): 2-19.

[66] Palmieri, Roberto, et al. "Evaluating Database-oriented Replication Schemes in Software Transactional Memory Systems." IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum, 2010. 1-8.

[67] Palmieri, Roberto, Francesco Quaglia and Paolo Romano. "OSARE: Opportunistic Speculation in Actively Replicated Transactional Systems." 30th IEEE Symposium on Reliable Distributed Systems (SRDS), 2011. 59-64.

[68] Pedone, Fernando and André Schiper. "Optimistic Atomic Broadcast." Distributed Computing. Ed. Shay Kutten. Vol. 1499. Springer Berlin Heidelberg, 1998. 318-332.

[69] Pedone, Fernando and Andre Schiper. "Optimistic Atomic Broadcast: a Pragmatic Viewpoint." Theoretical Computer Science, 291.1 (2003): 79-101.

[70] Pouwelse, Johan, et al. "The Bittorrent P2P File-sharing System: Measurements and Analysis." Peer-to-Peer Systems, IV (2005): 205-216.

[71] Qian, Ling, et al. "Cloud computing: an Overview." Cloud Computing. Springer, 2009. 626-631.

[72] Ramabhadran, Sriram, et al. "Prefix Hash Tree: An Indexing Data Structure over Distributed Hash Tables." Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing, 2004.

[73] Raman, Suchitra and Steven McCanne. "A Model, Analysis, and Protocol Framework for Soft State-based Communication." Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, 1999. 15-25.

[74] Ratnasamy, Sylvia, et al. "A Scalable Content-addressable Network." ACM SIGCOMM Computer Communication Review, 31.4 (2001): 161-172.

[75] Ratnasamy, Sylvia, et al. "Topologically-aware Overlay Construction and Server Selection." Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies, 2002. 1190-1199.

[76] Rhea, Sean, et al. "Handling churn in a DHT." Proceedings of the USENIX Annual Technical Conference, 2004. 127-140.

[77] Rhea, Sean, et al. “OpenDHT: a Public DHT Service and its Uses.” ACM SIGCOMM Computer Communication Review, 2005. 73-84.
[78] Rimal, B.P., Eunmi Choi and I. Lumb. “A Taxonomy and Survey of Cloud Computing Systems.” 5th International Joint Conference on INC, IMS and IDC, 2009. 44-51.
[79] Ripeanu, Matei. “Peer-to-Peer Architecture Case Study: Gnutella Network.” Proceedings of the 1st International Conference on Peer-to-Peer Computing, 2001. 99-100.
[80] Romano, Paolo, Nuno Carvalho and Luis Rodrigues. “Towards Distributed Software Transactional Memory Systems.” Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, 2008. 4:1-4:4.
[81] Romesburg, H Charles. Cluster Analysis for Researchers. Lulu.com, 2004.
[82] Rowstron, Antony and Peter Druschel. “Pastry: Scalable, Decentralized Object Location, and Routing for Large-scale Peer-to-Peer Systems.” Middleware, 2001. 329-350.
[83] Scherer III, William N and Michael L Scott. “Contention Management in Dynamic Software Transactional Memory.” PODC Workshop on Concurrency and Synchronization in Java Programs, 2004. 70-79.
[84] Schmidt, Matthias, et al. “Efficient Distribution of Virtual machines for Cloud Computing.” 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2010. 567-574.
[85] Schutt, Thorsten, Florian Schintke and Alexander Reinefeld. “Scalaris: Reliable Transactional P2P Key/Value Store.” Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG, 2008. 41-48.
[86] Shavit, Nir and Dan Touitou. “Software Transactional Memory.” Distributed Computing, 10.2 (1997): 99-116.
[87] Shvachko, Konstantin, et al. “The Hadoop Distributed File System.” IEEE 26th Symposium on Mass Storage Systems and Technologies, 2010. 1-10.
[88] StackDriver. “These Data Centers are at the Center of the Cloud.” Available in http://www.stackdriver.com/public-cloud-data-centers/ December 2013.
[89] Stoica, Ion, et al. “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications.” ACM SIGCOMM Computer Communication Review, 31.4 (2001): 149-160.
[90] Vogels, Werner. “Eventually Consistent.” Communications of the ACM, 52.1 (2009): 40-44.
[91] Weinhardt, Christof, et al. “Cloud Computing - a Classification, Business Models, and Research Directions.” Business & Information Systems Engineering, 1.5 (2009): 391-399.
[92] White, Tom. Hadoop: the Definitive Guide. O'Reilly, 2012.

[93] Winick, Jared and Sugih Jamin. “Inet-3.0: Internet Topology Generator.” Technical report, 2002.

[94] Yocum, Ken, et al. “Scalability and Accuracy in a Large-scale Network Emulator.” ACM SIGCOMM Computer Communication Review, 32.3 (2002): 28-28.

[95] Zhang, Bo and Binoy Ravindran. “Relay: A Cache-Coherence Protocol for Distributed Transactional Memory.” Proceedings of the 13th International Conference on Principles of Distributed Systems. Berlin, Heidelberg: Springer-Verlag, 2009. 48-53.

Chang-Yen, Ian R. Bachelor of Science, Louisiana Tech University, Fall 2000; Master of Science, University of Louisiana at Lafayette, Spring 2003; Doctor of Philosophy, University of Louisiana at Lafayette, Fall 2014
Major: Computer Science
Title of Dissertation: Peer-to-Peer Architectures for Data Discovery, Distribution and Consistent Replication
Dissertation Director: Dr. Nian-Feng Tzeng
Pages in Dissertation: 136; Words in Abstract: 265

ABSTRACT

Despite the push towards consolidation of both data and computational resources into increasingly larger data centers, a majority of companies and organizations still rely on multiple small server rooms whose sizes do not exceed 5000 sq. ft. Our research proposes a Peer-to-Peer based architecture that provides two of the major services offered by consolidated data center systems, namely rapid deployment of large virtual machine (VM) images and applications and high-performance distributed storage. Our VM dissemination approach uses a new multi-layered design combined with a two-stage query mechanism. These enable the publishing and querying of dynamically-changing VM information, thereby reducing the deployment time of virtual machines and applications to clients. The opportunistic replication of VM data afforded by such dissemination mechanisms was further coupled with the replicated transactional mechanisms demonstrated within our Peer-to-Peer (P2P) based storage scheme. Such combined systems provide the deployed virtual machines and applications with a fault-tolerant, high-performance computational space upon which to concurrently store and retrieve large volumes of data in a mutually consistent manner. Our deployment architecture has been evaluated against existing VM-dissemination mechanisms and demonstrated significant improvements in VM deployment time, with at least an 8% improvement over existing high-performance content-distribution designs. Similarly, our data-storage architecture improves the performance of established

computational benchmarks by at least 22% over existing replicated storage mechanisms. Our developed approaches were also able to facilitate these improvements without a corresponding major increase in traffic overhead, even as the size of the evaluated systems increased. This demonstrated the scalability of our designs and their suitability for use in connecting large numbers of widely distributed data centers.

BIOGRAPHICAL SKETCH

Ian Chang-Yen was born in Trinidad and Tobago in 1977 and came to the United States in the summer of 1995. He completed his Bachelor's degree in Computer Science at Louisiana Tech University in 2000 and moved to Lafayette to pursue his Doctoral degree at the University of Louisiana at Lafayette in the spring of 2001. He completed his Master's Degree in Computer Science in 2003. Subsequently, he accepted a full-time position at the Center for Advanced Computer Studies as a Senior Systems Administrator, while continuing to pursue his Doctoral degree.