Managing high data availability in dynamic distributed derived data management system (D4M) under Churn

AHMED KAMAL MIRZA

KTH Information and
Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

Managing high data availability in dynamic distributed derived data management system (D^4M) under Churn

Ahmed Kamal Mirza

akmirza@kth.se

2012.05.17

School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

Abstract

The popularity of decentralized systems is increasing day by day. These decentralized systems are preferable to centralized systems for many reasons, specifically they are more reliable and more resource efficient. Decentralized systems are more effective in the area of information management in the case when the data is distributed across multiple peers and maintained in a synchronized manner. This data synchronization is the main requirement for information management systems deployed in a decentralized environment, especially when data/information is needed for monitoring purposes or some dependent data artifacts rely upon this data. In order to ensure a consistent and cohesive synchronization of dependent/derived data in a decentralized environment, a dependency management system is needed.

In a dependency management system, when one chunk of data relies on another piece of data, the resulting derived data artifacts can use a decentralized systems approach but must consider several critical issues, such as how the system behaves if any peer goes down, how the dependent data can be recalculated, and how the data which was stored on a failed peer can be recovered. In case of a churn (resulting from failing peers), how does the system adapt the transmission of data artifacts with respect to their access patterns and how does the system provide consistency management?

The major focus of this thesis was to addresses the churn behavior issues and to suggest and evaluate potential solutions while ensuring a load balanced network, within the scope of a dependency information management system running in a decentralized network. Additionally, in peer-to-peer (P2P) algorithms, it is a very common assumption that all peers in the network have similar resources and capacities which is not true in real world networks. The peer's characteristics can be quite different in actual P2P systems; as the peers may differ in available bandwidth, CPU load, available storage space, stability, etc. As a consequence, peers having low capacities are forced to handle the same computational load which the high capacity peers handle, resulting in poor overall system performance. In order to handle this situation, the concept of utility based replication is introduced in this thesis to avoid the assumption of peer equality, enabling efficient operation even in heterogeneous environments where the peers have different configurations. In addition, the proposed protocol assures a load balanced network while meeting the requirement for high data availability, thus keeping the distributed dependent data consistent and cohesive across the network. Furthermore, an implementation and evaluation in the PeerfactSim.KOM P2P simulator of an integrated dependency management framework, D^4M, was done.

In order to benchmark the implementation of proposed protocol, the performance and fairness tests were examined. A conclusion is that the proposed solution adds little overhead to the management of the data availability in a distributed data management systems despite using a heterogeneous P2P environment. Additionally, the results show that the various P2P clusters can be introduced in the network based on peer's capabilities.

Sammanfattning

Populariteten av decentraliserade system ökar varje dag. Dessa decentraliserade system är att föredra framför centraliserade system för många anledningar, speciellt de är mer säkra och mer resurseffektiv. Decentraliserade system är mer effektiva inom informationshantering i fall när data delas ut över flera Peers och underhållas på ett synkroniserat sätt. Dessa data synkronisering är huvudkravet för informationshantering som utplacerade i en decentraliserad miljö, särskilt när data / information behövs för att kontrollera eller några beroende artefakter uppgifter lita på dessa data. För att säkerställa en konsistent och härstammar synkronisering av beroende / härledd data i en decentraliserad miljö, är ett beroende ledningssystem behövs.

I ett beroende ledningssystem, när en bit av data som beror på en annan bit av data, kan de resulterande erhållna uppgifterna artefakter använd decentraliserad system approach, men måste tänka på flera viktiga frågor, såsom hur systemet fungerar om någon peer går ner, hur beroende data kan omräknas, och hur de data som lagrats på en felaktig peer kan återvinnas. I fall av churn (på grund av brist Peers), hur systemet anpassar sändning av data artefakter med avseende på deras tillgång mönster och hur systemet ger konsistens förvaltning?

Den viktigaste fokus för denna avhandling var att behandlas churn beteende frågor och föreslå och bedöma möjliga lösningar samtidigt som en belastning välbalanserat nätverk, inom ramen för ett beroende information management system som kör i ett decentraliserade nätverket. Dessutom, i peer-to-peer (P2P) algoritmer, är det en mycket vanlig uppfattning att alla Peers i nätverket har liknande resurser och kapacitet vilket inte är sant i verkliga nätverk. Peer egenskaper kan vara ganska olika i verkliga P2P system, som de Peers kan skilja sig tillgänglig bandbredd, CPU tillgängligt lagringsutrymme, stabilitet, etc. Som en följd, är peers har låg kapacitet tvingade att hantera sammaberäkningsbelastningen som har hög kapacitet peer hanterar vilket resulterar i dåligsystemets totala prestanda. För att hantera den här situationen, är begreppet verktygetbaserad replikering införs i denna uppsats att undvika antagandet om peer jämlikhet, så att effektiv drift även i heterogena miljöer där Peers har olika konfigurationer. Dessutom säkerställer det föreslagna protokollet en belastning välbalanserat nätverk med iakttagande kraven på hög tillgänglighet och därför hålla distribuerade beroende datakonsekvent och kohesiv över nätverket. Vidare ett genomförande och utvärdering iPeerfactSim.KOM P2P simulatorn av en integrerad beroende förvaltningsram, D4M, var gjort

De prestandatester och tester rättvisa undersöktes för att riktmärka genomförandet avföreslagna protokollet. En slutsats är att den föreslagna lösningen tillagt lite overhead för förvaltningen av tillgången till uppgifterna inom ett distribuerade system för datahantering, trots med användning av en heterogen P2P miljö. Dessutom visar resultaten att de olikaP2P-kluster kan införas i nätverket baserat på peer-möjligheter.

Acknowledgement

This thesis would not have been possible without the support and wisdom of many respected and loving people around me. I would like to start by expressing my deepest gratitude to my immediate supervisor, Karsten Saller, and Prof. Schurr, for their invaluable assistance, continuous support and guidance. Their knowledge, encouragement, and leadership from the initial to final stage of project, were key motivating factors to my gaining a deep understanding of the subject and finally, completing this thesis project. One simply could not wish for a better or a friendlier supervisor. I am indebted to him more than he knows.

I would also like to thank TU-Damstard administration for facilitating and providing all the necessary assistance for my research work. Their in-time support and co-operation were very helpful during my stay at TU-Damstard.

I owe my deepest gratitude to Prof. Gerald Q.Maguire, my thesis supervisor at KTH for helping me to understand and pursue this thesis. His guidance on every step was helpful to fully comprehend the tasks at hand. I am really grateful for his timely inputs and advice. I will also thank the KTH administration and especially my current and former program coordinators, Ms Susy and Ms Jenny Lundin for managing all the activities in KTH during my studies and especially arranging an Erasmus studentship for this thesis project. Without their support and help this project would not be possible.

Last but not the least; I would like to thank my research fellows, Fahad Azeemi and Waqas Liaqat Ali, for providing me with moral support, technical help, and guidance.

Special thanks to my parents, family, and teachers for providing their guidance and wisdom throughout my studies. Without their kind help and support I would not be where I am.

Ahmed Kamal Mirza

Dedication

I dedicate this thesis to my lovely sister, who unfortunately passed away in 2007. She is always in my thoughts and prayers. I will never be able to forget her. I know she is looking at me from heavens with proud feelings. I wish she was here to cherish these moments with me.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms and Abbreviations

API	Application programming interface
CAN	Content-addressable network
CDN	Content distribution networks
CFS	Co-operative file system
D^4M	Dynamically distributed derived data management
DHT	Distributed hash table
HTTP	Hypertext transport protocol
IMAP	Internet message access protocol
IP	Internet protocol
KBR	Key-based routing
LDPC	Low density parity check
LMS	Local minima search protocol
P2P	Peer-to-peer
PC	Personal computer
QRT	Query routing table
SIDM	Scalable Distributed Information Management System
SQL	Structured query language
TTL	Time-to-live

1 Introduction

This chapter describes the motivation for this thesis project. Next the chapter summarizes the contributions, the author has made to the field as a result of this thesis project. Furthermore, the chapter concludes with a description of the structure of the rest of the thesis in outline section.

1.1 Motivation

In current era of the internet, decentralized architectures are popular in different domains across industries, including online file-storage providers [22], network management applications [23], and online content repositories. Large-scale systems based on this architecture are classified as peer to peer (P2P) systems [20]. The utilization of these peer to peer systems may vary, but the ultimate goal is similar among them: to ensure resource efficiency, scalability [19], and availability [24] of information/data content.

As compared to a centralized architecture, a decentralized architecture provides more robust, reliable, and resource efficient services with a self-adapting mechanism. Most information management applications operating today are based on the manager-agent model [25]. In other words, there is one centralized management control program running on a central entity which is managing or computing via some management protocol. This architecture leads to poor scalability and reliability, as when more peers join the network the amount of traffic exchanged between the central management identity and the agent peers increases and can create a bottleneck for the system. Additionally, this central management entity can become a single point of failure causing poor reliability.

In a decentralized architecture, resource efficiency is always a challenging task. By employing scalable and reliable services using P2P technology, we must deal with a number of peers in which data artifacts are stored in a distributed fashion. These data artifacts can be categorized into base data and derived data. The base data represents independent data, which is atomic in nature while derived data is dependent data that relies on other data artifacts and can be computed by a combination of different data artifacts.

In an interconnected computing environment, the importance of derived data cannot be overlooked for analytic data processing. Such derived data might represent performance aggregates or some other sort of network monitoring information which is monitored at the network level rather than individual scalar performance factors. These scalar performance factors can be regarded as base data. Furthermore in the data warehouse domain, dashboard applications represent calculated scores based on data mining analytics; these scores are used to measure market trends. The data warehouses store this derived (i.e. dependent) data so that aggregate data queries can be responded to quickly. For this purpose, the dependency management functionality is needed to monitor these dependencies in a cohesive and nearly consistent way. The consistency cannot be fully guaranteed as the artifacts data and dependencies in all the peers may not be consistent and synchronized all the time due to network latency. In contrast, cohesive data can be ensured when peers are informed to execute a synchronization process to update the modified data artifacts and dependencies before responding to requests. These dependencies can be distributed among peers without introducing a centralized dependency server in such a way that a dynamically distributed derived data management (D^4M) framework [2] can operate on a P2P overlay.

Due to unpredictable behavior of the peers in a P2P system, the peers may leave the network either erratically or when they wish. When a peer leaves the network erratically, its data will be become inaccessible to the network, this can lead to inconsistencies in the distributed *dependent* data. There are a variety of redundancy protocols for P2P systems which ensure high data availability even in the

event of a peer crashing. However, these replication protocols require the peers to store redundant data which may negatively affect the overall performance of the system.

Part of the problem is due to the fact that the data manipulation and storage responsibilities are assigned to each peer irrespective of its properties and capabilities. In general, all distributed hash table (DHT) based protocols assume that each peer in the network is equal and has similar properties (with respect to CPU performance, network bandwidth, available storage space, etc.). However, in the real world, large scale data networks usually have peers with differences in capabilities. Therefore, a replication protocol that can perform load balancing across the network is needed. Such a protocol should assign the data replication operations to the peers based on their properties, as well as handle the complex data dependencies in D^4M. In order to perform this load balancing, utility-based replication protocol is proposed in this thesis.

1.2 Contributions

The main contributions of this thesis are associated with two tasks. One task was to identify possible approaches to implement and integrate the D^4M framework in a P2P simulator, specifically PeerfactSim.KOM [1],[17]. Each of these approaches was to be implemented and compared in the existing environment of this simulator. The second task was to introduce a novel algorithm to handle the problem of churn while ensuring no data was lost when using a data dependency management framework operating in a decentralized environment. In order to assure data availability, even during churn, several replication strategies were proposed and compared. The most suitable and efficient were selected for use in our simulation environment. These contributions can be summarized as:

- Propose several different ideas about how to implement the D^4M system in the existing PeerfactSim.KOM P2P simulator.
- Propose a high data availability solution suitable for a heterogeneous and D^4M system running on P2P systems while handling churn.
- Introduce a novel algorithm to handle churn in this decentralized D^4M system *without* affecting consistency in the data relationships and dependencies.
- Introducing a novel algorithm for a heterogeneous system in order to efficiently utilize the distributed resources in the network.
- Implementing and evaluating the proposed utility based replication algorithm using simulations.

1.3 Outline

The rest of the thesis is structured as follows. Chapter 2 reviews related work. Chapter 3 discusses the underlying technologies, introduces P2P overlays with several different flavors of DHTs, and presents some of the essential working details of PeerfactSim.KOM simulator. After this, Chapter 4 presents details of the D^4M framework and information management with an integration solution as implemented in PeerfactSim.KOM. In chapter 5, a detailed problem statement is given and the new approach used in this thesis project to solve this problem is compared with the traditional approach. Additionally, this chapter presents the proposed approach to handle replication, along with its architecture and implementation details. Finally, a performance evaluation of the proposed approach is discussed in Chapter 6. Chapter 1 presents some conclusions and suggests some future work.

2 Related Work

This chapter reviews related work concerning replication, redundancy schemes, and replica placement policies. It also summaries some published work concerning an improved DHT that enhances churn tolerance for the specific case of a P2P dependency management system. The definition of churn is given in section 5.1. The chapter concludes with some comments on a hierarchical tree based information management system that aggregates data about a large scale networked system.

The conventional method used for replication is mirroring in which the mirrors are normally aware of the other mirrors (or at least a subset of them). Mirror based systems include Usenet News[37], Akamai[38], Lotus Notes[39], and Internet message access protocol(IMAP)[40]. Besides mirroring, caching is also a widely used method for replication in wide area web protocols [41][42]. Although caching is a less organized replication method, it is highly suitable for environments where certain data is highly demanded, such as in content distribution networks (CDNs)[43][44]. CDNs are the sets of inter-operating caches which replicate highly demanded data in order to reduce the load on the content server and to provide end users with a performance improvement (as the cache will typically be located near to them, hence there will be a lower network delay to deliver the content from this cache).

In the context of decentralized systems, a variety of data replication strategies have been proposed for file storage systems, such as CFS [45], PAST [46], and OceanStore[47]. These file storage systems uses different kinds of replica placement protocols and redundancy schemes. A redundancy scheme dictates the format of stored data in the replicas, whereas a replica replacement protocol defines the selection criteria for replica peers in a network.

The redundancy schemes include simple replication schemes, erasure coding schemes (e.g. Reed Solomon and Low-density parity-check(LDPC) codes, and hybrid replication schemes. A simple replication scheme is used to achieve high availability and data persistence. In this type of scheme, a file is replicated to n different peers (replicas) in the network and the replica information is stored in distributed indexes (e.g. DHTs). Later when the file is requested, these indexes are accessed to select any of the replica-holding peers to respond to the request. When using this scheme, the file will be available if any replica-holding peer is available in the network. An erasure coding scheme was introduced in the very early days of P2P network in which a file is decomposed equally into m data blocks and encoded into n encoded blocks, which are distributed to n different peers. The file can be retrieved by any accessing m encoded blocks. Unfortunately when using this scheme, lookups and updates for a file generates considerable overhead for the system as each file request turns into m requests. The hybrid scheme is a combination of the simple replication and the erasure coding schemes. A comparative study [48] [49] of simple replication schemes and erasure coding schemes showed that an erasure coding scheme provides higher data availability than a simple replication scheme, but in the case of a P2P distributed dependency management system (D^4M) where dependent artifacts are distributed across the network, the erasure coding scheme leads to high traffic overheads when it updates artifacts and due to the extra traffic generated by the replica synchronization process.

The replica placement protocol plays an important role in determining the implementation cost of an efficient replication process. According to the recent literature, leaf-set based replication and multi-key replication are the two main basic replica placement protocols. In the leaf-set replication protocol, the data block is replicated to its owner's closest neighbors in its leaf-set. The leaf-set is the list of neighbors directly attached to it. The neighbors holding a replicated copy of its data block can be either its successors/predecessors or both. In other words, the owner's data blocks are replicated to its closest neighbors. Both PAST[46] and DHash[50] use this protocol. A variant of this protocol is the Successor Replication protocol in which only the immediate successors of the owner peer store the replica of its data blocks. The owner peer is the peer who stores the actual data block. In the multiple key based replication protocol, to replicate the data blocks on k owner peers, k different storage keys will be computed for each data block. In other words for each data block, k different keys will be

generated for k replicas. Both CAN[51] and Tapestry[52] employs the multi-key replication protocol. Multiple key based replication has variants in the form of *Path* and *Symmetric replication* [53][54]. To implement a replication for a P2P dependency management system (D^4M), the leaf-set based replication protocol is used with extended conditions in order to choose the closest peer which has a replica.

The replica placement protocol proposed in [55] uses a co-ordinated and controlled strategy which is globally known to each peer in the network in order to place the replicas. It uses the globally known hashing allocation function h(m, d) where $m \geq 1$ is the index number of a replica instance and d is the identifier of each document. The allocation (hashing) function provides the address of a DHT peer on which a replica can be placed. Using this hash function with either the actually number of replicas present or the location of the closest replica, any peer in the network can find a potential replica's address. In this replica selection protocol, locks are used during replica addition/deletion. These locks are used to avoid temporary inconsistencies in lookup while replica r_d is being modified. These temporary inconsistencies may lead to these replicas not being selected for document retrieval in the worst case. These inconsistencies cannot be tolerated in a P2P dependency management system (D^4M) where most of the artifacts depend on other artifacts in the system. Additionally, if there is an inconsistent artifact in the system, it may lead the system to an unreliable state, thus negatively affecting the reliability of data artifacts. Furthermore, the locking mechanism may negatively affect lookup performance as well.

The self-adapting replication protocol presented in [56] is designed to achieve high data availability in DHT-based systems. Its design is based on an erasure coding scheme so it cannot be implemented with a P2P dependency management system (D^4M) due to the large amount of traffic that would be exchanged in this scheme, as this would negatively affect the system's performance.

The RelaxDHT suggested in [57] enhances churn tolerance and provides a cost efficient maintenance protocol to handle a high churn rate. The main purpose of RelaxDHT is to avoid the transfer of a data block if there are still at least the desired numbers of replicas available in the network. The owner peer (root) uses the replicated localization metadata to locate its replica-holding peers. This metadata is introduced to reduce the overhead which is normally generated due to migration of data blocks when a peer joins or leaves the P2P network. The root peer does not store its own data, but only stores its replica set and keeps track of its root peer if it has a replica of other peer. Therefore, there would be always at least one replica at all times in the network. The only unknown issue with this resilient replication protocol is how it handles the concurrency of data updates since this is not described in literature. The architecture of RelaxDHT shows that any replica peer can serve the look up request and is allowed to update the data block.

For a P2P dependency management system (D^4M) it is very important to get up to date data, especially in eager mode operation. This is the basic difference between the RelaxDHT and the replication strategy proposed in this thesis. In the proposed replication strategy, there is no need to track the data concurrency because only a single root peer is responsible for update its data artifacts and it can only serve lookup queries, thus other replica peers are unable to serve the look up requests or update the replicated data artifacts. Additionally, data blocks in RelaxDHT are non-uniformly distributed among peers during DHT maintenance, which is normally a basic property of DHTs. This will also affect the lookup performance of the system.

The Scalable Distributed Information Management System (SIDM) [58] is a hierarchical tree based information management system that aggregates data about a large scale networked system. It provides scalability through hierarchical aggregation and flexibility to accommodate a wide range of applications and data attributes. Furthermore, it performs lazy aggregation, on demand re-aggregation, and tunable spatial replication to ensure robustness. It is based on Astrolabe [59] which is highly robust due to its unstructured gossip protocol for data distribution and replication of all aggregated

attribute values associated with a subtree to all peers in the subtree. SIDM has extra initial processing overhead to build the aggregation trees it needs to operate, as well as for additional overhead during DHT maintenance.

3 Background

This chapter introduces the technologies and concepts which are used in the thesis, along with a brief summary of their details. Since, a distributed data management is being discussed in a decentralized environment we will begin by introducing some basic terms, such as peer to peer (P2P) overlays along with the concept of flavors of structured and unstructured overlays. Later-on the architecture and details of the simulator PeerfactSim.KOM and the distributed data dependency framework are discussed.

3.1 P2P Overlays

A peer to peer (P2P) overlay network [20] is the logical decentralized network topology which runs on the top of a physical network, typically the Internet. This logical network consists of addressable interconnected nodes which share part of their resources, such as content, bandwidth, processing power, and/or printers using self-organizing scalable routing and messaging operations. Each node behaves in symmetric manner taking both client and server roles.

This P2P overlay architecture is implemented by P2P systems. In P2P systems, participating peers form an overlay network and connect to each other according to a given overlay protocol. The P2P overlay protocols can be categorized into two kinds of protocols: structured and unstructured overlay protocols. These protocols vary in terms of their network graph structure and routing architecture. Unstructured overlays are not relevant to the goals of this thesis, hence there are only briefly discussed.

3.1.1 Unstructured P2P Overlays

In P2P computing, unstructured overlays are considered to be a second generation of overlay networks. An example of such a network is Gnutella [3][6][26]. In contrast, centralized overlays were used in first generation networks such as Napster [21]. As depicted in Figure 1, in an unstructured overlay a node can only directly access its immediately adjacent nodes. In order to deliver messages to other nodes in the overlay a flooding mechanism or random walk mechanism is used [16].

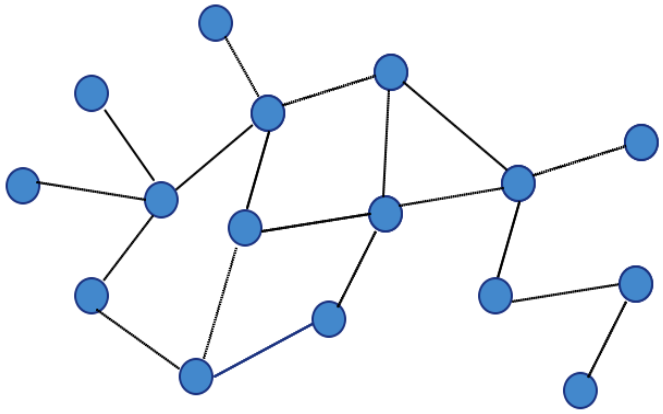


Figure 1: Unstructured Overlay network

An optimal network graph structure, efficient search, and efficient query propagation are the main design goals of an unstructured overlay. Some important unstructured overlays which addressed these design issues are listed in Table 1.

Table 1: Examples of unstructured overlays

Type	Design	Reference
Flooding	Gnutella, Fast Track	[3],[6] & [8]
Random Walk	Gia, local minima search(LMS)	[7],[9]
Hill climbing back tracking	Small world freenet	[12]
Preference directed queries	Tribler	[11]
Semantic routing	Internet-based Node Grouping Algorithms(INGA)	[10]

3.1.2 Structured P2P Overlays

Structured P2P overlays are third generation overlays. These are generally based on *Distributed Hash Tables (DHT)* [18] which employs key-based routing. In a structured DHT overlay, each node maintains a routing table. This routing table is employed by query propagation and routing algorithms, as specified by the overlay protocol. In other words, all nodes in the network cooperatively maintain routing data so that any one node can reach another node more efficiently (in terms of the number of hops) than is the case for unstructured overlays. The routing table helps nodes to forward queries closer to the target node thus reducing the time it takes until the query can be answered by the target node.

In order to maintain consistent routing data, nodes inform other nodes about changes in their routing table data, as specified by the selected overlay protocol. These changes can be a variation in network characteristics or a change in the offline/online state of a node in the overlay network. To ensure a node's activeness the protocol offers keep-alive services which send heartbeat messages to their neighbors and expect a reply. There is wide variety of structured DHT overlay protocols; some of the most influential protocols are Chord [13], Pastry [5], Kademlia [27], and Bamboo [28].

3.1.2.1 Chord

Chord [13] belongs to the family of DHT overlay protocols. Chord is scalable protocol which efficiently accommodates a node leaving or a new node joining the overlay network. Chord assigns keys to the nodes in the network using consistent hashing [4]. The use of consistent hashing ensures load balancing and scalability in such a way that during the bootstrap process or when a new node joins, keys are distributed uniformly, i.e. each node gets approximately the same number of keys. Chord uses SHA-1 [29] to hash the keys as it employs an m=160 bit identifier space. In other words, Chord uses a identifier space of 2^{160} -1 values for key assignment. Each node picks a random identifier by hashing its IP address to compute its position in the Chord ring. In the chord protocol, each node maintains a routing table which contains the number of neighbors specified in the protocol. The selection of neighbors is based on the node's own key. For instance, node K will select its neighbors in such a way that nodes in the network which have keys close to node K's key may be selected to become neighbors and their keys will be stored in node K's routing table, in order to perform routing. As a result the nodes arrange themselves in a ring fashion as depicted in figure 2, where the identifier space has an m=4-bit configuration.

In a Chord ring, data keys are stored according to the node's identifier. For example in figure 2, data key=10 will be stored at node 12 and data key=5 will be stored at node 5. The maintenance and look up operations details for Chord can be found in detail in [13].

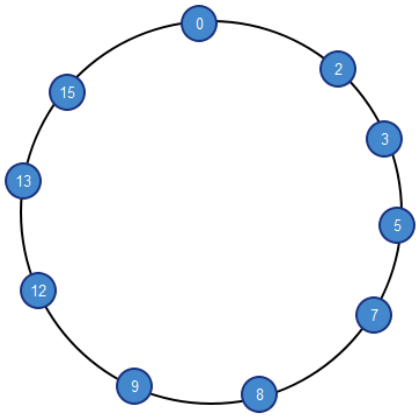


Figure 2: Chord ring with m= 4-bit (2^4-1)

3.1.2.2 Pastry

Pastry [5] is a fault-resilient, self-organizing, and robust structured P2P overlay protocol which makes routing and target positioning efficient. It is very similar to Chord [13], but in Pastry the identifier space is not organized as a Chord ring, rather a routing procedure based on numerical unique identifiers is used. In Pastry, when nodes join the Pastry overlay network, they are randomly assigned a unique 128 bit L-bit identifier in such a way that node identifiers are uniformly distributed within the 128-bit identifier space. This assigned nodeId is encrypted based upon a hash function of the node's IP address. Each Pastry node maintains a routing table, leaf set, and neighborhood set. The routing table contains the nodeIds of those nodes which have the same number of characters in their common prefix, grouped together in a row. The neighborhood set contains the nodeIds and IP address of nodes closest to the current node. This neighborhood set is not used for routing purposes, but rather to maintain locality properties as mentioned in [5]. The leaf set contains the number of nodes which are smaller than current node and the number of nodes which are larger than current node. The leaf set is employed for message routing. Pastry also exploits locality when routing messages in the overlay. In the absence of a node failure, it takes O(logN) steps to route a message to any node in the network. Some of the applications built on pastry are Scribe [15] and Past [16] [14].

3.2 PeerfactSIM.KOM Simulator

To implement and explore new innovative ideas, simulation is frequently used by researchers as a method of evaluating, comparing, and analyzing different systems [30]. Simulation is a modeling technique which provides an imitated environment in which the researcher can evaluate new approaches and concepts before creating a real world implementation. In P2P systems, this modeling has been a desirable way to test and evaluate different P2P protocols and their functionalities [31]. There are dozens of P2P simulators available for analysis, which vary in their functionalities and architecture, some of these simulators are PeerSim [32], PlanetSim [33], and Kompics [34].

PeefactSim.KOM [1][17] targets the general requirements of P2P simulators as given in [35] and [36]. It is a java-based P2P simulator for large-scale P2P systems which offers a simulated environment to execute a variety of P2P scenarios dealing with different kinds of protocols and functionalities. Additionally, it provides a user-friendly logging and statistics mechanism which facilitates the collection and interpretation of quantitative data during running simulations. This discrete-event based simulator has a layered-architecture which helps the layers to operate in a loosely coupled manner. In addition, the simulator can provide or utilize the services of other layers in the actual environment they are integrated in. In other words, each layer behaves as a component which

can be considered a plug-in for the simulator. Its modular design eases the implementation and integration of new components that can be defined in terms of its abstract base implementation. This modular design will be briefly discussed below along with its architecture details. Furthermore, visualization is integrated into the simulator to provide graphical visualizations of communication observed during simulations. This visualization can also be used for debugging purposes.

3.2.1 PeerfactSIM.KOM: General Concepts

During a simulation each peer has its own separate instance of each layer enclosed in it, as shown in Figure 3. This means a peer consists of a collection of layers which interact with each other using a message exchange process. For this reason, the simulator is considered as a message level simulator. In a message exchange process, each peer will communicate with other peers by sending and receiving messages. This communication is carried out using a lower layer, i.e. a network layer. The network layer utilizes the internet to send the message to another peer. The same approach is followed in the message receiving process.

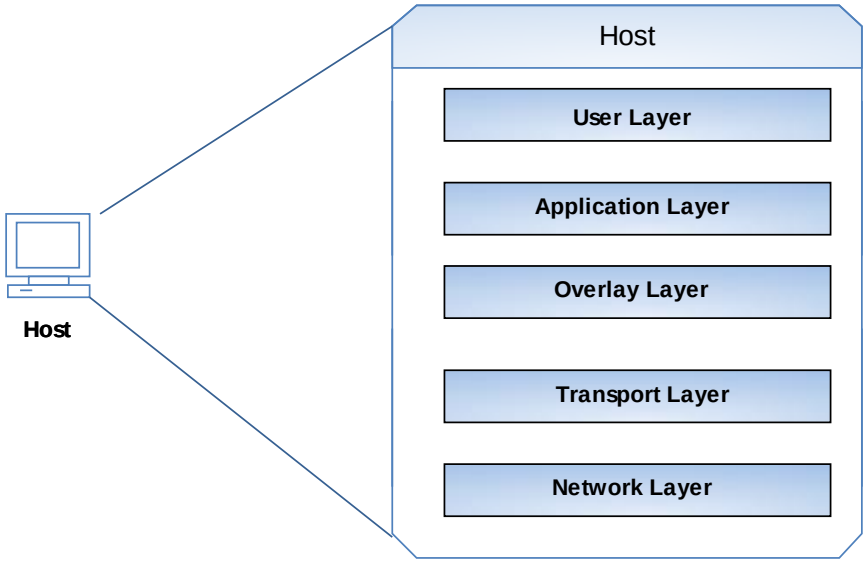


Figure 3: Host component during simulation

3.2.2 PeerfactSIM.KOM Architecture Overview

The layered architecture of the PeerfactSim.KOM simulator can be logically split into two main parts: the functional layers and the simulation engine as depicted in Figure 4. A functional layer in a simulator is comprised of components, providing well-defined interfaces for each component to expose its services and operations to other components and can communicate with others by exchanging messages. These well-defined interfaces allow the use of existing default implementations of components or to extend their concrete implementation. More specifically, this concept makes the simulator flexible for extension based development. Usually this architectural flexibility is a main requirement for simulators.

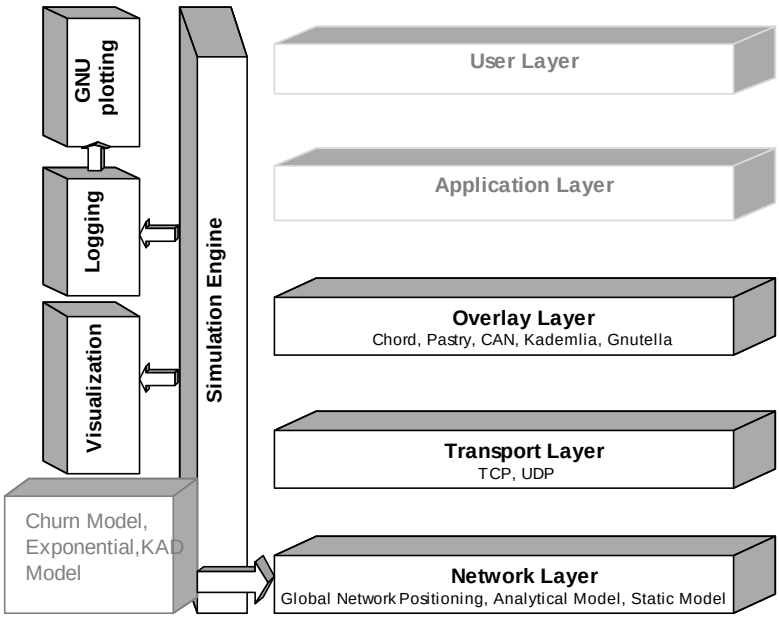


Figure 4: Layered Architecture

In order to provide this flexibility, the simulator introduces the concept of default and skeletal implementation. A default implementation is termed as implementation whose offered functionality is defined and implemented, and can be used without modification. For flexibility, the skeletal implementation is used (this is also termed abstract base implementation). To use this concept of a skeletal implementation, the concrete implementation of an interface can be tailored by extending the default implementation or by defining a new concrete implementation based on the abstract base implementation. The design and implementation of this concept is illustrated in Figure 5.

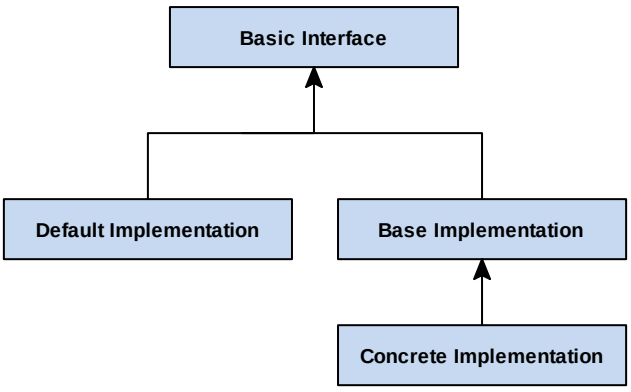


Figure 5: Represent Interface Implementation

3.2.2.1 Functional Layers

In the PeerfactSim.KOM simulator, the functional layers are components which provide services and operations to other layers and coordinate with lower and upper layers by exchanging messages with each other. During simulation every peer is represented by a Host as shown in Figure 3. The lowest three layers (the Network layer to Overlay Layer) are used in the implementation of the distributed

information management framework as discussed below.

Network layer

The network layer is the lowest layer of the PeerfactSim.KOM simulator which is based on a network model that allows peers to communicate with other peers in the simulated network with the help of the message exchange process. This model is based on two components: the *network layer* and *subnet*. The first component, the network layer, is installed as a separate component within each host during simulation. It is connected to the transport layer in a host and a lower component, called the subnet. The main purpose of this subnet component is to allow peers to communicate with other peers in a simulated network and to deal with other network aspects of a host. These network aspects include network latency, available bandwidth, exchange message size, and host status (i.e. if the host is offline or online). These aspects are discussed in [17] in detail. A subnet component is considered to be simulated network or internet through which all the peers communicate. The subnet simulates the transmission of data between hosts in the simulated network. This subnet component is a centralized identity and represents the network as depicted in Figure 6.

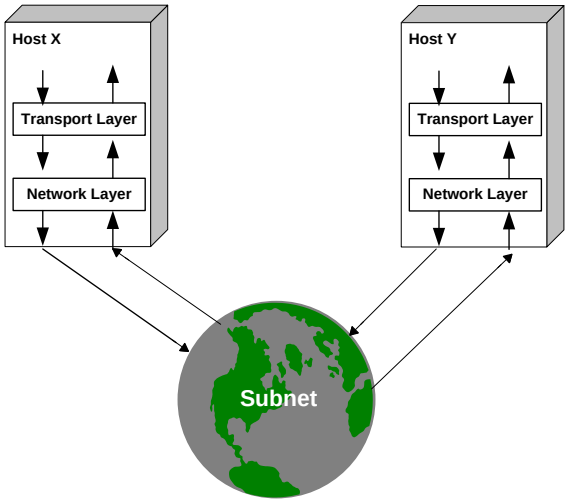


Figure 6: Each host is connected to the subnet which acts as the interconnection network

Each host is connected to the network, with the help of the subnet component. The subnet component can only be accessed through the network layer. In other words, the layers above the network layer can only send messages to another host in the network through the network layer, they cannot directly communicate.

The exchange process of the network layer component handles the sending and receiving of messages from the modeled network. When a host sends a message to another host, a message is pushed to the network layer, and then it is forwarded from the network layer component within a host to the subnet component, which is connected to all the hosts. This subnet component handles the transmission time calculations and imitates packet loss and induces jitter. Additionally, the subnet triggers the arrival of a message confirmation at the receiving host.

In order to use the given network layer in the simulator, the corresponding factory which is responsible for initializing the network layer components should be declared in the configuration file

as shown below in the code snippet in Figure 7. In this code snippet the network layer factory is declared as "*SimpleNetFactory*" which creates "*SimpleNetworkLayerFactory*" to build the network layer of each host and creates a centralized subnet identity for the simulation.

```
1 <Configuration>
2 [...]
3 <NetLayer  class="de.tud.kom.p2psim.impl.network.simple.SimpleNetFactory">
4 <LatencyModel
          class="de.tud.kom.p2psim.impl.network.simple.SimpleStaticLatencyModel"
          latency="10ms" />
5 </ NetLayer>
6 [...]
7 </ Configuration>
```

Figure 7: Representation of simple network layer in configuration file

Transport Layer

The transport layer of PeerfactSim.KOM shown earlier in Figure 4, is a higher layer than the network layer. The transport layer is connected to the overlay layer at one end and the network layer at the other end as was shown in Figure 6. The transport layer provides an end-to-end communication service to higher layers within host. These services include multiplexing using ports over a single connection, connection-oriented data streams, and flow control. A transport layer's details are abstract and its implementation depends upon the services being offered. The main task of the transport layer is to provide efficient simulations of the underlying network to higher layers.

In the simulator the implementation of the transport layer presents some standard and basic interfaces, and abstract classes. These interfaces define a network layer address along with a particular port number which is used in multiplexing multiple transport connections via single network connection. Additionally, transport message types are also specified with the help of these interfaces to allow listeners or event handlers to receive the incoming transport messages at the given port. These listeners are employed to catch events or messages sent by higher layers within a host and can notify the higher layers about the arrival of new incoming messages from other hosts in the network. The transport layer services can be used for two kinds of communication: (1) to forward messages from higher layers to the network layer within a host which further sends these messages to their respective hosts and (2) to deal with incoming messages from other hosts by forwarding them to the higher layers for further operations.

Furthermore, communication between hosts in the network can be done synchronous as well as asynchronous with the help of these interface implementation by using callback operations. This communication can be based on TCP messages or UDP messages. TCP is a reliable and connection-oriented protocol; while UDP provides unreliable and connectionless services. This layer also provides the timeout functionality for the sending operations to other hosts for both kinds of messages. The main purpose of this timeout functionality is to ensure that a target host sends back a reply for each message in order to ensure reliable communication within a given time bound, otherwise a timeout occurs it the nodes resends the message until it receives an acknowledgement message from the target host.

In order to implement the transport layer in the simulator, a Transport Layer factory is declared in the configuration file which is responsible for initializing the Transport Layer services.

The default transport layer factory is defined as shown in the code snippet in Figure 8.

```
1 <Configuration>
2 [...]
3 <TransLayer  class="de.tud.kom.p2psim.impl.transport.DefaultTransLayerFactory" />
4 [...]
5 <Configuration>
```

Figure 8: Representing Transport layer in configuration file

Overlay Layer

As overlay functionality is important for P2P simulators, in PeerfactSim.KOM the overlay layer plays a vital role which is encapsulated in the overlay layer. This encapsulation enables a programmer to easily implementation different P2P overlay models. As noted earlier in section 3.1, in general the overlay models can be classified into structured, unstructured, and hybrid overlays; but in our simulator only the structured and unstructured overlays are relevant.

In the simulator each peer is termed a overlay node. To implement an overlay node, the simulator provides interfaces to perform the operations and functions of the overlay node. These are shown in Figure 9: Representation of Overlay Node in overlay.

The purpose of exposing these interfaces is to allow the developer to vary the structure of the overlay routing table and the bootstrap mechanism provided by an overlay. These interfaces specify the structure and functionality of the overlay node in the simulated network. Interface <OverlayNode> specifies how an overlay node is represented in the simulator, whereas the inherited interface <JoinLeaveOverlayNode> dictates the joining and leaving operation of an overlay node in structured overlays during simulation. The interface <UnstructuredOverlay> represents the specific functionality of an overlay node in unstructured overlays. KBR stands for Key-based routing which is needed in structured overlays. <DHTNode> incorporates the DHT operations.

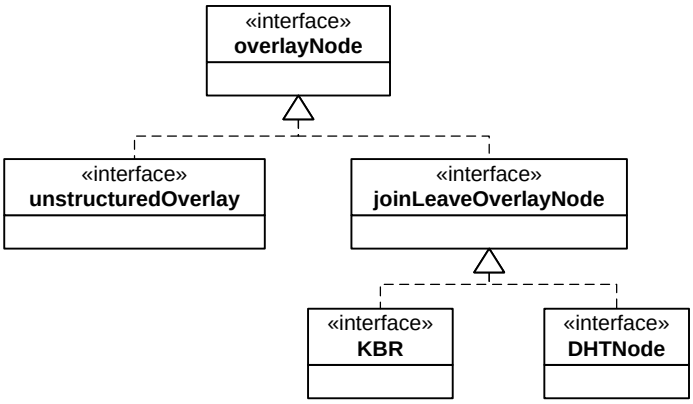


Figure 9: Representation of Overlay Node in overlay

The simulator can implement unstructured overlays, including Gnutella 0.4, Gnutella 0.6, and Gia; and structured overlays including CAN, Chord, Kademlia, andPastry. These overlays can be used in the simulator simply by declaring their factory implementation class in the configuration file. In addition to the above mentioned overlays, the simulator implements other types of overlays as well, but these are outside the scope of this discussion.

Some of the common overlays which are implemented in this simulator are discussed below.

a) Gnutella-like Overlays

Gnutella is a distributed search protocol which provides a fault tolerant decentralized model for unstructured overlays. In these overlays a multi-hop ping service is used to discover the peers in the network, using TTL (time-to-live). Its poor scalability led to the use of distributed hash tables for file-sharing applications. In the simulator a Gnutella API is available which offers basic connectivity methods (join() and leave()). Additionally, this API exposes some interfaces for publishing and querying documents and data. These interfaces are described in detail in the PeerfactSim.KOM manual.

b) Gnutella 0.6

Gnutella 0.6 is a modified version of the basic Gnutella 0.4 protocol as implemented by LimeWire.In this overlay, the network is divided into ultra-peers and leaves. Leaves having little bandwidth act as clients and ultra-peers are nodes that have a large amount of available bandwidth. The ultra-peers are used to manage Gnutella overlay traffic, these nodes act as servers. The leaves are connected to multiple ultra-peers to improve robustness. The ultra-peers maintain a query routing table (QRT) which stores the resources of connected leaves that a peer is sharing. This modified Gnutella provides a dynamic querying process which starts from the immediate ultra-peer neighbors and forwards the query from these immediate ultra-peers to further ultra-peers until it finds the sufficient results for the query.

In the simulator, the implementation employs the functionality of the LimeWire client. The protocol uses UDP and binary messages for transmission among peers, as this generate less-traffic than the ASCII HTTP communication used in Gnutella. A brief description of the implementation details are given in the PeerfactSim.KOM manual.

c) Gia

Gia improves upon the Gnutella –like overlays. These overlays are based on the functional values of the overlay nodes. These functional values can vary in terms of available bandwidth, storage space, or other functional aspect. A capacity value is associated with each overlay node. This capacity value is employed for querying process and the connection making mechanism. Gia offers means of dealing with low capacity nodes, provides a replication strategy to ensure consistency, and offers a querying mechanism as described in the PeerfactSim.KOM manual. In the simulator, Gia can be used by the common Gnutella API by declaring its factory component in the configuration file.

d) CAN

A Content-Addressable Network (CAN) belongs to the DHT family. It creates a node topology in a d-dimensional Cartesian coordinate space. This coordinate space is employed to store key-value pairs. This kind of overlay can organize itself. The details of the required operations (join(), leave(), lookup(), and store()) are discussed briefly in the PeerfactSim.KOM manual.

In order to use this overlay, the simulator provides a CAN API for its operations and services. This can be used by declaring the CanNodeFactory component in the configuration file of the simulator.

e) Chord

The Chord protocol is also a member of the DHT family. Its underlying concepts were described in section 3.1.2.1. In the simulator Chord can be used by specifying the ChordNodeFactory component in the simulator's configuration file as shown in Figure 10. Further implementation details are discussed in the PeerfactSim.KOM manual.

```
1 <Overlay class="de.tud.kom.p2psim.impl.overlay.dht.chord2.components.
2     ChordNodeFactory" />
```

Figure 10: Declaration of ChordNodeFactory in a configuration file

3.2.2.2 Simulation Engine

The simulation engine is a discrete-event based component in the simulator which manages the simulated peers in the network. These peers can communicate to each other by exchanging messages. Each layer in the peer can be accessed by this engine for logging purposes. The architecture of the simulation engine is comprised of the two components explained below:

Event Scheduler

With the help of the event scheduler, the simulator engine schedules events for execution at a certain timestamp. The method *scheduleEvent()* is executed to schedule an event (before it triggers). Any event can be scheduled immediately or after a certain timestamp. In addition, an event can be scheduled once or more than once. An event is associated with a certain operation such that when an event triggers, this operation is executed and can trigger other events. The main purpose of the event scheduler in simulator is to schedule operations of each layer within each host. The host will in turn execute these operations at the scheduled time. For instance, the scheduler is employed to carry out the stabilize operation in the overlay layer in order to refresh the overlay's routing table.

To process the events a logical timestamp is considered for event occurrences and their execution. For instance, an event A is scheduled to execute at timestamp $t_A = 10$ and the next event B is scheduled at $t_B = 100$. In this case the scheduler first gets event A, processes it and updates the current timestamp to 10 and go on to the next event, i.e., event B. When the scheduler extracts event B from the event queue, it processes this event and will update the current timestamp to 100.

Event Queue

The EventQueue is an ordered list of future simulated events which are to occur. A timestamp is associated with each simulated event. The timestamp represents the time at which this event will occur and upon its occurrence it will notify the associated simulation handler to do some operation. In each simulation step, the scheduler accesses the earliest event in EventQueue, calls its corresponding handler and performs the specified actions. This process is carried out until the EventQueue is empty. The further details can be found in PeerfactSim.KOM manual.

4 Information Management with D^{4}M Framework

In this chapter, the architecture and the functionality of the Dynamic Distributed Derived Data Management Framework (D^{4}M) are discussed. This chapter gives details of how to handle the remote data artifacts and dependent data in a coherent and *eventually consistent* way. In order to bring more understanding, the brief overview of framework is discussed in working scenario. The D^{4}M framework is specially designed for the management of dependent and derived data distributed across multiple peers in peer to peer (P2P) network.

4.1 Basic Idea

The D^{4}M Framework is a self-adapting decentralized derived data management and monitoring framework which works on P2P overlays and offers consistent and cohesive management of data and its dependent artifacts. These data artifacts are stored in different peers by employing the basic services of P2P overlays, such as Chord, Pastry, Kademlia, etc -- without using centralized dependency management servers. In general, the term consistency in P2P systems means ensuring data reliability, the state in which the respective peers have synchronized data with recent updates applied. The term cohesion means ensuring that the data is consistent with recent updates which are not yet applied on data but informed of these recent updates. Therefore, in D^{4}M the nodes must manage the data in such a way that data which is not yet synchronized with recent updates but is informed of the recent updates so that if the data is requested it will be first updated then will be delivered to the requestor. In this framework consistent and cohesive terms are distinguished with respect to the synchronized and update mechanism. Consistent data is synchronized and updated among respective peers all the time. In contrast, for cohesive data it may be possible that the data is not updated among the respective peers all the time, but those peers are informed that they must execute some update mechanism when this data is requested or pulled. As a result cohesive data may also be considered as updated and synchronized among the respective peers, but only upon demand for the data. The data propagation in the eager mode of the framework guarantees consistent data, while a lazy mode ensures only cohesive data. The D^{4}M Framework can be considered a milestone for data dependency management techniques in decentralized environments as still there exists no mechanism for self adapting and effective data management in graph-based structures.

Using the D^{4}M Framework, a variety of different system parameters and states can be monitored, enabling efficient and timely monitoring of networks in network monitoring applications. This approach can also be utilized where decentralized systems are being used, for example Wiki engines [60], social knowledge networks, and distributed development environments.

4.2 Architecture

In this framework, the data components are categorized in two kinds of data artifacts i.e. Basis and Derivatives. Basis represents atomic data having no incoming dependencies but may be involved in computation of the derived data with the help of different requirement artifacts using certain computation function. Additionally, it has a list of dependency artifacts which is used to propagate its updated value to all the peers holding its dependency artifacts. Derivatives are derived data collections which may be dependent on some other derivatives or basis. The derivatives are data artifacts which are generated using computation function. With each derivative there is a list of dependency artifacts and requirement artifacts, and at least one computation function which computes the derived data using its requirement artifacts. The internal structure of basis and derivative are illustrated in Figure 11.

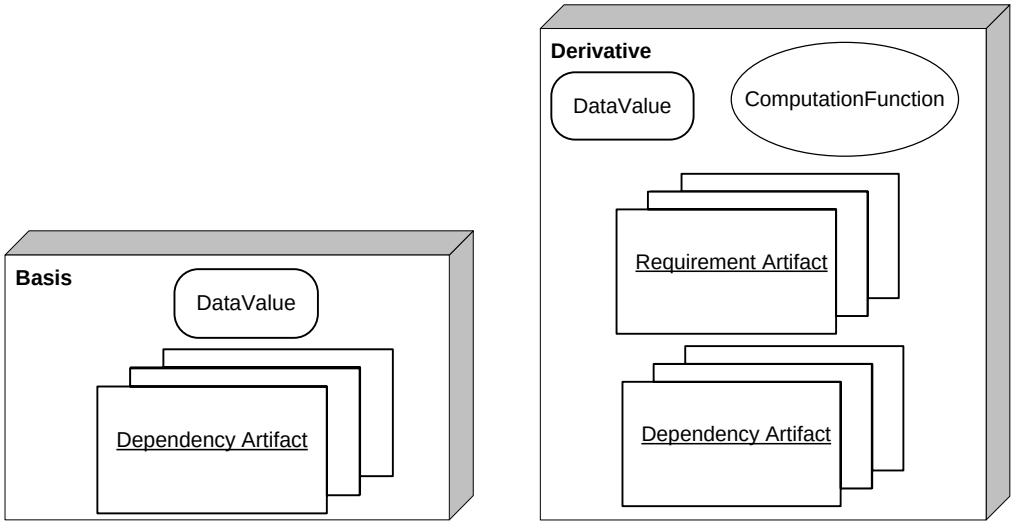
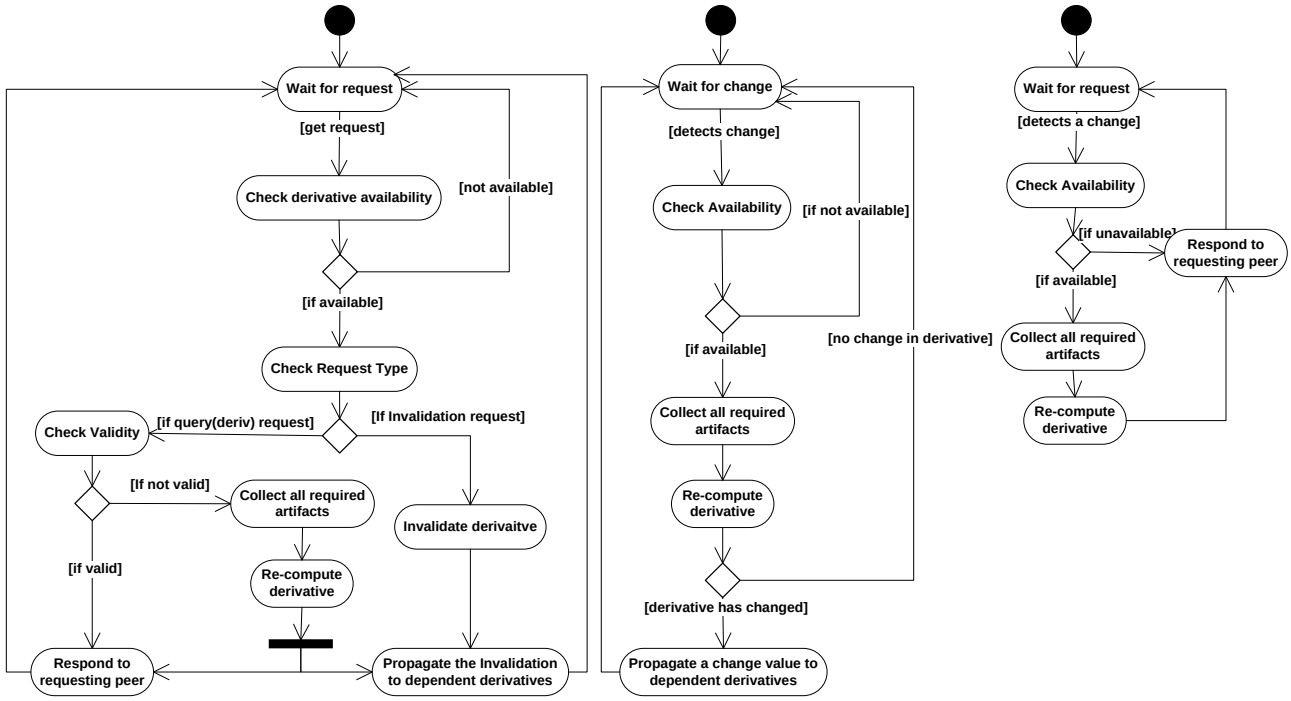


Figure 11 Artifact's data structure representation

The required/requirement artifacts are those artifacts which are needed to compute the respective derivative value. The dependency artifact represents the derivative whose data value should be updated if one of its Basis, computation function or required Derivatives have been changed. The requirement artifacts may be basis or derivative while the dependency artifacts can only be derivatives.

The D^{4}M framework operates in different modes of cache and data propagation which helps the derived data to re-compute with minimum number of computing steps and to distribute the data efficiently when it is required. These operating modes vary in terms of their activeness and prompt action. The three identified operating modes are illustrated in Figure 12.



(a) Lazy Mode (b) Eager Mode (c) Quiet Mode

Figure 12: Activity Diagrams of D^{4}M operating modes

Eager	If there is a change in artifact data (basis/derivative), its data value is distributed to all peers immediately who are holding its dependent derivatives. The cause of a change in artifact data may be the result of any derivative re-computation or basis value update which is present in its list of dependency artifacts. It is illustrated in given figure 12(b).
Lazy	Derivatives are re-computed and cached locally. The re-computed derivative is distributed to interested peers when they request it otherwise it will remain in local cache. Additionally, the peer who re-computed the derivative, send invalidation derivative message to the interested peers as depicted in figure 12(a). In this process interested peers are those who are having artifacts which depend on this re-computed or updated derivative. It is more relevant to the peers who are having derivatives.
Quiet	Derivative data is re-computed and distributed for each request but never be cached or distributed in advance. The whole process is explained in activity diagram figure 12(c).

4.3 Concrete Scenario

For the better understanding, the use case of a distributed relational database system operating in environment with federation of servers (given in Figure 13 where peers are arranged with respect to chord protocol) is discussed where data is stored across servers in the form of tables governing certain schema.

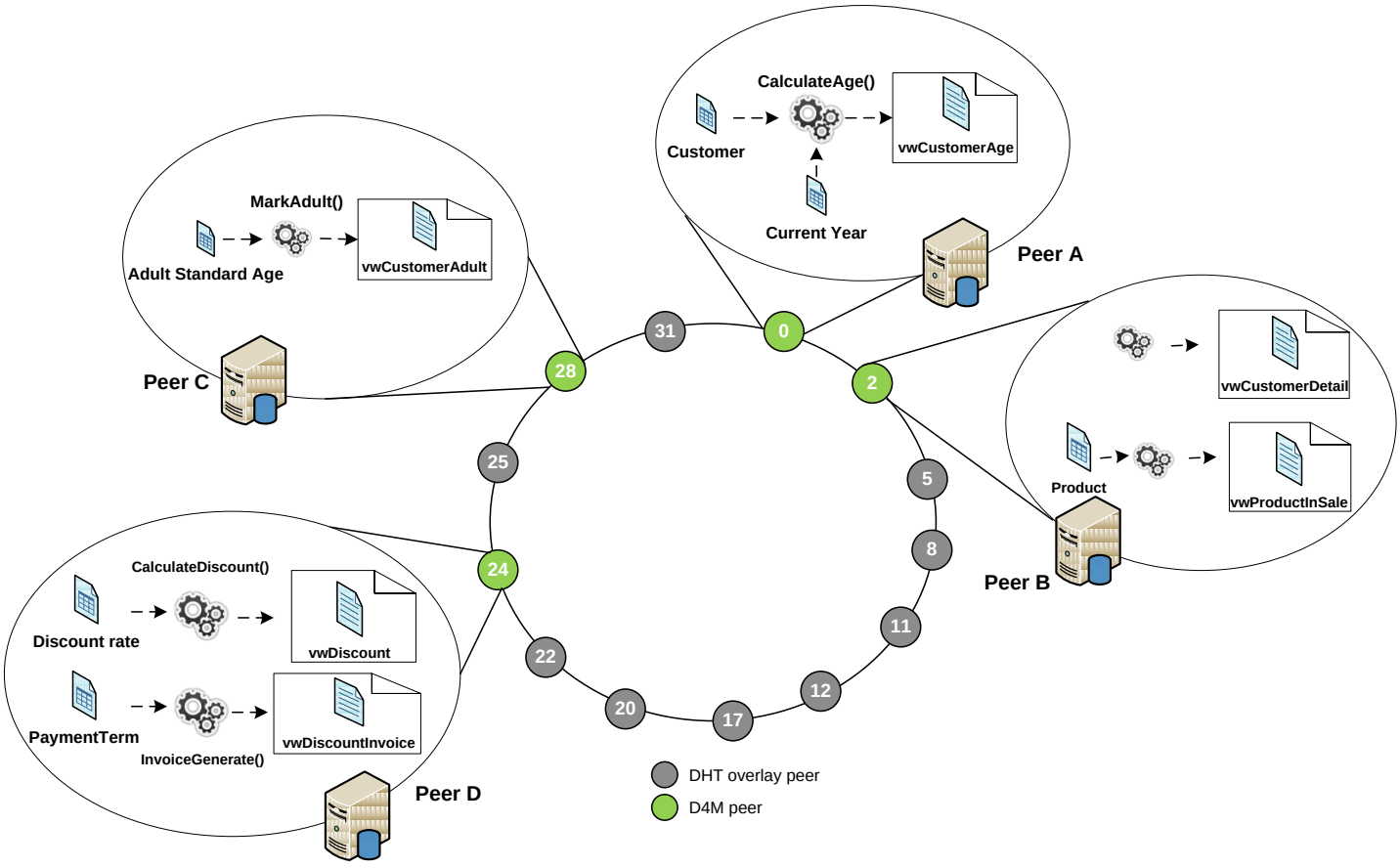


Figure 13: Federation of servers operating chord protocol with data stored

These data tables are structured using formal schema definition which specifies the columns types and other additional information like reference keys and their structure related information. Furthermore, there are views which help to access and manipulate the data from and to data tables. The view is the set of SQL statements which queries and updates the data stored in tables. In addition, certain tables are having derived columns which can be populated based on some other column values. The derived column is a computed column whose data value can be calculated by the other column data using certain operating function e.g. concatenation of columns 'FirstName' and 'LastName' may result in derived column 'FullName' which concatenates the first and last name. In this scenario we have a simple database schema as given below in table 2.

Table 2: Database schema definition

Data Tables Definition
Customer(CustomerID, Name, DateOf Birth, Age)
Product(ProductID,Name, StandardPrice, ListPrice, SalePrice)
Order(OrderNo,OrderDate,Status)
OrderDetail(OrderDetailID,OrderNo,ProductID, Quantity, UnitPrice, TotalAmount)
Discount(TransactionID, CustomerID, OrderNo, DiscountedAmount)
PaymentTerm (TermId, PaymentType, TermName)
Invoice(InvoiceId, InvoiceAmount, TransactionId)

In a given database schema definition, table Customer has column '*Age'* as its derived column that can be computed based on column '*DataOf Birth*' and current year. Similarly, the column '*SalePrice*' in a table *Product* is the derived column based on the difference of *StandardPrice* and *ListPrice* i.e. *SalePrice = difference(ListPrice, StandardPrice).* In a table *OrderDetail*, the derived column *TotalAmount* can be computed using formula *TotalAmount = UnitPrice * Quantity*, and later will be used to compute the column '*DiscountedAmount'* for table *Transaction*. The table *PaymentTerm* is used to generate the column '*InvoiceAmount'* for table *Invoice* based on payment term type.

Whenever there will be change in the columns data which are needed to compute the derived data columns, all the dependent derived columns should be updated accordingly. In order to map D^4M in this scenario environment, the distributed database systems are used in which the tables are stored across multiple peers based on certain structured overlay protocol like Chord, Pastry or Kademlia etc. The database servers in a federation are termed as the peers in this context. The non-derived columns in tables are considered as basis and derived columns are the derivatives which are computed with the help of non-derived columns and certain computational function.

On peer A, a basis *Customer* table is stored which has the derived columns *Age* and *isAdult* which can't be read directly from the table. In this scenario, there are certain views exposed to read the derived columns otherwise the derived columns can't be either readable or updated directly from table. The derived column *Age* is computed by the computation function *CalculateAge()* with the parameters *Customer* record from the table and the *CurrYear*, in a result it updates the computed age in the column *Age* which can be readable with the help of a view *vwCustomerAge*. Therefore, *CurrYear* and *Customer* record are the requirements to compute the *Age*. These requirements are also the dependencies in opposite way like the *vwCustomerAge* is a dependency arifact of table *Customer* and

CurrYear. In simple words when there is a change either in table *Customer* or *CurrYear*, their dependent artfact i.e. *vwCustomerAge* will be computed again. Similarly on peer C, to calculate the *isAdult derived* column with the help of its view *vwCustomerAdult* using computation function *MarkAdult()*, the locally stored table *AdultStandardAge* and remotely stored view *vwCustomerAge* are the requirements to compute *vwCustomerAdult*.. There is a table *OrderDetail* which has *Amount* as derivied column. The column *Amount* value read by the view *vwOrderAmount*, can be calculated using *CalculateAmount()* with the help of table *OrderDetail* which is stored locally. In a same way, the peer C stores the view *vwCustomerDetail*, which allows concatenating the *vwCustomerAge* and *vwCustomerAdult* and produce the value of *vwCustomerDetail* to read the complete details of a customer and its order. Similarly, on peer C the table *Product* is having the derived column SalePrice which can be computed using ListPrice and StandarPrice from the same table stored locally. On peer D, the view vwDiscount calculates the discount amount based on discount rate provided by the table *DiscountRate*. On the same peer, invoice amount can be calculated with the help of discounted value given by vwDiscount and the payment term given by table *PaymentTerm*. The table *PaymentTerm* specifies the terms of payment to calculate the invoice like visa card, cash etc. Therefore, the final invoice amount will be calculated by computation function *InvoiceGenerate()* using payment term and *vwDiscount* value. The computation functions specified can be defined as follows in Table 3.

Table 3: Computation function definition

Computation function	Function definition
CalculateAge(CustomerDOB, CurrYear)	Age := CurrYear - CustomerDOB

MarkAdult(AdultStandard, vwCustomerAge)	If vwCustomerAge.Age >= AdultStandard then
	isAdult = true
	else
	isAdult = false;
	vwCustomerAdult := isAdult

CalculateAmount(OrderDetail)	Amount := OrderDetail.UnitPrice * OrderDetail.ProductCount vwOrderAmount:= Amount
ConcatenateCustomer(Customer, CustomerAdult)	CustomerFullDetail:=CustomerDetail + vwCustomerAdult vwCustomerDetail:=CustomerFullDetail
CalculateDiscount(vwOrderAmount, Discountrate)	DiscountedAmount := vwOrderAmount – DiscountedAmount vwDiscount : = DiscountedAmount
InvoiceGenerate(vwDiscount, PermanentTerm)	If PermanentType == Cash InvoiceAmount:= vwDiscount Else if PermanentType == 'Visa card' InvoiceAmount:= vwDiscount + VisaCharges vwDiscountInvoice : = InvoiceAmount

When there will be change in records of simple tables discussed in scenario, their updates will be sent to the remote peers where their requirements and dependencies are stored. These requirements and dependencies are associated with each unit in the scenario. The discussed use case is shown in the

given Figure 14. It will be referenced later for the complex examples. In order to map this scenario with dependency management system, the views whose values is derived can be considered as derivatives and the tables or units like *CurrYear, AdultStandatdAge* can be considered as basis whose data value does not depend on any artifact or view.

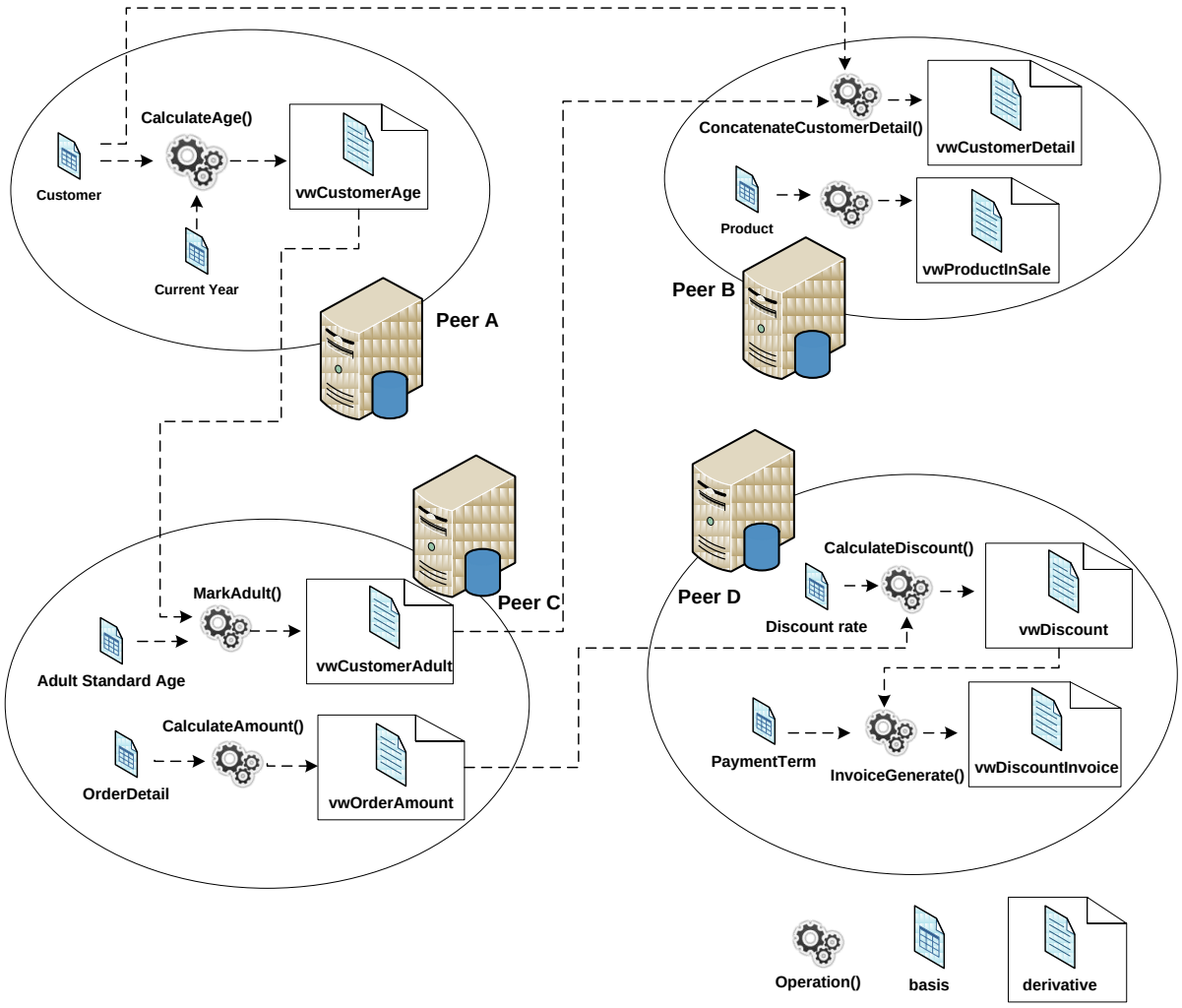


Figure 14: Representation of distributed database environment with data dependencies

4.4 Integration with Simulator PeerFactSim.KOM

The D^{4}M framework is integrated into PeerfactSim.KOM simulator in order to evaluate the framework with underlying DHT overlay protocol. The simulation results can be used with different overlay protocols to optimize the principles for profile-based optimization strategy.

4.4.1 Design Decisions and Considerations

The PeerfactSim.KOM simulator allows us to integrate the framework in two possible ways.

1. First, in an underlying DHT protocol each peer may have its storage space where the data artifacts can be stored. The data artifacts can be stored as key/value pair in each peer's storage media which is a storage component of DHT protocol. The storage media depends upon underlying protocol how it may store the data. In this case our framework cannot work independent of underlying protocol i.e. tightly coupled implementation needs to be done and may change for every other underlying DHT protocol.
2. The other possible way to integrate a D^4M framework in the PeerfactSim.KOM simulator is to build a 'cache' component on each peer which runs transparently on the underlying DHT protocol like Chord, Pastry or Kademlia. Each peer in the network is having a cache component to store its interested artifacts which includes D^4M data components i.e. basis and derivatives. It is a loosely-coupled implementation.

The first approach offers the storage services which are embedded in DHT protocol whereas the later approach provides flexibility for the implementation of different DHT protocols with D^4M framework in PeerfactSim.KOM. In general, each DHT protocol has different storage components and different internal structure which may disturb the implementation of storage services of D^4M framework in the first approach.

Therefore, in order to carry out the transparent implementation of D^4M framework in PeerfactSim.KOM, later approach is selected. In selected approach, the D^4M 'cache' component acts as an indepedenent component of a peer which provides the storage services as well as the services for data manipulation and data transmission tasks. It employs the overlay services of underlying DHT protocol for data lookups and transmission. In transparent implementation of D^4M framework in PeerfactSim.KOM each peer has layered architecture shown in figure 15. The lower layers like network layer, transport layer and overlay layer are provided by PeerfactSim.KOM simulator while D^4M Cache layer provides data handling services to manage the distributed data and its dependencies across the network.

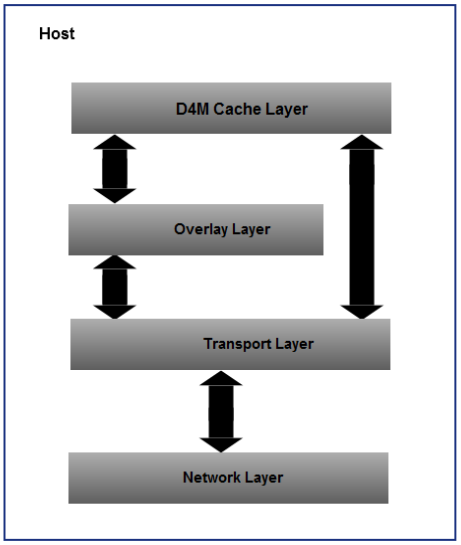


Figure 15: Peer layered architecture in PeerfactSim.KOM (after integration)

4.4.2 D^{4}M Cache Layer Components

Each peer in the network has a D^{4}M cache layer which holds the data handling components. First, the cache component is a storage container where it stores the data artifacts with their related information. These data artifacts can be either basis or derivatives, while the related information comprises of computational function and requirements/dependencies of the artifacts stored in data structures which are defined briefly later in this chapter. Beside the cache component, D^{4}M layer possess the handlers to manage the distributed data across the network, termed as *CacheHandler* and *PropagationHandler*. The cache handler is utilized to handle the insert/update/delete/query operations in cache whereas the propagation handler is used to propagate the artifact updates to their dependent derivatives in the network.

The operating modes for dependency management as defined in D^{4}M framework can be specified in cache component of a peer. The cache operating modes represent the degree of reactivity as how actively the computation is performed when the change occurs in any of the requirement artifact of derivative while propagation mode defines how to broadcast the artifact change to the peers having its dependent derivatives. The D^{4}M cache layer exposes its message listener to receive the data relevant messages and process it according to the D^{4}M framework message types. Due to the difference in propagation and storage functionality of the operating modes, there is a separate cache handler and propagation handler for each operating mode. For instance, the EagerCacheHandler and EagerPropagationHandler are used for Eager operating mode. Similarly, the other two operating modes have their own cache and propagation handler.

4.4.2.1 Cache Data Structures

The cache of a peer stores data artifacts with related reference artifacts which may be distributed over several peers or located on a same peer. These reference artifacts can be either data requirement artifacts or data dependency artifacts. In an implementation, these reference artifacts are implemented as data structures which implement the base class of "*RemoteArtifacts*" depicted in figure 16. The reference artifact is used to hold the responsible peer of a remote artifact.

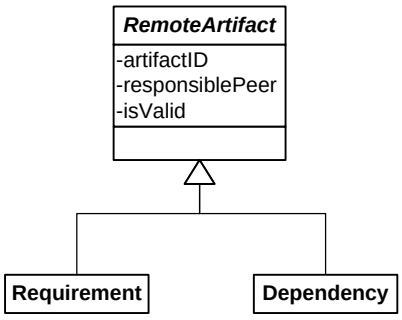


Figure 16: Implementation details of additional data structures

Requirements Artifacts

A list of requirement artifacts is attached with each derivative. These requirements artifacts are reference of basis or derivatives which are demanded for the computation function to compute only the derivatives. This data structure contains the responsible peer overlay id where a particular artifact is stored. The main purpose of a requirement artifact is to assist the computation of a derivative using an associated computation function.

Dependency Artifacts

The dependency data artifacts are associated with each basis and derivative as reference data structure. These data structures represent the derivatives which have to be updated if one of its Basis, computation function or required Derivatives has been changed. This reference data structure contains the peer overlay id on which dependent derivative is stored. These dependency artifacts are employed to propagate the artifact's changes to the peers holding its dependent derivatives.

4.4.3 Operation Services of D^{4}M Framework

In order to integrate the D^{4}M framework in discrete event based simulator PeerfactSim.KOM, certain operation services are implemented to handle the data dependencies and its change propagation across the network. This section discusses the details of various data handling operation services which are used in dependency data management.

4.4.3.1 Data Distribution Service – Bootstrap Process

The data distribution service performs the bootstrap process which is executed as soon as the network peers are established according to the given DHT protocol. This service is scheduled to run once in the start on each peer as configured in configuration xml file and allocates the data artifacts to the peers in the network. It reads the data config file '*d4M_data.xml'* and hashes the data artifacts into data keys to allocate them according to their hashed data key value to the corresponding peers. Its execution ensures the data specified in data file, has been delivered to the network. As soon as the data artifacts given in data config file are stored on peers in the network, the data stabilization process starts. Algorithm 1 shows the distribution of data.

```
Upon event (dataDistribution |INIT) do
        dataArtifacts := getData("d4M_data.xml");
        doHash(dataArtifacts);
        selectPeer := network.getRandomPeer();
        selectPeer.store(dataArtifacts);
        startTimer(t, STABILIZE)
end event
```

Algorithm 1: Data distribution operation

4.4.3.2 Data Stabilization Service

In stabilization process, each peer uses the neighborhood set table given by underlying DHT protocol to synch the data artifacts to its responsible peers. During this process, each peer stores its interested artifacts in its cache. When this service is triggered, it always looks into the cache of a peer for the data keys which are not in the range of the peer. If it founds any data key outside the range of its own peer id, it will push that data artifact to the closest peer according to the data key of an artifact; otherwise it will not do anything. It may take some time to move the artifacts to their proper responsible peers. The stabilization service is scheduled to run on each peer periodically to execute the process so that when a peer queries an artifact it can be retrieved from the network with a minimum response time.

Algorithm 2 details the stabilization process.

```
Upon event (timeout | STABILIZE) do
        for all artifact ε peerCache.getArtifacts() do
                closePeer := getClosestPeer(artifact.getDataKey() );
                if closePeer != currentPeer then
                        send(closePeer, artifact);
                end if
        end for
        startTimer(t, STABILIZE);
end event

upon event (receive | srcPeer, artifact) do
        if not peerCache.contains(artifact.getDataKey() ) then
                peerCache.insert(artifact);
        end if
        sendAck(srcPeer, ACK[artifact]);
end event

upon event (receive | srcPeer, ACK[artifact]) do
        peerCache.delete(artifact);
end event
```

Algorithm 2: Data Stabilization operation

4.4.3.3 LookUp Operation Service

The look up operation is used for finding the responsible peer for the required artifact. The operation is scheduled immediately when requirement artifact is needed to perform the derivative computation process or when the change in artifact is required to be propagated to the peers holding derivatives whose data value is dependent on this changed artifact. This lookup operation will only be triggered when a peer doesn't know the location of requirement artifact or dependent derivative where it is stored in the network. In simple words, if a peer doesn't know who is the responsible peer of requested requirement/dependent artifact the lookup operation will be used. In addition, it may also be called if the responsible peer is incorrect peer i.e. the requested requirement or dependent derivative is not found on responsible peer. Algorithm 3 explains the whole lookup operation.

A lookup operation uses the neighborhood table of requesting peer, given by the underlying protocol (Chord, pastry etc.). It compares the peer id of a neighboring peer with the requested data key and forwards the lookup request to the neighbor peer who is having peer Id closest to the data key. When data key is reached to the closest peer who is having the requested data artifact, the closest peer sends its peer overlay id in reply to the initiating peer of lookup request.

```
upon event (lookup | LOOKUP[dataKey])
        closePeer := getClosestPeerAmoungNeighbors(dataKey);
        send(closePeer, LOOKUP_FORWARD[currPeer, dataKey]);
end event
```

```
upon event (receive | srcPeer, LOOKUP_FORWARD[requestor, dataKey])
        if peerCache.contain(dataKey) then
                sendReply(requestor,  REPLY[dataKey]);
        else
                closePeer := getClosestPeerAmoungNeighbors (dataKey);
                send(closePeer, LOOKUP_FORWARD[requestor, dataKey]);
                sendAck(srcPeer, FORWARDED);
        end if
end event

upon event (receive | srcPeer, REPLY[dataKey])
        remoteArtifact:= remoteArtifactsList.get(dataKey);
        remoteArtifact.setResponsiblePeer(srcPeer);
        logMessage("LookUp completed successfully");
end event

upon event(receive | srcPeer,  FORWARDED)
        logMessage("LookUp Request is forwarded");
end event
```

Algorithm 3: Data Artifact Lookup Operation

4.4.3.4 Query Operation Service

A query operation is performed during a derivative evaluation process when the evaluating peer knows the responsible peer of the requirement artifact. The main purpose of this operation is to collect the values of the requirement artifacts which are distributed across the network as shown in Algorithm 4. In this operation, a peer queries the needed data key of the requirement artifact to the remote responsible peer. The query message is represented as "*QueryMessage*" in the implementation. The responsible peer receives the query message and searches the requested data key in its local cache. If it finds the data key in its local cache, it sends the value of requested data artifact in the reply message. This reply message is represented as "*ReplyQueryMessage*" in the implementation. But if the responsible peer is unable to find it in local cache, the requesting peer starts the look up operation for the data key.

```
upon event (query | QUERY[dataKey])
        if requirementArtifact.responsiblePeer != null then
                send( requirementArtifact.responsiblePeer, QUERY[dataKey]);
        else
                triggerEvent( lookUp, LOOKUP[dataKey] );
        end if
end event
```

```
upon event( receive | srcPeer, QUERY[dataKey])
        if peerCache.contains(dataKey) then
                sendReply( srcPeer, RESPONSE[artifactValue]);
        else
                sendReply(srcPeer, RESPONSE[null]);
        end if
end event

upon event( receive | srcPeer, , RESPONSE[artifactValue])
        if artifactValue is null then
                triggerEvent( lookUp, LOOKUP[artifact.dataKey] );
        else
                logMessage("query artifact recieved");
                . . . . .(any post operation)
        end if
end event
```

Algorithm 4: Query operation

4.4.3.5 Evaluation Operation Service

In an integration of D^4M framework, an evaluation operation is employed to compute the derivative using computation function associated with the derivative. The evaluation process of a derivative is based on the cache operating mode. In eager mode and lazy mode, the peer computes the derivative and stores the updated value in its cache whereas in quiet mode, the peer computes the derivative every time it is requested but doesn't store the computed value in its cache. As soon as the evaluation process completes, the updated derivative is propagated to the peers holding its dependent derivatives. The later process is termed as propagation process which is based on its propagation operating mode.

Given in Algorithm 5, in an evaluation process first all the requirement artifacts which are needed to compute the required derivative will be collected from the local cache of a peer. If these requirement artifacts are not stored in its local cache then they are pulled out from the peers in the network using pull operation services. During this collection/pull out operation, if the peer doesn't know the remote location/responsible peer of its requirement artifacts, the lookup operation will be used to search the needed requirement artifacts.

```
upon event(evaluate, EVALUATE[derivative])
        for all requirementArtifact ϵ derivative.getRequirementsArifacts() do
                if peerCache.contains(remoteArtifact) then
                    requirementArtifactValue := requirementArtifactValue ∪ { remoteArtifact .Value };
                else
                    triggerEvent( query, QUERY[remoteArtifact.getDataKey ]);
                end if
        end for
        if requirementArtifactValue.size = derivative.getRequirementArtifacts.size then
```

```
            triggerEvent( compute, COMPUTE[ derivative, requirementArtifactValue] );
        end if
end event

upon event( receive | srcPeer, , RESPONSE[artifactValue])
        if artifactValue is null then
            triggerEvent( lookUp, LOOKUP[artifact.dataKey] );
        else
            logMessage("query artifact recieved");
            requirementArtifactValue := requirementArtifactValue ∪ { artifactValue };
            if requirementArtifactValue.size = derivative.getRequirementArtifacts.size then
                triggerEvent( compute, COMPUTE[ derivative, requirementArtifactValue] );
        end if
        end if
end event

upon event (compute | COMPUTE [derivative, requirementArtifactValue])
        derivative.value = derivative.getComputeFunction(requirementArtifactValue);
        if derivative.value is changed then
            triggerEvent(propagate, PROPAGATE[derivative]);
        end if
end event
```

Algorithm 5: Derivative Evaluation Operation

4.4.3.6 Propagation Operation service

The propagation service propagates the updated artifact values to the peers having dependent derivatives of the updated artifacts, in the network. The propagation of updated artifacts is solely depends on the propagation operating mode of a peer who is updating the data value of the artifact. For this purpose, there is the handler for each propagation mode who manages the overall process of propagation according to the mode specification.

In eager propagation mode, the updated data value of the artifact is immediately broadcast to its peers having dependent derivatives as specified in D^4M framework. In an implementation, "*PropagationHandlerEager*" is used to handle this process. In lazy propagation mode, the updated data value of artifact is not broadcasted to peers having its dependent derivatives rather it only invalidates the dependent derivatives either stored in its local cache or on remote peers. Therefore it sends the invalidation messages to the remotely stored dependent derivatives. In an implementation, the handler named as "*PropagationHanderLazy*" is operating for this mode. Similarly, there is a *PropagationHandlerQuiet* which handles the data propagation process under quiet mode. Algorithm 6 briefs the propagation operation.

```
upon event (propagate | PROPAGATE[derivative])
        for all dependentDerivative ϵ derivative.getDependencies()  do
            if dependentDerivative.getResponsibilePeer() is not null then
```

```
            send(dependentDerivative.getResponsibilePeer(), UPDATE[derivative.getDataKey,
                    derivative.Value] Or INVALIDATE[dependentDervative] );
        else
            triggerEvent( lookup, LOOKUP[dependentDerivaitve.getDataKey]);
        end if
    end for
end event

upon event( receive | srcPeer, UPDATE[updatedDerivative.getDataKey, updatedDerivative.Value]
                                        Or INVALIDATE [dependentDervative])
    if UPDATE then
        for all derivative ϵ peerCache.getStoredDerivatives do
            if derivative.getRequirements().contain(updatedDerivative.getDataKey) then
                triggerEvent(evaluate, EVALUATE[derivative] );
            end if
        end for
    else if INVALIDATE then
        triggerEvent( invalidate, INVALIDATE[dependentDerivative]);
    end if
end event

upon event ( invalidate | INVALIDATE[dependentDerivative])
        if peerCache.contains(depedentDerivative.getDataKey) then
            derivative:= peerCache.getStoredDerivatives(depedentDerivative.getDataKey);
            derivative.isValid = false;
        end if
end event
```

Algorithm 6: Propagation Operation

5 Thesis Approach

A structured P2P system provides a scalable solution to share data among peers in the network, but at the same time offering high data availability is a challenge for these systems. This chapter addresses the problem of how to provide high data availability for D^4M - the dependency management system operating in a structured P2P environment, by introducing the redundancy of data, i.e. by exploiting data replication.

5.1 Problem Statement

The dynamic behavior of the peers in P2P systems allows them to join or leave the network at any time. The unpredictable departure of a peer in decentralized P2P environments is a very common behavior, termed churn. When a peer departs, it cannot be accessible by any peer in the network and its data becomes unavailable to the other peers. In particular, for a dependency management system where derived data artifacts are distributed across peers in the network, the departure of a peer may cause not only unavailability of its data but also introduces the inconsistencies to the remote data artifacts which are dependent on its data. The inconsistencies in the remote data artifacts may also affect the reliability of data. Let us illustrate the problem with a simple running scenario mentioned earlier in Figure 15.

Consider the following scenario. If peerA which stores the customer table (a basis artifact) and *vwCustomerAge* (a derivative artifact) crashes, then peerB will be unable to compute its locally stored and derivative *vwCustomerDetail* due to the absence of a customer table. Similarly, peerC will be unable to compute its locally stored and derivative *vwCustomerAdult* without the derivative *vwCustomerAge*. In the worst case, if the data values of customer table and *vwCustomerAge* are ignored when computing *vwCustomerDetail* and *vwCustomerAdult* on peerB and peerC, then the data values computed will be incorrect and may not be considered as reliably computed data. In this way, churn causes two challenges which a P2P data dependency management system must cope with: how to provide high availability and ensuring reliable computation. In order to handle both these challenges the high availability of data artifacts must be ensured. Therefore, in order to ensure high data availability and reliability in a P2P data dependency management system, the churn behavior has to be handled in an effective way.

5.2 Solution

In order to handle the churn behavior without any data loss, P2P systems typically rely on data replication strategies. Each peer in the network needs to replicate a copy of its data (as a backup) on a specified number of other peers in the network and synchronize all of these replicas when a change in this data is detected. In this way if any of the peers crashes or leaves the network, then its updated data should still be available to other peers via one or more of the replicas. Using replication the system can continue to produce reliable data as it has avoided the problem of data being unavailable. To achieve this, a churn handling model is proposed which provides an effective replication and recovery protocol with the capability of handling complex dependency relationships. The proposed replication protocol ensures data availability and reliability despite churn, particularly focusing on the case of D^4M – a P2P data dependency management system. Before going into the details of the proposed model, a common approach to implementing a replication protocol is discussed. This approach was considered during our design process.

5.2.1 Common Approach to Replication

A very common approach to replication designed for structured P2P systems, is to use the closest peers to replicate the peer data. In this way when churn occurs, the data of an unavailable peer can be recovered efficiently without any complex processing. The complex processing were refer to here is the recovery procedure in which another peer in the network takes the responsibility for the data which was stored on the unavailable peer and starts processing request for this (former) peer's data. The process is efficient due to the short distance between the newly selected responsible peer and the unavailable peer. The closer the newly responsible peer to the unavailable peer, the less resources and time it will take to synchronize the data to the responsible peer when using the standard DHT data allocation process described earlier in section 3.1.2.

In P2P data dependency management system (D^4M) as shown in Figure 17, each peer maintains a data cache to store the data for which it is responsible. This data comprises a collection of data artifacts, i.e. basis and derivative data. In the discussion, the peer who replicates its data to some other peer is referred to as a *root peer* while the one who stores a replica is a *replica holder*. In the traditional approach, data artifacts are replicated from its responsible peer to peers whose peer identifier is close to the data key. Therefore each peer in the network utilizes its immediate successors and predecessors as its replica holders, depending on the k number of replica holders that are desired. The maximum number of replica holders occurs when $k = 1 \ldots N - 1$, where N is the total number of peers in the network. In the case of maximum replication every node in the network has a copy of all of the data. At the end of the replication process, each peer in the network may act as both a root peer and a replica holder at the same time (as following replication each peer has a copy of the data).

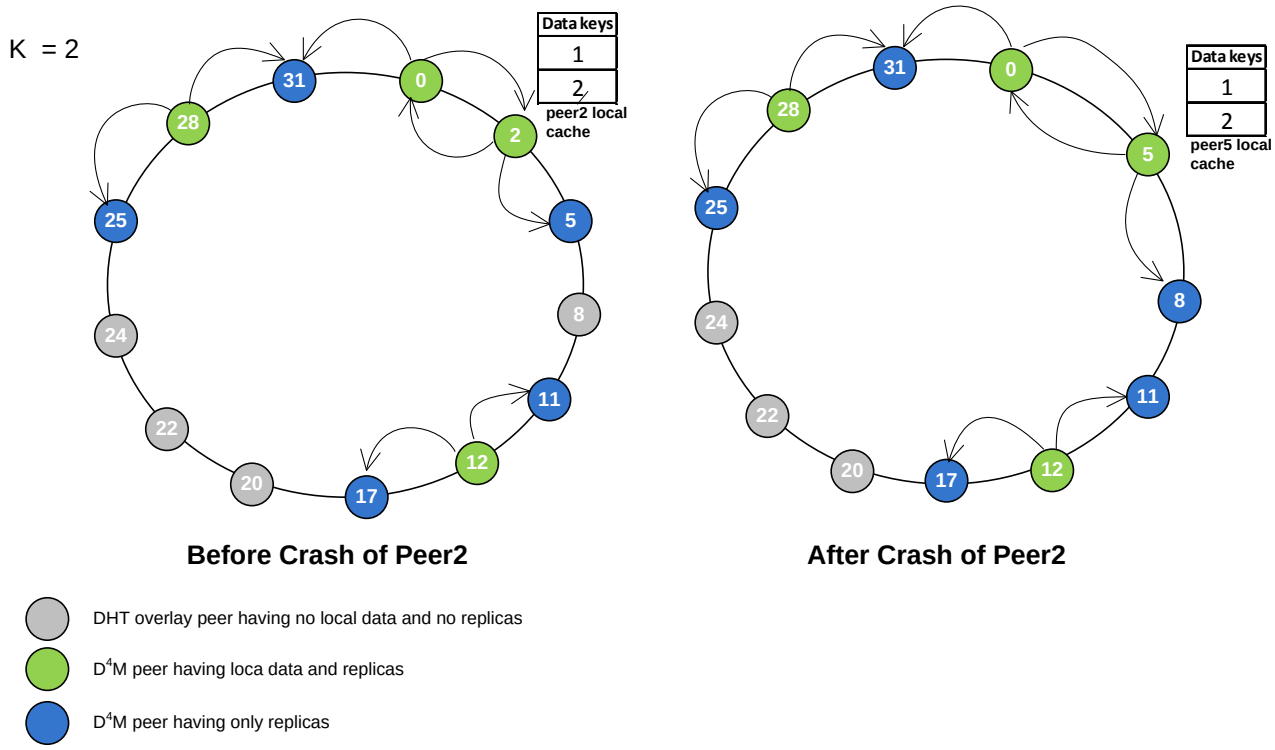


Figure 17: First Approach showing replication and recovery protocol

Given in Figure 17, now if any of the peers goes offline (i.e., become unavailable), then its data artifacts will still be available at one of its replica holders. As soon as its replica holders detect the

unavailability of a root peer, they will co-ordinate with each other to select one replica holder as the newly responsible peer for the data artifacts of the unavailable peer. The selection of a responsible peer among replica holders may utilize their knowledge of their distance to the unavailable peer. Now the selected peer will try to synchronize the data artifacts present in a replica of the unavailable peer according to their data keys, as was done during the data allocation process in a structured P2P systems. Furthermore, the unavailable peer was also holding a replica of some other root peer(s), thus as soon as each root peer detects that this peer is unavailable, it will choose another neighbor peer as its replica holder.

In the DHT overlay as depicted in the query routing table in Figure 17, the data keys are distributed across the network. The overlay network is setup for k=2 replicas represented by arrows. There are three different kinds of peers: the green and blue peers are the part of D^4M data dependency management system and store data artifacts in their local cache or store replicas of other peers, whereas the gray peers are part of the overlay network but do not store data artifacts or any replica in their cache, thus they are not the part of D^4M dependency management system. If peer2 crashes, who should store the data artifacts in its local cache as well the replica of peer0? As soon as peer0 detects the crash of peer2, it will select peer5 as its replica holder which is now its successor provided by underlying DHT protocol. Furthermore, the replica holders of the peer2 i.e. peer0 and peer5 will also detect the crash of their root peer and peer0 will take the responsibility of local data of peer2 due to its shortest distance from peer2. As soon as the data artifacts of peer2 will be re-stored in the local cache of peer0, the synchronization process will be started which sends the data artifacts i.e. key2, key1 to its responsible peer i.e. peer5. The peer5 who was storing only the replica earlier, now stores the data artifacts having key1, key2 in its local cache. In a result, the peer8 now stores the replica of peer5, thus it became the part of data dependency management system as compare to earlier state.

The redundant data in the network always minimizes the lookup time of data requests but at the same time it introduces the concurrency problem. In this case, to get rid of concurrency issue in p2p dependency management system, only the responsible peer (root peer) can serve the data look up requests and update its data artifacts. It means that a replica holder has a read-only copy of root peer; it cannot update or respond the data requests related to data artifacts stored as replicas on it. Normally the updates are prompted if any of the local or remote dependent data artifacts are changed. When the root peer receives the updates, it computes the relevant data artifacts and sends the updated data artifacts to its replica holders in order to synchronize them with updated data. In this solution, the main focus of replica maintenance is to ensure the data availability and reliability, rather to make efficient data lookups.

5.2.2 Proposed Approach

Recent measurements on P2P systems shows that peers characteristics are highly variable in peer to peer systems. The peers may differ in terms of bandwidth, CPU load, storage space and stability etc. But in existing structured p2p systems, such as Chord, Pastry, and Kademlia, it is a common assumption that all peers in the network are having similar distribution of resources and capacities. Therefore, the peers having low capacities are forced to handle the same computation load which the high capacity peers are handling. In a result, it affects the overall system performance badly. The earlier approach mentions the same assumption of peer's equality which needs to be handled in an effective way. Therefore, in order to handle this situation, the concept of utility function is presented. A utility function is introduced to dispatch the assumption of a peer's equality as well as to operate efficiently in heterogeneous environments where the peers have different configurations.

In this data replication approach, the selection of a replica holder peer is based on specified utility function. Referring from the previous section, the root peers are those peers who are replicating their local cache data and replica holders are the ones which keep the replica copy of root peer's cache. The

utility function measures the properties of a peer, depending on the application domain. For instance, in database management systems the utility function of a peer can be defined by available bandwidth and storage space. In contrast to that, in multimedia systems the peer's available latency and bandwidth may be defined as utility function. Each peer in the network measures its capacity using utility metric and selects its replica holders based on the utility function of its neighbor peers, which comprises of its successor and predecessor peers. Unlike the first approach, the utility function provides a load balanced network where peers are utilized according to their utilities and capacities. The peer with higher utility handles more computation/bandwidth load than the peer having low utility. This justified way of load handling results in an overall better performance across the network and allows the low capacity devices to become a part of replication process in the p2p network. The mobile phones and desktop PCs are the examples of low capacity devices. For instance, the mobile phones can store the replica in its memory and mobile users can play with the replicated data in offline mode, then revert back the changes to the original data source when gets online as discussed in [61].

In particular, D^4M is dealing with the distributed data in peer to peer environment, so the storage space as utility function may be considered as a critical decision. The storage space as utility function helps the peer to utilize its high utility neighbor peers for holding its cache replica. The proposed utility based algorithm is summarized concisely in a sequence of steps below.

1. Each peer in the network queries the available cache space (utility) of all its immediate neighbors.
2. As soon as a peer gets the utilities of all its neighbors, it selects the neighbor peer with highest available cache space (utility) as its replica holder.
3. This process repeats on each peer until its specified number of replica holders K is not accomplished or any of its replica holders is departed or crashed.

Note that the selected neighbor should have larger available cache space than the root peer's local cache space, in order to store the cache replica. It may arise the concurrency issues among the root peers and its replica holders like data artifacts may not be consistent among them and some replica holders have updated copy which is not available to other replica holders yet. Since we are dealing with D^4M, the requirement artifacts and dependency artifacts make this problem solved. These reference data structures (see section 4.4.2.1) contain the responsible peer's address, which represents the peer who has its original copy. Therefore, the query request of any of the data artifacts can only be served by its responsible peer who is having its original copy, not the replica copy. If the responsible peer is crashed then the look up operation will be performed for the requested data artifact across the network. There are more few complex cases in this scenario which are discussed in section 5.3.2.1

The further details of the approach are discussed with its architecture and internal protocol in the coming section.

5.3 Architecture

After discussing the proposed approach, this section gives a detailed overview of the architecture of the proposed solution, which addresses the replication and recovery protocol to handle the churn occurrence in D^4M for structured p2p environments. In order to manage the replication phenomena, the architecture of a peer provided by the p2p simulator PeerfactSim.KOM, given in earlier Figure 15 is altered by adding the functional components for replication and recovery protocol which are represented in Figure 18.

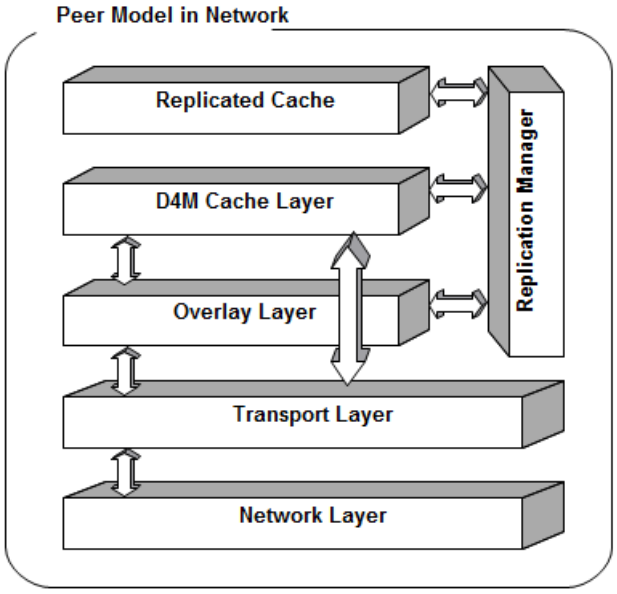


Figure 18: Representation of peer during replication protocol

In the given architecture, the replicated cache and replication manager are the additional components which deal with the replication process. When a peer joins the network, its storage space may be utilized as regular cache and replicated cache, to store its local data and data replicas of other peers. The regular cache comprises of peer's own data artifacts while the cache replica is served to store the replicas of the other peers as shown in Figure 19.

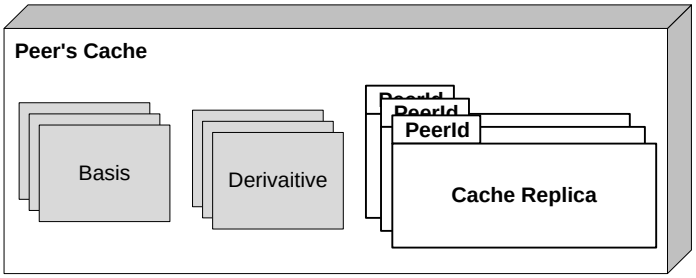


Figure 19: Internal Structure of peer's Cache

The data replicas are termed as cache replicas which contain the data artifacts, i.e., basis and derivatives of the root peer and associated key identifier to recognize which peer the replica belongs to. The internal structure of a cache replica is modeled in Figure 20.

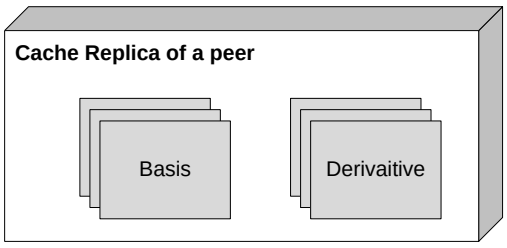


Figure 20: Internal Structure of Peer's Cache Replica

These cache replicas are used to assure the data availability in churn case. The replication manager of a peer is responsible to handle the replication operations i.e. to retrieve and store the cache replica of other peers into its replicated cache. At a time only one replica instance of a same peer can be stored in the replicated cache of the replicating peer. In other words, no peer in the network can store the replica of any peer more than once at the same time.

D^{4}M is implemented on underlying DHT overlay to utilize the services of DHT. In Figure 21, the D^{4}M data topology is built on a structured decentralized network. A chord protocol is selected to build a structured decentralized network for the demonstration of the recovery protocol. The selection of the chord protocol is done solely due to its simple structure and functionality.

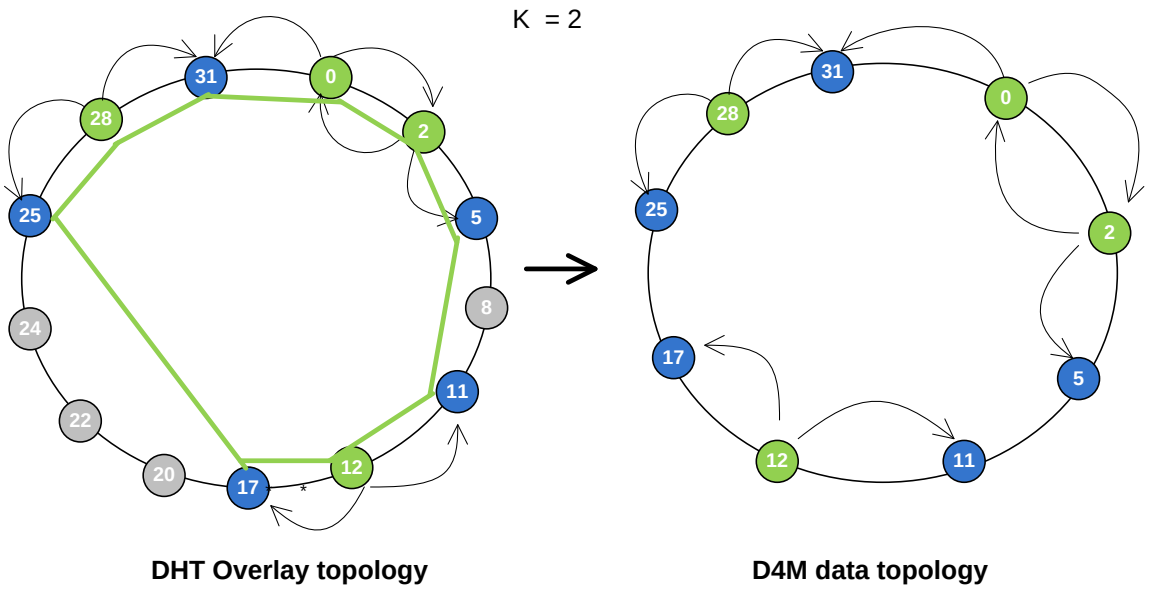


Figure 21: D^{4}M data topology in DHT overlay network

In Figure 21, the gray peers are considered as the part of decentralized network but do not belong to the D^{4}M data topology. The blue nodes are the peers who contain only cache replica of other peers in the network but don't store the data artifacts in their local cache. The green peers act as regular peer as well as replica holder of some other peer in the network. They store the cache replica of other peers as well as the data artifacts in their regular local cache. In a given figure 20, D^{4}M data topology is built on DHT overlay where K is the number of replicas each peer maintains. The D^{4}M data topology shows only the peers which store the D^{4}M data artifacts either as replicas or data artifacts in their local cache, termed as D^{4}M peers. In a case of churn, the DHT peers may be utilized to store the data of crashed peers as they have more available storage space than D^{4}M peers. Following the replication protocol, each peer maintains a number of replica holder peers to store their cache across the network for redundancy. Each replica holder peer knows about the other replica holders of its root peer.

5.3.1 Replication Protocol

The proposed data replication protocol comprises of different operations which includes the selection of replica holders, the data synchronization process in which root peers replicates its local cache items to its selected replica holders and the maintenance operation which ensures the synchronization of updated data from the local cache to the replica holders. Before these operations can be discussed further, the basic requirements are addressed to setup the environment for a D^4M replication protocol in p2p systems.

5.3.1.1 Basic Requirements for Replication Protocol

The data replication protocol needs the basic management services which facilitate the root peers to maintain and replicate the copies of its local cache to the number of replica holders in the network. These management services are executed on each peer periodically to perform replication management tasks which are mentioned below.

Keep-Alive Service

The keep-alive service runs periodically on each peer in the network i.e. replica holder peer as well as root peer. The main purpose of this service is to ensure that the peers are online and can participate in the certain network activities. It detects the neighbor peers when they crash. In this service, a peer sends the heart-beat message to all of its neighbor peers and expects the reply heart-beat message from each neighbor in reply. When it doesn't get a heart-beat reply message from a peer in specified time it detects the one as crashed peer. In order to manage the replication process properly this service should detect the peer when it goes offline otherwise the peers will be unable to detect the status of their replica holder peers.

Synch-Data Service

The synch-data service helps the root peers to synchronize their local cache data to its replica holder peers. During the replication protocol, when the root peer chooses its replicating peers using the utility-based selection strategy, the transmission of data will be performed from the root peer to its replica holders. Notice that the root peer's local cache comprises of a collection of basis and derivatives. The synch-data service will be executed either periodically or on data update event. When the local cache data of the root peer is changed, it executes the synch-data service to synchronize the local cache data to its replica holders if they are online so that the replica holders receive the updates accordingly.

5.3.1.2 Selection Process of Replica Holders

The selection operation performs the vital steps to setup the replication protocol. It includes the selection of the replica holder peers for each root peer. As mentioned earlier, the replica holder peers are selected based on utility function (storage space).

In an initial phase of the selection operation, a peer queries its neighbor peers for their available storage space (utility function). When it gets the responses from all the online neighbor peers, the peer which has the maximum amount of available storage space will be selected as a replica holder to store the cache replica of the root peer. In other words, each peer looks at its neighborhood set from the underlying DHT protocol and sends them queries asking for their available storage space. As soon as it gets a reply from all these neighbor peers, the node sorts the list of received storage space of all its neighbors and chooses the one which has the maximum amount of available storage space. Each peer may configure its K replica holder peers, where $K = 1 \ldots N\text{-}1$ and N is number of peers in the network.

For the selection of a specified number of replica holders, the root peer chooses the first *K* neighbor peers from the sorted list of available storage space replies. As soon as the replica holders are selected, the root peer initiates the synchronization process to replicate his local cache data to the selected replicating peer as a replica. The synchronization is performed using the synch-data service mentioned earlier. The whole selection process is described in using an activity diagram in Figure 22.

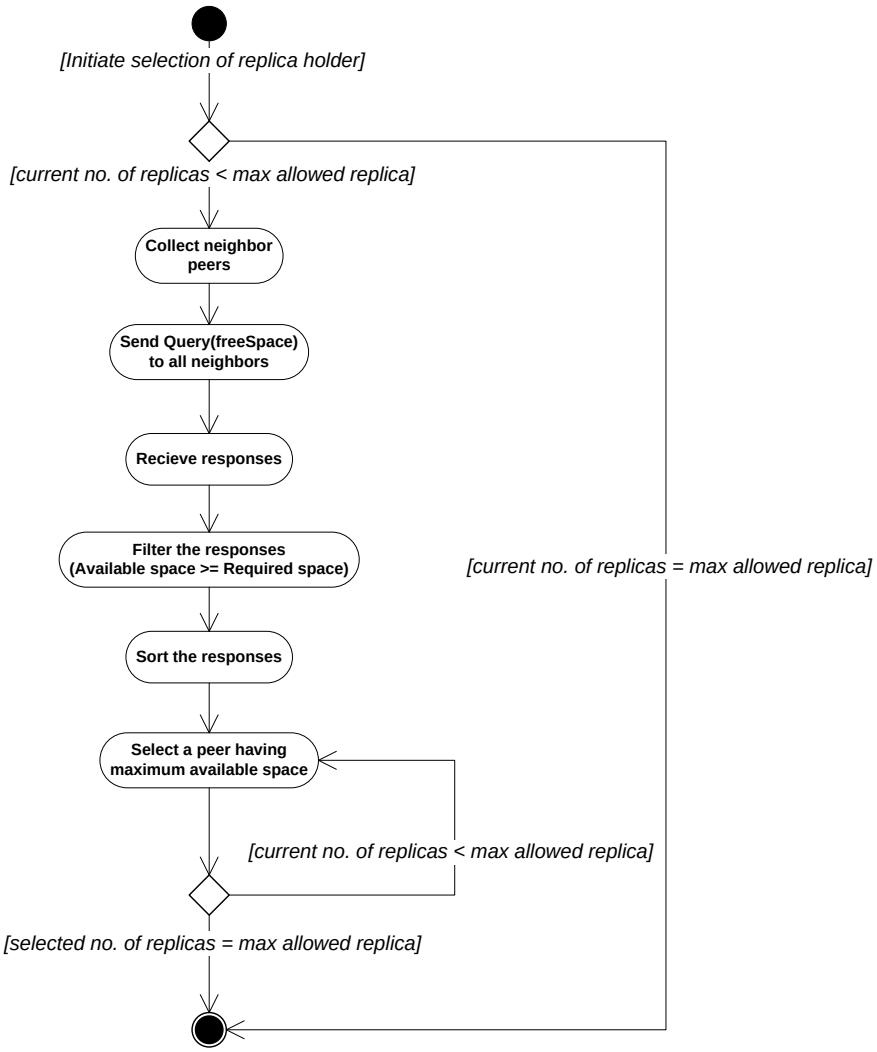


Figure 22: Activity Diagram of Replica Holder Selection (regular selection process)

During the selection process, it may be the case that none of the neighbor peers have enough available storage space to replicate the root peer's local cache. In this case the root peer randomly selects one of the neighbor peers from its neighborhood table and requests that it support the selection process of the root peer's replica holders. The randomly selected neighbor receives the request, executes the replica holder selection processes, and sends back its neighbor peer who is has the highest utility. The requesting peer receives the response which identifies the peer having the maximum available cache space and compares it with the needed cache space. If the selected peer has enough cache space to replicate the root peer's local cache, then the root peer will select it as its replica holder and initiate the synchronization process. But if its local cache requires more storage space than available, then this support requesting process will be executed for half the number of neighbors, i.e.

R = 1/2n where R is the number of randomly selected neighbors from whom to request help and n is the number of requesting peer's neighbors. Note that all the peers in the decentralized network may not be the part of the dependency management system's topology, as some peers might not store any artifact data in its local cache. This extended selection procedure of replication is illustrated in Figure 23 with the help of an activity diagram.

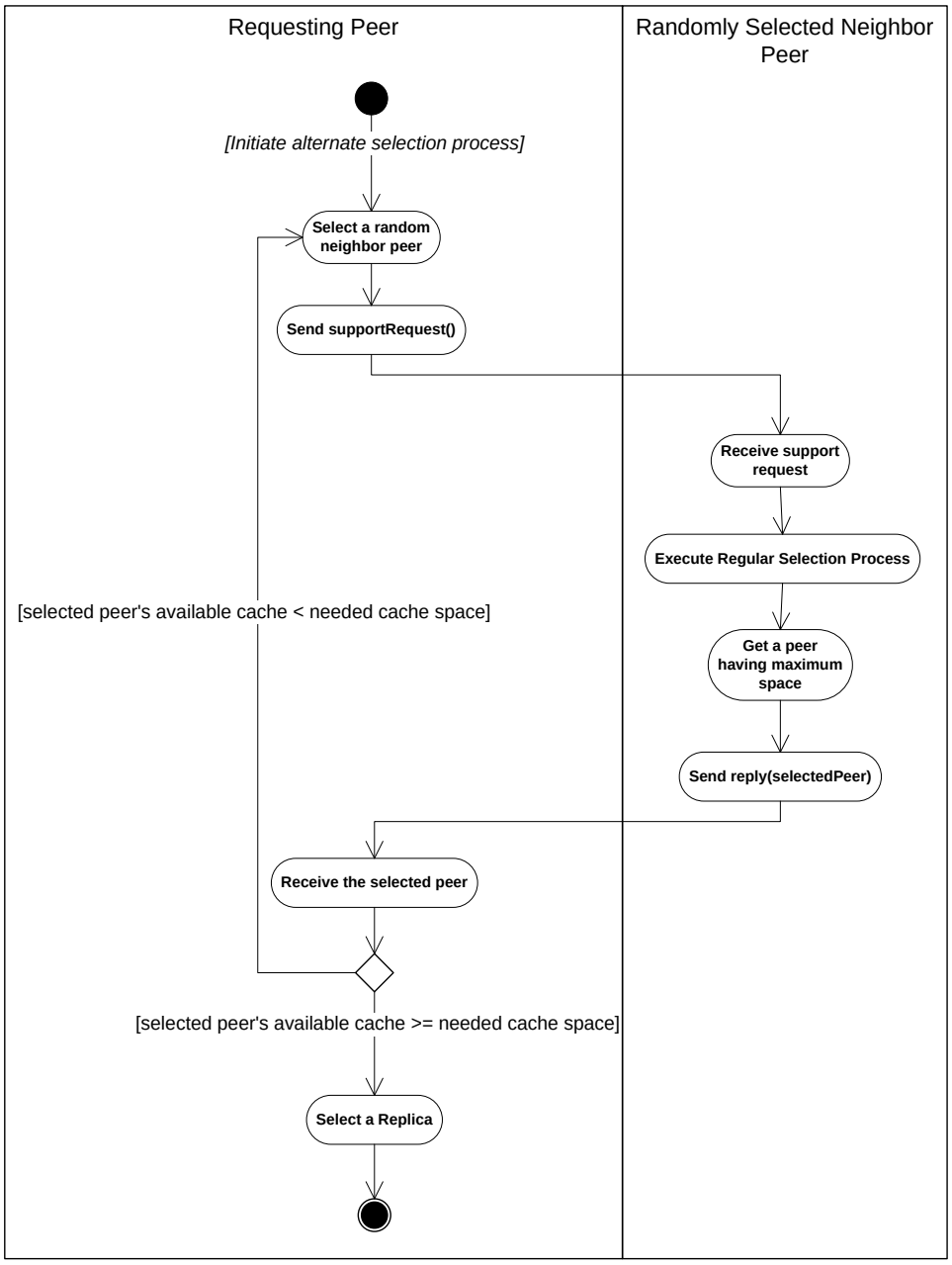


Figure 23: Activity Diagram of Extended Selection Process of Replica Holder

Implementation

This sub-section presents the implementation details of the proposed data replication protocol. To be able to implement the data replication protocol, the D^4M framework has been integrated into the PeerfactSim.KOM simulator in the first phase of this thesis project. The resulting system is used to study in detail various aspects of a data dependency management system with a variety of P2P protocols.

Given the architecture of the replica selection process mentioned above, for the implementation, each peer in the network chooses its replica holder peers from its neighborhood table as provided by underlying DHT protocol. The selected neighbor peers should have the maximum available storage space among all of its neighbors. Algorithm 7 reflects the workflow procedure for the selection of replica holders which was shown in the activity diagram in .

```
upon event (selectionPrototol , selectOnePeer ) do
    if selectedReplica.Count < k then
        for all neighbour ϵ currentPeer.getNeighbors() do
            send(neighbour, queryCacheSpace);
        end for
    end if
end event

upon event (receive | srcPeer, queryCacheSpace) do
    send(srcPeer, reply[peer.availableCacheSpace]);
end event

upon event (receive | srcPeer, reply[availableCacheSpace]) do
    neighborReply := neighborReply ⋃ { reply };
    if neighborReply.Count = peer.getNeighbors() then
        neighborReply.sort( availableCacheSpace );
        do while selectedReplica.Count =< k
            peer := highest( neighborReply[availableCacheSpace] ).srcPeer ;
            if peer.availableCacheSpace >= currentPeer.cacheSpaceNeeded then
                if selectOnePeer = true then
                    send( requestingPeer , peer );
                    break;
                end if
                selectedReplica := selectedReplica ⋃ { peer };
            else
                break;
            end if
        end do while
    end if
end event
```

Algorithm 7: Replica holder Selection

If none of the neighbor peers have enough cache space in order to store the root peer's cache replica, then the extended selection procedure is followed. The extended version of the selection protocol in algorithm 8 reflects the activity diagram in Figure 23 on page 39.

```
upon event (extendedSelectionProtocol) do
        randPeer := randomSelectOne( peer.getNeighbors());
        send(randPeer , helpToSelectReplicaRequest);
end event


upon event (receive |srcPeer,  helpToSelectReplicaRequest) do
        requestingPeer:= srcPeer;
        selectOnePeer := true;
        triggerEvent(selectionPrototol, selectOnePeer );
end event

upon event ( receive | srcPeer, selectedPeer ) do
        if selectedPeer.availableCacheSpace >= currentPeer. cacheSpaceNeeded then
                selectedReplica := selectedReplica ∪ { selectedPeer };
        else
                triggerEvent(extendedSelectionProtocol );
        end if
end event
```

Algorithm 8: Extended Algorithm for Replica holder Selection if neighbor peers do not need cache space

In order to fully understand the coordination among neighbor peers and root peer during selection process, a sequence diagram shown in Figure 24 describes the message transmission within the internal components of a peer and external communication with neighbor peers.

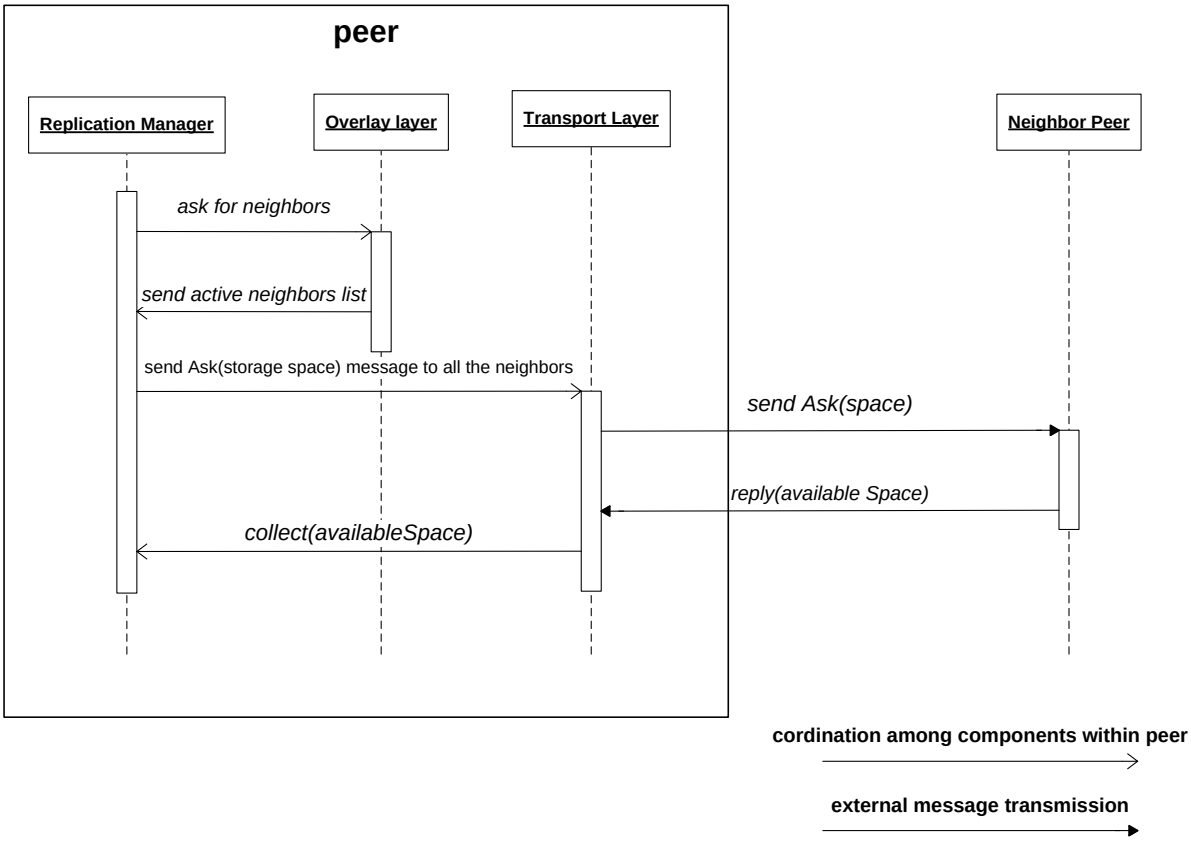


Figure 24: Coordination workflow during selection process of replica holders

5.3.1.3 Data Synchronization Process

After the selection of replica holders the synch-data service will be activated to replicate the local cache of the root peer to its replica holders as shown in the activity diagram in Figure 25 on next page. As soon as the synch-data service completes its data transmission operation, the root peer activates the keep-alive service to periodically to check the status of its replica holders. The same keep-alive service is executed on the replica holders to check the status of their root peer. In addition, a root peer informs each of its replica holders about all the other replica holders who are carrying its cache replica. In other words, each replica holder is aware of all the other replica holders of its root peer. This knowledge of the other replica holders of a root peer helps the replica holders to determine the responsible peer in the event of churn.

When the churn occurs, one of the replica holders will take responsibility for the cache items of the root peer. When a root peer calculates its data artifact, i.e. changes the basis value or computes derivatives due to a change in a dependent basis or derivative, then the synch-data service will transmit the updated data to the root peer's replica holders. The synch-data service is responsible for the maintenance of the cache replicas.

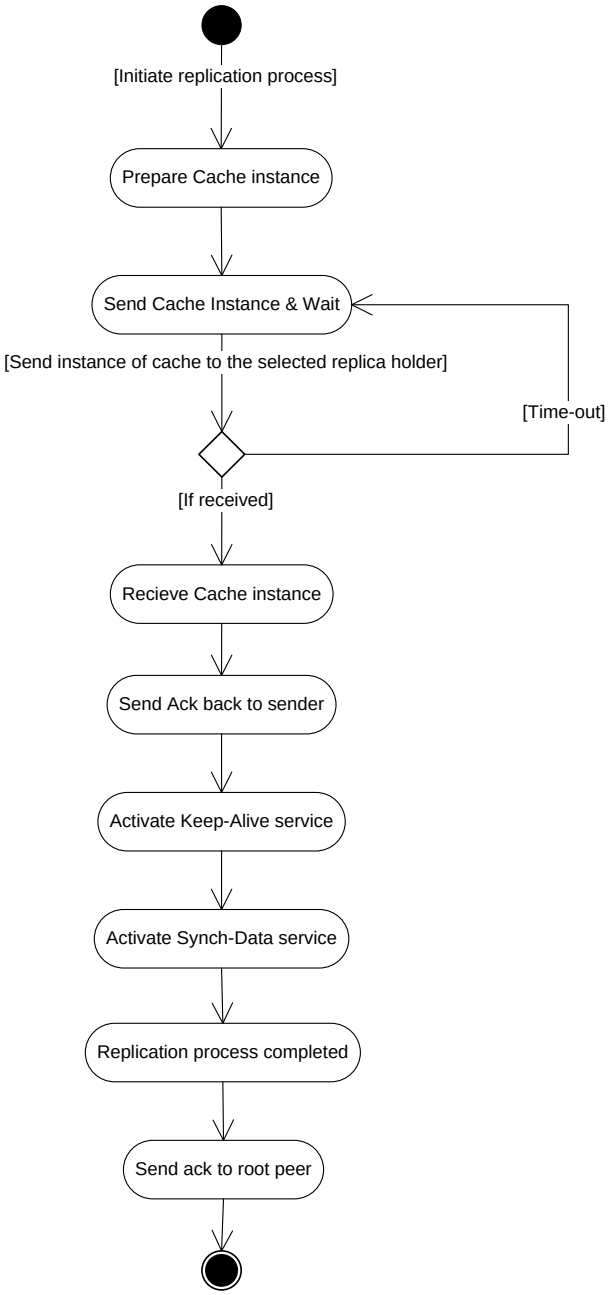


Figure 25: Data Synchronization Process

Implementation

The data synchronization process is implemented in PeerfactSim.KOM using message transmission services which send the root peer's local cache to its replica holders. Additionally, the keep-alive service is executed on the root peer to take care of replica holders and on each replica holder to be aware of the root peer's status. In order to ensure the delivery of messages, a timer is used in event management systems. If respective action is not taken within its allotted time, then the time out operation will be executed. The time out operation will try three times to repeat the unsuccessful operation, and then it will delete that operation from the event queue. In the algorithm for data synchronization, *startTimer(t, REPLICATE_REQUEST*) is the function which starts the timer when a replicate request is send to the replica holder to copy the local cache of a root peer. In this call t is the time span within which it must receive a replication completion message which confirms the root peer

that its local cache has been successfully replicated on the replica holder. If by t time, it does not receive a replication confirmation message, then it will try to send the instance of its local cache to the replica holder three times more. If none of these tries succeed in replicating the data onto a replica holder, then it will discard this operation from the queue. Algorithm 9 describes this data synchronization process.

```
upon event (dataSynch | INIT) do
        for all peer ϵ selectedReplicas do
                send( peer , replicate[ localCache ]);
                startTimer( t, REPLICATE_REQUEST[ peer ] );
        end for
end event
upon event( timeout | REPLICATE_REQUEST) do
        if numberOfTimes < 3
                send(REPLICATE_REQUEST.peer , replicate[ localCache ]);
                startTimer( t, REPLICATE_REQUEST[ peer ] );
                numberOfTimes := numberOfTimes + 1;
        end if
end event

upon event ( receive | srcPeer, replicate) do
        if replicate.localCache.spaceNeeded <= currentPeer.availableCacheSpace then
                trigger(replicate);
        else
                send( srcPeer, noSpaceAck);
        end if
end event

upon event(replicate) do
        for all data ϵ replicate.localCache[data] do
                replicatedCache:= replicatedCache ∪ {data};
        end for
        send( srcPeer, REPLICATE_COMPLETE);
end event

upon event(receive | srcPeer, noSpaceAck)
        trigger( selectionPrototol, false );
end event

upon event( receive | srcPeer, REPLICATE_COMPLETE) do
        replicaPeers := replicaPeers ∪ { srcPeer };
end event
```

```
upon event(timeout | HEARTBEAT) do
        for all replica ∈ replicaPeers do
                send( replica, HEARTBEAT );
        end for
        startKeepAliveServiceTimer( r, HEARTBEAT );
end event

upon event( receive | srcPeer, HEARBEAT) do
        alive := alive ∪ { srcPeer };
end event

upon event(dataSynch | CHANGE[ changedArtifacts] ) do
        for all replica ∈ replicaPeers do
                send(replica, changeArtifact);
                startTimer( c, CHANGE[ replica, changeArtifact] );
        end for
end event

upon event( receive | srcPeer, changeArtifact) do
        for all data ε replicate.localCache[data] do
                if data.id = changeArtifact.id
                        replicatedCache:= replicatedCache  - {data};
                        data.value := changeArtifact.value;
                        replicatedCache:= replicatedCache ∪ {data};
                end if
        end for
        send( srcPeer, Ack);
end event

upon event( timeout |CHANGE ) do
        if  try  < 3
                send(replica, changeArtifact);
                startTimer( c, CHANGE[ replica, changeArtifact] );
                try : = try +1;
        end if
end event
```

Algorithm 9: Data Synchronization process

5.3.2 Recovery Protocol

In order to assure data availability in D^{4}M, the recovery protocol utilizes the stage which was setup by the replication protocol. Following the previous section, the replication protocol allocates the replica holders to each peer who stores the cache replica of its regular cache items, i.e. data artifacts, in their replicated cache. When any of the peers in the network crashes, two cases need to be handled. First the root peers, who earlier selected the crashed peer as their replica holder, must detect its crash, so that they can select another peer in the network as a replica holder. When the root peer detects its replica holder has crashed, it will select another peer from its neighborhood table as its replica holder to maintain its specified number of replica holder peers. Secondly, the regular local cache of a crashed peer must be restored on its closest replica holder to assure data availability.

The process to restore the regular local cache of a crashed peer is a complex process. When the root peer fails, each of its replica holder peers detects the crash and tries to find the closest replicating peer to the root peer in the list. Each of them will select the closest replica holder to the root peer and send a request to this peer to take responsibility for local cache items of the crashed peer. As soon as the closest replica holder peer receives request messages from at least half of the replica holders of the crashed peer[1], it includes the cache replica items of crashed peer into its regular local cache. Now the synch-data service will synchronize the data artifacts according to the peer's key identifier. Figure 26 demonstrates this process.

In the structured decentralized network depicted in Figure 26, each peer has three neighbors in its neighborhood table. *Peer2* has as neighbors *Peer0*, *Peer5*, and *Peer12*; while the *Peer20* has neighbors *Peer22*, *Peer17*, and *Peer28*. Out of the three neighbors each peer has selected two peers as its replica holders. The peers in purple color are replica holder peers, the green nodes are not the part of D^{4}M data topology, the blue ones are simple neighbor peers which store only data artifacts in their regular local cache, and the gray nodes are peers who are both root peers and replica holder peers at the same time. *Peer22* is considered s root peer as it has data artifacts in its regular cache as well as a cache replica of *Peer20* in its replica cache.

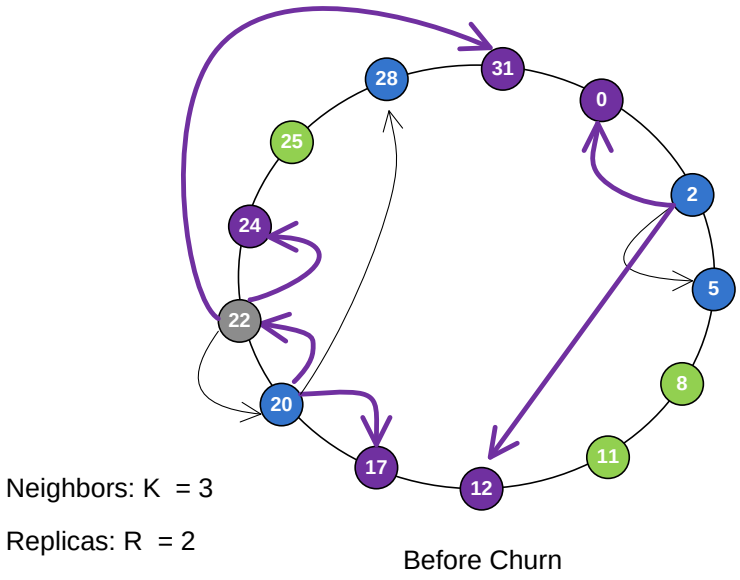


Figure 26: Demonstration of Recovery protocol in Chord topology

[1] We assume that there are an odd number of replica holder peers, otherwise we have to wait for half plus one request messages – to avoid the situation where two peers would each think that they have the responsibility for the crashed root peer.

If *Peer22* leaves or crashes, then its replica holders (i.e. *Peer24* and *Peer31)* will detect the crash of their root peer and will try to find the closest replica holder to the root peer. As a result, *Peer31* will find *Peer24* to be the closest replica holder peer of the crashed node; thus it will send a request message to *Peer24* to take responsibility for root peer's cache items. On the other hand, when *Peer24* detects the crash of *Peer22*, it will find itself as closest replica holder peer to the crash peer. So to confirm the crash, it waits for request messages from at least half of the replica holder peers of the crashed node. As soon as it receives the required number of request messages, it will include the cache replica items of *Peer22* in its regular local cache. In addition, it executes the data synchronization service to send the newly added cache items to its responsible peers. After the synch-data service completes, *Peer24* will send an acknowledgement message to the other replica holder peers of the crashed node (i.e. *Peer31* in above case), so that it can delete its cache replica of the crashed node. The crashed node, *Peer22* is also a replica holder which had stored the cache replica of *Peer20* in its replicated cache. When *Peer20* detects the crash of its replica holder peer, it will select another neighbor as its replica holder peer, such as *Peer24*. In this case *Peer24* will also store the cache replica of *Peer20* and therefore acts as a replica holder as well as a root peer. The state of this overlay network after churn recovery is shown in Figure 27 and the recovery process is described using an activity diagram in Figure 28.

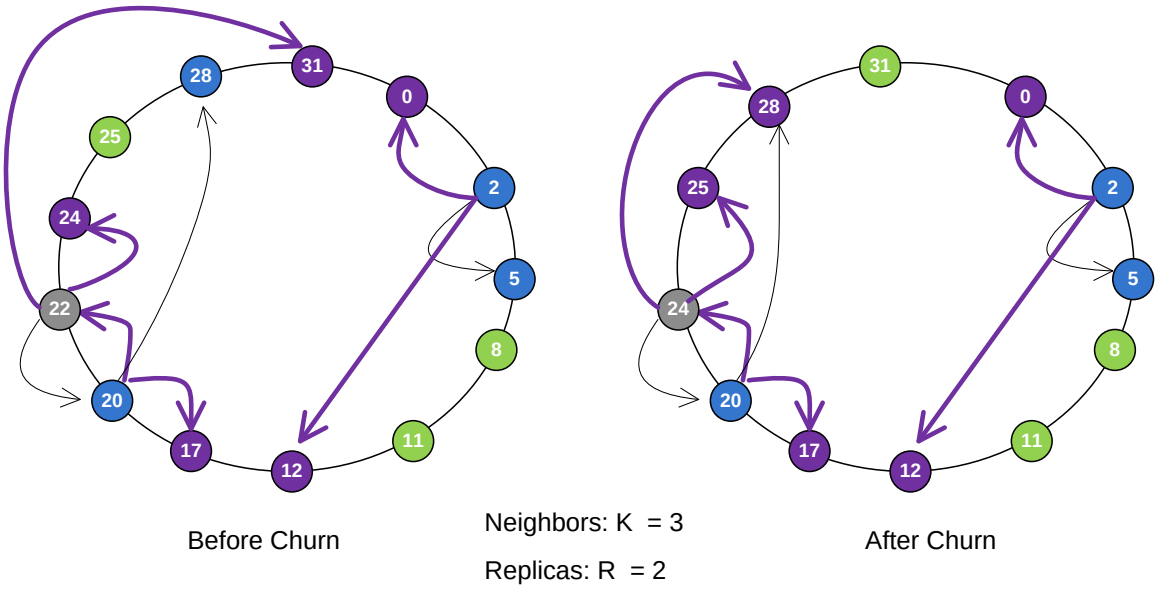


Figure 27: Recovery Protocol

If any of the data artifacts stored in a crashed peer's local cache is requested during the recovery period, the request will be stored in a queue at the peer who is supposed to store the requested data artifact. In other words, in structured P2P systems data blocks are stored on the peers based on their peer's unique key identifier, therefore based on this mechanism the request for a data artifact will be stored in a queue, maintained at the supposed-to-be-responsible peer. When the synch-data service is completely done with the synchronization of the data artifacts of the crashed peer to its responsible peer then this request will be retrieved from the queue and served, otherwise a timeout will occur and the request will be deleted from the queue at the supposed-to-be-responsible peer.

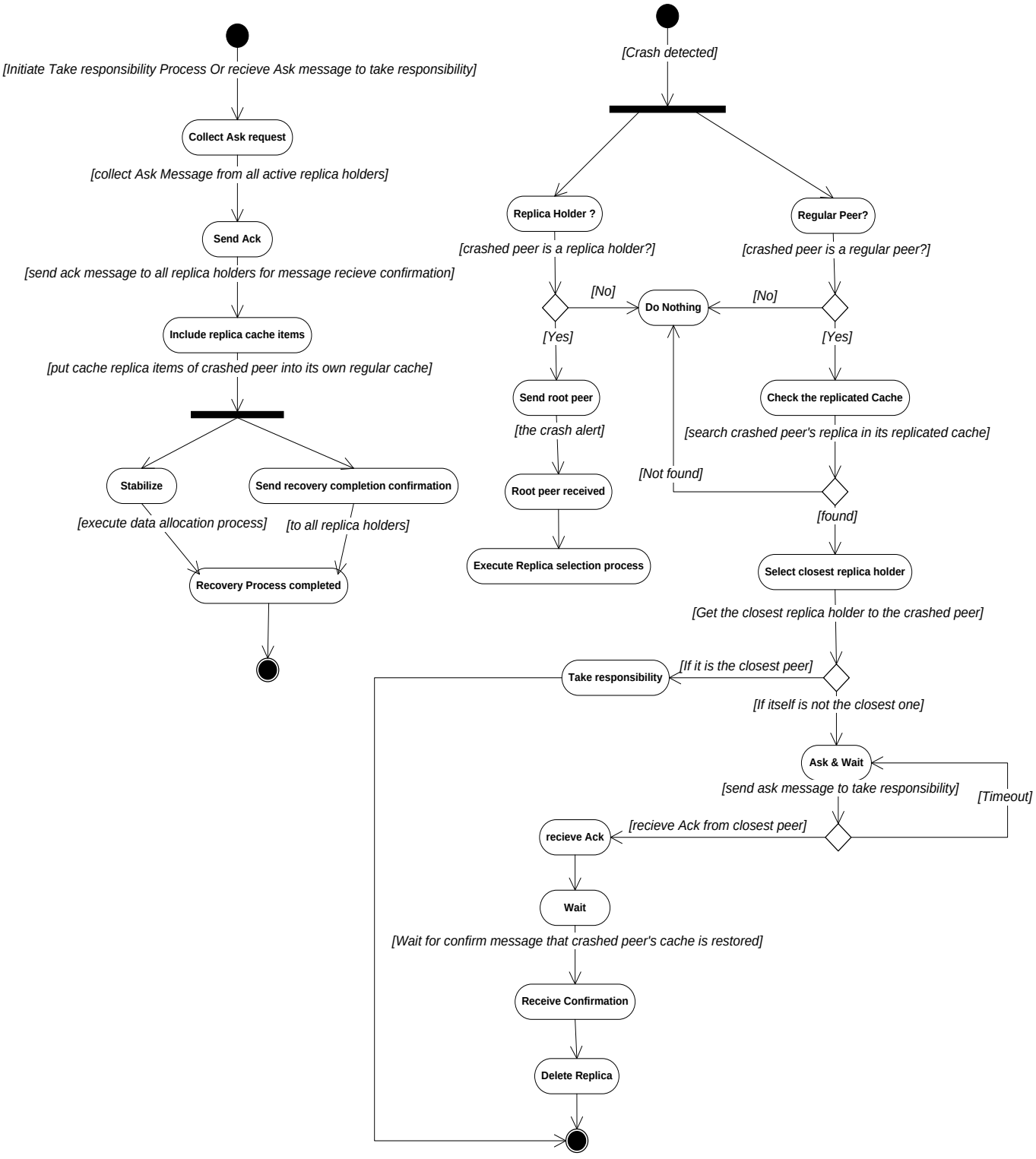


Figure 28: Recovery Process (left) and Churn Handling Process (right)

5.3.2.1 Handling of Complex Cases: Possible Risks for Concurrency

Some complex cases are identified which may lead to concurrency problems. As discussed earlier, while dealing with D^4M, there is no possible concurrency problem due to the presence of requirement/dependency artifacts which provides the responsible peer of data artifact containing its original copy. Apart from this solution, there are few cases which need to be handled.

Case I: If the root peer crashes who may take the responsibility of data which was stored on crashed peer i.e. data for which crashed peer was responsible?

According to the proposed recovery protocol, the closest replica to the crashed peer will take the responsibility. The complex scenario may occur in this situation like it might be possible that the root peer was having updated data artifacts which were just sent to its replica holders but half of the replica holders didn't get the updates due to network latency or any unexpected problem and before receiving them, the root peer has been crashed. In a result, now some replica holders may have updated data artifacts of crashed peer and some have older than the update one. In this case, the data structure *lastUpdatedTimeStampRootPeer* attached with the cache replica, will be utilized which shows the time of the root peer when it received the updates last time from the root peer. All the replica holders will exchange their last updated timestamp of the root peer that each of them received earlier and the replica holder who is having latest updated time of root peer (crashed), his replica copy will be restored to the newly responsible peer.

Case II: If the root peer crashes, who may serve the data requests during recovery time?

If any of the data artifacts which were stored on the crashed peer, are requested during recovery time, those requests will be served up by one of its replica holders. In this situation, when requesting peer doesn't find the data of responsible peer provided by requirement/dependency artifact, it will initiate the look up operation for the requested data artifact. In a result of lookup operation, any of the replica holders of crashed peer can be found and can serve the request for the requested data artifact.

6 Evaluation and Testing

In this section, the performance of the proposed replication protocol is evaluated through discrete event simulation using PeerfactSim.KOM simulator. The given evaluation shows that the proposed replication protocol provides a considerably fault tolerant system with little overhead to manage the dependent distributed data in p2p environment.

The rest of the section is organized as follows. The simulation setup is described along with simulation parameters. Then, the performance of the system is measured through analysis of communication cost in time and synchronization time of replicas during maintenance with variations in the number of peers. In the fairness evaluation section, the utilization of resources is analyzed using graphs to evaluate the fairness of system under the proposed replication protocol.

6.1.1 Environment Setup for Simulation

The simulation is based on an implementation of the Chord protocol which is a simple and efficient structured overlay protocol. In the simulation setup, a 300-peer network is considered with the parameters shown in Table 4.

Table 4: Simulation parameters

Simulation Parameters	Values
Number of overlay peers (N)	300
Number of D^4M peers (M)	160-240
Number of neighbours (G)	10
Number of data artifacts	100 (52 Basis + 48 Derivatives)
Number of replicas (K)	8
Number of data updates	30 updates/hr
D^4M Operating mode	Eager Mode
Crashed peers	10 peers/hr

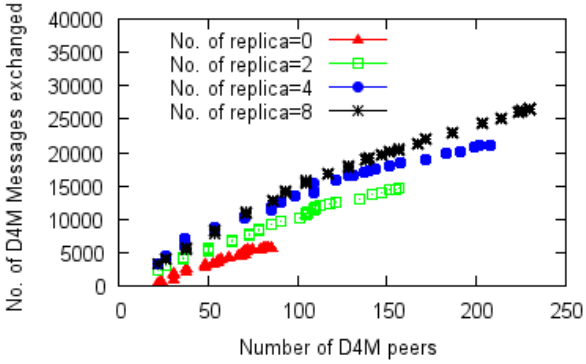
The number of the overlay peers can be distinguished from the number of D^4M peers as the overlay peers are the total number of peers in an overlay network, while D^4M peers are those overlay peers in the network which store the data artifacts either in their local cache or in replicated cache. The number of D^4M peers cannot be controlled by the user in these simulations because it varies with respect to the data keys assigned, thus the average of some simulations has been used for analysis.

The latency between any two peers in the simulated network is provided by the GNP Network Model given in PeerfactSim.KOM which is highly heterogeneous in nature. It is described in detail in PeerfactSim.KOM manual with the configuration setting. Similarly, the bandwidth between peers is also provided by GNP Network model in PeerfactSim.KOM.

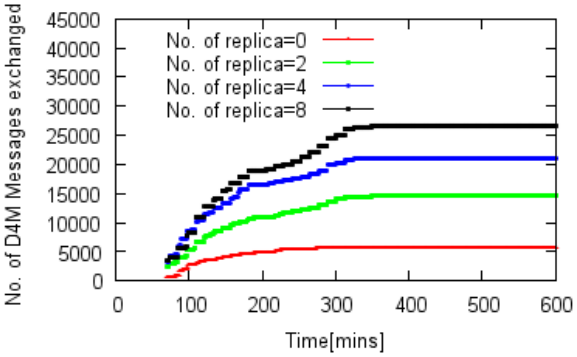
6.1.2 Performance Evaluation

In this section, the scalability of the system is evaluated through the study of communication costs as a function of the addition of peers. The communication cost is the total number of messages exchanged during the replica maintenance and data updates. In a given plot, the parameters are the same as given

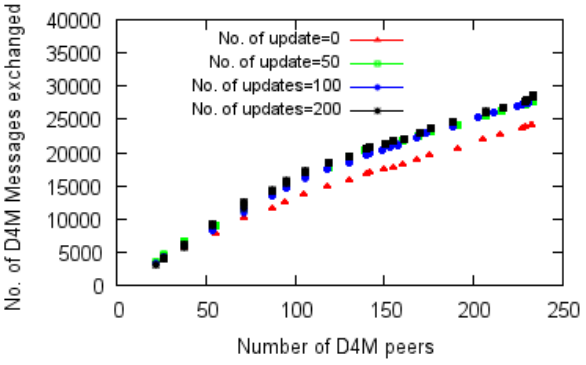
in Table 4 for all the plots *except* for the variation in number of replicas, number of data updates/hour and the number of artifacts. In each graph only one factor is varied in order to study the impact of this variation on the remaining factors.



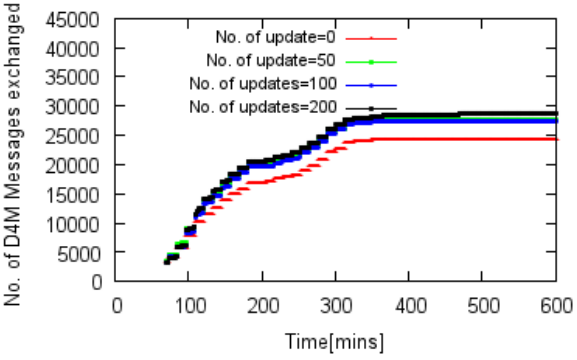
(a)



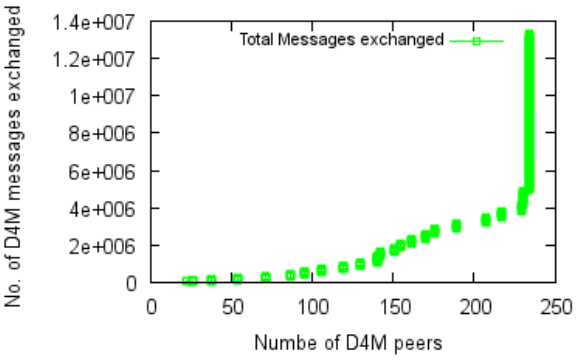
(b)



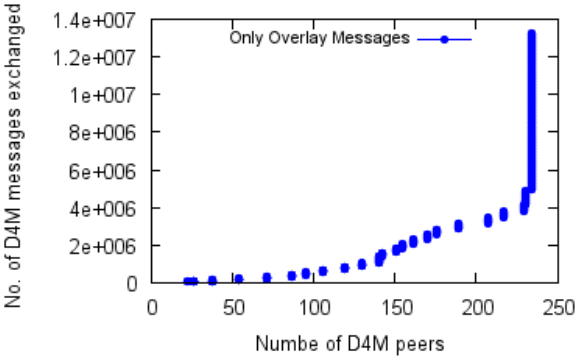
(c)



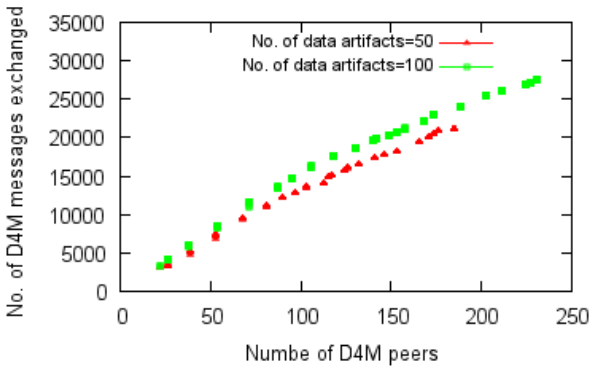
(d)



(e)



(f)



(g)

Figure 29: Communication cost

Figure 29 shows the relationship between the traffic of D^{4}M messages with respect to an increasing number of peers and time while the number of replicas is altered. In figure 29(a), the number of replicas is varied while D^{4}M traffic is shown on the y-axis and the number of D^{4}M peers on the x-axis. There is more traffic generated when each peer in the network maintains eight replicas, as compared to the least number of replicas. There is a gradual increase in traffic with the addition of D^{4}M data holding peers. The reason is, with more number of D^{4}M peers in the network, the data artifacts are distributed across the different peers in the network which need to be retrieved and updated. The number of update messages and lookup messages increases with the increase in number of D^{4}M peers. Similarly the same occurs with an increase in the number of updates as depicted in figure 29(c).

Figure 29(b) and (c) simulate the traffic generated in the system with respect to the time and compares the traffic for different numbers of replicas and updates. Initially, very heavy traffic occurs as shown in the plot due to the bootstrap process in which each peer starts its replica selection process and queries the available cache space of each neighbor which results in a rapid increase in traffic generation. After the cache replicas are established, the D^{4}M traffic generated is only to maintain the replicas and synchronize the data dependencies across the network. That is why the plot shows that after some time, the traffic is constant. Similarly in figure 29(d), the increase in number of updates with the time will affect the traffic generation.

In figure 29(e) and figure 29(f), we compare the overlay messages and D^{4}M messages to study the extra overhead introduced with D^{4}M replica maintenance. With the help of both the graphs displaying the number of messages exchanged and the number of peers, it can be concluded that the proposed replication protocol does not impose any major traffic overhead. Figure 29(g) shows the impact of varying the number of data artifacts on traffic generation. For more data artifacts, there will be more traffic generated due to the update propagation.

The synchronization and re-synchronization time for replicas during maintenance is same for an increasing number of peers, and is not affected by the variation in number of replicas or data artifacts.

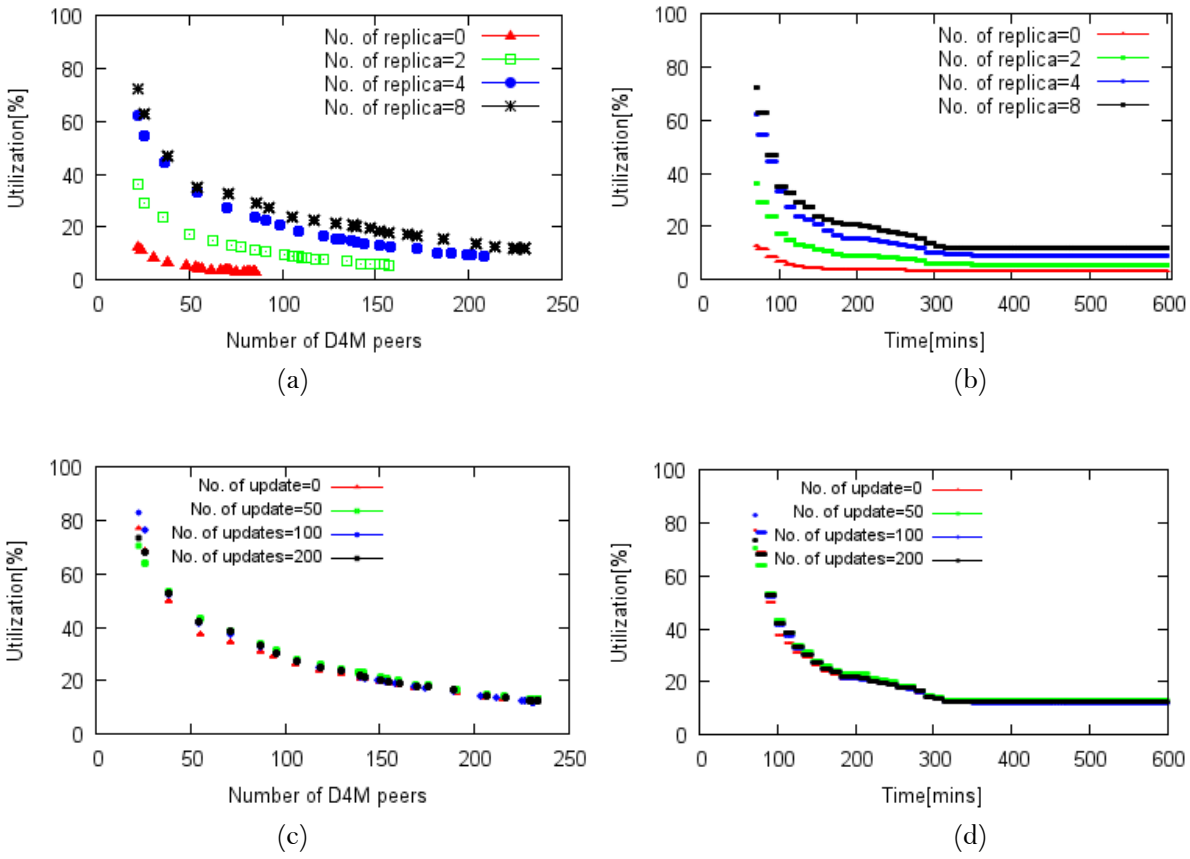
6.1.3 Fairness Evaluation

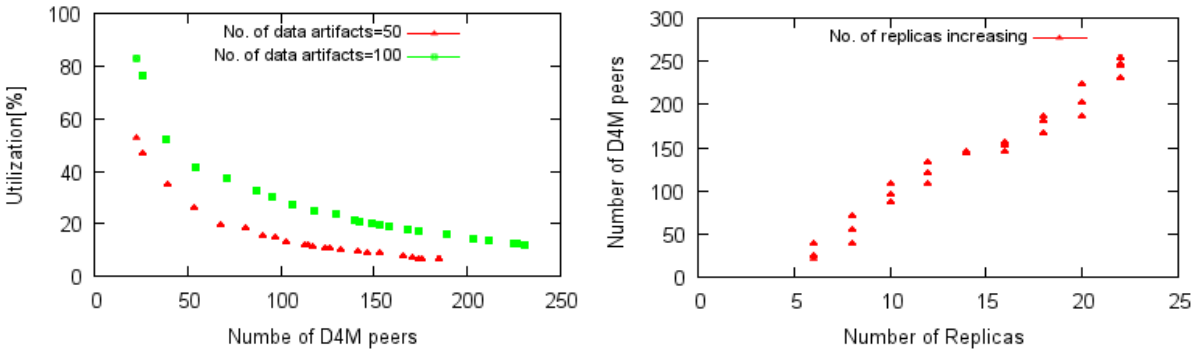
The percentage utilization of resources (such as memory) is used to study the fairness of the proposed replication protocol. This section discusses the details of this utilization analysis.

The plots shown in Figure 30, shows the utilization (as a percentage) of the resources used by the peers in the network with different numbers of replicas, updates, and data artifacts. Figure 30(a) shows that, with larger numbers of replicas, that the peers used more storage space; thus utilizing more

resources as compared to peers in the network with the least number of replicas. Similarly, figure 30(d) demonstrates that as soon as the peers join the system, the data artifacts are distributed among more peers in the network and free up the storage. But at the same time, the increasing number of replicas needs more storage space of peers to store the redundant cache data as depicted in figure 30(a) and (b). The change in the number of updates does not impact utilization due to the fact that updates do not take more space than the space before the update, as shown in figure 30(c). In the final graph, figure 30(e) shows that the number of artifacts needing storage space affects the overall utilization of system when new artifacts are introduced. Therefore, we conclude that the system resources will be minimum used as additional peers join the network. Initially when there are few peers in the network the utilization is quite high, but it gradually decreases with the addition of peers (as the total amount of state remains the same but is distributed over more peers).

Figure 30(f) indicates the relationship between the number of D^4M peers and the number of replicas. The number of D^4M peers will increase with the increase in the number of replicas. Based on this plot, we conclude that most of the peers would be D^4M peers if there is a heavy data loaded overlay network exists.





(e) (f)

Figure 30: Resource utilization

7 Conclusion

This section details the results and analysis, concluded in this thesis work. In this thesis, we presented a flexible and loosely coupled design to integrate the D^4M framework into PeerfactSim.KOM and a utility-based replication management protocol is used to achieve data availability with little overhead. The main goal was to tolerate churn in highly heterogeneous network, where high capacity devices and low capacity devices can participate in the replication protocol providing balanced use of resources in the network. To achieve this goal, the utility function plays an important role when selecting the replica holders based on their utility. The proposed replication approach uses a symmetric replication scheme. A symmetric replication scheme is a replication scheme well suited to structured overlay networks that can make a decision on replicas placement and detect when to replicate data.

The utility function leads us to different network designs. A p2p cluster of equal utility peers can be built. In our proposed protocol, the utility function selects the replica peers which have the best (maximum storage) utility. In order to build a cluster of equal utility peers, the utility function may select the replica peers which have a utility value comparable to the peer who initiates the replication. In this way, we can create different clusters of peers in network equal utility.

Although, we propose the use of our protocol to achieve data availability for the data dependency management system, we believe that this protocol can be used in general to replicate network monitoring services and other services in structured overlay networks.

8 FutureWork

Future work will include modification of the proposed replication protocol for concurrency handling, as redundancy always speed up data lookups across the network. The concurrency has to be handled in such a way that complex dependency relationships of data artifacts are maintained and are consistent across all the replicas. Uncoordinated concurrent updates of a data artifact may result in an unpredictable computed data value of dependent data artifacts. In addition to concurrency handling, we must investigate environments with high churn rates where all the replicas of a root peer may crash at the same time.

References

[1]. D. Stingl, C. Gross, J. Rückert, L. Nobach, A. Kovacevic, and R. Steinmetz. *PeerfactSim.KOM: A Simulation Framework for Peer-to-Peer Systems*. In Proc. of the IEEE International Conference on High Performance Computing & Simulation (IEEE HPCS '11), 2011

[2]. K. Saller, D. Stingl, and A. Schürr. (2011, March). *D^4M, a Self-Adapting Decentralized Derived Data Collection and Monitoring Framework.* Workshops of the conference scientific communication in Distributed Systems 2011, 37, 245 – 256, 2011

[3]. The Annotated Gnutella Protocol Specification v0.4, 2003: [www] http://rfc-gnutella.sourceforge.net/developer/stable/index.html#t1. Last access on 2012-05-17

[4]. D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R.Panigraphy, *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web*. In Proceedings of the 29th Annual ACM Symposium on Theory of Computing (El Paso, TX, May 1997), pp. 654–663.

[5]. A. Rowstron and P. Druschel, *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*. Lecture Notes in Computer Science, 2218, 2001.

[6]. M. Ripeanu, I. Foster, and A. Iamnitchi, *Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design.* IEEE Internet Computing,6(1), February 2002.

[7]. Y. Chawathe, S. Ratnasamy, L. Breslau, S. Shenker, and N. Lanham. *GIA: Making Gnutella-like P2P Systems, Scalable*. ACM SIGCOMM 2003.

[8]. J. Liang, R. Kumar, and K. Ross. *The FastTrack Overlay: A Measurement Study. Computer Networks*, 50, 2006, 842–858.

[9]. R. Morselli, B. Bhattacharjee, A. Srinivasan, and M. Marsh, *Efficient lookup on unstructured topologies.* Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing (Las Vegas, NV, USA, July 17 – 20, 2005). PODC '05. ACM Press, New York, NY, 77–86.

[10]. A. Löser, S. Staab, and C. Tempich, *Semantic Social Overlay Networks*. IEEE J. Sel. Areas. Communications, 25(1), 2007, 5–14.

[11]. J. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. Epema, M. Reinders, M. van Steen, and H. Sips. *Tribler: A Social-based Peer-to-Peer system*. Proc. of the 5thInternational Workshop on Peer-to-Peer Systems (IPTPS'06).

[12]. I. Clarke, O. Sandberg, B. Wiley, and T. Hong, *Freenet: A Distributed Anonymous Information Storage and Retrieval System*, in Lecture Notes in Computer Science : Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, USA, July 2000, Proceedings:, vol. 2009, Springer Berlin / Heidelberg, 2001, pp. 46–66.

[13]. I. Stoica, R. Morris, D. Liben-nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, 'Chord: a scalable peer-to-peer lookup protocol for internet applications', IEEE/ACM Transactions on Networking, vol. 11, pp. 17–32, 2003.

[14]. A. Rowstron and P. Druschel. *Storage management and caching in PAST, a large-scale,persistent peer-to-peer storage utility*. In Proc. ACM SOSP'01, Banff, Canada, Oct. 2001.

[15]. A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel, '*SCRIBE: The design of a large-scale event notification infrastructure*', in In Networked Group Communication, 2001, pp. 30–43.

[16]. P. Druschel and A. Rowstron. *PAST: A large-scale, persistent peer-to-peer storage utility*. In Proc. HotOS VIII, Schloss Elmau, Germany, May 2001.

[17]. D. Stingl, C. Gross, J. Rückert, L. Nobach, S. Kaune, and K. Pussep, *PeerfactSim.KOM: A Large-Scale Simulation Framework for Peer-to-Peer Systems*, [www] http://peerfact.com/pub/PeerfactSim.KOM-2011-Documentation.pdf, Last access on 2011-08-03

[18]. Distributed Hash Tables at Wikipedia [www] http://en.wikipedia.org/wiki/Distributed_hash_table. Last access on 2012-05-17

[19]. Definition of Scalability [www] http://www.cisco.com/univercd/cc/td/doc/product/dsl_prod/6160/hwguide/glossary.htm. Last access on 2012-05-17

[20]. Definition of peer-to-peer [www] http://en.wikipedia.org/wiki/Peer-to-peer. Last access on 2012-05-17

[21]. Napster [www] http://www.napster.com/. Last access on 2012-05-17

[22]. Online file storage providers [www] http://en.wikipedia.org/wiki/File_hosting_service. Last access on 2012-05-17

[23]. Network management and monitoring applications [www] www.im.ncnu.edu.tw/ycchen/nm/ch_13x.ppt. Last access on 2012-05-17

[24]. Definition of Availability [www] http://en.wikipedia.org/wiki/Availability. Last access on 2012-05-17

[25]. Client-Server architecture [www] http://en.wikipedia.org/wiki/Client-server_model. Last access on 2012-05-17

[26]. Gnutella, wikipedia [www] http://en.wikipedia.org/wiki/Gnutella. Last access on 2012-05-17

[27] P. Maymounkov and D. Mazières, '*Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*', in Revised Papers from the First International Workshop on Peer-to-Peer Systems, London, UK, UK, 2002, pp. 53–65.

[28]. Bamboo DHT [www] http://bamboo-dht.org/. Last access on 2012-05-17

[29]. FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce, National Technical Information Service (NIST), Springfield, VA, April 1995.

[30]. Systems Simulation: The Shortest Route to Applications: Modelling & Simulation. [www] http://home.ubalt.edu/ntsbarsh/simulation/sim.htm#rwis. Last access on 2012-05-17

[31]. R. Jain, the Art of Computer Systems Performance Analysis. John Wiley & Sons, Inc, 1991.

[32]. A. Montresor and M. Jelasity, '*PeerSim: A scalable P2P simulator*', 2009 IEEE Ninth International Conference on PeertoPeer Computing, no. 214412, pp. 99–100, 2009.

[33]. J. Pujol Ahullo and P. Garcia Lopez, "*PlanetSim: An Extensible Simulation Tool for Peer-to-Peer Networks and Services*" in Proc. of the 9th Int. Conf. on Peer-to-Peer Computing, 2009, pp. 85–86.

[34]. Kompics Framework, KTH, Sweden. [www] http://kompics.sics.se/trac. Last access on 2012-05-17

[35]. I. Baumgart, B. Heep, and S. Krause, "*OverSim: A Flexible Overlay Network Simulation Framework*" in IEEE Global Internet Symposium, 2007, pp. 79–84.

[36]. S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers, "*The State of Peer-to-Peer Simulators and Simulations*" SIGCOMM Computer Communication Review, vol. 37, 2007, pp. 95–98.

[37]. B. Kantor and P. Lapsley, '*Network News Transfer Protocol*', Internet Request for Comments, vol. RFC 977 (Proposed Standard), February 1986, Available at http://www.rfc-editor.org/rfc/rfc977.txt . Last access on 2012-05-17

[38]. A. Shaikh, R. Tewari, and M. Agrawal, '*On the Effectiveness of DNS-based Server Selection*', in In Proceedings of IEEE Infocom, 2001.

[39]. Lotus Software, IBM Software Group. Administering the Domino System Volume 1, May 1999.

[40]. M. R. Crispin. *Internet message access protocol - version 4* rev1. Internet Request for Comments. RFC 3501 (Proposed Standard), March 2003, Available at http://www.rfc-editor.org/rfc/rfc3501.txt . Last access on 2012-05-17

[41]. L. Fan, P. Cao, J. Almeida, and A. Broder, '*Summary cache: a scalable wide-area Web cache sharing protocol*', SIGCOMM Comput. Commun. Rev., vol. 28, no. 4, pp. 254–265, 1998.

[42]. H. Yu, L. Breslau, and S. Shenker, '*A Scalable Web Cache Consistency Architecture'*, 1999, pp. 163–174.

[43]. J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl, '*Globally Distributed Content Delivery*', IEEE Internet Computing, vol. 6, no. 5, pp. 50–58, Sep. 2002.

[44] G. Pierre and M. van Steen, '*Design and Implementation of a User-Centered Content Distribution Network*', in Proceedings of the The Third IEEE Workshop on Internet Applications, Washington, DC, USA, 2003, p. 42–.

[45]. F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "*Wide-area cooperative storage with CFS*" in Proc. of the 18th ACM Symp. on Operating Systems Principles, Banff, Canada, Oct. 2001.

[46]. A. Rowstron and P. Druschel, "*Storage management and caching in PAST, a large scale, persistent peer-to-peer storage utility*" in Proc. of the 18th ACM Symposium on Operating Systems Principles, Banff, Canada, Oct. 2001.

[47]. J. Kubiatowicz, D. Bindel, Y. Chen, S. Cwerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "*Oceanstore: an architecture for global scale persistent storage*", ASPLOS,pp. 190 -201, November 2000.

[48]. R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. *Total recall: System support for automated availability management*. In Proceedings of NSDI, 2004.

[49]. H. Weatherspoon and J. Kubiatowicz. *Erasure coding vs. replication: A quantitative comparison*. In Proceedings of IPTPS, 2002.

[50]. F. Dabek, J. Li, E. Sit, J. Robertson, F.F. Kaashoek,.and R. Morris: *Designing a DHT for low latency and high throughput*. In: NSDI 2004: Proceedings of the 1st Symposium on Networked Systems Design and Implementation, San Francisco, CA, USA (March 2004)

[51]. S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Schenker: *A scalable content-addressable network*. In: SIGCOMM, vol. 31, pp. 161–172. ACM Press, New York (2001)

[52]. B.Y. Zhao, L. Huang, , J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz: *Tapestry: A global-scale overlay for rapid service deployment*. IEEE Journal on Selected Areas in Communications (2003)

[53]. S., Ktari, M. Zoubert, A.Hecker, and H.Labiod: *Performance evaluation of replication strategies in DHTs under churn*. In: MUM 2007: Proceedings of the 6th international conference on Mobile and ubiquitous multimedia, pp. 90–97. ACM Press, New York (2007)

[54]. A. Ghodsi, L.O. Alima, and S. Haridi: *Symmetric replication for structured peer-to-peer systems*. In: G. Moro, S. Bergamaschi, S. Joseph, J.-H. Morin, and A.M. Ouksel, (eds.) DBISP2P 2005 and DBISP2P 2006. LNCS, vol. 4125, pp. 74–85. Springer, Heidelberg (2007)

[55]. M. Waldvogel, P. Hurley, and D. Bauer, '*Dynamic Replica Management in Distributed Hash Tables*', in Research Report RZ–3502, IBM, 2003.

[56]. P. Knežević, A. Wombacher, and T. Risse, *"Advanced Internet Based Systems and Applications"* ed. Ernesto Damiani et al. (Berlin, Heidelberg: Springer-Verlag, 2009), 201–210, http://dx.doi.org/10.1007/978-3-642-01350-8_19. Last access on 2012-05-17

[57]. S. Legtchenko, S. Monnet, P. Sens, and G. Muller, '*Churn-Resilient Replication Strategy for Peer-to-Peer Distributed Hash-Tables*', in Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems, Berlin, Heidelberg, 2009, pp. 485–499.

[58]. P. Yalagandula and M. Dahlin, '*A scalable distributed information management system*', in Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications, New York, NY, USA, 2004, pp. 379–390.

[59]. R. V. Renesse, K. P. Birman, and W. Vogels, '*Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining*', ACM Transactions on Computer Systems, vol. 21, p. 2003, 2001.

[60]. P. Mukherjee, C. Leng, A. Schürr. *Piki - A Peer-to-Peer based Wiki Engine*. Eighth International Conference on Peer-to-Peer Computing, 2008.

[61]. B. Schandl and S. Zander, "Autonomous RDF Replication on Mobile Devices (Poster and Demo)" (October 2009), http://eprints.cs.univie.ac.at/189/ . Last access on 2012-05-17

TRITA-ICT-EX-2012:60

www.kth.se